Deltek

Deltek Maconomy®

Language Reference Guide

September 24, 2021

While Deltek has attempted to verify that the information in this document is accurate and complete, some typographical or technical errors may exist. The recipient of this document is solely responsible for all decisions relating to or use of the information provided herein.

The information contained in this publication is effective as of the publication date below and is subject to change without notice.

This edition published September 2021.

Contents

General Functions

Introduction

This section documents the set of primitive standard functions that are available in the Expression Language that is used in MDML.

MDML and the Expression Language

A function call in the Expression Language has the following form:

```
maconomy:userEmployeeNumber()
```

> **Note: maconomy** is the namespace of the function and “userEmployeeNumber” is the function name.

Case is not significant, “currentdate()” means the same thing as “CURRENTDATE().”

> **Note:** The **maconomy** namespace is the default namespace and can be omitted from function calls.

A function may take arguments. Arguments are supplied comma-separated within the parenthesis:

```
hasRole('ProjectManager')
```

> **Tip:** For a full reference of the expression language and its use in MDML, see the MDML Language Reference.

Format

Functions in this document are written in the following format:

`maconomy:functionName( mandatoryParameter, [optionalParameter] )`

Description of the function

Parameters

mandatoryParameter – Description of accepted parameter values

optionalParameter – Description of accepted parameter values

Returns

Description of return value including data type.

> **Note:** Optional parameters are presented in square brackets.

Functions

Environment Variables

The **envVar** family of functions is used to access the environment directly and fetch values from it using the full environment path.

> **Note:** Most standard functions are shortcuts to frequently used individual values or parts of the environment. The **envVar** family of function provides access to the entire environment.

Each function takes one parameter that designates the path to fetch from the environment. The environment may contain more than one value at a given path. In this case an appropriate type-specific **envVar** function must be used to fetch the value.

Sample use of an envVar function:

```
envVar('user.info.username') = 'Administrator'
```

In this example the path user.info.username is fetched and compared to the string 'Administrator'. Assuming that the value that is contained in the path user.info.username is a string data value, the following expression is equivalent:

```
stringEnvVar('user.info.username') = 'Administrator'
```

The string specific **envVar** function fails if the value that is contained at the specified path in the environment does not have the expected data type.

envVar

`maconomy:envVar( path )`

Fetch a value from the environment.

Parameters

path – String value that designates the full path in the environment.

Returns

If a single value is located at the specified path and that value has a maconomy data type, this function returns that value. Otherwise, the function fails.

isEnvVarDefined

`maconomy:isEnvVarDefined( path )`

Determine whether a value exists in the environment for a given path.

Parameters

path – String value that designates the full path in the environment.

Returns

True if the environment contains a value at the path. Otherwise, False.

amountEnvVar

maconomy:amountEnvVar(path)

Fetch an amount value from the environment.

Parameters

path – String value that designates the full path in the environment.

Returns

If a single amount value is located at the specified path, this function returns that value. Otherwise, the function fails.

booleanEnvVar

maconomy:booleanEnvVar(path)

Fetch a boolean value from the environment.

Parameters

path – String value that designates the full path in the environment

Returns

If a single boolean value is located at the specified path, this function returns that value. Otherwise, the function fails.

dateEnvVar

maconomy:dateEnvVar(path)

Fetch a date value from the environment.

Parameters

path – String value that designates the full path in the environment.

Returns

If a single date value is located at the specified path, this function returns that value. Otherwise, the function fails.

decimalEnvVar

maconomy:decimalEnvVar(path)

Fetch a decimal value from the environment.

Parameters

path – String value that designates the full path in the environment.

Returns

If a single decimal value is located at the specified path, this function returns that value. Otherwise, the function fails.

integerEnvVar

maconomy:integerEnvVar(path)

Fetch an integer value from the environment.

Parameters

path – String value that designates the full path in the environment.

Returns

If a single integer value is located at the specified path, this function returns that value. Otherwise, the function fails.

popupEnvVar

maconomy:popupEnvVar(path)

Fetch a popup value from the environment.

Parameters

path – String value that designates the full path in the environment.

Returns

If a single popup value is located at the specified path, this function returns that value. Otherwise, the function fails.

stringEnvVar

maconomy:stringEnvVar(path)

Fetch a string value from the environment.

Parameters

path – String value that designates the full path in the environment.

Returns

If a single string value is located at the specified path, this function returns that value. Otherwise, the function fails.

timeEnvVar

maconomy:timeEnvVar(path)

Fetch a time value from the environment.

Parameters

path – String value that designates the full path in the environment.

Returns

If a single time value is located at the specified path, this function returns that value. Otherwise, the function fails.

System Parameters

The **sysPar** family of functions contains helper functions to fetch the value of system parameters.

Each function takes one mandatory parameter that designates the name of the system parameter to fetch, and one optional parameter that designates a company number.

If a **sysPar** function is called with a company number argument and no company-specific system parameter is defined, the general value for the system parameter is returned.

> **Note:** Some system parameters have a combined decimal/amount format. To fetch the value of such a parameter, one of the type specific variants of the sysPar function must be used.

sysPar

```
maconomy:sysPar( parameterName, [companyNumber] )
```

Fetch the value of a named system parameter.

Parameters

parameterName – String value that designates the name of the parameter to fetch

companyNumber – String value, the company number of the company to fetch the parameter for.

Returns

If the system parameter exists and contains a single value, this function returns that value. Otherwise, the function fails.

amountSysPar

```
maconomy:amountSysPar( parameterName, [companyNumber] )
```

Fetch the amount value of a named system parameter.

Parameters

parameterName – String value that designates the name of the parameter to fetch.

companyNumber – String value, the company number of the company to fetch the parameter for.

Returns

If the system parameter exists and contains an amount value, this function returns that value. Otherwise, the function fails.

booleanSysPar

maconomy:booleanSysPar(parameterName, [companyNumber])

Fetch the boolean value of a named system parameter.

Parameters

parameterName – String value that designates the name of the parameter to fetch.

companyNumber – String value, the company number of the company to fetch the parameter for.

Returns

If the system parameter exists and contains a boolean value, this function returns that value. Otherwise, the function fails.

dateSysPar

maconomy:dateSysPar(parameterName, [companyNumber])

Fetch the date value of a named system parameter.

Parameters

parameterName – String value that designates the name of the parameter to fetch.

companyNumber – String value, the company number of the company to fetch the parameter for.

Returns

If the system parameter exists and contains a date value this function returns that value. Otherwise, the function fails.

integerSysPar

maconomy:integerSysPar(parameterName, [companyNumber])

Fetch the integer value of a named system parameter.

Parameters

parameterName – String value that designates the name of the parameter to fetch.

companyNumber – String value, the company number of the company to fetch the parameter for.

Returns

If the system parameter exists and contains an integer value, this function returns that value. Otherwise, the function fails.

popupSysPar

maconomy:popupSysPar(parameterName, [companyNumber])

Fetch the popup value of a named system parameter.

Parameters

parameterName – String value that designates the name of the parameter to fetch.

companyNumber – String value, the company number of the company to fetch the parameter for.

Returns

If the system parameter exists and contains a popup value, this function returns that value. Otherwise, the function fails.

stringSysPar

maconomy:stringSysPar(parameterName, [companyNumber])

Fetch the string value of a named system parameter.

Parameters

parameterName – String value that designates the name of the parameter to fetch.

companyNumber – String value, the company number of the company to fetch the parameter for.

Returns

If the system parameter exists and contains a string value, this function returns that value. Otherwise, the function fails.

timeSysPar

maconomy:timeSysPar(parameterName, [companyNumber])

Fetch the time value of a named system parameter.

Parameters

parameterName – String value that designates the name of the parameter to fetch.

companyNumber – String value, the company number of the company to fetch the parameter for.

Returns

If the system parameter exists and contains a time value, this function returns that value. Otherwise, the function fails.

System Information

The **sysInfo** function family fetches system information values. Each function takes one parameter that designates the name of the system information value to fetch.

sysInfo

`maconomy:sysInfo( name )`

Fetch a named system information value.

Parameters

name – String value that designates the name of system information value to fetch.

Returns

If the system information value exists this function returns that value. Otherwise, the function fails.

amountSysInfo

`maconomy:amountSysInfo( name )`

Fetch a named system information amount value.

Parameters

name – String value that designates the name of system information value to fetch.

Returns

If the system information value exists and is of type amount, this function returns that value. Otherwise, the function fails.

booleanSysInfo

`maconomy:booleanSysInfo( name )`

Fetch a named system information boolean value.

Parameters

name – String value that designates the name of system information value to fetch.

Returns

If the system information value exists and is of type boolean, this function returns that value. Otherwise, the function fails.

dateSysInfo

`maconomy:dateSysInfo( name )`

Fetch a named system information date value.

Parameters

name – String value that designates the name of system information value to fetch.

Returns

If the system information value exists and is of type date, this function returns that value. Otherwise, the function fails.

decimalSysInfo

maconomy:decimalSysInfo(name)

Fetch a named system information decimal value.

Parameters

name – String value that designates the name of system information value to fetch.

Returns

If the system information value exists and is of type decimal, this function returns that value. Otherwise, the function fails.

integerSysInfo

maconomy:integerSysInfo(name)

Fetch a named system information integer value.

Parameters

name – String value that designates the name of system information value to fetch.

Returns

If the system information value exists and is of type integer, this function returns that value. Otherwise, the function fails.

popupSysInfo

maconomy:popupSysInfo(name)

Fetch a named system information popup value.

Parameters

name – String value that designates the name of system information value to fetch.

Returns

If the system information value exists and is of type popup, this function returns that value. Otherwise, the function fails.

stringSysInfo

maconomy:stringSysInfo(name)

Fetch a named system information string value.

Parameters

name – String value that designates the name of system information value to fetch.

Returns

If the system information value exists and is of type string, this function returns that value. Otherwise, the function fails.

timeSysInfo

maconomy:timeSysInfo(name)

Fetch a named system information time value.

Parameters

name – String value that designates the name of system information value to fetch.

Returns

If the system information value exists and is of type time this function returns that value. Otherwise, the function fails.

Misc System Functions

In addition to accessing system parameters and system information, there are functions for checking the presence of add-ons and short name of the current system.

hasAddOn

`maconomy:hasAddOn( addOnNumber )`

Determine if a numbered addon is present in the system.

Parameters

addOnNumber – An integer value that identifies the addon which presence is to be determined.

Returns

This function returns boolean data value indicating the presence of the addon.

shortname

`maconomy:shortname()`

Find the shortname of the system.

Returns

This function returns a string value indicating the shortname of the system.

getToken

`kona:getToken()`

Fetch the user's Kona token. The server retrieves this token from www.kona.com after a successful Kona login. The token is used to preserve the user's Kona session during one session with the Workspace client.

Returns

This function returns a string value indicating the token or an empty string if the user is not logged in to Kona..

Date and Time Functions

The family of date and time functions includes functions for querying the current date and time and extracting day, month, year, and others from date values.

date

maconomy:date(year, month, day)

Construct a date value from a year, month and a day. If the arguments are not a valid date this function fails.

Parameters

year – An integer that specifies the year of the resulting date value.

month – An integer that specifies the month of the resulting date value (1 is January, 2 February, etc.).

day – An integer that specifies the day of the resulting date value (1 is the 1st, 2 is the 2nd, etc.).

Returns

This function returns a date value with the specified values.

time

maconomy:time(hours, minutes, [seconds])

Construct a time value from hours, minutes and seconds. If the arguments are not a valid time this function fails.

Parameters

hours – An integer that specifies the hours of the time value (valid range is 0 through 23).

minutes – An integer that specifies the minutes of the time value (valid range is 0 through 59).

seconds – An optional integer that specifies the seconds of the time value (valid range is 0 through 59).

Returns

This function returns a time value with the specified values.

currentDate

maconomy:currentDate()

Find the current date and convert it into the time-zone of the server. This function can be used to get an approximate of the current server date.

Returns

This function returns a date value, the current date converted to the time-zone of the server.

currentTime

maconomy:currentTime()

Find the current time and convert it into the time-zone of the server. This function can be used to get an approximate of the current server time.

Returns

This function returns a time value, the current time converted to the time-zone of the server.

userDate

maconomy:userDate()

Find the current date *without* converting it into the time-zone of the server. When run on the client, this function returns the date of the client machine.

Returns

This function returns a date value, the current date.

userTime

maconomy:userTime()

Find the current time *without* converting it into the time-zone of the server. When run on the client, this function returns the time of the client machine.

Returns

This function returns a time value, the current time.

second

maconomy:second(time)

Find the second a time value. E.g. second(time(14:32:46)) evaluates to 46.

Parameters

time – The time value to convert.

Returns

This function returns an integer value, the second of the time value.

minute

maconomy:minute(time)

Find the minute a time value. E.g. minute(time(14:32:46)) evaluates to 32.

Parameters

time – The time value to convert.

Returns

This function returns an integer value, the minute of the time value.

hour

maconomy:hour(time)

Find the hour a time value. E.g. hour (time(14:32:46)) evaluates to 14.

Parameters
time – The time value to convert.

Returns
This function returns an integer value, the hour of the time value.

day

maconomy:day(date)

Find the day of month of a date value. E.g. day(date(2010/03/15)) evaluates to 15.

Parameters
date – The date value to convert.

Returns
This function returns an integer value, the day of month of the date value.

month

maconomy:month(date)

Find the month of a date value. E.g. month(date(2010/03/15)) evaluates to 3.

Parameters
date – The date value to convert.

Returns
This function returns an integer value, the month of the date value.

year

maconomy:year(date)

Find the year of a date value. E.g. day(date(2010/03/15)) evaluates to 2010.

Parameters
date – The date value to convert.

Returns
This function returns an integer value, the year of the date value.

week

maconomy:week(date)

Find the week number of a date value. Week numbers are computed using the ISO 8601 week numbering scheme with Monday as the first day of the week.
E.g. day(date(2010/03/15)) evaluates to 11.

Parameters
date – The date value to convert.

Returns
This function returns an integer value, the week number of the date value.

intWeekday

maconomy:intWeekday(date)

Find the week day of a date value. The week starts with Monday (1) and ends on Sunday (7).

Parameters
date – The date value to convert.

Returns
This function returns an integer value, the weekday of the date value.

stringWeekday

maconomy:stringWeekday(date)

Find the week day of a date value. The week day is returned as a localized string value.

Parameters
date – The date value to convert.

Returns
This function returns a string value, the name of the weekday of the date value.

addDays

maconomy:addDays(date, days)

Construct a date value by adding a number of days to a date.

Parameters
date – A date value.

days – An integer that specifies a number of days to add to the given date (may be negative).

Returns
This function returns a date value that is the argument date plus the number of days.

addMonths

maconomy:addMonths(date, months)

Construct a date value by adding a number of months to a date.

Parameters

date – A date value.

months – An integer that specifies a number of months to add to the given date (may be negative).

Returns

This function returns a date value that is the argument date plus the number of months.

addYears

maconomy:addYears(date, years)

Construct a date value by adding a number of years to a date.

Parameters

date – A date value.

years – An integer that specifies a number of years to add to the given date (may be negative).

Returns

This function returns a date value that is the argument date plus the number of years.

addPeriod

maconomy:addPeriod(date, years, months, days)

Construct a date value by adding a number of years, months and days to a date.

Parameters

date – A date value.

years – An integer that specifies a number of years to add to the given date (may be negative).

months – An integer that specifies a number of months to add to the given date (may be negative).

days – An integer that specifies a number of days to add to the given date (may be negative).

Returns

This function returns a date value that is the argument date plus the number of years, months and days.

addHours

maconomy:addHours(time, hours)

Construct a date time by adding a number of hours to a time.

Parameters

time – A time value.

hours – An integer that specifies a number of hours to add to the given time (may be negative).

Returns

This function returns a time value that is the argument time plus the number of hours.

addMinutes

maconomy:addMinutes(time, hours)

Construct a date time by adding a number of minutes to a time.

Parameters

time – A time value.

minutes – An integer that specifies a number of minutes to add to the given time (may be negative).

Returns

This function returns a time value that is the argument time plus the number of minutes.

addSeconds

maconomy:addSeconds(time, hours)

Construct a date time by adding a number of seconds to a time.

Parameters

time – A time value.

seconds – An integer that specifies a number of seconds to add to the given time (may be negative).

Returns

This function returns a time value that is the argument time plus the number of seconds.

daysBetween

maconomy:daysBetween(date1, date2)

Find the number of days in the interval between two dates.

Parameters

date1 – The start date of the interval.

date2 – The end date of the interval.

Returns

This function returns an integer value, the number of days between the two dates.

daysBetween

maconomy:daysBetween(date1, date2)

Find the number of days in the interval between two dates.

Parameters

date1 – The start date of the interval.

date2 – The end date of the interval.

Returns

This function returns an integer value, the number of days between the two dates.

monthsBetween

maconomy:monthsBetween(date1, date2)

Find the number of months in the interval between two dates.

Parameters

date1 – The start date of the interval.

date2 – The end date of the interval.

Returns

This function returns an integer value, the number of months between the two dates.

yearsBetween

maconomy:yearsBetween(date1, date2)

Find the number of years in the interval between two dates.

Parameters

date1 – The start date of the interval.

date2 – The end date of the interval.

Returns

This function returns an integer value, the number of years between the two dates.

secondsBetween

maconomy:secondsBetween(time1, time2)

Find the number of seconds in the interval between two times.

Parameters

time1 – The start time of the interval.

time2 – The end time of the interval.

Returns

This function returns an integer value, the number of seconds between the two times.

minutesBetween

```
maconomy:minutesBetween( time1, time2 )
```

Find the number of minutes in the interval between two times.

Parameters

time1 – The start time of the interval.

time2 – The end time of the interval.

Returns

This function returns an integer value, the number of minutes between the two times.

hoursBetween

```
maconomy:hoursBetween( time1, time2 )
```

Find the number of hours in the interval between two times.

Parameters

time1 – The start time of the interval.

time2 – The end time of the interval.

Returns

This function returns an integer value, the number of hours between the two times.

Utility Functions

A number of utility functions can be used to modify the output of other functions.

format

maconomy:format(value, format)

Format a value to a string of the given format. Depending on the type of the value different format string are allowed.

Integer, Real, Amount are formatted using a DecimalFormatter. For more information, see http://docs.oracle.com/javase/6/docs/api/java/text/DecimalFormat.html

Time and Dates are formatted using a DateFormatter. For more information, see http://docs.oracle.com/javase/6/docs/api/java/text/SimpleDateFormat.html

String are not formatted but returned as is.

PopupValue returns a string representation of the literal.

Booleans are formatted as “true” and “false.”

Returns
This function converts the value to a string by using the given format. The function returns a string.

urlEncode

maconomy:urlEncode(value)

Creates a canonical string representation of a value of any type and then the URL encodes this representation. The URL encoding is done using the URLEncoder. For more information, see http://download.oracle.com/javase/6/docs/api/

The encoding scheme is always UTF-8.

Returns
This function converts the value to a URL encoded string. The function returns a string.

Pop-Up Functions

popupTitle

maconomy:popupTitle(popupValue)

Extract the display title of a pop-up value.

Parameters

popupValue – The pop-up value to convert.

Returns

This function returns a string value containing the display title of the pop-up value.

popupLiteral

maconomy:popupLiteral(popupValue)

Extract the literal value of a pop-up value.

Parameters

popupValue – The pop-up value to convert.

Returns

This function returns a string value containing the literal value of the pop-up value.

popupOrdinal

maconomy:popupOrdinal(popupValue)

Extract the ordinal value of a pop-up value.

Parameters

popupValue – The pop-up value to convert.

Returns

This function returns the ordinal of the pop-up value as an integer value.

popup

maconomy:popup(type, literal, ordinal)

Construct a pop-up value.

Parameters

type – The type of the pop-up value.

literal – The literal of the pop-up value.

Ordinal – The ordinal of the pop-up value.

Returns

This function returns a pop-up value with the specified type, literal, and ordinal.

User Information

There are a few shortcut functions for accessing common user information, such as username and roles, despite the fact user information is stored in the environment under the path “user.info.*”

username

`maconomy:username()`

Find the username of the user.

Returns

This function returns a string value indicating the username of the user.

userEmployeeNumber

`maconomy:userEmployeeNumber()`

Find the employee number of the user.

Returns

This function returns a string value indicating the employee number of the user. If the user does not have an associated employee number, the empty string is returned.

isAdministrator

`maconomy:isAdministrator()`

Determine if the user is an administrator.

Returns

This function will return a boolean value indicating if the user is an administrator.

hasRole

`maconomy:hasRole( role )`

Determine if the user has a named role.

Parameter

role – Either (a) a string value indicated the name of the role, or (b) a value of the popup type GroupNameType.

Returns

This function returns a boolean value indicating whether the user has the named role.

User Derived Dimensions

The derived dimensions functions fetch the derived dimensions values of the user.

> **Note:** The functions first attempt to fetch the information stored on the employee associated with the user. If there is no employee associated with the user, then the value are acquired from the user information.

Each of the derived dimensions functions takes an optional date argument. It returns the value that applies at that date. If no date argument is supplied, the value that applies to the current date is returned.

companyNumber

```
maconomy:companyNumber( [date] )
```

Fetch the value of the derived dimension 'company number' for the user.

Parameter

date – An optional date value. If a value is supplied, the value valid at the specified date is returned. Otherwise, the value valid at the current date is returned.

Returns

This function returns a string value indicating the value of the derived dimension for the user.

accountNumber

```
maconomy:accountNumber( [date] )
```

Fetch the value of the derived dimension 'account number' for the user.

Parameter

date – An optional date value. If a value is supplied, the value valid at the specified date is returned. Otherwise, the value valid at the current date is returned.

Returns

This function returns a string value indicating the value of the derived dimension for the user.

locationName

```
maconomy:locationName( [date] )
```

Fetch the value of the derived dimension 'location name' for the user.

Parameter

date – An optional date value. If a value is supplied, then the value valid at the specified date is returned. Otherwise, the value valid at the current date is returned.

Returns

This function returns a string value indicating the value of the derived dimension for the user.

entityName

maconomy:entityName([date])

Fetch the value of the derived dimension 'entity name' for the user.

Parameter

date – An optional date value. If a value is supplied, the value valid at the specified date is returned. Otherwise, the value valid at the current date is returned.

Returns

This function returns a string value indicating the value of the derived dimension for the user.

projectName

maconomy:projectName([date])

Fetch the value of the derived dimension 'project name' for the user.

Parameter

date – An optional date value. If a value is supplied, the value valid at the specified date is returned. Otherwise, the value valid at the current date is returned.

Returns

This function returns a string value indicating the value of the derived dimension for the user.

purposeName

maconomy:purposeName([date])

Fetch the value of the derived dimension 'purpose name' for the user.

Parameter

date – An optional date value. If a value is supplied, the value valid at the specified date is returned. Otherwise, the value valid at the current date is returned.

Returns

This function returns a string value indicating the value of the derived dimension for the user.

specification1Name

maconomy:specification1Name([date])

Fetch the value of the derived dimension 'specification 1 name' for the user.

Parameter

date – An optional date value. If a value is supplied, the value valid at the specified date is returned. Otherwise, the value valid at the current date is returned.

Returns

This function returns a string value indicating the value of the derived dimension for the user.

specification2Name

maconomy:specification2Name([date])

Fetch the value of the derived dimension 'specification 2 name' for the user.

Parameter

date – An optional date value. If a value is supplied, the value valid at the specified date is returned. Otherwise, the value valid at the current date is returned.

Returns

This function returns a string value indicating the value of the derived dimension for the user.

specification3Name

maconomy:specification3Name([date])

Fetch the value of the derived dimension 'specification 3 name' for the user.

Parameter

date – An optional date value. If a value is supplied, the value valid at the specified date is returned. Otherwise, the value valid at the current date is returned.

Returns

This function returns a string value indicating the value of the derived dimension for the user.

localSpec1Name

maconomy:localSpec1Name([date])

Fetch the value of the derived dimension 'local spec 1 name' for the user.

Parameter

date – An optional date value. If a value is supplied, the value valid at the specified date is returned. Otherwise, the value valid at the current date is returned.

Returns

This function returns a string value indicating the value of the derived dimension for the user.

localSpec2Name

maconomy:localSpec2Name([date])

Fetch the value of the derived dimension 'local spec 2 name' for the user.

Parameter

date – An optional date value. If a value is supplied, the value valid at the specified date is returned. Otherwise, the value valid at the current date is returned.

Returns

This function returns a string value indicating the value of the derived dimension for the user.

localSpec3Name

```
maconomy:localSpec3Name( [date] )
```

Fetch the value of the derived dimension 'local spec 3 name' for the user.

Parameter

date – An optional date value. If a value is supplied, the value valid at the specified date is returned. Otherwise, the value valid at the current date is returned.

Returns

This function returns a string value indicating the value of the derived dimension for the user.

MDML-Specific Functions

MDML offers a collection of functions that are only available in the context of MDML.

inEmpty

maconomy:inEmpty()

Determine if a pane is in an empty state.

Returns

This function returns a boolean value indicating whether the pane is in an empty state.

inCreate

maconomy:inCreate()

Determine if a pane is in a 'create' state.

Returns

This function returns a boolean value indicating whether the pane is in a create state.

inUpdate

maconomy:inUpdate()

Determine if a pane is in an 'update' state.

Returns

This function returns a boolean value indicating whether the pane is in an update state.

hasWorkspace

maconomy:hasWorkspace(workspaceName)

Determine if a named workspace is an ancestor of the current pane.

Parameter

workspaceName – The name of the workspace whose presence in the tree context is to be determined.

Returns

This function returns a boolean value indicating whether the workspace is an ancestor of the pane.

workspace

maconomy:workspace()

Fetch the name of the main workspace in the current context.

Returns

This function returns a string value containing the name of the workspace in the current context.

container

maconomy:container()

Fetch the name of the container in the current context.

Returns

This function returns a string value containing the name of the container in the current context.

pane

maconomy:pane()

Fetch the name of the pane in the current context.

Returns

This function returns a string value containing the name of the pane in the current context.

entity

maconomy:entity()

Fetch the name of the entity in the current context.

Returns

This function returns a string value containing the name of the entity in the current context.

field

maconomy:field()

Fetch the name of the field in the current context.

Returns

This function returns a string value containing the name of the field in the current context. If there is no field in the current context, this function fails.

fieldValue

maconomy:fieldValue()

Fetch the value of the field in the current context.

Returns

This function returns a value containing the value of the field in the current context. If there is no field in the current context, this function fails.

isSelectedOption

maconomy:isSelectedOption([optionName])

Determine if a given filter option is selected. This only makes sense in filter panes; in other panes this function will always return false.

Parameters

optionName – The name of the option. This is a string value that matches the name attribute on the option element in the MDML specification.

Returns

This function returns a boolean value. True, if the specified option is the selected filter option. Otherwise, false.

selectedOption

maconomy:selectedOption()

Fetch the name of the currently selected option. This only makes sense in filter panes. In other panes this function will always return the empty string.

Returns

This function returns a string value containing the name of the currently selected filter option.

visibleFields

maconomy:visibleFields()

Fetch the list of the currently visible fields in the current pane. The list is sorted in layout order.

Returns

This function returns a list (number indexed data map) of the visible fields in the current pane.

visibleFieldTitles

maconomy:visibleFieldTitles()

Fetch the field titles of the currently visible fields in the current pane. The returned value is a map from field name to field title.

Returns

This function returns a data map from field name to the field titles for all visible fields in the current pane.

isActionEnabled

maconomy:isActionEnabled(actionName)

Test if an action is enabled.

Parameters

actionName – The name of the tested action. This is the name specified by the application and not the name specified by the name-attribute in MDML.

Returns

This function returns true if the action with the name actionName is enabled.

String Functions

String functions can be used to manipulate strings and perform validations.

length

maconomy:length(string)

Find the length of the input string.

Parameters

string – The string value to find the length of.

Returns

This function returns an integer of the number of characters in the argument string.

startsWith

maconomy:startsWith(haystack, needle, [fromIndex])

Test if a string begins with a given other string.

Parameters

haystack – The string to check.

needle – The string to check if haystack begins with.

fromIndex – Optional integer. The index in haystack to start looking for needle. Defaults to zero.

Returns

This function returns true, if haystack starts with needle at the index. Otherwise, false.

endsWith

maconomy:endsWith(haystack, needle)

Test if a string ends with a given other string.

Parameters

haystack – The string to check.

needle – The string to check if haystack ends with.

Returns

This function returns true, if haystack ends with needle at the index. Otherwise, false.

firstIndexOf

maconomy:firstIndexOf(haystack, needle)

Find the first occurrence of needle in haystack.

Parameters

haystack – The string to search in.

needle – The string to look for in haystack.

Returns

This function returns the index of the first occurrence of needle in haystack, or -1 if needle does not occur in haystack.

indexOf

maconomy:indexOf(haystack, needle, [fromIndex])

Find the first occurrence of needle in haystack after a given index.

Parameters

haystack – The string to search in.

needle – The string to look for in haystack.

fromIndex – Optional integer. The index in haystack to start looking for needle. Defaults to zero.

Returns

This function returns the index of the first occurrence of needle in haystack, or -1 if needle does not occur in haystack after the specified index.

lastIndexOf

maconomy:lastIndexOf(haystack, needle)

Find the last occurrence of needle in haystack.

Parameters

haystack – The string to search in.

needle – The string to look for in haystack.

Returns

This function returns the index of the last occurrence of needle in haystack, or -1 if needle does not occur in haystack.

contains

maconomy:contains(haystack, needle)

Test of needle occurs in haystack.

Parameters

haystack – The string to search in.

needle – The string to look for in haystack.

Returns

This function returns true if needle occurs in haystack. Otherwise, false.

substring

maconomy:substring(string, fromIndex, [toIndex])

Take a substring of a string.

Parameters

string – The string to take a substring of.

fromIndex – Integer value that specified the index where the substring starts.

toIndex – Optional integer. The index where the substring ends. If no index is specified, the substring continues to the end of the argument string.

Returns

This function returns a string value that is the specified substring.

trim

maconomy:trim(string)

Trim whitespace from the ends of a string.

Parameters

string – The string to trim.

Returns

This function returns a string value that is argument string trimmed for any leading and trailing whitespace.

upperCase

maconomy:upperCase(string)

Uppercase a string using US locale rules.

Parameters

string – The string to uppercase.

Returns

This function returns the argument string converted to uppercase.

lowerCase

maconomy:lowerCase(string)

Lowercase a string using US locale rules.

Parameters

string – The string to lowercase.

Returns

This function returns the argument string converted to lowercase.

replaceFirst

maconomy:replaceFirst(haystack, needle, replacement)

Look for needle in haystack and replace the first occurrence with replacement.

Parameters

haystack – The string to search in.

needle – The string to look for in haystack.

replacement – The string to replace the first occurrence of needle with.

Returns

This function returns a string that is haystack with the first occurrence of needle replaced with replacement.

replaceAll

maconomy:replaceAll(haystack, needle, replacement)

Look for needle in haystack and replace all occurrences with replacement.

Parameters

haystack – The string to search in.

needle – The string to look for in haystack.

replacement – The string to replace all occurrences of needle with.

Returns

This function returns a string that is haystack with all occurrences of needle replaced with replacement.

replaceFirstRegEx

maconomy:replaceFirstRegEx(haystack, needlePattern, replacement)

Look for needlePattern in haystack and replace the first occurrence with replacement.

NeedlePattern is a regular expression that follows the Java regular expression syntax. For more information, see http://docs.oracle.com/javase/6/docs/api/java/util/regex/Pattern.html

The replacement string may contain placeholders that capture part of the regular expression. For more information, see: http://docs.oracle.com/javase/1.4.2/docs/api/java/util/regex/Matcher.html#appendReplacement(java.lang.StringBuffer, java.lang.String)

Parameters

haystack – The string to search in.

needlePattern – String that contains a regular expression to match in haystack.

replacement – The string to replace the first occurrence of needle with.

Returns

This function returns a string that is haystack with the first occurrence of the needle pattern replaced with replacement.

replaceAllRegEx

maconomy:replaceAllRegEx(haystack, needlePattern, replacement)

Look for needlePattern in haystack and replace all occurrences with replacement.

NeedlePattern is a regular expression that follows the Java regular expression syntax. For more information, see: http://docs.oracle.com/javase/6/docs/api/java/util/regex/Pattern.html

The replacement string may contain placeholders that capture part of the regular expression. For more information, see: http://docs.oracle.com/javase/1.4.2/docs/api/java/util/regex/Matcher.html#appendReplacement(java.lang.StringBuffer, java.lang.String)

Parameters

haystack – The string to search in.

needlePattern – String that contains a regular expression to match in haystack.

replacement – The string to replace all occurrences of needle with.

Returns

This function returns a string that is haystack with all occurrences of the needle pattern replaced with replacement.

matchRegEx

maconomy:matchRegEx(haystack, needlePattern)

Match the needle pattern in haystack.

NeedlePattern is a regular expression that follows the Java regular expression syntax. For more information, see: http://docs.oracle.com/javase/6/docs/api/java/util/regex/Pattern.html

Parameters

haystack – The string to search in.

needle pattern – String that contains a regular expression to match in haystack.

Returns

This function returns true if haystack matches the needle pattern. Otherwise, false.

Mathematical Functions

These are common mathematical functions:

min

maconomy:min(num, nums...)

Find the smallest value of one or more numeric arguments.

Parameters

num – The first numeric value.

nums – Any number of additional numeric values.

Returns

This function returns the smallest of its arguments.

max

maconomy:max(num, nums...)

Find the largest value of one or more numeric arguments.

Parameters

num – The first numeric value.

nums – Any number of additional numeric values.

Returns

This function returns the largest of its arguments.

abs

maconomy:abs(num)

Find the absolute value of a numeric value.

Parameters

num – The numeric value.

Returns

This function returns the absolute value of its argument.

sign

maconomy:sign(num)

Find the sign of a numeric value.

Parameters

num – The numeric value.

Returns

This function returns 1 if the value is positive, 0 if the value is zero, and -1 if the value is negative.

round

maconomy:round(num)

Round the value to the nearest integer using the half away from zero rule.

Parameters
num – The numeric value.

Returns
This function returns its argument rounded to the nearest integer.

floor

maconomy:floor(num)

Round the value to the nearest integer using, always rounding down.

Parameters
num – The numeric value.

Returns
This function returns its argument rounded to the nearest integer.

ceiling

maconomy:ceiling(num)

Round the value to the nearest integer using, always rounding up.

Parameters
num – The numeric value.

Returns
This function returns its argument rounded to the nearest integer.

MDL

Overview

The Maconomy system contains a large number of dialog windows and printouts. Maconomy supplies tools to edit the layout of these dialogs and printouts. This Maconomy Layout Languages section describes Window Design using MDL.

A separate section on preprocessing commands is provided in “MDL and MPL Preprocessor.”

This version of the manual describes the functionality of the layout editing tools supported by TPU 2.4.x.

MDL and MPL Preprocessor

This section describes the preprocessor functionality in MDL and MPL. This was introduced in TPU 53.

The preprocessor functionality is also available from M-Script. For more information please see the M-Script section in the Deltek Maconomy Portal Extension Programmer's Guide.

The MDL and MPL preprocessor functionality allows code sections of MDL and MPL (both in standard prints and universe reports) to be dependent on add-ons as well as system parameters and system information. This means that you can, for example, define a dialog which only shows a certain island if a certain system parameter has been marked, or a printout which only shows certain information if a certain add-on has been installed.

The main reason for using this is to leave out parts of layouts, which are irrelevant in certain setups. This functionality may be used mostly by the Deltek R&D department, but can be used by any layout and MRL report developer.

Versions

The preprocessor is available in MDL and MPL as of TPU 53.

To be able to import MDL layouts using preprocessor directives, a Maconomy Windows client version 4.3.0 is necessary. The preprocessor will, however, work correctly on older clients, and MPL with preprocessor directives can be imported with older clients as well.

Application version 8.0SP11 is necessary for automatic recompilation of MPL after changes to system parameters and system information. This application version also renames all system parameters such that pseudo localization tags (@) are removed.

Preprocessor Options

The preprocessor is applied to MDL and MPL before the relevant compiler is invoked. This means that the preprocessor directives can occur anywhere in the MDL or MPL syntax.

Syntax

The syntax for preprocessor options is as follows:

```
#if <expression>
    ...
#endif
```

and

```
#if <expression>
    #else
#endif
```

The # directives must occur as the first token on a line.

`<expression>`

is one of

```
addon(<number>)
systemparameter.<systemparametername>
systeminformation.<systeminformationfield>
```

For systems running with Danish kernel language, this last option would be:

```
systemoplysning.<systeminformationfield>.
```

Note that a number of system parameters have an “@” as the first character. If this is the case, you must enclose the entire system parameter in a pair of backslashes, as illustrated in the following examples.

```
## WRONG:
#if systemparameter.@AllowChangeofVATOnInvoiceLines
    ...
## CORRECT:
#if systemparameter.\@AllowChangeofVATOnInvoiceLines\
    ...
```

The system parameter or the system information field must be of the type Boolean. Otherwise, an error message is produced (can be viewed in the `LayError.txt` resp. `PrintLayoutErrors.txt` file, which is placed in the Maconomy client folder). If an add-on, a system parameter or a system information field does not exist, it is treated as `false`.

Examples

Write a Text if an Add-On is Set

```
<island "Add-on">
    #if addon(65)
```

```
       "Add-on 65 is set"
   #endif
<end island>
```

Write a Text if an Add-On is Set, Otherwise Another

```
<island "Add-on">
#if addon(65)
   "Add-on 65 is set"
#else
   "Add-on 65 is not set"
#endif
<end island>
```

Write in Italics if an Add-On is Set

The preprocessor directives can be used everywhere in the layout.

```
"Hi"
#if addon(65)
   :italic+
#endif
```

Write a Text if a System Parameter is Set

```
#if systemparameter.UseDailyTimeSheets
   "We use daily time sheets"
#endif
```

Write a Text if a System Information Field is Set

```
#if systeminformation.DifferentialVAT
   "We use differential tax"
#endif
```

Negate a Boolean Criterion

There is no syntax for negating a Boolean criterion. Instead you write:

```
#if systemparameter.UseDailyTimeSheets
#else
   "We don’t use daily time sheets"
#endif
```

Warning

You should be aware that using the preprocessor functionality increases the risk of introducing illegal MDL/MPL layouts in the system. Consider the following fragment:

```
#if systemparameter.UseDailyTimeSheets
   "Add-on 65 is set"
```

```
#else
    <island "title> .unknownfield <end stack>
#endif
```

The fourth line contains three errors: Missing " after `title`, reference to an unknown field, and an attempt to match `<island>` with `<end stack>`.

Nevertheless, if the system is set up to use daily time sheets, this MDL will be validated with no problems. This is due to the fact that the preprocessor is invoked prior to invoking the MDL compiler, which would otherwise detect the problems.

Now suppose that the **Use Daily Time Sheets** system parameter is changed to `false`. Now the layout will suddenly be invalid, and users will not be able to open the window for which the layout is defined.

It is therefore recommended that all MDL/MPL is tested thoroughly before any preprocessor directives are inserted.

Recompilation

This section describes what goes on behind the scenes—it does, however, contain important information and is recommended reading.

Automatic Recompilation

Because MDL is compiled every time that a window is opened, changes in system parameters, system information, or add-on information are immediately reflected in the MDL. Likewise, MPL for Universe Reports is compiled when the report is executed, and does not need recompilation.

> **Note:** Automatic recompilation requires application version 8.0 SP11.

Standard MPL for prints, however, does need recompilation. For system parameters and system information fields, this is achieved in the following manner:

1. When the application detects that a system parameter or a system information field has been changed, all compiled MPL layouts are deleted. They are not immediately recompiled, because this will present the server with too heavy a load.
2. When a user opens a print window, all layouts that are marked as validated for the print in question are recompiled. If a layout cannot be recompiled due to errors, it is marked as non-validated and will not appear in the layout selection pop-up.

Hence, problems in MPL and MDL layouts are only detected when the layout is used. This stresses the point made previously that special care should be taken when writing MDL/MPL with preprocessor directives.

If an add-on is changed using MConfig, all layouts are recompiled. Note that it should be checked that no errors arose from this.

Currently, the MBuilder has no connection to the database. Therefore, system parameters and system information fields are not evaluated in preprocessor directives at installation time. Instead all these are assumed to be `true`. Hence, the `#else` branch is never checked.

Manual Recompilation

Automatic recompilation is not performed if add-on information, system parameters, or system information is changed by other means than the Maconomy client or MConfig. This could, for example, be the case if system parameters are imported into the system. In this case, it is important to manually run `MaconomyServer` with the options `-UP` and `-UVP`. This will ensure that the prints reflect the current settings of the system.

The consultant performing these changes might find it useful to also run `-UVP` to check if all user modified window layouts are still valid. We assume that all standard layouts are still valid. There is currently no way of validating installed MPL for Universe Reports.

Introduction

Maconomy Dialog Language (MDL) is a language that is used to describe the Maconomy window layout. In MDL you can specify the fields and texts of a window, and arrange the fields in columns and in islands. Based on this, Maconomy performs the final formatting of the window. By means of MDL it is thus possible to focus on the content and the logical construction of windows. A window description that is written in MDL is called a *layout*.

An MDL layout contains the logical structure of the window data. The actual placing of window elements is handled by Maconomy. A typical Maconomy window is described by stating:

- Which panes the layout consists of.
- Which islands the card pane consists of.
- Which texts and database fields each island consists of.
- Which columns the table pane consists of.

To determine the position in the window you specify which elements are to be placed next to or above other elements.

> **Note:** Readers who are already familiar with languages such as HTML or TeX will find that the basic idea of MDL is no different from these languages.

Prerequisites

To use this manual, it is required that you have a basic knowledge of Maconomy. In addition, you should be able to use an editor (for example, TextPad or Notepad) and Maconomy's layout windows as described in the section about the Set-Up module in the Maconomy Reference Manual.

This manual is meant as an introduction as well as a work of reference, meaning that it might at times be rather technical. On first reading, it might therefore be a good idea just to glance quickly through the reference section, because most concepts are explained later by means of examples.

Preprocessor

As of TPU 53, MDL supports the use of *preprocessor directives*. A preprocessor directive enables the layout programmer to check for the value of system parameters, the value of system information options, and the existence of add-on programs before the layout is displayed. Depending on whether, for example, a certain system parameter is `true`, the layout can display different information.

The preprocessor functionality applies both to MDL and to MPL, and is therefore described separately in "MDL and MPL Preprocessor" in this manual.

Getting Started

MDL is a tag-based language that consists of elements and attributes, similar to XML. Unlike XML, attributes in MDL have an associated type, and both tags and attributes can have a short form.

In addition, MDL is case-sensitive, meaning that the use of lower-case and upper-case letters is significant. All tags and attributes that are used in MDL must be written in lower-case letters. Names of database fields and variables are not case-sensitive.

Any tags that are mentioned in this section are explained in more detail in the reference section.

Installation

This manual concentrates on the MDL language. For a description of how MDL layouts are validated and imported into Maconomy, see the Set-Up module in the Maconomy Reference Manual and "Server Options" in the Maconomy System Administrator's Guide.

Error Handling

Error messages are saved to the file `LAYERR.TXT` in the Maconomy folder at the time of validation. If you are using the Maconomy client for the Java™ platform, layout errors are shown in a separate browser window. If no errors occur when importing a layout, no file is generated. Error messages are generated in case of errors in the layout syntax or semantics. See “Error messages” for a list of each possible error message, with an explanation of each message, including any problem-solving suggestions.

Parenthetical and Simple Tags

Maconomy makes a distinction between *parenthetical* and *simple* tags. Parenthetical tags (or *block tags*) are characterized by the fact that they only come in pairs. For instance, the tag `<layout>` is parenthetical—it has a starting tag `<layout>` and an ending tag `<end layout>`. Neither can be omitted.

Simple tags are contained within a single set of angle brackets. All attributes are listed within the tag. Example: the tag `<title` *attributes*`>`. They are most often used in their short form.

Attributes

Attributes are MDL’s way of connecting data to tags, and are specified as name/value pairs inside tags:

attributename=attributevalue

Attribute names are always written in lower-case characters. With attributes, you can control the formatting and other aspects of the layout. In the reference section, all attributes are described with the tag where they are used.

Attribute Types

Every attribute has an associated type. The type of each attribute is specified in the description of each tag in the reference section. The table below describes each type. The attribute that illustrates each type is highlighted in **bold**.

Attribute Type	Description
ID	Attribute word. An attribute word is a unique identifier, consisting of a maximum of 100 alphanumeric characters and underscores, which has a certain meaning for a certain attribute. Example: `<var **data=MonthVar** width=12>` Here, the `data` attribute is of the type *ID*, where the attribute value is the name of a variable or a database field. An ID is a unique identifier, consisting of alphanumeric characters and underscores.
STRING	Text strings. A text is specified by placing a maximum of 100 characters between double quotes. If you want the text to contain quotes, write two double quotes right after each other in the text where you want the quotes to appear. Example: "He said, ""You can use double quotes""." The above text string will result in the following text: He said, "You can use double quotes". Example: <text title="Employee No.">
FSTRING	A text in single quotes—for example, `'/'`. See “Very Short Texts.”
BOOLEAN	Truth value – true or false. Example: <text title="Employee No." **italic=true**>
INTEGER	Integer value (theoretical max.: 21474836467). Example: <layout title="My layout" **split=200**>
REAL	Floating-point number. Floating-point numbers are specified by a period. Further- more, floating-point numbers have a maximum of 3 decimals. This means that `1.0` is allowed, while `1.0000` is not allowed. Theoretical range: 1.7E308 to -1.7E308. Example: `<var data=MonthVar **width=12.5**>` You can specify integers instead of decimal numbers. This means that the following expression is allowed: <var data=MonthVar **width=12**>
PAIR	Values of the type *PAIR* are used for specifying coordinates. A pair is written as (*n*,*m*) where *n* and *m* are of the type *INTEGER*. Example: <layout title="Example" **position=(100,200)**>

Attribute Type	Description
TRIPLE	Values of the type *TRIPLE* are used for specifying the RGB value of colors. Triples are written as (*2.0*,*x2*,*x3*) where *2.0*,*x2*, and *x3* are of the type *REAL* (floating-point numbers). Integers can be used instead of floating-point numbers. Example: <text title="Example" **rgb=(100,50,100)**>

Nameless Attributes

Nameless attributes are attributes for which the name of the attribute is implied. The general rule is that the attribute name of a nameless attribute is implied by the type of the attribute value. This means that instead of:

```
<text title="Time Sheet Summary">
```

you can use:

```
<text "Time Sheet Summary">
```

The `"Time Sheets"` attribute is a *nameless attribute* because the attribute name `title` has been left out. In general, you can say that if an attribute is nameless, it is sufficient to specify the value of the attribute.

Example:

```
attribute_name=attribute_value
```

This attribute is nameless and it is therefore sufficient to use:

```
attribute_value
```

A maximum of one nameless attribute per type can be used per tag. For instance, the `tooltip` attribute cannot be nameless, because the `title` attribute has the same type (*STRING*) and can be used nameless in the `action` tag. Note that there may be a difference between which attribute is nameless in long or short form. For instance, `justification` is nameless in the short form of the tag `field`, whereas `data` is nameless in the long form.

The following table shows examples of tags where the attributes are not abbreviated, followed by the corresponding attributes used as nameless.

Full-Length Attributes	Nameless Attributes
`<layout title="Time Sheets">`	`<layout "Time Sheets">`
`<text title="Employee No." justification=right width=15.0>`	`<text "Employee No." right 15.0>`

See the description of each tag to see which attributes can be used as nameless.

Boolean Attributes

Boolean attributes are usually specified as follows:

```
attributename=true
```

```
attributename=false
```

However, Boolean attributes can have the following short forms:

```
attributename+

attributename-
```

For instance, you can abbreviate

```
<text title="A bold text" bold=true>
```

to

```
<text title="A bold text" bold+>
```

Units of Measure

Some attributes, such as `width`, require that you specify a number. The default unit of measure is *em,* meaning the width of the letter “M” in the selected font and font size. Since “M” (usually) is the widest letter of the alphabet, you can be sure that the space that you set aside in *em* will be sufficient for the text you specify whether the number of letters match the number of *em*s. Hence, the tag `<text title="A new text" width=10>` specifies that a text box with the text “My text” should be drawn, and that it should be 10 *em* wide.

Short Forms

For convenience, a short form notation exists for most simple tags in MDL. The short form is recommended over the long form, because it is more compact and readable. For instance, the tag `<text title="Hello world!">` is simply abbreviated to `"Hello world!"`. Short forms are different from tag to tag. The following reference section specifies both the long form and the short form of each MDL tag.

Comments and Whitespace

You can insert comments in your code by entering two hyphens. The rest of the line after the hyphens is ignored by the MDL compiler. See the following example. Remember to comment your code liberally to document it for the future.

```
-- Line comment – any text between the dashes and the next

-- newline is considered a comment.
```

Whitespace characters, that is blank characters, tab characters, and carriage returns, have no influence on the result of a MDL layout; only the logical structure has. Consider the following example:

```
<row>  "Name" Name1Var      <end row>
```

and

```
<row>

    "Name"

    Name1Var

<end row>
```

These examples result in the same position in the window (a row that contains a text string and a variable), because the `<row> ... <end row>` tag specifies the inclusion of a row, not whether the contents are represented on one or more lines.

How to Read the Syntax

In the following reference section, an example of the usage of the tag described is shown first. This is followed by a table that shows all of the available attributes to the tag. The table shows the type of the attribute, whether it is mandatory (“M”), whether it is nameless (“N”), and a description of the

tag. If the attribute is mandatory or nameless, it is marked by a “+” in the table (or a “(+)” if special conditions apply). For a description of each attribute type, see “ Attribute types.”

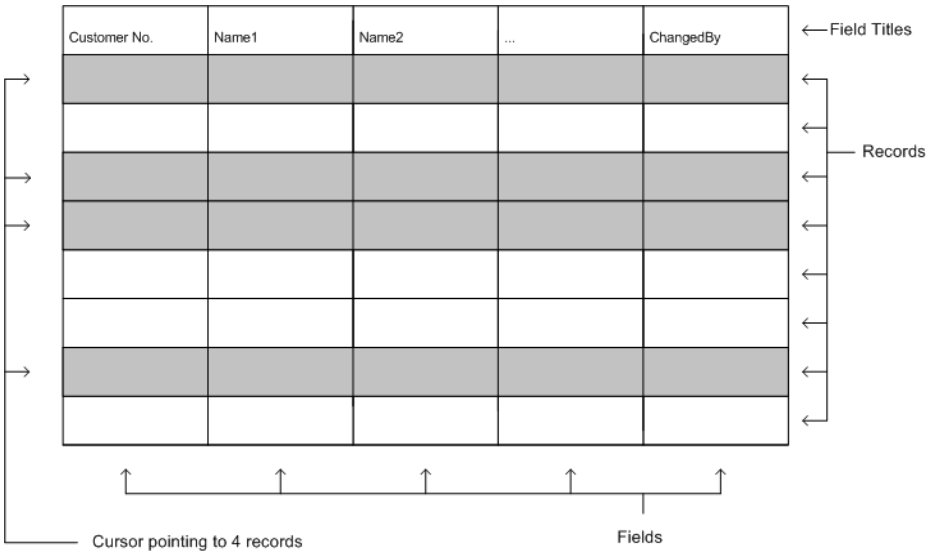
Before you use the reference section, you should be familiar with some general MDL concepts.

General Concepts

This section introduces terminology and concepts that are necessary for the understanding of this manual and for using MDL.

Database Fields and Variables

- **Relations** — Relations are collections of data in the Maconomy database. In this context, a relation can be perceived as a table of data. Such a table consists of a fixed number of columns and a varying number of rows. Each cell in a row contains data, and each column has a unique name, for example, CustomerNo. or Name1, signifying the contents of each column.
- **Fields and Records** — In SQL terminology, columns are called *fields*, and rows are called *records*. Thus a relation is a collection of records, each with a number of named fields containing data as shown in the figure below.



Maconomy is built on a number of different relations. For instance, one relation contains customers, and another relation contains vendors.

A *cursor* points out a number of the records that are contained in a relation. Sometimes a cursor points out all records, but often a cursor is subject to certain assigned *conditions*, which ensure that only specific records are included.

When using Maconomy, the system often displays values that are not contained in a relation. These values are calculated from data in the relations and values entered by the user. For example, the discount amount on an item line is calculated from the discount percentage and the extended price.

- **Variables** — The results of these calculations are put into *variables* that can be shown on the screen exactly like database fields.

Window Types

- **Parameter windows** — There are three types of windows in Maconomy that can be modified by MDL. Windows in which you specify selection criteria and perform an action are called parameter windows. Parameter windows typically result in a printout. Some parameter windows are available from the Menu menu, and all windows that are opened using the function “Print...” in the File menu are parameter windows. Parameter windows can only contain variable fields.
- **Dialog windows** — Apart from parameter windows, all “normal” windows from the Menu menu are called *dialog windows*, such as the Time Sheets or Customer Information Card windows. Dialog windows can contain variables as well as database fields.
- **Search windows** — When you press Ctrl+F to perform a search in a dialog window, a search pane opens. Even though the search pane is integrated into the dialog window, it has a separate MDL specification. Pressing Ctrl+G opens a separate search window. For more information, see “Panes.”

Panes

All of Maconomy’s windows are constructed of *panes*. A window contains one, two, or three panes. If there is more than one pane in a window, the panes are separated by a split line that can be moved by the user. There are three kinds of panes that can be described as follows:

- **Card panes** — In a *card pane* (or tab), the information is presented as fixed texts followed by fields. A card pane usually corresponds to the content of an entry (record) in a database relation, but it frequently also contains variables. Card panes are usually arranged in *islands* that have a logical grouping of texts and fields. An island has a frame and usually a heading. The islands are placed in one or two columns in the pane.
- **Table panes** — In a *table pane* (or sub-tab), the information is presented as a table that has columns, each of which has a heading. A table pane conceptually corresponds to a database cursor, and each row corresponds to an entry (record) in the cursor.
- **Search panes** — In a *search pane*, the information is presented as a table that has columns, similar to the table pane. In the Maconomy client, the user can choose to see more or fewer columns from the list of columns that are specified in the MDL layout. By default, the first ten columns are shown. Search panes have separate MDL layouts, and are not part of the dialog window layout.

MDL Panes

In MDL terminology, the search pane is described as a table pane. For the card pane, a distinction is made between single and double column panes. In a single-column pane, all of the pane elements are placed in one column in the pane. In double-column panes, the elements are placed in two columns. Single-column panes contain *boxes*; double-column panes contain *groups* (see the next section). Only single-column panes in parameter windows and in dialogs that have only one card pane can be empty. All pane types are described by a pane tag and an end-pane tag. Layout elements between these tags are placed in the specified pane.

> **Note:** Groups as formatting elements do not make sense in connection with single-column panes. Note that single-column pages must contain at least one box. However, parameter dialogs and card dialogs can contain an empty single-column pane.

Note that when you specify a pane in a parameter window, Maconomy draws the Cancel and Print buttons. Search windows are always one table pane, where Maconomy draws the rest.

See more information in the description of the pane tags in the reference section.

Window Formatting

You can set the default way in which panes and windows are to be formatted with regard to size and position, and so on. Apart from the `widthfactor` (connected to the `dualpane` tag), these attributes only apply the first time that the window is opened, because Maconomy saves the current size when the window is closed. For more information, see the description of the `layout` tag.

Window Layouts in Maconomy

Maconomy is shipped with one or two original layout(s) for every window. There is always a *standard* layout, and sometimes there is a *solution* layout (which applies to one of the Maconomy solutions).

Often you want to design a new window in such a way that it resembles the original layout; you might want to move a few fields or islands around, leave out some fields, or change some fixed texts. In this case, the adjustment can be made in a few minutes by using the original layout for the window in question. This is done by exporting the “standard” or “custom” layout and making the adjustments in a text editor. When an original layout is exported, the MDL layout is automatically named “Original.”

Even if the window layout has to be changed radically, it is usually a good idea to use the original layout as a template. This is because it makes it easy to find out which variables and database fields are available, because they are all included in the original layout.

For a description of exporting layouts, see the description of the Window Layouts window in the Set-Up module in the Maconomy Reference Manual.

Reference

This section contains a description of the general structure of an MDL layout and a description of each MDL tag and its possible attributes. For each tag, a table of attributes is shown with name, type, description, and indication whether the attribute in question is mandatory (“M”) or nameless (“N”). In addition, a subsection of the BNF structure of the tag in question is shown. For the full BNF structure, see “MDL Syntax.”

Note that a number of attributes apply to the Maconomy client for the Java™ version only. When such attributes are encountered by other Maconomy clients, they are silently ignored.

This section contains all of the information that you need to design your own layouts. Examples of layouts are discussed in “Layout Examples.”

General Structure

All layouts begin with a special *root tag*:

```
<mdl 4>
```

In addition to the root tag, MDL specifications have the following structure:

```
<mdl>
<layout>
    <pane>
                ...
    <end pane>
```

```
    <pane>
        ...
    <end pane>
<end layout>
```

Remember that tag names must be in lower case. Each of the tags that are mentioned previously is shown without attributes.

mdl

The `mdl` tag is the root tag and can be written as:

```
<mdl 4>
...
```

Attribute	Type	M	N	Description
N/A	*INTEGER*	+	+	MDL version. Can be less than 4, but `4` is recommended.

This tag is special in that it does have an attribute, but the attribute can only be specified as nameless.

The current version of MDL is backward-compatible with earlier versions of the language.

Structure

start = `<mdl` *INTEGER*`>` *layout* Version no. 1-4.

layout

The `layout` tag is a parenthetical tag. It follows immediately after the `mdl` tag and can be written as:

```
<mdl 4>
<layout>
    ...
<end layout>
```

Attribute	Type	M	N	Description
findsuppressed	BOOLEAN			If set and `true`, the Navigate menu is disabled in the current window in the Java™ client.
initialfocusincard	BOOLEAN			If set and `true`, when you open the window in the Java™ client the initial focus is in the card part, and not in the table part which is the default.
name	ID	(+)	+	MDL 4. This attribute is mandatory in MDL 4 and above. The name is the internal identifier of the current layout. Example: `name=Key_Employees`

Attribute	Type	M	N	Description
position	PAIR			Specifies the position of the window on the screen the first time the window is opened. The attribute is specified as follows: `position=(x,y)`, where the coordinates are specified in pixels (integers). If you do not specify this attribute, a suitable position is automatically calculated.
size	PAIR			Specifies the size of the window on the screen the first time the window is opened. The attribute is specified as follows: `size=(width,height)`, where the coordinates are specified in pixels (integers). If you do not specify this attribute, the size is automatically calculated on the basis of the window contents.
split	INTEGER			Only used for layouts with two panes. It specifies the position of the split line the first time the window is opened. The attribute is specified as follows: `split=`*height*, where the height is an integer specifying the number of pixels from the top of the window. If the window size does not allow the split line to be placed as specified, the line is moved upward until it fits.
toolbarsuppressed	BOOLEAN			If set and `true`, the toolbar is removed in the current window in the Java™ client.
title	STRING	+	+	The `title` attribute is mainly used for specifying texts to be shown on the screen. However, in the `layout` tag the title is not printed to the screen, but is only used for reference. The title is displayed in some of the window layout dialogs in Maconomy. You cannot use underscores (_) in the title, and the title cannot begin or end with spaces. MDL 4: As opposed to the `name` attribute, this attribute is external, can be localized and edited. Example: `title="My new layout"`
windowname	ID	(+)		MDL 4. This attribute is mandatory in MDL 4 and above. The window name is the internal name of the window with which the current layout is associated. If you attempt to import a layout for a different window than what is specified here, the import will fail. Example: `windowname=Jobs`

Attribute	Type	M	N	Description
windowtype	ID	(+)		MDL 4. This attribute is mandatory in MDL 4 and above. Specifies the type of the window to which the current layout applies. If you attempt to import a layout for a window of a different type than what is specified here, the import will fail. You can choose between the following options (types of window): `DialogWindow`, `SelectionWindow`, `ParameterWindow` (print window) and `Report` (Analyzer report).

Example

```
<mdl 4>
<layout Original "Original"
    windowtype=DialogWindow windowname=BudgetInspection>
    ...
<end layout>
```

Structure

Tag	Structure	Comments
layout	= <layout *attribute*$_1$ .. *attribute*$_n$> *availableactions* *availableshortcuts* *pane* <end layout>	*availableactions* and *availableshortcuts* can be omitted.
	| <layout *attribute*$_1$ .. *attribute*$_n$> *availableactions* *availableshortcuts* *pane*$_1$ *pane*$_2$ <end layout>	*pane*$_1$ must be *singlepane* or *dualpane*, *pane*$_2$ must be *tablepane*.

availableactions

Affects the Java™ client only. In windows that contain actions, you can define which of the actions should be available for selection in the current window. By default, all actions that are specified by Maconomy developers are available, as are any actions that are specified using the Maconomy extension language MEXL. However, if you specify available actions using MDL, only the actions specified here are available (or unavailable, depending on the attribute `exclude`).

The `availableactions` tag is a parenthetical tag and can be written as:

```
<mdl 4>
<layout>
   <availableactions>
      <action>
      ...
   <end availableactions>
   ...
```

Attribute	Type	M	N	Description
exclude	BOOLEAN			This attribute is by default `false`, meaning that the actions specified in the `<action>` tags below appear in the Action menu of the current window. If `exclude` is set to `true`, the actions specified in the `<action>` tags below are not available in the window.

Example

See the `action` tag for an example.

Structure

Tag	Structure	Comments
availableactions	= `<availableactions>` *action*$_1$ .. *action*$_n$ `<end availableactions>`	*availableactions* need not contain any actions.

action

Affects the Java™ client only. The simple `action` tag can be written as:

```
<mdl 4>
<layout>
   <availableactions>
      <action>
      ...
   <end availableactions>
   ...
```

Attribute	Type	M	N	Description
actionname	ID	+	+	This is the internal name of the action as specified in MEXL.

Attribute	Type	M	N	Description
title	STRING		+	This string specifies the menu title of the current action. If it is not specified, the title is the default title as specified in MEXL.
tooltip	STRING			This string specifies the tooltip shown when the mouse hovers above the action in the menu (or the button if the action is defined as a button).

Examples

The following examples are from the Time Sheets window.

Example 1

```
--The Action menu contains the actions specified in DDL/MEXL:
<mdl 2>
<layout "Standard">
    <dualpane>
        <group>
        ...
```

Example 2

```
-- The available actions are "Release_TimeSheet", "Copy_TimeSheet", and "B"
(where "B" is a Custom Action):
<mdl 2>
<layout "Standard">
    <availableactions>
        <action actionname=Release_TimeSheet title="Release TS">
        <action B title="B Button">
        <action Copy_TimeSheet>
    <end availableactions>
    <dualpane>
        <group>
        ...
```

Example 3

```
-- The actions "Release_TimeSheet" and "Copy_TimeSheet" are removed
-- from the list of available actions:
<mdl 2>
<layout "Standard">
    <availableactions exclude+>
        <action actionname=Release_TimeSheet>
        <action Copy_TimeSheet>
```

```
<end availableactions>
<dualpane>
    <group>
    ...
```

Structure

Tag	Structure	Comments
action	= <action *attribute*$_1$ .. *attribute*$_n$>	

availableshortcuts

Affects the Java™ client only. In windows that contain actions you can define shortcuts for one or more of the available actions. For each shortcut, you can specify a shortcut key within the range a..z, A..Z, and 0..9. Note that function keys (F1-F12) cannot be specified as shortcut keys. Note also that Maconomy does not validate your choice of shortcut key. If there is a conflict with an operating system shortcut key, the operating system prevails.

The `availableshortcuts` tag is a parenthetical tag and can be written as:

```
<mdl 4>
<layout>
    <availableshortcuts>
        <shortcut>
        ...
    <end availableshortcuts>
    ...
```

It does not take any attributes.

Example

See the `shortcut` tag for an example.

Structure

Tag	Structure	Comments
availableshortcuts	= <availableshortcuts> *shortcut*$_1$ .. *shortcut*$_n$ <end availableshortcuts>	*availableshortcuts* need not contain any shortcuts.

shortcut

Affects the Java™ client only. The simple `shortcut` tag can be written as:

```
<mdl 4>
<layout>
    <availableshortcuts>
```

```
    <shortcut>
    ...
<end availableshortcuts>
...
```

Attribute	Type	M	N	Description
actionname	*ID*	+	+	This is the internal name of the action as specified in DDL/MEXL to be executed when the shortcut is pressed in the client.
key	*STRING*	+	+	This is the shortcut key. One letter or digit only in the range a..z, A..Z, and 0..9.
shift	*BOOLEAN*			If specified, the Shift key must be pressed with the shortcut key defined in `key`.
alt	*BOOLEAN*			If specified, the Alt key must be pressed with the shortcut key defined in `key`.
ctrl	*BOOLEAN*			If specified, the Ctrl key must be pressed with the shortcut key defined in `key`. On the Macintosh, the Ctrl key maps to the Apple Command (⌘) key.

Example

The following example is from the Time Sheets window. Alt+S submits the current time sheet, and Ctrl+R executes the **Reopen Time Sheet** action:

```
<mdl 4>
<layout TSShortcuts "Time Sheets Shortcuts" windowtype=DialogWindow
windowname=TimeSheets>
    <availableshortcuts>
        <shortcut Submit "S" alt=true>
        <shortcut key="R" ctrl+ actionname=Reopen_TimeSheets>
    <end availableshortcuts>
    <dualpane>
        <group>
        ...
```

Structure

Tag	Structure	Comments
shortcut	= <shortcut *attribute*$_1$ .. *attribute*$_n$>	

singlepane

Specifies a single-column card part. The parenthetical `singlepane` tag can be written as:

```
<mdl 2>
<layout>
    <singlepane>
        <island>
        ...
    <end singlepane>
    ...
```

It does not take any attributes.

Example

For several examples of the use of `singlepane`, see “Layout Examples.” Note that `singlepane` is also the preferred tag for creating windows that have more than two columns. For an example of this, see “Islands in more than Two Columns.”

Structure

Tag	Structure	Comments
singlepane	= <singlepane> *Buttons* *Dropdownbutton* *bo2.0 .. box$_n$* <end singlepane>	*singlepane* need not contain any boxes, buttons, or dropdown button.

dualpane

Specifies a double-column card part. The parenthetical `dualpane` tag can be written as:

```
<mdl 2>
<layout>
    <dualpane>
        <island>
        ...
    <end dualpane>
    ...
```

Attribute	Type	M	N	Description
widthfactor	*REAL*		+	Specifies the width of double-column panes. Width is specified in relation to the standard width: If, for instance, you want the pane to be 20% larger than the standard size, you can write `widthfactor=1.2`. If the pane is to be 20% smaller, use the expression `widthfactor=0.8`. The width of a dual pane has influence on the format of the groups in the pane; particularly for the floating of the elements (typically for the way in which islands are placed). The

Attribute	Type	M	N	Description
				attribute value for `widthfactor` is *REAL* or *INTEGER* (as for the `width` attribute).

Example

For several examples of the use of `dualpane`, see “Layout Examples.”

Structure

Tag	Structure	Comments
dualpane	= <dualpane attribute$_1$ .. attribute$_n$> *buttons* *dropdownbutton* *group$_1$ .. group$_n$* <end dualpane>	*dualpane* need not contain any groups , buttons, or dropdownbutton.

tablepane

Specifies a table part. The parenthetical `tablepane` tag can be written as:

```
<mdl 2>
<layout>
   <tablepane>
      <field>
      ...
   <end tablepane>
   ...
```

Attribute	Type	M	N	Description
hidelineid	*BOOLEAN*		+	If the window uses hierarchical table structure, you can set this attribute to `true` if you do not want to show the line ID of each line. The line ID only makes sense in hierarchical table structures. Note that if `hidelineid` is `true`, the ability to show or hide the line ID in the client interface disappears.

Example

For several examples of the use of `tablepane`, see “Layout Examples.”

Structure

Tag	Structure	Comments
tablepane	= <tablepane *attribute*> *Buttons* *Dropdownbutton* *column*$_1$ .. *column*$_n$ <end tablepane>	*tablepane* must contain at least one column. Buttons and dropdownbutton can be omitted.

Buttons

You can create buttons in MDL that execute actions. The buttons are only displayed in the Maconomy client for the Java™ platform.

Buttons can be placed as the first element in the three *pane* tags: `dualpane`, `singlepane`, and `tablepane`. The buttons are displayed at the top of the pane, either the card part or the table part. By defining buttons, you make it easier for the user to select actions in the window.

A special type of button, the drop-down button, collects a number of actions in one button. The user first clicks the button and then selects an action.

The actions that are specified as buttons must be included in, or not excluded by, the `<availableactions>` tag in the beginning of the layout, if it exists. In addition, a number of other actions are shown as buttons, but are beyond the control of MDL:

- In the card part, a “Save” button is shown.
- If navigation is enabled by the Maconomy developer, buttons for navigating through records, a “New” button and a “Delete” button are added in the card part.
- In the table part, table action buttons that correspond to the File menu commands “Insert Line,” “Add Line,” and “Delete Line” are added.
- In addition, if the window uses hierarchical table structure, buttons that correspond to the Edit menu commands “Move Line Up” and “Move Line Down” are shown.

The `<buttons>` tag is a parenthetical tag and can be written as:

```
<mdl 2>
<layout>
   <dualpane>
      <buttons>
         <action>
         ...
      <end buttons>
   <end dualpane>
...
```

It does not take any attributes.

Example

The following example is from the Time Sheets window.

```
--Three buttons
<mdl 2>
    <layout "Standard">
        <dualpane>
            <buttons>
                <action actionname=Release_TimeSheet title="Release TS"
                tooltip="Release this time sheet">
                <action Reopen_TimeSheet title="Reopen timesheet" tooltip="Reopen
                this time sheet">
                <action Copy_TimeSheet>
            <end buttons>
            ...
```

Structure

Tag	Structure	Comments
buttons	= <buttons> *action*$_1$ .. *action* <end buttons>	*buttons* need not contain any actions.

dropdownbutton

This parenthetical tag contains one or more button definitions in the form of actions. The buttons are only displayed in the Maconomy client for the Java™ platform.

A dropdown button can be placed as the first element in the three *pane* tags: `dualpane`, `singlepane`, and `tablepane`. In the Java client, this is shown as one button that has a small arrow on the right. Clicking the arrow opens a list of the actions that are defined for the button. There is no default action for the button. The dropdown button is positioned to the right of any normal buttons. By defining a dropdown button, you can save space in your layout.

The actions that are specified in the dropdown button must be included in, or not excluded by, the `<availableactions>` tag in the beginning of the layout, if it exists.

The `<dropdownbutton>` tag is a parenthetical tag and can be written as:

```
<mdl 4>
<layout>
    <dualpane>
        <dropdownbutton>
            <action>
            ...
        <end dropdownbutton>
    ...
```

Attribute	Type	M	N	Description
title	*STRING*		+	The title of the button when it is closed (not “dropped down”).

Example

The following example is from the Time Sheets window.

```
--Two buttons and a dropdown button with three actions
<mdl 2>
    <layout "Standard">
        <dualpane>
            <buttons>
                <action actionname=Release_TimeSheet>
                <action actionname=Reopen_TimeSheet
            <end buttons>
            <dropdownbutton title="Actions">
                <action actionname=Approve_TimeSheet title="Approve Time Sheet">
                <action Copy_TimeSheet title="Copy Time Sheet">
                <action Transfer_Plan>
            <end dropdownbutton>
            <group>
            ...
```

Structure

Tag	Structure	Comments
dropdownbutton	= <dropdownbutton> *action*$_1$ .. *action*$_n$ <end dropdownbutton>	*dropdownbutton* need not contain any actions.

Group

A group does not correspond to any window element, but is only used to control the formatting of panes that have two columns (see `dualpane`).

If a pane that has two columns is read from left to right, the pane elements (typically islands) appear exactly in the order in which they appear in the MDL layout. More explicitly:

1. The first element of each group is placed to the left.
2. Subsequent elements are placed below the lowest column if the column width is large enough; otherwise they are placed to the left under both columns.

This process is called *floating*.

The parenthetical `group` tag can be written as:

```
<mdl 2>
```

```
<layout>
    <dualpane>
        <group>
            <island>
            ...
        <end group>
        …
    <end dualpane>
    ...
```

It does not take any attributes.

Example

For several examples of the use of `group`, see “Layout Examples.”

Structure

Tag	Structure	Comments
group	= <group> *bo2.0 .. box$_n$* <end group>	*group* must contain at least one box.

text

The most simple tag of MDL layouts describes text elements. The following expression, for instance, describes the text element `Employee No.`, which will be shown in the dialog window in accordance with the remaining attributes and enclosing formatting tag:

```
<text title="Employee No.">
```

Attribute	Type	M	N	Description
title	*STRING*	+	+	This nameless attribute specifies the title of the text element, that is the actual text written to the screen.

Attribute	Type	M	N	Description
text formatting				A number of text formatting attributes are available: ▪ bold ▪ color ▪ fontname ▪ fontsize ▪ italic ▪ justification ▪ rgb ▪ showasclosedfield ▪ stretch ▪ underline ▪ underline

Short Form

To further simplify the syntax of texts, MDL enables you to leave out the tag completely. If you write a string where a box is expected, Maconomy will regard it as a text.

Example: `"Employee No."` is regarded as

```
<text title="Employee No.">
```

Attributes are separated by colons. In the case of texts, `justification` and `width` are nameless. The following text uses all attributes:

```
<text title="My text"
    justification=center
    width=10.0
    stretch=high
    bold=true
    italic=true
    underline=true
    color=blue
    fontname=roman
    fontsize=12>
```

In the short form, the preceding expression will look like the following (the short form for Boolean attributes is also used in this example):

```
"My text":center:10.0:stretch=high:bold+:italic+:underline+:
color=blue:fontname=roman:fontsize=12
```

Very Short Texts

You often want a text to be as short as possible. This is especially the case when using separation characters such as '/' in a date specification or '-' in a range specification. In these cases, you might also want to center the separation characters. This can be done in the following way:

```
<text title="/" width=0.0 stretch=low justification=center>
```

Because this is used frequently, it also exists in a short form:

```
'/'
```

The `justification`, `width`, and `stretch` attributes cannot be used in connection with this short form; other attributes are specified as in the usual short form for texts.

Structure

Tag	Structure	Comments
text	= <text *attribute*$_1$ .. *attribute*$_n$> | *STRING:attribute*$_1$*: .. : attribute*$_n$ | *FSTRING:attribute*$_1$*: .. : attribute*$_n$	Long form. Double-quote text. Single-quote text.

field

A `field` tag is a reference to a database field. Fields can be used in both the card pane and in the table pane. However, in the card pane the field label is specified as a separate `text` tag (or using the `title` tag. See `title`.). In the table pane, the field label (that is, the column heading) is specified directly in the field tag as an attribute.

If, for instance, you want to show a field in the card part containing the database field `EmployeeNumber` and labeled with the text “Employee No.,” use the following expression:

```
<text title="Employee No.">
<field data=EmployeeNumber>
```

A similar field in the table part would look like the following (where `title` is the column heading):

```
<field title="Employee No." data=EmployeeNumber>
```

Attribute	Type	M	N	Description
data	*ID*	+	+	This nameless attribute specifies the database field to be source of content of the field.
fieldtitle	*ID*		(+)	This nameless attribute specifies the column heading of the field when used in the table part as the content of a field in the card part. Requires MDL version 2 or above.

Attribute	Type	M	N	Description
mandatory	*BOOLEAN*			This attribute is used for specifying that the user *must* fill in a database field or a variable. If the field/variable is open in *new* mode (see below), you cannot leave the *new* mode unless data is entered in the field/variable. If the field is open in *update* mode, and the user presses Enter, Maconomy will check that the field was not *changed* to be empty (cleared). In this way, the use of `mandatory` will ensure that fields that are open in *new* mode always contain a value. In connection with fields that are only open in *update* mode, `mandatory` only makes sense if the system completes the field with a value (a check will then be performed that this value is not deleted). The attribute can be used for ensuring that important information is entered. For instance, you might want to require a central national register number (social security number) to be entered when creating a new employee in Maconomy. This functionality is ensured by writing: `<field data=CNRNumber mandatory=true>` Note that the `mandatory` attribute cannot be used in connection with check boxes and pop-up fields.[1] Certain database fields are already made mandatory by Maconomy, and this cannot be changed. It is not possible to make closed fields mandatory, as the user will not be able to enter data as required. However, it is possible to close fields which are mandatory by Maconomy, as long as the format contains one occurrence of the field which is open in *new* mode, if the field is open in *new* mode by system default.
readonly	*BOOLEAN*			Because parameter windows have no *new* and *update* modes, the `ronew` and `roupdate` attributes (see below) have no meaning here. Instead, the `readonly` attribute, which closes variables for entry, is used (parameter windows can only contain variables).

[1] A check box always has a value (selected or not) and is therefore always mandatory. Certain pop-up fields are made mandatory by Maconomy. These pop-up fields are characterized by the fact that the blank value cannot be chosen. If MDL made it possible to make a pop-up field mandatory, the blank value would disappear—even in already entered data—resulting in potential conflicts.

Attribute	Type	M	N	Description
ronew	*BOOLEAN*			"ronew" is short for Read Only New and is used for specifying that no information can be entered in a field in the *new* mode. In Maconomy, this is, for instance, the case for all database fields and variables in the Time Sheets dialog except for `EmployeeNumber`, `WeekVar`, and `YearVar`. The `ronew` attribute can close fields that are usually open, but not the other way around. This attribute can only be used in dialog windows. In parameter windows, the `readonly` attribute is used (see above).
roupdate	*BOOLEAN*			"roupdate" is short for Read Only Update and is used for specifying that no information can be entered in a field in the *update* mode. In Maconomy, this is, for instance, the case for `EmployeeNumber`, `WeekVar`, and `YearVar` in the Time Sheets dialog (that is, the only database fields/variables that were not `ronew`).
summed	*BOOLEAN*			Table part of Java client only. If this attribute is set and `true`, a sum field is shown below the current table column, summing the fields above it, provided that the fields are of the type *REAL*, *INTEGER*, or *AMOUNT*. This is for instance used in the window Time Sheets in the Java client.
title	*STRING*		(+)	This nameless attribute specifies the column heading of the field when used in the table part.
variabletitle	*ID*		(+)	This nameless attribute specifies the column heading of the field when used in the table part. The heading is the content of a variable in the card part. Requires MDL version 2 or above.
visualizeastime	*BOOLEAN*			Java client only. If this attribute is set and `true`, the content of the field is formatted as time, respecting the "Interpret as minutes if above" setting in the Preferences window, provided that the field is of the type *REAL*. In the client, you can for example enter "90" and Maconomy will interpret it as 1 hour 30 minutes. This is for instance used in the SpeedSheet in the Java client.
zerosuppression	*BOOLEAN*			With this attribute, you can specify that zero values should be suppressed (omitted) from fields in the card part or the table part, provided that the fields are of the type *REAL*, *INTEGER*, or *AMOUNT*. Note that you cannot specify `mandatory` and `zerosuppression` for the same field at the same time.

Attribute	Type	M	N	Description
text formatting				A number of text formatting attributes are available: ▪ `bold` ▪ `color` ▪ `fontname` ▪ `fontsize` ▪ `italic` ▪ justificationon ▪ `rgb` ▪ `showaslabelifclosed` (in card part only) ▪ `stretch` (in card part only) ▪ `underline`

Exactly one of the attributes `title`, `fieldtitle`, and `variabletitle` can be used in the `field` tag when the tag is used in the table part. When the tag is used in the card part, use a text as a fixed text label.

If you omit the title specification in the table part:

```
<field data=JobNumber>
```

the heading of the column will be the Maconomy internal default title for the job number field.

A database field can be used several times in the same card pane, but only if at most one of the field occurrences in the pane is open in each mode (see “Modes”).

In the table part, you can also use a field or variable value from the card part as a column heading. This allows you to for example create a column whose heading corresponds to the date currently shown in a given field in the card part. If the date changes, either as a result of changing the value or browsing to another card part entry, the column heading changes accordingly. See the preceding attribute table. Example:

```
<field fieldtitle=fieldname data=JobNumber>
```

where *fieldname* is replaced by the name of a field that is available in the database relation that is used in the card part of the window. The preceding example results in a table column where you enter job numbers, but the heading of the column depends on the value of the specified field in the card part.

Usually, the same database field cannot appear more than once in the same table pane.

The description of the `mandatory`, `ronew`, `roupdate` and `readonly` attributes mentions the concept of *modes*. For more information about modes, see “Modes.”

Short Form

In the card pane, the short form for database fields consists of the database field with a leading dot, meaning that:

```
<field data=EmployeeNumber>
```

can be written as:

```
.EmployeeNumber
```

The dot tells MDL that this is a database field.

In the table pane, the short form for database fields is designed to resemble a fixed text (the heading) followed by a field.

Thus, the expression:

```
<field title="Job No." data=JobNumber>
```

can be written as:

```
"Job No." .JobNumber;
```

Note the semicolon that indicates the end of the short form. Or you can just enter:

```
.JobNumber;
```

if you want to use the default field title as heading.

Attributes are separated by colons, and are specified after the database field name.

Structure

Tag	Structure	Comments
field (card part)	= <field *attribute*$_1$ .. *attribute*$_n$> | .*ID*:*attribute*$_1$: .. :*attribute*$_n$	Long form. Short form.
column (table)	= <field *attribute*$_1$ .. *attribute*$_n$> | *STRING* .*ID*:*attribute*$_1$:..:*attribute*$_n$; | *ID* .*ID*:*attribute*$_1$:..:*attribute*$_n$; | .*ID* .*ID*:*attribute*$_1$:..:*attribute*$_n$;	Field column, long form. Field column with text heading from variable. Field column with heading derived from field.

Var

You can refer to MDL variables by using the `var` tag. This is done in exactly the same way as with database fields. If you want to have a field that contains, for instance, the variable `DepartmentNumberVar`, the following expression can be used:

```
<var data=DepartmentNumberVar>
```

For possible attributes, see the table in the section `field`.

The same distinction between variables in the card part and the table part is made as for database fields.

A variable can be used several times in the same card pane if only one of the occurrences is open in each mode (see “Modes”).

In the table part, a column that has the heading “Job Name” that contains the variable JobText10Varcan be specified as follows:

```
var title="Job Name" data=JobText10Var> fieldtitle , variabletitle
```

and omitting the title reference works as described for `field` above. Usually, the same variable cannot be used more than once in the same table pane.

Short Form

The short form for variables in the card pane corresponds closely to the short form for database fields. The only difference is that you do not place a dot before the variable name (the dot is used by MDL to distinguish between variables and database fields). Thus,

```
<var data=DepartmentNumberVar>
```

can be written as:

```
DepartmentNumberVar
```

The short form for variables in tables resembles the form for database fields. However, no period is used here.

```
<var title="Job Name" data=JobText10Var>
```

can be written as:

```
"Job Name" JobText10Var;
```

or simply:

```
JobText10Var;
```

if you want to use the default title as heading.

As in texts, attributes are separated by colons. `justification` and `width` are nameless.

Structure

Tag	Structure	Comments
variable (card part) =	= <var *attribute*$_1$ .. *attribute*$_n$> | *ID:attribute*$_1$: .. : *attribute*$_n$	Long form. Short form.
column (table)	| <var *attribute*$_1$ .. *attribute*$_n$> | *STRING ID:attribute*$_1$*:..:attribute*$_n$*;* | *ID ID:attribute*$_1$*:..:attribute*$_n$ | *.ID ID:attribute*$_1$*:..:attribute*$_n$*;*	Variable column long form. Variable column with text heading. Variable column with heading derived from variable. Variable column with heading derived from field.

title

In Maconomy, a default title is associated with some fields and variables. You can access this default title in MDL through the `title` tag. If you want the default title for the database field `EmployeeNumber`, the following expression can be used:

```
<title fieldtitle=EmployeeNumber>
```

Similarly, you can get the default title for a variable using:

```
<title variabletitle=DepartmentNumberVar>
```

Attribute	Type	M	N	Description
title	*STRING*	(+)	(+)	This nameless attribute is the title text.
fieldtitle	*ID*	(+)	(+)	This nameless attribute gets the built-in title of a field in the card part.
variabletitle	*ID*	(+)	(+)	This nameless attribute gets the built-in title of a variable in the card part.
text formatting				A number of text formatting attributes are available: ▪ `bold` ▪ `color` ▪ `fontname` ▪ `fontsize` ▪ `italic` ▪ justificationon ▪ `rgb` ▪ showasclosedfieldon ▪ `stretch` ▪ `underline`

The use of variable and field names in title tags does not influence the number of times that they can be used in the same card pane.

Short Form

As a short form for titles in card panes, use brackets ("`[`" and "`]`") to enclose the short form of the field or variable you want the default title for.

Thus:

```
<title fieldtitle=EmployeeNumber>
```

can be written as:

```
[.EmployeeNumber]
```

Structure

Tag	Structure	Comments
title	`=  <title` *attribute*$_1$ `..` *attribute*$_n$`>` `|  [.ID]:`*attribute*$_1$`: .. :` *attribute*$_n$	Long form. Short form for title associated with field. Short form for title associated with variable.

Tag	Structure	Comments
	| *[ID]:attribute$_1$: .. : attribute$_n$*	

image

The image tag is used to display pictures in the card pane. To show a picture, the picture file must have been imported into Maconomy using a document archive. In the document archive, documents are organized in document groups, so that all documents are identified by their name and document group. A reference to an image document in MDL takes the form:

DocumentGroup\DocumentName

where *DocumentGroup* is the name of a document archive, and *DocumentName* is a member of that archive. Maconomy currently supports these image formats:

- JPEG (Joint Photographic Experts Group). The usual file extensions are `.jpg` and `.jpeg`.
- PNG (Portable Network Graphics). The usual file extension is `.png`.

If you have imported a picture file of one of these formats into a document archive, you can show it in the card pane using the following expression:

```
<image title="MyImages\MyLogo.jpg" height=100 width=80>
```

or

```
<image fieldtitle=field1 height=100 width=80>
```

Attribute	Type	M	N	Description
title	*STRING*	(+)	+	The title specifies a direct reference to the image in the document archive, in the form *DocumentGroup\Document- Name* as described above.
fieldtitle	*ID*	(+)		This attribute specifies a database field which contains a direct reference to the image in the document archive, in the form *DocumentGroup\DocumentName* as described above.
variabletitle	*ID*	(+)		This attribute specifies a variable which contains a direct reference to the image in the document archive, in the form *DocumentGroup\DocumentName* as described above.
height	*REAL*	+		This attribute specifies the height in pixels of the frame for the current image. The manner in which the picture is shown within the frame can be adjusted using the `scale` attribute.
width	*REAL*	+		This attribute specifies the width in pixels of the frame for the current image. The manner in which the picture is shown within the frame can be adjusted using the `scale` attribute.

Attribute	Type	M	N	Description
scale	*ID*			This attribute specifies how the image is to be printed within the picture frame. It has the following possible values: ▪ `natural` (default): With this setting, the picture is maximized within the frame while keeping the natural image proportions. ▪ `fillbox`: With this setting, the picture is scaled to fill out the entire specified frame, irrespective of image proportions.

Exactly one of the `title`, `fieldtitle`, or `variabletitle` attributes must be present in the `<image>` tag.

By referring directly to an image using the `title` attribute, the same picture will be shown for all records. It is also possible to show data dependent images by referring to an image title from a field or a variable.

Example

```
<image variabletitle=var2 height=100 width=80 scale=fillbox>
```

Structure

Tag	Structure	Comments
image	= <image *attribute*$_1$ .. *attribute*$_n$>	

island

An island is a frame, placed around a box. The following expression draws an island with the title “Period.”

```
<island title="Period">
    ...
<end island>
```

Attribute	Type	M	N	Description
title	*STRING*	+	+	With this nameless attribute you specify the title of the island.
titlevisible	*BOOLEAN*			If this attribute is set and `true`, the title is suppressed in the current layout. This applies to the Java™ client only.

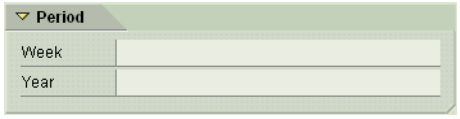
Attribute	Type	M	N	Description
text formatting				A number of text formatting attributes are available: ▪ bold ▪ color ▪ fontname ▪ fontsize ▪ italic ▪ rgb ▪ underline

Example

The layout for the Period island in the Time Sheets dialog can, for instance, be made by using the following expression:

```
<island "Period">
   <array>
      <row>
         <text title="Week">
         <var data=WeekVar>
      <end row>
      <row>
         <text title="Year">
         <var data=YearVar>
      <end row>
   <end array>
<end island>
```

This creates the following island:



You can find more examples of islands in “Floating.”

Structure

Tag	Structure	Comments
island	= <island *attribute*$_1$ .. *attribute*$_n$> *box* <end island>	

array

An array is used for placing elements in rows and columns. An array consists of one or more rows that are enclosed by an `array` tag and an `end array` end tag:

```
<array>
    rows
<end array>
```

Each row in an array is placed above each other and is left-aligned. Note that the boxes in the array are aligned. If a text is right-aligned in its box, the left side of the text will not be aligned.

Arrays do not take attributes.

Example

See the example in `island`.

Short Form

An array is abbreviated by placing rows in brackets. Hence,

```
<array>
    rows
<end array>
```

can be written as

```
{ rows  }
```

The content of the Period island in the Time Sheets window can be written like this (the long form of the same array was shown previously):

```
{
    "Week"     WeekVar
    "Year"     YearVar;
}
```

Structure

Tag	Structure	Comments
array	= <array> row_1 .. row_n <end array> | { row_1 .. row_n }	*array* must contain at least one row. Short form.

row

A row consists of one or more boxes (typically texts, database fields, and variables, but the boxes can also be islands or arrays).

```
<row>
    boxes
```

```
<end row>
```

Each box in a row is placed on the same line, meaning that the upper edges of the boxes are aligned.

If an array consists of more than one row, the first element of each row will be placed above each other (left-aligned), the second element of each row will be placed above each other, and so on.

Each row in an array must contain exactly the same number of elements. Arrays are also used to align boxes horizontally and vertically.

Rows do not take attributes.

Example

See the example in the section `island`.

Structure

Tag	Structure	Comments

row	`= <row>`
	`bo2.0 .. box`$_n$
	`<end row>`
	`|` *`bo2.0 .. box`*$_n$`;`
		row must contain at least one box.
		Short form.

Text Formatting Attributes

A group of text formatting attributes is used with a number of different tags. Instead of describing every text formatting attribute each time it can be used with the tags described previously, the descriptions are collected here, and references are made from the earlier description.

Maconomy usually chooses a reasonable text formatting (both text fields and the content of Maconomy fields), but in certain cases it can be convenient to deviate from the standard. To do this, you can choose from the following text formatting attributes. For each attribute, a list of applicable tags is given.

bold

The `bold` attribute is used for specifying that text (including the text in database fields, variables, and island titles) is to be bold. The attribute is specified as follows:

```
bold=true
```

```
bold=false
```

A `bold` attribute on a column specifies that both heading and content are to be written in **boldface**.

Examples

```
<text title="Month" bold=true>
```

```
<island title="Period" bold->
```

Applies to

```
<text>
```

<field>

<var>

<island>

<title>

color

The color attribute is used to specify text color (including the text in database fields, variables, and island titles). The attribute is specified as follows:

color=red

Possible values are:

black

yellow

magenta

red

cyan

green

blue

white

When a color for a column in a table pane is specified, both the data and the heading in the column are given the color in question. Note that colors do not apply in the Maconomy client for the Java™ platform.

See also the description of the rgb attribute.

Example

```
<island title="Employee" color=blue>
```

Applies to

<text>

<field>

<var>

<island>

<title>

fontsize

The fontsize attribute is used to specify the font size (including the size of text in database fields, variables, and island titles). The attribute is specified as follows:

fontsize=9

The font size is specified as an *INTEGER*. The default size is 9 (if nothing else is specified). The theoretical range is 0-32767. The specified font size influences both heading and column data.

Example

```
<field data=EmployeeNumber fontsize=12>
```

Applies to

`<text>`

`<field>`

`<var>`

`<island>`

`<title>`

italic

The `italic` attribute is used to specify that text (including the text in database fields, variables, and island titles) is to be in *italics*. The attribute is specified as follows:

```
italic=true
italic=false
```

The `italic` attribute on columns has an effect in both heading and content.

Example

```
<var data=MonthVar italic=true>
```

Applies to

`<text>`

`<field>`

`<var>`

`<island>`

`<title>`

justification

The `justification` attribute is nameless and specifies whether a text should be aligned left, right, or centered. Note that if a justification is not specified, the justification depends on the type of the field/variable (for instance, amounts are right- aligned), so in practice, it is seldom necessary to use this attribute.

You can select from three justification types:

```
justification=left
justification=right
justification=center
```

The `justification` attribute can be used on columns; if you do this, both the heading and the content are justified in the specified manner. In search windows, the field in the search line is also justified.

Example

```
<var data=MonthVar justification=right>
```

This right-aligns the content of the `MonthVar` field.

Applies to

<text>

<field>

<var>

<island>

<title>

rgb

The rgb attribute is nameless and makes it possible to use more color hues than when using the color attribute. An RGB color is specified as a triple (*R*,*G*,*B*), where *R*, *G,* and *B* are percentages between 0 and 100, specifying the intensity of red, green, and blue, respectively. The values can be specified with or without decimals (that is as *INTEGER* or *REAL*). Consider the following example that will result in a burgundy color:

```
rgb=(50,0,25)
```

Like color, rgb can be used on texts, database fields, variables, and islands (in islands the attribute only applies to the heading of the island). Note that colors do not apply in the Maconomy client for the Java™ platform.

Example

```
<island title="Employee" rgb=(30,100,50)>
```

Applies to

<text>

<field>

<var>

<island>

<title>

showasclosedfield

The design of the card part of some Maconomy dialog windows relies on the identical appearance of labels and closed fields. In the user interface of the Java client, labels and closed fields no longer have the same appearance. However, with this attribute and the showaslabelifclosed attribute you can explicitly change the appearance: labels may be changed to appear as closed fields, and closed fields may be specified to appear as labels.

The showasclosedfield attribute for the label tags text and title makes the card label appear as if it were a closed field.

This attribute only has effect in the Maconomy client for the Java platform. The Maconomy client for Windows ignores the attribute.

Example

```
<text "Employee No." showasclosedfield+>
```

Applies to

<text>

```
<title>
```

stretch

The `stretch` (extensibility) attribute is used to specify how extra space is to be divided between box elements that are placed next to each other. If a row of boxes (database fields, variables, or texts) has equal extensibility, the extra space is equally divided between the boxes; otherwise, the extra space is divided between the boxes with the highest extensibility.

Extensibility is specified as follows:

```
stretch=low
stretch=medium
stretch=high
```

If you do not use the `stretch` attribute, the text extensibility is `medium`, whereas the extensibility of variables and database fields is `high`. This means that in the typical scenario where database fields or variables follow fixed texts, the text gets its own size, and the field gets all of the extra space.

By specifying a row (fixed text followed by a variable), as in the following example, the variable field will have a width of 2 em, and any extra space will be used by the text (because it has a higher extensibility).

```
<row>
    <text title="Month">
    <var data=MonthVar stretch=low width=2.0>
<end row>
```

Applies to

```
<text>
<field>
<var>
<title>
```

underline

The `underline` attribute is used to specify that text (including the text in database fields, variables, and island titles) is to be underlined. The attribute is specified as follows:

```
underline=true
underline=false
```

The underline attribute on columns has effect in both heading and content.

Example

```
<field data=EmployeeNumber underline=true>
```

Applies to

```
<text>
<field>
<var>
```

<island>

<title>

width

The `width` attribute is used to specify the width of a text element or an image. The width is measured in *em*, except for images, which are specified as pixels.

```
width=10.0
```

The width of a box will never be smaller than its content. Hence, `"abc":width=0` will be at least as wide as `"abc."` See also “Very Short Texts.”

Example

```
<field title="Job No." data=JobNumber width=10>
```

Applies to

<text>

<field>

<var>

<image>

<island>

<title>

Working with Layouts

Floating

Groups

A group does not correspond to a window element, but is only used to control the formatting of panes with two columns (see the previous section concerning double-column panes).

If a pane with two columns is read from left to right, the pane elements (typically islands) appear exactly in the order in which they appear in the MDL layout. More explicitly:

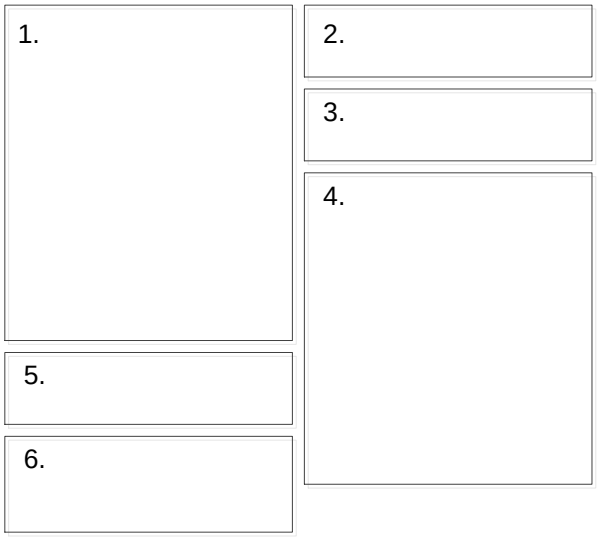
1. The first element is placed to the left.
2. Subsequent elements are placed below the lowest column if the width is large enough; otherwise they are placed to the left under both columns.

This process is called *floating*. Consider the following example:

```
<dualpane>
   <group>
      <island title="1.">
         ...
      <end island>
      <island title="2.">
         ...
      <end island>
```

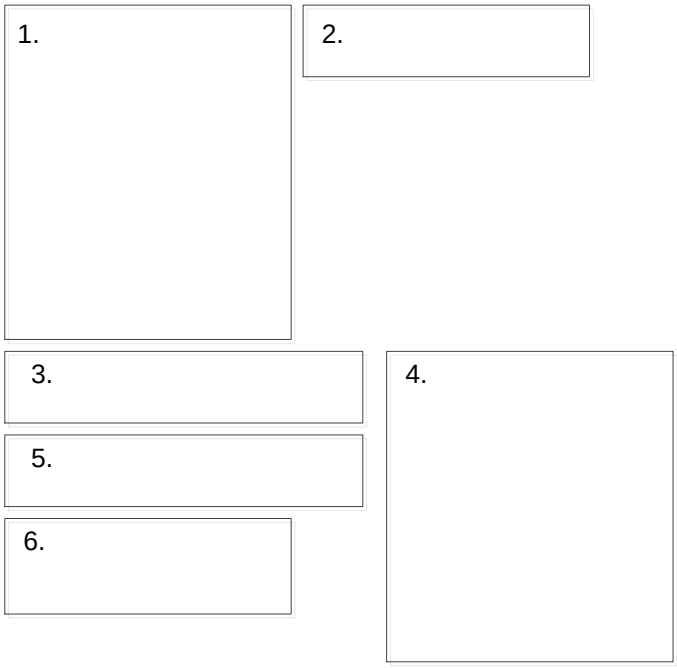
```
        <island title="3.">
            ...
        <end island>
        <island title="4.">
            ...
        <end island>
        <island title="5.">
            ...
        <end island>
        <island title="6.">
            ...
        <end island>
    <end group>
<end dualpane>
```

The result of the preceding example is the following layout (provided that all islands are narrower than half of the window):



This is the result of the preceding code example. All islands fit in two columns.

If, instead, it is assumed that the first island is exactly half the size of the window width, but that the third island is larger and the fourth smaller, you would get the following layout:



This is the result of the preceding code example. Some islands are wider than half of the window.

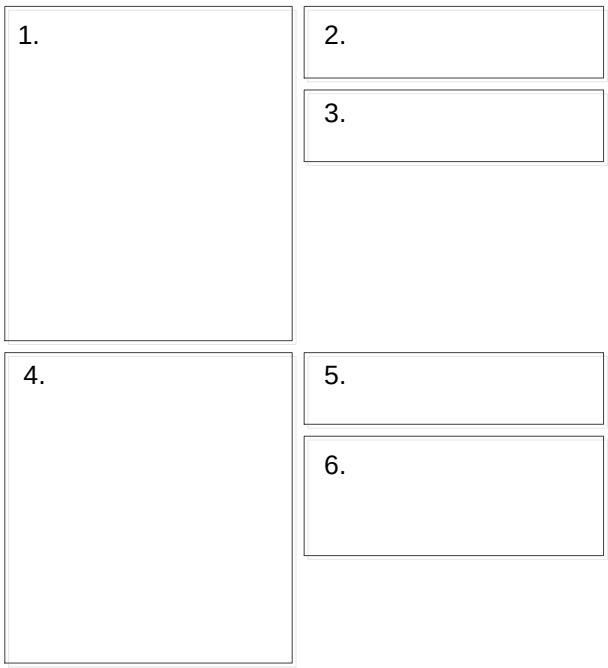
If you want to control the exact layout yourself, you can use arrays. However, you might often let Maconomy handle the precise formatting, and only indicate that certain elements are to be grouped.

If, for instance, islands 1, 2, and 3 are logically connected, and 4, 5, and 6 are connected, you could write:

```
<dualpane>
    <group>
       <island title="1.">
          ...
       <end island>
       <island title="2.">
          ...
       <end island>
       <island title="3.">
          ...
       <end island>
       <island title="4.">
          ...
       <end island>
       <island title="5.">
          ...
       <end island>
       <island title="6.">
```

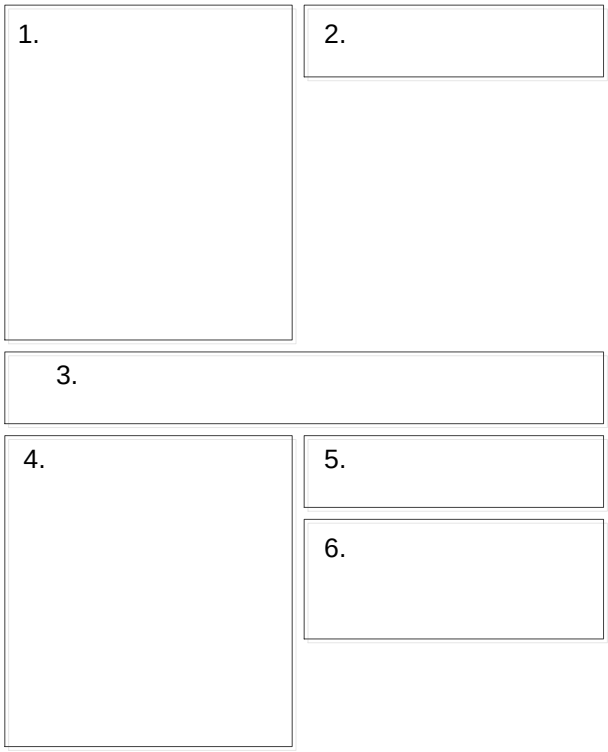
```
            ...
        <end island>
    <end group>
<end dualpane>
```

This yields (with the original assumption that all islands are narrower than half of the window) the following layout:



This is the result of the preceding code example. Islands are split into two groups.

If the third island is placed in a group of its own, the result is as follows:



This is the result of the preceding code example. Islands are split into three groups.

The preceding result is due to the fact that the width of an island that is placed in a group of its own is set to the total width of the window.

Modes

Using the `mandatory`, `ronew`, `roupdate` and `readonly` attributes on the `field` tag, you can specify whether it should be possible to edit fields or not. You can deny access to open fields (but not grant access to closed fields), and you can require that information is entered in certain fields.

Each pane in a dialog window is always in one of the following two modes: *New* mode or *Updating* mode.

Example: If you have opened the Exchange Rate Table window and you select “New Exchange Rate Table” in the Index or File menu, the card pane will be ready for entry of a new exchange rate table. When you subsequently enter the name of an exchange rate table and press Enter, the card pane will change to *Updating* mode, where it is possible to update (change) fields.

In the same way, a row in the table part is said to be in the *New* mode when it is being created (for instance by clicking on a new row). Once you have entered data and pressed Enter, the table pane is also put into *Updating* mode.

If the user cannot enter data into a field, the field is *closed*. A field can be open in *New* mode and closed in *Updating* mode (or another combination of open and closed). Fields in the search window cannot be closed. When a field is *mandatory* (required), you cannot leave the window before the field has been completed. Therefore, a field cannot be mandatory if it is closed in both New and Updating mode.

In the description of the `field` tag, it was noted that a variable or a database field can only be used several times in the same card pane if at most one of the instances of the field or variable is open in each of the two states. This means that if you want a field to be included several times in a card pane, you must specify for all of the instances of the field (or all of them except one) that it is

closed in *New* mode. Similarly, you must specify that all instances of the field (or all of them except one) are closed in *Updating* mode. Please note, however, that it is possible to have one instance of a field open in *New* mode and another instance open in *Updating* mode.

Layout Examples

This section contains a number of MDL layout examples. In the first part, it is illustrated that most layouts are quite simple to make. The second part explains and illustrates how some special layout types are handled.

When constructing layouts, it is convenient to use the standard layout as the basis for the dialog.

In the examples, the short form is generally used.

One Card Pane

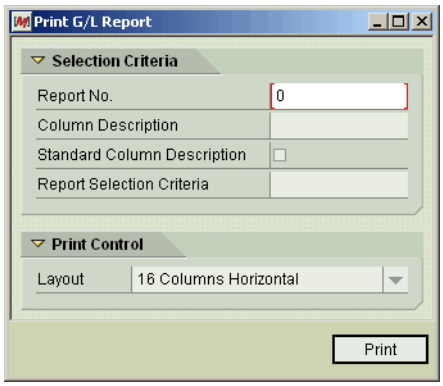
In this section, we construct a simple layout for the Print G/L Report parameter dialog. The dialog consists of one card pane. The dialog contains five variables: `ReportNumberVar`, `ColumnDescriptionNameVar`, `StandardColumnDescriptionVar`, `TargetGroupNameVar` and `Layoutname`. In `ReportNumberVar`, `ColumnDescriptionNameVar`, `StandardColumnDescriptionVar`, and `TargetGroupNameVar` you enter information about the content of the G/L report. It is therefore natural to place these four variables in a separate island called Selection Criteria. To the left of each variable, you then place a fixed text that describes the variable.

Each island contains an array, in which each row consists of a `text` tag (the fixed text), and a `var` tag. These considerations result in the following layout:

```
<mdl 4>
<layout Example "Example" windowtype=DialogWindow windowname=ExchangeRateTables>
    <singlepane>
        <island "Selection Criteria">
            <array>
                <row>
                    "Report No."     ReportNumberVar
                <end row>
                <row>
                    "Standard Column Description" StandardColumnDescriptionVar
                <end row>
                <row>
                    "Report Selection Criteria" TargetGroupNameVar
                <end row>
            <end array>
        <end island>
        <island "Print Control">
            <array>
                <row>
                    "Layout"  LayoutName
                <end row>
```

```
          <end array>
       <end island>
   <end singlepane>
<end layout>
```

This layout results in the following window:

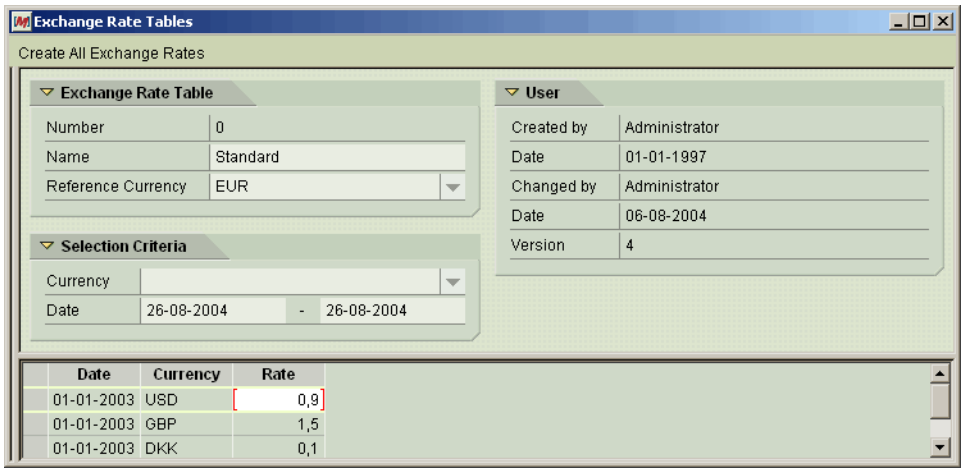


This is the result of the preceding code example: Print G/L Report.

Making Layout Changes

Suppose that you want to change the Exchange Rate Tables window. The original layout of the window looks as follows:

However, you want to change the layout to have the window look as follows:



This section gives a step-by-step description of how this layout is constructed. The layout contains two panes: A card pane and a table pane. If the layout is called *Example*, the `layout` tag is specified as follows:

```
<mdl 4>
<layout Example "Example" windowtype=DialogWindow windowname=ExchangeRateTables>
    ...
<end layout>
```

The Card Part

Because two islands are placed next to each other, the card part is a dual pane—meaning that the expression looks as follows:

```
<dualpane>
    ...
<end dualpane>
```

Islands and Floating

The islands are placed in such a way that the order corresponds to the order of the layout when the islands are read in the usual reading order (left to right, top to bottom). This means that the islands are to be written in the following order: Exchange Rate Table, User, Selection Criteria. Maconomy automatically places the islands in such a way that only one group is needed.

```
<group>
    <island "Exchange Rate Table">
        ...
    <end island>
    <island "User">
        ...
    <end island>
    <island "Selection Criteria">
        ...
    <end island>
<end group>
```

Elements

The Exchange Rate Table island contains six elements (three texts and three fields) that must be placed next to each other. The elements are placed as an array containing three rows:

```
{
    "Number"   .ExchangeRateTableNumber;
    "Name"     .Name;
    "Reference Currency"    .ReferenceCurrency;
}
```

The User island is also placed in an array, this time containing five rows, all having two elements:

```
{
    "Created by"     .CreatedBy;
    "Date"     .CreatedDate;
    "Changed by"     .ChangedBy;
    "Date"     .ChangedDate;
    "Version" .VersionNumber;
}
```

This type of island is very typical: Many islands can be described by means of a two-column array: fixed texts followed by database fields or variables.

The last island, Selection Criteria, is different, because the second row contains four elements: A fixed text, a database field, a hyphen, and a second database field. This can be done by letting the island contain two arrays: The first with two elements in one row, and the second with four elements in one row. However, this would not be a good idea because of the following reasons:

1. You could not be certain that the left side of the `CurrencyVar` pop-up field would align perfectly with the left side of the `FromDateVar` field.
2. The Selection Criteria island would be wider than the User island, because the Selection Criteria island would be perceived as containing several fields.

The following solution is more usable: The Selection Criteria island contains one array that has two columns and two rows; the second element in the second row is an array itself, containing three elements in one row. The result is as follows:

```
{
    "Currency"CurrencyVar;
    "Date"     {FromDateVar '-' ToDateVar;};
}
```

Such arrays-in-arrays are called *embedded* arrays. They are frequently used for advanced element formatting (more examples of embedded arrays later in this section).

This island also illustrates the use of short texts: The use of the apostrophes in `'-'` ensures that no space is placed around the hyphen.

The Table Part

The table part is constructed in a very simple way:

```
<tablepane>
    "Date"     .StartingDate;
```

```
    "Currency".Currency;
    "Rate"    .ExchangeRate;
<end tablepane>
```

Each line is a heading that is followed by a database field name.

The Entire Layout

The entire layout is now as follows:

```
<mdl 4>
<layout Example "Example" windowtype=DialogWindow windowname=ExchangeRateTables>
    <dualpane>
        <group>
            <island "Exchange Rate Table">
                {
                    "Number"  .ExchangeRateTableNumber;
                    "Name"    .Name;
                    "Reference Currency"    .ReferenceCurrency;
                }
            <end island>
            <island "User">
                {
                    "Created by"     .CreatedBy;
                    "Date"    .CreatedDate;
                    "Changed by"     .ChangedBy;
                    "Date"    .ChangedDate;
                    "Version" .VersionNumber;
                }
            <end island>
            <island "Selection Criteria">
                {
                    "Currency"CurrencyVar;
                    "Date"    {FromDateVar '-' ToDateVar;};
                }
            <end island>
        <end group>
    <end dualpane>
    <tablepane>
        "Date" .StartingDate;
        "Currency"    .Currency;
        "Rate" .ExchangeRate;
    <end tablepane>
```

```
<end layout>
```

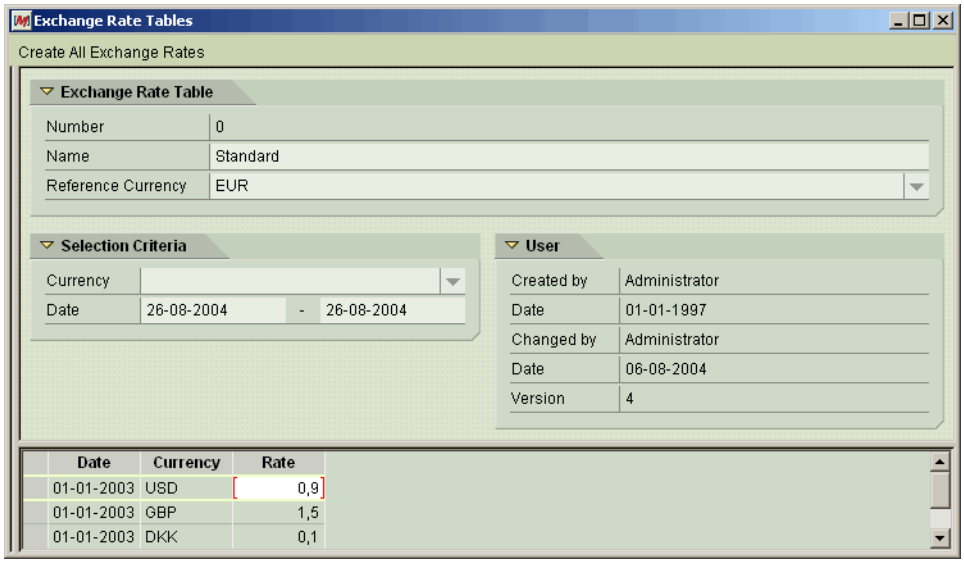
Use of Groups

If you place all islands in a double-column pane in one group, they will all be placed in two columns. You might not always want this, for example, if you want more space in one island than in others. Assume that you want to make more space for the exchange rate table name in the layout from the previous section. In this case, you can use the following expression:

```
<mdl 4>
<layout Example "Example" windowtype=DialogWindow
windowname=ExchangeRateTables>
    <dualpane>
        <group>
            <island "Exchange Rate Table">
                {
                    "Number"  .ExchangeRateTableNumber;
                    "Name"    .Name;
                    "Reference Currency"    .ReferenceCurrency;
                }
            <end island>
        <end group>
        <group>
            <island "Selection Criteria">
                {
                    "Currency"CurrencyVar;
                    "Date"    {FromDateVar '-' ToDateVar;};
                }
            <end island>
            <island "User">
                {
                    "Created by"     .CreatedBy;
                    "Date"    .CreatedDate;
                    "Changed by"     .ChangedBy;
                    "Date"    .ChangedDate;
                    "Version" .VersionNumber;
                }
            <end island>
        <end group>
    <end dualpane>
    <tablepane>
        "Date" .StartingDate;
        "Currency"    .Currency;
```

```
        "Rate" .ExchangeRate;
    <end tablepane>
<end layout>
```

This results in the following window layout:



If you place the Exchange Rate Table island in a group of its own, the layout allows the island to stretch across the entire window width. In addition, the Selection Criteria island is written before the User island to place the islands in the right order.

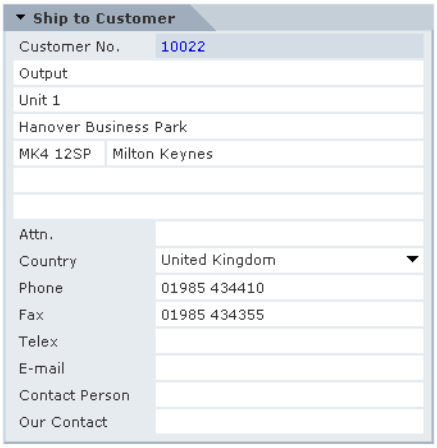
Using Absolute Lengths

The template for layout design that was described in the two previous sections is usually sufficient for you to express the window designs that you want. Groups are used for island placing, whereas arrays (occasionally embedded) are used for specific element placing.

It is usually sufficient to use these simple elements when constructing a layout, because it ensures homogeneous and easily read windows. In certain situations, however, you may not be able to do a satisfactory placing by means of these elements—and it might be necessary for you to make use of the `width` attributes.

Zip Code

Addresses are frequently used in Maconomy, usually with the following appearance:



What is unusual about the preceding placing is the fact that the fifth line contains two fields, and that these two fields are not of equal length. This can be done in MDL in the following way:

```
<island "Ship to Customer">
    {
        {
            "Customer No."   .CustomerNumber;
        };
        {
            .Name1;
            .Name2;
            .Name3;
        { .ZipCode:7.0:stretch=medium .PostalDistrict; };
            .Name4;
            .Name5;
        };
        {
            "Attn."    .Attention;
            "Country"  .Country;
            "Phone"    .Telephone;
            "Fax"      .Telefax;
            "Telex"    .Telex;
            "E-mail"   .ElectronicMailAddress;
            "Contact Person" .ContactPerson;
            "Our Contact"    .OurContact;
        };
    }
<end island>
```

To ensure that the field is not stretched more than the stated width of seven characters (remember that fields usually have high extensibility), set `stretch` to medium on the `ZipCode` database field.

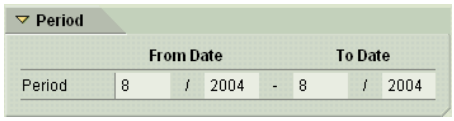
Note that in MDL it is not possible to put the `CustomerNumber` and `Attention` fields in the same column, but the MDL minimum width for texts ensures that they are still placed under each other. If

the `"Our Contact"` text had been much longer, this could only have been done by defining the widths of the text. However, you will often get a nicer layout if you do not.

One Heading above Several Array Columns

Many parameter windows have period specifications that correspond to the following example:

In this example, you might want the island to show headings above the dates, as follows:



This can be done by writing the following island specification:

```
<island "Period">
    {
        ''     "From Date":bold+:center    '' "To Date":bold+:center;
        "Period"      {MonthFrom:3.0       '/'    YearFrom:3.0;}       '-'
        {MonthTo:3.0  '/'    YearTo:3.0;};
    }
<end island>
```

As in the case of the zip code, you here let `MonthFrom`, `'/'`, and `YearFrom` constitute one array, which is embedded in another array (similar for `MonthTo`, `'/'`, and `YearTo`). Note how the first row uses empty strings in the first and the third columns to make sure that the correct number of elements is specified. The apostrophes are specified to ensure that the empty texts do not have an influence on the column widths.

> **Note:** In this example, using “” would result in the same layout.

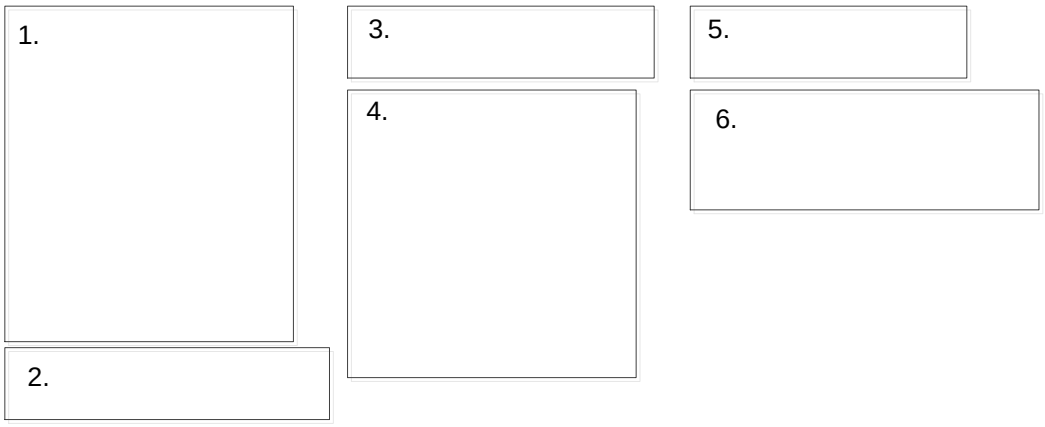
Islands in More than Two Columns

If you want to create a layout that has islands in more than two columns, use arrays. For example, perhaps you want three columns that have two islands in each column. To accomplish this, create a layout like this:

```
<singlepane>
    { --Array containing a row of 3 columns
        { --First column
            <island title="1.">
                ...
            <end island>;
            <island title="2.">
                ...
            <end island>;
        }
```

```
        { --Second column
            <island title="3.">
                ...
            <end island>;
            <island title="4.">
                ...
            <end island>;
        }
        { --Third column
            <island title="5.">
                ...
            <end island>;
            <island title="6.">
                ...
            <end island>;
        }; --This semicolon ends the row of three columns
    }
<end singlepane>
```

The result of this MDL code would be something like the following:



This is the result of the preceding code example. Islands are arranged in one row and three columns, with two islands each.

MDL Syntax

This section describes the formal MDL syntax. A formal syntax gives a brief and precise description that can be used as a definition of MDL and as a reference when you already know the language.

The syntax is described in a variant of the Backus-Naur Form. This notation is a precise method for describing valid language constructs.

Terminal symbols (that is, symbols that should be written in MDL as they appear in the BNF table) are written in `monospace font`, non-terminal symbols (symbols that are defined by the grammar) as *italic lower case*, and grammar primitives (similar to non-terminal symbols, but typically more

fundamental and defined less formally) in *ITALIC CAPITALS*. For a description of attribute types, see “Attribute Types.” For a description of comments and whitespace, see “Comments and Whitespace.”

Alternatives are specified by writing | between the options. If symbols can appear more than once, this is specified by means of . . . If a tag can have several attributes, it is written like this: *attribute*$_1$ . . *attribute*$_n$.

Syntax

The MDL syntax can be described by means of the following grammar stated. The start symbol of the grammar is *“start.”*

Symbol	Grammar	Comment
start	= <mdl *INTEGER*> *layout*	Version no. 1-4.
layout	= <layout *attribute+*> *availableactions?* *availableshortcuts?* *pane+* <end layout>	*availableactions* and *availableshortcuts* can be omitted.
	| <layout *attribute+*> *availableactions?* *availableshortcuts?* *pane*$_1$ *pane*$_2$ <end layout>	*pane*$_1$ must be *singlepane* or *dualpane*; *pane*$_2$ must be *tablepane*.
availableactions	= <availableactions?> *action** <end availableactions>	*availableactions* need not contain any actions.
action	= <action *attribute*+>	
availableshortcuts	= <availableshortcuts> *shortcut** <end availableshortcuts>	*availableshortcuts* need not contain any shortcuts.
shortcut	= <shortcut *attribute+*>	
pane	= *singlepane* | *dualpane* | *tablepane*	

Symbol	Grammar	Comment
singlepane	= <singlepane> *buttons?* *dropdownbutton?* *box** <end singlepane>	*singlepane* need not contain any boxes , buttons, or dropdownbutton.
dualpane	= <dualpane attribute?> *buttons?* *dropdownbutton?* *group** <end dualpane>	*dualpane* need not contain any groups, buttons, or dropdownbutton.
tablepane	= <tablepane attribute> *buttons?* *dropdownbutton?* *column+* <end tablepane>	*tablepane* must contain at least one column. Buttons and dropdownbutton can be omitted.
buttons	= <buttons> *action** <end buttons>	*buttons* need not contain any actions.
dropdownbutton	= <dropdownbutton> *action** <end dropdownbutton>	*dropdownbutton* need not contain any actions.
group	= <group> *box+* <end group>	*group* must contain at least one box.
box	= *text* | *field* | *variable* | *title* | *image* | *island* | *array*	
text	= <text attribute+> | *STRING:attribute** | *FSTRING:attribute**	Double-quote text. Single-quote text.

Symbol	Grammar	Comment
field	= <field *attribute+*> | *.ID:attribute**	Short form.
variable	= <var *attribute+*> | *ID:attribute**	Short form.
title	= <title *attribute+*> | [*.ID*]:*attribute** | [*ID*]:*attribute**	Short form for title associated with field. Short form for title associated with variable.
image	= <image *attribute+*>	
island	= <island *attribute+*> *box* <end island>	
array	= <array> *row+* <end array> | { *row+* }	*array* must contain at least one row.
row	= <row> *box+* <end row> | *box+;*	*row* must contain at least one box.
column	= <field *attribute+*>	Field column, long form.
	| *STRING .ID:attribute*;*	Field column with text heading.
	| *ID .ID:attribute*;*	Field column with heading derived from variable.
	| *.ID .ID:attribute*;*	Field column with heading derived from field.
	| <var *attribute+*>	Variable column long form.
	| *STRING ID:attribute*;*	Variable column with text heading.
	| *ID ID:attribute*;*	Variable column with heading derived from variable.

Symbol	Grammar	Comment
attribute	= *ID = attributevalue*	
	| *ID+*	Short form for Booleans. Note: This is a literal +.
	| *ID-*	Short form for Booleans.
	| *attributevalue*	Nameless.
attributevalue	= *BOOLEAN* | *INTEGER* | *REAL* | *STRING* | *ID* | (*INTEGER*,*INTEGER*) | (*REAL*,*REAL*,*REAL*)	

Error Messages

This section contains a list of all possible error messages from the MDL compiler. Error messages are saved to the "LAYERR.TXT" file in the Maconomy folder. If you are using the Maconomy client for the Java™ platform, the errors are displayed in a new browser window. If no errors occur when importing a layout, no file is generated. An explanation of the individual error messages is given below each message, including any problem-solving suggestions. Note that many error messages are prefixed by "Error in line *nn*:," but this prefix is left out in the following.

Syntax Errors

If an MDL layout does not follow the syntax that is described in the previous section, the following error message is given by Maconomy:

- Error in line *nn*: Syntax error.

  where *nn* specifies the line number in which the error occurs.

  You will often see the error by looking at the line identified by the interpreter, but sometimes the selected line will be correct. The most frequent cause of such an error[2] is that parenthetical tags have not been *balanced*, meaning that a tag has not been closed (or has been closed by another end tag) or that an opening tag is missing in an end tag. Such errors can be difficult to locate, but the process is made less difficult if the content of the tag/end tag is indented as in the examples shown in this manual. Consistent indentation makes the structure much easier to read.

- The layout title cannot be empty.

[2] Particularly if the MDL interpreter reports the error on the first or the last layout line.

This applies to MDL versions below 4. It is not allowed to specify an empty layout title. The layout contains: "`<layout "">,`‘ which should be changed to ‘`<layout "layout-title">.`‘

- The layout name cannot be empty.

  This applies to MDL version 4 and above. It is not allowed to specify an empty layout name. The layout contains: ‘`<layout "">,`‘ which should be changed to ‘`<layout "layout-name">.`‘

- The layout title cannot begin or end with blanks.

  It is not allowed to specify blank characters at the beginning and at the end of the layout title in the `layout` tag.

- The layout name cannot begin or end with blanks.

  It is not allowed to specify blank characters at the beginning and at the end of the layout name in the `layout` tag.

- The layout title cannot contain '_' characters.

  It is not allowed to specify the underscore character in the layout title in the `layout`tag.

- The layout name cannot contain '_' characters.

  This applies to MDL version 4 and above. It is not allowed to specify the underscore character in the layout name in the `layout` tag.

- Name is too long. Shortened to *nn*characters.

  MDL does not allow names with more than 100 characters. A name can be a database name, a variable name, an attribute name or an attribute value of the *ID* type.

- Text is too long. Shortened to *nn*characters.

  MDL does not allow texts—meaning strings that are enclosed by single quotes (`'...'`) or double quotes (`"..."`)—to consist of more than 100 characters. This will, however, seldom be a problem, because texts cannot contain line feed characters.

- Incomplete text.

  A string is not ended, meaning that a string begins with `"` but is not ended by `"`. Maconomy will try to read on, but is of course not able to find out where the string should have been ended. This can cause some rather odd errors in the later layout. You should just enter the end quote and try to import the layout again.

- Error in integer ’*ss*’.

  ss is an integer, which is too large. Integers cannot be larger than 21474836467.

- A real number must have between 1 and 3 digits after the decimal point.

  A decimal figure that has more than three digits has been specified. Note that leaving out the digit after the decimal point will result in a syntax error (and it will be reported as such), if no digits are specified (as in `‘32.’`).

- Error in real ’*ss*’.

  ss is a floating point number that is much too large (positive or negative). The range of permitted numbers is from 1.7E308 to -1.7E308.

Semantic Errors

Maconomy gives detailed error messages on semantic errors (errors that are related to matters of meaning). The following sections list all of the semantic error messages and their meanings.

Mandatory Fields

If a database field or a variable is defined by Maconomy as being mandatory in a dialog, it must be included in the layout.

In the following, only database field errors are mentioned, but the same errors can occur for variables.

- Error: The mandatory database field '*ss*' must be used in the upper pane.

  A mandatory database field with the name *ss* is omitted from the upper pane.
- Error: The mandatory database field '*ss*' must be used in the lower pane

  A mandatory database field with the name *ss* has been omitted from the lower pane.
- The database field '*ss*' is a check box and cannot be made mandatory.

  Checkboxes cannot be made mandatory.
- The database field '*ss*' is a popup and cannot be made mandatory.

  Pop-up fields cannot be made mandatory.
- The database field '*ss*' cannot be mandatory as well as closed.

  A database field cannot be totally closed as well as mandatory, as it will prevent the user from entering data.
- The closed database field '*ss*' cannot be made mandatory.

  A mandatory database field cannot be totally closed, as it will prevent the user from entering data. The database field was closed by Maconomy.
- The database field '*ss*' was already closed in one state. Hence, closing the field in the other state and making it mandatory is illegal.

  The attribute `mandatory` is used with either `ronew` or `roupdate`. As the database field was already closed in the other state by Maconomy, the field is totally closed and mandatory, which is an error.
- Error: The database field '*ss*' in upper pane was open in the new state and mandatory. One instance of the database field must be open in the new state.

  The database field '*ss*' is mandatory and open in *new* mode by system default. There must therefore be one occurrence of '*ss*' in the upper pane which is open in this mode.
- Error: The database field '*ss*' in lower pane was open in the new state and mandatory. One instance of the database field must be open in the new state.

  The database field '*ss*' is mandatory and open in *new* mode by system default. There must therefore be one occurrence of '*ss*' in the upper pane which is open in this mode.

Parameter Windows and Search Windows

In certain areas, parameter windows and search windows differ from the other dialogs.

- Fields or variables cannot be used as column headers in search windows.

The attribute `fieldtitle` or `variabletitle` was used in a layout description that pertains to a search window. These attributes are only allowed in layout descriptions for card/table windows.

- Only variable fields are allowed in parameter dialogs.

  A database field exists in a parameter dialog. Parameter windows can only contain variables.

- Only database fields are allowed in search windows.

  A variable exists in a search window. Search windows can only contain database fields.

Panes

Using other pane types than those defined by Maconomy will result in an error.

- A card dialog must contain one singlepane or one dualpane.

  Only card panes (not table panes) can be specified in a card dialog.

- There can be only one pane in a card dialog.

  A card dialog must contain exactly one pane.

- A table dialog must contain one tablepane.

  A table dialog can only contain one table pane (single panes and dual panes are not allowed).

- There must be exactly one pane in the layout.

  The current layout can only contain one pane.

- There can be only one pane in a table dialog.

  A table dialog can only contain one pane.

- Unexpected pane in card/card dialog.

  Both the upper and the lower pane in a card/card dialog must be card panes (meaning single panes or dual panes). From the line number, it is possible to determine which pane is actually causing the error.

- There are only two panes in a card/card dialog.

  The layout contains more than two panes.

- Unexpected pane in card/table dialog.

  The upper pane is not a card pane (meaning a single pane or a dual pane) or the lower pane is not a table pane. The line number indicates which pane is actually causing the error.

- There are only two panes in a card/table dialog.

  The layout contains more than two panes.

- A parameter dialog must contain one singlepane or one dualpane.

  A layout for a parameter dialog must contain a pane, which must be a single pane or a dual pane.

- There is only one pane in a parameter dialog.

  A layout for a parameter dialog cannot contain more than one pane.

- A search window must contain one tablepane.

A layout for a search window has to contain exactly one table pane.

- There is only one pane in a search window.

  A layout for a search window contains more than one pane.

- There must be exactly two panes in the layout.

  Only one pane is specified for a dialog that must contain two panes.

- Only parameter and card dialogs may contain empty singlepanes.

  A single pane can be empty if it is used as (the only) pane in a parameter dialog or a card dialog; otherwise, empty single panes are not allowed.

Rows

- The row has *xx* elements. *yy* was expected.

  The row consists of a wrong number of elements. The previous rows (or the previous row) all contained *yy* elements, but the current row contains *xx* (which is either larger or smaller than *yy*).

Database Fields and Variables

You can only specify database fields and variables that are connected to the pane in which they are specified. In addition, database fields and variables can only be specified more than once if they are always closed.

> **Note:** In the following, only database field errors are mentioned, but the same errors can occur for variables (with a few exceptions).

- The database field '*ss*' does not exist.

  This pane contains no database field with the name "*ss*."

- The database field '*ss*' is open more than once in the new or update state.

  Only one instance of a database field can be open in the new or update state.

- The database field '*ss*' is a *tt* and cannot use zero suppression.

  The `zerosuppression` attribute does not apply to database fields of a type different from *INTEGER*, *REAL*, or *AMOUNT*.

- The database field '*ss*' is mandatory and cannot use zero suppression.

  The attribute `zerosuppression` attribute does not apply to mandatory database fields.

- The database field '*ss*' is a *tt*and cannot be visualized as time.

  The `visualizeastime` attribute does not apply to database fields of a type different from *REAL*.

- The database field '*ss*' is mandatory and cannot be summed.

  The `summed` attribute does not apply to database fields of a type different from *INTEGER, REAL,* or *AMOUNT.*

- Repeated use of the database field '*ss*' is not allowed.

  Database fields only. The database field '*ss*' is used several times. This error message is only given for search windows, where the same database field cannot occur more than once.

- There is no relation in this pane, hence database field '*ss*' is not allowed.

  Database fields only. Certain panes have no related relations. In these cases, it is not possible to refer to database fields.

Attributes

When error messages that are related to attributes occur, it is usually convenient to consult the reference section for the current tag for legal attribute specifications.

- Only versions 1 to 4 of MDL are supported.

  The permitted values for the `version` attribute in the `mdl` tag is `1` through `4`.

- Only MDL version 2 and later supports fields or variables as column titles.

  The `fieldtitle` or `variabletitle` attribute was used on line *nn*, but MDL version 1 has been specified at the beginning of the layout description.

- Invalid attribute '*ss*'.

  The specified attribute is invalid for a tag in the layout.

- The attribute '*ss*' is invalid for the tag '*ss*'.

  The specified attribute is invalid for the tag mentioned in the message.

- Exactly one of attributes title, fieldtitle or variabletitle must be given.

  You have specified either none or several of the `title`, `fieldtitle` or `variabletitle` attributes in a tag of the type "`image`." You must specify one of these attributes.

- At most one of attributes title, fieldtitle or variabletitle can be given.

  You have specified more than one of the `title`, `fieldtitle` or `variabletitle` attributes in the tag. You must specify exactly one of these attributes.

- Exactly one of attributes fieldtitle or variabletitle must be given.

  You have specified either none or several of the `fieldtitle` or `variabletitle` attributes in a tag of the type "`title`." You must specify exactly one of these attributes.

- Some necessary attributes are missing for the tag '*ss*'.

  A necessary attribute was omitted in the "*ss*" tag. See the reference section for the current tag for legal and mandatory attributes. Note that the tag name '*ss*' does not necessarily occur in the layout. If short forms are used, the tag is omitted and derived from Maconomy. If the short form is specified in a wrong way, Maconomy might derive a wrong tag name, which might then be the cause of the error.

- The value '*ss*' cannot be used with the attribute '*tt*'.

  An attribute has been specified as `tt=ss`. *ss* has got the correct type, but it has a value that *tt* cannot accept, for example:

- stretch=unknownStretch

  where `unknownStretch` has the type *ID* (which is the correct type for values connected to the `stretch` attribute), but which is not a known value.

- Illegal window type '*ss*'.

  The window type specified in the `layout` tag is invalid. See the reference section for the "layout" tag for valid values.

- Invalid WindowName '*ss*', it must be '*ss*'.

When importing a layout, Maconomy checks that the window name in the layout matches that of the window into which the layout is imported.

- Invalid WindowType '*ss*', it must be '*ss*'.

  When importing a layout, Maconomy checks that the window type in the layout matches that of the window into which the layout is imported. See the reference section for the "layout" tag for valid values.

- The attribute '*ss*' is not defined for the tag '*tt*'.

  An attempt has been made to use the "*ss*" attribute with the "*tt*" tag. However, such a match is not possible. See the reference section for the current tag for legal attributes. Note that "*tt*" does not necessarily appear in the layout if short forms are used, and that a wrong short form can be the cause of the error.

- The attribute '*ss*' is already set for the present tag.

  The "*ss*" attribute has been used several times. In case of this error message, the name of the attribute will be stated explicitly or you will get the error message described below.

- The attribute '*ss*' of type '*ss*' is already set for the present tag.

  The "*ss*" attribute has been used several times. The attribute name "*ss*" is not explicitly stated, but is specified as nameless. As an example

  ```
  .EntityName:width=20.0:10.0
  ```

  will result in the error message

- Error in line nn: The attribute 'width' with type 'Real' is already set for the present tag.

  as all *REAL*s are regarded as nameless `width` attributes.

- The attribute '*ss*' has type '*tt*', but the given value has type '*uu*'.

  An attribute is specified as *ss*=*vv*. The "*vv*" value is a "*uu*" type, but the "*ss*" attribute must be given a value of the "*tt*" type. See the reference section for the current tag to see the type of the "*ss*" attribute.

- The Boolean attribute 'ss' does not exist.

  An attribute such as *ss*+ or *ss*- has been specified, but no attribute with the name "*ss*" exists.

- The attribute '*ss*' is not defined for the tag '*tt*'.

  An attempt was made to use the "*ss*" attribute in the "*tt*" tag, but it does not exist. See the reference section for the current tag for legal attributes.

- The attribute '*ss*' does not have type Boolean.

  The "*ss*" attribute is specified as *ss*+ or *ss*-, but "ss" does not have the *BOOLEAN* type. See the reference section for the current tag to see which type the "*ss*" should have been.

- No nameless attribute of type '*ss*' is defined for the tag '*tt*'.

  A nameless attribute of type "*ss*" has been specified for the "*tt*" tag. See the reference section for the current tag for nameless attributes that can be connected to each tag. In the table in "Attribute Types," the type of each attribute is specified.

- A value is missing after the attribute '*tt*'.

  A "*tt*" attribute is specified without a value. This error message is often caused by a *BOOLEAN* attribute specified without a "+" or "-."

- RGB values must be between 0 and 100.

Values for the "`rgb`" attribute are specified as a *TRIPLE* (*r, g, b*), where *r*, *g*, and *b* are numbers stating the intensity of red, green, and blue, respectively, as a percentage. This means that *r*, *g*, and *b* must be between 0 and 100 (both inclusive).

- Negative values cannot be used with attribute 'height'.

  `height` values cannot be negative.

- Negative values cannot be used with attribute 'width'.

  `width` values cannot be negative.

- Negative values cannot be used with attribute 'widthfactor'.

  `widthfactor` values cannot be negative.

- Value out of range for attribute 'fontsize'.

  Font sizes are specified in pixels and must be larger than 0 and smaller than or equal to 32767.

- The attributes "color" and "rgb" cannot be used simultaneously.

  The `color` and `rgb` attributes are both used for specifying color. It is not allowed to use both attributes in the same tag.

- The mandatory attribute 'title' is missing.

  Applies to the `layout` tag, MDL versions below 4. The `title` attribute is mandatory.

- The mandatory attribute 'title' is missing. The mandatory attribute 'name' is missing.—AND—The mandatory attribute 'windowname' is missing. The mandatory attribute 'windowtype' is missing.

  Applies to the `layout` tag, MDL version 4 and above. The attribute mentioned in the error message is mandatory.

- Action '*ss*' is missing.

  Applies to the `availableactions` or `shortcut` tag, MDL version 4 and above. A reference was made to an undefined action.

- The action '*ss*' is not allowed.

  An action was made available in a context where the action cannot be used.

MPL

Overview

This manual describes how printouts and Universe Reports are designed using the Maconomy Print Language (MPL). It covers all versions of MPL up to version 4.

MPL is a language used for defining both the contents (that it, data to be displayed) and the layout of printouts in Maconomy. It allows a developer to focus on specifying the logical structure of a printout, that is to say, how the elements on the printout should relate to each other, for example, that this column should stretch and those two elements should be aligned. This high-level description in MPL, called a layout, is then executed by the MPL engine, rendering an actual printout. The MPL engine makes sure that the physical layout of the rendered printout is best fitted to the content of the print.

This manual focuses on the language MPL. See the Set-Up section of the Maconomy reference manual for a description of the administration of MPL layouts after they have been created.

Prerequisites

This manual is both an introduction and a reference manual. For this reason, it sometimes seems very technical. The “Central Concepts” chapters give a good overview of the language. When reading the manual for the first time, it is recommended to browse through the technical sections quickly, as the most important concepts are later demonstrated through examples.

To be able to use this manual, you need basic knowledge of Maconomy. Furthermore, you need to be able to use an editor (such as TextPad, EditPad, or Notepad) and use the “Print Layout” windows in Maconomy as described in the Maconomy Reference Manual.

Version History

This section documents the history of changes to MPL. Because customers can use different versions of Maconomy, and hence, different MPL engine versions, this section gives a good overview of in which MPL version a particular feature has been introduced, as well as answer most backward-compatibility questions.

Changes in MPL Version 2

New Functionality

MPL 2 came out with the following new features:

- Print color (color attribute)
- Graphics/images in printouts (image tag)
- Move cursor to specific vertical position (goto tag)
- Style inheritance
- New attributes for blocks
- Extensions for Universe Reports

Changes in MPL Version 3

MPL version 3 is a major new version of MPL that breaks backward compatibility in some ways, but also provides a number of new features. The changes are described in details in “MPL Version 3,” while an overview is provided here.

MPL 2 and MPL 3 are currently both supported by the current Maconomy server, and it is up to the MPL layout writer to decide which MPL version to use. The Maconomy server, when compiling or executing an MPL layout, invokes either the old compiler and print engine (if the version tag states `<mpl 1>` or `<mpl 2>`), or the new compiler and print engine (if the version tag states `<mpl 3>`).

While MPL 3 is compatible with the great majority of existing MPL layouts, there are some things related to integration with existing client and server technologies to consider:

MPL 3 is only supported when using the Java client or the Portal. The MS Windows client does not support printing MPL 3 layouts (although it can be used to import them).

The MPL 3 print engine does not support adding RGL to the print through Active Scripting.

MPL 3 and MPL 2 cannot be mixed in situations where a number of layouts are used to generate one large print.

Also, you cannot mix MPL2 and MPL3 if there is a print layout selection rule that chooses between them. In practice that means that unless you always manually pick layouts when you print, either all layouts for a window must be MPL3, or none.

> **Note:** In the following, only database field errors are mentioned, but the same errors can occur for variables (with a few exceptions).

MPL 3 is intended to replace MPL 2 in the future, but until a new Maconomy server platform has been released, the two versions will coexist.

New Functionality

- These are the most import additions in MPL 3:
- Multiline text (`wrap` attribute on `<text>`, `<var>`, and `<data>` tags and the new `<concat>` tag)
- Ability to switch page orientation with `<newpage>`
- Conditionals can be negated and used with strings and database fields
- Supports PostScript (pfm, afm), OpenType (otf) and TrueType (ttf) fonts

Changed Functionality

The following list of changes in functionality is not a complete list, but it contains the most important ones:

- `<goto>` is no longer supported in headers or footers
- Conditionals no longer leave blank space when skipping content
- `<newpage>` in a row is no longer supported
- The scope of `<define>`, `<redefine>`, `<ruler>`, and `<subruler>` declarations and `<default>` has changed. They may now be specified anywhere in a parenthetical tag, not just at the beginning.

Changes in MPL Version 4 (as of TPU 16 SP0)

MPL 4 is a direct successor to and replacement of MPL 3 as of TPU 16 SP 0. Therefore, everything that applies to MPL 3 applies to MPL 4 as well, unless otherwise stated in this section.

This section provides an overview of the new features in MPL 4, as well as the changes that were introduced in this new version. For a detailed description of these matters, see “MPL Version 4.”

New Functionality

Version 4 allows for fetching custom data and performing custom calculations directly in an MPL layout. This new functionality is enabled through embedding in MPL 4 the *Expression Language* and *MQL*—two technologies that might already be familiar to a Maconomy consultant. The *Expression Language* is also used in other Maconomy layout languages like *MDML* and *MWSL*. *MQL*, on the other hand, is a statically typed database query language that is used for *Universe Reporting* and in *MScript* as well.

With these two new powerful tools at your disposal, you can now customize an MPL layout with any additional information that you might wish to include and that has not been included in the predefined print environment.

In particular, using the *MQL* support in MPL 4 you can now:

- Define reusable, parameterized database queries with the `<query>` tag. The queries are executed against the Maconomy universes, which feature database joins.
- Supply the query with the actual values of parameters it declares and instantiate it to a cursor by means of the `<cursor>` tag.
- Use the newly defined cursor as any other predefined cursor in a `<repeating>` tag.

Moreover, by using the Expression Language support in MPL 4, you can now:

- Perform arbitrarily complex calculations using expressions and standard functions.

- Bind the calculated values to mutable variables (<var>) and immutable constants (<val>).
- Assign new values to variables using the <assign> tag.
- Use expressions as conditions in the conditional tag as well as a path to the image in the <image> tag.

Changed Functionality

The following changes were introduced with respect to MPL 3:

- Field reference tag <field> has been desupported.
- Variable reference tag <var> has been desupported, and instead.
- The <var> tag means variable definition in MPL 4.
- When defining a tag, it is now disallowed to have white spaces in between the opening angle bracket "<" and the following tag name.
- Print structure check has been loosened up. For more information, see "Repeating Structure (MPL 4)."

Changes in MPL Version 4 (as of TPU 16 SP2)

The 2.1.1 release of Maconomy, which was delivered as TPU 16 SP2, came out with a couple of new features. Since this version of MPL is entirely backward-compatible with the previous version 4, the version number has not been changed.

New Functionality

As of this release, you can:

- Keep executing a block of MPL code while a certain condition is true by using the new <while> tag.
- Include static PDF documents in your MPL layouts by means of the new <includepdf> tag.
- Generate barcodes (14 different types) and QR codes using the new <barcode> tag.
- Manually control page numbering using the new <nextpagenumber> tag.
- Control whether headers and footers should be skipped when the enclosing <while>/<repeating> tag is empty (that is, no iteration took place) by using the new skipHeaderFooterIfEmpty attribute on the <while> and <repeating> tags.

Central Concepts

This section introduces the terminology and concepts necessary for understanding this manual and for using MPL. It starts off with an example of a simple layout and based on that, it explains the most fundamental concepts in MPL.

Example

MPL is a language for defining print layouts in Maconomy, usually for reporting purposes. When designing a print layout, you must address at least two kinds of concerns:

1. The *contents* of the print — Which data is this layout supposed to present?
2. The *layout* of the print (the presentation layer) — How to best present the data on this print so that it looks visually compelling?

This section examines a very simple layout that lists all of the employees in the Maconomy database, specifying their names, employee numbers, and the location of companies they work for. In other words, based on the data in the Maconomy system, we want to generate a simple table like the following.

NAME	EMPLOYEE N0	COMPANY BASED IN
Charlie Kaufman	01	New York
William Shakespeare	02	United Kingdom
Tigran Hamasyan	03	France
Roman Polanski	04	France
Pedro Almodóvar	05	Spain
Witold Gombrowicz	06	France
David Lynch	07	New York
Spike Jonze	08	New York

To generate such a printout using MPL, you define the following MPL layout:

```
1   <mpl 4>
2   <layout title="All Employees"
3           print="P Employee"
4           originallayout="Standard">
5   <page "A4">
6   <paper>
7     <ruler name=employeesRuler [[85pt]|[65pt]|[80pt]]>
8
9     <repeating cursor=Employee script="L_Employee">
10      <header atStart=true>
11        <array ruler=employeesRuler justification=center fontsize=10>
12          "NAME" "EMPLOYEE N0" "COMPANY BASED IN";
13          <hline>;
14        <end array>
15      <end header>
16
17      <array ruler=employeesRuler justification=right>
18        .Name1:justification=left Employee.EmployeeNumber CompanyNameVar;
19      <end array>
20    <end repeating>
22  <end paper>
```

We will now go through this layout step by step and explain the central concepts in it.

It is hard not to notice that MPL is a markup language—it consists of *tags* that can have *attributes*, very much like XML. Some tags like `paper` or `repeating` are *parenthetical tags*; they have an opening and closing tag, in between which reside their children tags. Other tags, like `ruler` or `page` are *simple tags*—they do not have any children, and therefore do not have a closing tag.

MPL Header

Lines 1–5 specify what is known as the *MPL header*. Apart from the MPL version number (line 1) and the paper format (line 5), we specify what the title of this layout is (line 2), which standard Maconomy layout this layout is based on (line 4), and what the *print environment* of this layout (line 3) is.

Print Environment

A *print environment* gathers all of the data that is available to be used in the layouts based on this environment. This data can be either in the form of database *cursors*, which are collections of database records, or single variables that store the results of calculations that are performed on the data stored in the Maconomy database. Every MPL layout must be based on a predefined print environment in Maconomy.

Accessing Data from the Print Environment

The data in the print environment comes in very handy—cursors and variables are initiated with the right values and are ready to be used in your layout.

Because a cursor is a collection of database records, the best way to access the data that a cursor holds is to iterate over these records one by one. In MPL this is achieved by using the `repeating` tag, like in line 9 of our example, where we iterate over all of the records in the `Employee` cursor. For each iteration of the cursor, we can reference fields in the current record by name prepended with a dot. For example, to reference the field `EmployeeNumber` in the cursor `Employee` (line 18), we can just say `Employee.EmployeeNumber`. We are not required to mention the cursor name, though, because it can be implicitly resolved by the MPL engine, so we can just as well say `.EmployeeNumber` or, as shown in the example, `.Name1`, to reference the field `Name1`.

To print out the value of a variable, similarly, you only have to reference its name (this time without a preceding dot); for example in line 18 we print out the value of the variable `CompanyNameVar`, which holds the first part of the company name that the employee works for—in this case it is the location in which the company is based.

You might be wondering why the variable `CompanyNameVar` updates its value for every iteration of the `Employee` cursor. It must be recalculated for each `Employee`. This is exactly what happens; to be more precise, all of these calculations and updates take place in *scripts* that are executed for every cursor iteration, before any of the `repeating`'s children is executed. In our example this task is carried out by the script `L_Employee` referenced in line 9.

Scripts

Scripts are typically used to update the values of variables that are available in the print environment. However, they can also have database side effects, for example when reprinting invoices the script `P_JobInvoice` is called, which results in incrementing the `VersionNumber` field on the `Invoice` relation in question.

Defining a Print Layout

When we know how to print out the data that we want to present in the layout, it is time to think of how this data should be presented. In other words, we want to define the layout of our print.

In our simple example, we want to display the employee name, number, and company in a table. The table should have a header, so that when a page break occurs, the header is also printed on the next page, followed by the continuation of the table content.

To define how many columns the table should have, as well as what shape these columns should be (for example, width, justification, stretching, and so on), we can use the `ruler` tag. In line 7 we define a ruler that is a column specification for *arrays* that are used in MPL to represent tables. Our ruler definition consists of three columns having the width of 85, 65, and 80 points respectively (1pt = 1 point = 1/72 inch). In between the first and the second column, as well as the second and the third, there is a vertical line, specified in the ruler as the "|" character.

Having defined the ruler, we can now use it in the `array` definition. Lines 11-14 define an array inside a header, which will be the header of our table. The `ruler` attribute of the array is set to the `employeesRuler` ruler that we have just defined. Similarly, the main array printing the employee data (lines 17 - 19) points to `employeesRuler` as well.

Inside an array we specify semicolon-separated rows, each of them complying with the ruler definition. Since our `employeesRuler` specifies three columns, the rows inside the array should have three elements in each row.

Short Versions for Tags and Attributes

Some tags and attributes are used quite often. To avoid unnecessary typing, these tags and attributes can have short forms. For example, instead of enclosing an array definition in between the opening `<array ruler=employeesRuler ..>` and closing `<end array>` tags, we can also use the short form of the array tag and just type `{:employeesRuler..}`. Similarly, sometimes we can skip an attribute name and just specify its value, for example, instead of writing `<repeating cursor=Employee>`, we can just write `<repeating Employee>`. The "MPL Language Basics" section explains the short versions of attributes and tags in more detail. Our example would look somewhat like this when written using short forms for tags and attributes:

```
1   <mpl 4>
2   <layout title="All Employees"
3           print="P Employee"
4           originallayout="Standard">
5   <page "A4">
6   <paper>
7     <ruler employeesRuler [[85pt]|[65pt]|[80pt]]>
8
9     <repeating Employee script="L_Employee">
10      <header atstart+>
11        {:employeesRuler:justification=center:fontsize=10
12          "NAME" "EMPLOYEE N0" "COMPANY BASED IN";
13          <hline>;
14        }
15      <end header>
16
17      {:employeesRuler:justification=right
18        .Name1:left Employee.EmployeeNumber CompanyNameVar;
19      }
20    <end repeating>
22  <end paper>
```

Importing and Executing a Layout in Maconomy

After we have defined an MPL layout, we want to import into Maconomy. To this end, we can use any Maconomy client that supports the “Print Layout” window—we just execute the “Import Layout” action in there and select our layout.

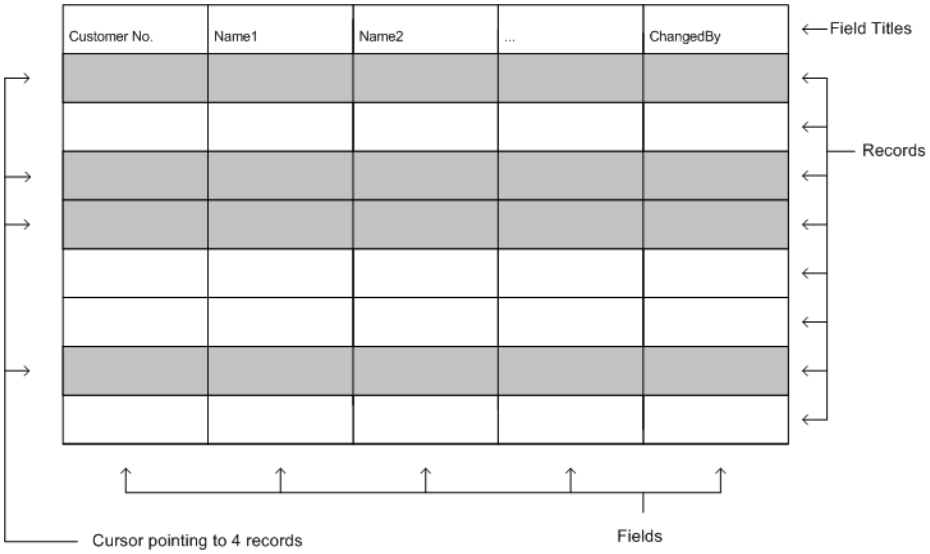
Upon importing, the layout is *compiled* by the MPL compiler. This means that the compiler tries to turn a given MPL text file into an internal MPL representation of this MPL layout. If the given text file represents a valid MPL layout, it gets imported and can be used when printing. Otherwise, error messages that point out which parts of the layout contain errors are reported.

Print Content In More Detail

In this section we describe more formally and in more detail the different kinds of data that are available in a print environment, that is, database cursors and variables. Moreover, as of version 4 of MPL you can fetch custom data from the Maconomy database (the data that has not been included in the print environment) using MQL queries and carry out custom calculations using the Expression Language.

Database Fields and Variables

- **Relations** — Relations are collections of data in the Maconomy database. In this context, a relation can be perceived as a *table of data*. Such a table consists of a fixed number of *columns* and a varying number of *rows*. Each row’s cells contain data, and each column has a unique name, for example, “Customer Number” or “Name1,” signifying the contents of each column.
- **Fields and Records** — In SQL terminology columns are called *fields* and rows are called *records*. Thus a relation is a collection of records, each of which has a number of named fields that contain data as shown in the following figure.



Maconomy is built on a number of different relations. For instance, one relation contains customers, and another relation contains vendors.

A *cursor* points to a number of the records that are contained in a relation. Sometimes a cursor points to all records, but often a cursor is subject to certain assigned *conditions* that ensure that only specific records are included.

When you use Maconomy, the system often displays values that are not contained in a relation. These values are calculated from data in the relations. For example, the discount amount on an item line is calculated from the discount percentage and the extended price.

- **Variables** — The result of these calculations is contained in a *variable* that can be included in printouts exactly like database fields.

Scripts

A script is a program fragment that is used to carry out calculations. Typically, a script ensures that a variable is calculated correctly, but it can also be used to make complicated calculations or updates in the database. For example, the script `S_Page` in the printout “Print Posting Journal” carries out updates in the database—the printout is actually responsible for the entire bookkeeping process.

Scripts cannot be changed using MPL. You must specify when predefined scripts are called, and the MPL compiler ensures that the calls are executed in the same way as in the original printouts.

To ensure that variables, which can be updated by scripts, hold valid values, as well as that the database side effects that scripts might have are executed in the right order, custom layouts must conform to the script structure of their respective original layouts. In other words, the use of scripts in a custom layout must be the same in its original layout—the scripts must be executed the same number of times and in the same order. To ensure this, we must compare the tags that have scripts attached, as well as all repeating and conditional blocks that the tags with scripts are embedded in, because they may change the number of times that the scripts are executed. For more details on script structure, see “Print Structure.”

Custom Data and Custom Calculations

As of version 4 of MPL you can fetch custom data into an MPL layout using MQL queries that are run against Maconomy universes. MQL is a rich query language that is very similar to SQL, but more powerful in some respects. It is a *statically typed language*, which means that queries can be validated when you compile layouts. This validation step can capture a wide variety of errors, and, on the other hand, prove the absence of a certain class of errors (for example, the number of parameters that are passed to the query, their types, as well as incompatible field types of returned records, and so on). MQL queries are *reusable*, which means that you can declare a query once and instantiate its different incarnations with different values of parameters. Moreover, they run against Maconomy Universes, which predefine a lot of very useful joins that are quite hard to get right without knowing the Maconomy database schema intimately.

In addition to these custom data sources, MPL 4 also introduces a means to perform custom calculations by embedding the Expression Language[3] into MPL. You can define new values and variables and populate them with the results of arbitrarily complicated calculations.

MPL Language Basics

This section describes the basic elements of an MPL definition. It describes how the logical structure of a printout is defined using tags, and how the formatting of the individual tags can be modified using attributes.

[3] The Expression Language is used in other Maconomy layout languages like MDML as well.

Simple Tags

Simple tags describe a print element. For instance, `<text ...>` is a simple tag that specifies a text string.

Parenthetical Tags

Parenthetical tags occur in pairs that consist of a start tag and an end tag (corresponds to a start parenthesis and an end parenthesis). The contents are enclosed in the two parenthetical tags, for example, the description of an island is enclosed in the tags `<island>`.

```
... <end island>.
```

Attributes

Attributes can be assigned to all tags. Attributes contain additional information about the layout element that describes the tag. Attributes are specified between the tag name and ">" and are written as *`attribute name=attribute value`*.

Example

If you want the printout to contain the text string "Hello!," you use the tag `text`. Furthermore, you should specify that the text (the title) should be "Hello!" You do this by setting the attribute to "Hello!" The complete code is:

```
<text title="Hello!">
```

Certain attributes are mandatory, whereas you can leave others out. An example of a mandatory attribute is the fact that `text` tags should be assigned a `title` attribute, which specifies the text that should be printed. The attribute that specifies the font of the text, however, is not mandatory, but has a default value that is used if the attribute is not specified.

Attribute Values

All attributes have a *type*. When specifying an attribute, you can only specify attribute values of the permitted type. Furthermore, the type is used for deriving the attribute used when a nameless attribute is used (see "Nameless attributes").

MPL recognizes the following attribute types:

- *STRING* — Strings are specified in quotes and are typically used in connection with text to be shown on the printout, for example, "Hello!"
- *ID* — Identifiers are specified as is, that is, without quotes, for example, DateVar.
- *INTEGER* — Integers, for example, 3 or 19245. An INTEGER cannot be negative.
- *LENGTH* — Lengths are specified as an integer or decimal value followed by a unit of length. The basic length units are `pt` (points, 1/72 inch), `mm` (millimeter), `cm` (centimeter), and `in` (inch). Examples: `3pt`, `0.14cm`, and `12mm`. You cannot specify a negative length.

  You can also use `pagewidth` and `pageheight`, which are the width and height of the printable part of the paper for which the printout is formatted. The length specification `pageheight` can only be used for vertical height, and `pagewidth` can only be used for horizontal width. Example: `0.5pagewidth` specifies half the width of the paper less the left and right margins.

  You can define a length unit yourself, called grid (see "Grid" in "Advanced MPL"). If a grid is defined, you can, for example, specify:

`1grid`.

Finally, you can specify lengths using constants. A number of constants are predefined by the system; you can find these in “Predefined Lengths” in “Advanced MPL.” You can also define your own constants; see “Length Constants” in “Advanced MPL.”

- *BOOLEAN* — Truth values. The only permitted values are true and false.
- *POS* — Positions. These are specified as a pair of lengths, corresponding to horizontal and vertical position, respectively. Example: `(2cm,4pt)`.
- *RULER* — Specification of column layout; see “Rulers—Column Definitions” in “Arrays.”
- *PARAMLIST* — Specification of a list of parameters used for links in MPL for Universe Reports. The parameter list contains a number of parameter/value pairs, separated by a comma. The parameter is a simple text value. The value can be a field name, a variable, or a text. Example with all three types of parameters:

  ```
  [param1=.field1, param2=var1, param3="Text"]
  ```

  For more information, see “MPL for Universe Reports.”
- EXPRESSION — Represents the type of compound calculations that yield a value, either to be printed out, bound to a mutable variable (*var)* or an immutable value (*val)*, or assigned to a tag attribute of type *EXPRESSION*.

  Expressions are always delimited by curly braces, that is, { and }. Valid values of expressions are, for example, {123.4} and { x > 7}.

  For more information, see “Expressions.”

Short Forms

Certain tags are used so often that it is convenient to write them using a short form. For instance, you can abbreviate the tag `<text title="Hello!">` to `"Hello!"` Short forms are different from tag to tag. The short form of a given tag is therefore mentioned when the tag is described.

Short Attribute Forms

If a tag is written using a short form, you cannot specify attributes between the tag name and “>”. Instead, you specify attributes immediately after the short form, separated by colons. Example: the code

```
<text title="Hello!" fontname="Times" fontsize=12>
```

can be abbreviated to

```
"Hello!":fontname="Times":fontsize=12
```

You can also abbreviate the attribute value in logical attributes, that is, attributes that can be assigned the values true and false. Instead of

```
"Hello!":italic=true
```

you can write

```
"Hello!":italic+
```

Also, instead of writing

```
"Hello!":bold=false
```

you can write

```
"Hello!":bold-
```

Nameless Attributes

Some attributes are used very often. Therefore sometimes you can exclude their names to limit the amount of code to be written. Based on the type of attribute value, the MPL compiler automatically identifies the attribute. Such attributes are called *nameless attributes*.

Example

Instead of writing

```
<text title="Hello!">
```

you can write

```
<text "Hello!">
```

because the attribute `title` is nameless for the tag `text`. In its short form, the attribute `justification` for the tag `text` is nameless. Thus, instead of writing

```
"Hello!":justification=center
```

you can write

```
"Hello!":center
```

The description of each tag lists the attributes that can be nameless for the tag in question.

Tags and Attributes

The following sections describe all of the tags that are used in MPL. The description of each tag also lists the tag's attributes. You can find a complete list of all tags and their attributes in "Attribute List" in "Grammar."

Comments

You specify comments by entering two hyphens. The MPL compiler ignores rest of the line after the hyphens. Here is an example:

```
-- This is a comment
-- to a small layout with one text string
<mpl 2>
-- A layout called "Example"
<layout "Example" print="P Invoice"
   originallayout="Standard">
<page "A4">
-- Top of page
<paper>
   -- The printout contains the text: "Hello World":center
-- Bottom of page / end of layout
<end paper>
```

Original Structure Layout

If you are basically satisfied with a printout, but you want to change a few text strings and remove a couple of elements, you should export the original layout so that you get an MPL layout that corresponds to the chosen printout. This makes it easy to implement minor changes.

If you want make extensive changes to the layout, it is often helpful to export the structure layout. A structure layout is a viable template for a printout: The structure of an MPL layout. However, printouts with this layout are empty. After you have exported the structure layout you can fill in the contents and format the layout.

A structure layout can differ for various original layouts that have been assigned to the same printout. The concept of structure layouts does not apply to MPL for Universe Reporting.

Structure of an MPL Layout

This section describes the elements that are necessary in every MPL layout. Specifically, there must always be a heading that specifies the general properties of the printout, followed by a specification of the contents.

mpl

A layout description must always begin with

```
<mpl 2>
```

This tag specifies the version of the MPL language in which the layout is written (in this case, version 2). The MPL compiler supports layouts that were written in previous versions of the language. See “Enhancements in MPL Version 2” for a list of the new features in this version of MPL.

layout

The tag `layout` names the layout and specifies its association with original layouts. The following attributes are mandatory in standard MPL, but ignored in MPL for Universe Reports:

- `title` specifies the name of the layout. This name is used as the name of the layout in the table part of the window Print Layout and in layout selection pop-up fields in print windows. `title` has the type *STRING* and is both nameless and mandatory.
- `print` specifies the name of the printout. This is an existing name that can be found in the window Print Layout. `print` has the type *STRING* and is mandatory.
- `originallayout` specifies the name of the layout that was used as a template for the layout. For more information, see “Print Structure.” `originallayout` has the type *STRING* and is mandatory.

Example

```
<layout "My first layout"
   print="Print_Invoice"
originallayout="Standard" >
```

page

The tag `page` indicates which paper format should be used for the printout. The MPL layout heading must contain a page declaration. Paper formats are specified in the Paper Formats window in Maconomy.

If you, for example, specify an A4 paper format, no elements will be wider than the A4 paper (minus margins), and page breaks will take place when the A4 paper has been filled up vertically. When the MPL definition is compiled, no check is made as to whether the printout is actually printed on the specified type of paper.

The following attributes exist for `page`:

- `name` specifies the paper format for which the layout should be formatted. You can specify paper format names that are defined in the window Paper Formats. The attribute has the type *STRING* and is both nameless and mandatory.
- `orientation` takes the values `portrait` (the paper is standing on the short edge) or `landscape` (the paper is on the long edge). The attribute has the type *ID* and is nameless. The default value (that is, if the attribute is not specified) is `portrait`.

Example

If you want to format the printout for A4, type:

```
<page "A4">
```

If you want to format the printout for US Letter landscape, type:

```
<page "US Letter" landscape>
```

paper

The tag `paper` defines the printout as such. It is a parenthetical tag that surrounds all of the elements that make up the printout (excluding the front page; see below). The `paper` tag takes two attributes:

- `cursor` specifies a cursor name — One page is printed per record within the cursor.

  The template layout determines whether a cursor should be assigned, and, if so, which cursor. For more information, see “Print Structure.” The attribute has the type *ID* and is nameless.
- `script` is used to specify the name of a script to be run for each item in the specified cursor. As is the case with `cursor`, the template layout determines `script`. The attribute has the type *STRING* and is nameless.

Example

```
<paper cursor=Journal script="S_Page"
    ...
<end paper>
```

frontpage

Printouts can have a front page. This page is printed before the rest of the printout, if the printout is executed from a print window (a window with a “Print” button). These dialogs are usually displayed when you select options in the submenu Reporting or use the function “Print...” in the File menu. When you print using the function “Print This” in the File menu, the front page is not printed.

```
<frontpage>
    ...
<end frontpage>
```

The `frontpage` tag takes no attributes. As with other *stacking* tags (see “Stacking Tags”), front pages consist of definitions followed by elements. You cannot use field tags, headers, footers, repetitions, conditions, `newpage`, or `border` on front pages, and the content of the front page is limited to one page. If the information cannot fit on one page, an error message is displayed. Furthermore, you cannot use the `stacking` tag with scripts on a front page. The use of front pages is illustrated in “A Printout Example: Time Sheets.”

Visible Elements

A printout is constructed from the basic elements text, field, variable, island (frame), and line. The remainder of an MPL layout is all about formatting: The placement of the basic elements, whether part of the printout should be repeated for every item in a cursor, and whether parts of the printout should only be printed under certain circumstances.

This section describes the use of the basic elements, and will enable you to create your first simple layout, which we look at in the next section. In this section, we do not look at islands and lines; these are described later.

Predefined Data From the Print Environment

One kind of data that we can print out in an MPL layout is the data from the predefined print environment of our layout, that is, cursor fields and variables.

Fields

The `field` tag

```
<field attributes>
```

specifies that the contents of a database field are to be printed. This `field` tag takes one mandatory attribute:

- `data` specifies the name of the database field to be printed. The attribute is both nameless and mandatory.

Often you want to use the short form of `field` tags:

```
.field
```

Note the dot. This is the short form of

```
<field data=field>
```

If you want to specify from which cursor the field should be taken, you can specify the following attribute:

- `cursor` specifies the name of the cursor from which the field is to be taken. The attribute has the type *ID*.

Often you want to use the short form of `field` tags:

```
cursorname.field
```

Note the dot. This is the short form of

```
<field data=field cursor=cursorname>
```

If you do not specify a cursor name, the value is taken from the nearest cursor with a field of the specified name. However, it is good design practice to specify a cursor name if several cursors exist that contain the same field name.

The attributes for specifying the typography of the field (for example, `fontname`, `fontsize`, `bold`, `italic`, and `underline`) work in exactly the same way as for texts, and are thus not explained here. If you do not enter a font name and a size, the field has the font Helvetica 9 pt. Apart from this, you can specify the following attributes:

- `justification` specifies the justification of the field. `justification` is nameless in the short form and has the type *ID*. It can take the values `left`, `center`, and `right`. If you have not specified `justification`, a field is justified according to its type: Fields of the

types *INTEGER*, *REAL*, and *AMOUNT* are right-justified, whereas all other fields are left-justified.

- `width` functions just like the `width` attribute for text. However, the width of fields is set to the default value if `width` is not specified. This is due to the fact that the value of the field is not available at the time when the layout is being compiled.
- `indent` specifies an extra indentation of a field. The box will also be wider in accordance with the specified length. You cannot use the `indent` attribute in combination with right justification and centering. `indent` has the type *LENGTH*. You cannot use this attribute when the field appears in a canvas.
- `pos` specifies the positioning of a field element within canvas. The attribute is nameless and has the type *POS*. You can only use it when the field appears on a canvas, and is mandatory, if so (see “Canvas” for further information about the canvas).
- `zerosuppression` specifies whether the value 0 is to be printed out, if the value is zero for numeric fields, that is, fields of the type *AMOUNT*, *INTEGER*, or *REAL*. You cannot specify this attribute for fields of other types.
- `link` specifies that the text is to function as a link. This applies to MPL for Universe Reporting only. For more information, see “Links.”
- `textdirection` specifies the algorithm that should be used to render text. `textdirection` is nameless in the short form and has the type *ID.* The valid values are **NOBIDI** (non-bi-directional algorithm), **LTR** (bi-directional algorithm with left-to-right direction preference), and **RTL** (right to left bi-directional algorithm with right-to-left direction preference). For a thorough explanation of this attribute and its possible values, see “Bi-Directional Printing” under the Advanced MPL chapter.

In MPL 3 the attributes `wrap` (BOOLEAN), `lines` (INTEGER), `height` (LENGTH) are also supported and are used to control text wrapping around multiple lines, as well as adding explicit line breaks. See “wrap Attribute for <text>, <field>, and <var> Tags.”

> **Note:** In MPL 4, the <field> tag has been desupported. Its short form, however, is still valid as one of the short forms of the <eval> tag. For more detail, see “Field and Variable Reference Tags Desupported in MPL4”.

Example

```
Journal.Journalnumber:bold+:fontname="Courier"
```

Variables

The `var` tag

```
<var attributes>
```

specifies that the contents of a variable are to be printed. The variable tag takes one mandatory attribute:

- `data` specifies the name of the variable to be printed. The attribute has the type *ID*.

Often you want to use the short form of `var` tags:

```
var
```

which is the short form of

```
<var data=var>
```

The attributes `justification`, `fontname`, `fontsize`, `bold`, `italic`, `underline`, `width`, `indent` (replaced by `pos` in canvases), `zerosuppression`, and `link` work in the same way for variables as for fields (see the previous section).

In MPL 3 the attributes `wrap` (BOOLEAN), `lines` (INTEGER), and `height` (LENGTH) are also supported and are used to control text wrapping around multiple lines, as well as adding explicit line breaks. See “wrap Attribute for <text>, <field> , and <var> Tags.”

> **Note:** In MPL 4, the `<field>` tag has been desupported. Its short form, however, is still valid as one of the short forms of the `<eval>` tag. For more detail, see “Field and Variable Reference Tags Desupported in MPL4.”

Example

```
Companyname:indent=2cm:bold+
```

Predefined Page Number Variables

Every print environment includes the following variables for printing page numbers:

- **PageNumber** – prints out the current page number.
- **TotalNumberOfPages** – prints out the total number of pages of the generated print

These variables can be used in any expression to achieve the required page number formatting, for example:

- ^{ "Page " + PageNumber + " of " + TotalNumberOfPages } --e.g. Page 3 of 16
- ^{ PageNumber + "/" + TotalNumberOfPages } -- e.g. 3/16
- ^{ "Page " + PageNumber + ", remaining " + (TotalNumberOfPages - PageNumber))} -- e.g. Page 3, remaining 13

User defined data

In addition to printing out the predefined data, we can define the data—whether it is simple static text or complex arbitrarily expressions.

Texts

The `text` tag

```
<text attributes>
```

specifies that a text is to be printed, for example, a column heading or a label. The `text` tag takes one mandatory attribute:

- `title` specifies the text that is to be printed. The attribute has the type *STRING* and is both nameless and mandatory.

Often you want to use the short form of text tags:

```
"text"
```

which is the short form of

```
<text title="text">
```

Apart from this, you can specify the following attributes:

- `justification` specifies whether the text should be left-, center-, or right-justified.

  If the `width` attribute is also used, the justification takes place within the specified width. Otherwise, the justification occurs within the space that is made available by the surrounding elements. For instance, a right-justified text on the page is placed to the extreme right on the page, and a centered text in an island is placed in the middle of the island.

  > **Note:** If the attribute stretch on the island is false, the island will only be as wide as its contents, and if the text is the only content, you cannot tell whether justification is left, center, or right.

  `justification` has the type *ID* and is nameless. It can take the values left, center, and right. If justification is not specified, texts are left-justified.

- `fontname` specifies the font(s) to use for printing the text. This attribute is of type LIST OF STRINGS, but single STRING values are also accepted for backwards compatibility reasons. When printing text, the MPL engine selects the appropriate font for each character in that text..For example, the first font from the list of fonts that supports the given character will be selected. The default font is Helvetica. For more information on fonts, see “Font Administration in Maconomy” in the *Deltek Maconomy System Administrator Guide*.

- `Fontsize` specifies the size of the selected font. The attribute has the type *INTEGER*. The default font size is 7pt.

- `bold` specifies that the text should be printed in **boldface**. The attribute has the type *BOOLEAN*.

- `italic` specifies that the text should be printed in *italics*. The attribute has the type *BOOLEAN*.

- `underline` specifies that the text should be printed underlined. The attribute has the type *BOOLEAN*.

- `width` specifies the width of the box that surrounds the text. The attribute has the type *LENGTH*.

  If this attribute is not specified, the remaining part of the layout is formatted according to the actual width of the text. The text box is hence adjusted to fill in the available width space.

  If `width` is specified, the remaining part of the layout is formatted as if the text had the specified width, and the text box is not adjusted. This has the following consequences:

  - If the length is specified to be shorter than the text’s own width, the text is cut off.
  - Justification is relative to the specified width.
  - If the text appears in a column, this column is given a width that ensures space for the width specified by the attribute width.

- `indent` specifies an extra indentation of the text. The text box will also be wider in accordance with the specified length. You cannot use the `indent` attribute in combination with right justification and centering. `indent` has the type *LENGTH*. You cannot use this attribute when the text appears in a canvas (see “Canvas” for further information about the canvas).

- `pos` specifies the positioning of a text element of a canvas. The attribute is nameless and has the type *POS*. You can only use it when the text appears on a canvas, and is mandatory, if so (see “Canvas” for further information about the canvas).

- `link` specifies that the text is to function as a link. This applies to MPL for Universe Reporting only. For more information, see “Links.”

  In MPL 3 the attributes `wrap` (BOOLEAN), `lines` (INTEGER), and `height` (LENGTH) are also supported and are used to control text wrapping around multiple lines, as well as adding explicit line breaks. See “wrap Attribute for <text>, <field>, and <var> Tags.”

- `textdirection` specifies the algorithm that should be used to render text. `textdirection` is nameless in the short form and has the type *ID.* The valid values are **NOBIDI** (non-bi-directional algorithm), **LTR** (bi-directional algorithm with left-to-right direction preference), and **RTL** (right to left bi-directional algorithm with right-to-left direction preference). For a thorough explanation of this attribute and its possible values, see “Bi-Directional Printing” under the Advanced MPL chapter.

Example

```
"Hello world!":center:width=4cm:fontsize=18

"你好，世界 สวัสดีเพื่อนของฉัน hello":fontname=["Helvetica", "JhengHei", "Leelawade"]

-- in this example, JhengHei will be chosen to print Simplified Chinese
characters, Leelawade to print Thai characters and Helvetica to print the Latin
characters.
```

See also “Alternative Text Tag” in “Advanced MPL.”

Arbitrary Expressions

The `eval` tag, introduced in MPL 4,

```
<eval {expression} other_attributes>
```

specifies that the given expression should be evaluated and the result value printed. The `eval` tag takes one mandatory attribute:

- `expression` of type *EXPRESSION*, which denotes an arbitrary *Expression Language* construct that is evaluated and the result of that evaluation is printed. For more information on expressions, see “Expressions” and “Literal Values for Different Types
- When declaring a `var` or a `val` or just using literals as values in expressions, it is useful to know how the different literals look for values of different types:

Type	Comma separated example values
INTEGER	`457, 77, -123`
REAL	`41.789, 99.4`
AMOUNT	`AMOUNT(99.74), AMOUNT(6.45)`
BOOLEAN	`true, false`
DATE	`DATE(2013, 12, 25), DATE(1987, 2, 15)`
TIME	`TIME(12, 23, 58), TIME(23, 15, 33)`
STRING	`"Text”, "example string”`
POPUP	`GenderType'Male, CountryType’France`

Standard FunctionsWhen declaring a `var`, `val` or just using literals as values in expressions, it is useful to know how the different literals look like for values of different types:

Type	Comma separated example values
INTEGER	`457, 77, -123`
REAL	`41.789, 99.4`
AMOUNT	`AMOUNT(99.74), AMOUNT(6.45)`
BOOLEAN	`true, false`
DATE	`DATE(2013, 12, 25), DATE(1987, 2, 15)`
TIME	`TIME(12, 23, 58), TIME(23, 15, 33)`
STRING	`"Text”, "example string”`
POPUP	`GenderType'Male, CountryType’France`

The preferred form of evaluating and printing out expressions is the short form of the `eval` tag, for example,

```
^{addMonths(currentDate, 20)}
```

The full form of the above `eval` tag is:

```
<eval expression={addMonths(currentDate, 20)}>
```

There is no compelling reason, though, to use the full form, and it is in the language mostly for completeness reasons.

The `justification`, `fontname`, `fontsize`, `bold`, `italic`, `underline`, `width`, `indent` (replaced by `pos` in canvases), `zerosuppression`, `link`, `wrap`, `lines`, and `textdirection` attributes work in the same way for `eval` in MPL 4 as for `field` and `variable` tag for MPL 3 (see the previous sections).

Example MPL Layout

This section contains an example of the creation of an MPL layout. The layout developed in the example is a new layout for the printout “Print Warehouse Information Card” and will be based on the layout “Standard.”

Getting Started

When you want to define a new layout, it is common practice to export either the structure layout or the original layout for a layout with the desired functionality. Thus to define a new layout for the layout “Print Warehouse Information Cards” we export the structure layout for the layout “Standard” from the window Print Layout in the Maconomy client.

The result of the export is a file in which the printout name and original layout name have been correctly set, and cursor and script names have been attached to blocks as required. A list of all available variables and fields is provided in the document “List of variables in MPL.” This document is updated for each version of Maconomy and can be found on the Maconomy Partner web site.

Header

The printout should be formatted for A4 paper and named “Simple.” Thus you should insert the following layout heading:

```
<mpl 2>
<layout "Simple"
    print="Print_Inventory_Info_Card"
    originallayout="Standard">
<page "A4">
```

Page

The layout “Standard” for the printout “Print Warehouse Information Cards” prints one warehouse information card per page. This is due to the specification of a cursor in the `paper` tag. The cursor is “`Inventory`.” A script is also specified, namely “`Inventory_PageScript1`.” The resulting code is

```
<paper cursor=Inventory script="Inventory_PageScript1">
    ...
<end paper>
```

Note that the heading specified in the last section, combined with using the `paper` tag mentioned before, is very similar to the resulting layout of an export of the structure layout.

Contents

The procedure for placing elements side by side in the structure is described later in this document, so we will therefore use the following structural form for this exercise:

```
descriptive text
field/variable
field/variable
```

where the descriptive text is formatted as bold text. Enter the following code to print out name and address:

```
"Name":bold+
    .InventoryName:indent=1cm
"Address":bold+
    .Address1:indent=1cm
    .Address2:indent=1cm
    .Address3:indent=1cm
    .Address4:indent=1cm
```

Complete Layout

Now we have the complete layout:

```
<mpl 2>
<layout "Simple"
          print="Print_Inventory_Info_Card"
          originallayout="Standard">
<page "A4">
<paper cursor=Inventory script="Inventory_PageScript1">
    "Name":bold+
        .InventoryName:indent=1cm
    "Address":bold+
        .Address1:indent=1cm
        .Address2:indent=1cm
        .Address3:indent=1cm
        .Address4:indent=1cm
    "Attention":bold+
        .AttPerson:indent=1cm
    "Company No.":bold+
        .CompanyNumber:indent=1cm
    "Company Name":bold+
       CompanyName1Var:indent=1cm
    "Phone":bold+
        .Telephone:indent=1cm
    "Fax":bold+
        .Telefax:indent=1cm
    "Telex":bold+
        .Telex:indent=1cm
<end paper>
```

The result of importing this definition and printing with the new layout is a number of pages, each of which looks like this:

Name
Standard
Address
The Docks
Harbourtown, Berks

Attention

Company No.
20
Company Name
Advertising Agency 1
Phone
234 521555
Fax
234 521556
Telex
warehouse@20.com

Basic Tags

This section contains a detailed description of a number of most commonly used MPL tags.

Stacking Tags

A *stacking* tag is a parenthetical tag that is used for placing elements on top of each other. Most parenthetical tags in MPL are stacking tags. This means that all elements in such a tag are placed on top of each other—they are stacked. We have already seen an example of a stacking tag: In the `paper` tag, the things written between `<paper>` and `<end paper>` are all placed below each other. The only parenthetical tags in MPL that are not stacking are `row` and `canvas`.

The contents of stacking tags are a number of elements that are to be printed. Before these elements, you can specify various definitions (see “Rulers—Column Definitions” in “Arrays” and “Length Constants” in “Advanced MPL” for further information about these definitions). These definitions apply within the tag in which the definitions are made. The stacking tag in which the definition appears is called the definition’s *scope*.

Stacks

The parenthetical tag

```
<stack attributes>
    ...
<end stack>
```

is the simplest example of a stacking tag. Without attributes it is used to ensure that elements are stacked on top of each other. This can be useful in connection with rows that make it possible to place two stacks next to each other.

> **Note:** `stack` cannot appear in a canvas or in rows if the `script` attribute has been specified, or if the stack has a header or footer. For more information, see “Print Structure.”

`stack` has the following attributes:

- `script` specifies the name of a script (program) that is to be run before the contents are printed. Typically, the script takes care of initializing variables that are to be used later in the printout. Scripts can only be used in the same way as in the original layout. For more information, see “Print Structure.” `script` is nameless and has the type *STRING*.
- `baseline` is used to specify the baseline of the stack. If two elements are placed next to each other, they can be placed in such a way that their baselines are aligned. If `baseline` is set to `top`, the stack inherits the baseline of the upper element—that is, if a stack is placed next to a text, the text appears at the same baseline as the upper element of the stack. If `baseline` is set to `bottom`, the stack inherits the baseline of the lower element. The default value for `baseline` is `bottom`, and `baseline` has the type *ID*.

  > **Note:** The baseline for texts, variables, and fields is determined by the font used. Here the baseline is the line on which the text is printed. The baseline is placed immediately below the bottoms of letters such as “a” and “b,” while letters such as “q” and “g” extend below the baseline.

- `height` specifies the height of the stack. This only influences the placing of subsequent elements. If the contents of the stack cannot be placed within the specified margin, or if the height of the contents cannot be determined (due to conditions or repetitions), an error message is displayed. Furthermore, you cannot use the `height` attribute if the stack contains other stacks with a header or a footer, or if the `script` attribute has been set—the system displays an error message. If `height` is not specified, the height of the stack is dictated by its content. The attribute has the type *LENGTH*.
- `width` specifies the stack's width. This influences the positioning of elements to the right of the stack as well as the justification of elements in the stack. If the contents of the stack cannot be placed within the specified margins, the system displays an error message. If `width` is not specified, the stack's width is stretched to fit the space available. The attribute has the type *LENGTH*.
- `indent` specifies extra indentation of the stack (all elements contained in the block are indented). The attribute has the type *LENGTH*. This attribute cannot be used when the stack appears in a canvas.
- `pos` specifies the positioning of the stack in a canvas. The attribute is nameless and has the type *POS*. You can only use it when the stack appears in a canvas, and is then nameless and mandatory. See also "Exact Positioning of Elements" in "Canvas."

Note: Stacks cannot appear in a canvas or in rows if the `script` attribute has been set, or if the stack is provided with a header or a footer. For more information, see "Print Structure."

Example

```
<stack>
    "Upper"
    "Lower"
<end stack>
```

Islands

The parenthetical tag

```
<island attributes>
    ...
<end island>
```

defines a stacking tag. The tag results in the drawing of a frame around the contents and is used for highlighting or grouping information on printouts.

The behavior of an island can be modified using the following attributes:

- `stretch`. If this attribute is set to `false`, the island's width is determined by its contents. If the attribute is not set or is set to `true`, the island's width is determined by its surrounding elements. If the island is defined on the page, it is stretched across the entire width of the page. If the island is positioned in a column, it is stretched to the width of the column. The attribute has the type *BOOLEAN*.
- `baseline` is used to specify the baseline of the island. As for the `stack` tag, you can set `baseline` to either `top` or `bottom` to specify whether the island should inherit the baseline of the upper or lower element. In addition, you can set `baseline` to `title`,

meaning that the island inherits the baseline of the island title. The default setting is `bottom`, and the attribute has the type *ID*.

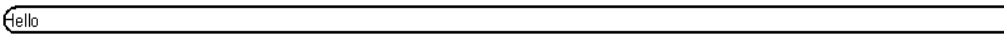
- `justification` only makes sense if the `width` attribute has been set or if the `stretch` attribute has been set to `false`. In these cases, the island may be smaller than the space allotted to it by the surrounding elements. It can therefore be placed to the left, in the middle (default), or to the right. `justification` has the type *ID* and can take the values `left`, `center`, and `right`.
- `rounded` is used to specify whether the corners of the island should be rounded (default) or square. The attribute has the type *BOOLEAN*.
- `topmargin` is used to specify the upper inner margin of the island, that is, the distance between the upper line of the island and the top of the upper element. The attribute has the type *LENGTH*.
- `bottommargin` is used to specify the lower inner margin of the island, that is, the distance between the lower line of the island and the bottom of the lower element. The attribute has the type *LENGTH*.
- `leftmargin` is used to specify the left inner margin of the island, that is, the distance between the left line of the island and left side of the contents. The attribute has the type *LENGTH*.
- `rightmargin` is used to specify the right inner margin of the island, that is, the distance between the right line of the island and right side of the contents. The attribute has the type *LENGTH*.
- `indent` is used to indent the island and can therefore you can only use it if the `justification` attribute has been set to `left`. The attribute has the type *LENGTH*.
- `pos` is used to specify the position of an island in a canvas. The attribute is nameless and has the type *POS*. You can only use it when the island is a part of a canvas, and in those cases it is nameless and mandatory. See also “Exact Positioning of Elements” in “Canvas.”
- `height` is used to specify the height of the island’s contents. If the island’s contents are higher than the value specified, the compiler displays an error message. The attribute has the type *LENGTH*.
- `width` is used to specify the width of the island’s contents. If the island’s contents are wider than the value specified, the compiler displays an error message. If `width` has been set, the `stretch` attribute value is ignored. The attribute has the type *LENGTH*.

You can also use island to frame rows (see “Rows in Stacking Tags” in “Arrays”), but only if the attributes `leftmargin`, `rightmargin`, and `indent` have been set to zero (or not set, because zero is the default value). If these attributes have not been set correctly, you cannot refer to rulers that have been defined outside the island.

Example

```
<island>
    "Hello"
<end island>
```

will result in the following printout



The MPL fragment

```
<island leftmargin=1cm
        rightmargin=1cm
        topmargin=5mm
        bottommargin=5mm
        stretch->
    "Hello"
<end island>
```

will result in the following printout

For further information and an illustration of the lengths that can make up the formatting of an island, see “Island Lengths” in “Tips and Tricks.”

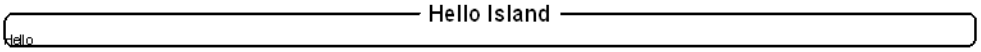
Island Titles

You can assign a title to an island. Such a title will be printed as part of the top line of the island. The `island` tag has a number of attributes that determine the behavior of the title:

- `title` is used to specify the island title text. The attribute has the type *STRING* and is nameless.
- `titlejustification` is used to specify whether the title should be placed on the left, in the middle, or to the right on the top line of the island. The possible values are thus `left`, `center` (default), and `right`.
- `fontname`, `fontsize`, `bold`, `italic`, and `underline` are used to specify the text format. See the description of the `text` tag for further information on the use of these attributes. The default font is 16pt, bold, Helvetica.

Example

```
<island title="Hello Island" bold+ fontsize=12>
    "Hello"
<end island>
```



Repetitions, Conditions, and While

The result of executing an MPL print definition of course depends on the data that is stored in the Maconomy system. In this manual, we started out by setting up a print definition that would print selected setup information about warehouses. When the print definition is executed, that is, not when it is compiled, the data on the resulting printout is an exact representation of data in the Maconomy system.

An MPL print definition also depends on other forms of data. When you print an invoice, naturally you want to print all invoice lines, and the number of invoice lines depends on data that is in the Maconomy system at the time of printing. To do this, you can use the `repeating` tag that prints out its contents for each record in a specified cursor. Similarly, you may want only to print lines with cash discount if the customer actually receives a discount. To do this, you can use the `conditional` tag that prints out its contents subject to a condition specified in the MPL definition.

You cannot define your own repetitions and conditions in MPL, because the MPL compiler demands that you follow the structure of the original layout. For more information, see “Print Structure.” To ensure this, it is recommended that you base your layout on a structure layout that contains the necessary conditions and repetitions.

Repeating Blocks

The parenthetical tag

```
<repeating attributes>
    ...
<end repeating>
```

is used for repeating the contents for each record in a given cursor. The use of `repeating` iterating over the predefined standard cursors must follow the way in which it is used in the original layout. For more information, see “Print Structure.” The following describes how the block is repeated is specified using the `cursor` attribute:

- `cursor` specifies the name of the cursor, which defines the repetition. It can be either a standard predefined cursor or a custom MQL cursor defined in an MPL 4 layout. The contents of the repetition (between `<repeating ...>` and `<end repeating>`) will be printed for each record in the cursor, and fields from the cursor will be available within the tag. `cursor` has the type *ID* and is mandatory and nameless.

`repeating` also has the following attributes:

- `groupby` is used to define the `cursor` attribute. It enables grouping of the records in the cursor. Records where the fields that are marked by the `groupby` attribute are alike will become grouped, and each group can be assigned a common heading.

  An inner repeating block for the same cursor but without the `groupby` attribute must always be placed inside a repeating block with the `groupby` attribute—meaning that with several nested repeating blocks with the same cursor, the innermost block should have no `groupby` attribute. A repetition within a repetition block will terminate when the value of the fields specified by the `groupby` attribute of the surrounding `groupby` change. Note that the `groupby` attribute must be used in exactly the same way as in the original layout.

  The following example is from the printout “Print Item Group Statistics,” layout “Standard”:

```
<repeating cursor=ITEM
    groupby=[ItemGroup]
        script="ITEMGROUPBLOCK">
            .ItemGroup:bold+
                ...
                <repeating cursor=ITEM script="ITEMBLOCK">
                    .ItemText1
                <end repeating>
<end repeating
```

  The outer repetition block will begin with an item record in some item group (and print it). As long as the item group does not change, the inner repetition block will print out the item text, but as soon as the inner repetition block reaches an item record in a new item group, the repetition will terminate, and the outer repetition block will take over, so that a new item group will be printed.

The printout “Print Item Group Statistics” will therefore look like this:

ITEM GROUP	Printers						
ITEM		SALES in DKK		GROSS MARGIN in DKK		MARKUP %	
		MONTH	YTD	MONTH	YTD	MONTH	YTD
1025	HP Light 54-XP2	3.498,75	87.101,73	0,00	-366,56	0,0	-0,4
1026	Compaq XT-68	4.158,06	78.670,52	0,00	-332,60	0,0	-0,4
1028	HP Soft 405	13.435,20	13.435,20	-16.564,80	-16.564,80	-55,2	-55,2
1031	HP Gold X-4	1.186,00	63.444,21	-200,00	-310,82	-14,4	-0,5
1038	Leenex 406-PJ	61.298,10	61.298,10	61.298,10	61.298,10	0,0	0,0
TOTAL	Printers	83.576,11	303.949,76	44.533,30	43.723,32	114,1	16,8

ITEM GROUP	Misc.						
ITEM		SALES in DKK		GROSS MARGIN in DKK		MARKUP %	
		MONTH	YTD	MONTH	YTD	MONTH	YTD
1043	Ixo 19" Monitor	1.728,00	1.728,00	-72,00	-72,00	-4,0	-4,0
1048	Special Spring Offer	14.000,00	14.000,00	14.000,00	14.000,00	0,0	0,0
TOTAL	Misc.	15.728,00	15.728,00	13.928,00	13.928,00	773,8	773,8

Be aware that incorrect use of repeating blocks might crash the Maconomy server. Consider the following MPL-fragment:

```
<repeating cursor=myCursor groupby=[myField]>
   -- (1) Fields of myCursor will assume the "first" record values
   -- for each group of records (with identical myField)
   <repeating cursor=myCursor>
      -- (2) Fields of myCursor will assume all record values
   <end repeating>
      -- (3) Fields of myCursor cannot be accessed. This should result
      -- in an MPL compilation error, but unfortunately
      -- crashes the server
<end repeating>
```

Fields from the cursor named `myCursor` can only be used in subfragments (1) and (2) and cannot be used in subfragment (3). Note that only some fields make sense in (1), notably `myField` and derived fields.

The values for the `groupby` attribute are specified as [ID_1,...,ID_2] where each ID specifies a field name in the cursor. The attribute is nameless.

When repeating over an MQL cursor in MPL 4, the use of `groupby` is strongly discouraged in favor of defining a multilevel MQL query and using nested repeating to iterate over groups. For more detail, see “Multilevel Queries.”

- `script` specifies the name of a script (program) that is to be run before the contents are printed. Typically, the script takes care of initializing variables that are to be used later in the printout. Scripts can only be used in the same way as in the original layout. `script` is nameless and has the type *STRING*.
- `skipHeaderFooterIfEmpty` specifies whether the header and footer should be skipped if no iterations took place, that is, when the cursor had no records. It is of type *BOOLEAN* and defaults to `false`. If you want to reference cursor fields of this repeating in either the header or the footer, you must set this attribute to `true`.
- `height` specifies the height of each iteration of the repetition. If the contents cannot be placed within the specified height, or if the height of the contents cannot be determined (due to conditions or repetitions), an error message is displayed. If `height` is not specified, the height of the stack will be dictated by its content. The attribute has the type *LENGTH*.

- `width` specifies the repetition's width. This influences the positioning of elements to the right of the stack as well as the justification of elements in the stack. If the contents of the stack cannot be placed within the specified width, the system will issue an error. If `width` is not specified, the stack's width will be stretched to fit the space available. The attribute has the type *LENGTH*.
- `indent` specifies extra indentation of the repetition (all elements contained in the block are indented). The attribute has the type *LENGTH*.

The use of the attributes `cursor`, `groupby`, and `script` must be exactly as in the original layout, when iterating over standard, predefined cursors. For more information, see "Print Structure."

> **Note:** Repetitions cannot appear in a canvas (see "Canvas") or in rows (see "Arrays").

Conditionals

The parenthetical tag

```
<conditional attributes>
    ...
<end conditional>
```

is used to specify elements which should only be printed under some circumstances. Whether the contents should be printed is determined by the value of exactly one of the following attributes:

- `variable` specifies the name of the variable that must be `true` for the contents to be printed. In the print definition for "Print Posting Journal,", for instance, a conditional with the variable "IncludeTotals" is used to determine whether totals should be printed. This variable is set to `true` when you select the field "Include Totals" in the client Print window. The `variable` attribute is nameless and has the type *ID.*
- `field` (*supported in MPL 3 and 4 only*) specifies the name of the field that must be `true` for the contents to be printed. This attribute goes hand in hand with the optional `cursor` attribute that the specified field must belong to. If not specified, `cursor` is resolved automatically to the first cursor in the scope chain that contains the `field` in question. Both `field` and `cursor` attributes are of type *STRING.*
- `expression` (*supported by MPL 4 only*) specifies a Boolean expression that must evaluate to `true` for the contents to be printed. `expression` is nameless and has the type *EXPRESSION.*

The remaining attributes of the tag `<conditional>` include:

- `script` specifies the name of a script (program) that is to be run before the contents are printed (that is, if the condition is true). `script` is nameless and has the type *STRING*.
- `negate` is of the type *BOOLEAN*. If it is set to `true`, the value of the conditional will be negated—that is, the content of the conditional will be printed if its value is `false`. This applies to MPL 3 and MPL for Universe Reports only.
- `baseline` is used to specify the baseline of the conditional block. As for stacks, `baseline` can be set to either `top` or `bottom`, specifying whether `baseline` should be inherited from the upper or lower element.
- `height` specifies the height of the conditioned block. If the condition is `true` (and the contents are printed), the following elements will treat the conditioned block as if it had the given height. If the contents do not fit in the allotted space, or the height of the contents

cannot be determined (see the `height` attribute for `stack`), an error message is displayed. If the attribute is not specified, the height is dictated by the contents. The attribute has the type *LENGTH*

- `width` specifies the width of the conditioned block. This influences the positioning of elements to the right of the conditional as well as the justification of elements in the conditional. If the contents of the conditional block cannot be placed within the specified space, the system will display an error message. If `width` is not specified, the conditional's width will be stretched to fit the space available. The attribute has the type *LENGTH*.
- `indent` specifies extra indentation of the conditional block (all elements contained in the block are indented). The attribute has the type *LENGTH*.

The use of conditions is subject to certain restrictions that are dictated by the original layout. See "Print Structure."

> **Note:** A conditional cannot appear in a canvas. At most, one conditional can appear in any given row (this restriction does not apply to MPL 3).

In MPL 3 the attributes `field` (STRING) and `cursor` (STRING) are also supported and are used to make a conditional depend on values of a database field. For more information on conditional changes in MPL 3, see "Conditionals" and "Skipping False Conditionals." MPL 4 adds expressions as a possible means of specifying the guarding condition for conditionals. For more detailed description of expression, see "Expressions."

Example 1

In this example we use a nameless attribute `variable` to specify the condition guarding this `conditional`:

```
<conditional EUSalesVar>
    "VAT NO."
    PayerVATNumber:indent=1cm
<end conditional>
```

Example 2

In this example we use a nameless attribute `expression` to specify the condition guarding this `conditional`:

```
<conditional {AgeVar > 75 && congratulateSeniorsVar} >
    "Congratulations to " .EmployeeName
    "on having lived for"  AgeVar "years!"
<end conditional>
```

While Loops

The parenthetical tag

```
<while attributes>
    ...
<end while>
```

is a looping construct in MPL—that is, it keeps executing its children as long as its guarding `condition` attribute evaluates to `true`. `while` loops can be used together with the Expression Language to perform arbitrary calculations. On top of that, they can be also employed to repeat the execution of a chosen part of an MPL layout while a certain condition holds. In the latter scenario, though, you should remember that `while` loops are part of the layout script structure in the same ways as repeating blocks and conditionals are, which basically means that when you customize an existing layout you cannot add a `while` loop around a block of MPL code from the original layout that might potentially call scripts (at least one of the tags has a script attached). For more details on this restriction, see “Print Structure.”

The `<while>` tag has the following attributes:

- `condition` specifies the condition that must be `true` for the next iteration of the loop to execute. The attribute is mandatory, nameless, and of type *EXPRESSION.*
- `skipHeaderFooterIfEmpty` specifies whether the header and footer of this `while` loop should be skipped if no iterations took place, that is, when the guarding condition evaluated to false the first time the loop was executed. The attribute is of type *BOOLEAN* and defaults to `false`. When set to `true`, both the header and the footer are not printed when no iterations took place.

`while` loops support `indent`, `height`, and `width` attributes in the exact same way as repeating blocks.

Note: The <while> tag has been introduced in MPL 4 as of TPU 16 SP2.

Example 1

The following code implements the insertion sort algorithm, sorting characters in the input `numbers` string in ascending order. It goes through the characters in the `numbers` string one by one, and inserts them in ascending order into the resulting `sortedNumbers` string. Because `sortedNumbers` string is at all times sorted, the inner `while` loop stops as soon as it finds an element bigger than or equal to the current one—this is exactly the right spot to insert the current element.

```
<var numbers {"918273645"}>
<var sortedNumbers {""}>

<var currentIndex {0}>
<val numbersLength {length(numbers)}>

-- Iterate over all the numbers
<while {currentIndex < numbersLength}>
  <val currentElem {charAt(numbers, currentIndex)}>
  <var insertionIndex {0}>

  -- Find insertionIndex to insert currentElem into the resulting sortedNumbers
  <while {    insertionIndex < currentIndex
          and charAt(sortedNumbers, insertionIndex) < currentElem} >
    -- Iterate as long as currentElem is bigger than the elements in
    -- sortedNumbers
    <assign insertionIndex {insertionIndex +1}>
  <end while>

  -- Insert currentElem into sortedNumbers at the found insertionIndex
  <val prefix {substring(sortedNumbers, 0, insertionIndex)} >
  <val postfix {substring(sortedNumbers, insertionIndex)}>
  <assign sortedNumbers {prefix + currentElem + postfix}>
```

```
  <assign currentIndex {currentIndex +1}>
<end while>

^{"Input numbers: " + numbers} -- prints 918273645
^{"Sorted numbers: " + sortedNumbers} -- prints 123456789
```

Example 2

In addition to performing calculations, you can also use `while` loops to repeat executing an MPL code snippet as long as a certain condition is satisfied.

For example, suppose we have a layout that lists all of the employees in a repeating over the `Employee` cursor. A skeleton of it could look somewhat like this:

```
<repeating Employee>

…

<end repeating>
```

We want to print the contents for each employee twice, adding the text “Copy 1 of 2” the first time and “Copy 2 of 2” the second time. To this end, we can modify the above snippet in the following way:

```
<var copyNumber {1}>
<while {copyNumber <= 2}>
  <repeating Employee>
   …
  <end repeating>


  ^{"Copy " + copyNumber + " of 2″}
  <assign copyNumber {copyNumber +1}>
<end while>
```

Horizontal Lines and Spaces

Inside a stacking block, it is sometimes convenient to control horizontal spacing between the stacked elements as well as to print horizontal lines in between them. This can be achieved be means of the `<skip>` and `<hline>` tags in MPL.

skip

The simple tag

```
<skip attribute>
```

is used to insert extra vertical space between elements and has the following attribute:

`height` is used to specify the height of the extra space that is to be inserted. The attribute is mandatory and nameless and has the type *LENGTH*.

Example

```
<island>

    "Hello!"

<end island>

<skip 5mm>

<island>
```

```
    "Bye bye!"
<end island>
```

Here the `skip` attribute is included in the block to ensure space between the two islands.

hline

The simple tag

```
<hline attribute>
```

is used for inserting a horizontal line in a stacking tag. The line will be stretched across the width of the stacking tag. The tag has one attribute:

`multi` is used to specify the number of lines that are to be inserted. For example, set the `multi` attribute to `2` to insert a double line underneath a total. The attribute has the type *INTEGER*. If not specified, one line is inserted.

Example

```
<stack width=1cm>
    "123":right
    "+234":right
    <hline>
    "357":right
    <hline multi=2>
<end stack>
```

This results in the following printout:

Arrays

Arrays are used for arranging elements into rows and columns. Because arrays can be used within arrays, they offer almost unlimited possibilities for formatting data.

Columns and Rows

array

The parenthetical tag

```
<array attributes>
    ...
<end array>
```

specifies an array. The contents of an `array` tag must be rows. The rows are placed above each other, and the elements in the rows are arranged in columns. You can use the following short form:

```
{ shortattributes
    ...
```

```
}
```

`array` has the following attributes:

- `ruler` is used to specify the format of the columns of the array. Rulers are described in "Rulers—Column Definitions." You can specify a ruler value or an ID that refers to a ruler with a specified name. The attribute is nameless.
- `baseline` is used to specify the baseline of the array. As for the tag `stack`, you can set `baseline` to `top` or `bottom`,specifying whether the baseline is inherited from the upper or lower row. The default value is `bottom`. The attribute has the type *ID*.
- `height` specifies the height of the array. This only influences the placing of the elements that follow the array. If the contents of the array cannot be placed within the specified margin, the system will display an error message. If `height` is not specified, the height of the array is dictated by the contents of the array. The attribute has the type *LENGTH*.
- `width` specifies the array's width. If the contents cannot be placed within the specified width, the system will display an error message. If none of the columns is stretchable, the width of the array will be the sum of the columns' widths, whether width is specified or not. If one or more columns are stretchable, the array is given the specified width. The attribute has the type *LENGTH*.
- `indent` specifies extra indentation of the array. The attribute has the type *LENGTH*. This attribute cannot be used when the array appears in a canvas or if the array uses a named ruler.
- `pos` specifies the positioning of the array in a canvas. The attribute is nameless and has the type *POS*. You can only use it when the array appears in a canvas, and in that case it is nameless and mandatory. See also "Exact Positioning of Elements" in "Canvas."

Following the description of the `row` tag in the next section, you will find a number of examples of the use of arrays.

row

The parenthetical tag:

```
<row attributes>
    ...
<end row>
```

is used to specify rows. Often you will want to use the short form:

```
...;shortattributes
```

A row consists of elements placed next to each other. A row cannot contain repetitions and can contain only one condition.

- `align` is used to specify how the elements in the row are placed next to each other. The attribute can be given the following values:
  - `baseline` is used to specify that the elements in the row are placed in accordance with the baseline. A simple element's baseline—that is, texts, fields, and variables—is selected on the basis of the font (characters such as "w" and "t" are placed on the baseline, while "g," "j," and "q" extend a little below the baseline). Baseline for a stacking tag is decided by this tag's baseline attribute. If no value is specified for align, baseline is standard.
  - `top` is used to specify that the elements should be placed in such a way that the upper edge of the elements are aligned.

- `center` is used to specify that the elements should be placed in such a way that the middles of the elements are aligned.
- `bottom` is used to specify that the elements should be placed in such a way that the lower edges of the elements are aligned.

The `align` attribute has the type *ID* and is nameless in short as well as in long form.

- `height` specifies the height of the row. This only influences the placing of subsequent rows. If the contents of the row cannot be placed within the specified margin, the system will display an error message. If `height` is not specified, the height of the row is dictated by the contents of the row. The attribute has the type *LENGTH*.

skip

A previous section described how you can use the `skip` tag in stacking tags. You can also use the `skip` tag to define the distance between rows. To do this, the tag must be the only element in a row.

Note: A row that only contains a `skip` tag is not included when checking the number of columns in an array.

Examples

The following array:

```
{
    "REFERENCE"      .Reference:width=5cm;
    "RECEIVER".Receiver:width=5cm;
}
```

results in the following printout with that contains two rows and two columns:

REFERENCE Peter Smith
RECEIVER Allan Thompson

Note: It is superfluous to set the width of both fields because the width of the column is derived from the widest element. The example, therefore, shows how to set the width of the column, instead.

The elements in rows can also be stacking tags. For example:

```
{
    <stack baseline=bottom>
        "CUSTOMER"
        "REFERENCE"
    <end stack>
    .Reference:width=5cm;
    <skip 2mm>;
    "RECEIVER".Receiver:width=5cm;
}
```

which results in the following printout:

```
CUSTOMER
REFERENCE  Peter Smith

RECEIVER    Allan Thompson
```

As you can see, the printout contains two rows: the first element in the first row is a stack that has two elements, which therefore span across two lines.

Finally, arrays can occur in rows:

```
{
    {
        "CUSTOMER NO."      .ThePaymentCustomer:width=5cm;
        "REFERENCE"   .Reference:width=5cm;
        "RECEIVER"    .Receiver:width=5cm;
    }
    {
        "ORDER NO."   .OrderNumber:width=5cm:left;
        <skip 5pt>;
        "INVOICE DATE"      .InvoiceDate:width=5cm;
    }
    ;:top     -- end row with two arrays
}
```

which results in the following printout

```
CUSTOMER NO 203108                    ORDER NO.    2060001
REFERENCE    Peter Smith
RECEIVER     Allan Thompson           INVOICE DATE 06-04-04
```

This representation is often used in printout headers (for example, at the top of an invoice) where the information is represented in two columns, and information is grouped by inserting `<skip 5pt>` between the elements. You achieve two columns with information by inserting an array with one row (stretching from line 2 to 12). This row contains two array elements, each of which represents a column. Thus the first column is defined between lines 2 and 6, and the other column is defined between lines 7 and 11.

Rows in Stacking Tags

```
{
    <conditional EUSalesVar>
        "TAX NO."     PayerTaxNumber;
    <end conditional>
    "CUSTOMER NO."   .PaymentCustomer;
}
```

Horizontal Lines

Just as in stacking tags, you can specify horizontal lines in arrays. You can use these as sum or header lines, or in combination with vertical lines to make tables. The syntax for horizontal lines (`hline`) is as follows:

```
<hline attributes>
```

In arrays the tags have the following attributes:

- `columns` specifies how many columns the line is to stretch across. `columns` has the type *INTEGER* and is nameless. If you do not specify a `span` attribute, one of the following things happens:
  - If the `hline` tag appears as the only element in the row, the line will stretch across all columns.
  - If the `hline` tag appears with other elements in the row, the default value for `columns` is `1`.
- `multi` specifies how many lines you want. You might, for example, set `multi` to `2` if you want to make a double line under a sum. The attribute has the type INTEGER. If it is not specified, a single line is drawn.
- `left` and `right`. Horizontal space is placed between the columns to ensure a nice array layout. By using these attributes, you can specify whether the line is to underline this extra space to the left or right of the columns across which the line is stretched. The attribute has the type *BOOLEAN*. The default value is `true`—that is, if they are not specified, the line will stretch across the space between the columns to the left as well as to the right.

> **Note:** Because all columns have extra space on the left and on the right, the specification of left+ or right+ means that half of the distance between the columns is underlined. The first column only has extra space on the left if a column separator has been specified before the first column (see “Column Separators”). Similarly, the last column only has extra space on the right if a column separator has been specified after the last column.

Example

```
{
    <hline columns=3>;
    <conditional ItemSalesOut script="Item Sales">
    "TOTAL ITEM SALES"      .Currency .ItemSumCurrency;
    <end conditional>
}
```

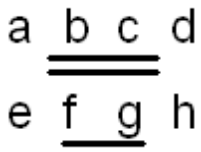
Note that both rows contain three elements, the first due to the `span` attribute. You could also leave out the `column` attribute.

The following example illustrates the use of the other attributes:

```
{
    "a" "b" "c" "d";
    "" <hline columns=2 multi=2>  "" ;
    "e" "f" "g" "h";
```

```
    "" <hline columns=2 left- right-> "" ;
}
```

which results in the following printout (enlarged to illustrate the effect of `left-` and `right-`):



Note that the bottom line is shorter than the double line because of the use of `left-` and `right-`.

Rulers—Column Definitions

A *ruler* specifies how the columns of an array should behave—how wide they should be and how they should be separated. To begin, we will explain how a ruler is used to specify column widths.

> **Note:** Rulers correspond to tab stops in word processors. However, rulers have more options.
>
> A column is viewed as a tag that only has a short form.

A ruler value is given as:

```
[columndefinition]
```

where `columndefinition` is a number of columns separated by `columnseparators` (defined in “Column Separators”). Furthermore, a `columndefinition` can start and end with a column separator. A *column* is written as:

```
[columnattributes]
```

A ruler value should contain as many columns as the array that it describes.

This means that a ruler value without column separators has the following format:

```
[[columnattributes][columnattributes] ... [columnattributes]]
```

Every occurrence of `[columnattributes]` defines the behavior of one column. The following column attributes can be provided:

- `stretch` indicates whether the column should stretch. If the attribute is not provided, the column will not stretch. If at least one column in an array is allowed to stretch, the whole array will stretch to fill the surrounding space. The extra space will be distributed evenly among the stretchable columns. The attribute has the type *BOOLEAN*.
- `width` specifies the minimum width of the column. The attribute has the type *LENGTH* and is nameless.

Example

Consider:

```
<island width=10cm>
    {:ruler=[[][stretch+]]
        "ORDER NO."   .OrderNumber;
        "INVOICE DATE"      .InvoiceDate;
        "VERSION NO." .VersionNumber;
    }
```

```
<end island>
```

The ruler has two columns, just as the three rows in the array. The first column has no specifications for the column containing the fixed text. Thus the width of the column will only be determined by its contents, and it will therefore be made wide enough to contain the three fixed texts. The `stretch` attribute is specified for the other column. This means that the array will stretch across the space available in the island: 10 cm. Therefore, the second column will be 10 cm wide minus the width of the widest text line (probably “INVOICE DATE”) and minus any column spacing.

When you prepare tables like this you should consider carefully which fields should be assigned any additional space. Usually, fields and variables that contain text must stretch, because you cannot predict their width.

Naming Rulers

You often want the columns of two arrays to have the same width. This is done in MPL by letting the two arrays share the same ruler—the ruler is given a name, and the two arrays refer to the name of the ruler definition. Ruler definitions can occur in the beginning of the following stacking tags: `frontpage`, `paper`, `header`, `footer`, `conditional`, `repeating`, `island`, `stack`, and `span`. The scope of the ruler is explained in “Ruler Scope.”

To name a ruler, the following tag is used:

```
<ruler attributes>
```

The tag has two attributes:

- `name` specifies the name by which the ruler should be known. The attribute has the type *ID* and is mandatory.
- `value` specifies the ruler that is to be named. The attribute value must be a ruler. The attribute is nameless and mandatory.

The attribute can also have the type *ID*, in which case it must be the name of a ruler. This attribute type can be used to assign a new name to a ruler, but it is not particularly useful.

Example

Consider

```
<ruler common [[][]]>
{:common
       "ORDER NO.":fontsize=12:bold+
       .OrderNumber:width=5cm:left:fontsize=12:bold+;
}
PaymentAddress1
PaymentAddress2
PaymentAddress3
{:common
    "CUSTOMER NO."   .PaymentCustomer:width=5cm;
    "INVOICE DATE"   .InvoiceDate:width=5cm;
}
```

If you only look at the first five lines of this fragment, the ruler will seem superfluous (because no column attributes have been specified, this corresponds to not specifying a ruler). The ruler's purpose is exclusively to ensure that the two arrays are formatted with the same column definition. This ensures that `.PaymentCustomer` and `.InvoiceDate` will appear in the same column as the order number:

Subrulers

You might want two arrays to share certain (but not all) columns. For example, the sum amount on an invoice is to be placed in the same column as the amounts on the invoice lines, but the information on the sum lines does not have to be placed in the same column as the information on the invoice lines. For this purpose a subruler is used. With a subruler you can take a selection of columns from a ruler and group other columns into one column.

This might sound complicated but, for example, the definition:

```
<subruler new [[2][3:4]] parent=old>
```

will create a ruler with the name "new" consisting of two columns. The first column is the second column from a ruler named "old" and the second column in "new" consists of a grouping of the third and fourth column in the ruler "old."

Subruler definitions can appear in the beginning of the following stacked tags: `frontpage`, `paper`, `header`, `footer`, `conditional`, `repeating`, `island`, `stack`,and `span`.

The syntax for subruler is:

```
<subruler attributes>
```

The tag has three attributes:

- `name` specifies the name that you want to give to the subruler. The attribute has the type *ID* and is nameless and mandatory.
- `value` is the subruler itself. The attribute value must be a subruler. The attribute has the type *ID* and is nameless and mandatory.
- `parent` specifies the name of the ruler from which you want to define a subruler.

  The ruler must be recognized and can also be the name of a subruler. The attribute has the type *ID* and is nameless and mandatory.

A subruler value is very similar to a ruler value and is specified as:

```
[subcolumndefinition]
```

where the `subcolumndefinition` is a row of subcolumns divided by column separators. Furthermore a `subcolumndefinition` can be started and ended by a column separator. A subcolumn is written as:

```
[Range]
```

or

```
[INTEGER]
```

A range is specified as *INTEGER*:*INTEGER*. A range n_1:n_2 specifies that the column must correspond to the columns n_1 to n_2 (both included). If only *INTEGER n* is specified, this is actually an abbreviation of *n*:*n*.

As for columns, subcolumns can be viewed as tags that only have a short form: [shortattributes]. The only attribute is a range that has the type *RANGE*. This type is either *INTEGER* or *INTEGER*:*INTEGER*. The attribute interval is nameless and, therefore, you will usually only write a subcolumn as [*INTEGER*] or [*INTEGER*:*INTEGER*].

Subcolumns must refer to the parent ruler's columns from left to right, and they cannot overlap.

Iteration: If [n1:n2] [m1:m2] is a part of the subruler, n1≥n2 and m1≥m2.

Examples of legal subrulers:

```
[[1][3][5]]
[[1:2][4:5]]
[[1:3][4][5]]
```

Examples of illegal subrulers:

```
[[5][3][1]]   -- Columns not ordered
[[2:1][4:5]] -- Columns not ordered [[1:3][3:4][5]]  -- Two columns are
overlapping
```

Example

The following example is taken from a simplified layout to the layout "Print Invoice," but many details have been left out. We define the following rulers:

```
<ruler lines [[1cm][stretch+][15mm]]>
<subruler primarylines[[1:2][3]] parent=lines>
<subruler secondarylines [[2][3]] parent=lines>
```

The first line defines all of the columns on an invoice line. Consider the ruler "lines" as defining three tab stops. The first column is 1 cm wide and is used as the indent marker that is to be inserted before secondary lines ("Discounts" and "Extra Text" in this example). The second column is the text column. This should be as wide as possible and has therefore been made stretchable. The last column contains amounts and is set to 15 mm.

The second line defines primary lines. Here the text must start from the very left and the first column therefore consists of the first and second column from the ruler "lines." The second column is the amount column, which should contain the price.

The third line defines secondary lines that are related to primary lines. This ruler is similar to the ruler "primarylines," but in this case the text column is indented as it only consists of column 2 from the ruler "lines."

The rulers can now be used as follows:

```
<repeating InvoiceLine script="InvoiceLineBlock">
    <conditional PricesOut>
        {:primarylines
            .ExternalItemText CorrectedPriceWithoutDiscount;
        }
```

```
    <end conditional>
    <conditional DiscountOut>
        {:secondarylines
            .DiscountText CorrectedDiscountCurrency;
        }
    <end conditional>
    <repeating InvoiceBOMLine script="BOMLineBlock">
        <conditional PricesOut>
        {:secondarylines
            InvoiceBOMLine.ExternalItemText
            CorrectedPriceWithoutDiscount:zerosuppression+;
        }
        <end conditional>
    <end repeating>
<end repeating>
```

which results in the following printout

Special Spring Offer	0,00
KP Keyboard	1.200,00
Mouse, Standard X-4	200,00
Ixo 19" Monitor	1.800,00
Zitech Pentium 166 MHz	3.800,00
Ixo 19" Monitor	1.800,00
May discount	-72,00

“Spanning Columns” contains a description of an alternative way of achieving this formatting. You can use whichever of the two methods you prefer the most.

Ruler Scope

The name of the ruler is recognized in all of the stacking tags in which it is defined. However, it will not be recognized in a tag where the left or right edge has been indented in relation to the stacking tag in which the ruler is defined. Such indentation is inserted due to the use of the `indent` attribute in a parenthetical tag or the use of `leftmargin` or `rightmargin` in islands.

The following example illustrates the ruler scope:

```
-- my_ruler is not recognized here
<stack>
    <ruler my_ruler [[][stretch+]]>
    -- my_ruler is recognized here
    <island>
        -- my_ruler is recognized here
    <end island>
    <island leftmargin=1cm>
        -- my_ruler is not recognized here as the island’s
        -- contents are indented
    <end island>
```

```
    <stack>
        -- my_ruler is recognized here
    <end stack>
    <stack indent=1cm>
        -- my_ruler is not recognized here as the stack
        -- is indented
    <end stack>
<end stack>
-- my_ruler is not recognized here
```

Column Separators

You can insert column separators between columns in rulers or subrulers. A column separator specifies what should be inserted between the columns.

A column separator is a tag; you can only use the `vline`, `text`, and `text2` tags as column separators. The `vline` tag is described later in this manual; it is used more often as a column separator than as an independent tag. The `text2` tag is described in “Alternative Text Tag.”

The following fragment constitutes a legal ruler:

```
[[]<text "Hello">[]<vline>[]]
```

This ruler will write the text “Hello” in each row between the first and the second column and insert a vertical line between the second and the third column. You will often use the short form of these tags, and the ruler will look as follows:

```
[[]"Hello"[]|[]]
```

Note that the short form of `vline` is just a vertical line: `|`.

The most common characters in column separators are dividing characters (“-,” “/,” and so on). Furthermore, you can also insert multiple spaces using this tag to obtain extra space between columns. To make this easier, you can simply specify a width:

```
[]2cm[]]
```

This ensures a column separation of 2 cm.

Thus you can specify the following column separators:

- Vertical lines are used to create tables. A vertical line is written as `<vline` *attributes*`>` or with the short form “|*shortattributes*.”

  The only attribute to the `vline` tag is `justification` (see also “vline”). The attribute is rarely used when `vline` is used as the column separator.

- Strings that should be repeated in all columns are specified using the `text` or `text2` tags.

- The distance between columns. If you want to change the distance between all columns, you should change the length constant `InterColumnSpacing` instead (see “Length Constants” in “Advanced MPL”).

Subrulers do not inherit column separators from the parent ruler. If, for example, you want to repeat vertical lines, you must specify these again in the subruler.

Example

If you want to create a simple table, it can be done as follows:

```
<ruler tableline[|[]|[]|]>
{:tableline
    <hline>;
    "Field":bold+    "Selection Criteria":bold+;
    <hline>;
    "Delivery Mode"  DeliveryModeVar; "Delivery Terms" DeliveryTermsVar;
    "Carrier" TransporterVar;
    "Consignment Type"      ConsignmentTypeVar;
    <hline>;
}
```

which results in the following printout:

Field	Selection Criteria
Delivery Mode	Mail
Delivery Terms	FOB
Carrier	
Consignment Type	

The following is an example of the use of `subruler`:

```
<ruler tableline[|[]|[2cm]"-"[2cm]|]>
<subruler headingline[|[1]|[2:3]|] parent=tableline>
{:headingline
    <hline>;
    "Field":bold+ "Selection Criteria":bold+:center;
}
{:tableline
    <hline>;
    "Invoice No."    FromInvoiceNumber:right ToInvoiceNumber:left;
    "Ordrenr."FromOrderNumber:right ToOrderNumber:left;
    "Company No."    FromCompanyNumberVar:right ToCompanyNumberVar:left;
    "Bill to Customer No." FromPaymentCustomer:right ToPaymentCustomer:left;
    <hline>;
}
```

This fragment also shows how you can use a text as a column separator. The result is the following printout:

Field	Selection Criteria
Invoice No.	200001 - 200007
Ordrenr.	0 - 0
Company No.	-
Bill to Customer No.	-

Spanning Columns

It is often useful to make an element in an array stretch across several columns, for example, if a heading should span across two columns. The `span` tag is a parenthetical tag that can contain only one element. This element can be a parenthetical tag, although not a row. You use this tag to specify the number of columns across which the contents should span. The syntax is as follows:

```
<span attributes>
    ...
<end span>
```

The contents of a `span` tag is one element (possibly preceded by definitions of rulers, defaults, and length constants). Often you will use the short form, that is, surrounding parentheses:

```
(
    ...
)shortattributes
```

The tag has the following attribute:

- `columns` is used to specify the number of columns across which the contents should span. The tag has the type *INTEGER* and is mandatory and nameless both in the short and the long forms.

Example

First we will consider an example based on the print layout for “Print Balance”:

```
{:[[1cm][stretch+][][15mm][15mm][15mm][15mm]]
    "" "" "" ("Period":center):2 ("Year to Date":center):2; ("Account"):2 "Tax
    Code" "Debit":center "Credit":center "Debit":center "Credit":center;
    <repeating cursor=ACCOUNTSTRUCTURELINE script="B_BLOCK">
        <conditional variable=ShowAccount>
            AccountNoField .Description VATCodeField
            MovementDebitField:right
            MovementCreditField:right
            BalanceDebitField:right
            BalanceCreditField:right;
        <end conditional>
    <end repeating>
}
```

Here you should pay attention to the fact that the `span` tag is used twice on line 2. The heading “Period” spans across two columns: The debit and credit columns for ”Period.” Similarly, the heading “Year to Date” spans across a debit column and a credit column. On line 3, the heading “Account” spans across two columns (account number and account description). The result is as follows:

Account		Tax Code	Period Debit	Period Credit	Year to Date Debit	Year to Date Credit
	REVENUE :					
	External Revenues :					
1010	Licenses		0,00		0,00	
1020	Update Contracts		0,00		0,00	
1030	Services		0,00			189.740,31
1040	3rd. Party Products & Services		0,00		0,00	
1510	License fee Parent Company		0,00			7.000,00
1520	CSS fee Parent Company		0,00		0,00	
2010	Licenses & Update Contr.		0,00		0,00	
2020	Oracle Licenses		0,00		0,00	
2030	Services		0,00		0,00	
2040	3rd. Party Products & Services		0,00		16.495,38	

We will now return to the example from “Subrulers.” Here we define the ruler:

```
<ruler invoicelines [[1cm][stretch+][15mm]]>
```

The rulers can now be used as follows:

```
{:invoicelines
    <repeating Invoiceline script="Invoicelineblock">
        <conditional PricesOut script="PricesOut">
            (.ExternalItemText):2
            CorrectedPriceWithoutDiscount;
        <end conditional>
        <conditional DiscountOut script="DiscountOut">
            "" .Discounttext CorrectedDiscountCurrency;
        <end conditional>
        <repeating InvoiceBOMLine script="BOMLineBlock">
            <conditional PricesOut>
                {:secondarylines
                    InvoiceBOMLine.ExternalItemText
                    CorrectedPriceWithoutDiscount:zerosuppression+;
            <end conditional>
        <end repeating>
    <end repeating>
}
```

The resulting printout matches the printout on the figure above, but without the use of subrulers.

vline

We are already familiar with the `vline` tag and how it is used as a column separator. Vertical lines can also be used as an element in rows:

```
<vline attributes>
```

The short form of the `vline` tag is simply:

```
| shortattributes
```

The tag has a single attribute:

- `justification` is nameless in both the long and the short forms and has the type *ID*. It can take the values `left`, `center`, and `right`. If justification is not specified, the default value is `left`.

It is not an easy task to use the `vline` tag to create tables; it is recommended to use the tag as column separator instead.

Example

The program fragment:

```
{
    <hline 5 left- right->;
    |:left "1" |:center "2" |:right;
    <hline 5 left- right->;
    "" |:left "3" |:right "";
    "" <hline 3 left- right-> "";
}
```

results in the following printout:

Notice how `left-` and `right-` have been used to prevent the horizontal lines from stretching below the column separators that are inserted before and after the vertical lines.

Printout Example: Time Sheets

In this section we will create a layout for the printout “Print Time Sheet.” The section describes the practical procedures involved in the creation of a new layout.

Getting Started

Export the structure layout for the printout “Print Time Sheet” from the window Print Layout in the Maconomy client. The result is a valid MPL definition. The structure is shown below:

```
<mpl 1>
<layout title="Standard"
          print="Print_TimeSheets"
          originallayout="Standard">
<page "A4">
<frontpage>
<end frontpage>
<paper>
   <header onfirstpage+ height=61pt>
      <end header>
      <stack>
         <repeating TimeSheetHeader
                script="S_TimeSheetHeader">
         <repeating TimeSheetLine
                script="S_TimeSheetLine">
            <header>
            <end header>
         <end repeating>
         <stack>
         <end stack>
      <end repeating>
   <end stack>
<end paper>
```

The structure layout defines the structure of the original layout, that is, the allowed structure of new layouts. For more information, see “Print Structure.” The structure layout is therefore a good starting point for the design of your own layouts.

To see which fields and variables you are able to access for each printout in Maconomy, use the manual “Variables and Cursors in Printouts,” which is updated for each application version of Maconomy. If you want to see *where* and *how* the fields and variables are used, it is a good idea to export the original layout.

Create the New Layout

The Header

We want to create a layout that can be printed on A4 paper in landscape orientation. We shall name the layout “Landscape, standard.” The header of the MPL definition will look as follows:

```
-- Layout:
--     "Landscape, standard" to be used by consultants
--
-- Design:
-- Peter Smith (May 5 2004)
--
-- Based on structure layout version:
-- Application version: Maconomy W 8.0
-- Application date (yyyy/mm/dd): 2004/02/24
<mpl 2>
<layout title="Landscape, standard"
       print="Print_TimeSheets"
       originallayout="Standard">
<page "A4" landscape>
```

As you can see, a comment has been included in the MPL header, making it easy to pinpoint:

- The layout designer
- The purpose of the layout
- The Maconomy version for which the layout has been designed

The information that you want to include in the comments is, of course, up to you.

The Front Page

The front page of the print layout “Print Time Sheet” has two purposes:

1. Inform about who is printing the time sheet
2. Show the selection criteria specified for the printout

The information about the printout is bundled in an island that we have named “Time Sheets.” This island contains an array with two columns where the first column contains the user name and company name, and the other contains the time and date. Both columns have been stretched so that they take up the entire space and become equally sized (ensuring that we have room for the contents of the fields). Furthermore, the time and date have been right-adjusted.

The selection criteria information has also been placed in an island that has been made smaller than the paper. The array in this island has three columns: A Text column, a From column, and a To column. The last two columns are separated by hyphens (specified as ‘-’ instead of “-” in the ruler; the difference is explained in “Alternative Text Tag” in “Advanced MPL”). The text column cannot stretch, and the two other columns share the extra space.

The first row is aligned to this ruler, but the next row is not, because only one superior is specified and not a range. This is done by letting `SeniorEmployeeVar` stretch across two columns.

The fifth row is different from the rest, as the “From” and “To” information is a date consisting of week number, slash (“/”), and year. For each of these pieces of information we specify an embedded array consisting of one row with the date.

```
<frontpage>
    -- Header
    <island "Time Sheets" leftmargin=5mm rightmargin=5mm
              bottommargin=2mm>
        {:[[stretch+][stretch+]]
            CompanyName Time:right;
            UserName  TodaysDate:right;
        }
    <end island>
    <skip 1cm>
    -- Selection Criteria
    <island width=14cm
        leftmargin=2cm rightmargin=2cm
        topmargin=2mm bottommargin=2mm>
        {:[[][stretch+]'-'[stretch+]]
            "Employee No." FirstEmployeeNumber
            LastEmployeeNumber;
            "Superior"(SeniorEmployeeVar):2;
            "Secretary"      (SecretaryEmployeeVar):2;
            "Submitted"      (SubmittedVar):2;
            "Week No."{FirstWeek '/' FirstYear;}
                   {LastWeek '/' LastYear;};
            "Layout Name" (LayoutName):2;
        }
    <end island>
<end frontpage>
```

In the preceding print definition we have used span and embedded arrays in the rows that did not fit into the overall ruler. This is easier to understand if there are only a few rows—if there had been more rows, the easiest and most elegant solution would be to use named rulers and subrulers:

```
<frontpage>
    -- Header
        ...
    -- Selection Criteria
    <island width=14cm
        leftmargin=2cm rightmargin=2cm
        topmargin=2mm bottommargin=2mm>
        <ruler DateRangeLine
            [[][stretch+]'/'[stretch+]'-'[stretch+]'/'[stretch+]]>
        <subruler RangeLine [[1][2:3]'-'[4:5]] parent=DateRangeLine>
```

```
       <subruler SimpleLine [[1][2:3]] parent=RangeLine>
       {:RangeLine
          "Employee No."   FirstEmployeeNumber LastEmployeeNumber;
       }
       {:SimpleLine
          "Superior"SeniorEmployeeVar;
          "Secretary" SecretaryEmployeeVar;
          "Submitted" SubmittedVar;
       }
       {:DateRangeLine
          "Week No."FirstWeek FirstYear LastWeek LastYear;
       }
       {:SimpleLine
          "Layout Name"    LayoutName;
       }
   <end island>
<end frontpage>
```

If you want to include more date range lines, this definition also ensures the alignment of the week and year fields.

The Paper Content

The Layout for Each Time Sheet Line

The inner repeating block from the structure layout looks as follows:

```
<repeating cursor=TIMESHEETLINE
       script="S_TIMESHEETLINE">
<end repeating>
```

The contents of this block will be printed for each line on a time sheet. The desired information here is job number, job name, activity number, entry description, number of hours for each of the 7 days of the week, and an hours total. This information should be presented in one row so that any extra space is distributed to the two text fields (job name and entry description). As job numbers can be quite long, 1.5 cm has been set aside for this column.

```
<repeating cursor=TIMESHEETLINE script="S_TIMESHEETLINE">
   {:[[1.5cm][stretch+][][stretch+][][][][][][][][]]
       .JobNumber JobName .ActivityNumber .TextofEntry
       .NumberOfDay1 .NumberOfDay2 .NumberOfDay3
       .NumberOfDay4 .NumberOfDay5 .NumberOfDay6
       .NumberOfDay7 TotalLine;
   }
<end repeating>
```

The Layout for Each Time Sheet

Information about the time sheet, the column headings, and the totals should be printed on each time sheet.

The headings are placed in the same columns as the fields. The headings and the fields should therefore share a ruler. This is achieved by letting the array “flow” out of the inner repeating tag. By inserting a line after the headings, we get the following representation:

```
<repeating cursor=TIMESHEETHEADER script="S_TIMESHEETHEADER">
    {:[[1.5cm][stretch+][][stretch+][][][][][][][][]]
        "Job No."     "Job Name"    "Activity No."      "Description"
        "Mon"  "Tue"  "Wed"  "Thu"  "Fri"  "Sat"  "Sun"  "Total";
        <hline 12>;
        <repeating cursor=TIMESHEETLINE script="S_TIMESHEETLINE">
            ...
        <end repeating>
    }
<end repeating>
```

The information about the time sheet should include employee number and name, period, whether the time sheet has been submitted, and if so, when:

```
{:[][][stretch+]]
    "Employee".EmployeeNumber .EmployeeName;
    "Period"  .PeriodStart:width=1.5cm   .PeriodEnd:width=1.5cm;
    "Submitted"      .Submitted .DateSubmitted;
}
```

Finally, the totals are made part of the array that surrounds the time sheet lines. Before the totals we insert a line, and after the totals we insert a little extra space to separate the information from the information for the next time sheet, if any:

```
<hline 12>; "" "" "" ""
SumDay1 SumDay2 SumDay3 SumDay4 SumDay5 SumDay6 SumDay7 GrandTotal;
<skip 5mm>;
```

The Entire Layout

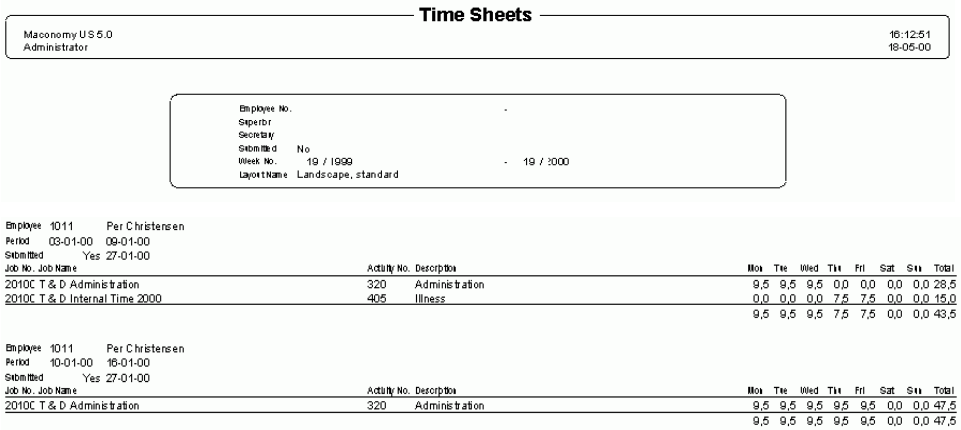
If you put the fragments described above together, you get the following layout:

```
-- Layout:
--     "Landscape, standard" to be used by consultants
--
-- Design:
-- Peter Smith (May 5 2004)
--
-- Based on structure layout version:
-- Application version: Maconomy W 8.0
-- Application date (yyyy/mm/dd): 2004/02/24
```

```
<mpl 2>
<layout title="Landscape, standard"
       print="Print_TimeSheets"
       originallayout="Standard">
<page "A4" landscape>
<frontpage>
   -- Header
   <island "Time Sheets" leftmargin=5mm rightmargin=5mm
             bottommargin=2mm>
      {:[[stretch+][stretch+]]
         CompanyName Time:right;
      }
      leftmargin=2cm rightmargin=2cm
      topmargin=2mm bottommargin=2mm>
      {:[[][stretch+]'-'[stretch+]]
         UserName  TodaysDate:right;
   <end island>
   <skip 1cm>
   -- Selection Criteria
   <island width=14cm
      "Employee No." FirstEmployeeNumber
      LastEmployeeNumber;
      "Superior"    (SeniorEmployeeVar):2;
      "Secretary"   (SecretaryEmployeeVar):2;
      "Submitted"   (SubmittedVar):2;
      "Week No."    {FirstWeek '/' FirstYear;}
      {LastWeek '/' LastYear;};
      "Layout Name" (LayoutName):2;
      }
   <end island>
<end frontpage>
<paper>
   <repeating cursor=TIMESHEETHEADER
             script="S_TIMESHEETHEADER">
      {:[[][][stretch+]]
         "Employee".EmployeeNumber     .EmployeeName;
         "Period"  .PeriodStart:width=1.5cm   .PeriodEnd:width=1.5cm;
         "Submitted"      .Submitted    .DateSubmitted;
      }
      {:[[][stretch+][][stretch+][][][][][][][][]]
```

```
        "Job No." "Job Name"    "Activity No."      "Description"     "Mon"
           "Tue"  "Wed"  "Thu"  "Fri"  "Sat"  "Sun"  "Total";
        <hline 12>;
        <repeating cursor=TIMESHEETLINE
              script="S_TIMESHEETLINE">
           .JobNumber JobName .ActivityNumber .TextOfEntry
           .NumberOfDay1 .NumberOfDay2 .NumberOfDay3 .NumberOfDay4
           .NumberOfDay5 .NumberOfDay6 .NumberOfDay7 TotalLine;
        <end repeating>
        <hline 12>;
        "" "" "" ""
        SumDay1 SumDay2 SumDay3 SumDay4 SumDay5 SumDay6 SumDay7
        GrandTotal;
        <skip 5mm>;
    }
  <end repeating>
<end paper>
```

Below both the front page for the printout and the following page are shown.



Time Sheets

Maconomy US 5.0
Administrator

16:12:51
18-05-00

Employee No. -
Superbr.
Secretary
Submitted No
Week No. 19 / 1999 - 19 / 2000
Layout Name Landscape, standard

Employee 1011 Per Christensen
Period 03-01-00 09-01-00
Submitted Yes 27-01-00

Job No.	Job Name	Activity No.	Description	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Total
2010C	T & D Administration	320	Administration	9,5	9,5	9,5	0,0	0,0	0,0	0,0	28,5
2010C	T & D Internal Time 2000	405	Illness	0,0	0,0	0,0	7,5	7,5	0,0	0,0	15,0
				9,5	9,5	9,5	7,5	7,5	0,0	0,0	43,5

Employee 1011 Per Christensen
Period 10-01-00 16-01-00
Submitted Yes 27-01-00

Job No.	Job Name	Activity No.	Description	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Total
2010C	T & D Administration	320	Administration	9,5	9,5	9,5	9,5	9,5	0,0	0,0	47,5
				9,5	9,5	9,5	9,5	9,5	0,0	0,0	47,5

Basic Tags Continued

Having analyzed the Time Sheets example, we can continue our tour through the basic MPL tags.

Headers and Footers

Headers and footers are used to specify print elements that must be printed at the top or bottom part of each page. They are often used to specify print information (page number, date, the name of the printout), information about the contents (for example, which employee's time sheets are being printed on a given page), and column headers for repetitions.

In MPL, you can specify several types of headers and footers:

- With page headers, you can specify text elements that you want printed in the top part of each page.
- As with page headers, you can use page footers to specify text elements that you want printed in the bottom part of each page.
- You can specify block headers in repetitions, conditions, and stacking tags. These headers contain text elements that you want printed in the top part of each page as long as the block is printed.
- As with block headers, you can use block footers to specify text elements that you want printed in the bottom part of each page as long as the block is printed.

If you specify both a page and a block header/footer, both headers/footers will be printed. The usual rules of scope apply, meaning that if a given print element results in a page break, the block header/footer that belongs to the closest surrounding block (and only that header/ footer) will be printed.

Page Headers and Footers

You can specify page headers and footers anywhere in the page definition, that is, within the parenthetical tag `<paper> ... <end paper>` and not within any other parenthetical tags. You should always specify these headers and footers after any possible definitions (ruler statements and so on) on the page; they can share rulers with the contents of the page.

Headers and footers function as stacking tags; however, they cannot contain other headers or footers, `newpage` tags, repetitions, conditions, or stacks with scripts.

The contents of a page header will be printed in the top part of each page. Using an attribute, you can control whether a header should be printed on the first page. Similarly, the contents of a page footer will be printed in the bottom part of each page, and you can use an attribute to control whether a footer should be printed on the last page.

Headers and footers are often used to print out information about the contents of the printout, the date, and possibly page numbers (available as a variable in all prints).

Block Headers and Footers

You can specify block headers and footers anywhere in repetitions, conditions, and stacks. Headers/footers will be printed in the top/bottom part of the page if a page break is triggered while printing the contents of the repeating/conditional tag or stack.

If you have specified a header both on the page and in a block, both headers will be printed. The page header will be printed first, followed by the block header. Similarly, a block footer will be printed over the page footer.

You can specify headers/footers in several nested blocks. If an element specified inside such nested blocks triggers a page break, the header/footer in the closest surrounding block with a header/footer is printed.

header

Headers are specified using the parenthetical tag:

```
<header attributes>
    ...
<end header>
```

The `header` tag has the following attributes:

- `onfirstpage` is used only in page headers and specifies whether the header should be printed on the first page. The attribute has the type *BOOLEAN* and the default value is `false`.
- `atstart` is used in repetitions/conditions and stacks and specifies whether the header should be printed immediately before the printing of the actual block is initiated or only at page breaks. The attribute has the type *BOOLEAN* and the default value is `false`.
- `script` is used to specify the name of a script (program) that is executed each time that the header is printed. The attribute has the type *STRING*. The use of scripts must correspond to their use in the original layout. For more information, see “Print Structure.”
- `height` specifies the height of the header and is therefore used to determine where the text after the header should begin. If the contents are higher than the specified height, an error message is displayed. The attribute has the type *LENGTH*.

In addition to the height of the header, the length constant `HeaderSkip` also determines the distance between the header and the header text.

footer

Footers are specified using the parenthetical tag:

```
<footer attributes>
    ...
<end footer>
```

The footer tag has the following attributes:

- `onlastpage` is used only in page footers and specifies whether the footer should be printed on the last page. The attribute has the type *BOOLEAN*.
- `atend` is used in repetitions/conditions and stacks, and specifies whether the footer should be printed immediately after the block has been printed or only at page breaks. The attribute has the type *BOOLEAN*.
- `script` specifies the name of a script (program) that is executed each time the footer is printed. The attribute has the type *STRING*. The use of scripts must correspond to their use in the original layout. For more information, see “Print Structure.”
- `height` specifies the height of the footer and is therefore used to determine how far down the page the main text extends to. If the contents are higher than the specified height, an error message is displayed. The attribute has the type *LENGTH*.

This means that text is only printed until the bottom margin of the page plus the height of the footers plus the length constant `FooterSkip`.

- `pagebottom` is used in repetitions/conditions and stacks, and specifies whether the footer should be printed at the bottom of the page or follow the block. The attribute has type *BOOLEAN* and its default value is `true`.

Example

We will now attempt to use headers and footers in the time sheet layout described in the previous section.

Key information about the printout should be included in the page header:

```
<header onfirstpage+>
    <island "Time Sheets"
              leftmargin=5mm rightmargin=5mm bottommargin=2mm>
        {:[[stretch+][stretch+]] CompanyName TodaysDate:right;
        }
    <end island>
<end header>
```

You can then, for example, specify the page number in the page footer:

```
<footer onlastpage+>
    {:[[stretch+][][][][stretch+]]
        "" "-" PageNumber:center "-" "";
    }
<end footer>
```

The first and last columns are empty but will be stretched. This ensures that the three middle columns are centered.

As specified in the print definition in the last section, a line with column headings was printed before each list of time sheet lines. If such a list triggers a page break, it would be convenient if the line were repeated on each new page. This can be achieved by replacing the heading line with a header in the inner repeating block. By setting the attribute `atstart` to `true`, the heading is also printed before the first time sheet line:

```
<header atstart+>
    "Job No. ""Job Name"    "Activity No. "     "Description"
    "Mon"     "Tue"  "Wed"  "Thu"  "Fri"  "Sat"  "Sun"  "Total";
    <hline 12>;
<end header>
```

If a page break is triggered in the middle of a time sheet, we want the printout to show that the time sheet is continued on the next page.

In this printout, it is not possible to display subtotals. The variables SumDay1, SumDay2, and so on, are updated by the script S_TIMESHEETLINE. This script is executed for each record before the contents of the repeating block are printed. This means that .NumberOfDay1 is added to SumDay1 on each line before the contents are printed, and therefore, before the printout breaks to a new page. A footer based on SumDay1 will, therefore, display subtotals that include the first line from the following page.

This is achieved by inserting a footer in the inner repeating block:

```
<footer>
   ("Continued":right:italic+):12;
<end footer>
```

We do not use the `atend` attribute here because the footer should not be printed when the repeating block has terminated. With these changes, the `paper` part of the time sheet layout will look as follows:

```
<paper>
   <header onfirstpage+>
      <island "Time Sheets"
            leftmargin=5mm
            rightmargin=5mm
            bottommargin=2mm>
         {:[[stretch+][stretch+]]
            CompanyName TodaysDate:right;
         }
      <end island>
   <end header>
   <footer onlastpage+>
      {:[[stretch+][][][][stretch+]]
         "" "-" PageNumber "-" "";
   }
   <end footer>
   <repeating cursor=TIMESHEETHEADER
            script="S_TIMESHEETHEADER">
      {:[[][][stretch+]]
         "Employee".EmployeeNumber .EmployeeName;
         "Period"  .PeriodStart:width=1.5cm   .PeriodEnd:width=1.5cm;
         "Submitted"      .Submitted .DateSubmitted;
      }
      {:[[][stretch+][][stretch+][][][][][][][][]]
         <repeating cursor=TIMESHEETLINE
               script="S_TIMESHEETLINE">
            <header atstart+>
               "Job No." "Job Name"    "Activity No."      "Description" "Mon"
                  "Tue"  "Wed"  "Thu"  "Fri"  "Sat"  "Sun"  "Total";
```

```
                <hline 12>;
            <end header>
            .JobNumber JobName.ActivityNumber .TextOfEntry
            .NumberOfDay1 .NumberOfDay2 .NumberOfDay3 .NumberOfDay4
            .NumberOfDay5 .NumberOfDay6 .NumberOfDay7 TotalLine;
            <footer>
                ("Continued":right:italic+):12;
            <end footer>
        <end repeating>
        <hline 12>; "" "" "" ""
        SumDay1 SumDay2 SumDay3 SumDay4 SumDay5 SumDay6 SumDay7
        Grandtotal;
        <skip 5mm>;
    }
<end repeating>
<end paper>
```

The next page displays the result of including headers and footers in the definition. The printout displays the bottom of a page and the top of the following page.

Employee 1028 Sophie Nielsson
Period 24-01-00 30-01-00
Submitted Yes 05-04-00

Job No.	Job Name	Activity No.	Description	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Total
25252	Blue Mountain RP4	160	Consultancy	7,5	7,5	7,5	7,5	8,0	0,0	0,0	38,0
				7,5	7,5	7,5	7,5	8,0	0,0	0,0	38,0

Employee 1028 Sophie Nielsson
Period 31-01-00 06-02-00
Submitted Yes 05-04-00

Job No.	Job Name	Activity No.	Description	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Total
25252	Blue Mountain RP4	205	Requirement analysis	5,0	6,0	2,0	4,0	7,0	0,0	0,0	24,0
25252	Blue Mountain RP4	160	Consultancy	2,0	2,0	4,0	3,0	0,0	0,0	0,0	11,0

Continued

Time Sheets

Maconomy US 5.0 18-05-00

Job No.	Job Name	Activity No.	Description	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Total
25252	Blue Mountain RP4	330	Training, received internally	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
				7,0	8,0	6,0	7,0	7,0	0,0	0,0	35,0

Employee 1028 Sophie Nielsson
Period 07-02-00 13-02-00
Submitted Yes 05-04-00

Job No.	Job Name	Activity No.	Description	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Total
25252	Blue Mountain RP4	205	Requirement analysis	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
25252	Blue Mountain RP4	160	Consultancy	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
25252	Blue Mountain RP4	330	Training, received internally	7,5	7,0	8,0	7,0	8,0	0,0	0,0	37,5
				7,5	7,0	8,0	7,0	8,0	0,0	0,0	37,5

Canvas

This section describes the use of a canvas in MPL. A canvas is used for placing elements at exact positions on the paper.

Exact Positioning of Elements

One of the main advantages of using MPL is that the you do not need to worry about the exact position of each element, but can concentrate on the logic of the print definition. In some circumstances it is, however, necessary to position print elements at exact places on the paper. This can be useful if you want a printout to match a preprinted form, and even necessary if you want texts or fields to overlap. You use the parenthetical tag `canvas` to achieve this. You can also use this tag can if you want to draw lines that are not horizontal or vertical.

canvas

While stacking tags can be illustrated as a sheet of ruled paper where the text is positioned horizontally underneath each other, a canvas corresponds to the canvas of a painter, where the element can be positioned freely. The syntax for canvases is:

```
<canvas attributes>
    ...
<end canvas>
```

Every element in a canvas must be assigned a `pos` attribute that specifies where the element is to be positioned. You specify the position of the element's top-right corner in relation to the top-left corner of the canvas. The canvas tag has the following attributes:

- `height` specifies the height of the canvas. The attribute has the type *LENGTH*. If the tag is specified, all elements in the canvas must observe the specified height. If not specified, the height of the canvas is calculated based on the contents.
- `width` specifies the width of the canvas. The attribute has the type *LENGTH*. If the tag is specified, all elements in the canvas must observe the specified width. If not specified, the width of the canvas is calculated based on the contents.
- `indent` specifies the indentation of the canvas. The attribute has the type *LENGTH*. The attribute cannot be used if the canvas is part of another canvas.
- `pos` specifies the position of a canvas in another canvas. The attribute has the type *POS*. When the attribute is used (that is, when the canvas is part of another canvas) it is nameless and mandatory.

If you place a canvas in a row, the baseline for the last element in the canvas will be the baseline for the canvas itself. Because the order of elements is not subject to any rules, you can specify the elements from which the baseline should be inherited.

Elements in a canvas can overlap each other.

Example

The following example illustrates the use of canvas:

```
{
    <canvas>
        "hello":(0mm,0mm)
        "world":(8mm,8mm)
    <end canvas>
    "!!";
}
```

which results in the following printout:

```
hell
      world !!
```

Note that baseline for the canvas is inherited from the last element (that is, “world”). “!!” is therefore positioned on the same line as “world.”

Lines

You can use a canvas to draw lines:

```
<line attributes>
```

The tag has the following attributes, both of which are mandatory. Both attributes have the type *POS*:

- `start` specifies the starting point of the line.
- `end` specifies the ending point of the line.

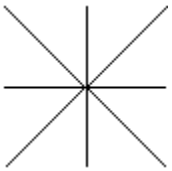
Lines can go in any direction and have any length (as long as they are within the canvas), and lines are allowed to cross each other or other elements.

Example

The following example illustrates the use of `line`:

```
<canvas width=2cm height=2cm>
   <line start=(0cm,0cm) end=(2cm,2cm)>
   <line start=(1cm,0cm) end=(1cm,2cm)>
   <line start=(2cm,0cm) end=(0cm,2cm)>
   <line start=(2cm,1cm) end=(0cm,1cm)>
<end canvas>
```

which results in the following printout



Custom Calculations

MPL 4 introduces expressions as a unified means of referring to the predefined environment values (that is, fields and variables) as well as calculating new ones. The newly calculated values can then be bound to named symbols—*vals* (that is, constant values) or *vars* (that is, mutable variables). *Vals* represent constants, and their values cannot be changed throughout the execution of the print. On the other hand, *vars* represent mutable variables and can be assigned new values using the `<assign>` tag.

To define a new *val*, you can use the `<val>` tag, for example:

```
<val currencySymbol {if currencyName = "USD" then "$"
                     else if currencyName = "EUR" then "€"
                     else if currencyName = "GBP" then "£"
                     else currencyName }>
```

the `currencySymbol` val represents a new value that can be used as any other predefined environment variable in the scope in which it is defined. So, for instance, you could use it to print out an amount like this:

```
"Billing price" billingPriceVar currencySymbol;
```

Vars are defined in very similar ways to vals, but in contrast to them, vars can be assigned new values by means of the `<assign>` tag. Let us consider an example where we want to calculate an average of hours per time sheet line. In MPL 4, you can code it in the following way (provided that the TimeSheetLine cursor contains a field called `.HoursRegistered`):

```
<var HoursSum {0}>
<var LineCount {0}>
<repeating TimeSheetLine>
   <assign HoursSum {HoursSum + .HoursRegistered}>
   <assign LineCount {LineCount + 1}>
   ...
<end repeating>
<val AverageHoursPerLine {HourSum / LineCount}>
```

At the end, we create a *val* that holds the resulting average of hours per line.

Both *vals* and *vars* can be referenced only in the scope in which they are defined or in any of its nested scopes.

Value Definitions

The `val` tag:

```
<val attributes>
```

is used to define a new constant value that can be used in expressions. Once this is defined, the value cannot be changed—it is immutable.

The `val` tag takes three attributes:

- `name` specifies the name of the new value. The attribute has the type *ID* and is mandatory and nameless.
- `value` is an expression that specifyies the value that the new `val` should have. The attribute has the type *EXPRESSION* and is mandatory and nameless.
- `type` optionally specifies the type of the new value. The MPL compiler will check that the type of the expression supplied by the `value` attribute matches the specified type. If this attribute is not specified, the value's type is inferred by the MPL compiler. The attribute is optional and has the type ID, but the passed ID must match a valid type name.

Variable Definitions

The `var` tag:

```
<var attributes>
```

is used to define a new mutable variable that can later be modified using the `assign` tag. A variable that is defined using this tag can be used in expressions in the same way as existing variables.

The `var` tag takes three attributes:

- `name` specifies the name of the new variable. The attribute has the type *ID* and is mandatory and nameless.
- `value` is an expression that specifyies the initial value of the new variable. The attribute has the type *EXPRESSION* and is mandatory and nameless.
- `type` optionally specifies the type of the new variable. The MPL compiler will check that the type of the expression supplied by the `value` attribute matches the specified type. If this attribute is not specified the variable's type is inferred by the MPL compiler. The attribute is optional and has the type ID, but the passed ID must match a valid type name.

Variable Assignments

The `assign` tag:

```
<assign attributes>
```

is used change the value of a variable defined using the `var` tag.

The `assign` tag takes two attributes:

- `var` specifies the name of the variable to modify. The attribute has the type *ID* and is mandatory and nameless.
- `value` is an expression that specifies the new value of the variable. The attribute has the type *EXPRESSION* and is mandatory and nameless.

Database Queries

Starting with MPL version 4, you can declare your own database queries.

Up until MPL version 3 you were limited to using only the predefined cursors found in the standard Maconomy prints. However, this is not always sufficient for the task at hand. Sometimes the predefined cursors do not provide the information that you need, or they cannot be used freely due to the structure requirements on customized layouts (see “Print Structure”). In the “Print Employee” standard layout, for example, you will find this repeating block.

```
<repeating Employee script="L_Employee">
    {:ruler12
        .EmployeeNumber   .Name1   .Telephone;
    }
<end repeating>
```

This repeating block is iterating over a predefined cursor named “Employee,” but in this instance that cursor is used with a script and due to the structure requirements the cursor cannot be reused in another location in the customized layout.

To enable you to get to the information that you need in your prints, MPL version 4 introduces custom database queries. These are based on the Maconomy query language (MQL) that queries the Maconomy universes.

The description of MQL database queries and their use in MPL is described below using examples. The examples contain very little formatting, if none at all. This means that if you try to run the examples, the output will not be very beautiful. However, the examples are easier to read because we focus only on the data part in this section.

After describing queries by example we will give a more formal definition of the new tags: `<query>`, `<cursor>`, and `<parameter>`.

Queries, Cursors, and Repeating Blocks

Before looking into defining queries and cursors and using them in an MPL layout, let us first get an overview of the concepts involved.

You use the `<query>` tag to define a new database query. Custom queries in MPL take parameters, which, for example, can be used in the where clause. The query itself is written using MQL. Defining a query does not execute anything against the Maconomy database. A simple query could look like this:

```
<query EmployeeQuery>
    mselect EmployeeNumber, Name1, CostPrice
    from Employee
<end query>
```

A query must be instantiated, yielding a cursor, which can then be used in a repeating block. If a query is defined to take parameters, actual values for the parameters must be supplied when the query is instantiated. Because a query can be instantiated multiple times with different parameter values, queries in MPL are reusable. In our simple example, however, there are no parameters. To instantiate the query and get a cursor we would write:

```
<cursor name=EmployeeCursor query=EmployeeQuery>
<end cursor>
```

We now have a cursor. Still, nothing is executed against the database. To do that we need to use a `<repeating>` block:

```
<repeating EmployeeCursor>
   <header atstart+>
      {:ExampleRuler
         "EMPLOYEE NUMBER."  "EMPLOYEE NAME"  "COST PRICE";
      }
   <end header>
   {:ExampleRuler
      .EmployeeNumber  .Name1  .CostPrice;
   }
<end repeating>
```

Now, our query is executed and the three fields *EmployeeNumber*, *Name1*, and *Costprice* are printed.

You can use a cursor in repeating blocks multiple times. It will, however, return the same data every time (provided that the database has not changed). More interesting is to create another cursor that binds the query parameters to different values.

Queries with Parameters

Let us have a closer look at defining queries. The query we looked at before was without parameters, so let us define a query that takes parameters. If we, for example, want to print out all employees who have a particular superior employee, we could define this query:

```
<query EmployeeWithSuperior>
   mselect name1, costprice from employee
   where SuperiorEmployee = superiorParam
   using parameters superiorParam type string
<end query>
```

This query takes one parameter, the employee number of the superior whom we want to use for restricting the query.

To supply the actual values for the parameters of our query, we use the `<parameter>` tag inside our cursor definition:

```
<cursor name=EmployeeCursor query=EmployeeWithSuperior>
   <parameter superiorParam {"1010"}>
<end cursor>
```

When using the cursor *EmployeeCursor* in a repeating block, we will list all subordinate employees of the employee with employee number *1010*.

Instantiating a Query inside a Repeating Block

Also, as previously mentioned, you can instantiate a query with parameters multiple times. This is, for example, useful when binding a query inside another repeating block.

Consider the two query definitions from before:

```
<query EmployeeQuery>
```

```
    mselect EmployeeNumber, Name1, CostPrice
    from Employee
<end query>
<query EmployeeWithSuperior>
    mselect Name1, Costprice
    from Employee
    where SuperiorEmployee = SuperiorParam
    using parameters SuperiorParam type string
<end query>
```

We want to use these two queries to list all employees, and for each employee we want to list the subordinates of that employee. To do that we use two embedded repeating blocks and bind one of our cursors inside the outer repeating block like this:

```
<cursor name=EmployeeCursor query=EmployeeQuery>
<end cursor>
<repeating EmployeeCursor>
    -- Instantiate the current employee
    -- as the superior employee.
    <cursor name=SubordinateCursor
              query=EmployeeWithSuperior>
           <parameter parameter SuperiorParam
                 {EmployeeCursor.EmployeeNumber}>
    <end cursor>
    -- List current employee
    {:Ruler3Col
           .EmployeeNumber .Name1 .CostPrice;
    }
    -- List all subordinates
    <repeating SubordinateCursor>
           {:Ruler2Col
              .Name1 .CostPrice;
           }
    <end repeating>
<end repeating>
```

The cursor *SubordinateCursor* is redefined for each iteration of the outer repeating block. Each time the cursor is defined, the *superiorParam* parameter is bound to the current employee. After defining the *SubordinateCursor*, we print out information about the current employee and then we iterate over the *SubordinateCursor*.

Scoping of Queries, Cursors, and Fields

Queries and cursors are scoped, which means that they cannot be used outside the block where they are defined. Likewise, the fields of a cursor cannot be accessed outside the scope where the cursor is executed—that is, a field on a particular cursor cannot be accessed outside a repeating

block iterating over that cursor. When looking up fields we can either just name the field using the field syntax that is prefixing the field with a "." or we can explicitly name the cursor in which the field is going to be looked up. When MPL determines which field to print it always tries to find the field in the nearest repeating block.

If we revisit the example above, the scoping rules mean that the fields *Name1* and *CostPrice* are looked up on the nearest cursor—that is, a field on a particular cursor cannot be accessed outside a repeating block iterating over that cursor. When looking up fields we can either name the field using the field syntax that is prefixing the field with a "." or we can explicitly name the cursor in which the field is going to be looked up. When MPL is the *SubordinateCursor*, a field on a particular cursor cannot be accessed outside a repeating block iterating over that cursor. When looking up fields we can either just name the field using the field syntax that is prefixing the field with a "." or we can explicitly name the cursor in which the field is going to be looked up. Since the field names exist on that cursor, the information is taken from the cursor. However, both cursors in the example define fields with these particular names. To access the fields with the same names on the outer cursor, *EmployeeCursor*, we must prefix the names with the cursor name.

Take a look at the inner repeating, where we access fields on both the inner and the outer cursor by prefixing with the cursor name where necessary. Note that the field *EmployeeNumber* does not exist on the inner cursor, so MPL finds the field on the outer cursor, even though the field is not prefixed with the cursor name:

```
<repeating EmployeeCursor>
    -- Bind the current employee as the superior employee.
    -- (cursor definition as before)


    -- List current employee
    -- (left out for brevity)


    -- List all subordinates
    <repeating SubordinateCursor>
        {:Ruler2Col
            .Name1          -- SubordinateCursor.Name1
            .CostPrice      -- SubordinateCursor.CostPrice
            .EmployeeNumber -- EmployeeCursor.EmployeeNumber
            EmployeeCursor.Name1
            EmployeeCursor.CostPrice
        }
    <end repeating>
<end repeating>
```

Multi-Level Queries

MQL supports multi-level queries, and that can be used in MPL prints. Let us consider an example of a multi-level query:

```
<query RegTimeQuery>
    mselect
        -- Outer query
```

```
        [
          Employee.EmployeeNumber,
          Employee.EmployeeName,

          -- Inner query
          [
             Job.JobNumber,
             -- alias for field
             Registered.SumNumberRegistered as Hours
          ] as cursor Job    -- name of inner cursor
       ] as cursor Employee -- name of outer cursor

   -- selecting from pre-defined Maconomy Jobs universe
   -- featuring joins over employees, timesheets,
   -- jobs, etc.
   from JobCost::Universes::Jobs
<end query>
```

The preceding query selects its data from the predefined Maconomy *Jobs* universe, which can be seen from the line:

```
from JobCost::Universes::Jobs
```

Two embedded queries are defined in this query definition. This corresponds to a “groupby” definition in plain SQL, but it enables us to do embedded iteration over the result of the query—that is, a field on a particular cursor cannot be accessed outside a repeating block that iterates over that cursor. When looking up fields we can either name the field using the field syntax that is prefixing the field with a “.” or we can explicitly name the cursor in which the field is going to be looked up. With MPL, we can use two embedded repeating blocks to iterate over the result.

In this example, the outer cursor iterates over all distinct values of the fields *Employee.EmployeeNumber* and *Employee.EmployeeName* in the *Jobs* universe. Note that fields in an MQL universe can contain “.” in their names. For each distinct value pairs of these two fields, the inner cursor will iterate over *Job.JobNumber* and *Hours* (which is an alias for the field *Registered.SumNumberRegistered*).

The outer and inner cursors have been assigned the names “Employee” and “Job,” and these names are used when iterating over a cursor that is defined using the query:

```
<cursor name=RegTimeCursor query=RegTimeQuery>
<end cursor>

-- Iterating the outer “Employee” cursor
<repeating RegTimeCursor.Employee>
    .Employee.EmployeeNumber .Employee.EmployeeName

    -- Iterating the inner “Job” cursor
    <repeating RegTimeCursor.Employee.Job>
        .Job.JobNumber .Hours
```

```
    <end repeating>
<end repeating>
```

Note how the cursors are accessed. The defined `<cursor>` name, *RegTimeCursor*, is used to prefix the cursor names that are defined in the query, and the full path to a cursor inside the query is used. To access the outermost query, *Employees*, you must write:

```
<repeating RegTimeCursor.Employee>
```

To access the innermost cursor, *Jobs*, which is embedded in the outermost cursor, *Employees*, you must type the full path:

```
<repeating RegTimeCursor.Employee.Job>
```

Aggregates

MQL also supports aggregates on values—that is, a field on a particular cursor cannot be accessed outside a repeating block iterating over that cursor. When looking up fields we can either name the field using the field syntax that is prefixing the field with a "." or we can explicitly name the cursor in which the field is going to be looked up. With MPL that is count, sum, and maximum, minimum, and average values. Revisiting and extending the example from the above we get:

```
<query RegTimeQuery>
    mselect
        -- Outer query
        [
            Employee.EmployeeNumber,
            Employee.EmployeeName,

            -- Inner query
            [
                Job.JobNumber,
                -- alias for field
                Registered.SumNumberRegistered as Hours
            ] as cursor Job    -- name of inner cursor
        ] as cursor Employee -- name of outer cursor


    -- make sum of hours available on outer cursor
    aggregate sum() on Hours


    -- selecting from pre-defined Maconomy Jobs universe
    -- featuring joins over employees, timesheets,
    -- jobs, etc.
    from JobCost::Universes::Jobs
<end query>
```

The only addition is the line marked in boldface. This line tells MQL to calculate the sum of the field *Hours* and make it available on the cursor one level out from where this field is defined. In

our case the sum will be available on the *Employees* cursor as the field *Hours$SUM* and what we get is the sum of registered hours per employee.

We must update our example from before to include the new field:

```
<repeating RegTimeCursor.Employee>
    .Employee.EmployeeNumber .Employee.EmployeeName
    .Hours$SUM
    <repeating RegTimeCursor.Employee.Job>
        .Job.JobNumber .Hours
    <end repeating>
<end repeating>
```

The new field is marked in boldface. Adding a bit more formatting yields a print like the following, where we see the time registrations per job per employee and also see the sum of hours registered per employee.

Employee No.	Name	Hours Total
E018	Employee18	85.0
Job.		Hours
10250181		35.8
10250185		49.2
E019	Employee19	44.9
Job.		Hours
10250195		44.9
E021	Employee21	42.7
Job.		Hours
10250211		42.7

Metadata Cursor

When executing an MQL query a special top-level metadata cursor name *main* exists. However, by default this top-level cursor is hidden.

Let us have a look at a simple query definition:

```
<query EmployeeQuery>
   mselect
       EmployeeNumber,
       Name1
   from Employee
<end query>
```

As usual, we bind the query to a cursor:

```
<cursor name=EmployeeCursor
       query=EmployeeQuery>
<end cursor>
```

And then we execute the query:

```
<repeating EmployeeCursor>
   .EmployeeNumber .Name1
```

```
<end repeating>
```

To get access to the metadata top-level cursor, we change the cursor definition slightly:

```
<cursor name=EmployeeCursor
       query=EmployeeQuery
       showmaincursor+>
<end cursor>
```

We now have a multi-level cursor where the top-level cursor is called *main*, and the next level cursor is called *query*. The name of the inner cursor, *query*, is automatically chosen by MQL because we did not name the cursor in the MPL query definition. We can now change the repeating block like this:

```
<repeating EmployeeCursor.main>
    .query$RowCount -- Total row count of lowest sublevel
    .query$Date     -- Date of execution
    .query$Time     -- Time of execution
    .query$Count    -- Row count of query on next level
    <repeating EmployeeCursor.main.query>
        .EmployeeNumber .Name1
    <end repeating>
<end repeating>
```

The three first of the new fields are only present at the top level and show us the total row count of the query (which is the same as the sum of rows fetched by each of the iterations of the innermost cursor of the query). The last field, *query$Count*, is present on any level except the innermost and tells of the count of rows on the cursor immediately below. The first part of the field's name is named after the cursor that it is counting—in this case the *query* cursor.

If we revisit the first multi-level query example from before, we get a better feel of how automatic *count* fields work:

```
<query RegTimeQuery>
   mselect
       -- Outer query
       [
          Employee.EmployeeNumber,
          Employee.EmployeeName,

       -- Inner query
       [
          Job.JobNumber,
          -- alias for field
          Registered.SumNumberRegistered as Hours
       ] as cursor Job    -- name of inner cursor
       ] as cursor Employee -- name of outer cursor

   -- selecting from pre-defined Maconomy Jobs universe
```

```
    -- featuring joins over employees, timesheets,
    -- jobs, etc.
    from JobCost::Universes::Jobs
<end query>
```

We now bind the query using the `showmaincursor` attribute:

```
<cursor name=RegTimeCursor
    query=RegTimeQuery
    showmaincursor=true>
<end cursor>
```

We can now iterate over the cursor like this:

```
<repeating RegTimeCursor.main>
    .query$RowCount  -- Total row count of Jobs listed
    .query$Date      -- Date of execution
    .query$Time      -- Time of execution
    .Employee$Count  -- Number of Employees listed
    <repeating RegTimeCursor.main.Employee>
        .Job$Count     -- Number of Jobs for this Employee
        .Employee.EmployeeNumber
        .Employee.EmployeeName
        <repeating RegTimeCursor.main.Employee.Job>
            .Job.JobNumber
            .Hours
        <end repeating>
    <end repeating>
<end repeating>
```

We now see that we have two count fields: *Employee$Count* and *Job$Count*. The prefixes before the “$” sign are now the names of the cursors that we named in the query definition. Also note that the name of the *main* cursor now must be part of the prefix when accessing the cursors in the `<repeating>` tags.

Aggregates on the Top Level

If you use aggregates in MQL on fields that belong to the outermost cursor of your query, you must access the *main* cursor to access the aggregate values. These always exist on the cursor that is immediately above the cursor whose fields are aggregated. The cursor that is immediately above the top-level cursor that is defined in an MPL query definition is the *main* cursor.

Consider this query definition:

```
<query EmployeeQuery>
    mselect EmployeeNumber,
        Name1,
        CostPrice
    aggregate min() on CostPrice,
```

```
        max() on CostPrice
    from Employee
<end query>
```

Here, we are asking for the minimum and maximum cost price. To access the aggregate values we must access the main cursor:

```
<cursor name=EmployeeCursor
        query=EmployeeQuery
        showmaincursor+>
<end cursor>
```

Now, we can print the aggregate values (again with no formatting to simplify the example):

```
<repeating MyCursor.main>
    -- Print the aggregate values
    .CostPrice$Min .CostPrice$Max

    -- Print the employee information
    <repeating MyCursor.main.query>
        .EmployeeNumber .Name1 .CostPrice;
    <end repeating>
<end repeating>
```

Tags for Database Queries

In the previous section we saw a number of examples that illustrate the use of database queries in MPL. In this section we will give a more formal definition of the tags that are involved and their attributes.

Queries

The `query` tag:

```
<query attributes>
    MQL query
<end query attributes>
```

is used to define a new database query. You can place the query tag anywhere in the layout where a repeating can also be placed. A query definition is scoped and cannot be accessed outside the scope where it is defined. The `query` tag has no short form.

The tag is a parenthetical tag that encloses the MQL query that should be executed against the database. Note that defining a `query` does not execute the MQL query. The `query` must be instantiated to a `cursor`, and the `cursor` is iterated over in a repeating tag. The MQL query is executed just before the iteration starts.

The `query` tag takes one attributes:

- `name` specifies the name of the query. The name is used when a `cursor` tag binds the query. The attribute has the type *ID* and is nameless and mandatory.

For a thorough description of the `query` tag including examples of use, see “Database Queries.”

Example

```
<query EmployeeQuery>
   mselect EmployeeNumber, Name1, CostPrice
   from Employee
<end query>
```

Cursors

The `cursor` tag:

```
<cursor attributes>
   parameter bindings...
<end cursor>
```

is used to instantiate a previously defined `query` that yields a `cursor` that you can use in a `repeating` block. You can place a `cursor` tag anywhere in the layout where a `repeating` can be placed, but it can only refer to already defined queries that are in scope, which means that the referenced `query` must be defined in the same block or in an enclosing block in the layout. The query is *not* executed by being instantiated by a `cursor` tag. The query is executed only when the cursor is iterated over in a `repeating` block.

The tag is a parenthetical tag that encloses a number of `parameter` tags that must match the `query` definition's *parameter* section.

The `cursor` tag takes three attributes:

- `name` specifies the name of the cursor. The name is used when a `repeating` tag refers the cursor for execution. The attribute has the type *ID* and is mandatory.
- `query` specifies the name of the query that is bound by this cursor definition. The referenced query must be defined in the same or an enclosing scope as the cursor definition. The attribute has the type *ID* and is nameless and mandatory.
- `showmaincursor` designates whether the *main* metadata cursor is accessible when iterating over the cursor in a `repeating` block. The attribute has the type *BOOLEAN* and is optional. Its default value is `false`.

For a thorough description of the `cursor` tag including examples of use, see "Database Queries."

Example

```
<query EmployeeQuery>
   mselect EmployeeNumber, Name1, CostPrice
   from Employee
<end query>


<cursor name=EmployeeCursor query=EmployeeQuery>
<end query>
```

Cursor Parameters

The `parameter` tag:

`<parameter attributes>`

is used to bind formal parameters declared in `query` definition to actual values inside a `cursor` tag. The parameters given inside a `cursor` tag must match exactly the parameters that are declared in the `query` that is referenced by the enclosing `cursor` tag.

The `parameter` tag takes two attributes:

- `name` specifies the name of the formal parameter to bind. The name must match the name of one of the parameters that is declared by the referenced query. The attribute has the type *ID* and is mandatory and nameless.
- `value` specifies the value to bind to the formal parameter and can be any expression. The attribute has the type *EXPRESSION* and is mandatory and nameless.

For a thorough description of the `parameter` tag including examples of use, see “Database Queries.”

Example

```
<query EmployeeWithSuperior>
   mselect name1, costprice from employee
   where .SuperiorEmployee = superiorParam
   using parameters superiorParam type string
<end query>


<cursor name=EmployeeCursor query=EmployeeWithSuperior>
   <parameter superiorParam {"1010"}>
<end cursor>
```

Print Structure

All MPL layouts must be based on an existing layout. This layout—the *original layout*—specifies which cursors, fields, variables, and scripts can be used in the layout. To ensure that the definition of cursors makes sense and that the scripts are executed in the predefined order, all MPL layouts must have the same structure as the original layout.

Usually, you will not run into any problems in the practical development of layouts because you often base your new layouts on an existing original or structure layout. This section is therefore only relevant to you if you receive error messages about the structure of your layout or if you simply want to better understand the concept of print structure.

Structure

The MPL compiler ensures that the structure of an MPL layout sufficiently matches the structure of the original layout. The structure consists of all the printout's stacks, repetitions, conditions, headers, footers, and the paper itself. The structure describes how these elements are contained in one another.

If you export the structure layout, stacks, headers, and footers that do not have scripts assigned will be excluded.

Whether the MPL layout matches the structure of an original layout is defined using the concepts "script structure" and "stackless structure," which only contain a part of the structure described above.

Trees

When talking about the structure of MPL layouts it is often easier to perceive the structure as a tree. The root of the tree is the `paper` tag, and each block (repetition, condition, or stack) is a branch of the tree. You often refer to such a branch as a *node*. The analogy between MPL layouts and trees is used in the rest of this section to explain and illustrate the presented concepts.

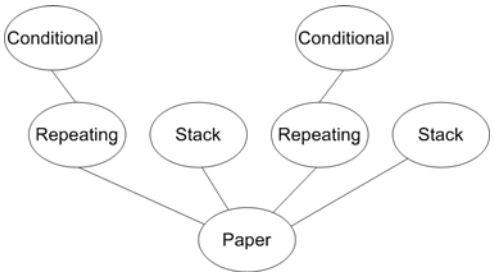
Example

See the structure layout for the printout "Print Employee Report."

```
<paper cursor=EMPLOYEE
       script="STANDARD_PAGESCRIPT1">
   <repeating cursor=CUSTOMERTIMEACTIVITY
          script="CUSTOMERTIMEACTIVITY">
       <conditional script="CUSTOMERTIMEACTIVITYTOTAL"
             variable=ShowTotalVar>
       <end conditional>
   <end repeating>
   <stack script="CUSTOMERTIMETOTAL">
   <end stack>
   <repeating cursor=INTERNALTIMEACTIVITY
             script="INTERNALTIMEACTIVITY">
   <conditional variable=ShowTotalVar>
   <end conditional>
```

```
    <end repeating>
    <stack script="EMPLOYEETOTAL">
    <end stack>
<end paper>
```

The corresponding tree looks as follows:



Script Structure

The *script structure* is a layout that is made up of all entities that can have a script assigned—that is, paper, stacks, repetitions, conditions, headers, and footers. An entity is a part of script structure if it either:

1. has a script assigned.
2. is a condition, repetition, while loop, or a paper that contains a block observing rule 1 or 2.

If you view the tree, the rule can be defined as follows: If a block observes rule 1, all conditions , repetitions, while loops, and a paper along the path between the block and the root must be included.

The script structure in the user-defined layout must be identical to the script structure in the original layout. The purpose of this condition is to ensure that scripts are executed the same number of times and in the same order for all layouts with the same original layout.

The examples in this section (below) have been constructed for the purpose of illustration. Note that layouts with a similar structure exist.

Example 1

The following structures have the same script structure:

```
<paper script="s1">
    <stack script="s2">
        <repeating cursor=c1>
        <end repeating>
    <end stack>
<end paper>
```

and

```
<paper script="s1">
    <stack script="s2">
    <end stack>
<end paper>
```

The script structures are identical, because the repeating block is not a part of the script structure.

Example 2

The following structures do not have the same script structure:

```
<paper script="s1">
   <repeating cursor=c1>
      <stack script="s2">
      <end stack>
   <end repeating>
<end paper>
```

and

```
<paper script="s1">
   <stack script="s2">
   <end stack>
<end paper>
```

In this example, the repeating block is part of the script structure, because it contains a block with a script.

Example 3

The following structures do not have the same script structure:

```
<paper script="s1">
   <stack script="s2">
   <end stack>
   <stack script="s3">
   <end stack>
<end paper>
```

and

```
<paper script="s1">
   <stack script="s3">
   <end stack>
   <stack script="s2">
   <end stack>
<end paper>
```

The script structures are not identical, because the order of the scripts that are used is different in the two fragments.

Example 4

The following structures have the same script structure:

```
<paper script="s1">
   <stack>
      <stack script="s2">
      <end stack>
```

```
    <end stack>
<end paper>
```

and

```
<paper script="s1">
    <stack script="s2">
    <end stack>
<end paper>
```

Example 5

The script structure for the printout “Print Employee Report” looks as follows:

```
<paper cursor=EMPLOYEE
       script="STANDARD_PAGESCRIPT1">
    <repeating cursor=CUSTOMERTIMEACITIVITY
          script="CUSTOMERTIMEACTIVITY">
    <end repeating>
    <stack script="CUSTOMERTIMETOTAL">
    <end stack>
    <repeating cursor=INTERNALTIMEACTIVITY
    <end repeating>
    <stack script="EMPLOYEETOTAL">
    <end stack>
<end paper>
```

Stackless Structures (MPL 1 and 2)

The concept of stackless structure has been simplified and loosened up in MPL 4; see the next section about the repeating structure for more details.

Until MPL 4, a *stackless structure* is defined as the structure that is obtained by removing all stacks and conditionals (that is, `stack` and `conditional` elements).

The stackless structure of the user-defined layout must be a subtree of the stackless structure of the original layout. A subtree is achieved as a result of pruning the tree; that is, nodes or branches must be removed from the top of the tree. In other words, you can leave out part of the stackless structure from the original layout, but you can only leave out an entire block, not part of its contents.

Example 1

The following structure:

```
<paper script="s1">
    <conditional variable=v1>
    <end conditional>
<end paper>
```

is a subtree of:

```
<paper script="s1">
```

```
    <conditional variable=v1>
        <repeating cursor=c1>
        <end repeating>
    <end conditional>
<end paper>
```

whereas the following structure:

```
<paper script="s1">
    <repeating cursor=c1>
    <end repeating>
<end paper>
```

is not a subtree, because the elements have not been removed from the inside.

Example 2

The stackless structure of the printout “Print Employee Report” looks as follows:

```
<paper cursor=EMPLOYEE
       script="STANDARD_PAGESCRIPT1">
    <repeating cursor=CUSTOMERTIMEACTIVITY
                 script="CUSTOMERTIMEACTIVITY">
    <end repeating>
    <repeating cursor=INTERNALTIMEACTIVITY
                 script="INTERNALTIMEACTIVITY">
    <end repeating>
<end paper>
```

Repeating Structure (MPL 4)

Repeating structure is a simplified and loosened up model of what was known as a *stackless structure* until MPL 4.

A *repeating structure* consists of all of the repeatings in an MPL 4 layout. We say that the repeating structure of a custom layout (or just *custom repeating structure*) must comply with the repeating structure of its original layout (or just *original repeating structure*).

Before digging into the details of what it actually means for a custom repeating structure to comply with its original repeating structure, let us first understand what purpose this check serves.

Every print environment contains a set of predefined cursors that can be iterated over by means of the `<repeating>` tag. These cursors can reference other cursors in the *where* clauses of their underlying *SQL* queries. For instance:

```
<repeating TimeSheetHeader >
    <repeating TimeSheetLine>
    <end repeating>
<end repeating>
```

for the `TimeSheetLine` cursor to contain only the lines of the `TimeSheetHeader` to which it belongs, `TimeSheetLine` must reference the `TimeSheetHeader` cursor in the *where* clause of

its underlying *SQL* query. For this to work, it is necessary that any repeating over the `TimeSheetLine` cursor is embedded inside of a repeating over the `TimeSheetHeader` cursor.

In general, for all of the cursors in a custom layout we want to guarantee that their underlying references are initialized, which basically means that such a cursor is embedded in the same sequence of parent repeatings as at least one of the sequences of parent repeatings in the original layout.

As an example, let us consider the following original repeating structure:

```
<repeating A>
    <repeating B>
           <repeating D>
           <end repeating>
    <end repeating>
    <repeating D>
    <end repeating>
    <repeating C>
    <end repeating>
<end repeating>
```

The following custom repeating structure complies with the given original repeating structure, because each repeating in the custom structure is embedded in the exact same sequence of parent repeatings as in the original layout:

```
<repeating A>
    <repeating D>
        <repeating C>
        <end repeating>
    <end repeating>
<end repeating>
```

In particular, the repeating over *C* in the original structure expects to be embedded in a repeating over *A*, which is clearly the case in the custom structure as well. Likewise, *D* in the original structure is embedded both just in *A*, and in the sequence *A* followed by *B*. The custom structure complies with the first parent sequence, namely a single parent *A*. *A* has no parents both in the original and custom structure, hence the custom repeating structure complies with the original structure

On the other hand, the custom repeating structure below does not comply with the original repeating structure:

```
<repeating B>
    <repeating D>
    <end repeating>
<end repeating>
```

D in the original layout is embedded in *A* followed by *B*, or in a single parent repeating *A*. In the custom layout, though, *D* is embedded only in *B*; therefore, the repeating over *D* violates the repeating structure. Likewise *B*, because it is expected to be embedded in *A* like in the original structure.

In the given examples, we were using name aliases to represent repeating blocks. As far as the repeating structure is concerned, for the two repeatings to be considered the same, not only do

they have to iterate over the same cursor, but also have the same values of attributes: `script` and `groupBy` when these attributes are set.

Advanced MPL

This section describes the remaining tags in MPL. Among other things, we will cover two tags that make it possible to control page breaks and to redefine embedded formatting constants. Furthermore, these sections will describe a number of tags that are not necessary in the structural definition of printouts, but make it easier to build elegant MPL definitions that are easily edited and maintained.

Additional Tags

border

A `border` tag specifies a vertical limit above which all preceding elements must be printed. Text elements that are specified before the `border` statement in the MPL definition will only be printed above the position of the border and continue on the following page, while elements that are specified after the border will be printed further down on the page. The elements after the border start at the height specified in the border.

```
<border attributes>
```

Note, however, that footers will be placed below the border.

The tag has the following attributes. You must specify exactly one of the following attributes:

- `top` specifies the position on the page where the border should be inserted. The attribute has the type *LENGTH* and defines the distance from the top of the page (that is, including page margins).
- `bottom` is similar to top, but specifies the distance to the bottom of the page. The border tag can only be specified in the `paper` tag; that is, it cannot be specified within other parenthetical tags.

If you specify more than one border tag, each tag only affects the positioning of the elements after the previous border tag:

```
elem1  -- placed in the top 10 cm of the page
...
elemm  -- placed in the top 10 cm of the page
<border top=10cm>
elemm+1 -- placed in the top 5 cm of the page
...
elemn  -- placed in the top 5 cm of the page
<border top=5cm>
elemn+1 -- placed 5 cm from the top of the page
```

Note that the element $elem_{m+1}$ following the first border is placed 10 cm from the top edge of the page, but because it is followed by a border, which is placed 5 cm from the top edge of the page, a page break is triggered before this element, and $elem_{m+1}$ is printed in the top part of a new page.

Example

Let us assume that you want to print out an invoice on stationery with a preprinted giro payment slip in the lower part of the page. Let us also assume that the height of the giro payment slip is 101.66 mm. You should therefore add a border of 101.66 mm from the bottom of the page.

The following is an extract from the print definition for “Print Invoice”:

```
<paper cursor=INVOICE script="SIDE">
    -- Invoice lines
    <repeating cursor=INVOICELINE
              script="INVOICELINEBLOCK">
        <repeating cursor=INVOICEBOMLINE
                  script="BOMLINEBLOCK">
            ...
        <end repeating>
    <end repeating>
    -- Invoice totals
        "TOTAL DUE" .Currency .TotalCurrency;
    <conditional script="VATBLOCK"
              variable=VATBlockIncludedVar>
        <repeating cursor=CODE script="VATLOOPBLOCK">
            ...
        <end repeating>
    <end conditional>
    -- Giro payment slip
    <border bottom=101.6mm>
    <canvas>
        ...
    <end canvas>
<end paper>
```

Because the border has been specified after the invoice lines and totals, none of these will be printed in the bottom 101.6 mm of the page.

If we assume that the printout contains many invoice lines, these lines will be printed on the first page until reaching the 101.6 mm bottom border mark, after which a page break is triggered. Nor will the totals be printed in the bottom 101.6 mm of the page, meaning that if the printing of the invoice lines ends 12 cm from the bottom of the page, the first 18.4 mm of the totals will be printed, followed by a page break and the remaining totals. When all invoice lines and totals have been printed, the next print start position will be moved to 101.6 mm from the bottom of the page, where printing is resumed. Thereafter, the border has no effect.

newpage

You can force a page break anywhere in your definition using the `newpage` tag:

```
<newpage>
```

The `newpage` tag has no attributes in MPL 2. In MPL 3, the attribute `orientation` (of type ID) is supported in some cases. See “Paper Orientation Change” and “<newpage> in row no longer allowed” for more on `newpage` changes in MPL 3.

You can include the tag in stacking tags (however not in headers and footers). In MPL 2, it can also be used in arrays where it should be inserted as the only element in a row.

Example

If you want a page break after each time sheet in the printout from “A Printout Example: Time Sheets,” you can force a page break for each iteration of the TIMESHEETHEADER repetition.

```
<repeating cursor=TIMESHEETHEADER
             script="S_TIMESHEETHEADER">
   ...
      {:[][stretch+][][stretch+][][][][][][][][]]
         ...
         "" "" "" ""
         SumDay1 SumDay2 SumDay3 SumDay4 SumDay5 SumDay6 SumDay7 Grandtotal;
         <newpage>;
      }
<end repeating>
```

Here we have replaced `<skip 5mm>` with `<newpage>`. You could also choose to insert the page break outside the array:

```
   ...
      "" "" "" ""
      SumDay1 SumDay2 SumDay3 SumDay4 SumDay5 SumDay6 SumDay7
      Grandtotal;
   }
   <newpage>
<end repeating>
```

The result will be exactly the same. Note that a forced page break will be inserted after each time sheet, that is, there will also be a page break after the last time sheet (the last page of the printout is an empty page).

Alternative Text Tag

In the above we saw how you could set default values for attributes. This functionality is often used to set defaults for text formatting. However, you often need two sets of defaults; Maconomy’s default fonts for fixed texts are formatted to 7 pt, whereas texts that are parts of fields (for example, hyphens in ranges or slashes in dates) are formatted to 9 pt.

To support two different default values for texts, MPL has an alternative text tag called `text2` that has the same characteristics as `text` but can be assigned separate default attribute values:

```
<text2 attributes>
```

You will often use the short form:

```
'text'
```

which is the short form for

```
<text2 title="text">
```

The short form of `text2` is different from the short form of `text` as the text is in single quotes instead of double quotes. `text2` has exactly the same attributes as text. The default value for `fontsize` is 9; the other default values are the same.

In “The front page” in “A Printout Example: Time Sheets” we became familiar with the use of the `text2` tag. Here we used the short form “/” in dates and “-“ in ranges to ensure that the separators were assigned the same font sizes as the fields.

Grid

The `grid` tag enables you to define your own length units. The length unit is called grid, but it can vary according to orientation (horizontal or vertical). A grid definition (only one) is provided immediately after the margin definition (if any) in the print header.

```
<grid attributes>
```

The tag has the following attributes, which are both mandatory and have the type *LENGTH*.

- `hor` specifies the length of a horizontal grid unit.
- `ver` specifies the length of the vertical grid unit.

When you have defined a grid, you can use the tag as a unit alongside `cm`, `in`, and so on.

Example

To use Danish inches to specify the horizontal grid unit and Danish feet to specify the vertical length unit, you can define the following grid:

```
<mpl 2>
<layout "Simple"
       ancestor="Print_Inventory_Info_Card"
       ancestorlayout="Standard">
<page "A4">
<grid hor=2.62cm    -- A Danish inch
       ver=31.40cm> -- A Danish foot
<paper>
   <island width=5grid height=0.5grid>
   <end island>
<end paper>
```

Here the island is assigned a width of 5 Danish inches and a height of 1/2 Danish foot.

Colors

MPL enables you to change the print color. You can specify the printing color by means of two attributes:

1. The `color` attribute has type *ID* and the following legal values: black, red, blue, green, yellow, cyan, white, magenta.
2. The `rgb` attribute has type *RGB*, which is a triple that contains percentages of Red, Green, and Blue respectively: (*INTEGER*, *INTEGER*, *INTEGER*).

The `color` attribute is really a short form, because it represents all combinations of `rgb` where values are either 0 or 100, for example, (100,0,0) = red, (100,100,100) = white.

The `color` and `rgb` attributes are both style attributes, which can be specified for almost all tags. Furthermore, the `rgb` attribute is nameless. The attributes can be used in the following tags and tag types:

- All parenthetical tags
- Ruler columns
- vline
- text
- text2
- field
- var
- line
- hline
- title

When the three percentages in an `rgb` are the same (for example, (30, 30, 30)), the result is a shade of gray.

Example

```
"Text":color=blue
.FieldName:(50,50,50)
<stack rgb=(100,50,50)>
    ...
<end stack>
```

Images

The `image` tag:

```
<image attributes>
```

specifies that an image is to be printed.

MPL supports images of the following types:

- JPEG (Joint Photographic Experts Group). The usual file extensions are .jpg and .jpeg.
- PNG (Portable Network Graphics). The usual file extension is .png.

For an image to be printed, it is necessary that it must be imported into the database using the Maconomy client window Document Archives. In this window, documents are organized in user-defined document archives, so that all documents are identified by their names and document archives. A reference to an image document in MPL takes the following form:

DocumentGroup\DocumentName

It is mandatory that an image has exactly one of four attributes:

- `title` specifies a direct reference to the image document. The `title` attribute is nameless.
- `fieldname` specifies the name of a field that contains a reference to the image document.
- `varname` specifies the name of a variable that contains a reference to the image document.
- `expression` specifies an expression that yields a reference to the image document (that is, a value of type STRING). The `expression` attribute is nameless. For more information on expressions, see “Expressions.”

Example

```
<image title="MyImages\MyLogo.jpg">
```

If you want to specify the cursor from which the fieldname should be taken, you can use the following attribute:

- `cursor` specifies the name of the cursor from which the field is to be taken. The attribute has the type *ID*.

If you do not specify a cursor name, the value will be taken from the nearest cursor with a field of the specified name.

In addition to this, you can specify attributes concerning the image size and justification:

- `height` has the type *LENGTH* and specifies the height of the image.
- `width` has the type *LENGTH* and specifies the width of the image.
- `justification` has the type *ID*. It can take the values `left`, `center`, and `right`.

  If justification is specified, `width` must be specified as well. This applies to MPL for Universe Reports only.
- `link` specifies that the image is to function as a link. This applies to MPL for Universe Reporting only. For more information, see “Links.”

By default, the image will be scaled to the height and width that are indicated by these attributes. If only the `height` attribute is provided, the width will be scaled to maintain the default correlation between height and width for the image. The same goes when only the `width` attribute is provided. If neither a `height` nor a `width` attribute is provided, the default size of the image is used.

Note, however, that when an MPL layout is compiled, the default size of the image is not known. The height of an image is therefore assumed to be 0 if the `height` attribute is not provided, regardless of whether the `width` attribute is provided or not. The same goes for the width of the image. This means that a construction like:

```
<image title="MyImages\MyLogo.jpg"> "Text"
```

will result in “Text” being written on top of the image. To avoid this, while still keeping the default size of the image, you can use the following attributes:

- `scaleheight` has the type *BOOLEAN* and specifies whether the image height should be scaled. The default value is `true`.
- `scalewidth` has the type *BOOLEAN* and specifies whether the image width should be scaled. The default value is `true`.

If we change the example above to:

```
<image title="MyImages\MyLogo.jpg" scaleheight- height=100pt> "Text"
```

the image will keep its default size, but the compiler will assume that the image height is 100pt and consequently print “Text” 100pt below the top of the image.

Barcodes and QR codes

Barcodes and QR codes are ways of encoding information in a visual form that can be easily scanned by electronic devices. Every modern smartphone these days is capable of scanning QR codes and most types of barcodes, which avoids the unnecessary manual process of typing the information into an IT system.

The `<barcode>` tag enables you to place a barcode in an MPL layout in a very easy way. It is enough to specify the barcode type and the data to be encoded, and the barcode image will be generated by the MPL engine. Such a barcode can be treated as any other image in MP; all of the attributes applicable to images, apart from the ones used to specify the path to the image, also apply to barcodes.

The three main barcode attributes are:

- `data` specifies the data that is to be encoded in the barcode. Different barcode types expect different data formats. This attribute is mandatory, nameless, and of type *EXPRESSION*.
- `type` specifies which barcode type should be rendered to encode the given data. It is of type *ID*. Exactly one of the `type` or `typeExpression` attributes must be specified. MPL supports 15 types of barcodes:
  - `EAN13` — International Article Number consisting of 13 digits (EAN-13), equivalent to UCC-13.
  - `UPCA` — Universal Product Code A (UPC-A) consisting of 12 digits, equivalent to EAN-12 or UCC-12.
  - `EAN8` — International Article Number consisting of 8 digits (EAN-8), equivalent to UCC-8.
  - `UPCE` — Universal Product Code E, consisting of 8 digits.
  - `EANSUPP` — EAN-13 with an EAN-5 supplement (specified using the `dataSupplement` attribute).
  - `CODE128` — Plain barcode 128, variable length.
  - `CODE128UCC` — UCC/EAN-128 with a full list of 128 ASCII characters, variable length.
  - `INTER25` — Interleaved 2 of 5 barcode, encoding an even number of digits (variable length). When the attribute `generateCheckSum` is `true`, an extra checksum digit is generated and hence an odd number of digits is required as input data text.
  - `POSTNET` — Postal Numeric Encoding Technique, which can be:
    - ZIP — Consisting of 5 digits.
    - ZIP+4 — Consisting of 9 digits.
    - ZIP+4 + DP (Delivery Point) — Consisting of 11 digits.
  - `PLANET` — Postal Alpha Numeric Encoding Technique, variable length code consisting of digits only.
  - `CODE39` — Code 39, also known as Alpha39, Code 3 of 9, Code 3/9, Type 39, USS Code 39, or USD-3. Variable length, the valid character set includes: uppercase letters (A through Z), numeric digits (0 through 9), and a number of special characters

(-, ., $, /, +, %, and space). An additional character (denoted '*') is used for both start and stop delimiters.

- `CODABAR` — Also known as Ames Code, NW-7, Monarch, Code 2 of 7, Rationalized Codabar, ANSI/AIM BC3-1995, or USD-4. Variable length, allowed symbols: 12 from the group: (digits 0-9, dash, $), 4 symbols from the group: (:/+.), and 4 start/stop symbols: (ABCD, in some specifications EN*T).
- `PDF417` — PDF417 barcode, variable length.
- `DATAMATRIX` — DATAMATRIX barcode, variable length.
- `QRCODE` — QR code, variable length.

- `typeExpression` — the same as `type`, except that this attribute is of type *EXPRESSION*, which allows for choosing a barcode type at run time, depending on the data in Maconomy. Exactly one of the `type` or `typExpression` attributes must be specified.

In addition to the above attributes, the `<barcode>` tag supports a number of attributes that are applicable only to certain barcode types. If not applicable to a certain barcode type, these attributes simply take no effect.

- `dataSupplement` — The `EAN-5` data supplement of type *EXPRESSION,* applicable for `EANSUPP` barcodes only.
- `guardBars` — Some barcodes can render guard bars around them. The attribute is of type *BOOLEAN* and should be set to `true` if the guard bars are to be generated.
- `textJustification` — Some barcodes support justification of the data text rendered along with the barcode. The attribute is of type *ID*, and its valid values are: `left`, `right`, and `center`.
- `generateChecksum` — Some barcodes can generate a checksum character for the data to be encoded. The attribute is of type *BOOLEAN* and should be set to `true` if a checksum is to be generated.
- `checksumText` — If the `generateChecksum` attribute is set to `true`, the generated checksum can be also displayed as part of the data text rendered along with the barcode, if the `checksumText` attribute is set to `true` as well.
- `startStopText` — Of type *BOOLEAN*, `true` if the data text rendered along with the barcode should be surrounded by the start/stop text (if applicable).
- `extended` — Of type *BOOLEAN*, `true` if the barcode should operate on the extended character set (if applicable).

In addition, the attributes `width`, `height`, `scalewidth`, `scaleheight`, `justification`, and `pos` work in the exact same way as for images.

The `<barcode>` tag was introduced in MPL 4 as of TPU 16 SP2.

Example

The following code snippets generate all 15 kinds of barcodes that MPL 4 supports.

```
<default tag=eval attribute=fontsize value=12>
<val ean8Data {"96385074"}>
^{"EAN 8 : " + ean8Data}
<barcode {ean8Data} EAN8 height=50pt>
```

```
<skip 10pt>
<val ean13Data {"5901234123457"}>
^{"EAN 13 : " + ean13Data}
<barcode {ean13Data} EAN13 height=50pt>

<skip 10pt>
<val upcaData {"785342304749"}>
^{"UPC-A : " + upcaData}
<barcode {upcaData} UPCA height=50pt>

<skip 10pt>
<val upceData {"03456781"}>
^{"UPC-E : " + upceData}
<barcode {upceData} UPCE height=50pt>

<skip 10pt>
<val eanSupp_ean {"1234567891234"}>
<val eanSupp_sup {"54321"}>
^{"EAN-SUP, EAN: " + eanSupp_ean + ", SUPP: " + eanSupp_sup}
<barcode {eanSupp_ean} datasupplement={eanSupp_sup} EANSUPP height=50pt>

<skip 10pt>
<val code128Data {"0123456789 hello $%*@"}>
^{"CODE 128 : " + code128Data}
<barcode {code128Data} CODE128 height=50pt>

<skip 10pt>
<val code128UCCData {"01912345678901213101000355l0ABC123"}>
^{"CODE 128 UCC: " + code128UCCData}
<barcode {code128UCCData} CODE128UCC height=50pt>

<skip 10pt>
<val code128UCCData2 {"(01)00000090311314(10)ABC123(15)060916"}>
^{"CODE 128 UCC, example 2: " + code128UCCData2}
<barcode {code128UCCData2} CODE128UCC height=50pt>

<skip 10pt>
<val codeInter25 {"41-1200076041-00"}>
^{"Barcode Interleaved 2 of 5: " + codeInter25}
<barcode {codeInter25} INTER25 height=50pt>

<skip 10pt>
<val codeInter252 {"06110123456783"}>
^{"Barcode Interleaved 2 of 5, 2: " + codeInter252}
<barcode {codeInter252} INTER25 height=50pt>

<skip 10pt>
<val codePostnet {"01234"}>
^{"POSTNET, ZIP: " + codePostnet}
<barcode {codePostnet} POSTNET height=20pt>

<skip 10pt>
<val codePostnet2 {"012345678"}>
^{"POSTNET, ZIP+4: " + codePostnet2}
<barcode {codePostnet2} POSTNET height=20pt>

<skip 10pt>
<val codePostnet3 {"01234567890"}>
^{"POSTNET, ZIP+4 and dp: " + codePostnet3}
<barcode {codePostnet3} POSTNET height=20pt>

<skip 10pt>
```

```
<val codePlanet{"01234567890"}>
^{"PLANET: " + codePlanet}
<barcode {codePlanet} typeExpression={"PLANET"} height=20pt>

<skip 10pt>
<val code39 {"MPL 4 IN ACTION"}>
^{"Barcode 3 of 9: " + code39}
<barcode {code39} CODE39 height=50pt >

<skip 10pt>
<val code39_2 {"MPL 4 in action"}>
^{"Barcode 3 of 9 Extended: " + code39_2}
<barcode {code39_2} CODE39 height=50pt  extended+ >

<skip 10pt>
<val codeBar {"A123A"}>
^{"CODABAR: " + codeBar}
<barcode {codeBar} CODABAR height=50pt startstoptext+>

<skip 10pt>
<val text {"This is some pretty long text with strange characters like
%^&^!~"}>
^{"PDF 417: " + text}:wrap+
<barcode {text} PDF417 height=50pt>

<skip 10pt>
^{"DATAMATRIX: " + text}:wrap+
<barcode {text} DATAMATRIX height=80pt>

<skip 10pt>
<val qrData {"MPL 4 supports barcodes now!"}>
^{"QRCode : " + qrData}
<barcode {"MPL 4 supports barcodes now!"} QRCODE height=150pt>
```

Running this code snippet as a part of an MPL 4 layout results in the printout that is shown on the next two pages.

EAN 8 : 96385074

EAN 13 : 5901234123457

UPC-A : 785342304749

UPC-E : 03456781

EAN-SUP, EAN: 1234567891234, SUPP: 54321

CODE 128 : 0123456789 hello $%*@

CODE 128 UCC: 0191234567890121310100035510ABC123

CODE 128 UCC, example 2: (01)00000090311314(10)ABC123(15)060916

Barcode Interleaved 2 of 5: 41-1200076041-00

Barcode Interleaved 2 of 5, 2: 06110123456783

POSTNET, ZIP: 01234

POSTNET, ZIP+4: 012345678

Overlay images in QR codes

It is possible to overlay images at the center of a QR code without compromising the QR code's readability; a QR code scanner/app should be able to scan such a QR code successfully even though part of the QR code area is overlaid by an image.

For example, the following code snippet:

```
<barcode {"Hello world, how are you doing?"}
    QRCODE
    height=46mm
    qrOverlayImage={"qrcode\swiss_cross.png"}
```

will overlay the `qrcode\swiss_cross.png` image from the Maconomy's document archive at the center of the QR code, rendering the following result:

Technically speaking, overlaying images over QR codes is possible because QR codes implement *error correction,* which means that a QR code can still be read by a scanner even though part of it is damaged or covered by an image. The QR code specification lists the following error correction levels and their corresponding theoretical recovery rates (the percent of the coding words that can be missing without affecting the QR code's readability).

- L: 7%
- M: 15%
- Q: 25&
- L: 30%

In practice, the length of the encoded text influences the number of the coding words in the QR code. Moreover, the placement of the overlay image and its size/shape affect the number of the coding words covered by the image. Therefore, in practice, for the QR codes to scan correctly, the overlay image tends to require a size smaller than what those theoretical percentage values would suggest.

It is up to the MPL layout developer to choose the right error correction level and overlay image size depending on the use case and to test the chosen values in practice. These two properties can be set using the following `<barcode>` attributes:

- `qrErrorCorrection` - the requested error correction level. Valid values are: L (lowest), M, Q, H (highest). The higher the error correction level, the bigger the overlay image can be without affecting the QR code's readability.

  The default values are

  - L without an overlay image
  - M with an overlay image

- `qrOverlayAreaRatio` - the ratio of the overall QR code's area (including the QR code's silent zone - the white area around the QR code) that the overlay picture should take up. Valid values are from 0.0 to 1.0.

  The default value for error correction level M is 0.05.

Apart from those attributes, one can also set the requested overlay image quality with the `qrOverlayQuality` attribute. Since the overlay image is drawn over the QR code image, the QR code image has to be scaled up to have enough pixels to draw the overlay image over it without losing quality.

- `qrOverlayQuality` - the requested quality level. Valid values are: L (lowest), M, Q, H (highest).

Level H indicates that the overlay image should be drawn with the original quality and the QR code image should be scaled up accordingly (up to 2000 x 2000 pixels max). The higher the quality level, the more RAM memory the QR code image will consume when generating a PDF and the bigger the resulting PDF size will be. It is recommended to use the lowest possible value that renders a satisfying quality for the PDF size not explode. The default value is Q.

The following example shows how to set all the above described attributes:

```
<barcode {"Hello world, how are you doing?"}
   QRCODE
   height=46mm
   width=46mm
   qrOverlayImage={"qrcode\swiss_flag.png"}
   qrErrorCorrection=M
   qrOverlayAreaRatio=0.025
   qrOverlayQuality=L>
```

and renders the following result:

Setting Next Page Number

It is sometimes convenient to be able to control page numbering in an MPL printout. For example, when you are batch printing invoices you might want each invoice to start with page number 1, instead of just continuing with increasing page numbers. Also, when you are including static PDFs, it is sometimes convenient to set the page number to account for the size of the included PDF document.

This functionality is provided by the `<nextpagenumber>` tag, which has one mandatory and nameless attribute value of type *EXPRESSION*. The semantics of this tag are that the value of the `PageNumber` variable will be set to the new value on the next page to be printed, after the `<nextpagenumber>` tag has been executed. The `PageNumber` variable controls the page numbers that are printed in an MPL layout.

Note: The `<nextpagenumber>` tag was introduced in MPL 4 as of TPU 16 SP2.

Example 1

In the following example, the `PageNumber` variable will be reset to 1 after printing each invoice.

```
<paper cursor=InvoiceEditingHeader>
  "Invoice body”
```

```
  ...
  <nextpagenumber {1}>
<end paper>
```

Example 2

To see how you can use the `<nextpagenumber>` tag with `<includepdf>`, see “Example 2” in “Including static PDF documents.”

Including static PDF documents

MPL is a language for dynamically generating PDF documents based on the data that is in Maconomy. There are certain situations, however, when you want to include an already generated, static PDF into a Maconomy print, for example when attaching a customer satisfaction survey or a letter of fulfillment to an invoice.

The `includepdf` tag

```
<includepdf attributes>
  ...
<end includepdf>
```

has been designed to meet these exact needs. It enables you to include static PDF documents from the Maconomy Document Archive directly into an MPL print.

The following attributes are supported:

- `path` denotes a path to the document in the Document Archive that we want to include. It is of type *EXPRESSION,* meaning that the path can be calculated dynamically at run time, depending on the context that we are currently in. The path attribute is mandatory and nameless.
- `skipNewPageAfterPdf` specifies whether there should be a page break (new page) added after the included PDF. This attribute is of type *BOOLEAN* and defaults to `false`.

  There are two basic scenarios when including a PDF in an MPL print. We might want to include the PDF at the end of a `paper` tag content, for example when attaching a satisfaction survey document to an invoice. In this case the value of `skipNewPageAfterPdf` should be set to `true`, because the `paper` tag will make an implicit new page after each `paper` cursor record has been printed. On the other hand, when including a PDF somewhere in the middle of a `paper` tag, we want to make a page break after the included PDF, so we should set the `skipNewPageAfterPdf` to `false`, which is its default value.

When including a PDF in an MPL print, a natural question arises: what about the page numbering? Should we account for the included PDF pages or not? MPL leaves it up to the layout developer, who can decide to either account for the included PDF document’s number of pages or not. To this end, the `includepdf` tag brings a variable `noOfIncludedPdfPages` into its children’s scope, where we can use the `<nextpagenumber>` tag to set the next page number to any value we wish.

> **Note:** The `<includepdf>` tag was introduced in MPL 4 as of TPU 16 SP2.

Example 1

When including a PDF somewhere in the middle of a `paper` tag, we would usually do something like this (provided that there is a MyExample.pdf document in the PDFs group in the Document Archive).

```
<paper>
  "First part of the print"
  ...
  <includepdf {"pdfs\MyExample.pdf"}>
  <end includepdf>
  ...
  "Second part of the print"
  <footer onLastPage+>
   ^{"Page " + PageNumber}
  <end footer>
<end paper>
```

Note that an automatic page break will be inserted after the included PDF document.

Example 2

Let us modify the previous example so that we account for the number of pages of the included PDF in the document page numbering. To this end, we will use the `<nextpagenumber>` tag together with the `noOfIncludedPdfPages` variable. We must remember that the `<nextpagenumber>` tag will set the page number for the next page to be printed after this tag has been executed. For that reason, we should postpone adding a page break after the included PDF to the point after the execution of the `<nextpagenumber>` tag. These considerations lead to the following piece of code (changes to the previous example appear in bold):

```
<paper>
  "First part of the print"
  ...
  <includepdf {"pdfs\MyExample.pdf"} skipNewPageAfterPdf+>
    <nextpagenumber {PageNumber + noOfIncludedPdfPages + 1}>
  <end includepdf>
  <newpage> --triggers nextpagenumber to take effect
  ...
  "Second part of the print"
  <footer onLastPage+>
   ^{"Page " + PageNumber}
  <end footer>
<end paper>
```

Example 3

When including a PDF document at the end of a `paper` tag, we should set the `skipNewPageAfterPdf` attribute to `true`, because the `paper` tag will make an implicit page break for each `paper` cursor record.

```
<paper>
  "The paper content"
```

```
  ...
  <includepdf {"pdfs\MyExample.pdf"} skipNewPageAfterPdf+>
  <end includepdf>
<end paper>
```

goto

The `goto` tag:

```
<goto attributes>
```

specifies moving the print cursor to a position on the page indicated by the attributes.

It is mandatory that a `goto` has exactly one of two attributes:

- `top` specifies the position on the page to which the cursor should move. The attribute has the type *LENGTH* and defines the distance from the top of the page (that is, including page margins).
- `bottom` is similar to `top`, but specifies the distance to the bottom of the page.

Example

```
<goto top=5cm>
```

Note that you can only move the print cursor forward. This means that a construction like:

```
<goto top=6cm> "Text1"
<goto top=5cm> "Text2"
```

will result in “Text2” being printed on the page that follows the page that contains “Text1.”

Together with the `stop` attribute, the `goto` tag enables you to make border constructions within all parenthetical tags:

```
<stack stop=5cm>
   ...
<end stack>
<goto bottom=5cm>
```

This construction will have the result that nothing (except footers) from the stack is printed below the 5cm margin to the page bottom. When the stack has been printed, the printing will continue 5cm from the page bottom.

title

Within Maconomy, a default title is associated with some fields and variables. In MPL, this default title can be accessed through the tag:

```
<title attributes>
```

It is mandatory that a title has exactly one of two attributes:

- `fieldname` has type *ID* and specifies the name of a field.
- `varname` has type *ID* and specifies the name of a variable.

If you want to specify the cursor from which the fieldname should be taken, you can use the following attribute:

- `cursor` specifies the name of the cursor from which the field is to be taken. The attribute has the type *ID*.

If you do not specify a cursor name, the value will be taken from the nearest cursor with a field of the specified name. Note that you can print titles “out of scope,” that is, you can print the default title of a field even though you are not within the scope of the cursor to which the field belongs. To do this, the title must be fully qualified with the cursor name(s).

Example

```
<title fieldname=JournalNumber cursor=Journal>
```

Often you will see a short form being used:

```
[cursorname.field]
```

```
[.field]
```

```
[VAR]
```

Example of usage “out of scope”:

```
<title fieldname=cursor1.cursor2.JournalNumber>
```

The `title` tag is closely related to the `text` tag, and you can therefore specify the same attributes as for `text`. These attributes all have the same meaning, except for:

- `title` has the type *STRING* and specifies a title, which is used if the system default title is empty.

Example

```
<title varname=VARNAME title="DefaultTitle">
```

or

```
[VARNAME]:"DefaultTitle"
```

All attributes that are nameless for the `text` tag are also nameless for the `title` tag.

Other attributes:

- `uppercase` has the type *BOOLEAN* and specifies whether the title should be writ- ten in uppercase. The default value is `false`.

Example

```
[.FieldName]:uppercase+
```

Length Constants

MPL enables you to define length constants. By assigning a name to a length that you want to use in more than one place, you ensure that any change that you make to the length will affect all occurrences.

MPL also has a number of predefined lengths that control the formatting. You can refer to these constants to align a length with an embedded length, or you can change the lengths to change the formatting of your print.

All length constants have an orientation (horizontal or vertical) and can only be used with the specified orientation.

Define

You can use the define tag to define a new length constant:

```
<define attributes>
```

Definitions with the `define` tag must be inserted in the beginning of parenthetical tags, that is, along with `default` statements. The defined constants apply within the parenthetical tag in which they are defined (and inside all embedded tags).

The tag has the following attributes:

- `name` specifies the name of the new constant. The attribute is mandatory and has the type *ID*.
- `value` specifies the value of the new constant. The attribute is mandatory and has the type *LENGTH*.
- `orientation` specifies the orientation of the new constant. The attribute has the type *ID* and can be assigned the values `vertical` or `horizontal`. The attribute is mandatory if the value is specified using a `grid` unit. If the attribute is not set, the orientation is derived from the first use of the constant.

Example

Consider the following:

```
<define name=nine value=9mm orientation=horizontal>
<define name=mylen value=1.5cm>
"Hello":indent=nine
"Hi!":indent=mylen
```

Both lengths can only be used horizontally: `nine` because it has been specified using the `orientation` attribute, and `mylen` because it is used in `indent`, which is a horizontal length. The resulting printout is:

Predefined Lengths

MPL has a number of predefined length constants. These lengths can be used in the same way as lengths defined using `define`, that is, they can be used as values in all length attributes. For example, you can use the radius of the island curves as margins:

```
<island leftmargin=IslandCornerRadius
       rightmargin=IslandCornerRadius>
   ...
<end island>
```

You often redefine these lengths (see the next section) and thus change the formatting of MPL layouts.

The following lengths are predefined in MPL:

- `InterColumnSpacing` specifies the distance between two columns if no column separators have been inserted between the columns. Moreover, the length defines the distance from a column to a column separator if any such has been inserted. It is the space that you can underline with the attributes to `hline`, namely `left` and `right`. The orientation of the length is horizontal and its default value is 4 points.
- `IslandTitleIndent` specifies the distance from the left-justified island title to the left edge of the island (and the distance from a right-justified island title to the right edge of the

island). For further illustration of this length, see the figure in “Island Lengths” in “Tips and Tricks.” The default value is 5 points.

- `IslandCornerRadius` specifies the radius of the quarter circles that make up the corners of the islands (when the `rounded` attribute has not been set to `false`). The default value is 5 points.

  The orientation of `IslandCornerRadius` is horizontal.

- `MinTextWidth` specifies the smallest width allowed for a (used) text block. If not set, the default value is 0 (zero) and text blocks will therefore never be wider than their actual width. The default value is 0.
- `MinFieldWidth` specifies the smallest width allowed for a field or a variable. As the contents of the field is not known during compilation/interpretation of the layout, this length ensures that fields always have a minimum width. The default width only provides a space for 5-6 characters with normal font which is rarely enough space for text fields. On the other hand it is also inexpedient to set a larger width than is actually needed; often the width of fields will be controlled by other attributes (for example, width and column stretchability). The default value is 18 points.
- `HlineSpacing` specifies the distance between lines in an `hline` with the multi attribute set to a higher value than 1. This constant is rarely used. The default value is 2 points.
- `HlineSkip` specifies the extra space after an `hline`. The default value of the constant is 0. This constant is rarely used.
- `HeaderSkip` specifies the extra space after headers, that is, the distance between a header and the main text (and between a page header and a block header). The default value is 4 points.
- `FooterSkip` specifies the extra space before footers, that is, the distance between the main text and a footer (and between a block footer and a page footer). The default value is 4 points.
- `MinBaseLineSkip` specifies the smallest distance between the baseline of text lines. Thus you set the line spacing by redefining this constant. Note that in certain cases the line spacing can seem smaller than `MinBaseLineSkip`. This usually occurs when two stacking tags of different height are placed alongside in a row. Here the baseline of the rows is the bottom text line by default. This means that the baseline of the row can be `MinBaseLineSkip` from the preceding baseline, without it being the case of the baseline of the top element. The default value is 10 points.

redefine

You can assign a new value to an existing length constant using this constant:

`<redefine` *`attributes`*`>`

The tag can occur in the same places as the `define` tag and has the same attributes:

- `name` specifies the name of the constant to which you want to assign a new value.

  The length constant should be known, meaning that it should either be predefined in MPL or be defined using a `define` tag in the scope where the `redefine` tag is specified. The attribute is mandatory and has the type *ID*.

- `value` specifies the new value that you want to assign to the constant. The attribute is mandatory and has the type *LENGTH*.
- `orientation` specifies the orientation of the new value of the constant. Because the constant keeps the orientation that has possibly been specified earlier, you should only use this attribute if the constant has not already been assigned an orientation.

  The attribute has the type *ID* and can be assigned the values `vertical` or `horizontal`. The attribute is mandatory if the value has been specified with a `grid` unit, and the length constant has not already been assigned an orientation. If `orientation` is not specified, and the constant does not have an assigned orientation, the orientation is derived from the first use of the constant.

If the constant has not already been assigned an orientation, a new orientation will not be tied to the old value. When the scope of the `redefine` tag is concluded, the constant will, therefore, still not have an assigned orientation.

The new value of the length constant will apply to the parenthetical tag in which the `redefine` tag is inserted. After the parenthetical tag, the original value applies.

Example 1

Consider the following:

```
<paper>
    <define name=mylen value=1cm>
    "1cm indent":indent=mylen
    <stack>
        <redefine name=mylen value=2cm>
        "2cm indent":indent=mylen
    <end stack>
    "1cm indent":indent=mylen
<end paper>
```

In the example the first and last text blocks are indented 1 cm, while the middle text block is indented 2 cm. Note how `stack` is exclusively used to create a local scope.

Example 2

In this example we use `redefine` to set some of the embedded length constants:

```
<redefine name=MinBaseLineSkip value=1cm> "Hello"
"Hello again"
<stack>
    <redefine name=InterColumnSpacing value=1cm>
    {:[|[]|[]|]
        <hline 2>;
        "1" "2";
        <hline 2>;
    }
    {
```

```
        "a" "b";
    }
<end stack>
{:[|[]|[]|]
    <hline 2>;
    "1" "2";
    <hline 2>;
}
```

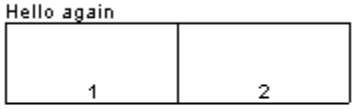
Redefining `MinBaseLineSkip` ensures that the line spacing is at least 1 cm (both between the simple text lines and the rows).

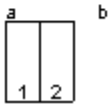
Redefining `InterColumnSpacing` changes the distance between columns to 1 cm. The difference is made more obvious by the fact that the same array is repeated after the stack in which the redefinition no longer applies.

The resulting printout is:

Hello

Hello again





Margins

The `margin` tag is part of the heading in the MPL definition. It should be specified immediately following the `page` tag and specifies the margins of the printout. If no margin tag is specified, the margin values from the Maconomy client window Paper Formats are used instead.

The tag is used as follows:

`<margins` *attributes*`>`

The tag has the following attributes which are all mandatory and have the type *LENGTH*:

- `top` specifies the top margin.
- `bottom` specifies the bottom margin.
- `left` specifies the left margin.
- `right` specifies the right margin.

Margins in Standard Prints

Some comments should be added to the use of the `margin` tag in connection with the standard prints. The behavior of this tag depends on the version of the current TPU and on the client that is using the standard printout.

Defaults

All attributes in all tags have a default value that is used if the attribute is not specified (not mandatory attributes). You can, of course, always change this value, but if you must do this often, it is easier to change the default value. A typical example is when you want to use another default font or font size. You can also use a style where islands always have the same inner margin and you therefore need to change the default value for the four attributes that decide the inner margins of the islands.

It often results in a more elegant output if the margin size is larger than zero. The disadvantage is the fact that rulers that are defined outside the island are not recognized inside the island if `leftmargin` or `rightmargin` has been set to a value that is other than zero.

You can change default values using the `default` tag that is specified in the beginning of parenthetical tags (for example, in the beginning of `paper` or in a repetition). The changing of default has an effect on the inner part of the parenthetical tag that is surrounded by the default statement (see the example below).

The syntax is:

```
<default attributes>
```

The tag has the following attributes that are all mandatory:

- `tag` specifies to what tag the attribute for which you want to change the default value belongs. In other words, the combination of `tag` and `attribute` identifies the attribute. The attribute has the type *ID*.
- `attribute` specifies for what attribute you want to change the default value. The attribute has the type *ID*.
- `value` specifies the default value for the specified attribute. The attribute type depends on what attribute is specified.

Because mandatory attributes must be specified, these have no default values and thus cannot be changed using the `default` tag.

Note that the defaults for `text` and `vline` will also affect column separators. You can specify default attribute values for the attributes in columns by using the `col` tag name (affecting both ruler and subruler definitions).

Example 1

The default font size for texts is 7 pt, whereas it is 9 pt for fields and variables. This results in an elegant layout when you print fixed texts in capitals, but it is not always as elegant if you want to print fixed texts in normal size.

The default font size can therefore be changed by writing:

```
<paper>
   <default tag=text attribute=fontsize value=9>
   ...
<end paper>
```

Because the default value is specified in the beginning of the `paper` tag, the default value will apply to the entire print definition.

As an exercise, you could insert this line into the time sheet layout defined earlier. Note that the island titles are also printed in 7 pt by default, and that it can also be useful to change this default value.

Example 2

The following example illustrates the scope of the `default` statement:

```
<paper>
   <island>
      <default tag=island
            attribute=leftmargin value=1cm>
      <default tag=island
            attribute=rightmargin value=1cm>
      <default tag=island
            attribute=topmargin value=5mm>
      <default tag=island
            attribute=bottommargin value=5mm>
      ...
      <island>
         "Hello"
         <island stretch- rightmargin=4cm>
            "Hi!"
         <end island>
      <end island>
      ...
   <end island>
<end paper>
```

Here the two inner islands will be assigned an inner margin, while the outer islands will not. The innermost island will have a right margin of 4 cm because of the specified attribute, while the other three margins are defined by the `default` statements.

Inheritance of attribute values

In MPL, tags can inherit values of certain attributes from their parent tags. The following section describe the rules guiding the inheritance of certain kinds of attributes.

Style Inheritance

In a layout you sometimes want to have different styles in different sections of the layout. For instance, you may want a different font for your page header or right justification in a column of an array. You can achieve this by specifying style attributes on parenthetical tags and ruler columns.

The attributes classified as style attributes are:

- justification
- fontname
- fontsize
- bold
- italic

- underline
- color
- rgb
- dateformat
- timeformat
- amountformat
- realformat
- integerformat
- booleanformat
- textdirection

In this section we will use the term *block* as a common name for all the parenthetical tags—`stack`, `array`, `repeating`, `conditional`, `canvas`, `island`, `page`, `frontpage`, `header`, `footer`, and `link`—including `layout`. Note, however, that `island` tags behave differently from the other block tags as described below.

Block Inheritance

The meaning of adding a style attribute to, for example, a `text` tag is to print the text with a given style. When you add a style attribute to a block, the result is that this style attribute is passed on to all tags within its scope:

```
<stack justification=center>
    "Text"
    cursor.field
    VAR
<end stack>
```

will give the same result as:

```
<stack>
    "Text":justification=center
    cursor.field:justification=center
    VAR:justification=center
<end stack>
```

Note, however, that inherited style attributes will never overwrite any attributes on the tag itself. Thus:

```
<stack
    justification=center> "Text"
    cursor.field:justification=left
    VAR
<end stack>
```

will give the same result as:

```
<stack>
    "Text":justification=center
```

```
    cursor.field:justification=left
    VAR:justification=center
<end stack>
```

Similarly, inherited style attributes do not overwrite default style definitions:

```
<stack justification=center>
    <default tag=field attribute=justification value=left> "Text"
    cursor.field
    VAR
<end stack>
```

will again give the same result as:

```
<stack>
    "Text":justification=center
    cursor.field:justification=left
    VAR:justification=center
<end stack>
```

Note that unlike the `default` definitions, the inherited style applies to all tags, regardless of type.

Column Inheritance

It is also possible to add style attributes to columns in ruler definitions:

```
<ruler Ruler1 [ [italic+] [fontsize=10] ]>
```

The result of this is that the first column is written in italic, and the second with font size 10.

Blocks placed in the columns will also inherit the tags and pass them on as described above. The rules about not overwriting tag and default attributes described above also apply to column inheritance.

However, column inheritance has precedence over general inheritance:

```
<ruler Ruler1 [ [] [fontsize=10] ]>
{ :Ruler1:fontsize=8
    "Text1" "Text2";
}
```

will result in “Text1” being printed with font size 8 and “Text2” with font size 10.

Islands

The `island` tag differs from the other parenthetical tags in that the style attributes specified for an island concern the island title and the island frame. Style attributes on islands are therefore not passed on to the elements within the island. The style attributes are, however, passed on to the island itself:

```
<stack justification=center fontsize=10>
    <island "IslandTitle" fontsize=20>
        "Text"
    <end island>
<end stack>
```

will give the same result as:

```
<stack>
   <island "IslandTitle" fontsize=20 justification=center
   titlejustification=center>
      "Text":fontsize=10
   <end island>
<end stack>
```

Style Precedence Summary

The style of a tag is decided as follows in order of precedence:

1. Style attributes on the tag itself.
2. Default definitions in the same scope as the tag.
3. Column style attributes for the column containing the tag, if any.
4. Style attributes for the surrounding parenthetical tag(s) including the `layout` tag.

Note that when a `color` attribute is inherited, it does not overwrite an `rgb` attribute, and vice-versa. Furthermore, where both the `color` and `rgb` attributes exist on a tag, the `rgb` attribute is always used in preference over the `color` tag.

Block Attributes

The attributes described in this section are defined on all non-fixed parenthetical tags: `stack`, `array`, `repeating`, `conditional`, `island`, and `canvas`.

- `keeptogether`: The attribute `keeptogether` has type *BOOLEAN* and indicates whether the block should be printed on the same page if possible. The default value is `false`. If the value is `true`, the block is moved to the top of the next page if the current page does not have room for it.

Example

```
<stack keeptogether+>
   "This block" "should be printed"
   "on the same page"
<end stack>
```

Note that for repeating tags, the actual number of elements inside the block is not known at the time of compilation. Therefore, adding a `keeptogether` to these tags will not guarantee that the entire repetition is written on the same page. It does however ensure that each element of the repetition is written on the same page if possible.

Example

```
      <repeating Cursor script="L_Cursor" keeptogether+>
         .fieldname
         "This field and this"
         "text should be printed"
         "on the same page, but the"
         "next field may be written"
```

```
    "on the next page"
<end repeating>
```

- `movepos`: The attribute `movepos` has type *BOOLEAN* and indicates whether the position of the print cursor should move after the block has been printed. The default value is `true`. If the value is `false`, the tags following the block are printed on top of it.

  This attribute enables you to print on top of background frames, watermark images, and so on.

  Note that `movepos` only works within one page. Adding a `movepos-` to a block therefore automatically triggers that `keeptogether` is true.

Example

```
<stack movepos->
    "This text will be overwritten"
<end stack>
"This text overwrites the text in the block"
```

- `stop`: The attribute `stop` has type *LENGTH* and indicates a distance to the bottom of the page. When a `stop` attribute is added to a block, it means that no contents of the block are written below the point given by the stop value.

  Some printers—among others, matrix printers—do not support this feature.

  The `stop` attribute can, together with the `goto` tag, be used to create border behavior within parenthetical tags (see `goto`).

Example

```
<stack stop=100pt>
    ...
<end stack>
```

Bi-Directional Printing

By default, all visible elements in mpl (e.g. text, expressions) are rendered from left to right, using a one-directional rendering algorithm. Whilst this kind of text rendering is suitable for the vast majority of the world's languages, some languages like Arabic or Hebrew require right to left rendering. To properly handle mixing left-to-right and right-to-left languages, MPL supports *bi-directional* printing that is controlled though the **textdirection** style attribute.

The **textdirection** attribute supports the one-directional left-to-right mode of printing, as well as two bi-directional modes of printing which are both capable of printing right-to-left languages correctly.

The succeeding paragraphs use English as the left-to-right language, and Arabic as the right-to-left language. The concepts discussed should apply to all left-to-right and right-to-left languages, respectively.

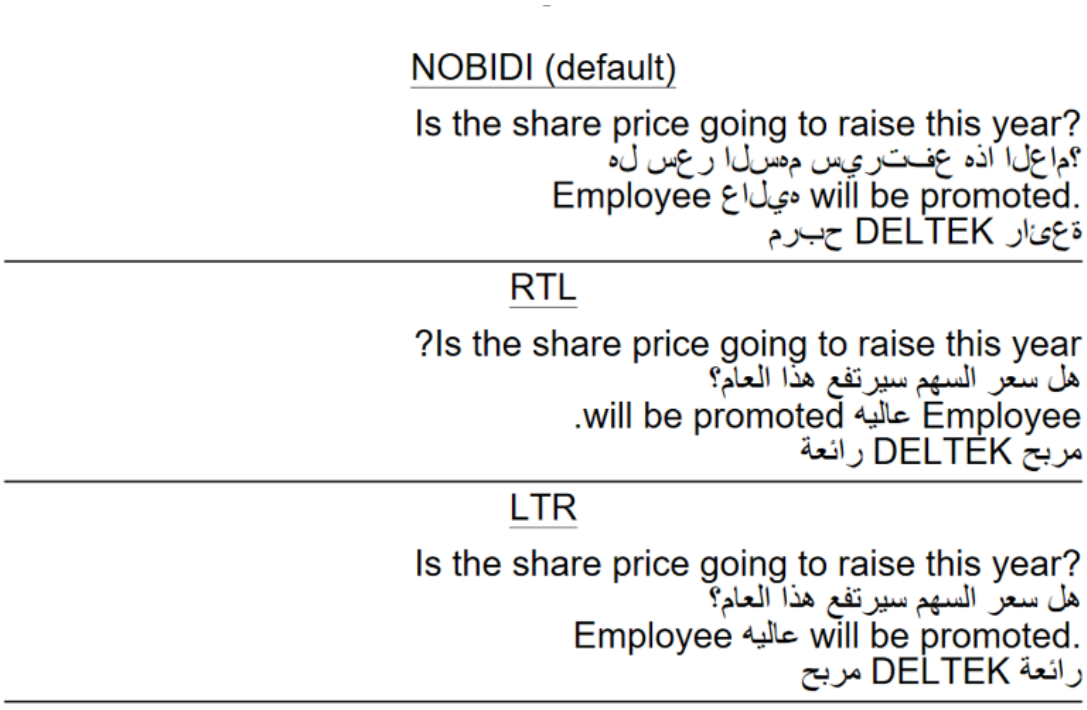
The **textdirection** attribute can assume any of the following values:

- **NOBIDI** – The one-directional (or non bi-directional) default that is equivalent to how MPL used to work prior to the introduction bi-directional printing. Text is always printed left to right independently of the language used.
- **RTL** – Uses bi-directional reordering with left-to-right preferential run direction. With this algorithm, the text is divided into English and Arabic blocks. Characters comprising Arabic blocks are always printed right to left. Moreover, the blocks themselves are printed in the **right to left** order. This setting is ideal for embedding English within Arabic text, where all text reads right to left apart from English words which read left to right.
- **LTR** – Uses bi-directional reordering with right-to-left preferential run direction. With this algorithm, the text is also divided into English and Arabic blocks. Characters comprising Arabic blocks are always printed right to left. Moreover, the blocks themselves are printed in the **left to right** order. This setting is ideal for embedding Arabic within English text, where all text reads left to right apart from Arabic words which read right to left.

To better understand these settings, consider the following sample sentences:

1. Is the share price going to raise this year?
2. هل سعر السهم سيرتفع هذا العام؟ *[English translation of 1]*
3. Employee عاليهه will be promoted. *[English: Employee* Aaliyah *will be promoted.]*
4. مربح DELTEK رائعة *[English: Company DELTEK is profitable]*

The following screenshot shows how these sentences will be printed in MPL using the three settings.



Note that when using LTR, sentences 1, 2 and 3 are printed correctly, but 4 is incorrect. On the other hand, when using RTL sentences 2 and 4 are correct, and 1 and 3 incorrect. NOBIDI is not suitable for printing Arabic at all, as it prints everything left to right.

To conclude: When using Arabic words in the English context, the LTR textdirection is appropriate. For embedding English words in the Arabic context, RTL is a better fit.

Setting Text Direction

You can use the **DefaultTextDirection** attribute to specify the system-wide Maconomy setting. To do this, update the MaconomyDir/Definitions/MaconomyCustom.ini server config file.

For example:

```
[MPLConfig]
DefaultTextDirection=LTR
```

You can then use **textdirection**, a regular style attribute in MPL, to make the specification on any of the various elements in an MPL layout.

For example:

```
{:r:textdirection=ltr
  "رحبا بالعالم! ماذا تفعل؟";"
}

<stack textdirection=rtl>
  "رحبا بالعالم! ماذا تفعل؟""
<end stack>

<default tag=text attribute=textdirection value=rtl>
"Hello world":textdirection=nobidi
```

Setting Text Justification

When working with right-to-left languages, you should also pay attention to the proper text justification. For example, a native speaker of Arabic would probably expect Arabic text to be right-justified. To adjust text justification, use the **justification** style attribute.

Standard Printouts in the Maconomy Clients for Windows and Java

When selecting “Print...” and “Print this” in the standard Maconomy client, margins are defined in MPL along with paper format and orientation. This is illustrated by the following MPL fragment:

```
<mpl 1>
<layout title="16 Columns Horizontal"
       print="Print Finance Report"
       originallayout="16 Columns Horizontal">
<page "A4" landscape>
<margins top=24pt bottom=30pt left=20pt right=20pt>
...
```

If `orientation` is left out, `portrait` is chosen as default. If `margins` is left out, the default is derived from the margins of the paper format. The paper format is mandatory.

In the Windows client, paper formats are defined in the window Paper Formats. This allows Maconomy users to define their own paper formats, and have MPL prints be formatted according to those definitions. On the other hand, paper formats need not be related to paper formats available on printers.

MPL is preformatted when it is installed, and all elements are given positions. This formatting is performed with no knowledge of the printer that will be used. In older versions of Maconomy (before TPU 27), the server formatted the print according to “standard” margins.

On Windows servers, the standard margins were derived from the default printer installed on the server. On UNIX they were given fixed values.

In later versions of Maconomy, printable elements are given positions relative to the paper edge. When the report is actually printed, it is adjusted to the physical margins of the selected printer. This ensures that prints look the same on all printers.

To ensure backward compatibility, a configuration option controls whether prints should be formatted in the new way or the old way. This option is set in the file Maconomy.ini located in the folder MaconomyDir/Definitions on the server. To place printable elements in the new way, set `RunTimePrinterDependence=true`.

Unless you have important reasons to do otherwise, you should always set this to `true`.

The meaning of the name is that the formatting of the report depends on the actual printer chosen at run time.

If a few of your reports have been aligned to specific printers in an old system, you can force the use of old formatting by setting the attribute `runtimeprinterdependence` to `false` on the `layout` tag:

```
<layout title="16 Columns Horizontal"
       print="Print Finance Report"
       originallayout="16 Columns Horizontal"
       runtimeprinterdependence->
```

If the report is printed from Maconomy client for the Java™ platform, PDF is produced and presented in Acrobat Reader. The PDF that Maconomy creates contains information on the dimensions of the paper format chosen. Acrobat Reader allows the user to choose that rotation and paper source should be selected according to these dimensions. Furthermore, you can scale the print; choosing not to scale ensures that the output matches the specifications of your MPL.

If the report is printed in the Windows client, you can displace the print in the Print Displacement window (found on the “File” menu). This is a last resort, only to be used if your print does not come out right on your printer. Print displacements are not saved as part of user settings.

The paper dimensions specified in MPL are ignored by the Windows client. Instead, you must manually choose paper format and orientation in the Page Setup window (on the “File” menu). The format chosen in the Page Setup window also determines the blue frame in Print Preview windows.

RGL

This is similar to standard prints as described above. RGL printouts are affected by the `RunTimePrinterDependence` option.

Universe Reports and the Analyzer

These are similar to standard prints as described above, except for the following: If a report is too wide, the paper size in the generated PDF grows with the content. In this case, using Acrobat Reader's scaling possibilities is recommended.

These are not affected by `RunTimePrinterDependence`.That is, it always works in the new way.

Filling in Forms

Printouts from Maconomy are often used to fill in preprinted forms. Forms can also be combined with printouts with a free design, for example, combining a special offer with a tear-off coupon, or an invitation to a business event with a registration form. In Denmark and other European countries an invoice is often accompanied by a giro form that allows a direct transfer of funds to the issuer of the invoice. The format of a giro form is defined in every detail to allow banks to read the forms automatically. This section describes the design of a specific sort of giro form, but the techniques employed are readily applicable to other preprinted forms.

A Giro Form

In the following example, it is assumed that all paper margins have been set to “0.”

```
<margins left=0cm right=0cm top=0cm bottom=0cm>
```

Furthermore, it is assumed that the font “OCR-B” is installed (below we use font “SAdvOCR-B”). The following lines of MPL will fill in a preprinted giro form at the bottom of an invoice:

```
<border bottom=101.6mm>
<stack>
    <default tag=field attribute=fontname value="SAdvOCR-B">
    <default tag=var attribute=fontname value="SAdvOCR-B">
    <default tag=text attribute=fontname value="SAdvOCR-B">
    <default tag=field attribute=fontsize value=14>
    <default tag=var attribute=fontsize value=14>
    <default tag=text attribute=fontsize value=14>
    <canvas height=101.6mm>
        .ThePaymentCustomer
    :(5mm,26mm):width=33.5mm:right:fontname="Courier":fontsize=9
        .InvoiceNumber
    :(45mm, 26mm):width=15mm:right:fontname="Courier":fontsize=9
        .InvoiceDate
    :(66.5mm, 26mm):width=18mm:right:fontname="Courier":fontsize=9
        <stack (10mm,34mm)>
            <default tag=var attribute=fontsize value=9>
            <default tag=var attribute=fontname value="Courier">
            <default tag=var attribute=width value=70mm> PaymentAddress1
            PaymentAddress2 PaymentAddress3 PaymentAddress4 PaymentAddress5
            PaymentAddress6 PaymentAddress7 PaymentAddress8
        <end stack>
        <stack (91mm,17mm)>
            <default tag=var attribute=fontsize value=9>
            <default tag=var attribute=fontname value="Courier">
            <default tag=var attribute=width value=40mm>
                CompanyGiroNumber:bold+
                CompanyAddress1
```

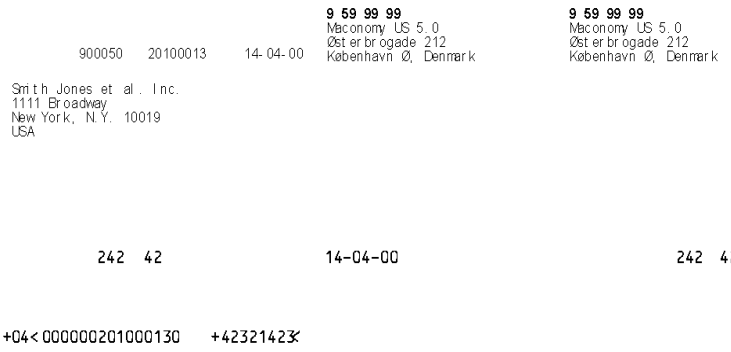
```
            CompanyAddress2

            CompanyAddress3

            CompanyAddress4

            CompanyAddress5

        <end stack>

        DollarAmount:(10.5mm,72mm):width=28.5mm:right
        CentAmount:(44mm,72mm):width=10mm:left

        .DueDate:(90mm,72mm):width=25mm:left

    -- The Receipt

        <stack (153mm,17mm)> -- the box of (148mm, 14.81mm)

            <default tag=var attribute=fontsize value=9>

            <default tag=var attribute=fontname value="Courier">

            <default tag=var attribute=width value=40mm> CompanyGiroNumber:bold+

            CompanyAddress1 CompanyAddress2 CompanyAddress3 CompanyAddress4
            CompanyAddress5

        <end stack> DollarAmount:(158.5mm,72mm):width=28.5mm:right
        CentAmount:(191.5mm,72mm):width=10mm:left "+04<":(7.62mm,90mm)

        .PayerIdentification:(17.78mm,90mm):width=5cm:left "+":(60.96mm,90mm)

        ExtAccountAccountNumberVar:(63.5mm,90mm):width=2cm:left
        "<":(81.28mm,90mm)

    <end canvas>
<end stack>
```

First a border is placed 101.6mm (the height of a giro form) from the bottom. This ensures that no elements before the border are printed on the preprinted giro form. Furthermore, it ensures that the stack that follows the border starts at exactly 101.6mm from the bottom of the paper.

The stack that follows the border has only one function: It ensures that the scope of the definitions in the stack is limited to the giro form. These definitions consist of a number of default declarations setting the standard font to “OCR-B” 14pt.

The formatting of the elements on the giro form is made by a canvas that places the individual texts and amounts correctly on the preprinted giro form. Note that the payment field and the payment receiver are stacks for which the content is simply stacked, but for which the stack itself is placed using coordinates in the canvas.

The result is as follows (slightly reduced in size):



Tips and Tricks

This section contains useful hints that can help you construct better and more elegant layouts using MPL.

Some of the most widely used layouts can initially seem difficult to construct using MPL. This section provides you with a number of tips and tricks and also gives you the necessary inspiration to get the most out of using MPL.

A layout can often be achieved using MPL—how you prefer to use MPL is naturally a matter of personal taste. But some methods can help to ensure that your MPL layouts are more easily maintained and updated.

Overlapping Fields

You often come across lines that need to include a debit field and a credit field. These lines can be constructed as follows:

```
{:[[stretch+][3cm][3cm]]
    .text
    .debit:zerosuppression+
    .credit:zerosuppression+;
}
```

Only one of the two fields can contain a value at a time, and you therefore waste space if you position them in separate columns. Instead you should let the two columns overlap each other.

Basically, this functionality is not supported by MPL, but if the two fields are placed in a canvas, they will overlap each other. You can then place the canvas in the same column:

```
{:[[stretch+][4cm]]
    .text
    <canvas height=10pt width=4cm>
        .debit:(0cm,0pt):width=3cm:zerosuppression+
        .credit:(1cm,0pt):width=3cm:zerosuppression+
    <end canvas>;
}
```

Paper Format Independence

Consider the following two MPL fragments:

```
{:[[75pt][stretch+][75pt]]
    .JobNumber JobName .ActivityNumber;
}
```

And:

```
{:[[75pt][380pt][75pt]]
    .JobNumber JobName .ActivityNumber;
}
```

These two definitions result in exactly the same formatting on a sheet of A4 paper with margins of 1 cm. Still, for several reasons, you should prefer the first definition:

- If you want to use the definition on another paper format (for example, landscape A4, US Letter), the first definition will automatically adjust to the paper format, whereas the other definition will have to be adapted, either because the entire sheet is not used, or because it has become too wide and cannot be compiled.
- If you change the margin in the definition or redefine the constant `InterColumnSpacing`, you will get an incorrect result.
- The first MPL fragment is a more logical representation than the second; that is, the length of `JobName` is unknown, and it should therefore have all excess space assigned to it, whereas the second fragment suggests that you should know for certain that 380 pt is an appropriate length for all occurrences of `JobName`.

Alignment of Headers and Footers

You often use headers in column headings and footers in totals. To align column headings with the contents, you can embed the repeating block in an array as in the following example:

```
{
   <repeating InvoiceLine script="InvoiceLineBlock">
      <header atstart+>
         "ITEM NO.""ORDERED"    "INVOICED"   "UNIT" "ITEM DESCRIPTION"  "UNIT
         PRICE"    "PRICE";
         <hline 7>;
      <end header>
      <footer>
         <hline 7>;
         ("Total Due":right):6   .TotalCurrency;
      <end footer>
      <conditional PricesOut script="PricesOut">
         .ItemNumber .NumberOrdered
      CorrectedNumberInvoiced .Unit
      ExternalItemText CorrectedUnitPrice
      CorrectedPriceWithoutDiscount;
      <end conditional>
   <end repeating>
}
```

You cannot use a similar technique for page headers and footers, since the `paper` tag cannot occur inside an array. If you want to align page headers or footers with the paper contents you must use rulers. An example of the use of this method can be seen in the following complete, however simple, layout for “Print Chart of Accounts”:

```
<mpl 2>
<layout title="Simple" print="Print_Chart_Of_Accounts"
       originallayout="Standard">
<page "A4">
<paper>
   <ruler main [[][stretch+][][15mm]]>
```

```
    <header onfirstpage+>
        {:main
            "ACCOUNT NO." "TEXT" "P&L, B/S" "TAX";
        }
    <end header>
    <repeating cursor=ACCOUNT>
        {:main
            .AccountNumber .AccountText
            .ProfitAndLossStatus .FinanceVATCode;
        }
    <end repeating>
<end paper>
```

Empty Stretchable Columns

In the footer of the time sheet print definition we used the `stretch` attribute to center the page number:

```
{:[[stretch+][][][][stretch+]]
    "" "-" PageNumber:center "-" "";
}
```

If you only want to insert a page number without hyphens on each page, `PageNumber:center` is of course sufficient, but by using a stretchable element you ensure that all three columns are centered equally. The purpose of this is to assign the necessary space to the two text blocks and the `PageNumber` variable, and all the remaining space is then equally distributed between the preceding and following columns.

In the example:

```
{:[[stretch+][][stretch+][][stretch+]]
    "" "a" "" "b" "";
}
```

we use the same method to ensure that “a” is positioned after a one-third indentation, and that “b” is positioned after a two-thirds indentation.

Stretchability and Embedding

If an array can be stretched—that is, at least one of its columns is stretchable—the array will stretch to the edges of the space that the surrounding elements provide. This is worth considering when arrays are contained in other columns. Columns in themselves are not stretchable, and the stretchability of the array will therefore not have any effect.

This point is illustrated in the following example:

```
<ruler main [[][]]>
{:main
    {:[[][stretch+]]
        "text";
        .field;
```

```
    }
        "another text";
}
```

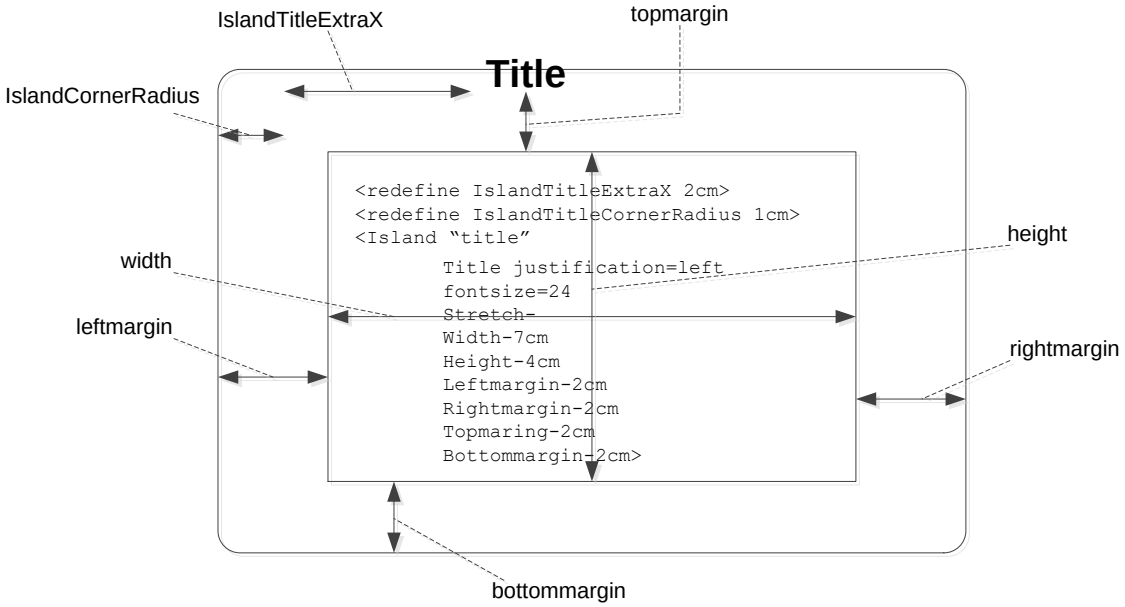
In this example, the array is contained in the first column of another array. Because the first column is not stretchable, the width of “text” plus the minimum width for fields is assigned to the inner array. If you change `main` to:

```
<ruler main [[stretch+][]]>
```

the column that contains the inner array will become stretchable. Therefore the column that contains `.field` can be stretched, and the field is assigned the width of the paper minus the width of “text” and “another text.”

Island Lengths

The formatting of an island is defined using a number of attributes and length constants. The following figure illustrates the functionality of the various lengths.



If an island cannot be stretched, it is assigned the greatest of the following values:

- 2 * IslandCornerRadius + 2 * IslandTitleIndent + the width of the title
- leftmargin + rightmargin + the width of the text (possibly as specified by the width attribute).

Fixed Frames and Watermarks

You cannot put repeating blocks inside islands. You can, however, put a fixed island frame around a repeating block by using the `movepos` attribute:

```
<ruler line [2pt[][][]]>
<repeating Employee script="L_EMPLOYEE">
    <header atstart+>
        <island height=4.5cm movepos->
```

```
        <end island>
        { :line
            "EMPLOYEE NO."   "NAME" "PHONE";
            <hline left- right->;
        }
    <end header>
    {:line
        .EmployeeNumber      .Name1 .Telephone;
    }
<end repeating>
```

In this example, the island frame is written in the header, but the print position is not moved. The header and content of the repeating block are thereby written inside the island frame. Note however that the frame is fixed, and that content from the repeating block therefore could exceed the frame, or text that follows the block could be written inside the block. This approach is therefore probably best suited for whole-page frames.

The same approach can be used for printing with background watermarks:

```
<ruler line [[3cm][stretch+][5cm fontsize=20]]>
<repeating Employee script="L_Employee">
    <header atstart+>
        <stack movepos->
            <image title="Images\MaconomyBackground.JPG" height=18cm>
        <end stack>
    <end header>
    {:line:keeptogether+
        .EmployeeNumber      .Name1 .Telephone;
        ""      .Name1 "";
        ""      .Name1 "";
    }
<end repeating>
```

Dynamic Frames

To put repeating blocks into a frame which expands with the size of the block, you can use a combination of the `hline` and `vline` tags and the `pagebottom` attribute:

```
<ruler line [|[][stretch+][5cm]|]>
<redefine HeaderSkip 0pt>
<redefine FooterSkip 0pt>
<repeating Employee script="L_Employee">
    <header atstart+>
        {:line
            <skip 3mm>;
            <hline>;
            "EMPLOYEE NO."   "NAME" "PHONE";
```

```
            <hline left- right->;
        }
    <end header>
    <footer atend+ pagebottom->
        <hline>
    <end footer>
    {:line
        .EmployeeNumber      .Name1 .Telephone;
    }
<end repeating>
```

In this example, the first `hline` in the header creates the top of the frame, and the `vline` tags in the ruler create the sides of the frame. The bottom of the frame is created by the `hline` in the footer, and setting the `pagebottom` attribute to `false` in the footer ensures that its content attaches itself to the frame. The two `redefine` tags are there to avoid space between header and vertical lines, respectively.

Using stop and goto

The `stop` attribute, together with the `goto` tag, enables you to write blocks on preprinted paper (for example, checks):

```
<repeating Employee script="L_Employee" stop=20cm>
    <header atstart+>
        ...
    <end header>
    <footer atend+ pagebottom->
        <hline>
    <end footer>
    {:line
        .EmployeeNumber      .Name1 .Telephone;
    }
<end repeating>
<goto bottom=10cm>
<hline>
"Check":color=red:italic+:fontsize=24:right
```

The `stop` attribute ensures that no content of the block is written below a line 20cm from the page bottom. After the loop has finished, the `goto` tag moves the printing position to a line 10cm from the page bottom.

Grammar

This section provides an easy-to-use overview of the syntax of MPL. The first section contains a definition of the language syntax in the form of a BNF (Backus-Naur Form) diagram, and the following section contains an overview of the MPL tags and their attributes.

Backus-Naur Form (BNF)

The syntax is described in a variant of the Backus-Naur Form. The notation (abbreviated BNF) is a precise method for describing valid language constructions.

Terminal symbols are written in the `typewriter font`, non-terminal symbols as regular text, and grammar primitives in *CAPITALS*.

Terminal symbols are characters that should be written as they appear. Non-terminal symbols are symbols that are defined by the grammar. Grammar primitives are similar to non-terminal symbols, but they are typically more fundamental and are defined less formally.

If a symbol *s* can be left out, it is written as s^0. If *s* can appear any number of times (0 or more), it is written as s*, and if *s* should only appear at least once (that is, once or more), it is written as s+. If either s_1 or s_2 is to appear, it is written as $s_1|s_2$.

In BNF, you would normally use the form [s], but because the square brackets occur in the grammar, that notation would make the grammar hard to read.

Syntax

The grammar primitives *ID*, *STRING*, *INTEGER*, *LENGTH*, and *BOOLEAN* are explained in “Attribute Values” in “Basic MPL.” *STRING2* is a string that is specified in apostrophes, for example, `'abc'`. *RGB* is a triple of integers denoting the percentage of the colors red, green, and blue, respectively, for example, (100,50,0).

Start	<mpl2>
	<`layout` Attributes>
	Page
	Margin^0
	Grid^0
	Frontpage^0
	Paper
Page	<`page` Attributes>
Margin	<`margins` Attributes>
Grid	<`grid` Attributes>

Frontpage <frontpage Attributes>

Definition*

Elem*

<end frontpage>

Paper : paper Attributes> Definition*

Elem+

<end paper>

| <paper Attributes> Definition* FrameOrg

<end paper>

Header : <header Attributes> Definition*

Elem*

<end header>

Footer : <footer Attributes> Definition*

Elem*

<end footer>

Canvas : <canvas Attributes> Definition*

Elem*

<end canvas>

Conditional : <conditional Attributes> Definition*

Elem*

<end conditional>

Repeating : <repeating Attributes> Definition*

Elem*

<end conditional>

Island : <island Attributes> Definition*

Elem*

<end island>

Stack : <stack Attributes> Definition*

Elem*

<end stack>

Span : <span Attributes> Definition*

Elem

<end span>

| (Definition* Elem) Shortattributes

Array : <array Attributes>

Elem+

<end array>

| { Shortattributes Elem+ }

Elem : Conditional

Paper : <paper Attributes> Definition*

Elem+

<end paper>

| <paper Attributes> Definition* FrameOrg

<end paper>

Header : <header Attributes> Definition*

Elem*

<end header>

Footer <footer Attributes> Definition*

Elem*

<end footer>

Canvas : <canvas Attributes> Definition*

Elem+

<end canvas>

Conditional : <conditional Attributes>

Definition*

Elem*

<end conditional>

Repeating : <repeating Attributes>

Definition*

Elem*

<end repeating>

Island : <island Attributes> Definition*

Elem*

<end island>

Stack : <stack Attributes> Definition*

Elem*

<end stack>

Span : <span Attributes> Definition*

Elem

<end span>

| (Definition* Elem) Shortattributes

Array : <array Attributes>

Elem+

<end array>

| { Shortattributes Elem+ }

Elem : Conditional

| Repeating

| Array

| Text

| Field

| Variable

| Island

| Stack

| Skip

| Canvas

| Line

| Border

| Newpage

| Header

| Footer

| Row

| Span

| Hline

| Vline

| Image

| Goto

| Title

Row : <`row`> Attributes>

Elem+

<end row>

| Elem+; Short attributes

Text : <`text` Attributes>

| *STRING* Short attributes

Text2 : <`text2` Attributes>

| *STRING2* Short attributes

Field	:	<field Attributes>
	|	*ID⁰.ID* Short attributes
Variable	:	<var Attributes>
	|	*ID* Short attributes
Title		<title Attributes>
	|	[*ID*] Short attributes
	|	*ID⁰.ID* Short attributes
Line	:	<line Attributes>
Border	:	<border Attributes>
Newpage	:	<newpage Attributes>
Hline	:	<hline Attributes>
Vline	:	<vline Attributes>
	|	| Short attributes
Skip	:	<skip Attributes>
Image	:	<image Attributes>
Goto	:	<goto Attributes>
Definition	:	defaultdef
	|	RulerDef
	|	Setlength
Setlength	:	<define Attributes>
	|	<redefine Attributes>
Defaultdef	:	<default Attributes>
RulerDef	:	<ruler Attributes>
	|	<subruler Attributes>
Attributes	:	Attribute*
Shortattributes	:	(:Attribute)*
Attribute	:	*ID* = Attributevalue

| *ID*+

| *ID*-

| Attributevalue

Attribute value : BOOLEAN

| INTEGER

| STRING

| *ID*

| *RGB*

| Length

| (Length, Length)

| List

Paper : `<paper` Attributes>

Definition*

Elem+

<end paper>

Header : `<header` Attributes>

Definition*

Elem*

<end header>

Footer : `<footer` Attributes>

Definition*

Elem*

<end footer>

Canvas : `<canvas` Attributes>

Definition*

Elem+

<end canvas>

Conditional : `<conditional` Attributes>

Definition*

Elem*

<end conditional>

Repeating : `<repeating` Attributes>

Definition*

Elem*

<end repeating>

Island : `<island` Attributes>

Definition*

Elem*

<end island>

Stack : `<stack` Attributes>

Definition*

Elem*

<end stack>

Span : `<span` Attributes>

Definition*

Elem

<end span>

Array : `<array` Attributes>

Elem+

<end array>

| { Shortattributes Elem+ }

Elem : Conditional

| Repeating

| Array

| Text

| Field

| Variable

| Island

| Stack

| Skip

| Canvas

| Line

| Border

| Newpage

| Header

| Footer

| Row

| Span

| Hline

| Vline

| Image

| Goto

| Title

Row : `<row` Attributes>

Elem+

<end row>

| Elem+; Short attributes

Text : `<text` Attributes>

| *STRING* Short attributes

Text2 : `<text2` Attributes>

| *STRING2* Short attributes

Field	:	`<field` Attributes>
	|	*ID*^{0}*.ID* Short attributes
Variable	:	`<var` Attributes>
	|	*ID* Short attributes
Title	:	`<title` Attributes>
	|	[*ID*] Short attributes
	|	[*ID*^{0} *.ID*] Short attributes
Line	:	`<line` Attributes>
Border	:	`<border` Attributes>
Newpage	:	`<newpage` Attributes>
Hline	:	`<hline` Attributes>
Vline	:	`<vline` Attributes>
	|	| Short attributes
Skip	:	`<skip` Attributes>
Image	:	`<image` Attributes>
Goto	:	`<goto` Attributes>
Definition	:	Defaultdef
	|	RulerDef
	|	Setlength
Setlength		`<define` Attributes
	|	`<redefine` Attributes>
Defaultdef	:	`<default` Attributes>
RulerDef	:	`<ruler` Attributes>
	|	`<subruler` Attributes>
Attributes	:	Attribute*
Shortattributes	:	(:Attribute)*
Attribute	:	*ID* = Attributevalue

	|	*ID*+
	|	*ID*-
	|	Attributevalue
Attribute value	:	BOOLEAN
	|	INTEGER
	|	STRING
	|	*ID*
	|	*RGB*
	|	Length
	|	(Length, Length)
	|	List
	|	Ruler
	|	SubRuler
Length	:	LENGTH
	|	*ID*
Ruler	:	[(ColSep^0 Col)+ ColSep^0]
Col	:	[Attributes]
ColSep	:	LENGTH
	|	Vline
	|	Text
SubRuler	:	[Col+]
List	:	[*ID* (, *ID*)*]

Attribute List

The following table is a list of tags that are used in MPL and the attributes that belong to each tag. To improve readability, some attributes, which are shared between many tags, have been assembled into attribute groups.

The first column displays the tag name, and the second column displays attributes or groups of attributes that you can assign to the tag in question. Attribute group names are written in *italics*. If an attribute group is available in a tag, it means that all of the attributes that belong to that group

are available in the tag. At the end of this section is a table of the attributes that are available within each attribute group.

The third, fourth, and fifth columns contain information about the use of the current tag: “M” stands for mandatory, “N” stands for nameless, and “S” specifies that the tag is nameless in a possible short form. The sixth column displays the attribute value type.

Tag	Attribute	M	N	S	Type
array	ruler pos indent height width baseline	 M	N N	S S	*ID >> Ruler* *POS* *LENGTH* *LENGTH* *LENGTH* *ID*
assign *(MPL 4)*	var value	M M	N N		*ID* *EXPRESSION*
border	top bottom *style*				*LENGTH* *LENGTH*
canvas	pos indent height width *style* *block*	M M M	N		*POS* *LENGTH* *LENGTH*
concat	pos indent width justification style *wrap*		N N	 S	*POS* *LENGTH* *LENGTH* *ID*
conditional	variable script indent height negate width baseline *style* *block* expression *(MPL 4)*	M	N N N		*ID* *STRING* *LENGTH* *LENGTH* *BOOLEAN* *LENGTH* *ID* *EXPRESSION*
cursor *(MPL 4)*	query name showmaincursor	M	N		*ID* *ID* *BOOLEAN*
default	attribute value tag	M M M			*ID* *Any*

Tag	Attribute	M	N	S	Type
define	name value orientation	M M	N N		*ID* *LENGTH* *ID*
eval *(MPL 4)*	expression zerosuppression pos indent width justification *style* *wrap*	M M	N N	 S	*EXPRESSION* *BOOLEAN* *POS* *LENGTH* *LENGTH* *ID*
field *(desupported in MPL 4)*	data cursor zerosuppression pos indent width	M M	N N	 S	*ID* *ID* *BOOLEAN* *POS* *LENGTH* *LENGTH*
footer	onlastpage attend script height *style*		 N		*BOOLEAN* *BOOLEAN* *STRING* *LENGTH*
frame	height width *style*				
framecolumn	style				
framerow	style				
grid	hor ver pagebottom	M M			*LENGTH* *LENGTH* *BOOLEAN*
goto	top bottom				*LENGTH* *LENGTH*
header	onfirstpage atstart script height *style*		 N N		*BOOLEAN* *BOOLEAN* *STRING* *LENGTH*
hline	multi columns left right *style*	 M	 N		*INTEGER* *INTEGER* *BOOLEAN* *BOOLEAN*

Tag	Attribute	M	N	S	Type
image	title fieldname varname cursorname justification width height scalewidth scaleheight expression *(MPL 4)*		N N N		*STRING* *ID* *ID* *ID* *ID* *LENGTH* *LENGTH* *BOOLEAN* *EXPRESSION*
island	title titlejustification titlecolor titlergb rounded pos indent height width fontname fontsize stretch italic underline bold topmargin bottommargin leftmargin rightmargin justification baseline *block*	 M 	N N 		*STRING* *ID* *ID* *RGB* *BOOLEAN* *POS* *LENGTH* *LENGTH* *LENGTH* *STRING* *INTEGER* *BOOLEAN* *BOOLEAN* *BOOLEAN* *BOOLEAN* *LENGTH* *LENGTH* *LENGTH* *LENGTH* *ID* *ID*
layout	title print originallayout	M M M	N		*STRING* *STRING* *STRING*
line	start end *style*	M M			*POS* *POS*
margins	top bottom left right	M M M M			*LENGTH* *LENGTH* *LENGTH* *LENGTH*
page	name orientation	M	N N		*STRING* *ID*

Tag	Attribute	M	N	S	Type
paper	cursor script *style*		N N		*ID* *STRING*
parameter	name vale	M M	N N		*ID* *EXPRESSION*
query	name	M	N		*ID*
redefine	name value orientation	M M	N N		*ID* *LENGTH* *ID*
repeating	cursor script groupby indent height width *style* *block*	M	N N N		*ID* *STRING* *LIST* *LENGTH* *LENGTH* *LENGTH*
row	align height *style* *block*		N	S	*ID* *LENGTH*
ruler	name value	M M	N N		*ID* *ID >> Ruler*
skip	height	M	N		*LENGTH*
span	columns	M	N	S	*LENGTH*
stack	indent pos script height width baseline *style* *block*	M	N N		*LENGTH* *POS* *STRING* *LENGTH* *LENGTH* *ID*
subrulers	name value parent	M M M	N N		*ID* *SUBRULER* *ID*
text	title pos indent width justification	M M	N N	 S	*STRING* *POS* *LENGTH* *LENGTH* *ID*

Tag	Attribute	M	N	S	Type
	fontname fontsize bold italic underline		N	S	*STRING* *INTEGER* *BOOLEAN* *BOOLEAN* *BOOLEAN*
text2	same attributes as text				
val *(MPL 4)*	name value type	M M	N N		*ID* *EXPRESSION* *ID*
var *(desupported in MPL 4)*	data zerosuppression pos indent width justification *style*	M M	N N	S S	*ID* *BOOLEAN* *POS* *LENGTH* *LENGTH* *ID*
var *(MPL 4)*	name value type	M M	N N		*ID* *EXPRESSION* *ID*
vline	justification style		N	S	*ID*
Col	stretch width style		 N	 S	*BOOLEAN* *LENGTH*
ColSpec	interval	M	N	S	*INTERVAL >>* *INTEGER*
title	fieldname varname cursorname uppercase title width *style*			 N	*ID* *ID* *ID* *STRING* *LENGTH*

Attribute Group List

The following is a table of attribute groups that can be used for several tags, namely the ones where the group name in question appears in italics in the Attribute column in the attribute list. In this table, the first column displays the attribute group name. The Attribute column shows the attributes that are available in tags where the attribute group in question is available.

Group	Attribute	M	N	S	Type
style	justification[4] fontname fontsize bold italic underline color rgb dateformat[5] timeformat amountformat realformat integerformat booleanformat		N	S	*ID* *STRING* *INTEGER* *BOOLEAN* *BOOLEAN* *BOOLEAN* *ID* *RGB* *STRING* *STRING* *STRING* *STRING* *STRING* *STRING*
block	keeptogether movepos stop				*BOOLEAN* *BOOLEAN* *LENGTH*
wrap	wrap lines height				*BOOLEAN* *INTEGER* *LENGHT*

MPL for Universe Reports

This section describes a number of MPL extensions that have been created to support Maconomy Universe Reports. Universe Reports are used in the Maconomy Portal to create dynamic, interlinked reporting tools using HTML or PDF to display output. The layout of such reports is based on MPL. However, you cannot use all MPL commands in Universe Reports. For a list of the MPL features that are not available in Universe Reports, see “Standard MPL vs. Reporting MPL.” Conversely, you cannot use the MPL extensions for Universe Reports in standard MPL used for the layout of printed reports, unless specifically stated.

A Universe Report is generated using MQL statements. The Maconomy Query Language (MQL) is a language for interacting with the Maconomy database. In structure, it is similar to SQL, but there are a number of differences.

- With MQL, you do not have to be aware of the relation structure of the database.
- The data model is separated from the command—in MQL, the data model is defined in universes.
- MQL is aware of the Maconomy types, and the types are validated before command execution.
- MQL is database-independent.

[4] This attribute is nameless for the long form of the tag text, and the short form of the tags text, field, and var.

[5] Format attributes are used in MPL for Universe Reporting. See “Formats.”

In MQL, data is selected using an `mselect` statement. For more information, see the “MQL Language Reference.”

The extensions can be divided into three categories: links, tables, and charts. The following sections describe these extensions.

Links

Using the parenthetical tag `link`, you can add a hypertext link in the report. The syntax of links is the following:

```
<link attributes>
   ...
<end link>
```

The hyperlink that is specified in the attributes can be a regular URL, possibly with parameters; it can be a JavaScript; or it can be a link to a Portal component or another Universe report. The `link` tag has the following attributes:

- `href` specifies the address (URL) of the link. The attribute has the type *STRING*. If the tag is specified, the link opens a browser window to the specified URL. If a `parameters` attribute is specified, the parameters are added to the URL in the form described below.
- `script` specifies a piece of JavaScript code. The attribute has the type *STRING*. If the tag is specified, the specified script is executed when the link is clicked. You cannot combine this attribute with other attributes. For example:

  ```
  field2:script="var x='Hello World';alert(x);"
  ```

- `component` specifies the name of a Portal component that will be opened when the link is clicked. The attribute has the type *STRING*. This attribute is usually used in conjunction with the `parameters` attribute (see below).
- `report` specifies the name of another report that will be called when the link is clicked. This attribute is usually used in conjunction with the `reportsetup` and `parameters` attributes (see below).
- `reportsetup` specifies how a linked report should be called. This attribute currently takes one parameter of the type *PARAMLIST*, namely `rpAction`. If the `reportsetup` attribute is not specified, the linked report will open with an empty selection criteria selection window. This attribute only has effect when used in conjunction with `report`. See the description of `parameters` below for an example.

  This parameter takes the following values:

  - `drill`: If this value is specified, the report will take the parameters specified through the `parameters` attribute and use them in the selection criteria specification window, if possible, and open the report directly, if possible (that is, if all mandatory selection fields are completed). Furthermore, it will be possible to return to the original report (a drill-down path is displayed).
  - `drillaround`: This value essentially does the same as `drill`, except that the old report is not entered in the drill-down path, but replaced by the new report. This is useful if two different reports present the same data in different ways and therefor exist at the same level in the drill-down hierarchy.
  - `edit`: This value forces the user to specify selection criteria by opening the selection criteria specification window before drilling down.

- `parameters` specifies parameters that are to be added to the URL specified in the `href`, `component`, or `report` attributes. The attribute has the type *PARAMLIST*. The parameters can be the content of fields in the underlying MQL `mselect` statement, variables, or text. When this attribute is specified, the parameters are added to the URL (as specified in the `href`, `report`, or `component` attribute) in the following form:

  ```
  <URL>?param1=value1&param2=value2
  ```

  Example of the use of `parameters` in conjunction with `report` as a link from a table field:

  ```
  .EmployeeNumber
      :report="Demo::ByJob"
      :parameters=
          [
              ParEmployeeNumber=.EmployeeNumber
          ]
  :reportsetup=[rpAction="drill"]
  ```

- `transferparameters`: This attribute is of type *BOOLEAN*. If this attribute is set to `true`, all MRL parameters from the current report are automatically transferred as parameters in the link to the next report. This is useful if universe reports that consist of several MRL definitions share parameters. The attribute only has effect for report links.
- `target` specifies the target window of the link. The attribute is of type ID, and possible values are `self` (replace the current window or frame), `new` (show in a new browser window), and `rightside` (show to the right of the Portal). The attribute is ignored for report links and has no effect for script links. For `component` and `href` links, the default is `rightside`.

Note that the link attributes can be added to a number of different MPL tags. Apart from the `link` tag, the following tags accept the link attributes. Links are of course only relevant in reports in HTML or PDF formats:

- `text` (see “Texts”)
- `field` (see “Fields”)
- `var` (see “Variables”)
- `text2` (see “Alternative text tag”)
- `image` (see “Images”)
- `title` (see “Title”)
- `fields` (see “Tables”)

Tables

Tables are a very common feature in reports, and the generation of tables has been simplified with the MPL `table` tag. This tag is a form of preformatted template, where an HTML or PDF table is generated by iterating through a cursor. This section describes:

- Syntax of the table tag
- Layout of individual fields
- Attributes applicable to fields tag
- Attributes applicable to individual fields

- Table headers and footers
- Example

Syntax of the Table Tag

The general syntax of tables is the following:

```
<table cursor=cursorName>
    <fields>
        .field1
        .field2
            ...
        .fieldN
    <end fields>
<end table>
```

The attribute `cursor` is mandatory and should be the name of the cursor that contains the `mselect` result columns to be shown in the table. The cursor is defined in MQL using the `as cursor` keyword.

You can only use the `table` tag in stacking environments; that is, you cannot place a table in an array row or a canvas.

If you want to show all of the fields that are contained in the current cursor, the following short-hand is available:

```
<table cursor=cursorName>
    <fields>
        *
    <end fields>
<end table>
```

The order of the fields in the table is determined by the way in which the underlying `mselect` statement chose to sort the current cursor.

You can exclude fields or apply various formatting by combining the asterisk `(*)` notation with explicit field references (the `table` tag is left out from the examples from now on):

```
<fields>
    *
    .field5:hidden+
    .field6:color=red
<end fields>
```

This will display all fields with the exception of `field5`. Furthermore, `field6` is colored red.

There can be a maximum of one asterisk in a `fields` section. All fields in the current cursor that have not been explicitly referenced anywhere in the `fields` section will be placed in the location of the asterisk, in the order in which they appear in the cursor.

You can nest tables. If the underlying `mselect` statement has defined two cursors, you can add a new `table` tag within the first `table` tag using the new cursor name like this:

```
<table cursor=cursorName1>
```

```
    <fields>
        .field1
        .field2
    <end fields>
    <table cursor=cursorName2>
        <fields>
            .field1
            .field2
                ...
            .fieldN
        <end fields>
    <end table>
<end table>
```

This enables you to group entries within a table. For an example of this, see “Example” below.

Layout of Individual Fields

You can hide or apply special formatting to individual fields by applying attributes directly to the fields, for example:

```
<fields>
    .field1:bold+
    .field2:bold+
    .field3:hidden+
    *
    .field6:color=red
<end fields>
```

The following attributes can be applied to individual fields:

- href, script, parameters, component, report

  These tags allow you to add links to the fields in the table. For more information, see “Links.”

- Attributes in the attribute group “style”

  For more information, see “Attribute Group List.”

Note that you can place other elements than fields in the `fields` section of the `table` tag. Texts, titles, images, and stacks can also be inserted in the report columns. When using a stack, the stack can only contain elements of the same type, for example, a stack of images. For more information, see “Texts,” “Title,” “Images,” and “Stacks.” For instance, you can use a stack like this:

```
<fields>
    .field1:bold+
    .field2:bold+
    <stack>
        .field3
```

```
        .field4
        .field5
    <end stack>
    .field6:color=red
<end fields>
```

This will place the contents of fields 3 to 5 in the same table cell, written above one another.

Note that both the long form and the short form can be used when specifying tags. In the examples in this section, only the short forms are used.

Attributes Applicable to Fields Tag

Instead of applying attributes to the individual fields, you can add attributes to the `fields` tag. Attributes specified for individual fields will then overwrite the attributes specified in the `fields` tag. In the following example, all fields except `field6` are colored blue:

```
<fields:color=blue>
    *
    .field6:color=red
<end fields>
```

The attributes which you can use are the same as those mentioned above for the individual fields, with the exception of `hidden`. See also “Style Inheritance.”

Formats

The output formats for fields (and variables) are by default the formats that are set up by the client that is connected to the server. For Universe Reports, the client is the web server, so the format settings are defined by the M-Script format settings. You can change these formats locally in a report by using these format attributes:

- dateformat
- timeformat
- amountformat
- realformat
- integerformat
- booleanformat

These attributes are all of type *STRING*, and their values should be a (valid) format string for the type that they represent. See the M-Script manual for a description of format strings.

The format attributes are part of the *style* attributes (see “Attribute Group List”) and can therefore be added to any tag in MPL. They are inherited in the same way as *style* attributes.

Example

```
<fields dateformat="E:DDMMYY">
    .dateField1
    .dateField2:dateformat="E:YYDDMM"
<end fields>
```

Here `dateField1` will be written as "DD:MM:YY," whereas `dateField2` will be written as "YY:DD:MM".

As a special case, specifying the empty string as the value for a format attribute is interpreted as the default format attribute value.

Example

```
<fields dateformat="E:DDMMYY">
    .dateField1
    .dateField2:dateformat=""
<end fields>
```

Here `dateField1` will be written as "DD:MM:YY," whereas `dateField2` will be written in the default format.

A warning is issued if you specify an invalid format value or if you specify a format attribute on a field (or variable), where the type of the field does not match the type of the attribute (for example, a date format attribute on a time field). In both cases, the format attribute that caused the error is ignored.

Attributes Applicable to Individual Fields

All of the attributes mentioned in “Attributes Applicable to fields Tag” and “Formats” can be applied to individual fields as well. Apart from those, the following attributes can be applied to individual fields:

- headertitle

  Using this attribute, you can assign your own header to the column represented by the field. For instance, if you want to change the default header “Description” for the field `.Activity`, you can write the following:

  ```
  .Activity:headertitle="Act. Name"
  ```

  This example will output the header “Act. Name” above the “Activity” column. The alternative to using the `headertitle` attribute is to specify a field in a separate header (see “Table Headers and Footers”). A separate header is, for example, useful if you want to add attributes to the header text. If, for instance, you add:

  ```
  :color=redto the example above, the field will be colored red, not the
  header text.
  ```

- footerfunction

  Using this attribute, you can replace the default footer function with another function specified in the MQL `aggregate` option. The default function is `SUM`. If the specified function is not defined in the underlying `mselect` statement, no footer is displayed. For more information, see the MQL Reference manual. For example:

  ```
  .RegTime:footerfunction="Max"
  ```

  This example will output the maximum number of registered hours below the “RegTime” column. The alternative to using the `footerfunction` attribute is to specify a field in a separate footer (see “Table Headers and Footers”).

- hidden

  Using this attribute, you can hide the current field. See “Syntax of the Table Tag” for an example of this.

- tooltip

Using this attribute, you can add a tooltip text to the current field. The type of this attribute is *STRING*. The tooltip is naturally only displayed in HTML output, even though you can assign it in MPL layouts for printed output. For example:

```
.field2:tooltip="Click here to open the MyCustomer component showing the
current customer."
```

Table Headers and Footers

Headers and footers are enabled by default in tables. If you do not want a header or footer in a table or nested table, add the attribute `header-` and/or `footer-` to the `table` tag:

```
<table cursor=cursorName header- footer- >
```

However, you can also add your own headers and footers to the table using the parenthetical tags `header` and `footer`. The following example shows the syntax:

```
<header>
    "Title for field1"
    "Title for field2"
    *
    "Title for fieldN"
<end header>
<footer> "Total Time"
    *
    .RegTime$Sum:color=red
<end footer>
```

In the footer example, the field `.RegTime$Sum` is the result of the underlying MQL creating a sum field. Any function that is used in a footer must be made available by the underlying MQL statements (see also the description of the attribute `footerfunction` above).

For nested tables, headers can only be specified for the outermost table. Footers, however, can (must) be specified for each table. You can add multiple headers to a table by repeating the `header` tag. Note also that the use of the tags `header` and `footer` overwrites the specifications in the `headertitle` and `footerfunction` attributes.

You can make headings in a header span multiple columns using the same technique as for columns in regular MPL (see “Spanning Columns”). For example:

```
<header>
    "Title for field1"
    ("Title for field2 and field3"):2
<end header>
```

If a span exceeds the number of available columns, the span is automatically reduced to match the number of available columns.

Furthermore, you can use fields, titles, and images in headers and footers. For example:

```
<header>
    "Employee Name"
    *
    <image title="MyImages\regtime.png" scaleheight- height=100pt>
```

```
<end header>
```

You can add more headers to a table, simply by specifying multiple `header` tags. If you want to use the title of a field as a header, you can reference the field's title using the common MPL syntax: `[.field]`. You can also reference fields that are beyond the scope of the current cursor by qualifying the name with the full cursor path to the desired field, relative to the context that you are in. For example, if the outer table uses cursor `EmployeeCursor`, but you really want the title of a field in the `embedded cursor ActivityCursor`, you can use the following path within a `header` tag:

```
[EmployeeCursor.ActivityCursor.Activity]
```

Example

Below is a basic example of the use of the `table` tag. This example will be expanded as all of the features of this tag are explained.

```
<table cursor=EmployeeCursor>
    <fields>
        .EmployeeName
        .Activity
        .RegTime
    <end fields>
<end table>
```

This example will yield a simple list of employee registrations:

Employee Name	Description	Registered Time
Joe Daniels	Carpentry	20.0
Jack Johnson	Photo Copies	26.0
Jack Johnson	Consulting	25.0
Jack Johnson	Photo Copies	4.0
Sally Rogers	Photo Copies	5.0
		80.0

Now we want to avoid the repetition of the employee name, and we want different titles for our header. The titles are by default the visible title (label) of the referenced field. Fortunately, our `mselect` statement has defined another cursor as well, so we use a nested table with the cursor `ActivityCursor`:

```
<table cursor=EmployeeCursor>
<header>
    "Employee Name"
    "Act. Name"
    "Registered Time"
<end header>
```

```
    <fields>
        .EmployeeName
    <end fields>
    <table cursor=ActivityCursor>
        <fields>
            .Activity
            .RegTime
        <end fields>
    <end table>
<end table>
```

Employee Name	Description	Registered Time
Joe Daniels	Carpentry	20.0
		20.0
Jack Johnson	Photo Copies	26.0
	Consulting	25.0
	Photo Copies	4.0
		55.0
Sally Rogers	Photo Copies	5.0
		5.0
		80.0

To remove the subtotals from the table and leave only the grand total, add the attribute `footer-` to the inner table. Also add a bolded footer text to the first column:

```
<table cursor=EmployeeCursor>
<header>
    "Employee Name"
    "Act. Name"
    "Registered Time"
<end header>
<footer>
    "Total Time":bold+
<end footer>
    <fields>
        .EmployeeName
    <end fields>
    <table cursor=ActivityCursor footer->
```

```
        <fields>
            .Activity
            .RegTime
        <end fields>
    <end table>
<end table>
```

Employee Name	Description	Registered Time
Joe Daniels	Carpentry	20.0
Jack Johnson	Photo Copies	26.0
	Consulting	25.0
	Photo Copies	4.0
Sally Rogers	Photo Copies	5.0
		80.0

If you want to consolidate the activities (for example, to avoid multiple entries for “Photo Copies” for the same employee), an additional cursor is needed in the underlying `mselect` statement, which could then be incorporated in another nested table in MPL.

Charts

Charts are diagrams which depict report data graphically. Currently, MPL supports pie charts and bar charts.

Charts are currently considered one entity. This means that they cannot be broken into several pages in a PDF. If a chart does not fit on a page, the excess part is cut off. This is similar to specifying an image that is too large for one page in regular MPL. However, this restriction will be resolved in the near future.

Pie Charts

The syntax of a pie chart is as follows:

```
<piechart cursor=cursorName height=xxxpt>
    <legend>
        .field3
    <end legend>
    <fields>
        .field1
        .field2
    <end fields>
<end piechart>
```

This will draw two pie charts for `field1` and `field2`, respectively. The default title of the field is printed above each chart.

The pie chart definition is similar to the table definition, with the `legend` tag taking the place of the `header` tag. Therefore, the description of the pie chart will focus on where the specification is different from the table definition.

The `cursor` attribute is not mandatory, but is usually assigned a value. See “Cursorless Charts.” The cursor contains the columns available in the result of the underlying `mselect` statement. The

attribute `height` is mandatory. The `height` attribute denotes the height of the chart excluding any legend.

The `legend` tag prints a legend that consists of colored boxes and the default text of the referenced field. You can in principle add any MPL inside the `legend` tag, but you should keep the legend simple. For instance, you can change the text that is displayed using the attribute `headertitle` (see the description in "Attributes Applicable to Individual Fields").

Also note in this connection that the `fields` section is not mandatory. This enables you to print the chart legend separately from the actual chart, even in another frame (see the example in "Frames").

The titles of the referenced fields are shown in the pie chart. You can add links to the fields, and you can add style information similar to ordinary table fields. The style is reflected in the title of the fields.

Note that the use of the `*` wildcard in the list of fields in the `fields` tag is not supported. The wildcard is ignored.

Bar Charts

The bar chart definition is very similar to that of the pie chart:

```
<barchart cursor=cursorName height=xxxpt>
    <legend>
        .field3
    <end legend>
    <fields>
        .field1
    <end fields>
<end barchart>
```

The `height` attribute specifies the height of the y axis of the chart.

You can use bar charts in two ways. In the preceding example, the title of the field that is referenced in the `legend` tag is printed along with a colored box below the x axis as a caption for the chart. The individual values of `field1` are used as bars in the chart.

If you add more fields in the `fields` section of the bar chart definition, a bar will be shown with the sum of the referenced columns—that is, a bar for each referenced field. Each bar will have a different color, which is also reflected in the bar chart legend.

The bar chart has two additional, optional attributes:

- `interbarspace`: This attribute is of the type *LENGTH*. If you, for example, assign a value of `2pt` to this attribute, the individual bars in the chart will be spaced 2 points apart. For example:

  ```
  <barchart cursor=EmployeeCursor height=100pt interbarspace=2pt>
  ```

- `maxbarwidth`: This attribute is of the type *LENGTH*. The bar width is by default calculated by dividing the width of the bar chart by the number of bars (and any space between). However, when the number of bars is low, this can produce very wide bars. To avoid this, MPL operates with a default maximum bar width of 20pt. You can change the maximum bar width by adding the attribute `maxbarwidth` to the `barchart` tag. For example:

```
<barchart cursor=EmployeeCursor height=100pt interbarspace=2pt
maxbarwidth=16pt>
```

Cursorless Charts

Even though charts are usually based on cursors, you can create them without using a cursor from an `mselect` statement, simply by omitting the `cursor` attribute from the `pie chart` or `barchart` tags. When no cursor is used, the chart picks its data from the currently active row in the `MSelect` statement.

When you specify a chart without a cursor, the `headertitle` attribute and the `legend` tag change their meaning: `headertitle` (or, if omitted, the default title) will function as the legend beneath the x axis in bar charts, and the legend is used as a title and is printed above the chart.

Frames

In the Portal, *dashboards* are used for organizing multiple Portal components and reports into one view. Using MPL, you can show the top-level layout of Universe Reports in a dashboard-style layout using *frames*. This can, for example, be used for combining multiple reports in one view.

A frame has a fixed size. If the size is insufficient to hold the content of the frame, a scrollbar will appear in the HTML output from the report. In PDF output, overflow is clipped, and little red arrows will indicate that clipping has been performed.

You can only specify framesets (the sequence of frames) at the top level, in the `paper` tag. Dashboard-style framesets are organized using `framerow` and `framecolumn` tags, which can be nested arbitrarily. The `frame` tags are placed inside the frame rows and columns.

All the frame-related tags can take the *style* attributes (which will affect the content). For more information, see “Attribute Group List.”

framerow

A frame row is created using:

```
<framerow attributes>
    ...
<end framerow>
```

Frame rows can be nested in other frame rows or columns. Note that frames defined within a `framecolumn` without a surrounding `framerow` tag will be organized in rows.

framecolumn

A dashboard column is created using:

```
<framecolumn attributes>
    ...
<end framecolumn>
```

Frame columns can be nested in other frame columns or rows. Note that frames defined within a `framerow` without a surrounding `framecolumn` tag will be organized in columns.

frame

The frame itself is specified using the parenthetical `frame` tag:

```
<frame attributes>
    ...
<end frame>
```

The behavior of a frame can be modified using the following attributes:

- `height` is used to specify the height of the frame's contents. If this attribute is not specified, the contents of the frame will determine its height. The attribute has the type *LENGTH*.
- `width` is used to specify the width of the frame's contents. If this attribute is not specified, the contents of the frame will determine its width. The attribute has the type *LENGTH*.

The *`framedefinition`* is any content—for example, text, a pie chart, or a table.

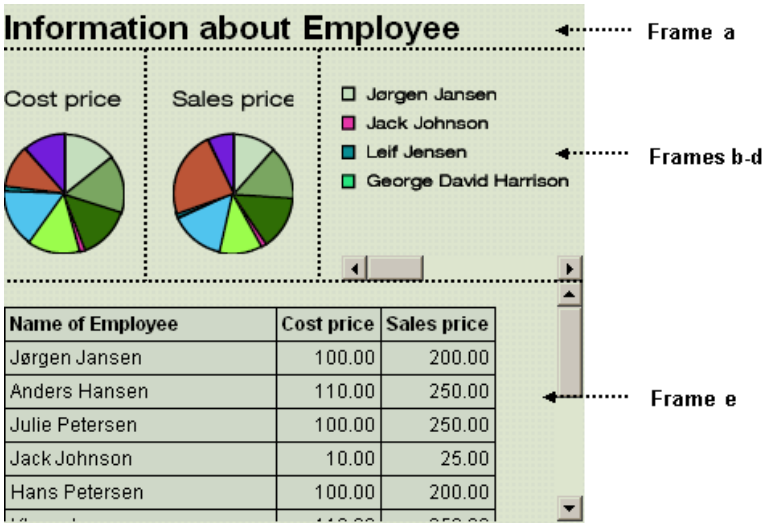
Example

The following is an example of a dashboard that is created by using MPL (annotations are described below):

```
<mpl 2>
<layout>
<paper>
    1    <framecolumn>
    2    <frame>
        "Information about Employee":fontsize=16:bold+
    <end frame>
    3    <framerow>
        4   <frame width=70pt>
            <piechart Employeecursor height=50pt >
                <fields>
                    .costprice:headertitle="Cost price"
                <end fields>
            <end piechart>
        <end frame>
        <frame width=70pt>
            <piechart Employeecursor height=50pt >
                <fields>
                    .salesprice:headertitle="Sales price"
                <end fields>
            <end piechart>
        <end frame>
        <frame width=100pt fontsize=7>
            <piechart Employeecursor height=70pt>
                <legend>
                    .name1
                <end legend>
```

```
            <end piechart>
        <end frame>
    <end framerow>
        5  <frame height=100pt width=240pt>
            <table Employeecursor>
                <fields>
                    .name1:headertitle="Name of Employee"
                    .costprice:headertitle="Cost price"
                    .salesprice:headertitle="Sales price"
                <end fields>
            <end table>
        <end frame>
    6  <end framecolumn>
<end paper>
```

This layout yields the following result, where the defined frames are marked:



Annotations

1. First, a frame column is defined.
2. In the column, frame *a* is inserted (the heading text).
3. Then a frame row within the frame column is defined, containing three frames (*b*-*d*). The frames are “stacked” horizontally within the row; that is, they are placed next to each other, moving right.

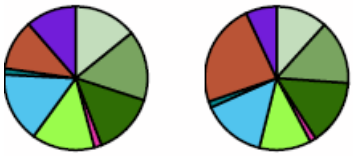
   Note that a specific width is applied to the frames within the frame row.
4. After the `<end framerow>` tag, a new frame is inserted. This is stacked in the column; that is, it is placed below the preceding frame. This frame (that contains the table) is given a fixed height and width; this is typically necessary because the size of a table cannot be calculated without knowing its content. It is not necessary to give a size to the frames that contain the pie charts (frames *b* and *c*). We have chosen to do it in this layout, however, to align the pie chart row with the table below.

5. The frame column is ended.

The PDF version of the same layout is shown below. Note the little red markers that indicate that part of the frame is missing.

Information about Employee

Cost price Sales price



- Jørgen Jansen
- Jack Johnson
- Leif Jensen
- George David Harrison

Name of Employee	Cost price	Sales price
Jørgen Jansen	100.00	200.00
Anders Hansen	110.00	250.00
Julie Petersen	100.00	250.00
Jack Johnson	10.00	25.00
Hans Petersen	100.00	200.00
Klaus Jørgensen	110.00	250.00
Leif Jensen	10.00	25.00
Kurt Christensen	80.00	400.00
Michael Roberts	80.00	120.00

Calculating MPL Attribute Values Using M-Script

This section contains a description of how to modify your layouts on the fly by calling M-Script functions from the layout. This functionality was introduced to support Traffic Lighting, for example, assigning a certain color to the content of a field depending on the value of the field. However, the functionality has a number of additional uses.

To calculate the value of an attribute, assign the attribute value to an M-Script function and pass parameters to the function like this: `attributename=f(a1,...,an)` where `f` is any M-Script function name that is defined in the M-Script package that is associated with the current report, and the arguments `a1 - an` consist of fields or constants.

Do not use this functionality for complex M-Script calculations or large SQL queries, because performance will be too slow. To access the database, use the MQL part of the report.

You can achieve special layout effects by using global M-Script variables; for example you can color every other table row. However, because of the internal structure of the compiled MPL layouts, unpredictable side effects may occur, of which you should be aware.

Attributes and Return Value Types

The following table lists the MPL attributes for which a value can be calculated in M- Script and the expected type of the value that is returned by the M-Script function.

Attribute Name	Return Value Type
rgb | titlergb	struct: {r: v1, g: v2, b: v3} -where `vn` are integers
color | titlergb | fontname | component | href | target | report | justification | `title` (on both `text` and `island` tags) | headertitle	*STRING*
bold | italic | underline | zerosuppression	*BOOLEAN*
fontsize	*INTEGER*
interbarspace | maxbarwidth	struct { val: v, unit: s } -where `v` is a *REAL* (the length value) and `s` is a *STRING* (`"pt"`, `"in"`, `"mm"` or `"cm"`)

Attribute Name	Return Value Type
pos	struct { x: v1, xunit: s1, y: v2, xunit: s2 }-where $v_1$and v_2 are *REAL* (the length values) and s1 and $s_2$are *STRING* ("pt", "in", "mm" or "cm")

Note that no automatic conversion is performed. A function that returns `{val:9,unit:"pt"}` to the attribute `interbarspace` will give a runtime error.

Instead, return `{val:real(9),unit:"pt"}` (or just `9.0`).

The type of the parameters that are passed to the M-Script function is the original database type.

Returning Null from an M-Script Function

To call an M-Script function without applying the return value to any tag, use the tag `functioncall`. This functionality is, for example, used for initializing and updating global variables in the M-script before other functions are called.

```
functioncall < functioncall call=f(…) >
```

- `call` is the only attribute, and it is mandatory. The short form is simply `f(…)`. If the tag is used in rows, it counts as an element, but nothing is shown. Note that `f` must be a function, but the value is ignored, so you might just return `null`.

Example

Consider the following example, which calculates the color of the header title depending on the value of the employee's degree of utilization:

```
<table EmployeeCursor>
    <fields>
        .employeenumber
        .Utilization:color=calcColor(.Utilization)
    <end fields>
<end table>
```

For every employee in the cursor `EmployeeCursor`, a row is built that consists of the employee number and the utilization of the employee. The color of the utilization percentage (the `color` attribute) is determined by the M-script `calcColor`, which takes the MQL field `.Utilization` as its parameter.

The M-script, which is defined in the M-Script package associated with the report, might look like this:

```
public function calcColor(util)
{
    if (util == null)
        return {r:0, g:0, b:0};
    if (util < 50)
        return {r:100, g:0, b:0};
    if (util < 70)
        return {r:100, g:100, b:0};
```

```
    if (util > 100)
        return {r:0, g:100, b:0};
    return {r:0, g:0, b:0};
}
```

If the utilization is `null` or between 70 and 100, the color black is returned. If the utilization is less than 50, red is returned; if it is less than 70, yellow is returned; if it is above 100, green is returned.

Note that the `util` value is also tested for `null`. This is to catch instances where the `calcColor` function is called with no value. For instance, in the preceding example, MPL automatically transfers a default `headertitle` attribute, as if the layout had said:

```
.Utilization:headertitle=<column name>:color=calcColor(.Utilization)
```

Therefore, the M-Script function is called implicitly by the `headertitle` attribute, but without a valid parameter. This case is caught by the M-script testing for `null` and then returning the color black.

Note that attributes (in the preceding example: the `color` attribute) are copied to headers and footers, and therefore the function call is copied as well. However, the parameter that is sent to the M-script might be out of scope in the header and footer. This is a good reason to include the possibility of a `null` value in your M-script.

Standard MPL vs. Reporting MPL

This section contains a description of the differences between standard MPL and the MPL extensions for Universe Reporting.

- Variables

  In MPL for Universe Reporting, no MSL variables are available. The only variables that are available are system variables, of which there currently is only one: `pagenumber`.

- Field sizes

  Because MPL for Universe Reporting is interpreted, rather than compiled and run as for standard prints, the real sizes of the fields are used when formatting. This means that you do not need to set the `width` attribute for fields, unless you want a column to be smaller or bigger than the text size. If you give an element in the `fields` section of a `table` tag a specific width using the `width` attribute, the output is truncated to that width.

- gotoand border

  The `goto` and `border` tags are currently not supported in MPL for Universe Reporting. If you use a `goto` or `border` tag in a layout, it will be ignored.

- Page width

  In standard MPL, an error is issued if the width of an element exceeds the content width (or height). For example, you will get an error or warning if you specify an island to be wider than the page. In MPL for Universe Reporting, these checks are omitted with regard to page width. This means that the MPL compiler does not check whether the layout exceeds the page width. You will, however, still get a warning/error if an element exceeds the width or height of another element, such as an island.

- layoutand page

  The attributes `title`, `print`, and `originallayout`for the `layout`tag are mandatory in standard MPL, but have no meaning in the reporting context and are consequently ignored if they are specified.

  In standard MPL, you must specify the `page` tag. This is not the case in MPL for Universe Reporting. If you omit the `page` tag, the page size is assumed to be A4. The page size is used for calculating the location of page breaks when creating PDF output, for calculating the width of islands, the maximum width of charts, and for right- and center-justification of elements.

- Warnings

  When standard layouts are imported (during installation or by running `MaconomyServer -UP`), warnings are active; that is, layouts with warnings will be compiled and installed. In all other cases warnings are treated as errors, and the import fails.

  In MPL for Universe Reporting, when you run a report for which the layout produces warnings, you will still get an output. However, when you install a report, all warnings are treated as errors.

- Miscellaneous

  The justification of islands has no effect on the HTML output of MPL for Universe Reporting. Islands are always right-justified.

MPL Version 3

MPL version 3 is a major new version of MPL; it breaks backward compatibility in some cases, and it provides new functionality. Since the entire MPL framework has been rewritten and moved to the Java platform, there are a number of areas (not only in the MPL language, but also in font handling and so forth) that have been affected by this. Additionally, a number of new features have been added to MPL 3.

The following sections describe the new futures as well as the changed functionality.

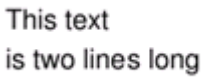
New Features

wrap Attribute for <text>, <field>, and <var> Tags

New in Maconomy version 12 is the ability to have some text fields that can contain multiple lines (using linefeed/newline characters). To be able to preserve this new functionality when printing, the `<text>`, `<field>`, and `<var>` tags now support a new Boolean attribute `wrap`. The attribute has type *BOOLEAN*.

When this attribute is `true`, text that is too long to be displayed in the width that was allocated to the `text`, `field`, or `var` tag will be wrapped, resulting in text that can be more than one line tall. Also any newline characters in the text will result in a line break.

For example, "This text\nis two lines long":wrap+ will result in this output:



The behavior of a `<text>`, `<field>`, and `<var>` tag with the `wrap` attribute, when these attributes are defined:

- When `width` is not specified, the tag with `wrap+` will expand to fill all available width. See below for implications when used in a column of an array.
- When `height` is not specified for a `<var>` or `<field>` tag, the height of the tag becomes unknown (because it is not known at compile time how much text the tag will contain at run time). Therefore, when the tag is used in a context where a fixed height is required (the `canvas` tag, for instance, or a `stack` with a specific height set), you will get a compile error. For the `text` tag, the height is calculated automatically, based on what the width is set to (recall that it will expand to fill).
- `lines` is an additional attribute that you can use to specify the height of the tag in terms of the number of lines that can be shown. You can specify only `lines` or `height`, not both. The attribute has type *INTEGER*.

Furthermore, you should keep these points in mind:

- Because the content of wrapped `<field>` and `<var>` tags is unknown at compile time, the baseline for these is always set to the top line, meaning that they will be aligned with other items in a row using the first line as the baseline.
- Even in a `canvas` tag, you are not required to specify `width`. The tag will simply stretch from its horizontal position to the rightmost boundary of the canvas. However, you must specify `height` in a canvas.

Implicit Conversion to Stretching Column

When a wrapped `text` field or `var` occurs as a column in an array without an explicit width, and that column does not have any width set in a ruler definition, the ruler definition for that column is implicitly converted to become stretchable. This is done to make as much width available for multiline texts as possible.

In the following example, `ruler1` and `ruler2` are equivalent, because `ruler1` becomes `[[stretch+]]` by implicit conversion. Because the field in `ruler3` specifies a width, the column in `ruler3` is not made stretchable. The same applies to the column in `ruler4` because that column definition specifies a width. In effect, the arrays that use `ruler1` and `ruler2` will stretch to fit the entire the entire page width, while the arrays that use `ruler1` and `ruler2` will become 5 cm wide.

```
<ruler ruler1 [[]]>
<ruler ruler2 [[stretch+]]>
<ruler ruler3 [[]]>
<ruler ruler4 [[5cm]]>
<repeating MyCursor>
    {:ruler1
        .MultilineField:wrap+;
    }
    {:ruler2
        .MultilineField:wrap+;
    }
    {:ruler3
        .MultilineField:wrap+:width=5cm;
    }
    {:ruler4
        .MultilineField:wrap+;
    }
<end repeating>
```

The `wrap` attribute propagates upward in the hierarchy, so if, for example, a `stack` contains a wrapped text, and the `width` attribute is not specified on the `stack`, this `stack` will also be stretched and make columns stretch.

In a context where a subruler contains a tag with the `wrap` attribute on, the implicit stretching will affect the root ruler column definition if there is no explicit width set (as demonstrated above). In the following example, column 1 and 2 of `ruler1` will become stretchable because the first column of `ruler2` spans those two columns:

```
<ruler ruler1 [[][][]]>
<subruler ruler2 parent=ruler1 [[1:2]]>
{:ruler1
    ...
}
{:ruler2
```

```
    .MultilineField:wrap+;
}
```

concat

A new parenthetical tag, <concat>, is introduced. The <concat> tag is not a stacking tag; instead, it can concatenate text from <text>, <field>, and <var> tags to form one single text. This text behaves as if it had the wrap attribute set to true; that is, it will wrap if contents become too wide or newline characters are encountered.

You can specify style attributes on the concat tag; the tags within will inherit these. Only <text>, <field>, and <var> tags can be put inside concat. Most attributes on the <text>, <field>, and <var> tags inside a concat will be ignored (for example the width attribute), but most style attributes will be applied, for example:

```
<concat>"We confirm that " .Quantity1:bold+ " items have been ordered."<end
concat>
```

Indent and justification style attributes are ignored for tags inside the concat (but can be applied to the concat tag itself).

In addition to style attributes, you can specify these attributes for the concat tag:

- width specifies whether the concat tag should have a fixed width. If this attribute is not specified, it will expand to use all available width in its context. If it is used in a column in an array, the concat behaves like a <text>, <var>, or <field> tag with regard to implicit stretching of those columns; see “wrap Attribute for <text>, <field>, and <var> Tags.” The attribute has type *LENGTH*.
- height specifies a fixed height for the tag. If the height is not sufficient to display the contents when printing, only the lines that are completely visible within the available height will be shown. Alternatively, instead of using height, you can use the lines attribute instead. Only lines or height can be provided, not both. The attribute has type *LENGTH*.
- lines specifies the height of the tag in terms of number of lines that can be shown. Note that the height of the lines depends on the font that is associated with the concat tag. If other font sizes are used in the tags inside the concat tag, the number of lines actually shown may not match the specified number of lines. Only lines or height can be provided, not both. The attribute has type *INTEGER*.
- indent specifies an extra indentation of the text. The attribute has type *LENGTH*.
- pos specifies the positioning of a text element of a canvas. The attribute is nameless and has the type *POS*. It can only be used when the text appears on a canvas, and is mandatory, if so (see “Canvas” for further information about the canvas).

Furthermore, keep these points in mind:

- Because the content of a concat tag is generally unknown at compile time, the baseline for the concat tag is always set to the top line, meaning that it will be aligned with other items in a row using the first line as the baseline. Furthermore, using different font sizes and fonts inside a concat can reduce the accuracy of the baseline calculations.
- Even in a canvas tag, width need not be specified. The tag will simply stretch from its horizontal position to the rightmost boundary of the canvas. height must be specified in a canvas though.

Conditionals

The `conditional` tag has been modified in several ways. You can now use a conditional on fields (not just variables), and you can also use string fields or variables as a conditional. These two new attributes have been added to the `conditional` tag:

- `field` specifies the name of the field that must be `true` for the contents to be printed and has the type *STRING*. Note that unlike the `variable` attribute, this attribute cannot be nameless.
- `cursor`. If you want to specify the cursor from which the field should be taken, you can use this attribute. The attribute has the type *ID*. If you do not specify a cursor name, the value will be taken from the nearest cursor with a field of the specified name.

Furthermore, you can now negate a conditional (this functionality previously existed in MPL for Universe Reports only).

If a string variable or field is used in the conditional, the conditional will evaluate to `false` if the string is empty (that is, ""); otherwise, the conditional evaluates to `true`.

See also “Skipping False Conditionals” for the changed functionality for the `conditional` tag: a false conditional no longer skips space.

Paper Orientation Change

You can now switch the paper orientation of a layout when using the `newpage` tag in the outermost parenthetical tag (that is, the paper). This attribute is now recognized:

You can set the orientation to landscape or portrait. The new page after the page break will have this orientation.

Here is a simple example:

```
<paper>
...
<newpage orientation=landscape> -- switch to landscape for the rest of the print
<end paper>
```

Note that there are some side effects:

Page headers and footers will be shown using the least width available throughout the entire print; that is, they will not become wider when switching from portrait to landscape orientation.

Be careful when reusing ruler definitions across different paper orientations. As a general rule, avoid using the same ruler both on a page with landscape orientation and on a page with portrait orientation. Doing this can lead to rulers becoming too wide (in this case a compile error will be issued).

Varying Header/Footer Height

Headers and footers no longer need to have a fixed height. This allows for using wrapped texts and the `concat` tag in a `header` or a `footer`. The header and footer heights are calculated for every page at run time. This also means that they can potentially become taller than a page, leading to a runtime error.

Text Length Greater than 255 Characters

MPL 3 supports texts that are longer than 255 characters. Because fields and variables in Maconomy cannot yet be longer than 255 characters, this change is only visible when using the `text` or the `concat` tags.

Scopes

Definitions of lengths, defaults, and rulers in MPL (the `define`, `redefine`, `default`, `ruler`, and `subruler` tags) can now occur anywhere. They are no longer limited to be specified in the beginning of a parenthetical tag (such as `stack`). The definitions take effect from the point of definition until the end of the parenthetical tag in which they occur. For example:

```
<stack>
    <stack>
        {:ruler1
            ... -- error: ruler1 is not yet defined
        }
        <ruler ruler1 [[][]]>
        {:ruler1
            ... -- fine: ruler1 is now defined
        }
        <ruler ruler1 [[][][]]> -- redefinition of ruler1
        {:ruler1
            ... -- fine: using the new definition of ruler1
        }
    <end stack>
    {:ruler1
            ... -- error: ruler1 is no longer defined
    }
<end stack>
```

goto is No Longer Allowed in Headers and Footers

You can no longer use the `goto` tag in headers and footers. However, because the positions of headers and footers and footers are (mostly) fixed on the page, it is often fairly easy to translate a `goto` tag into a `skip` tag that will place the contents in the same place on the page. For more precise control, you can apply a <`stack movepos->` with a `canvas` tag inside. Both solutions are demonstrated in this example:

```
<margins top=46pt bottom=30pt left=44pt right=20pt>
...
<header height=110pt>
    <stack movepos->
        <canvas>
            "At (44pt,146pt) of the page":(0pt,100pt)
```

```
        <end canvas>
    <end stack>
    "Line 1 in the header"
    "Line 2 in the header"
    <skip 80pt>
    "At (44pt,146pt) of the page"
<end header>
```

You must specify the height of the header manually, because the stack with `movepos-` does not influence the height of the header. The top margin of the page plays a role in determining the position as you can see in the preceding example; while we place the text at 100pt in the canvas, the real position on the page is 146pt (100pt+46pt top margin). Furthermore, the `<skip 80pt>` relies on each of the two lines being 10pt tall (the default baseline skip for small font sizes).

<newpage> in Row No Longer Allowed

You can no longer make a new page in a row, so this example:

```
{:ruler1
    ...
    <newpage>;
    ...
}
```

must be rewritten as:

```
{:ruler1
        ...
}
<newpage>
{:ruler1
    ...
}
```

Skipping False Conditionals

The functionality of the `conditional` tag has been improved in MPL 3 (see also “Conditionals”). An important change that can impact existing layouts is that when the content of a conditional is not shown (because the conditional evaluates to `false`), the conditional leaves no blank space where it would normally have been. For example, in MPL 2 a conditional such as:

```
{
    "First row";
    <conditional SomeVariable> Second row";
    <end conditional> "Third row";
}
```

would always produce three rows; even if the conditional evaluated to `false`, and the contents of the second row were not printed, there would be an empty row instead. Now, this row is skipped

entirely. This applies to conditionals in general, no matter where they occur: If the content of a conditional is skipped, the surrounding block shrinks. This means that for an island such as this:

```
<island>
    <conditional SomeVariable>
    ...
<  end conditional>
    ...
<end island>
```

the island (including the island frame) will shrink and grow depending on whether the conditional inside is shown or not.

However, there are some pitfalls that you must be aware of. At compile time we include the potential height of conditionals (as if they were `true`) when calculating the heights of blocks. This is necessary to check whether they can possibly fit on pages or inside a user-specified height. Often this will not cause any problems, but if you rely on bottom baseline alignments between stacks containing conditionals, the result might not be as expected if the conditional content is not skipped.

Header and Footer Height was not Checked in MPL 2

If, for instance, the specified height of a header was less than the height actually occupied by the content, things would just be printed on top of the header, instead of giving an error message in MPL 2. This is no longer allowed. If you want to achieve this effect, you must use `movepos-` on a stacking block inside the header.

This snippet of MPL would work in MPL 2:

```
<header height=60pt>
    <stack height=80pt>
        …
    <end stack>
<end header>
```

but must be changed in MPL 3. You must either update the header height so that it can contain the contents, or you must put a `movepos-` attribute on the stack inside (if it is intended that the stack is higher than the header).

Too-Long Localized Strings are now Clipped

Widths of columns and other tags are calculated at compile time when importing an MPL layout. When dynamically localizing at runtime, this can lead to the localized text becoming too wide to fit the width that was originally calculated. Previously, this would result in this text overlapping other texts. In MPL 3 the text is now clipped to the size of the calculated width, so it will not overwrite other texts and will not be shown in its full length. There is one exception: In a canvas, the text is not clipped.

Length Orientations

The distinction between horizontal and vertical lengths has been mostly removed. For instance, this is now possible:

```
<define MyWidth MinBaselineSkip>
```

```
"Some text":width=MinBaselineSkip
```

Previously, this would not be possible because MinBaselineSkip is a vertical length, whereas the width attribute expects horizontal widths.

The only place where the orientation of lengths remains is with lengths that come from the `<grid>` tag. For instance specifying `"Some text":width=2.5grid` will use the horizontal grid length, whereas `height` attributes will use the vertical grid length (as expected).

Minor Changes

There are some additional changes that should not influence most existing prints:

- The `pagebottom` attribute of the footer tag is no longer supported. It had no effect previously anyway.
- The image tag now requires a `pos` attribute when placed in a canvas (like all other tags).
- The `HeaderSkip` length was, contrary to what the MPL manual says, not used between the page header and a block header. Now `HeaderSkip` is used.

Converting MPL 2 to MPL 3

Most layouts will work immediately just by changing the `<mpl 1>` or `<mpl 2>` version tag to `<mpl 3>`. The layout should of course always be tested.

Content Becomes too Wide

In some cases the converted layout will not work out of the box; width and height calculations have changed in MPL 3, so sometimes rulers become a few points (1pt=1/72 inch) wider in MPL 3. This might make the ruler too wide to fit the available width (as determined by the surrounding tag, for example, the `paper` tag). In this case you must reduce the ruler size by using smaller fonts, decreasing intercolumn spacing, or by providing more space for the ruler by, for example, decreasing page margins.

MPL Version 2 and MPL Version 3 Interoperability

In this section we describe some issues with using both MPL 2 and MPL 3 in one Maconomy system. These are not, as such, related to the MPL language.

Customization

You can customize an MPL 2 original layout so that it becomes an MPL 3 layout, both by shadowing the original MPL 2 layout or by importing it as a new layout.

However, after an original layout has been upgraded to MPL 3, customizing using MPL 2 is not recommended. You should upgrade existing customized MPL 2 layouts.

Print Layout Selection

When you specify Print Layout Selection rules (see the Maconomy Reference Guide), it is important that MPL 2 (or 1) and MPL 3 layouts do not participate in the same set of rules. If an MPL 3 layout is selected when MPL 2 is executing, or vice-versa, an error will occur. It is therefore quite important to verify that this cannot happen when you set up the layout selection rules. Also, updating a single layout that participates in layout selection rules requires updating all layouts that participate in the same rules.

M-Script printGetPDF Function

The M-Script Maconomy API function `printGetPDF` cannot mix MPL 2 (or 1) and MPL 3 print handles. See the"M-Script Maconomy API Reference" for more information.

Font Path Setup

The method for making additional fonts available in MPL 3 has changed slightly from MPL 2. See "Font Administration in Maconomy" in the Maconomy System Administrator's Guide.

MPL Version 4

MPL 4 is a new major version of MPL, which is a direct successor to and replacement for MPL 3. It introduces some new features, most notably custom database queries and expressions, both defined directly in an MPL layout. Because the concept of an expression is just a generalization of variable/field reference, the `<var>` and `<field>` tags have been replaced by a more general `<eval>` tag, which is used in MPL 4 for evaluating expressions. Note that this change breaks the backward compatibility of MPL 4 with respect to version 3. However, the TPUs as of M16sp0 (Maconomy 2.1) include a tool for automatic migration of MPL 3 layouts to version 4, so this should not be a problem in practice.

You can find the conversion tool in a TPU in ..\JavaMPL\MPL3To4MigrationTool.jar. Run it with the –help option to see the example usage.

This section describes in detail these new features in MPL 4, as well as possible backward compatibility issues with respect to MPL 3.

New Features

This section provides a closer look at expressions, standard functions, and custom database queries using MQL.

Expressions

In MPL 3, there were several ways of printing data on a layout. You could reference:

- A variable in the print environment, by means of the `<var>` tag.
- A cursor field in the print environment, by means of the `<field>` tag.
- A text literal.

In most programming languages these are just examples of atomic expressions, which can be combined with other expressions by means of arithmetic/relational operators, functions, and so on.

This uniform treatment of expressions is now enabled by embedding into MPL 4 the *Expression Language*, which is also used in other Maconomy specification languages like MDML or MWSL. For a detailed reference of the *Expression Language*, see “Functions and Expressions” in the Deltek Maconomy 2.1. MDML Language Reference Guide.

Expressions in MPL are always delimited by curly braces, that is, an opening bracket “{” and a closing bracket “}”. Here are a few examples of valid expressions:

- return a currency symbol for known currency names, otherwise use the currency name:

```
{ if currencyName = "USD" then "$"
  else if currencyName = "EUR" then "€"
  else if currencyName = "GBP" then "£"
  else currencyName }
```

  Note that the `if-then-else` construct is an expression, which means that it always returns a value.

- calculate an average of two variables, x and y, and turn them into a string that represents a percentage value:

```
{(x + y) / 2.0 * 100 + "%"}
```

- using standard function (evaluates to `"Monday"`):

```
{stringWeekday(date(2013,9,30))}
```

Every expression has a type, which can be determined while compiling an MPL 4 layout. It can be one of the following primitive types: *BOOLEAN*, *INTEGER*, *REAL*, *AMOUNT*, *DATA*, *TIME*, *STRING*, or an instance of a pop-up type, for example, *GenderType*.

You can use expressions as attribute values in the following tags:

- **conditional** — where the `expression` attribute of type *EXPRESSION* denotes a condition guarding the conditional tag, for example:

```
<conditional expression={overtime > 10.0}>, or just
<conditional {overtime > 10.0}>
```

- **image** — where the `expression` attribute of type *EXPRESSION* denotes a path to the image to be printed, for example:

```
<image expression={"MyLogos\Logo_" + VenderNoVar + ".png"}>
```

or just

```
<image {"MyLogos\Logo_" + VenderNoVar + ".png"}>
```

In addition to these tags, MPL 4 introduces a number of tags that enable you to perform custom calculations and bind their results to variables/values/ parameters. These include:

- **val** — where the `value` attribute of type *EXPRESSION* denotes a value to be bound to this value (that is, value or constant), for example:

```
<val name=workingDayTime value={8}>, or just
<val workingDayTime {8}>
```

- **var** — where the `value` attribute of type *EXPRESSION* denotes an initial value to be bound to this var (variable), for example:

```
<val name=sum value={0}>, or just
<val sum {8}>
```

- **assign** — where the `value` attribute of type *EXPRESSION* denotes a new value to be assigned to the given var (that is, variable), for example:

```
<assign var=sum value={x + 42}>, or just
<assign sum {x + 42}>
```

- **eval** — where the `expression` attribute of type *EXPRESSION* denotes an expression to be evaluated and printed out, for example:

```
<eval expression ={upperCase(s)}>, or
<eval {upperCase(s)}>
```

or in the short and in most cases preferred form:

```
^{upperCase(s)}
```

- **parameter** — where the `expression` attribute of type *EXPRESSION* represents an actual parameter value to which a query parameter in question will be bound when instantiating the query to a cursor, for example:

```
<cursor name=DraftInvoiceEntry query=DraftInvoiceEntryQuery>
  <parameter JobNumberPar {InvoiceEditingHeader.JobNumber}>
<end cursor>
```

For a more in-depth discussion of these tags and how the `expression` attribute is used in them, see the respective sections describing the tag in question.

Literal Values for Different Types

When declaring a `var` or a `val` or just using literals as values in expressions, it is useful to know how the different literals look for values of different types:

Type	Comma separated example values
INTEGER	`457, 77, -123`
REAL	`41.789, 99.4`
AMOUNT	`AMOUNT(99.74), AMOUNT(6.45)`
BOOLEAN	`true, false`
DATE	`DATE(2013, 12, 25), DATE(1987, 2, 15)`
TIME	`TIME(12, 23, 58), TIME(23, 15, 33)`
STRING	`"Text", "example string"`
POPUP	`GenderType'Male, CountryType'France`

Standard Functions

Along with the Expression Language, MPL 4 enables some of the Maconomy standard functions that are also available in other Maconomy Layout languages like MDML. This section lists all of the functions that are enabled in MPL 4, with an example use for each of them. For a more detailed reference on these functions, see the "MDML and Expression Language Standard Functions" document.

Let us first define some values to be used in the examples.

```
<val d  {date(2013, 10, 11)} >
<val d2 {date(2012, 12, 20)} >
<val t  {time(12, 0, 45)} >
<val t2 {time(10, 1, 18)} >
<val s  {"hello world!"}
```

Generally applicable functions			
Name	**Example**	**Result Value**	**Result type**
isNullOrEmpty	^{ isNullOrEmpty (t)}	No	Boolean

Time Functions			
Name	**Example**	**Result Value**	**Result type**
hour	^{hour(t)}	12	INTEGER

Time Functions			
minute	^{minute(t)}	0	INTEGER
second	^{second(t)}	45	INTEGER
addHours	^{addHours(t, 5)}	5:00:45 PM	TIME
addMinutes	^{addMinutes(t, 10)}	12:10:45 PM	TIME
addSeconds	^{addSeconds(t, 1)}	12:00:46 PM	TIME
secondsBetween	^{secondsBetween(t2, t)}	7167	INTEGER
minutesBetween	^{minutesBetween(t2, t)}	119	INTEGER
hoursBetween	^{hoursBetween(t2, t)}	1	INTEGER

Date Functions			
Name	**Example**	**Result Value**	**Result type**
year	^{year(d)}	2013	INTEGER
month	^{month(d)}	10	INTEGER
day	^{day(d)}	11	INTEGER
intWeekday	^{intWeekday(d)}	5	INTEGER
stringWeekday	^{stringWeekday(d)}	friday	STRING
addDays	^{addDays(d, 7)}	2013-10-18	DATE
addMonths	^{addMonths(d, 20)}	2015-6-11	DATE
addYears	^{addYears(d, 10)}	2023-10-11	DATE
addPeriod	^{addPeriod(d, 1, 12, 12)}	2015-10-23	DATE
daysBetween	^{daysBetween(d2, d)}	295	INTEGER
monthsBetween	^{monthsBetween(d2, d)}	9	INTEGER
yearsBetween	^{yearsBetween(d2, d)}	0	INTEGER

String Functions			
Name	**Example**	**Result Value**	**Result type**
length	^{length(s)}	12	INTEGER
startsWith	^{startsWith(s, "hello")}	Yes	BOOLEAN
endsWith	^{endsWith(s, "ape")}	No	BOOLEAN
firstIndexOf	^{firstIndexOf(s, "ll")}	2	INTEGER
indexOf	^{indexOf(s, "ll", 1)}	2	INTEGER
lastIndexOf	^{lastIndexOf(s, "ll")}	2	INTEGER
contains	^{contains(s, "world")}	Yes	INTEGER
substring	^{substring(s, 4)}	o world!	STRING
trim	^{trim(s)}	hello world!	STRING
upperCase	^{upperCase(s)}	HELLO WORLD!	STRING
lowerCase	^{lowerCase(s)}	hello world!	STRING
replaceFirst	^{replaceFirst(s,"ll","dd")}	heddo world!	STRING
replaceAll	^{replaceAll(s, "l", "X")}	heXXo worXd!	STRING
replaceFirstRegEx	^{replaceFirstRegEx(s,"\\w","X")}	Xello world!	STRING
replaceAllRegEx	^{replaceAllRegEx(s,"\\w","X")}	XXXXX XXXXX!	STRING
matchRegEx	^{matchRegEx(s,"\\w+")}	No	BOOLEAN
format	^{format(123456.78, "##0.#####E0")}	123,45678E3	STRING
charAt	^{charAt("MPL4", 1)}	P	STRING

Conversion Functions			
Name	**Example**	**Result Value**	**Result type**
toInteger	toInteger(8.2)	8	INTEGER
toAmount	toAmount(1)	1.00	AMOUNT
toReal	toReal(amount(12.34))	12.34	REAL

In Maconomy, mathematical functions are polymorphic; that is, their return types depend on the types of the arguments that the function takes. As mentioned earlier, every expression in MPL 4 must have a well-defined type; therefore, in case of mathematical functions you must wrap them in a conversion function that indicates to the compiler the expected return type.

Mathematical Functions			
Name	**Example**	**Result Value**	**Result type**
max	^{toReal(max(8.2, 8.3))}	8.3	REAL
min	^{toInteger(min(-1, -2))}	-2	INTEGER
abs	^{toReal(abs(-amount(6.6)))}	6.6	REAL
sig	^{toReal(sign(-7))}	-1.0	INTEGER
floor	^{toAmount(floor(-7.4))}	-8.00	AMOUNT
ceiling	^{toAmount(ceiling(123.4))}	124.00	AMOUNT

Database Queries

MPL 4 allows for defining custom database queries against the standard Maconomy universes using MQL. For a detailed description, see “Database Queries.”

Backward Compatibility Issues

MPL 4 is not entirely backward-compatible with its immediate predecessor, MPL 3. This section describes the small incompatibilities between MPL 3 and 4. Note that the TPUs as of M16 SP0 (Maconomy 2.1) include a tool for the automatic migration of MPL 3 layouts to version 4, so this should not be a problem in practice.

You can find the conversion tool in a TPU in ..\JavaMPL\MPL3To4MigrationTool.jar. Run it with the –help option to see the example usage.

Field and Variable Reference Tags Desupported in MPL 4

As mentioned, in MPL 4 there existed separate tags that represent certain types of values to be printed, that is, field and variable references as well as static text. For completeness reasons, you could use these tags in their full forms, that is, `<field>` (see “Fields”), `<var>` (see Variables”), and `<text>` (see “Texts”). These long forms were, however, discouraged as overly verbose, and

short forms were recommended to be used instead (that is, `varName`, `.fieldName`, and `"text literal`"). In fact, in the entire standard application there was not a single use of these tags in their full form.

In the context of expressions, all of these tags were just particular examples of evaluating an expression and printing out its result. For this reason, `<field>` and `<var>` tags were desupported, and their short forms are now treated as any other instance of the short form for the `<eval>` tag, used for evaluating and printing out expressions. The `<text>` and `<text2>` tags, however, still exist in MPL 4 to account for the static text localization that they support. After text localization is implemented in the Expression Language and enabled in MPL 4 expressions, these tags will be desupported as well, and their short forms will be merged with the `<eval>` tag.

Since the full versions of `<field>` and `<var>` tags are very unlikely to occur in any layout, the only problem that you might encounter in practice is when you use their names in the `<default>` tag, for example:

```
<default tag=field attribute=justification value=right>
<default tag=var attribute=fontsize value=10>
```

In MPL 4, you would use the `<eval>` tag instead to set a default for field/variable references as well as other expressions to be printed out:

```
<default tag=eval attribute=justification value=right>
<default tag=eval attribute=fontsize value=10>
```

Note that the arguably error-prone distinction that allowed for setting different defaults for field and variable references has now been droppe; the default setting applies uniformly to all printable expressions apart from the old-style text literals that are still treated differently.

<var> Tag Means Variable Declaration in MPL 4

Because the old use of the `<var>` tag is desupported in MPL 4 and hence the name *var* became available, it has been used to denote a different concept in MPL 4, namely a variable definition. See “Variable Definitions” for a detailed description of the variable definition tag.

Tags Cannot Contain Spaces before the Tag Name

In MPL 3 it was legal to have a number of whitespaces in between the opening angle bracket “<” and the following tag name. For instance, it was legal to write `<  skip>`. MPL 4 is stricter in this respect, and white spaces are not allowed. Therefore, you must write, for example, `<skip>`.

Errors and Warnings

This section contains a list of all of the error messages and warnings that the MPL compiler can display. Error messages and warnings are saved to the file `PrintLayoutErrors.txt` in the Maconomy client folder when the layout is imported.

Some messages can be either an error or a warning. If the MPL layout is used in connection with running a Universe report, the interpretation of the layout is less strict, and a number of messages that in standard MPL are treated as errors are instead treated as warnings. This means that the layout in question can be compiled and run, and that output will be displayed when the layout is used for a Universe report. When the same layout is used in standard MPL, however, the layout will not be compiled, and no output will be printed.

During the import of standard layouts (either during installation or when running `MaconomyServer -UP`), all layouts with warnings will be compiled and installed.

An explanation of the individual error messages is given below each message, including any problem-solving suggestions. If a message can appear as a warning, it is noted also. Furthermore, a section of the messages below (under the heading “Warnings”) can only appear as warnings.

Basic Errors

- (#100) Only version 1 of MPL is supported.

  A version number different from 1 has been specified in the `mpl` tag. This message only appears on servers that do not support MPL 2.

- (#102) Syntax Error.

  The MPL definition is not coherent with the MPL syntax. The error messages only tell you that the error has been detected; the error itself can occur on a previous line.

- (#103) File '*ss*' does not exist.

  The referenced file does not exist.

- (#104) Wrong original print name '*ss*'.

  The name specified in the `print` attribute in the `layout` tag does not match the name of the layout that you are to which you are trying to create/import the MPL layout.

- (#105) Unknown original layout name '*ss*'.

  The name specified in the `originallayout` attribute in the `layout` tag does not exist or is not an original layout name.

- (#108) Wrong layout name '*ss*'.

  The name specified in the `title` attribute in the `layout` tag does not match the name of the layout that you are trying to import the MPL layout to.

- (#109) Layout name length cannot exceed 51 characters.

  The name specified in the `title` attribute in the `layout` tag is longer than 51 characters.

- (#110) Original layout '*ss*' is a Layout Designer layout.

  The layout that you are trying to use as the original layout is not one of the layouts that are created in the system, but have been designed using the Maconomy Layout Designer.

- (#111) Access to original layout '*ss*' is denied.

The `originallayout` attribute in the `layout` tag exists but the user trying to compile the layout does not have access to the layout.

- (#112) Layout shadows an original layout. In tag 'layout' the attribute 'original- layout' must refer to this layout.

  An MPL layout has been imported into an original layout. In this case the `originallayout` attribute must have the same value as the `title` attribute.

- (#113) Layout name cannot be empty.

  The value of the layout attribute `name` cannot be an empty string. In MPL 3, this error message has changed to "Layout title cannot be empty."

- (#114) Tag ’page’ is missing.

  The `page` tag has not been specified.

- (#115) Tag 'ss' is not allowed in standard prints.

  A tag that is only allowed in MPL for Universe Reporting has been specified for a standard print.

- (#116) MPL version dd and above is not supported by this compiler.

  The version of MPL specified in the layout is newer than the version that the MPL compiler supports.

  In MPL 4, the message associated with this error has slightly changed to:

  “Only MPL version 4 is supported by this compiler. Migrate all print layouts starting with <mpl 3> using the *MPL3to4MigrationTool* which is found in the TPU. Ask your system administrator for help.”

- (#117) Structure layout 'ss' was not found in the database.

  When you install Maconomy, every print layout exports its Structure layout (used for structure check comparison. See “Print Structure”) to the Maconomy database. If the structure layout for a particular layout was not found in the database, you should contact your system administrator because it is most likely a system set-up error.

- (#118) Invalid syntax of expression 'ss1'. Details: 'ss2'

  This message denotes a syntax error in the given expression. The detailed error message from the *Expression Language* compiler is given.

Lexical Errors

- (#200) Name is too long. Truncated to *dd*characters.

  Names longer than 100 characters are not allowed in MPL. A name can be a cursor, a database name, a variable name, an attribute name, or an attribute value of the type ID.

- (#201) Text is too long. Truncated to *dd*characters.

  Attribute values of the type *STRING* longer than 255 characters are not allowed in MPL.

- (#202) Incomplete text.

  A string has not been finished; that is, the string starts with “ or ‘, but has not been finished by “ or ‘. Maconomy will try to read on, but naturally cannot guess where the string should have been finished. This may cause confusing error messages later in the layout. Simply insert “ or ‘, and try to import the layout again.

- (#203) Error in integer '*ss*'.

ss is an integer that is too big. Integers cannot exceed 21474836467.

- (#204) A real number must have between 1 and 3 digits after the decimal point.

  A real number of more than three decimals has been specified. Note that this is a syntax error (and is displayed as one) if no decimals are specified (as in '32').

- (#205) Error in real '*ss*'."

  ss is a real number that is too high or too small (the number is VERY high!).

Structure

- (#300) The script structure is invalid.

  The script structure of the MPL printout is not identical with the script structure of the original layout. See also ""Script Structure" in "Print Structure."

- The error message in MPL 4 has been changed to:

  The script structure of the layout is not identical to the script structure of the original layout. Expected script structure 'ss.' Actual script structure: 'ss2.'

- (#301) The stackless structure is invalid.

  The stackless structure of the MPL printout is not a subtree of the stackless structure of the original layout. See "Stackless Structures" in "Print Structure."

- The error message in MPL 4 has been changed to:

  'ss' violates the cursor structure of the original layout , as the corresponding repeating/paper with the same values of attributes does not exist in the original layout. This cursor block is valid only embedded in the following sequence of repeating/paper blocks 'ss'2.

Definitions

- (#400) Unknown paper name '*ss*' or orientation '*ss*' given.

  Either a paper name (the `name` attribute) has been specified in the `page` tag, which is not defined in the Maconomy client window Paper Formats, or the `orientation` attribute has been set to a different value than `landscape` or `portrait`. MPL distinguishes between capitals and regular characters in the paper name.

- (#401) Unknown paper name '*ss*' given.

  A paper name (the `name` attribute) has been specified in the `page` tag, which is defined in the Maconomy client window Paper Formats. MPL distinguishes between capitals and regular characters in the paper name.

- (#402) Grid used but not defined.

  The length unit `grid` has been used, but a `grid` tag has not been specified in the heading as a definition of grid lengths.

- (#403) Grid defined in terms of grid.

  You cannot use the length constant `grid` to define a grid.

- (#405) Unknown parent '*ss*'.

  The `parent` attribute in the `subruler` tag refers to an unknown ruler. This can be due to one of following reasons:

- Misspelling of the parent ruler
- The scope defining the ruler is left
- The subruler has been defined in an indented tag, for example, an `island` with a fixed inner margin or a tag with a fixed `indent`. It is not necessarily the closest surrounding tag that is indented.

- (#406) The specification of the subruler is not ordered.

  The specification of the ruler in a `subruler` tag is invalid. The reason is that the referrals to the columns of the parent ruler are not increasing from left to right (see also “Subrulers” in “Arrays“).

- (#407) Subruler not allowed in 'array'.

  The `ruler` attribute defined in an `array` has been assigned a valid subruler value. This is not allowed. Subrulers must be named using the `subruler` tag.

- (#408) Unknown ruler identifier '*ss*'.

  An `array` tag refers to an unknown value. This can be due to one of the following reasons:

  - Misspelling of the ruler
  - The execution of the scope defining the ruler is left
  - The array has been defined in an indented tag, for example, an `island` with a fixed inner margin or a tag with a fixed `indent`. It is not necessarily the closest surrounding tag that is indented.

- (#409) Invalid format for ruler value.

  The ruler specified using the `value` attribute in the `ruler` tag is not correctly formatted. In other words, one or more columns have been specified as intervals (as in subruler values).

- (#410) Invalid format for subruler value.

  The `value` attribute in a `subruler` tag has been set to a value that is not recognized as a subruler. The reason for this is that there is a missing range in a column (*INTEGER* or *INTEGER*:*INTEGER*) or that `stretch` or `width` has been specified in one of the columns.

- (#411) Cannot set default values for tag '*ss*'.

  The value specified in the `tag` attribute in the `default` tag is not recognized. Either the tag does not exist or you have specified a standard value for one of the tags `grid`, `layout`, `margins`, `page`, or `paper`. You cannot specify a standard value for these tags as they should all be specified in the root of the structure. The `default` statement must be specified in a stacking tag.

- (#412) Unknown attribute '*ss1*' in tag '*ss2*'.

  The attribute name *ss1* specified in the `attribute` attribute in the `default` tag does not match the tag name *ss2* specified in the `tag` attribute. In other words, the *ss2* tag does not have an attribute with the name *ss1*. Check the attribute list or the section about the tag in question to find out which attributes are allowed in the tag. This message is a warning if it occurs in MPL for Universe Reports.

- (#413) Attribute '*ss1*' in tag '*ss2*' has type '*ss3*'. The given value had type '*ss4*'.

You have attempted to define a default value for the attribute *ss1* in the tag *ss2*. The specified value (in the `value` attribute) does not have the correct type for this attribute. The specified value has the type *ss3*, but should have had the type *ss4*.

- (#414) Attribute '*ss1*' is mandatory in tag '*ss2*'. Therefore it cannot be set.

  You have attempted to define a default value for the attribute *ss1* in the tag *ss2*. This attribute is mandatory and you are therefore not allowed to set a default value. This message is a warning if it occurs in MPL for Universe Reports.

- (#415) Negative paper width.

  The margins set using the margins tag are so large that the paper width minus the left and right margin have become less than 0.

- (#416) Negative paper height.

  The margins set using the margins tag are so large that the paper height minus the top and bottom margin have become less than 0.

- (#417) Interval specifiers cannot be zero.

  A range (or possibly just an integer) in a subruler definition contains the number 0.

- (#418) Interval specifiers must be less than or equal to the length of the parent ruler (*nn*).

  A range (or possibly just an integer) in a subruler definition refers to a column in the parent ruler that does not exist. This is true for `r3` and `r4` in the following example:

  ```
  <ruler r1 [[][][][]]>
  <subruler r2 parent=r1 [[1][2:3][4]]>
  <subruler r3 parent=r1 [[1][2][4:5]]>
  <subruler r4 parent=r2 [[1][4]]>
  ```

  The reason for this is that *r1* has four columns, but *r3* refers to column #5. Similarly, *r2* only has three columns, but *r4* refers to column #4.

Incorrect Use of Tags

- (#500) Header already defined in this '*ss*'.

  Each parenthetical tag (`paper`, `repeating`, `conditional`, or `stack`) can only be assigned one header.

- (#501) Footer already defined in this '*ss*'.

  Each parenthetical tag (`paper`, `repeating`, `conditional`, or `stack`) can only be assigned one footer.

- (#502) The tag '*ss1*' is not allowed in '*ss2*'.

  An invalid *ss1* tag has been specified in the *ss2* parenthetical tag. This error message may be the result of one of the following situations:

  - A header or footer has been specified in another tag than `paper`, `conditional`, `repeating`, or `stack`.
  - The `span` tag has been specified in another `span` tag.
  - The `newpage` tag has been specified in a row (possible deep within) - this is only allowed if newpage has been specified as the only tag in the row.

- (#503) The tag '*ss*' is not allowed in canvases.

A tag of the type `skip`, `newpage`, `repeating`, or `conditional` has been specified in a canvas. This is not allowed.

- (#504) The tag '*ss*' is not allowed in islands.

  An illegal tag (for example, `repeating`) has been specified in an island.

  You cannot use repetitions, tables, pie charts, and so on, in islands, because these constructions make it impossible for the MPL compiler to determine the size of the island.

- (#505) The tag '*ss*' is not allowed in stacking environments.

  The tags `span` and `vline` cannot be used in stacking tags.

- (#506) The tag '*ss*' is not allowed in headers or footers.

  You cannot use the following tags in headers and footers:

  - header
  - footer
  - repeating
  - conditional
  - newpage, or
  - stackwith scripts, headers, or footers.

- (#507) The tag '*ss*' is not allowed in frontpage.

  The tags `border`, `newpage`, `field`, `repeating`, and `conditional` cannot be specified in a frontpage.

- (#508) The tag '*ss*' can only be used in *ss*.

  Lines can only be used in `canvas`, and `border` can only be used on `paper`.

- (#509) *nn1*elements in row: *nn2*was expected.

  The compiler has encountered a row with *nn1* elements where it expected *nn2* elements. The reason for this is either that the array has a ruler with *nn2* columns or that all previous rows have had *nn2* columns.

- (#510) The tag '*ss*' in arrays should contain rows.

  This error is displayed if the MPL compiler expects that a header or footer contains rows (and not elements). The reason for this might be that the header or footer is specified in a stack, repetition or condition, which should contain rows.

- (#511) Tag 'stack' in *ss*cannot contain scripts, headers or footers.

  If the `script` attribute has been specified in a stack or a header or footer has been assigned to the stack, the stack cannot occur in rows, canvases, or frontpages.

- (#512) At most one conditional can appear in a row.

  A maximum of one condition can be specified in a row. Note that all conditions in the row are counted, regardless of whether or not they are embedded.

- (#513) No repeating tags can appear in a row.

  Repetitions cannot be specified in rows.

- (#514) Rulers are not allowed in '*ss*' containing rows.

  You can only define rulers (using the ruler tag) in stacking tags that contain elements- not in stacking tags that contain rows.

- (#515) Rulers are not allowed in canvases.

  You cannot define rulers (using the ruler tag) in canvases.

- (#516) Attribute '*%s*' cannot be set in tag 'hline' appearing in stacking environments.

  `Hline` with spanning is not allowed in stacking environments. The *%s* attribute may be `columns`, `left`, and `right`.

- (#517) The tag '*ss*' is already defined in '*ss*'.

  A tag which can only be defined once in a context has been defined twice (for example, the `fields` tag has been specified twice in a table).

- (#518) Tag 'newpage' can only have orientation in paper.

  The `orientation` attribute of the `newpage` tag can only be used in the `paper` tag.

Fields, Variables, Cursors, and Expressions

- (#600) Variable '*ss*' is unknown.

  The specified variable *ss* is not recognized in this printout.

- (#601) Cursor '*ss*' is unknown.

  The specified cursor *ss* is not recognized in this printout.

- (#602) Field '*ss*' is unknown.

  The specified field *ss* has not been recognized in the position specified. A field is defined in a cursor. It can therefore only occur in the repetition or paper to which the cursor is assigned.

- (#603) Field '*ss1*' is unknown in cursor '*ss2*'.

  A field with an explicit cursor name has been specified (that is, either *ss2.ss1* or `<field title=`*ss1* `cursor=`*ss2*`>)`. The field *ss1* is not recognized in cursor *ss2*.

- (#604) Variable '*ss*' cannot be used in conditionals as its type is wrong.

  A variable which is not of the type *BOOLEAN*, *INTEGER* or *ENUMERATION* (or *STRING* for MPL 3) has been specified as the condition in a conditional.

- (#605) Ruler column index 'nn' out of bounds

  Too many elements were added to the ruler. Remedy: reduce the number of elements, change the definition of the ruler to include more elements, or use another ruler.

- (#606) Duplicate var/val definition: 'ss'.

  It is not allowed to define a new var/val with the same name as another val/var in the same scope.

- (#607) Error(s) found in expression {ss}. 'Detailed error message'.

  An error occurred while compiling the given expression. A detailed description of the error issued by the *Expression Language* compiler is given.

- (#608) Assignment to an unknown var 'ss'.

  The variable *ss* used in the <assign> tag is not defined in the current scope.

- (#609) Incompatible types. Cannot assign 'ss1' to 'ss2'.

MPL is a strongly typed language, which means that every value in MPL has one of the following primitive types: *BOOLEAN*, *INTEGER*, *REAL*, *AMOUNT*, *DATA*, *TIME*, *STRING* or an instance of a popup type, for example, *GenderType*.

Generally speaking, you can only assign to a variable a new value of the same type as the variable's type. The only exception to this rule is when assigning *AMOUNT* and *REAL*values to a variable of type *INTEGER*.

When trying to assign a value of incompatible type to a variable, it usually denotes a semantic error in your code. There are, however, some corner cases when one has to convert between various numeric types. To this end, MPL 4 provides the following conversion functions: `toReal`, `toInteger`, and `toAmount`, taking a value of numeric type as a parameter and converting it to a value of the type specified in the name of the function. Note that these conversions might lose precision, for example, when converting a REAL to an INTEGER.

- (#610) Expression 'ss' cannot be used in conditionals as its type is wrong. Expected *BOOLEAN*, *INTEGER* or *STRING*. Got: 'ss2'.

  A condition defined for an MPL conditional must evaluate to a *BOOLEAN*, *INTEGER* or *STRING* value.

- (#611) Duplicate query definition 'ss'.

  A query with the same name 'ss' is already defined in the given scope. Consider renaming your query.

- (#612) Syntax error: invalid query body.

  The query tag is expected to contain a valid *MQL*query body.

- (#613) Cursor 'ss' requires a parameter 'ss2'.

  The MQL query that the cursor 'ss' refers to declares a parameter 'ss2'. Cursor 'ss' must supply an actual value of this parameter using the `<parameter>` tag.

- (#614) Duplicate cursor definition 'ss'.

  A cursor with the same name 'ss' is already defined in the given scope. Consider renaming your cursor.

- (#615) "Cursor 'ss' refers to an unknown query 'ss1."

  The query 'ss1' is not defined in the scope of cursor 'ss.'

- (#616) "Query 'ss' does not expect a parameter 'ss2.'

  The query that the current cursor refers to does not take a parameter named 'ss2.'

- (#617) "Invalid parameter type. Expected type: 'ss'. Got: 'ss2'.

  The parameter in question is expected to be of type 'ss', whereas the type of the given actual parameter value is 'ss2.'

- (#618) Duplicate parameter binding definition: 'ss.'

  The parameter 'ss' has been already supplied with a value.

- (#619) Error validating MQL query. Details: 'Detailed error message'.

  An error occurred while validating the MQL query. A detailed error message from the MQL compiler is given.

- (#620) Cursor name 'query' is reserved for a default top level MQL cursor name. Please use another name.

As explained above, the cursor name “query” has a special meaning in MQL and hence is reserved.

Warnings

- (#650) Variable '*ss*' is unknown.

  The specified variable *ss* is not recognized in this printout.

- (#651) Cursor '*ss*' is unknown.

  The specified cursor *ss* is not recognized in this printout.

- (#652) Field '*ss*' is unknown.

  The specified field *ss* has not been recognized in the position specified. A field is defined in a cursor. It can therefore only occur in the repetition or paper to which the cursor is assigned.

- (#653) Field '*ss1*' is unknown in cursor '*ss2*'.

  A field with an explicit cursor name has been specified (that is, either *ss2.ss1* or `<field title=`*ss1* `cursor=`*ss2*`>)`. The field *ss1* is not recognized in cursor *ss2*.

- (#776) Attribute 'zerosuppression' can only be used with fields of type 'Amount', 'Integer' and 'Real'.

  The attribute `zerosuppression` has been specified for a field that is not a value field (that is, has the type Amount).

Attributes

- (#700) In '*ss1*': no nameless attributes of type '*ss2*'.

  You have attempted to specify a nameless attribute of the type *ss2* in the tag *ss1*. The error is displayed because the there are no nameless attributes of the type in question.

- (#701) In '*ss1*': no nameless shortattributes of type '*ss2*'.

  The tag *ss1* has been specified in short form combined with an attempt to specify a nameless attribute of the type *ss2*. The error is displayed because there are no nameless short attributes of the type in question.

- (#702) In '*ss1*': attribute '*ss2*' is mandatory.

  The mandatory attribute in the tag *ss1* has been left out.

- (#703) In '*ss1*': unknown attribute '*ss2*'.

  An unknown attribute, *ss2*, has been specified in the tag *ss1*.

- (#704) Attribute '*ss*' is defined more than once.

  The same attribute *ss* has been used more than once. Note that it can be specified as a nameless attribute or as a short form. In the following three examples the `title` attribute to the `text` tag has been specified more than once:

  ```
  <text "Hello" "Hello"> "Hello":title="Hello"
  ```

  ```
  <text "Hello" title="hello">
  ```

- (#705) Attribute '*ss*' in '*ss*' cannot be given with unit '*ss*'.

  You have attempted to define a grid using lengths with the unit grid. This is not allowed as `grid` has not been defined at this point.

- (#706) In attribute '*ss1*': type 'Real' expected. Got '*ss2*'.

  The attribute *ss1* has the type *REAL*, but a value of the type *ss2* has been specified.

- (#707) In attribute '*ss1*': type 'Boolean' expected. Got '*ss2*'.

  The attribute *ss1* has the type *BOOLEAN*, but a value of the type *ss2* has been specified.

- (#708) In attribute '*ss1*': type 'Integer' expected. Got '*ss2*'.

  The attribute *ss1* has the type *INTEGER*, but a value of the type *ss2* has been specified.

- (#709) In attribute '*ss1*': type 'String' expected. Got '*ss2*'.

  The attribute *ss1* has the type *STRING*, but a value of the type *ss2* has been specified.

- (#710) In attribute '*ss1*': type 'ID' expected. Got '*ss2*'.

  The attribute *ss1* has the type *ID*, but a value of the type *ss2* has been specified.

- (#711) In attribute '*ss1*': type 'Length' or 'ID' expected. Got '*ss2*'.

  The attribute *ss1* has the type *LENGTH* (meaning that you should specify a length or a name of a length constant), but a value of the type *ss2* has been specified.

- (#712) In attribute '*ss1*': type 'Interval' expected (format 'Integer' or 'Integer:Integer'). Got '*ss2*'.

  This error message is displayed if you have specified the `interval` attribute (in the named version) with an incorrect value of the type *ss2* in a column or a subcolumn.

- (#713) In attribute '*ss1*': type 'Position' expected. Got '*ss2*'.

  The attribute *ss1* has the type *POS*, but a value of the type *ss2* has been specified.

- (#714) In attribute '*ss*': type 'List' expected. Got '*ss*'.

  The `groupby` attribute to the `repeating` tag has been incorrectly specified.

- (#715) In attribute '*ss*': type 'Ruler' or 'ID' expected. Got '*ss*'.

  A value of an incorrect type has been specified for the attribute *ss1* (either the `ruler` attribute to `array` or the `value` attribute to `ruler` or `subruler`). The attribute must be a ruler or an ID that refers to a defined ruler.

- (#716) Unknown justification '*ss*'.

  An invalid value for the `justification` attribute, *ss*, has been specified. The value must be either `left`, `center`, or `right`. Note that `justification` is nameless in some tags, meaning that if you specify a nameless value of the type *ID*, this may be interpreted as a `justification` value.

- (#717) Unknown length identifier '*ss*'.

  An unknown value for an attribute of the type *LENGTH*, *ss*, has been specified. The reason for this is that the name has been misspelled (the MPL compiler is case sensitive) or that the constant has not been defined using `define`.

- (#718) Conflicting orientations: identifier '*ss*' specifies a horizontal length, but is used in a vertical context.

  The length constant *ss* is horizontal but is used in a context where only vertical lengths can occur. The length constant *ss* is horizontal either because that it is so defined, or because it has been used previously in a horizontal context. Furthermore, length constants, as defined using `textwidth`, are horizontal.

- (#719) Conflicting orientations: identifier '*ss*' specifies a vertical length, but is used in a horizontal context.

  The length constant *ss* is vertical but is used in a context where only horizontal lengths can occur. The length constant *ss* is vertical either because that is so defined, or because it has been used previously in a vertical context. Furthermore, length constants, as defined using `textheight`, are vertical.

- (#720) The first coordinate of the pair is a vertical unit.

  A pair of coordinates (*x,y*) has been specified in which the first coordinate *x* is not horizontal as expected. The reason for this can be that a length with a `textheight` unit has been specified, or that a vertical length constant has been used (see the description of error message #719 for further information about vertical constants).

- (#721) The second coordinate of the pair is a horizontal unit.

  A pair of coordinates (*x,y*) has been specified in which the second coordinate *y* is not vertical as expected. The reason for this can be that a length with a `textwidth` unit has been specified, or that a horizontal length constant has been used (see the description of error message #718 for further information about horizontal constants).

- (#722) Attribute 'orientation' is mandatory for grid length definitions.

  If the length unit for the value specified in a `define` tag is `grid`, an orientation has to be specified. The rule only applies to the `redefine` tag if the redefined length constant has not already been assigned an orientation.

- (#723) Unknown orientation '*ss*'.

  This error message is displayed if a different value than `horizontal` or `vertical` has been specified in the `orientation` attribute to a define or redefine tag.

- (#726) Attribute 'zerosuppression' can only be used with fields of type 'Amount', 'Integer' and 'Real'.

  The attribute `zerosuppression` has been specified for a field which is not a value field (that is, has the type Amount).

- (#728) Attribute 'columns' in tag 'span' must be greater than 0.

  The attribute `columns` in the `span` tag has been set to 0.

- (#729) Attribute 'columns' in tag 'hline' must be greater than 0.

  The attribute `columns` in the `hline` tag has been set to 0.

- (#730) Unknown alignment: '*ss*'

  If the `align` attribute is specified in the `row` tag, it should be assigned one of the values `base`, `top`, `center`, or `bottom`.

- (#731) Unknown baseline: '*ss*'

  If the `baseline` attribute is specified in one of the tags `array`, `stack`, `conditional`, or `repeating`, it should be assigned one of the values `top` or `bottom`. If the attribute has been specified in the `island` tag, the allowed values are `title`, `top`, or `bottom`.

- (#732) Unknown script name '*ss*' (or script is empty).

  An unknown script name has been specified using the `script` attribute.

- (#733) Attribute 'indent' cannot be greater than zero in '*ss*' containing rows.

The indent attribute cannot be used in conditions, repetitions, stacks, and islands containing rows. Only a zero length is allowed (this is only relevant if you have set the standard values differently).

- (#734) Attribute 'height' can only be used in '*ss*' if it contains no repetitions

  The `height` attribute cannot be used in stacks, conditions, or repetitions if the MPL compiler cannot determine the height of the contents. This error message is displayed if the block contains repetitions, conditionals or wrapped fields and variables.

- (#735) Attribute 'stretch' cannot be false in 'island' containing rows.

  The `stretch` attribute cannot be set to `false` in islands containing rows.

- (#736) If 'indent' attribute is set, 'justification' cannot be '*ss*'.

  You can only use the `indent` attribute if the element (`field`, `variable`, `text`, or `island`) is left-adjusted. Note that islands are centered unless otherwise specified, and that certain fields and variables are right-adjusted (depending on type). In this case, it is thus necessary to set the `justification` attribute.

- (#737) Attribute 'baseline' cannot be set to 'title' unless a title is given.

  The `baseline` attribute in an `island` tag has been set to the value `title`, which means that the island's baseline is inherited from the island title. This error message is displayed if no island title has been specified.

- (#738) Horizontal margins cannot be greater than zero in 'island' containing rows.

  You cannot set the `leftmargin` and `rightmargin` attributes in the `island` tag if the island contains rows, because these rows share the ruler with the rows outside of the island. Only a zero length is allowed (this is only relevant if you have set the standard value differently).

- (#739) Length identifier '*ss*' already defined.

  You cannot use the `define` tag to define a length constant already defined. Instead you should use `redefine`. Note that the MPL compiler is case-sensitive.

- (#740) Length identifier '*ss*' not known.

  You have attempted to redefine a length constant, *ss*, using `redefine`, but the constant is unknown. Note that the MPL compiler is case sensitive.

- (#741) Exactly one of 'top' and 'bottom' attributes must be given in tag '*ss*'.

  The `border` or `goto` tag should contain one of the attributes `top` (the distance to the top edge of the page) or `bottom` (the distance to the bottom edge of the page).

- (#742) The '*ss*' attribute value cannot begin or end with blanks.

  The attributes `title, print`, and `originallayout` in the layout tag cannot begin or end with a space.

- (#743) The '*ss*' attribute value cannot contain '*cc*' characters.

  The `title` attribute in the `layout` tag cannot contain underscores, question marks or punctuation marks.

- (#744) Unknown color '*ss*'.

  The `color` attribute has been defined with an unknown color value.

- (#745) RGB values must be between 0 and 100.

  The values in an `rgb` must be between 0 and 100.

- (#746) Exactly one of '*title*', 'varname' and 'fieldname' must be defined for tag 'image'.

  The `image` tag should contain one of the attributes `varname` (variable name), `fieldname` (field name) or `title` (string containing image document reference).

- (#748) Exactly one of '*varname*' and '*fieldname*' must be defined for tag 'title'.

  The `title` tag should contain one of the attributes `varname` (variable name) or `fieldname` (field name).

- (#749) Style attributes are not allowed on images.

  The `image` tag does not allow any of the style attributes : `justification`, `fontname`, `fontsize`, `bold`, `italic`, `underline`, `color`, `rgb`.

- (#750) Exactly one of attributes 'href', 'component', 'report' and 'script' must be given.

  A link should contain exactly one of the stated attributes.

- (#751) In attribute '*ss*': type 'PARAMLIST' expected. Got '*ss*'.

  An attribute value of another type than *PARAMLIST* has been specified for the attribute.

- (#753) Justification attribute is only allowed on images if a width is defined.

  To use `justification` on images the width must be defined.

- (#754) Attributes indent, keeptogether and stop are not allowed on stacks in tables.

  You cannot use these attributes on the `stack` tag if it is defined within a table.

- (#755) Unknown target '*%s*' (only 'self', 'new' and 'rightside' allowed).

  The `target` attribute on links only supports the listed targets.

- (#756) Format attribute '*ss*' does not match the type '*ss*' of '*ss*'.

  The type of the field/variable is not the same as the format specified (for example, an `amountformat` has been specified on a field of type *DATE*). This message is a warning if it occurs in MPL for Universe Reports.

- (#757) The content of format attribute '*ss*' in tag '*ss*' is not valid.

  The format specification is invalid. This message is a warning if it occurs in MPL for Universe Reports.

- (#758) In attribute 'rgb': type 'Triple' expected. Got 'ss'.

  The attribute `rgb` has the type *TRIPLE*, but a value of type *ss* has been specified.

- (#759) In tag 'ss': attribute 'pos' and 'indent' cannot be set together.

  Both attribute `pos` and attribute `indent` have been specified together in the same tag. This occurs if you specify the `indent` attribute on. Note: in the old engine specifying `indent` on, for example, a text tag in a canvas would result in this error "(#703) In 'text': unknown attribute 'indent'.

- (#760) Exactly one of 'variable' and 'field' must be defined for tag 'conditional'.

  When using the `conditional` tag, either the (possibly nameless) `variable` attribute must be set or the (non-nameless) `field` attribute (and possibly also the `cursor` attribute) must be set. The `variable` attribute cannot be specified together with the `field` and `cursor` attributes or vice-versa.

- (#761) In tag 'ss': attributes 'height' and 'lines' cannot be set together.

When setting the `height` attribute on a `text`, `field`, or `var` tag, the `lines` attribute cannot be set at the same time (and vice-versa). Use only one of them to specify the desired height of the tag.

- (#762) In attribute 'ss': type 'Subruler' expected. Got 'ss'.

  When declaring a subruler, the value must be a *SUBRULER* type.

Sizes

- (#800) Tag '*ss*' too wide: element and indentation was *l1*, stacking environment only allows *l2*.

  An element, *ss*, was wider than the width allowed by the surrounding elements. The width of the element including its indent was *l1*, whereas the surrounding elements only allowed the total element width *l2*. This message is a warning if it occurs in MPL for Universe Reports.

- (#801) Tag '*ss*' too wide: element was *l1*, stacking environment only allows *l2*.

  An element, ss, was wider than the width allowed by the surrounding elements. The width of the element was l1, whereas the surrounding elements only allowed the total element width l2. This message is a warning if it occurs in MPL for Universe Reports.

  A common reason for this error is text that was added to a ruler that does not fit the field size allocated on the ruler. In response to this, MPL3 extends the ruler, thus making it too large for the page. The solution is to adjust the ruler to accommodate for the size of the text, or to reduce the size of the added text. Note that the line number that prefixes the error message points to the added text, not the ruler definition.

- (#802) Contents of '*ss*' too high: content was *l1*, 'height' attribute only allows *l2*.

  The contents of the tag *ss* is higher than the height allowed by *ss* in its `height` attribute. This error message may also be displayed for the `frontpage` attributes, if the contents of `frontpage` is higher than the height allowed by the paper format. It could also be because the required row height is greater than the available height. This message is a warning if it occurs in MPL for Universe Reports.

- (#803) Contents of 'array' too wide: content was *l1*, 'width' attribute only allows *l2*.

  The width attribute has been specified as *l2* for an array, but the sum of the columns is *l1*, which is too wide. This message is a warning if it occurs in MPL for Universe Reports.

- (#804) Tag '*ss*' too wide: its far right was at *l1*, canvas only allows *l2*.

  The positioning and width of the tag *ss* results in a right margin edge of *ss* extending beyond the right side of the canvas *ss* is placed in. This message is a warning if it occurs in MPL for Universe Reports.

- (#805) Tag '*ss*' too high: its bottom was at *l1*, canvas only allows *l2*.

  The positioning and height of the tag *ss* results in a bottom edge of *ss* below the bottom of the canvas *ss* is placed in. This message is a warning if it occurs in MPL for Universe Reports.

- (#806) The 'height' *l1*given in 'footer' cannot hold its content (height *l2*).

  A footer height has been specified using the `height` attribute, but the contents of the footer is higher than allowed.

- (#807) The height of 'footer' (*l1*) is greater than the height of the page (*l2*).

A footer height has been specified using the `height` attribute, which is larger than the page size of the current paper format.

- (#808) The 'height' *l1*given in 'header' cannot hold its content (height *l2*).

  A header height has been specified using the `height` attribute, but the contents of the header is higher than allowed.

- (#809) The height of 'header' (*l1*) is greater than the height of the page (*l2*).

  A header height has been specified using the `height` attribute, which is greater than the page size of the current paper format.

- (#810) The border '*ss*' is placed outside the margins of the page.

  A `border` or `goto` position has been specified using the `top` or `bottom` attribute, which has resulted in positioning of the `border`/`goto` outside the paper margin (possibly outside the paper). This message is a warning if it occurs in MPL for Universe Reports.

- (#811) Canvas element is overlapping with another element (*%s*).

  %s is a line number. This error message is given to help you identify too long localization strings, and can occur when running `MaconomyServer -ULP`. The `ULP` parameter localizes MPL layouts before they are installed on the server. This message is a warning if it occurs in MPL for Universe Reports.

- (#812) Position of 'goto' in '*ss*' was above previous element.

  When using `goto` in a header you cannot go to a position above a previous element in the header, because that would imply a page break, which cannot occur in a header.

- (#813) Position of 'goto' in footer did not make room for ensuing elements.

  When using `goto` in a footer, there has to be room for the elements following the `goto` on the same page.

- (#814) Tag 'goto' is not allowed in '*ss*' with attribute '*ss*'.

  In headers and footers, `goto` is not allowed if attribute `atstart` or `atend` is set.

- (#815) The height of a frontpage must be fixed.

  When using a `field` or `var` tag with the `wrap` attribute set to `true`, or when using a `concat` tag, the height is not fixed. The frontpage must have a fixed height, so you must specify either the `height` or the `lines` attribute on those tags when used in the frontpage.

- (#817) A stack with movepos=false cannot be larger than the page height.

  When using a stack tag with the attribute `movepos` set to false, this stack's height cannot be larger than the available page height.

- (#818) Tag 'ss' must specify height when used in a canvas.

  When using a `field` or `variable` tag with the attribute `wrap` set to `true`, or when using the `concat` tag, the `height` or `line` attribute must be specified as we cannot calculate the height of these tags. Note that if the `width` attribute is not specified for these tags, they will fill the amount of horizontal space available for the canvas.

MDL and MPL Preprocessor

This document describes the preprocessor functionality in MDL and MPL. This was introduced in TPU 53.

The preprocessor functionality is also available from M-Script. For more information see the M-Script Language Reference Manual.

Introduction

The MDL and MPL preprocessor functionality allows code sections of MDL and MPL (both in standard prints and universe reports) to be dependent on add-ons as well as system parameters and system information. This means that you can, for example, define a dialog that only shows a certain island if a certain system parameter has been marked, or a printout that only shows certain information if a certain add-on has been installed.

The main reason for using this is to leave out parts of layouts that are irrelevant in certain setups. This functionality may be used mostly by the Deltek R&D department, but can be used by any layout and MRL report developer.

Versions

The preprocessor is available in MDL and MPL as of TPU 53.

To be able to import MDL layouts using preprocessor directives, a Maconomy Windows client version 4.3.0 is necessary. The preprocessor will, however, work correctly on older clients, and MPL with preprocessor directives can be imported with older clients as well.

Application version 8.0SP11 is necessary for automatic recompilation of MPL after changes to system parameters and system information. This application version also renames all system parameters such that pseudo localization tags (`@`) are removed.

Preprocessor Options

The preprocessor is applied to MDL and MPL before the relevant compiler is invoked. This means that the preprocessor directives can occur anywhere in the MDL or MPL syntax.

Syntax

The syntax for preprocessor options is as follows:

```
#if <expression>
   ...
#endif
```

and

```
#if <expression>
   #else
#endif
```

The # directives must occur as the first token on a line.

```
<expression>
```

is one of

```
addon(<number>)
```

```
systemparameter.<systemparametername>
systeminformation.<systeminformationfield>
```

For systems that run with Danish kernel language, this last option would be:

```
systemoplysning.<systeminformationfield>.
```

Note that a number of system parameters have an “@” as the first character. If this is the case, you must enclose the entire system parameter in a pair of backslashes, as illustrated in the following examples.

```
## WRONG:
#if systemparameter.@AllowChangeofVATOnInvoiceLines
    ...
## CORRECT:
#if systemparameter.\@AllowChangeofVATOnInvoiceLines\
    ...
```

The system parameter or the system information field must be of the type Boolean. Otherwise, an error message is produced (this can be viewed in the `LayError.txt` resp. `PrintLayoutErrors.txt` file, which is placed in the Maconomy client folder). If an add-on, a system parameter, or a system information field does not exist, it is treated as `false`.

Examples

Write a Text if an Add-On is Set

```
<island "Add-on">
    #if addon(65)
        "Add-on 65 is set"
    #endif
<end island>
```

Write a Text if an Add-On is Set, Otherwise Another

```
<island "Add-on">
#if addon(65)
    "Add-on 65 is set"
#else
    "Add-on 65 is not set"
#endif
<end island>
```

Write in Italics if an Add-On is Set

The preprocessor directives can be used everywhere in the layout.

```
"Hi"
#if addon(65)
    :italic+
#endif
```

Write a Text if a System Parameter is Set

```
#if systemparameter.UseDailyTimeSheets
   "We use daily time sheets"
#endif
```

Write a Text if a System Information Field is Set

```
#if systeminformation.DifferentialVAT
   "We use differential tax"
#endif
```

Negate a Boolean Criterion

There is no syntax for negating a Boolean criterion. Instead, you write:

```
#if systemparameter.UseDailyTimeSheets
#else
   "We don't use daily time sheets"
#endif
```

Warning

You should be aware that using the preprocessor functionality increases the risk of introducing illegal MDL/MPL layouts in the system. Consider the following fragment:

```
#if systemparameter.UseDailyTimeSheets
   "Add-on 65 is set"
#else
   <island "title> .unknownfield <end stack>
#endif
```

The fourth line contains three errors: Missing `"` after `title`, reference to an unknown field, and an attempt to match `<island>` with `<end stack>`.

Nevertheless, if the system is set up to use daily time sheets, this MDL will be validated with no problems. This is because the preprocessor is invoked prior to invoking the MDL compiler, which would otherwise detect the problems.

Now suppose that the system parameter “Use Daily Time Sheets” is changed to `false`. Now the layout will suddenly be invalid, and users will not be able to open the window for which the layout is defined.

It is therefore recommended that you test all MDL/MPL thoroughly before any preprocessor directives are inserted.

MCSL

This section provides a guide to the Maconomy Coupling Service Language (MCSL).

Introduction

The MCSL is designed to enable the configuration of various configurable aspects of the Coupling Service. You can use MCSL to configure the following areas:

- Access control
- Environment holding contents of all Maconomy defined variables
- Dictionaries used for client localization
- Update sites locations

Attributes and Referred Types

Attributes of a given tag are described using the following table.

Attribute Name	Type	Usage
Attr1	Type of attr1	This attribute is used to…
Attr2	Type of attr2	Indicates…

In addition, attributes can have one of the following referred types.

Boolean	A Boolean Attribute (True or False)
Key	A string that is case-insensitive and never exposed to any end user. It is used for references (internally or from other parts in the spec/other specs).
Display	A string that is meant to be displayed to an end user (and which is therefore localized). It can never be used for any reference.
Id	A string that is case-insensitive and used to reference items in environments that are not controlled by Maconomy. It is never exposed to any end user.

Document Structure

An MCSL specification has the following basic structure:

```
<MCSL version="0.5" xmlns="http://www.deltek.com/ns/mcsl">
  <Access>
    <Role source="All Workspaces">
      <Access all="true" />
    </Role>
```

```
    <Role source="TimeSheet User">
      <Access>
        <Workspace source="TimeSheet" />
      </Access>
    </Role>
  </Access>
  <Environment>
    <Binding entity="UserInformation" namespace="user:info">
      <Restriction condition="NameOfUser = userName()" />
      <Fields>
        <Field source="AccessToAllTimeSheets" />
        <Field source="CanSeeAllExpenseSheets" />
        <Field source="EmployeeNumber" />
        <Field source="NameOfUser" />
      </Fields>
    </Binding>
    <Binding entity="UserDialogGroup"
namespace="user:groups">
      <Restriction condition="NameOfUser = userName()" />
      <Fields>
        <Field source="GroupName" />
      </Fields>
    </Binding>
  </Environment>
  <Configuration>
    <UpdateSites>
      <UpdateSite location="http://<somelocation>:1234/" />
      <UpdateSite location="http://<otherlocation>:1234/" />
    </UpdateSites>
  </Configuration>
</MCSL>
```

The document has the MCSL element as root.

- The Access element contains rules for some or all of the roles in the system that specifies which workspaces each role gives access to.
- The Environment element contains specifications of which environment contributions are present and from which database entities they are obtained.
- The Configuration element contains specifications for the dictionaries and update sites location.

Namespace and Root Tag

Namespace

The namespace of the MCSL format is as follows:

```
http://www.deltek.com/ns/mcsl
```

In this document, the MCSL namespace is the default namespace to avoid namespace prefixes on tags.

<MCSL>-Tag

The MCSL tag is the root tag of the MCSL specification.

Attribute Name	Type	Usage
Version	Key	This is the MCSL version. This property must be specified.

Access Specification

Workspace Access Model

The access model for the workspace access control such that if at least one of a user's roles gives access to a workspace, the user has access to the workspace. This means that each of the roles that a user has can give access to any number of workspaces, but not restrict the access to workspaces.

Outer <Access>-Tag

The outer Access tag is placed directly in the root MCSL-tag. It serves as a container for the set of role-access specifications in the system. The access tag contains any number of Role tags.

<Role>-Tag

The Role tag specifies the access privileges for a role in the system. The MCSL specification has, at most, one Role tag for each role in the system. The specification cannot contain multiple specifications for a role.

Each role has a compulsory inner Access tag that contains the specification of which workspaces the role may access.

Attribute Name	Type	Usage
Source	Key	This attribute specifies the role to which this Role tag refers. For the Role tag to have an effect, the role must be defined in the system. Roles are defined in Groups in the Maconomy system.
Name	Key	(Optional) This attribute refers to name of this role access specification.

Inner <Access>-Tag

The inner Access tag is placed in a role to specify the workspaces to which the role gives access. The Access tag can contain the list of these workspaces.

Attribute Name	Type	Usage
All	Boolean	(Default: false) This attribute specifies that the role gives access to all workspaces in the system, except those in the Exclude tag of this Access tag.

The following example grants users with role R access to workspaces WS1 and WS2:

```
<Role source="R">
  <Access>
    <Workspace source="WS1" />
    <Workspace source="WS2" />
```

```
  </Access>
</Role>
```

You can also use the all-attribute with the Exclude tag to specify that a role gives access to all workspaces, except for a given list of workspaces. The following example grants users with role R access to all workspaces except WS1 and WS2:

```
<Role source="R">
  <Access all="true">
    <Exclude>
      <Workspace source="WS1" />
      <Workspace source="WS2" />
    </Exclude>
  </Access>
</Role>
```

<AllRoles>-Tag

The AllRoles tag is similar to the Role tag. It is used to describe access rules that span all roles. This tag has the same content as the Role tag.

<Workspace>-Tag

The Workspace tag is used in an Access tag or an Exclude tag to reference a workspace. For the reference to have an effect, the system must contain a workspace with the specified name.

Attribute Name	Type	Usage
Source	Key	This attribute indicates the name of the referenced workspace. The value must correspond to the name attribute of the referenced workspace.
Name	Key	(Optional) This attribute refers to name of this workspace reference.

<Exclude>-Tag

The Exclude tag is an optional tag that is used in the Access tag in a role specification. The Exclude tag contains zero or more Workspace references that represent exceptions from the list of workspaces in the Access rule.

The following example grants users with the role R access to workspace WS1 (the user does not get access to WS2 through the role R):

```
<Role source="R">
  <Access>
    <Exclude>
```

```
      <Workspace source="WS2" />
    </Exclude>
    <Workspace source="WS1" />
    <Workspace source="WS2" />
  </Access>
</Role>
```

The Exclude tag is most useful in combination with the all attribute:

```
<Role source="R">
  <Access all="true">
    <Exclude>
      <Workspace source="WS1" />
      <Workspace source="WS2" />
    </Exclude>
  </Access>
</Role>
```

Scope of Exclude

Note that the Exclude tag does not prevent access to a workspace if the user has access to the workspace through another role. In the following example, a user with roles R1 and R2 has access to WS1:

```
<Role source="R1">
  <Access>
    <Workspace source="WS1" />
  </Access>
</Role>
<Role source="R2">
  <Access>
    <Exclude>
      <Workspace source="WS1" />
    </Exclude>
  </Access>
</Role>
```

Environment Specification

<Environment>-Tag

The Environment tag is placed directly in the root MCSL tag. It is designed to specify the contributions that are available in an environment that contains variable values that are used in various functions of the Maconomy system. The Environment tag contains an optional Define tag and any number of Binding tags.

<Binding>-Tag

The Binding tag corresponds to a single database query, whose results are contributed to the current calculated environment entries. To specify the Binding tag correctly, a valid entity name for query and namespace expression must be provided such that there is a correct translation between the query result and environment contribution.

Attribute Name	Type	Usage
entity	Key	This attribute refers to the database entity that contributes entries to the environment.
namespace	Expression	This attribute refers to the environment path expression under which the current contributions are placed.

The namespace is semantically equivalent to the environment path except for the separators, which are a colon (:) in the binding namespace and a dot (.) in the environment path. Expression as a namespace type is used because environment contributions are placed under a path value, which is a result of evaluating an expression.

Each output query row of the system parameters contributions is placed under a different path, which contains, as a last sub-path, the value of an **envVar('InternalName')** function call. This function returns the value of a current processed row's column value, which is set in the context environment.

Thus, it is impossible to specify the path in advance, but it is possible to evaluate it after the query row environment contribution is obtained and appended to the context for the namespace expression evaluation. To support the need to specify the namespace expression easily, a new placeholder expression type has been introduced.

The values specified in this attribute are treated as constant expressions with the special escape sequence ^{} that treats everything located in the curly braces as an expression for evaluation. For example, the expression "ab^{2+3}c" is evaluated to "ab5c."

Example of Syntax Use

```
"system:parameters:^{envVar('InternalName')}".
```

<Restriction>-Tag

The Restriction tag is used to provide the condition for filtering rows of the query that correspond to the Binding tag. The condition can be specified as an attribute of the Restriction tag or as a child Condition tag.

Attribute Name	Type	Usage
condition	Expression	(Optional) This attribute refers to the restriction condition that is used for a query.

<Condition>-Tag

The Condition tag is an optional tag that can be used instead of the condition attribute of the Restriction tag. The condition expression is placed directly within the Condition tag root element.

Binding <Fields>-Tag

The Binding Fields tag compresses a set of fields that are translated into a list of selection columns that are used in the query that corresponds to a given Binding tag.

<Field>-Tag

The Field tag corresponds to a single column in the entity. It is used in the Fields tag as a root tag, as a single column that is part of the specified query set. As an alternative, an option has been added to specify the value directly in the specification, allowing contributions that are not obtained from the database. This is done by specifying a pair of attributes; the first one is called **name** and corresponds to the contribution name, and the second one is called **valueString** or **value** depending on the type that is used.

It is only possible to specify either a **source** attribute or a pair of **name** and **valueString** or **value** attributes. Any other combination of attributes is considered an error.

Attribute Name	Type	Usage
source	Key	This attribute refers to the column name of Binding's entity.
name	Key	This attribute refers to the name of the contribution.
valueString	Expression	This attribute refers to the value for a given name of the placeholder string type.
value	Expression	This attribute refers to the value for a given name of the general expression type.

<Define>-Tag

The Define tag is an optional tag that is used to define the named groups of fields that can be referenced, reducing the number of Field tags for Bindings that share a common set. It contains any number of define Fields tags. It serves as a facility that simplifies the process of specification writing.

Define <Fields>-Tag

The Define Fields tag is placed directly in the Define tag as a root tag and used to map a unique name to a set of fields that can be referenced from the Reference Fields tag.

Attribute Name	Type	Usage
name	Key	This attribute refers to the name of this Fields group for reference.

Reference <Fields>-Tag

The Reference Fields tag can only be used as a child of the Fields tag to insert all fields from the corresponding Define Fields tag. It can be a child of either the Define Fields tag or Finding Fields tag.

Attribute Name	Type	Usage
ref	Key	This attribute refers to the reference to the name of the Define Fields tag group.

Example of Environment Specification

```
<Environment>
  <Define>
    <Fields name="MyFields1">
      <Field name="a0" valueString="v0" />
      <Field source="a1" />
      <Fields ref="MyFields2" />
    </Fields>
    <Fields name="MyFields2">
      <Field source="b1" />
      <Fields ref="MyFields3" />
      <Field source="b2" />
    </Fields>
    <Fields name="MyFields3">
      <Field source="c1" />
      <Field source="c2" />
      <Field source="c3" />
    </Fields>
```

```
</Define>
<Binding entity="A" namespace="a">
  <Fields>
    <Fields ref="MyFields1"/>
  </Fields>
</Binding>
<Binding entity="B" namespace="b">
  <Fields>
    <Fields ref="MyFields2"/>
  </Fields>
</Binding>
<Binding entity="C" namespace="c">
  <Fields>
    <Fields ref="MyFields3"/>
  </Fields>
</Binding>
</Environment>
```

This specification results from the following query to environment path translation:

- Query on entity A with column selection a1,b1,b2,c1,c2,c3 inserted into environment “a” path
- Query on entity B with column selection b1,b2,c1,c2,c3 inserted into environment “b” path
- Query on entity C with column selection c1,c2,c3 inserted into environment “c” path

Environment path “a” is additionally contributed with a contribution to the sub-path “a0” having the value “v0” from the specification.

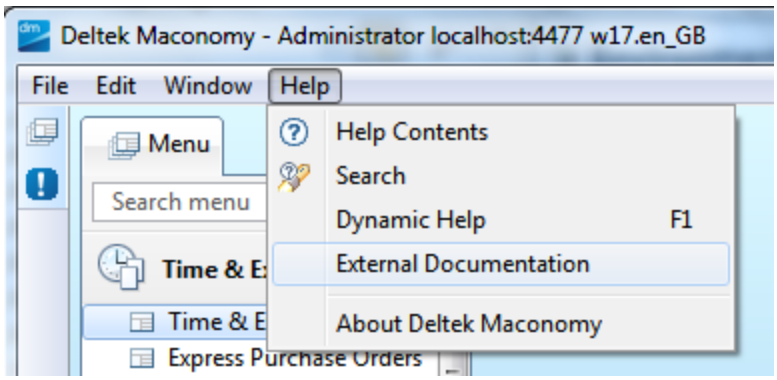
Environment – Reserved Bindings

The environment area of MCSL is used to specify generic bindings that clients can access. Certain bindings are, however, reserved for particular clients. The Workspace Client has a special built-in interpretation of the reserved bindings that described in this section. Other clients, such as Navigator, may impose special semantics on other bindings at a later stage.

External Documentation Link

The primary means of providing end-user documentation in the Workspace Client is through the Help panel (accessed by pressing F1). Adding content to this section is, however, a fairly complex process and requires detailed technical understanding of the Workspace Client architecture.

A simpler and more discrete solution is to provide an external documentation link to an external web site, such as a company intranet. The link appears as a clickable item in the Help group in the menu bar (shown in the following figure).



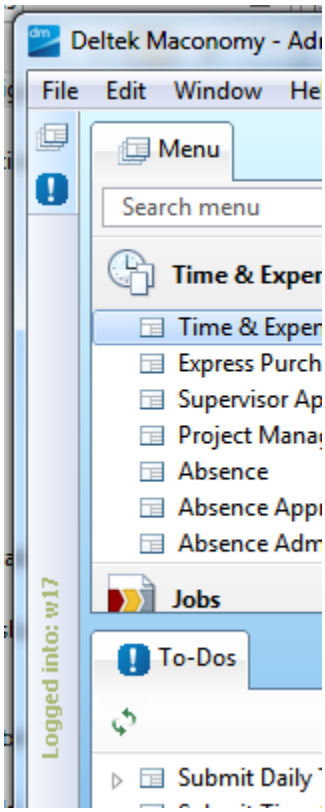
The preceding link appears if the following reserved binding is added to the MCSL environment specification. The value of the **Url** field determines the external documentation link's target. This can be any URL that is accessible from a standard browser on the end user's machine.

```
<Binding namespace="documentation">
  <Fields>
    <Field name="Url" valueString="www.kona.com" />
  </Fields>
</Binding>
```

Menu Dock Decorations

A typical use case during implementation projects is to switch between different systems with the same Workspace Client. To help a user to distinguish between the different systems, the vertical bar left of the menu, known as the *Menu Dock*, can be decorated with data-dependent text. This can, for instance, be used to show the system shortname such that a user can distinguish between a Test and a Production system.

The following figure and code sample show an example configuration. Notice that the Expression Language can be used to make the text data-dependent using the normal placeholder syntax also known from MDML and MWSL.




```
<Binding namespace="client:menudock">

  <Fields>

    <Field name="Title" valueString="^{'Logged into: ' +
shortname()}"/>

    <Field name="Color" valueString="rgb=#abba76" />

    <Field name="TopToBottom" valueString="false" />

    <Field name="FontHeight" valueString="8" />

  </Fields>

</Binding>
```

You can specify another color by assigning another field called "Color" to a valueString that has one of the two formats:

```
rgb=num,num,num (where num is a decimal number in the range 0-255)
```

OR

```
rgb=#xxyyzz (where xx, yy and zz are two-digit hexadecimal numbers in
the range 00-FF).
```

Other attributes that you can specify are:

- FontHeight, which indicates the font height to use (default is 12).
- TopToBottom, which indicates whether the text is drawn top-to-bottom (this is the default) or bottom-to-top (if the value is specified as false).

Configuration Specification

The configuration area of MCSL is used to specify the dictionaries and update sites specifications.

<Configuration>-Tag

The Configuration tag is used in the root MCSL-tag to encapsulate the configuration specification.

Dictionaries Specification

You can specify the list of dictionaries that will be enabled to the client to use (providing the localization into the dictionaries' languages). Specification of the dictionaries is made in MCSL because the dictionaries are contained in the Coupling Service installation.

First, there is a shorthand notation to enable all available dictionaries.

```
<Dictionaries all="true"/>
```

Second, you can restrict the set of available dictionaries, so that the client can use a defined subset.

```
<Dictionaries>
  <Dictionary source="id1"/>
  <Dictionary source="id2"/>
  ...
  <Dictionary source="idn"/>
</Dictionaries>
```

The **id1**,**id2**..., **idn** variables refer to the identifiers of the dictionaries that correspond to the names of the dictionaries files.

Update Sites Specification

The update sites specification is used to define the locations of update sites from which the latest updates for the client can be downloaded.

<UpdateSite>-Tag

An instance of the UpdateSite tag is used to define a single update site location. It has a single attribute **location** that is used for this purpose.

Attribute Name	Type	Usage
location	Key	This attribute refers to the URL that specifies the update site location.

Example of Configuration Specification

```
<Configuration>
  <Dictionaries all="true"/>
  <UpdateSites>
    <UpdateSite location="http://<somelocation>:1234/" />
    <UpdateSite location="http://<otherlocation>:1234/" />
    ...
  </UpdateSites>
</Configuration>
```

Pop-Ups Specification

Pop-ups specification is used to define the filtering and sort order of the pop-up values. The specification consists of a Popups element that specifies global settings for pop-ups filtering and sorting that consists of any number of Popup elements that override the global specification for a particular pop-up type.

<Popups>-Tag

An instance of the Popups tag is used to define the pop-up value filtering and sort order for all of the pop-ups. It has the following attributes.

Attribute Name	Type	Usage
restrain	Expression	This attribute refers to the Boolean expression written in the Expression Language that filters out all of the pop-up values that satisfy it. It applies to all of the pop-up types.
order	Order	This attribute refers to ordering that will be applied to the pop-up values for all of the pop-up types. Valid values are: ▪ Ascending — Alphabetical sort order in increasing order. ▪ Descending — Alphabetical sort order in decreasing order. ▪ Inherent — No sorting applied, the order is as passed by the server. ▪ Standard — One of the preceding values as specified in the configuration; the default is Inherent.

<Popup>-Tag

An instance of the Popup tag is used to define the pop-up value filtering and sort order for a specific pop-up type that is specified by the source attribute. The restrain and order attributes override the values in the Popups element if they have been specified.

Attribute Name	Type	Usage
source	Key	This attribute refers to the pop-up type name.
restrain	Expression	The same as for the restrain attribute in the Popup element, except that it only applies to the pop-up that has the type that is specified in the source attribute.
order	Order	The same as for the order attribute in the Popup element, except that it only applies to the pop-up that has the type that is specified in the source attribute.

Example of Configuration Specification

```
<Configuration>

  <Dictionaries all="true"/>

  <UpdateSites/>

  <Popups restrain="startsWith(popup.literal,'A')"    order="inherent">

    <Popup source="CountryType"
restrain="startsWith(popup.literal,'S') or startsWith(popup.literal,'B')"
order="descending" />

    <Popup source="PostingType" restrain="startsWith(popup.literal,'B')"
order="descending" />

    <Popup source="ZipCodeType" restrain="startsWith(popup.literal,'0')" />

  </Popups>

</Configuration>
```

MDML

Quick Reference

This section is a quick reference to MDML. It briefly describes all of the tags and associated attributes. Additional information can be found in the following documents:

- MDML – Language Reference — This document contains more elaborate examples of the various features of MDML.
- Standard Functions in MDML and EL — This document contains an overview of standard and MDML-specific functions.

Attributes of a given tag are described using a table. An empty table means that there are no attributes.

Attribute Name	Type	Usage
Attr1	*Type of attr1*	This attribute is used to…
Attr2	*Type of attr2*	Indicates…

The referred types of the attributes can be one of the following.

Attribute	Description
Expression (type)	An expression that can use Maconomy standard functions, for example: ▪ envVar('user.info.username') ▪ MDML specific functions such as fieldValue() ▪ Functions defined in the current layout ▪ An included fragment ▪ Global definitions For some attributes, the result type is specified in parenthesis to indicate that it is fixed for this attribute. Valid result types include: ▪ Boolean ▪ Integer ▪ Real ▪ Amount ▪ String ▪ Date ▪ Time ▪ Enum
Boolean	A Boolean attribute (true or false).

Attribute	Description
Integer	An integer value, for example, 1,117.
Key	A string that is case-insensitive and never exposed to an end user. It is used for reference (internally or from other parts in the spec/other specs).
Key List	A space-separated list of keys.
Display	A string that is meant to be displayed to an end user and is therefore localized. It can never be used for any reference.
Id	A case-sensitive string used for referencing items in environments that are not controlled by Maconomy. These are never exposed to an end user.
Styles	A prioritized list of style references. Attributes of this type are evaluated by attempting to apply the listed styles in the order in which they appear. The first style, with either no condition or a satisfied condition, is applied and the rest are ignored for the current evaluation.
(Custom)	Some attributes have a custom type that is only applicable for that specific attribute. In these cases, the type name is in parentheses, and the valid values are described in the Usage column.
Suggestions	This indicates the suggestion strategy for fields that are based on a foreign key. The default behavior is to show suggestions as soon as you start typing, that is, search-as-you-type. Valid values are: ▪ Standard. ▪ None — No pop-up appears. ▪ Automatic — This is the default. ▪ onDemand — This is an extension that allows you to see an unrestricted set of search results.

General Tags

<MDML>-tag

This refers to the general container tag for the layout specification. A layout specification can be one of the following types:

- A concrete layout that is marked by the <Layout>-tag.
- A global definition that is automatically included by all layouts. This is indicated by the <Global>-tag.
- A fragment that contains additional definitions that can be reused across multiple layouts. A fragment is indicated by the <Fragment>-tag.

Attribute Name	Type	Usage
version	(Version Type)	The current version of this layout specification. Uses a number with format **xx.yy** where **xx** is the major version and **yy** is the minor version.

<Layout>-tag

A concrete layout contains views on components, that is, cards, filters, or tables, based on a certain source. A view is a way of rendering a component and may be parameterized. The views are divided into different panes:

- Filter pane
- Pane/upper pane
- Lower pane

Attribute Name	Type	Usage
name	Key	The unique identifier of this layout. In the current version, this attribute does not have any significance.
source	Key	Indicates the source of the component that this concrete layout can render. The source attribute on the concrete layout must match the source that is specified on the corresponding component in MWSL.
style	Styles	The style that applies to this tag and its descendants.

<Global>-tag

The global definition file, marked by the <Global>-tag, has two purposes:

- It is used to specify a global style that all layouts inherit. Changing the style here can have dramatic consequences, so you should exercise some caution.

- It provides definitions that are shared by all layouts. These can be commonly used functions, styles, form groups, and table columns.

Attribute Name	Type	Usage
style	Styles	The style that applies to this tag and its descendants, that is, a global style that is inherited by ALL layouts.

<Fragment>-tag

A fragment is very similar to the global definition file. It is used as a container for reusable functions, styles, form groups, and table columns. The fragment consists of definitions that are included in concrete layouts. The main difference from the global definition is that a fragment must be explicitly included using the <Include> tag in the definitions block of a concrete layout. It is not possible to include fragments in the global definition.

Attribute Name	Type	Usage
name	Key	The unique identifier of the fragment. This name should match the file name of the containing file.

<FilterPane>, <Pane>, <UpperPane>, and <LowerPane>-tags

A concrete layout specification contains a set of views on the underlying source. These views are organized into different panes:

- Filter pane
- Regular/upper pane
- Lower pane

These pane types have similar attributes but different content.

Pane Tag	MWSL Component	Possible Views
<FilterPane>	filter	<Filter>
<Pane>, <UpperPane>	card	<Wizard>, <Form>, <Browser>, <Report>
<LowerPane>	card, table	<Form>, <Browser>, <Report>, <Table>

Their shared attributes are shown in the following table.

Attribute Name	Type	Usage
title	Display	The title that is displayed in the workspace tab if nothing is specified in MWSL for that component. Individual views can override the title that is specified on the pane.

Attribute Name	Type	Usage
style	Styles	The style that applies to this tag and its descendants.

<If><ElseIf><Else>-tags

A non-visual grouping construct that can be used to either include or exclude the members of this tag. The value of the condition attribute can be simple true/false literals or complex expressions using Maconomy standard functions, MDML-specific functions, and functions that are defined in the current layout or in an included fragment or global definition. The evaluation of these conditions takes place each time that the pane receives data. These tags can be used in most places in a layout specification (use syntax completion in your editor to find them all).

Attribute Name	Type	Usage
condition	Expression	The condition that must be satisfied for the children of this tag to be included. This attribute is required on the <If>- and <ElseIf>-tags and prohibited on the <Else>-tag.
default	Boolean	The default attribute indicates a fallback value in case the condition cannot be evaluated without errors. This can be caused by either a syntactically invalid condition or an error in the client environment (for example, invoking a function that is not available).

<Scope> tag

Use the <Scope> tag to declare a scope where certain conditions may apply. For example, if you want to apply a special style to a subset of elements, you can put these elements in a scope and attach a special <Style> tag to that scope. The scope does not have any visual representation and is solely used as a logical grouping construct.

Attribute Name	Type	Usage
style	Styles	The style that applies to this tag and its descendants.

Definitions

<Define>-tag

The <Define> tag contains a set of named definitions of functions, styles, form groups, and table columns. The definitions are made available in the scope in which they are declared. This means that:

- Definitions that are declared in the global definition are available everywhere.
- Definitions that are declared in fragments are available in the panes in which they are included.
- Definitions that are declared in panes are only available in the scope of the declaring pane.

Name collisions are not allowed.

Attribute Name	Type	Usage
-	-	-

<Include>-tag

The <Include> tag indicates the inclusion of a named fragment. The inclusion is performed by macro expansion; that is, the definitions are inserted verbatim into the included definitions block at run time. You cannot use the <Include> tag in the global definition, because definitions that are declared at that level should be kept at a minimum.

Attribute Name	Type	Usage
fragment	Key	The filename of the fragment to include. The file extension (**.mdml.xml**) should be left out.

<Function>-tag

Functions are basically named expressions. Functions can optionally declare parameters. You cannot declare the type of these parameters, so it is solely the responsibility of the caller to invoke a function with arguments of the correct type. A function has a result type that is indicated by the type-attribute. Result types include:

- Boolean (the default)
- Integer
- Real
- Amount
- String
- Date
- Time

- Enum

If the expression is very verbose, you can specify it as XML text data by nesting a <Value> tag within the <Function> tag. The nested <Value> tag and the value attribute are mutually exclusive. The following are examples of a function that is defined in these two alternate ways:

<Function>-tag

```
<Define>

  <Function name="billingPriceLessThanCost"
            value="BillingPriceBaseCurrency &lt;
CostBaseCurrency"/>
```

<Value>-tag

You can embed a CDATA section and thereby avoid replacing **<** with **<**.

```
<Define>

  <Function name="billingPriceLessThanCost">

    <Value><![CDATA[BillingPriceBaseCurrency <
CostBaseCurrency]]></Value>

  </Function>
```

Attribute Name	Type	Usage
name	Key	The name that is used to invoke this function. The name must be unique in the evaluation context.
type	(Field Type)	The result type of the function. If nothing is specified, the default is type Boolean. Valid values are: ▪ Boolean ▪ Integer ▪ Real ▪ Amount ▪ String ▪ Date ▪ Time ▪ Enum
value	Expression	The body of the function. This should be a syntactically valid expression. This expression can invoke other functions if they are available in the same evaluation context and field that are available from the current source.

Attribute Name	Type	Usage
parameters	Key List	A space-separated list of parameter names. Use these names in the body of the function. If nothing is specified, the function does not accept parameters.

<Trigger>-tag

A <Trigger> tag includes the definition of a trigger, which contains a number of assignments, validations, and refresh steps that are executed when the trigger is invoked. A trigger can be defined once and assigned to all types of actions, links, wizards, and wizard pages.

Attribute Name	Type	Usage
name	Key	The name that is used to refer to this trigger. The name must be unique in the evaluation context.
condition	Expression (Boolean)	The trigger only runs when this condition is satisfied. The condition is evaluated each time that the trigger is invoked.

<Assignment>-tag

Use an <Assignment> tag to define an assignment as a trigger execution step. It includes a source field that is used for assignment and either value or valueString attributes.

Attribute Name	Type	Usage
source	Key	The name that is used to invoke this function. The name must be unique in the evaluation context.
value	Expression	Indicates an expression that is evaluated to a value and assigned to the field.
valueString	String	A placeholder that contains a string value and is used for the assignment of string values to the fields.

<Validation>-tag

Use a <Validation> tag to define a validation as a trigger execution step. A validation step includes a condition that is checked. If it is satisfied, the trigger continues to the next execution step; otherwise, the trigger shows an error message and puts the focus in the specified field.

Attribute Name	Type	Usage
condition	Expression (Boolean)	The condition that should be satisfied as a part of the trigger execution step.

<Error>-tag

An <Error> tag defines an error message that is displayed if the validation step fails, as well as the error field to set the focus on.

Attribute Name	Type	Usage
template	(Placeholder String)	The template string that is used as the basis for placeholder substitution and represents the error message. This attribute is required. The format for placeholders is ^ and an integer that represents the index of the argument in the arguments list.
arguments	Expression List	A semicolon-separated list of arguments.
focusField	String	A field name that defines where to put the focus if the validation fails.

<Refresh>-tag

A <Refresh> tag defines the type of the refresh the next time that the pane initiates a request. The current implementation supports triggers that refresh the complete workspace.

Attribute Name	Type	Usage
type	String	The type of refresh for that pane.

Example

Running an action in one pane that creates or copies a record in another pane would require updating the filter pane to include a new record. This can easily be accomplished with a new trigger refresh step that will refresh the entire workspace after the record has been created.

<Style>-tag (Style Definition)

A style definition is similar to a regular style. The main difference is that it can be referred to or from multiple places, depending on its declaration scope. You can name a style definition and include parameters. Parameters are used for conditional styles and are similar to the parameters for the <Function> tag.

Attribute Name	Type	Usage
name	Key	The name that is used to refer to this style. The name must be unique in the evaluation context.
parameters	Key List	A space-separated list of parameter names. These names can be used in the body of the style's condition. If nothing is specified, the style definition does not accept parameters.

Styles

<Style>-tag

Use the <Style> tag to declare a general style to be applied to the tag's immediate parent tag and all descendent tags. It is not necessary to specify all style attributes. If a given attribute is left unspecified, the next style that is higher up in the hierarchy is used. If no such parent style can be found, a client default value is used. If the client default is used instead of inheriting a style from a parent tag, the **Standard** value can be used for all style attributes.

In the following example, a style in which the foreground color is changed to red is applied to the group Employee Information and its two descendant fields **EmployeeNumber** and **EmployeeName**. The green background color is inherited from the <Form> tag, and the remaining style attributes use the client default values because nothing is specified.

```
<MDML version="0.17" xmlns="http://www.maconomy.com/ns/mdml">
  <Layout source="Jobs">
    <Pane>
      <Form>
        <Style backgroundColor="green" />
        <Column>
          <Group title="Employee Information">
            <Style foregroundColor="red" />
            <Field source="EmployeeNumber" />
            <Field source="EmployeeName" />
          </Group>
```

A style is applied to the widgets that correspond to a given MDML element. For a <Field> tag, the style is applied to the widget that renders the title and the widget that renders the value. Some widgets do not respond to certain style attributes. For example, it does not make sense to apply a negative number format to a calendar.

Because colors play a major part in the user interface style, the following table of predefined, named colors has been included. If these colors do not suit the MDML you, you can specify an RGB color using the following syntax: `backgroundColor="rgb=220,255,220"`.

RED	LIME	BLUE	YELLOW
AQUA	FUCHSIA	BLACK	WHITE
GRAY	GREEN	MAROON	NAVY
PURPLE	SILVER	OLIVE	TEAL

The <Style> tag can have several nested tags. These tags, except for <Table> and <Switch> (discussed later), are all used to specify context styles. A context style is a style that is specific to a certain context.

- A label context style, indicated by the <Label> tag, only applies to the title part of an element.
- A field context style, indicated by the <Field> tag, only applies to the value part of an element. The field context styles can be further subdivided into the different types of values that a field can contain:
  - <Boolean>
  - <Integer>
  - <Real>
  - <Amount>
  - <String>
  - <Multiline> (a specialization of string fields)
  - <Date>
  - <Time>
  - <Popup>

Attribute Name	Type	Usage
ref	Key	Apply all of the attribute values from a referred style to the current <Style> tag, except for those that are explicitly defined in the local context. Can also include a list of values separated by semicolons, which could be applied, for example, to the grid cells respectively.
condition	Expression (Boolean)	The style is only applied when this condition is satisfied. The condition is evaluated each time that the pane receives data, that is, during each server communication. This attribute is used to facilitate traffic lighting, for example, emphasizing missing or illegal values.
size	(Size Type)	The size hint that is applied to a given widget. Size hints typically affect the width of the entire widget. Valid values are: ▪ Standard ▪ Tiny ▪ Small ▪ Medium ▪ Large ▪ Huge ▪ Vast

Attribute Name	Type	Usage
anchor	(Anchor Type)	Each widget in the user interface is allocated a certain amount of horizontal space. Anchoring determines where the widget is placed within this space, for example, left, right, or center. You can also specify that the widget stretch to take up the entire available space using the **Both** value. Valid values are: ▪ Standard ▪ Left ▪ Center ▪ Right ▪ Both
fontName	Id	Indicates the font that should be used. The value refers to the name of the font in the operating system.
fontSize	(Font Size Type)	Indicates a font size to be applied to the text of a given widget. Valid values are: ▪ Standard ▪ Tiny ▪ Small ▪ Normal ▪ Large ▪ Huge
foregroundColor	(Color)	Indicates the color of the text in a given widget. Can be a named color, a valid RGB color, or the client default (**Standard**).
backgroundColor	(Color)	Indicates the background color for a given widget. Can be a named color, a valid RGB color, or the client default (**Standard**).
bold	Boolean	Indicates that the text that should be bold. Valid values are: ▪ Standard ▪ True ▪ False
italic	Boolean	Indicates that the text that should be italic. Valid values are: ▪ Standard ▪ True ▪ False

Attribute Name	Type	Usage
underline	Boolean	Indicates that the text should be underlined. Valid values are: ▪ Standard ▪ True ▪ False
justify	(Justification Type)	Justification determines how the text of a widget is placed inside that widget. This is different from anchoring, which determines the placement of the entire widget. Valid values are: ▪ Standard ▪ Left ▪ Center ▪ Right
zeroSuppresion	Boolean	Zero suppression means suppressing or not showing a zero when a field has no value. If the field has the value **zero**, zero suppression does not take effect. In other words, you can use this to distinguish the value **zero** from not having a value at all. Valid values are: ▪ Standard ▪ True ▪ False
height	Integer	Indicates the number of lines that should be displayed for multiline fields. Valid values are: ▪ Standard ▪ Integers
linkForegroundColor	(Color)	Indicates the color of a link in a given text widget. Can be a named color, a valid RGB color, or the client default (**Standard**).
linkUnderline	Boolean	Indicates that the link in a text should be underlined. Valid values are: ▪ Standard ▪ True ▪ False
linkHoverForegroundColor	(Color)	Indicates the color of a link in a given text widget when the mouse hovers above the link. Can be a named color, a valid RGB color, or the client default (**Standard**).

Attribute Name	Type	Usage
linkHoverUnderline	Boolean	Indicates that the link in a text should be underlined when the mouse hovers above the link. Valid values are: ▪ Standard ▪ True ▪ False
spelling	Expression (String)	(Spellcheck is currently only available on OS X). You can enable spellcheck for all open String fields. Valid values are: ▪ **None** — No spell check ▪ **Automatic** — The OS decides on the language to use for spellcheck. The language is derived from the actual text that the user types. ▪ **Standard** — Spelling is configured to the standard setting for this type of field, for example, typically enabled for multiline fields and disabled for single-line fields. ▪ (Locale) — The locale to use for spellchecking, for example, en_US. Refer to Java documentation for valid locales[6].

[6] http://www.oracle.com/technetwork/java/javase/javase7locales-334809.html

<Table>-tag (Table Style)

The table style is specified using a <Table> tag nested within a <Style> tag. Use it to specify a style specifically for the entire table widget.

Attribute Name	Type	Usage
rowHeight	(RowHeight)	Indicates the number of lines that should be displayed for multiline fields in tables. Valid values are: ▪ Standard ▪ 2 ▪ 3 ▪ 4 ▪ 5 ▪ 10 ▪ 15
rowHeightMode	(RowHeightMode)	Indicates how multiline fields should be displayed in tables. Valid values are: ▪ **Standard** ▪ **singleLine** — Multiline fields are displayed as single-line fields. ▪ **fitToContents** — The contents and the width of the table cell decide how many lines are necessary. ▪ **allMultiline** — All rows in the table are displayed as multiline fields. ▪ **selectedMultiline** — Only the selected row displays cells with multiple lines.

<Chart>-tag (Chart Style)

The chart style is specified using a <Chart> tag nested within a <Style> tag. Use this to specify a style specifically for the chart widgets (pie/bar/dial).

Attribute Name	Type	Usage
chartOrientation	(Orientation)	(Bar chart only) Indicates chart orientation: ▪ Vertical (bars are vertical) ▪ Horizontal
textOrientation	(Orientation)	(Bar chart only) Indicates text orientation for (Y) axis values: ▪ Vertical or Horizontal

Attribute Name	Type	Usage
		▪ Decimal values from 90 to 90
showLabels	Boolean	(Bar and pie charts) Indicates whether to display text labels with values on the chart.
showLegend	Boolean	(Bar and pie charts) Indicates whether to display a legend block on the chart.
showGridLines	Boolean	(Bar chart only) Indicates whether to display grid lines on the chart area.
gridLinesColor	(Color)	(Bar chart only) Indicates the color of grid lines. Can be a named color, a valid RGB color, or the client default (**Standard**).
palette	Key	(Bar and pie charts) References a palette to be used for styling individual bars/pie sectors.

<Palette>-tag

Palette is a sequence of styles to be applied to bars in a bar chart and to pie sectors in pie charts. The <Palette> tag is nested within a <Chart> tag. You can add styles to palette as inner <Style> tags or by referencing via style attributes.

Attribute Name	Type	Usage
name	Key	The name that is used to refer to this palette. The name must be unique in the evaluation context.
ref	Key	Apply the values from a referred palette to the current <Palette> tag, except for those that are explicitly defined in the local context.
style	Key List	A list of style names separated by semicolons.

<Switch>-tag

A switch style is a very specialized style that is currently only supported by the Calendar widget. The concept of a switch corresponds to the switch statement in programming languages such as Java. The input that is indicated by the source attribute is interpreted in a piecemeal fashion, where each piece is matched against the nested <Case> tags. Each case branch can have an associated style and tool tip.

The following example shows how you can use a switch style to render the individual days of the calendar. The styling depends on the input that is received from the **TimeSheetsStatusVar** field, which is expected to be a sequence of 0s and 1s.

```
<Calendar title="Time Sheets" source="DateVar">
    <Style fontSize="huge">
      <Switch source="TimeSheetStatusVar">
        <Case char="0">
          <Tooltip text="Approved"/>
          <Style foregroundColor="green"/>
        </Case>
        <Case char="1">
          <Tooltip text="Due"/>
          <Style foregroundColor="red"/>
        </Case>
      </Switch>
    </Style>
</Calendar>
```

Attribute Name	Type	Usage
source	Key	The input source that should be interpreted during evaluation of the switch style.
type	(SwitchType)	The type of switch style. Currently, only the calendar.

<Case>-tag

Each possible type of input value that can be processed during the evaluation of the containing <Switch> tag should be represented by a case branch. See the preceding example.

Attribute Name	Type	Usage
char	Character	A single character from the input source.

<Tooltip>-tag

This refers to the tool tip that corresponds to a given case-branch. See the preceding example.

Attribute Name	Type	Usage
text	Display	The tool tip that is displayed when the mouse hovers above a cell where this case is applied.

Actions

<Actions>-tag

Use the <Actions> tag to specify the contents of the action bar above a pane. If only the <Actions> tag is specified, only the Refresh action is available. By setting the all-attribute to **True** or explicitly nesting the actions, you can achieve a more complex set of actions.

The contents can be made highly dynamic by using expressions in some attributes and in conditionals that can be placed on all levels.

The visible actions in an action bar depend on both the MDML specification and the current state of the pane, that is, if it is in init-, empty-, or exists-state.

Attribute Name	Type	Usage
all	Expression (Boolean)	Indicates whether all of the available actions should be included. The default is **False**, in which case only the Refresh action is displayed.
collapseOrder	Key List	A list of references to named action groups. The order determines which groups are collapsed first when the horizontal space that is available for the action bar decreases.

<Exclude>-tag

Use the <Exclude> tag to exclude certain actions. If you specify an action in this section, it does not appear in the action bar, even if the same action is included somewhere else in the context of the parent <Actions> tag. To facilitate more flexibility, MDML supports conditionals within the exclude section such that actions can be excluded depending on certain conditions.

Attribute name	Type	Usage
-	-	-

<Order>-tag

These actions are divided into two groups: standard actions and default actions. The default order of appearance of actions is:

1. Standard actions (for example, Refresh, Create) are first.
2. Default actions (for example, Convert Job To Order, Close Job) are second.

Use the <Order> tag to revise this default ordering by creating groups and changing the order of appearance.

Any actions that appear in the context of the <Order> tag can only specify the ref-attribute. This is because the <Order> tag is only intended for the grouping and ordering of actions.

Overriding individual properties on an action such as icon or title should be specified before the <Order> tag.

Attribute Name	Type	Usage
standardFirst	Expression (Boolean)	The Standard Actions group should be displayed first. If this attribute is **False**, the Standard Actions group is displayed after the actions and groups specified in the <Order> tag.

<Standard>-tag

Use the <Standard> tag when the Standard Actions group should be placed in between other action groups. If this tag is present in the context of the <Order> tag, it overrides the setting in the standardFirst-attribute on the parent <Order> tag.

Attribute Name	Type	Usage
title	Display	Used to override the title of the Standard Actions group.
tooltip	Display	Indicates the tool tip that is used when the mouse hovers above the Standard Actions group.

<Group>-tag (in the Context of <Actions><Order>)

The default grouping strategy for actions is to split them into two groups:

- **Standard Actions** — These include actions such as refresh, create, update, delete, and so on.
- **Default Actions** — These include application-specific actions such as convert to quote, open job, and so on.

When you use the <Order> tag for custom ordering, you can use the <Group> tag to reorder the pane's current actions. You may, for instance, decide to introduce two new groups in a layout for the Jobs card to group actions that are semantically similar, for example:

```
<Actions all="true" collapseOrder="convert status">
  <Order>
    <Group name="convert" title="Convert">
      <Action ref="ConvertToOrder"/>
      <Action ref="ConvertToQuote"/>
    </Group>
    <Group name="status" title="Status">
      <Action ref="CloseJob"/>
```

```
        <Action ref="ReopenJob"/>
        <Action ref="CopyJob"/>
      </Group>
    </Order>
  </Actions>
```

You should name custom groups so that they can take part in the action bar's collapse order, that is, the order in which a row of actions collapses into groups when the available horizontal space decreases.

Attribute Name	Type	Usage
name	Key	You should name groups uniquely so that they can be referred to from the collapseOrder-attribute on the <Actions> tag as displayed in the preceding example. Aside from that, the name does not play any significant role.
icon	Id	The resource identifier for an icon to be associated with this action group.
title	Display	The displayable title of this group. The title is required and should convey a common characteristic of its members.
appearance	(Action Appearance Type)	The visual appearance of an action group determines how it is rendered in the user interface. Should the title, the icon, or both be displayed? Valid values are: ▪ Standard ▪ All ▪ Title ▪ Icon
tooltip	Display	Indicates a tool tip title that is displayed when the mouse hovers above the action group.
grouped	(Grouping Type)	Indicates the collapse behavior of the group. Valid values are: ▪ **Always** — The group always appears as a group. ▪ **Dynamic** — The member actions collapse to a group when there is insufficient space available. ▪ **Never** — The member actions never collapse into a group.

Attribute Name	Type	Usage
collapseOrder	Key List	Indicates the order in which the individual member actions are collapsed into the group when the available space decreases.

<Action>-tag

Use the <Action> tag to include an application action or a standard action from another pane, that is, an external action.

- If the all-attribute on the <Actions> tag is **True**, use the <Action> tag to override individual properties of an action.
- If the all-attribute is **False**, the <Action> tag is also used to include this action.

Attribute Name	Type	Usage
name	Key	The unique name of this action. Use this to distinguish actions with the same source. If no name is specified, the name defaults to the same name as the source.
ref	Key	The ref-attribute is used in the context of the <Order> tag to refer to previously defined actions during reordering. The ref-attribute is the only legal attribute in this context.
icon	Id	The resource identifier for an icon to be associated with this action.
title	Display	The displayable title for this action. If nothing is specified, the action takes its title from the server specification.
appearance	(Action Appearance Type)	The visual appearance of an action determines how it is rendered in the user interface. Should the title, the icon, or both be displayed? Valid values are: ▪ Standard ▪ All ▪ Title ▪ Icon
tooltip	Display	Indicates a tool tip title that is displayed when the mouse hovers above the action. If nothing is specified, the tool tip is the same as the title.
source	Key	The external source identifier of this action. This attribute is required. The source can be a qualified identifier that makes it possible to refer external actions, for example, **parent.create**.

Attribute Name	Type	Usage
wizard	Key	The identifier of the wizard that is associated with this action. When you specify a wizard, invoking the action causes the wizard to be invoked. The actual invocation of the action against the server is postponed until the sequence of wizard pages has been completed.
preTrigger	Key	The identifier of the trigger that is associated with this action. When you specify a preTrigger, invoking the action calls the trigger to run before the action is executed. If the trigger runs successfully, the action is executed normally. If the trigger fails, the action is never executed, and the error message is displayed.

<Create>-tag

Use the <Create> tag to include the Create action.

- If the all-attribute on the <Actions> tag is **True**, use the <Create> tag to override individual properties of the Create action.
- If the all-attribute is **False**, the <Create> tag is also used to include this action.

Create action can define two triggers:

- initPostTrigger — Executed after the init action is invoked.
- preTrigger — Executed just before the final create.

Attribute Name	Type	Usage
name	Key	The unique name of this action. Use this to distinguish actions with the same source. If no name is specified, the name defaults to the same name as the source.
ref	Key	The ref-attribute is used in the context of the <Order> tag to refer to previously defined actions during reordering. The ref-attribute is the only legal attribute in this context.
icon	Id	The resource identifier for an icon to be associated with this action.
title	Display	The displayable title of this action. If nothing is specified, the action receives its title from the server specification.
appearance	(Action Appearance Type)	The visual appearance of an action determines how it is rendered in the user interface. Should the

Attribute Name	Type	Usage
		title, the icon, or both be displayed? Valid values are: ▪ Standard ▪ All ▪ Title ▪ Icon
tooltip	Display	Indicates a tool tip title that is displayed when the mouse hovers above the action. If nothing is specified, the tool tip is the same as the title.
wizard	Key	The identifier of the wizard that is associated with this action. When you specify a wizard, invoking the action causes the wizard to be invoked. The actual invocation of the action against the server is postponed until the sequence of wizard pages has been completed.
initPostTrigger	Key	The identifier of the trigger that is associated with the Init part of the <Create> action. When you specify an initPostTrigger, it is called after the init action is invoked, and before you enter any data. ▪ If the trigger runs successfully, you can enter data. ▪ If the trigger fails, the error message is displayed.
preTrigger	Key	The identifier of the trigger that is associated with the Create part of the action. When you specify a preTrigger, invoking Create calls the trigger just before the action is executed. ▪ If the trigger runs successfully, the Create action is executed normally. ▪ If the trigger fails, the action is never executed, and the error message is displayed.

Create action in <Table> context can represent Add-row or Insert-row action. Therefore, in tables, you can define an <Add> tag and <Insert> tag and override individual properties of the <Create> action, such as title in the following example:

```
<Create>
    <Add title="Add task" />
    <Insert title="Insert task" />
</Create>
```

<Update>, <Delete>, <Refresh>, <Move>, <Indent>, <PrintThis>-tag

Use these tags to include the Update, Delete, Refresh, and PrintThis actions. Insert the <Move> tag to enable the Move Up and Move Down actions in tables. Insert the <Indent > tag to enable the Indent and Outdent actions in tree tables.

- If the all-attribute on the <Actions> tag is **True**, use the preceding tags to override individual properties of these actions.
- If the all-attribute is **False**, the preceding tags are also used to include the corresponding actions.

Attribute Name	Type	Usage
name	Key	The unique name of this action. Use this to distinguish actions with the same source. If no name is specified, the name defaults to the same name as the source.
ref	Key	Use the ref-attribute in the context of the <Order> tag to refer to previously defined actions during reordering. The ref-attribute is the only legal attribute in this context.
icon	Id	The resource identifier for an icon to be associated with this action.
title	Display	The displayable title of this action. If nothing is specified, the action receives its title from the server specification.
appearance	(Action Appearance Type)	The visual appearance of an action determines how it is rendered in the user interface. Should the title, the icon, or both be displayed? Valid values are: ▪ Standard ▪ All ▪ Title ▪ Icon

Attribute Name	Type	Usage
tooltip	Display	Indicates a tool tip title that is displayed when the mouse hovers above the action. If nothing is specified, the tool tip is the same as the title.
preTrigger	Key	The identifier of the trigger that is associated with this action. When you specify a preTrigger, invoking the action calls the trigger to run before the action is executed. ▪ If the trigger runs successfully, the action is executed normally ▪ If the trigger fails, the action is never executed, and the error message is displayed.

<Print>-tag

Use the <Print> tag in the Print action.

- If the all-attribute on the <Actions> tag is **True**, the <Print> tag overrides individual properties of the Print action.
- If the all-attribute is **False**, the <Print> tag is also used to include this action.

Attribute Name	Type	Usage
name	Key	The unique name of this action. Use this to distinguish actions with the same source. If no name is specified, the name defaults to the same name as the source.
ref	Key	Use the ref-attribute in the context of the <Order> tag to refer to previously defined actions during re-ordering. The ref-attribute is the only legal attribute in this context.
icon	Id	The resource identifier for an icon to be associated with this action.
title	Display	The displayable title for this action. If nothing is specified, the action receives its title from the server specification.
appearance	(Action Appearance Type)	The visual appearance of an action determines how it is rendered in the user interface. Should the title, the icon, or both be displayed? Valid values are: ▪ Standard ▪ All ▪ Title

Attribute Name	Type	Usage
		▪ Icon
tooltip	Display	Indicates a tool tip title that is displayed when the mouse hovers above the action. If nothing is specified, the tool tip is the same as the title.
layout	Id	Indicates the MPL-layout that is used to produce this report.
wizard	Key	The identifier of the wizard that is associated with this action. When you specify a wizard, invoking the action causes the wizard to be invoked. The actual invocation of the action against the server is postponed until the sequence of wizard pages has been completed.
preTrigger	Key	The identifier of the trigger that is associated with this print action. When you specify a preTrigger, invoking the action calls the trigger to run before the action is executed. ▪ If the trigger runs successfully, the action is executed normally ▪ If the trigger fails, the action is never executed, and the error message is displayed.

<Report>-tag

Use the <Report> tag in the Report action.

- If the all-attribute on the <Actions> tag is **True**, the <Report> tag overrides individual properties of the Report action.
- If the all-attribute is **False**, the <Report> tag is also used to include this action.

Attribute Name	Type	Usage
name	Key	The unique name of this action. Use this to distinguish actions with the same source. If no name is specified, the name defaults to the same name as the source.
ref	Key	Use the ref-attribute in the context of the <Order> tag to refer to previously defined actions during reordering. The ref-attribute is the only legal attribute in this context.
icon	Id	The resource identifier for an icon to be associated with this action.

Attribute Name	Type	Usage
title	Display	The displayable title of this action. If nothing is specified, then the action receives its title from the server specification.
appearance	(Action Appearance Type)	The visual appearance of an action determines how it is rendered in the user interface. Should the title, the icon, or both be displayed? Valid values are: ▪ Standard ▪ All ▪ Title ▪ Icon
tooltip	Display	Indicates a tool tip title that is displayed when the mouse hovers above the action. If nothing is specified, the tool tip is the same as the title.
source	Expression (String)	Indicates the source of the report. The precise definition of the source depends on the third-party system that produces the report. This attribute is required.
view	Expression (String)	Indicates the view to be used from the source of the report. The precise definition of the view depends on the third-party system that produces the report. The default is the empty string.
output	(Output Type)	The output format of the report. You can use a report action to open a report inline or in an external window. The default format of the report is HTML. Valid values are: ▪ html ▪ pdf ▪ xls
preTrigger	Key	The identifier of the trigger that is associated with this report action. When you specify a preTrigger, invoking the action calls the trigger to run before the action is executed. ▪ If the trigger runs successfully, the action is executed normally ▪ If the trigger fails, the action is never executed, and the error message is displayed.

<Link>-tag

Use the <Link> tag in the Link action. You can have several link actions, even with the same underlying target. Links can be made data-dependent by embedding a dynamic element in the url-attribute or in the nested <Url> tag.

See the description of the <Url> tag in the browser for documentation of this construct.

In addition, links can also navigate to another workspace. The following is an example of a data-dependent link action that generates a mailto-link:

```
<Actions>

  <Link name="mailTo" title="Send a mail">

    <Url value="'mailto:' + ElectronicEmailAddress">

      <Query field="subject"

             value="Regarding your time sheet for the period
from ^1 to ^2."

             arguments="PeriodStart;PeriodEnd" />

      <Query field="body"

             value="Dear ^1, we need to discuss your time
sheet. Your Manager"

             arguments="EmployeeName" />

    </Url>

  </Link>
```

Attribute Name	Type	Usage
name	Key	The unique name of this action. Use this to distinguish actions with the same source. If no name is specified, the name defaults to the same name as the source.
ref	Key	Use the ref-attribute in the context of the <Order> tag to refer to previously defined actions during reordering. The ref-attribute is the only legal attribute in this context.
icon	Id	The resource identifier for an icon to be associated with this action.
title	Display	The displayable title of this action. Because the Link action does not have any specification from the server, but is solely defined in the MDML layout, the title attribute is required.

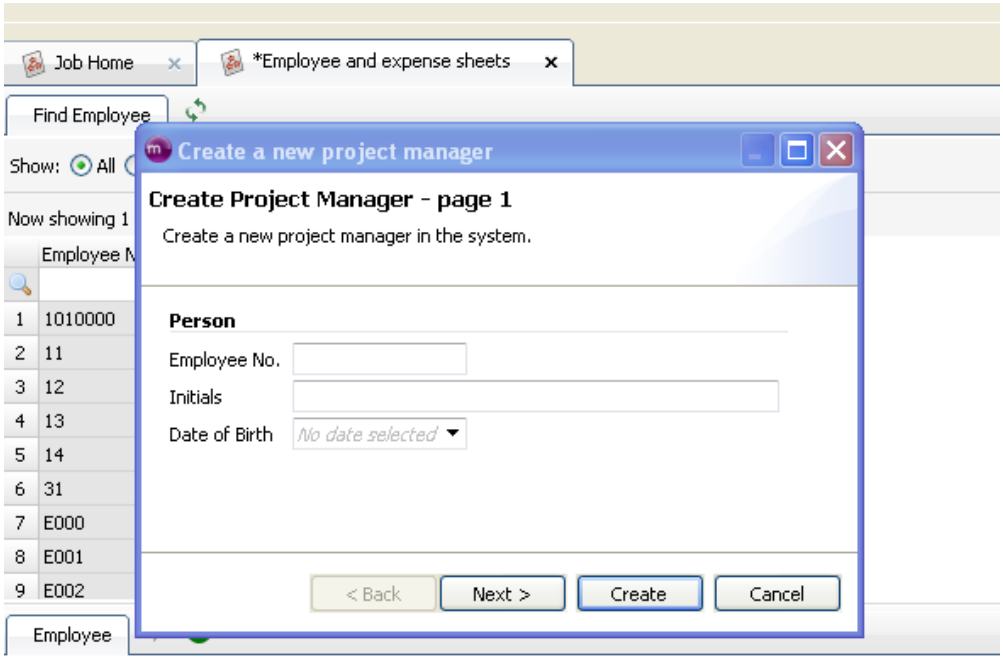
Attribute Name	Type	Usage
		The target URL of an action can be data-dependent, so you should exercise some caution when choosing a relevant title.
appearance	(Action Appearance Type)	The visual appearance of an action determines how it is rendered in the user interface. Should the title, the icon, or both be displayed? Valid values are: ▪ Standard ▪ All ▪ Title ▪ Icon
tooltip	Display	Indicates a tool tip title that is displayed when the mouse hovers above the action. If nothing is specified, the tool tip is the same as the title.
url	Expression (String)	A string expression that results in a URL. Alternatively, you can use the nested <Url> tag as displayed in the preceding example.
preTrigger	Key	The identifier of the trigger that is associated with this link action. When you specify a preTrigger, invoking the action calls the trigger to run before the action is executed. ▪ If the trigger runs successfully, the action is executed normally. ▪ If the trigger fails, the action is never executed, and the error message is displayed.

Wizards

<Wizard>-tag

A wizard is a sequence of pages used to elicit information for a particular action. You render a wizard as a model dialog, which means that it prevents you from doing other tasks from the time that it is invoked until it is completed.

You can specify the layout for each page using a normal <Form> tag. The main difference from a normal form is that the wizard does not submit its data until the entire sequence of pages is complete.



Another noticeable difference is that evaluation of <If><ElseIf><Else> tags uses a combination of the new and old values of the underlying pane. This means that wizard conditionals can depend on data that was entered on a previous page.

The following example shows a simple two-page wizard with a data-dependent description on the second page:

```
<UpperPane title="Project Manager">

  <Wizard name="CreateWizard" title="Create a new project
manager"

          description="Create a new project manager in the
system.">

    <Page form="CreateWizardPage1" title="Create Project
Manager - page 1"/>

    <Page form="CreateWizardPage2" title="Create Project
Manager - page 2">

      <Description template="Enter detailed information for
^1"
```

```
                    arguments="EmployeeName"/>
        </Page>
    </Wizard>
```

Attribute Name	Type	Usage
name	Key	Wizards must have unique names that can be referred from the actions that invoke them. The name must be unique only in the scope of the current MDML pane.
title	Display	The wizard's title is displayed as the window or dialog title. This title should convey the purpose of the action that this wizard is used to accomplish.
description	Display	A default description that is displayed on each page if nothing is specified below the <Page> tag. The description is used to explain the purpose of the page. An alternative to using this attribute is to nest a <Description> tag inside this tag. This makes it possible to use a data-dependent description by means of placeholder substitution.
mandatory	Expression	Specifies the default mandatories of the pages in this wizard. If left out, the value is considered **True**.
postTrigger	Key	The identifier of the trigger that is associated with this wizard. When you specify a preTrigger for the wizard, it is invoked on the last page of the Wizard after submitting. ▪ If the trigger runs successfully, the wizard completed normally ▪ If the trigger fails, the error message is displayed, and the focus is set in the error field.

<Page>-tag

Each wizard page is rendered using a normal form. As displayed in the preceding screenshot, a wizard page has both a title and a description to convey its purpose.

Attribute Name	Type	Usage
name	Key	The unique identifier of the individual wizard page.
title	Display	The required title of the wizard page. This title should convey the purpose of this page.
description	Display	A description is intended to explain the purpose of the page. An alternative to using this attribute is to nest a <Description> tag inside this tag. This makes it possible to use a data-dependent description by means of placeholder substitution. If nothing is specified, the description is inherited from the containing wizard.
form	Key	The identifier of the form with the layout for this page. The form should be defined in the same MDML pane as the wizard.
mandatory	Expression	Defines whether this page is considered mandatory. You *must* visit a mandatory page before the wizard can be completed. This only applies to the current page path; therefore, if there are different paths in the wizard, only the path that is applicable for the current values is considered. If this value is not specified, the value defaults to the mandatory value of the wizard tag.
preTrigger	Key	The identifier of the trigger that is associated with the wizard page when it is initially loaded. When you specify a preTrigger, it is called when the wizard page is initially loaded and before you enter any data. If the trigger runs successfully, you can data. If the trigger fails, an error message is displayed, and the focus is set in the error field.
postTrigger	Key	The identifier of the trigger that is associated with the wizard page when it is submitted. When you specify a postTrigger, it is called after you enter data and move to the next page and finish the wizard. ▪ If the trigger runs successfully, the wizard continues normally.

Attribute Name	Type	Usage
		▪ If the trigger fails, an error message is displayed, and the focus is set in the error field.

<Description>-tag (in the context of <Wizard>)

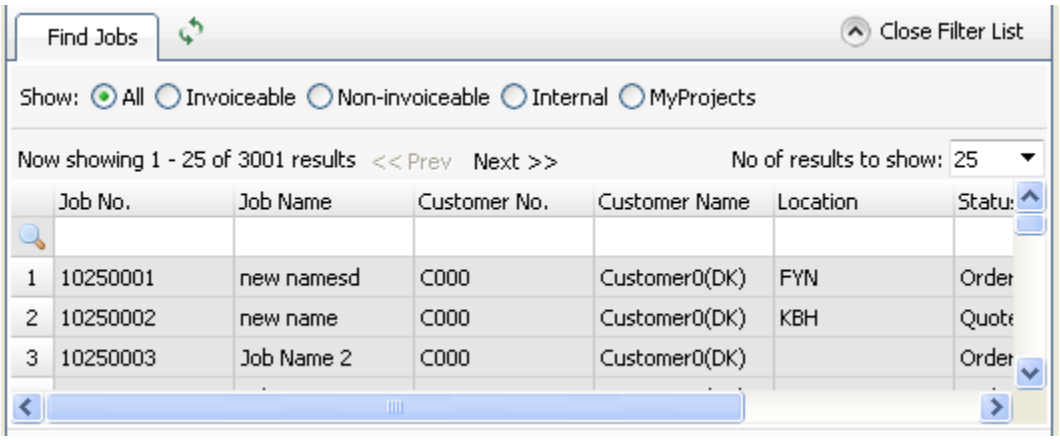
A description is an elaboration of the purpose of a wizard page. The description can be made data-dependent by using a placeholder substitution. The template-attribute is used as a template, and each placeholder (for example, ^1, ^2…^n) is substituted by the current value of the argument on evaluation time.

In the preceding example, the **^1** placeholder is replaced by the name of the employee. The description may also be specified in the <Define> block and referenced by ref attribute.

Attribute Name	Type	Usage
template	(Placeholder String)	The template string that is used as the basis for placeholder substitution. This attribute is required. The format for placeholders is **^** and an integer that represents the index of the argument in the arguments list.
arguments	Expression List	A semicolon-separated list of arguments.
ref	String List	Reference to the description that is specified in the <Define> block.

Filters

The following is a screenshot of a filter view as rendered in the client.



<Filter>-tag

A filter is a view to render the MWSL component filter type. Conceptually, a filter can exist in two different modes: normal and compact. The main child tag of a filter is its <Columns> tag, which holds a list of the displayed columns and their underlying field.

See the section about tables for documentation of this tag.

The normal mode is the expanded version, with a filter table and a control that are often seen in the top of a workspace's hierarchy of components. The normal mode is also used to lay out the model dialog, called advanced search, which is used during foreign key searches (CTRL+G searches).

Use the compact mode when the normal filter in a workspace is collapsed. This happens when you double-click a record. In this case, the filter only shows a single record in a summarized form (by means of the <Description> tag). The compact mode of the filter is also used with value pickers. The value picker widget is a variant of foreign key searches, where the results of a limited search are displayed in a small pop-up window. The layout of this window is determined by a compact filter that can be specified in MDML.

A set of filter extensions are available to improve MDML filtering performance.

Attribute Name	Type	Usage
name	Key	The name of this view. The name should uniquely identify this within the scope of the containing MDML pane.
title	Display	Use the view title to override the title that is defined on the pane. This does, however, not take effect if a title is defined on the corresponding MWSL component.

Attribute Name	Type	Usage
style	Styles	The style that applies to this tag and its descendants.
open	(Open Type) or Boolean Expression	The open state of this tag and all its descendants. Valid values are: ▪ False ▪ Never ▪ Create ▪ Update ▪ Always ▪ True In addition, you can specify any Boolean expression that can be evaluated in the relevant context. In this way, you can make the open state dependent on the context. For example, you might want to close certain fields on summary lines but not on non-summary lines.
loginRules	Expression List	A semicolon-separated list of additional login rules that are applicable for this view.

<Parameters>-tag

Any view can accept parameters from the corresponding MWSL specification. Use the <Parameters> tag to hold these parameter declarations. If a view with parameters is invoked from MWSL without arguments, an exception occurs.

Attribute Name	Type	Usage
-	-	-

<Parameter>-tag

Each view parameter has a mandatory name and type. The name is used to bind variables in the view during the evaluation of conditionals and expressions. It should therefore be unique and not overlap any existing identifiers in the evaluation context.

Attribute Name	Type	Usage
name	Key	The name of the parameter. This name can be used in the conditionals and expressions of this view as a valid identifier. The name is required.
type	(Field Type)	The type of the parameter. Failure to comply with the type restriction that is imposed by this

Attribute Name	Type	Usage
		declaration causes an exception when invoking the view from a MWSL specification. Valid values are: ▪ Boolean ▪ Integer ▪ Real ▪ Amount ▪ String ▪ Date ▪ Time ▪ Enum

<ControlBar>-tag

The control bar of a filter is a visual component used to show various information about the filter. The primary purpose is to show the available selections (or search restrictions) on the data in the filter table.

Attribute Name	Type	Usage
style	Styles	The style that applies to this tag and its descendants.

<Selection>-tag

A selection is a collection of search restrictions that you can apply to the data in the filter. Each search restriction is represented by a nested <Option> tag.

Attribute Name	Type	Usage
name	Key	The unique identifier of this selection.
title	Display	The title that is displayed before the actual options.
default	Key	The default selected option.
style	Styles	The style that applies to this tag and its descendants.
restriction	Expression (Boolean)	Specifying a restriction on the selection enforces the restriction as a *minimal* restriction. That is, no matter which option you select, this restriction is always enforced.

Attribute Name	Type	Usage
		If a <Restriction> tag is put directly under the <Selection> tag, it takes precedence over the restriction attribute.

<Restriction>-tag

The <Restiction> tag has the same semantics as the restriction attribute of the <Selection> tag. If both are specified, the restriction of the <Restriction> tag is used instead of the restriction-attribute. The Restriction tag gives the layout author the option of specifying more complex restrictions using newlines, indentation, and standard relational operators such as **<** and **>** instead of the more acquired **`lt`** and **`gt`**.

```
<Selection>

  <Restriction> <![CDATA[

    not Template

    or LocationName = locationNumber(currentDate())

    ]]>

  </Restriction>

</Selection>
```

This selection ensures that only records that are not templates, or where the location name happens to be associated with the current user, are matched by the filter; therefore, it makes a difference how the filter is invoked.

In particular, this means that if this filter is addressed by a link (for example, a notification link), only records that fulfill the general restriction are displayed, even if explicitly mentioned in the link.

<Option>-tag

The <Option> tag corresponds to a search restriction on the filter data. This search restriction is defined as a predicate using the expression language. The restriction can either be defined in the restriction-attribute or in a nested <Restriction> tag. The two forms are mutually exclusive.

```
<Option title="Pending Approval" restriction="not Approved"
/>

The nested <Restriction>-tag is recommended for complex
restriction, especially those where a CDATA section is needed
to facilitate readability.

<Option title="Pending Approval">

  <Restriction>not Approved</Restriction>

</Option>
```

Attribute Name	Type	Usage
name	Key	The name of this option. The name should be unique within the scope of the current selection and

Attribute Name	Type	Usage
		can be referred from the default-attribute on the <Selection>-tag.
title	Display	The title that is associated with this search restriction. This attribute is required.
restriction	Expression (Boolean)	The predicate that represents the search restriction. This expression is translated to an MQL expression on the server.
style	Styles	The style that applies to this tag.
sortColumn	Key	The name of the column that is sorted when a given filter option is selected.
sortOrder	(Sort OrderType)	The attribute that defines the sorting to be applied on a column. Valid values are: ▪ Ascending ▪ Descending

<Compact>-tag

This refers to the specification of the compact mode of this filter. If no <Compact> tag is specified, an auto-generated compact filter is inserted that contains the same columns as its containing filter. Specifying the compact mode of a filter allows the MDML author to specify custom columns using the <Columns> tag.

See the section about tables for documentation of this tag.

Attribute Name	Type	Usage
maxRows	Integer	The maximum number of rows that are visible when the compact filter is used to render the pop-up window from a value picker. In other words, this is a predefined window height (measured in table rows).
maxSuggestions	Integer	The maximum number of suggestions that are retrieved during a foreign key search for value pickers in “automatic” mode, that is, search-as-you-type mode.
style	Styles	The style that applies to this tag and its descendants.

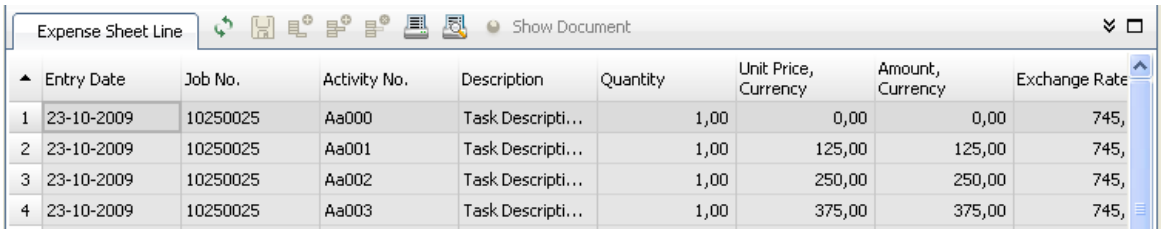
<Description>-tag (in the Context of <Compact>)

A description is a summary of the currently selected record in a compact filter. The description is made data-dependent by using placeholder substitution. The template-attribute is used as a template, and each placeholder (for example, ^1, ^2…^n) is substituted by the current value of the argument on evaluation time. The description may be also specified in the <Define> block and referenced by a ref-attribute.

Attribute Name	Type	Usage
template	(Placeholder String)	The template string that is used as the basis for placeholder substitution. This attribute is required. The format for placeholders is ^ and an integer representing the index of the argument in the arguments list.
arguments	Expression List	A semicolon-separated list of arguments.
ref	String List	Reference to the description that is specified in the <Define> block.

Tables

The following figure is a screenshot of a table view as rendered in the client.



	Entry Date	Job No.	Activity No.	Description	Quantity	Unit Price, Currency	Amount, Currency	Exchange Rate
1	23-10-2009	10250025	Aa000	Task Descripti...	1,00	0,00	0,00	745,
2	23-10-2009	10250025	Aa001	Task Descripti...	1,00	125,00	125,00	745,
3	23-10-2009	10250025	Aa002	Task Descripti...	1,00	250,00	250,00	745,
4	23-10-2009	10250025	Aa003	Task Descripti...	1,00	375,00	375,00	745,

<Table>-tag

A table is a view that is used to render the MWSL component table type. A table view can take parameters similar to a filter view.

Attribute Name	Type	Usage
name	Key	The name of this view. The name should uniquely identify this view within the scope of the containing MDML pane.
title	Display	Use the view title to override the title that is defined on the pane. This does, however, not take effect if a title is defined on the corresponding MWSL component.
style	Styles	The style that applies to this tag and its descendants.
open	(Open Type) or Boolean Expression	The open state of this tag and all its descendants. Valid values are: ▪ False ▪ Never ▪ Create ▪ Update ▪ Always ▪ True In addition, you can specify any Boolean expression that can be evaluated in the relevant context. In this way, you can make the open state dependent on the context.
loginRules	Expression List	Indicates additional login rules that are applicable for this view.

<Parameters>-tag

Any view can accept parameters from the corresponding MWSL specification. The <Parameters> tag is used to hold these parameter declarations. If a view with parameters is invoked from MWSL without arguments, an exception occurs.

Attribute Name	Type	Usage
-	-	-

<Parameter>-tag

Each view parameter has a mandatory name and type. The name is used to bind variables in the view during evaluation of conditionals and expressions. It should therefore be unique and not overlap any existing identifiers in the evaluation context.

Attribute Name	Type	Usage
name	Key	The name of the parameter. This name can be used in the conditionals and expressions of this view as a valid identifier. The name is required.
type	(Field Type)	The type of the parameter. Failure to comply with the type restriction that is imposed by this declaration causes an exception when invoking the view from a MWSL specification. Valid values are: ▪ Boolean ▪ Integer ▪ Real ▪ Amount ▪ String ▪ Date ▪ Time ▪ Enum

<Columns>-tag

The collection of columns used in a table, filtertable, or compact filtertable. Apart from various table elements, you can also insert other <Columns> tags as child tags.

When used with the ref-attribute, it facilitates macro expansion of columns from an included fragment, a defines-block, or the global definition.

The following is an example from a filter table:

```
<Filter>
  <Columns>
    <Description title="Summary"
                 template="^1 (^2), ^3" arguments="JobName
JobNumber Name1"/>
    <Columns ref="jobFilterColumns"/>
  </Columns>
```

Attribute Name	Type	Usage
name	Key	A unique name for this collection of columns. It is primarily used inside <Define> tags to declare a collection of table elements that can be inserted in other layouts.
ref	Key	The reference attribute is used when inserting referred fields, that is, the macro expansion of fields.
style	Styles	The style that applies to this tag and its descendants.

<Field>-tag (in the Context of <Columns>)

A table field is a column that is based on a single source (for example, a database field).

Attribute Name	Type	Usage
name	Key	A unique identifier for this table column.
mandatory	Boolean Expression	Indicates whether it is mandatory to have a value for this field when submitting.
visibility	(Visibility Type)	Indicates the visibility of this table column. Valid values are: ▪ Required ▪ Visible

Attribute Name	Type	Usage
		▪ Hidden
open	(Open Type) or Boolean Expression	The open state of this column. Valid values are: ▪ False ▪ Never ▪ Create ▪ Update ▪ Always ▪ True In addition to this, you can specify any Boolean expression that can be evaluated in the relevant context. In this way, you can make the open state dependent on the context.
size	(Size Type)	Indicates the size (width) of the column. Valid values are: ▪ Standard ▪ Tiny ▪ Small ▪ Medium ▪ Large ▪ Huge ▪ Vast
autoSubmit	Boolean	Indicates whether this field should be automatically submitted when it is edited.
searchLayout	Key	The layout that is used if this element launches a foreign key search (CTRL+G). If nothing is specified, a default search layout (for example, **Find_X.mdml.xml**) is retrieved from the server.
searchView	Key	The layout view that is used if this element launches a foreign key (CTRL+G) search. The view should be part of the layout that is specified in the searchLayout-attribute. If nothing is specified, the first view from the search layout is chosen.
searchOption	Key	Indicates the name of the search option to be applied when doing “lightweight” searches from within the field. The name must match the name of the filter layout/view implicitly or explicitly stated.
suggestions	Suggestions	Indicates the suggestion strategy for fields that are based on a foreign key. Valid values are:

Attribute Name	Type	Usage
		▪ Standard ▪ None ▪ Automatic ▪ onDemand
title	Display	The title that is displayed as column header. This attribute is used for static titles.
titleValue	Key	The title that is displayed as column header. This attribute is used for titles derived from the value of a referred field.
titleSource	Key	The title that is displayed as column header. This attribute is used for titles derived from the title of a referred field.
suggestions	Suggestions	Indicates the suggestion strategy for fields based on a foreign key. Valid values are: ▪ Standard ▪ None ▪ Automatic ▪ onDemand
source	Key	The source (for example, a database field) of this table column.
style	Styles	The style that applies to this tag and its descendants.

<UnitField>-tag (in the Context of <Columns>)

A table unit field is a column based on one or two sources (for example, database fields). A primary field that provides the value and optionally a unit source which provides the unit. The unit can alternatively be a simple static title.

Attribute Name	Type	Usage
name	Key	A unique identifier for this table column.
mandatory	Boolean Expression	Indicates whether it is mandatory to have a value for this field when submitting.
visibility	(Visibility Type)	Indicates the visibility of this table column. Valid values are: ▪ Required ▪ Visible

Attribute Name	Type	Usage
		▪ Hidden
open	(Open Type) or Boolean Expression	The open state of this column. Valid values are: ▪ False ▪ Never ▪ Create ▪ Update ▪ Always ▪ True In addition to this, you can specify any Boolean expression that can be evaluated in the relevant context. In this way, you can make the open state dependent on the context.
size	(Size Type)	Indicates the size, that is, width, of the column. Valid values are: ▪ Standard ▪ Tiny ▪ Small ▪ Medium ▪ Large ▪ Huge ▪ Vast
autoSubmit	Boolean	Indicates whether this field should be automatically submitted when it is edited.
searchLayout	Key	The layout that is used if this element launches a foreign key (CTRL+G) search. If nothing is specified, a default search layout (for example, **Find_X.mdml.xml**) is retrieved from the server.
searchView	Key	The layout view that is used if this element launches a foreign key (CTRL+G) search. The view should be part of the layout specified in the searchLayout-attribute. If nothing is specified, the first view from the search layout is chosen.
searchOption	Key	Indicates the name of the search option to be applied when doing “lightweight” searches from within the field. The name must match the name of the filter layout/view implicitly or explicitly stated.
suggestions	Suggestions	Indicates the suggestion strategy for fields based on a foreign key. Valid values are:

Attribute Name	Type	Usage
		▪ Standard ▪ None ▪ Automatic ▪ onDemand
title	Display	The title that is displayed as column header. This attribute is used for static titles.
titleValue	Key	The title that is displayed as column header. This attribute is used for titles derived from the value of a referred field.
titleSource	Key	The title that is displayed as column header. This attribute is used for titles derived from the title of a referred field.
suggestions	Suggestions	Indicates the suggestion strategy for fields based on a foreign key. Valid values are: ▪ Standard ▪ None ▪ Automatic ▪ onDemand
source	Key	The source (for example, a database field) of this table column.
style	Styles	The style that applies to this tag and its descendants.
unitSource	Key	The source of the unit for this field.
unitTitle	Display	A static unit.
unitPosition	(Unit Position Type)	Indicates the position of the unit in relation to the field value. It can have either a hardcoded position or let the Currency-setting of the operating system decide (currently, this can be set in the client preferences). Valid values are: ▪ Standard ▪ Prefix ▪ Postfix ▪ Currency

<Description>-tag (in the Context of <Columns>)

A description is a summary of the currently selected record in a table. The description should be made data-dependent by using a placeholder substitution. The template-attribute is used as a template, and each placeholder (for example, ^1, ^2…^n) is substituted by the current value of the argument on evaluation time. The description can be specified in the <Define>-block and referenced by ref attribute.

Attribute Name	Type	Usage
title	Display	The title that is displayed as column header. This attribute is used for static titles.
titleValue	Key	The title that is displayed as column header. This attribute is used for titles that are derived from the value of a referred field.
titleSource	Key	The title that is displayed as column header. This attribute is used for titles that are derived from the title of a referred field.
name	Key	A unique identifier for this table element.
visibility	(Visibility Type)	Indicates the visibility of this table column. Valid values are: ▪ Required ▪ Visible ▪ Hidden
template	(Placeholder String)	The template string that is used as the basis for placeholder substitution. This attribute is required. The format for placeholders is **^** and an integer that represents the index of the argument in the arguments list.
arguments	Expression List	A semicolon-separated list of arguments.
style	Styles	The style that applies to this tag and its descendants.
ref	String List	Reference to the description specified in the <Define> block.

<Link>-tag

This refers to a table column possibly containing data-dependent links. A link can be a regular hyperlink (for example, a URL), a reference to an action that is available in the current pane, or a link to another workspace. The url-attribute, the action-attribute, and the workspace links are mutually exclusive. The following shows examples of these three kinds of link columns:

```
<Columns>

  <Link title="Approve time sheet" template="Approve"

        action="self.ApproveTimeSheet"/>

  <Link name="Link to Jobs workspace"  workspace="Jobs"/>

  <Link name="mailTo" title="Send a mail" template="Send
mail"

        url="'mailto:' + ElectronicEmailAddress"/>
```

Attribute Name	Type	Usage
name	Key	A unique identifier for this table element.
size	(Size Type)	Indicates the size, that is, width, of the column. Valid values are: ▪ Standard ▪ Tiny ▪ Small ▪ Medium ▪ Large ▪ Huge ▪ Vast
visibility	(Visibility Type)	Indicates the visibility of this table column. Valid values are: ▪ Required ▪ Visible ▪ Hidden
title	Display	The title that is displayed as column header. This attribute is used for static titles. Cannot be used in the context of <Override>.
titleValue	Key	The title that is displayed as column header. This attribute is used for titles that are derived from the value of a referred field. Cannot be used in the context of <Override>.

Attribute Name	Type	Usage
titleSource	Key	The title that is displayed as column header. This attribute is used for titles that are derived from the title of a referred field. Cannot be used in the context of <Override>.
action	Key	Indicates the action that should be invoked when this link is clicked. Use a qualified identifier such as **parent.delete**.
template	(Placeholder String)	The title of the link as rendered in an individual cell. You can insert placeholders that are substituted with the arguments from the arguments-attribute.
arguments	Expression List	A semicolon-separated list of arguments.
url	Expression (String)	The hyperlink that should be invoked when clicking the link. The process of opening the hyperlink is delegated to the operating system. For example, a www… link is opened by the operating system's default browser.
icon	Id	The resource identifier for an icon to be associated with this link column. This icon (if defined) is displayed in each cell in the column.
IconPosition	(Position Type)	Indicates whether the icon should be placed before or after the title. Valid values are: ▪ Prefix ▪ Postfix
style	Styles	The style that applies to this tag and its descendants.
workspace	Key	Indicates a workspace name that should be navigated when the link is clicked. Can be prefixed with a namespace qualifier.
workspaceTitle	Display	Indicates a workspace title for the workspace that is opened. This title overrides the default title for that workspace.
disabled	Boolean	Indicates whether the link is disabled.

<Waypoint>, <Target>, <Restriction>, <Focus>, and <Match> (in the Context of <Link>)

<Waypoint> and <Target>

If a link represents a link to a workspace, a workspace name and, if necessary, additional parameters such as waypoints and targets, should be specified. **<Waypoint>** is a pane in a path that is taken to navigate to the final **<Target>** pane in a link. Both **<Waypoint>** and **<Target>** represent an item in the link path and have the same attributes and elements.

Attribute Name	Type	Usage
pane	Key	Name of pane.

<Restriction>

You can set restriction on a waypoint or target pane and specify it as an expression.

Attribute Name	Type	Usage
title	Key	Restriction title (required).
expression	Expression (Boolean)	Restriction expression indicates a selection of records that should be displayed in a pane, similar to the filter selection.

<Focus>

The focus indicates on which records the focus should be set. It represents a list of match points.

<Match>

The match indicates a record that satisfies the field and value selection and is used to set a focus on that record.

Attribute Name	Type	Usage
field	Key	A field name.
value	Expression	Indicates a value of a field on which the focus should be set.

<Override> and <ElseOverride> (in the Context of <Link>)

You can define override rules that may override the attributes of a link in certain conditions. In tables, you can use this feature to disable a link or change its template based on the state of the individual row. In forms, it works similarly to include multiple links guarded by **<If>** and **<ElseIf>** conditionals.

```
<Link title="Link with multiple alternatives"
      action="self.ApproveTimeSheet"
      icon="approvetimesheet.png"
      template="Approve">
  <Override condition="Approved">
    <Link template="(approved)" icon="" disabled="true" />
  </Override>
</Link>
```

In the preceding example, a link is rendered with the text “approve” and an approve icon. If the record in a row is already approved, the following properties for the link are overridden:

- The text is changed.
- The icon is removed and disabled.
- The other properties are inherited from the outer link.

You can provide several overrides using the **<ElseOverride>** element. If multiple overrides are provided, the first override (if any) whose condition evaluates to **True** is applied.

Attribute Name	Type	Usage
condition	Expression	The condition that triggers the override.
default	Boolean	Default value if the condition fails to evaluate due to an error. Defaults to **False**.

<OrderBy>-tag

Table (and filter) columns can be initially sorted. The source of sorting should be the same as the source of the sorted field. Order type defines sorting direction and can be blank if the default value is used. The following is an example of predefined sorting:

```
<Columns>
  <OrderBy source="JobNumber" order="ascending" />
```

Attribute Name	Type	Usage
source	Key	The sorted field source (for example, a database field) of this table column.
order	(Order Type)	Indicates the sorting order. Valid values are: ▪ Ascending (default) ▪ Descending

<Override> and <ElseOverride> (in the Context of <OrderBy>)

You can define override rules that may override the attributes of an order by an element in certain conditions. You can use this feature to change a sorted field and/or its order type.

```
<OrderBy source="JobNumber" order="ascending">
  <Override condition="anyExpression">
    <OrderBy source="NumberOfSubJobs" order="descending" />
  </Override>
</OrderBy>
```

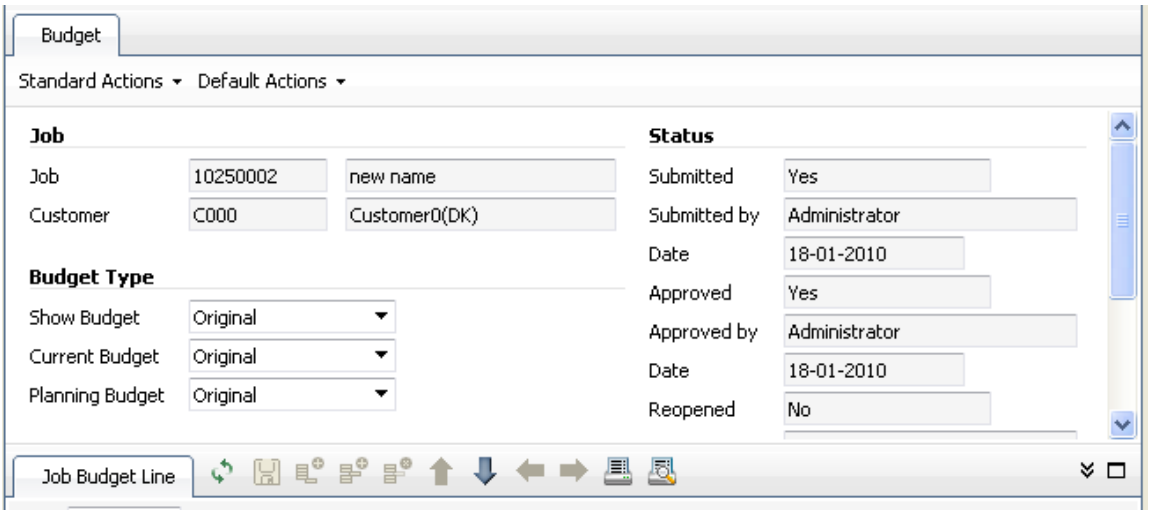
In the preceding example a **JobNumber** field is sorted with the order type **Ascending** if the **anyExpression** condition returns **False**. Otherwise, it is overridden by the inner **OrderBy** element, and the **NumberOfSubJobs** field is sorted with the order type **Descending**.

You can provide several overrides using the **<ElseOverride>** element. If you provide multiple overrides, the first override (if any) whose condition evaluates to **True** is applied.

Attribute Name	Type	Usage
condition	Expression	The condition that triggers the override.
default	Boolean	Default value if the condition fails to evaluate due to an error. Defaults to false.

Forms

The following is a screenshot of a form view as rendered in the client.



<Form>-tag

A form is a view to render the MWSL component card type. A form view can take parameters similar to a filter view.

Attribute Name	Type	Usage
name	Key	The name of this view. The name should uniquely identify this view within the scope of the containing MDML pane.
title	Display	The view title can be used to override the title that is defined on the pane. This does, however, not take effect if a title is defined on the corresponding MWSL component.
style	Styles	The style that applies to this tag and its descendants.
open	(Open Type) or Boolean Expression	The open state of this tag and all its descendants. Valid values are: ▪ False ▪ Never ▪ Create ▪ Update ▪ Always ▪ True In addition, you can specify any Boolean expression that can be evaluated in the relevant

Attribute Name	Type	Usage
		context. In this way, you can make the open state dependent on the context.
loginRules	Expression List	Indicates additional login rules that are applicable for this view.
frame	Boolean	Indicates whether the descendent elements of this container should, by default, be rendered as framed.

<Parameters>-tag

Any view can accept parameters from the corresponding MWSL specification. Use the <Parameters> tag to hold these parameter declarations. If a view with parameters is invoked from MWSL without arguments, an exception occurs.

Attribute Name	Type	Usage
-	-	-

<Parameter>-tag

Each view parameter has a mandatory name and type. The name is used to bind variables in the view during evaluation of conditionals and expressions. It should therefore be unique and not overlap any existing identifiers in the evaluation context.

Attribute Name	Type	Usage
name	Key	The name of the parameter. This name can be used in the conditionals and expressions of this view as a valid identifier. The name is required.
type	(Field Type)	The type of the parameter. Failure to comply with the type restriction imposed by this declaration causes an exception when invoking the view from an MWSL specification. Valid values are: ▪ Boolean ▪ Integer ▪ Real ▪ Amount ▪ String ▪ Date ▪ Time ▪ Enum

<Ruler>-tags

A ruler tag specifies which tab stops of the contained ruler should be exposed. The content of a ruler is dependent on the context in which the ruler occurs. All ruler-tags accept other ruler-tags as content (which inherit the context of the parent ruler). The nested ruler tags can be mixed with the same tag as can be accepted in the context. In other words, if, instead of the ruler, you insert **tag A**, then **tag A** can occur within the ruler.

Attribute Name	Type	Usage
encloseAlignment	Boolean	No tab stops, other than **start** and **end**, are exposed for this tag.
encloseLabelAlignment	Boolean	The **Label** tab stop is not exposed for this tag.
encloseIntervalAlignment	Boolean	The **IntervalStart** and **IntervalEnd** tab stops are not exposed for this tag.
equalizeSpans	(Span Pair Type)	Makes two sets of consecutive columns equal in size. Cannot be used together with **equalizeIntervalFields**.
fixLabelSize	Boolean	Specifies whether labels of elements inside the ruler should be allowed to shrink below their preferred size, that is, whether the label should always have exactly the size of its text, no matter how you shrink the pane. You implement this by setting the minimum width of all labels equal to the preferred size. Rules for inheritance apply; that is, specifications of **fixLabelSize** in inner tags overwrite this specification.

<Row>

This refers to a container for form members and elements. The contents of a row are laid out horizontally.

Attribute Name	Type	Usage
name	Key	The unique identifier of this tag.
open	(Open Type) or Boolean Expression	The open state of this tag and all its descendants. Valid values are: ▪ False ▪ Never ▪ Create ▪ Update ▪ Always

Attribute Name	Type	Usage
		▪ True In addition, you can specify any Boolean expression that can be evaluated in the relevant context. In this way, you can make the open state dependent on the context.
ruler	Key	Reference to a named ruler. This attribute is used to reuse ruler definitions.
style	Styles	The style that applies to this tag and its descendants.
frame	Boolean	Indicates whether the descendent elements of this container should, by default, be rendered as framed.

<Column>

This refers to a container for form members and elements. The contents of a column are laid out vertically.

Attribute Name	Type	Usage
name	Key	The unique identifier of this tag.
open	(Open Type) or Boolean Expression	The open state of this tag and all its descendants. Valid values are: ▪ False ▪ Never ▪ Create ▪ Update ▪ Always ▪ True In addition, you can specify any Boolean expression that can be evaluated in the relevant context. In this way, you can make the open state dependent on the context.
ruler	Key	A reference to a named ruler. This attribute is used to reuse ruler definitions.
style	Styles	The style that applies to this tag and its descendants.

Attribute Name	Type	Usage
frame	Boolean	Indicates whether the descendent elements of this container should, by default, be rendered as framed.

<Group>-tags (in the Context of a <Form>)

A group is a visual grouping of elements and can have a title. You cannot nest groups within groups; however, the contents of a group can be ordered in rows and columns.

Attribute Name	Type	Usage
name	Key	The unique identifier of this group.
title	Display	The title that is displayed above the group.
titleValue	Key	The title that is displayed above the group (based on a field's value).
open	(Open Type) or Boolean Expression	The open state of this tag and all its descendants. Valid values are: ▪ False ▪ Never ▪ Create ▪ Update ▪ Always ▪ True In addition, you can specify any Boolean expression that can be evaluated in the relevant context. In this way, you can make the open state dependent on the context.
ruler	Key	A reference to a named ruler. This attribute is used to reuse ruler definitions.
ref	Key	A reference to a group from a definitions block that is in scope. If this attribute is used, the contents of the referred group are inserted by means of macro expansion. The properties (for example, title) of the referred group are inherited unless explicitly overridden on the local tag.
style	Styles	The style that applies to this tag and its descendants.
frame	Boolean	Indicates whether the descendent elements of this container should, by default, be rendered as framed.

Forms (Elements)

A form element is the basic visual component of a form. Form elements are conceptually divided into one or more blocks. Each block can render either a title or a value. For example, a simple <Field> tag consists of a title- and a value-block, whereas a <Pair> tag consists of a title-, two value-, and one separator-block. The MDML layout engine attempts to align the blocks of different elements to provide an attractive user interface.

The appearance attribute of most form elements facilitates a certain degree of customization of the default alignment for an element. The standard value for appearance is **Normal**. If the element's value block should take up all available space and thereby preclude the title block from appearing, the appearance attribute should be set to **Maximized**.

Certain elements (<Field>, <ComboField>, <PeriodYear>, and <UnitField>) support a special appearance called **Interval**, which means that the element's value block is aligned with the first value block of an <Interval>. This setting only takes effect if there is an <Interval> in scope.

These four tags also support an appearance called **Unaligned**, which means that the element is not aligned with any other elements in its scope.

<Label>, <EmptyLabel>

A label is a single block that renders a title. The <EmptyLabel> is equivalent to a <Label>-tag with an empty string as title. Do not use **titleValue** or **titleSource** in the <EmptyLabel>-tag.

Attribute Name	Type	Usage
title	Display	Indicates the title. This attribute is used for static titles.
titleValue	Key	Indicates the source of the title. This attribute is used for titles that are derived from the value of a referred field.
titleSource	Key	Indicates the source of the title. This attribute is used for titles that are derived from the title of a referred field.
name	Key	The unique identifier of this tag.
style	Styles	The style that applies to this tag and its descendants.
appearance	(Element Appearance Type)	Indicates the appearance of this element.  See the explanation in the Forms (Elements) section. Valid values are: ▪ Standard ▪ Normal ▪ Maximized

<PairHeader>

A pair header is an element with three title blocks. These blocks align with the title- and two value-blocks of a normal <Pair>. The element is useful when a column with multiple pairs requires headings for its value parts. In other words, the pair header and a column of pairs can be thought of as a two-column grid.

Attribute Name	Type	Usage
title	Display	Indicates the title. This attribute is used for static titles.
titleValue	Key	Indicates the source of the title. This attribute is used for titles that are derived from the value of a referred field.
titleSource	Key	Indicates the source of the title. This attribute is used for titles that are derived from the title of a referred field.
firstTitle	Display	Indicates the title of the block aligned with the first value-block of a <Pair>. This attribute is used for static titles.
firstTitleValue	Key	Indicates the source of the title of the block aligned with the first value-block of a <Pair>. This attribute is used for titles derived from the value of a referred field.
firstTitleSource	Key	Indicates the source of the title of the block aligned with the first value-block of a <Pair>. This attribute is used for titles derived from the title of a referred field.
secondTitle	Display	Indicates the title of the block aligned with the second value-block of a <Pair>. This attribute is used for static titles.
secondTitleValue	Key	Indicates the source of the title of the block aligned with the second value-block of a <Pair>. This attribute is used for titles that are derived from the value of a referred field.
secondTitleSource	Key	Indicates the source of the title of the block aligned with the second value-block of a <Pair>. This attribute is used for titles that are derived from the title of a referred field.
name	Key	The unique identifier of this tag.
style	Styles	The style that applies to this tag and its descendants.
appearance	(Element Appearance Type)	Indicates the appearance of this element. See the explanation in the Forms (Elements) section. Valid values are: ▪ Standard

Attribute Name	Type	Usage
		▪ Normal ▪ Maximized

<Field> (in the Context of <Form>)

A field is an element with a title- and a value-block. It is used to render a single source.

Attribute Name	Type	Usage
source	Id	The source of the value-block for this element (for example, a database field).
shadowTitle	Display	The title that is displayed in the value-block when no value is visible. Use Shadow titles as hints for data entry.
title	Display	Indicates the title. This attribute is used for static titles.
titleValue	Key	Indicates the source of the title. This attribute is used for titles that are derived from the value of a referred field.
titleSource	Key	Indicates the source of the title. This attribute is used for titles that are derived from the title of a referred field.
name	Key	The unique identifier of this tag.
mandatory	Boolean Expression	Indicates whether this element is mandatory, that is, whether you must supply a value before submitting.
open	(Open Type) or Boolean Expression	The open state of this tag and all its descendants. Valid values are: ▪ False ▪ Never ▪ Create ▪ Update ▪ Always ▪ True In addition, you can specify any Boolean expression that can be evaluated in the relevant context. In this way, you can make the open state dependent on the context.
size	(Size Type)	The size hint that should be applied to the element. The size hints affects the width of the entire element. Valid values are: ▪ Standard ▪ Tiny

Attribute Name	Type	Usage
		▪ Small ▪ Medium ▪ Large ▪ Huge ▪ Vast
autoSubmit	Boolean	Indicates whether the value that you enter into this element should be automatically submitted when the element loses focus.
searchLayout	Key	The layout that is used if this element launches a foreign key (CTRL+G) search. If nothing is specified, a default search layout (for example, **Find_X.mdml.xml**) is retrieved from the server.
searchView	Key	The layout view that is used if this element launches a foreign key (CTRL+G) search. The view should be part of the layout specified in the searchLayout-attribute. If nothing is specified, the first view from the search layout is chosen.
searchOption	Key	Indicates the name of the search option to be applied when doing “lightweight” searches from within the field. The name must match the name of the filter layout/view implicitly or explicitly stated.
suggestions	Suggestions	Indicates the suggestion strategy for fields based on a foreign key. Valid values are: ▪ Standard ▪ None ▪ Automatic ▪ onDemand
type	(Data Type)	Defines the type to which this value should be rendered. For example, a field of type **timeDuration** can be rendered as **real**.
style	Styles	The style that applies to this tag and its descendants.
appearance	(Element Appearance Type)	Indicates the appearance of this element. See the explanation in the Forms (Elements) section. Valid values are: ▪ Standard

Attribute Name	Type	Usage
		▪ Normal ▪ Maximized ▪ Unaligned ▪ Interval
frame	Boolean	Indicates whether the field should be rendered with a frame. If **False**, the field is rendered without a frame (as a label). Defaults to **True** unless the frame attribute is specified in one of the containing elements, in which case the value of the closest ancestor is used as the default value.

<Description> (in the Context of <Form>)

A description is an element that is used to render labels based on expressions or fields in the current record. The description should be made data-dependent by using placeholder substitution. The template-attribute is used as a template, and each placeholder (for example, ^1, ^2…^n) is substituted by the current value of the argument on evaluation time. You can also specify the description specified in the <Define>-block and referenced by the ref attribute.

Attribute Name	Type	Usage
title	Display	Indicates the title. This attribute is used for static titles.
titleValue	Key	Indicates the source of the title. This attribute is used for titles that are derived from the value of a referred field.
titleSource	Key	Indicates the source of the title. This attribute is used for titles that are derived from the title of a referred field.
name	Key	The unique identifier of this tag.
size	(Size Type)	The size hint that should be applied to the element. The size hints affect the width of the entire element. Valid values are: ▪ Standard ▪ Tiny ▪ Small ▪ Medium ▪ Large ▪ Huge ▪ Vast
template	(Placeholder String)	The template string used as the basis for placeholder substitution. This attribute is required. The format for

Attribute Name	Type	Usage
		placeholders is ^ and an integer representing the index of the argument in the arguments list.
arguments	Expression List	A semicolon-separated list of arguments.
style	Styles	The style that applies to this tag and its descendants.
appearance	(Element Appearance Type)	Indicates the appearance of this element.  See the explanation in the Forms (Elements) section. Valid values are: ▪ Standard ▪ Maximized
ref	String List	A reference to the description specified in the <Define>-block.

<ZipCity>

This element is used to render a title- and two value-blocks representing a zip code and a city name. It automatically fits the space corresponding to a single field of a specific size. Therefore, the zip and city fields are fitted to the size that is in scope. For example, if the size is **Tiny**, they share very little space, and if the size is **Huge** they adapt to the larger size.

Attribute Name	Type	Usage
title	Display	Indicates the title. This attribute is used for static titles.
titleValue	Key	Indicates the source of the title. This attribute is used for titles that are derived from the value of a referred field.
titleSource	Key	Indicates the source of the title. This attribute is used for titles that are derived from the title of a referred field.
name	Key	The unique identifier of this tag.
zipSource, citySource	Key	Indicates the source of the value (for example, a database field).
zipShadowTitle, cityShadowTitle	Display	The title that is displayed in the value-block when no value is visible. Use Shadow titles as hints for data entry.
zipMandatory, cityMandatory	Boolean Expression	Indicates whether this element is mandatory, that is, whether you must supply a value before submitting.

Attribute Name	Type	Usage
zipOpen, cityOpen	(Open Type) or Boolean Expression	The open state of this tag and all its descendants. Valid values are: ▪ False ▪ Never ▪ Create ▪ Update ▪ Always ▪ True In addition, you can specify any Boolean expression that can be evaluated in the relevant context. In this way, you can make the open state dependent on the context.
zipSearchLayout, citySearchLayout	Key	The layout that is used if this element launches a foreign key (CTRL+G) search. If nothing is specified, a default search layout (for example, **Find_X.mdml.xml**) is retrieved from the server.
zipSearchView, citySearchView	Key	The layout view that is used if this element launches a foreign key (CTRL+G) search. The view should be part of the layout specified in the searchLayout-attribute. If nothing is specified, the first view from the search layout is chosen.
zipSearchOption, citySearchOption	Key	Indicates the name of the search option to be applied when doing “lightweight” searches from within the field. The name must match the name of the filter layout/view implicitly or explicitly stated.
zipSuggestions, citySuggestions	Suggestions	Indicates the suggestion strategy for fields based on a foreign key. Valid values are: ▪ Standard ▪ None ▪ Automatic ▪ onDemand
autoSubmit	Boolean	Indicates whether the value that you enter into this element should be automatically submitted when the element loses focus.
style	Styles	The style that applies to this tag and its descendants.
appearance	(Element Appearance Type)	Indicates the appearance of this element. See the explanation in the Forms (Elements) section.

Attribute Name	Type	Usage
		Valid values are: ▪ Standard ▪ Normal ▪ Maximized ▪ Unaligned
frame	Boolean	Indicates whether the field should be rendered with a frame. If **False**, the field is rendered without a frame (as a label). Defaults to **True** unless the frame attribute is specified in one of the containing elements, in which case the value of the closest ancestor is used as the default value.

<ComboField>

A combo field represents the combination of two value-blocks that are rendered using a single value. Only the first value-block is editable. Use this element for value-combinations that are not handled by the <UnitField> (in the Context of a <Form>).

Attribute Name	Type	Usage
title	Display	Indicates the title. This attribute is used for static titles.
titleValue	Key	Indicates the source of the title. This attribute is used for titles that are derived from the value of a referred field.
titleSource	Key	Indicates the source of the title. This attribute is used for titles that are derived from the title of a referred field.
name	Key	The unique identifier of this tag.
firstSource, secondSource	Key	Indicates the source of the value (for example, a database field).
firstShadowTitle, secondShadowTitle	Display	The title that is displayed in the value-block when no value is visible. Use Shadow titles as hints for data entry.
firstMandatory, secondMandatory	Boolean Expression	Indicates whether this element is mandatory, that is, whether you must supply a value before submitting.
firstOpen, secondOpen	(Open Type) or Boolean Expression	The open state of this tag and all its descendants. Valid values are: ▪ False ▪ Never ▪ Create

Attribute Name	Type	Usage
		▪ Update ▪ Always ▪ True In addition, you can specify any Boolean expression that can be evaluated in the relevant context. In this way, you can make the open state dependent on the context.
firstSize, secondSize	(Size Type)	The size hint that should be applied to the element. The size hints affect the width of the entire element. Valid values are: ▪ Standard ▪ Tiny ▪ Small ▪ Medium ▪ Large ▪ Huge ▪ Vast
firstSearchLayout, secondSearchLayout	Key	The layout that is used if this element launches a foreign key (CTRL+G) search. If nothing is specified, a default search layout (for example, **Find_X.mdml.xml**) is retrieved from the server.
firstSearchView, secondSearchView	Key	The layout view that is used if this element launches a foreign key (CTRL+G) search. The view should be part of the layout specified in the searchLayout-attribute. If nothing is specified, the first view from the search layout is chosen.
firstSearchOption, secondSearchOption	Key	Indicates the name of the search option to be applied when doing “lightweight” searches from within the field. The name must match the name of the filter layout/view implicitly or explicitly stated.
firstSuggestions, secondSuggestions	Suggestions	Indicates the suggestion strategy for fields based on a foreign key. Valid values are: ▪ Standard ▪ None ▪ Automatic ▪ onDemand
autoSubmit	Boolean	Indicates whether the value that you enter into this element should be automatically submitted when the element loses focus.

Attribute Name	Type	Usage
firstType, secondType	(Data Type)	Defines the type to which this value should be rendered. For example, a field of type **timeDuration** can be rendered as **real**.
style	Styles	The style that applies to this tag and its descendants.
appearance	(Element Appearance Type)	Indicates the appearance of this element.  See the explanation in the Forms (Elements) section. Valid values are: ▪ Standard ▪ Normal ▪ Maximized ▪ Unaligned ▪ Interval
frame	Boolean	Indicates whether the field should be rendered with a frame. If **False**, the field is rendered without a frame (as a label). Defaults to **True** unless the frame attribute is specified in one of the containing elements, in which case the value of the closest ancestor is used as the default value.

<PeriodYear>

This element is used to render a title- and two value-blocks representing a period and a year.

Attribute Name	Type	Usage
title	Display	Indicates the title. This attribute is used for static titles.
titleValue	Key	Indicates the source of the title. This attribute is used for titles that are derived from the value of a referred field.
titleSource	Key	Indicates the source of the title. This attribute is used for titles that are derived from the title of a referred field.
name	Key	The unique identifier of this tag.
periodSource, yearSource	Key	Indicates the source of the value (for example, a database field).
periodShadowTitle, yearShadowTitle	Display	The title that is displayed in the value-block when no value is visible. Use Shadow titles as hints for data entry.

Attribute Name	Type	Usage
periodMandatory, yearMandatory	Boolean Expression	Indicates whether this element is mandatory, that is, whether you must supply a value before submitting.
periodOpen, yearOpen	(Open Type) or Boolean Expression	The open state of this tag and all of its descendants. Valid values are: ▪ False ▪ Never ▪ Create ▪ Update ▪ Always ▪ True In addition, you can specify any Boolean expression that can be evaluated in the relevant context. In this way, you can make the open state dependent on the context.
periodSize, yearSize	(Size Type)	The size hint that should be applied to the element. The size hints affect the width of the entire element. Valid values are: ▪ Standard ▪ Tiny ▪ Small ▪ Medium ▪ Large ▪ Huge ▪ Vast
autoSubmit	Boolean	Indicates whether the value that you enter into this element should be automatically submitted when the element loses focus.
style	Styles	The style that applies to this tag and its descendants.

Attribute Name	Type	Usage
appearance	(Element Appearance Type)	Indicates the appearance of this element.  See the explanation in the Forms (Elements) section. Valid values are: ▪ Standard ▪ Normal ▪ Maximized ▪ Unaligned ▪ Interval
frame	Boolean	Indicates whether the field should be rendered with a frame. If **False**, the field is rendered without a frame (as a label). Defaults to **True** unless the frame attribute is specified in one of the containing elements, in which case the value of the closest ancestor is used as the default value.

<UnitField> (in the Context of a <Form>)

Use this element to render a value with a unit. The unit can either be retrieved from a source (for example, a database field), or specified as a static title. The value and unit are visually rendered by a single widget, which makes the value-blocks of the <UnitField>-tag appear as a single block.

Attribute Name	Type	Usage
title	Display	Indicates the title. This attribute is used for static titles.
titleValue	Key	Indicates the source of the title. This attribute is used for titles that are derived from the value of a referred field.
titleSource	Key	Indicates the source of the title. This attribute is used for titles that are derived from the title of a referred field.
name	Key	The unique identifier of this tag.
source, unitSource	Key	Indicates the source of the value (for example, a database field).
shadowTitle, unitShadowTitle	Display	The title that is displayed in the value-block when no value is visible. Use Shadow titles as hints for data entry.
valueMandatory, unitMandatory	Boolean Expression	Indicates whether this element is mandatory, that is, whether you must supply a value before submitting.

Attribute Name	Type	Usage
valueOpen, unitOpen	(Open Type) or Boolean Expression	The open state of this tag and all its descendants. Valid values are: ▪ False ▪ Never ▪ Create ▪ Update ▪ Always ▪ True In addition, you can specify any Boolean expression that can be evaluated in the relevant context. In this way, you can make the open state dependent on the context.
valueSize, unitSize	(Size Type)	The size hint that should be applied to the element. The size hints affect the width of the entire element. Valid values are: ▪ Standard ▪ Tiny ▪ Small ▪ Medium ▪ Large ▪ Huge ▪ Vast
unitTitle	Display	Indicates the unit in the form of a static title. This is used, for example, for units such as **%**.
unitPosition	(Unit Position Type)	Indicates the position of the unit in relation to the value. It is possible to pre- or postfix the unit, or alternatively rely on the currency setting of the operating system to determine the position. Valid values are: ▪ Standard ▪ Prefix ▪ Postfix ▪ Currency
autoSubmit	Boolean	Indicates whether the value that you enter into this element should be automatically submitted when the element loses focus.
type	(Data Type)	Defines the type to which this value should be rendered. For example, a field of type **timeDuration** can be rendered as **real**.

Attribute Name	Type	Usage
style	Styles	The style that applies to this tag and its descendants.
appearance	(Element Appearance Type)	Indicates the appearance of this element.  See the explanation in the Forms (Elements) section. Valid values are: ▪ Standard ▪ Normal ▪ Maximized ▪ Unaligned ▪ Interval
frame	Boolean	Indicates whether the field should be rendered with a frame. If **False**, the field is rendered without a frame (as a label). Defaults to **True** unless the frame attribute is specified in one of the containing elements, in which case the value of the closest ancestor is used as the default value.

<Pair>

A pair of values can be represented by the <Pair>-tag, which consists of a title- and two value-blocks. If several pairs are placed in a column, the <PairHeader>-tag provides a nice way of supplying the additional titles.

Attribute Name	Type	Usage
title	Display	Indicates the title. This attribute is used for static titles.
titleValue	Key	Indicates the source of the title. This attribute is used for titles that are derived from the value of a referred field.
titleSource	Key	Indicates the source of the title. This attribute is used for titles that are derived from the title of a referred field.
name	Key	The unique identifier of this tag.
firstSource, secondSource	Key	Indicates the source of the value (for example, a database field).
firstShadowTitle, secondShadowTitle	Display	The title that is displayed in the value-block when no value is visible. Use Shadow titles as hints for data entry.

Attribute Name	Type	Usage
firstMandatory, secondMandatory	Boolean Expression	Indicates whether this element is mandatory, that is., whether you must supply a value before submitting.
firstOpen, secondOpen	(Open Type) or Boolean Expression	The open state of this tag and all its descendants. Valid values are: ▪ False ▪ Never ▪ Create ▪ Update ▪ Always ▪ True In addition, you can specify any Boolean expression that can be evaluated in the relevant context. In this way, you can make the open state dependent on the context.
firstSize, secondSize	(Size Type)	The size hint that should be applied to the element. The size hints affect the width of the entire element. Valid values are: ▪ Standard ▪ Tiny ▪ Small ▪ Medium ▪ Large ▪ Huge ▪ Vast
firstSearchLayout, secondSearchLayout	Key	The layout that is used if this element launches a foreign key (CTRL+G) search. If nothing is specified, a default search layout (for example, **Find_X.mdml.xml**) is retrieved from the server.
firstSearchView, secondSearchView	Key	The layout view that is used if this element launches a foreign key (CTRL+G) search. The view should be part of the layout specified in the searchLayout-attribute. If nothing is specified, the first view from the search layout is chosen.
firstSearchOption, secondSearchOption	Key	Indicates the name of the search option to be applied when doing “lightweight” searches from within the field. The name must match the name of the filter layout/view implicitly or explicitly stated.
firstSuggestions, secondSuggestions	Suggestions	Indicates the suggestion strategy for fields based on a foreign key. Valid values are:

Attribute Name	Type	Usage
		▪ Standard ▪ None ▪ Automatic ▪ onDemand
autoSubmit	Boolean	Indicates whether the value that you enter into this element should be automatically submitted when the element loses focus.
firstType, secondType	(Data Type)	Defines the type to which this value should be rendered. For example, a field of type **timeDuration** can be rendered as **real**.
style	Styles	The style that applies to this tag and its descendants.
appearance	(Element Appearance Type)	Indicates the appearance of this element.  See the explanation in the forms (Elements) section. Valid values are: ▪ Standard ▪ Normal ▪ Maximized
frame	Boolean	Indicates whether the field should be rendered with a frame. If **False**, the field is rendered without a frame (as a label). Defaults to **True** unless the frame attribute is specified in one of the containing elements, in which case the value of the closest ancestor is used as the default value.

<Reference>

The <Reference> tag is a variant of the <Pair> tag. Use it to render a key and a description from a foreign key. If multiple keys and descriptions are defined on a referred foreign key, an arbitrary key and description are chosen. You can override both the key- and description-source attributes. This effectively uses the reference as a pair.

Attribute Name	Type	Usage
foreignKey	Key	The name of the foreign key to which this element refers.
title	Display	Indicates the title. This attribute is used for static titles.

Attribute Name	Type	Usage
titleValue	Key	Indicates the source of the title. This attribute is used for titles that are derived from the value of a referred field.
titleSource	Key	Indicates the source of the title. This attribute is used for titles that are derived from the title of a referred field.
name	Key	The unique identifier of this tag.
keySource, descriptionSource	Key	Indicates the source of the value (for example, a database field).
keyShadowTitle, descriptionShadowTitle	Display	The title that is displayed in the value-block when no value is visible. Use Shadow titles as hints for data entry.
keyMandatory, descriptionMandatory	Boolean Expression	Indicates whether this element is mandatory, that is, whether you must supply a value before submitting.
keyOpen, descriptionOpen	(Open Type) or Boolean Expression	The open state of this tag and all its descendants. Valid values are: ▪ False ▪ Never ▪ Create ▪ Update ▪ Always ▪ True In addition, you can specify any Boolean expression that can be evaluated in the relevant context. In this way, you can make the open state dependent on the context.
keySize, descriptionSize	(Size Type)	The size hint that should be applied to the element. The size hints affect the width of the entire element. Valid values are: ▪ Standard ▪ Tiny ▪ Small ▪ Medium ▪ Large ▪ Huge ▪ Vast
keySearchLayout, descriptionSearchLayout	Key	The layout that is used if this element launches a foreign key (CTRL+G) search. If nothing is specified, a default

Attribute Name	Type	Usage
		search layout (for example, **Find_X.mdml.xml**) is retrieved from the server.
keySearchView, descriptionSearchView	Key	The layout view that is used if this element launches a foreign key (CTRL+G) search. The view should be part of the layout specified in the searchLayout-attribute. If nothing is specified, the first view from the search layout is chosen.
keySearchOption, descriptionSearchOption	Key	Indicates the name of the search option to be applied when doing “lightweight” searches from within the field. The name must match the name of the filter layout/view implicitly or explicitly stated.
keySuggestions, descriptionSuggestions	Suggestions	Indicates the suggestion strategy for fields based on a foreign key. Valid values are: ▪ Standard ▪ None ▪ Automatic ▪ onDemand
autoSubmit	Boolean	Indicates whether the value that you enter into this element should be automatically submitted when the element loses focus.
style	Styles	The style that applies to this tag and its descendants.
appearance	(Element Appearance Type)	Indicates the appearance of this element. See the explanation in the Forms (Elements) section. Valid values are: ▪ Standard ▪ Normal ▪ Maximized ▪ Unaligned ▪ Interval
frame	Boolean	Indicates whether the field should be rendered with a frame. If **False**, the field is rendered without a frame (as a label). Defaults to **True** unless the frame attribute is specified in one of the containing elements, in which case the value of the closest ancestor is used as the default value.

<Interval>

An interval represents a relationship between two values. The element is rendered as a title- and two value-blocks. The value-blocks are separated by an optional separator.

The alignment of interval is different from “for instance pairs” since the alignment is made using the separator rather than the beginning of the value-blocks.

Attribute Name	Type	Usage
title	Display	Indicates the title. This attribute is used for static titles.
titleValue	Key	Indicates the source of the title. This attribute is used for titles that are derived from the value of a referred field.
titleSource	Key	Indicates the source of the title. This attribute is used for titles that are derived from the title of a referred field.
name	Key	The unique identifier of this tag.
firstSource, lastSource	Key	Indicates the source of the value (for example, a database field).
firstShadowTitle, lastShadowTitle	Display	The title that is displayed in the value-block when no value is visible. Use Shadow titles as hints for data entry.
firstMandatory, lastMandatory	Boolean Expression	Indicates whether this element is mandatory, that is, whether you must supply a value before submitting.
firstOpen, lastOpen	(Open Type) or Boolean Expression	The open state of this tag and all its descendants. Valid values are: ▪ False ▪ Never ▪ Create ▪ Update ▪ Always ▪ True In addition, you can specify any Boolean expression that can be evaluated in the relevant context. In this way, you can make the open state dependent on the context.
firstSize, lastSize	(Size Type)	The size hint that should be applied to the element. The size hints affect the width of the entire element. Valid values are: ▪ Standard ▪ Tiny ▪ Small ▪ Medium

Attribute Name	Type	Usage
		▪ Large ▪ Huge ▪ Vast
firstSearchLayout, lastSearchLayout	Key	The layout that is used if this element launches a foreign key (CTRL+G) search. If nothing is specified, a default search layout (for example, **Find_X.mdml.xml**) is retrieved from the server.
firstSearchView, lastSearchView	Key	The layout view that is used if this element launches a foreign key (CTRL+G) search. The view should be part of the layout specified in the searchLayout-attribute. If nothing is specified, the first view from the search layout is chosen.
firstSearchOption, lastSearchOption	Key	Indicates the name of the search option to be applied when doing “lightweight” searches from within the field. The name must match the name of the filter layout/view implicitly or explicitly stated.
firstSuggestions, lastSuggestions	Suggestions	Indicates the suggestion strategy for fields based on a foreign key. Valid values are: ▪ Standard ▪ None ▪ Automatic ▪ onDemand
separator	Display	Indicates the separator that should be displayed between the two value-blocks.
autoSubmit	Boolean	Indicates whether the value that you enter into this element should be automatically submitted when the element loses focus.
firstType, lastType	(Data Type)	Defines the type to which this value should be rendered. For example, a field of type **timeDuration** can be rendered as **real**.
style	Styles	The style that applies to this tag and its descendants.
appearance	(Element Appearance Type)	Indicates the appearance of this element. See the explanation in the Forms (Elements) section. Valid values are: ▪ Standard ▪ Normal

Attribute Name	Type	Usage
		▪ Maximized
frame	Boolean	Indicates whether the field should be rendered with a frame. If **False**, the field is rendered without a frame (as a label). Defaults to **True** unless the frame attribute is specified in one of the containing elements, in which case the value of the closest ancestor is used as the default value.

<Range>

A range is similar to an <Interval) in the sense that it represents the relation between two values. The distinguishing characteristic of a range is that it does not align on the separator as an interval does.

Attribute Name	Type	Usage
title	Display	Indicates the title. This attribute is used for static titles.
titleValue	Key	Indicates the source of the title. This attribute is used for titles that are derived from the value of a referred field.
titleSource	Key	Indicates the source of the title. This attribute is used for titles that are derived from the title of a referred field.
name	Key	The unique identifier of this tag.
firstSource, lastSource	Key	Indicates the source of the value (for example, a database field).
firstShadowTitle, lastShadowTitle	Display	The title that is displayed in the value-block when no value is visible. Use Shadow titles as hints for data entry.
firstMandatory, lastMandatory	Boolean Expression	Indicates whether this element is mandatory, that is, whether you must supply a value before submitting.
firstOpen, lastOpen	(Open Type) or Boolean Expression	The open state of this tag and all its descendants. Valid values are: ▪ False ▪ Never ▪ Create ▪ Update ▪ Always ▪ True In addition, you can specify any Boolean expression that can be evaluated in the relevant context. In this way, you can make the open state dependent on the context.

Attribute Name	Type	Usage
firstSize, lastSize	(Size Type)	The size hint that should be applied to the element. The size hints affect the width of the entire element. Valid values are: ▪ Standard ▪ Tiny ▪ Small ▪ Medium ▪ Large ▪ Huge ▪ Vast
firstSearchLayout, lastSearchLayout	Key	The layout that is used if this element launches a foreign key (CTRL+G) search. If nothing is specified, a default search layout (for example, **Find_X.mdml.xml**) is retrieved from the server.
firstSearchView, lastSearchView	Key	The layout view that is used if this element launches a foreign key (CTRL+G) search. The view should be part of the layout specified in the searchLayout-attribute. If nothing is specified, the first view from the search layout is chosen.
firstSearchOption, lastSearchOption	Key	Indicates the name of the search option to be applied when doing “lightweight” searches from within the field. The name must match the name of the filter layout/view implicitly or explicitly stated.
firstSuggestions, lastSuggestions	Suggestions	Indicates the suggestion strategy for fields based on a foreign key. Valid values are: ▪ Standard ▪ None ▪ Automatic ▪ onDemand
separator	Display	Indicates the separator that should be displayed between the two value-blocks.
autoSubmit	Boolean	Indicates whether the value that you enter into this element should be automatically submitted when the element loses focus.
firstType, lastType	(Data Type)	Defines the type to which this value should be rendered. For example, a field of type **timeDuration** can be rendered as **real**.
style	Styles	The style that applies to this tag and its descendants.

Attribute Name	Type	Usage
appearance	(Element Appearance Type)	Indicates the appearance of this element.  See the explanation in the forms (Elements) section. Valid values are: ▪ Standard ▪ Normal ▪ Maximized
frame	Boolean	Indicates whether the field should be rendered with a frame. If **False**, the field is rendered without a frame (as a label). Defaults to **True** unless the frame attribute is specified in one of the containing elements, in which case the value of the closest ancestor is used as the default value.

<UnitFieldInterval>

This element represents the relationship between two values with units.

 See <UnitField> (in the Context of a <Form>) for a description of values with units.

The alignment of this element follows the convention from the <Interval> tag.

Attribute Name	Type	Usage
title	Display	Indicates the title. This attribute is used for static titles.
titleValue	Key	Indicates the source of the title. This attribute is used for titles that are derived from the value of a referred field.
titleSource	Key	Indicates the source of the title. This attribute is used for titles that are derived from the title of a referred field.
name	Key	The unique identifier of this tag.
firstSource, firstUnitSource, lastSource, lastUnitSource	Key	Indicates the source of the value (for example, a database field).

Attribute Name	Type	Usage
firstShadowTitle, firstUnitShadowTitle, lastShadowTitle, lastUnitShadowTitle	Display	The title that is displayed in the value-block when no value is visible. Use Shadow titles as hints for data entry.
firstValueMandatory, firstUnitMandatory, lastValueMandatory, lastUnitMandatory	Boolean Expression	Indicates whether this element is mandatory, that is, whether you must supply a value before submitting.
firstValueopen, firstUnitOpen, lastValueOpen, lastUnitOpen	(Open Type) or Boolean Expression	The open state of this tag and all its descendants. Valid values are: ▪ False ▪ Never ▪ Create ▪ Update ▪ Always ▪ True In addition, you can specify any Boolean expression that can be evaluated in the relevant context. In this way, you can make the open state dependent on the context.
firstValueSize, firstUnitSize, lastValueSize, lastUnitSize	(Size Type)	The size hint that should be applied to the element. The size hints affect the width of the entire element. Valid values are: ▪ Standard ▪ Tiny ▪ Small ▪ Medium ▪ Large ▪ Huge ▪ Vast
firstUnitTitle, lastUnitTitle	Display	Indicates the unit in the form of a static title. This is used, for example, for units such as **%**.
unitPosition	(Unit Position Type)	Indicates the position of the unit in relation to the value. You can explicitly pre- or postfix the unit, or alternatively rely on the currency setting of the operating system to determine the position. Valid values are:

Attribute Name	Type	Usage
		▪ Standard ▪ Prefix ▪ Postfix ▪ Currency
separator	Display	Indicates the separator that should be displayed between the two value-blocks.
autoSubmit	Boolean	Indicates whether the value that you enter into this element should be automatically submitted when the element loses focus.
style	Styles	The style that applies to this tag and its descendants.
appearance	(Element Appearance Type)	Indicates the appearance of this element.  See the explanation in the Forms (Elements) section. Valid values are: ▪ Standard ▪ Normal ▪ Maximized
frame	Boolean	Indicates whether the field should be rendered with a frame. If **False**, the field is rendered without a frame (as a label). Defaults to **True** unless the frame attribute is specified in one of the containing elements, in which case the value of the closest ancestor is used as the default value.

<PeriodYearInterval>

This element represents the relationship between two period year instances. The element has one title- and four value-blocks. The first two and the last two value-blocks are separated by an end-points-separator. The actual interval, that is, the space between value-blocks two and three, is separated by the separator specified in the separator-attribute. The alignment of this element follows the convention from the <Interval> tag.

Attribute Name	Type	Usage
title	Display	Indicates the title. This attribute is used for static titles.
titleValue	Key	Indicates the source of the title. This attribute is used for titles that are derived from the value of a referred field.

Attribute Name	Type	Usage
titleSource	Key	Indicates the source of the title. This attribute is used for titles that are derived from the title of a referred field.
name	Key	The unique identifier of this tag.
firstPeriodSource, firstYearSource, lastPeriodSource, lastYearSource	Key	Indicates the source of the value, for example a database field.
firstPeriodShadowTitle , firstYearShadowTitle, lastPeriodShadowTitle, lastYearShadowTitle	Display	The title that is displayed in the value-block when no value is visible. Use Shadow titles as hints for data entry.
firstPeriodMandatory, firstYearMandatory, lastPeriodMandatory, lastYearMandatory	Boolean Expression	Indicates whether this element is mandatory, that is, whether you must supply a value before submitting.
firstPeriodOpen, firstYearOpen, lastPeriodOpen, lastYearOpen	(Open Type) or Boolean Expression	The open state of this tag and all its descendants. Valid values are: ▪ False ▪ Never ▪ Create ▪ Update ▪ Always ▪ True In addition, you can specify any Boolean expression that can be evaluated in the relevant context. In this way, you can make the open state dependent on the context.
firstPeriodSize, firstYearSize, lastPeriodSize, lastYearSize	(Size Type)	The size hint that should be applied to the element. The size hints affect the width of the entire element. Valid values are: ▪ Standard ▪ Tiny ▪ Small ▪ Medium ▪ Large

Attribute Name	Type	Usage
		▪ Huge ▪ Vast
separator	Display	Indicates the separator that should be displayed between the period year combinations.
endPointsSeparator	Display	Indicates the separator that should be displayed between each period and year.
autoSubmit	Boolean	Indicates whether the value that you enter into this element should be automatically submitted when the element loses focus.
style	Styles	The style that applies to this tag and its descendants.
appearance	(Element Appearance Type)	Indicates the appearance of this element.  See the explanation in the Forms (Elements) section. Valid values are: ▪ Standard ▪ Normal ▪ Maximized
frame	Boolean	Indicates whether the field should be rendered with a frame. If **False**, the field is rendered without a frame (as a label). Defaults to **True** unless the frame attribute is specified in one of the containing elements, in which case the value of the closest ancestor is used as the default value.

<PeriodYearRange>

This element represents the relationship between two period year instances. The element has one title- and four value-blocks. The first two and the last two value-blocks are separated by an end-points-separator. The actual interval, that is, the space between value-blocks two and three, is separated by the separator specified in the separator-attribute. The alignment of this element follows the convention from the <Interval> tag.

Attribute Name	Type	Usage
title	Display	Indicates the title. This attribute is used for static titles.
titleValue	Key	Indicates the source of the title. This attribute is used for titles that are derived from the value of a referred field.

Attribute Name	Type	Usage
titleSource	Key	Indicates the source of the title. This attribute is used for titles that are derived from the title of a referred field.
name	Key	The unique identifier of this tag.
firstPeriodSource, firstYearSource, lastPeriodSource, lastYearSource	Key	Indicates the source of the value (for example, a database field).
firstPeriodShadowTitle, firstYearShadowTitle, lastPeriodShadowTitle, lastYearShadowTitle	Display	The title that is displayed in the value-block when no value is visible. Use Shadow titles as hints for data entry.
firstPeriodMandatory, firstYearMandatory, lastPeriodMandatory, lastYearMandatory	Boolean Expression	Indicates whether this element is mandatory, that is, whether you must supply a value before submitting.
firstPeriodOpen, firstYearOpen, lastPeriodOpen, lastYearOpen	(Open Type) or Boolean Expression	The open state of this tag and all its descendants. Valid values are: ▪ False ▪ Never ▪ Create ▪ Update ▪ Always ▪ True In addition, you can specify any Boolean expression that can be evaluated in the relevant context. In this way, you can make the open state dependent on the context.
firstPeriodSize, firstYearSize, lastPeriodSize, lastYearSize	(Size Type)	The size hint that should be applied to the element. The size hints affect the width of the entire element. Valid values are: ▪ Standard ▪ Tiny ▪ Small ▪ Medium ▪ Large

Attribute Name	Type	Usage
		▪ Huge ▪ Vast
separator	Display	Indicates the separator that should be displayed between the period year combinations.
endPointsSeparator	Display	Indicates the separator that should be displayed between each period and year.
autoSubmit	Boolean	Indicates whether the value that you enter into this element should be automatically submitted when the element loses focus.
style	Styles	The style that applies to this tag and its descendants.
appearance	(Element Appearance Type)	Indicates the appearance of this element.  See the explanation in the Forms (Elements) section. Valid values are: ▪ Standard ▪ Normal ▪ Maximized
frame	Boolean	Indicates whether the field should be rendered with a frame. If **False**, the field is rendered without a frame (as a label). Defaults to **True** unless the frame attribute is specified in one of the containing elements, in which case the value of the closest ancestor is used as the default value.

<ComboFieldInterval>

This element represents the relationship between two combo fields, that is, as defined by the <ComboField> tag. The alignment of this element follows the convention from the <Interval> tag.

Attribute Name	Type	Usage
title	Display	Indicates the title. This attribute is used for static titles.
titleValue	Key	Indicates the source of the title. This attribute is used for titles that are derived from the value of a referred field.

Attribute Name	Type	Usage
titleSource	Key	Indicates the source of the title. This attribute is used for titles that are derived from the title of a referred field.
name	Key	The unique identifier of this tag.
firstFirstSource, firstSecondSource, lastFirstSource, lastSecondSource	Key	Indicates the source of the value (for example, a database field).
firstFirstShadowTitle, firstSecondShadowTitle, lastFirstShadowTitle, lastSecondShadowTitle	Display	The title that is displayed in the value-block when no value is visible. Use Shadow titles as hints for data entry.
firstFirstMandatory, firstSecondMandatory, lastFirstMandatory, lastSecondMandatory	Boolean Expression	Indicates whether this element is mandatory, that is, whether you must supply a value before submitting.
firstFirstOpen, firstSecondOpen, lastFirstOpen, lastSecondOpen	(Open Type) or Boolean Expression	The open state of this tag and all its descendants. Valid values are: ▪ False ▪ Never ▪ Create ▪ Update ▪ Always ▪ True In addition, you can specify any Boolean expression that can be evaluated in the relevant context. In this way, you can make the open state dependent on the context.
firstFirstSize, firstSecondSize, lastFirstSize, lastSecondSize	(Size Type)	The size hint that should be applied to the element. The size hints affect the width of the entire element. Valid values are: ▪ Standard ▪ Tiny ▪ Small ▪ Medium

Attribute Name	Type	Usage
		▪ Large ▪ Huge ▪ Vast
firstFirstSearchLayout, firstSecondSearchLayout, lastFirstSearchLayout, lastSecondSearchLayout	Key	The layout that is used if this element launches a foreign key (CTRL+G) search. If nothing is specified, a default search layout (for example, **Find_X.mdml.xml**) is retrieved from the server.
firstFirstSearchView, firstSecondSearchView, lastFirstSearchView, lastSecondSearchView	Key	The layout view that is used if this element launches a foreign key (CTRL+G) search. The view should be part of the layout specified in the searchLayout-attribute. If nothing is specified, the first view from the search layout is chosen.
firstFirstSearchOption, firstSecondSearchOption, lastFirstSearchOption, lastSecondSearchOption	Key	Indicates the name of the search option to be applied when doing “lightweight” searches from within the field. The name must match the name of the filter layout/view implicitly or explicitly stated.
firstFirstSuggestions, firstSecondSuggestions, lastFirstSuggestions, lastSecondSuggestions	Suggestions	Indicates the suggestion strategy for fields based on a foreign key. Valid values are: ▪ Standard ▪ None ▪ Automatic ▪ onDemand
autoSubmit	Boolean	Indicates whether the value that you enter into this element should be automatically submitted when the element loses focus.
style	Styles	The style that applies to this tag and its descendants.
separator	Display	Indicates the separator that should be displayed between the two value-blocks.
appearance	(Element Appearance Type)	Indicates the appearance of this element.  See the explanation in the Forms (Elements) section.

Attribute Name	Type	Usage
		Valid values are: ▪ Standard ▪ Normal ▪ Maximized
frame	Boolean	Indicates whether the field should be rendered with a frame. If **False**, the field is rendered without a frame (as a label). Defaults to **True** unless the frame attribute is specified in one of the containing elements, in which case the value of the closest ancestor is used as the default value.

<Link>

The link element represents an in-line link in a form. When it is active, you can click a link then invoke an action, a hyperlink (for example, a www- or mailto-link), or a link to another workspace. If the link invokes a hyperlink, the URL can be a dynamic expression specified using an attribute or a nested element.

See the <Link> tag in the Action section for details.

Attribute Name	Type	Usage
name	Key	A unique identifier for this table element.
size	(Size Type)	Indicates the size, that is, width, of the column. Valid values are: ▪ Standard ▪ Tiny ▪ Small ▪ Medium ▪ Large ▪ Huge ▪ Vast
visibility	(Visibility Type)	Indicates the visibility of this table column. Valid values are: ▪ Required ▪ Visible ▪ Hidden

Attribute Name	Type	Usage
title	Display	The title that is displayed as column header. This attribute is used for static titles. Cannot be used in the context of <Override>
titleValue	Key	The title that is displayed as column header. This attribute is used for titles that are derived from the value of a referred field. Cannot be used in the context of <Override>
titleSource	Key	The title that is displayed as column header. This attribute is used for titles that are derived from the title of a referred field. Cannot be used in the context of <Override>
action	Key	Indicates the action that should be invoked when this link is clicked. Use a qualified identifier such as **parent.delete**.
template	(Placeholder String)	The title of the link as rendered in an individual cell. You can insert placeholders that are substituted with the arguments from the arguments-attribute.
arguments	Expression List	A semicolon-separated list of arguments.
url	Expression (String)	The hyperlink that is invoked when clicking the link. The process of opening the hyperlink is delegated to the operating system. For example, a www… link is opened by the operating system's default browser.
icon	Id	The resource identifier for an icon to be associated with this link column. This icon (if defined) is displayed in each cell in the column.
IconPosition	(Position Type)	Indicates whether the icon should be placed before or after the title. Valid values are: ▪ Prefix ▪ Postfix
style	Styles	The style that applies to this tag and its descendants.
preTrigger	Key	The identifier of the trigger that is associated with this link. When a preTrigger is specified, invoking the link calls the trigger to run before the link is invoked. If the trigger runs successfully, the link is opened normally. If the trigger fails, the link is not opened, and an error message is displayed.
workspace	Key	Indicates a workspace name which should be navigated when the link is clicked.

<Waypoint>, <Target>, <Restriction>, <Focus>, and <Match> (in the context of <Link>)

<Waypoint> and <Target>

If a link represents a link to a workspace, a workspace name and, if necessary, additional parameters such as waypoints and target, should be specified. `<Waypoint>` is a pane in a path that is taken to navigate to the final `<Target>` pane in a link. Both `<Waypoint>` and `<Target>` represent an item in the link path and have the same attributes and elements.

Attribute Name	Type	Usage
pane	Key	The name of a pane.

<Restriction>

You can set a restriction on a waypoint or target pane, and is specified as an expression.

Attribute Name	Type	Usage
title	Key	Restriction title (required).
expression	Expression (Boolean)	Restriction expression indicates a selection of records that should be displayed in a pane. This is similar to the filter selection.

<Focus>

The focus indicates on which records the focus should be set. Represents a list of match points.

<Match>

The match indicates a record that satisfies the field and value selection and is used to set a focus on that record.

Attribute Name	Type	Usage
field	Key	A field name.
value	Expression	Indicates a value of a field on which the focus should be set.

<Override> and <ElseOverride> (in the Context of <Link>)

You can define override rules that may override the attributes of a link in certain conditions. In tables, you can use this feature to disable a link or change its template based on the state of the individual row. In forms. It works similarly to including multiple links guarded by <If> and <ElseIf> conditionals.

```
<Link title="Link with multiple alternatives"

      action="self.ApproveTimeSheet"
```

```
                    icon="approvetimesheet.png"

                    template="Approve">

                <Override condition="Approved">

        <Link template="(approved)" icon="" disabled="true" />

                </Override>

    </Link>
```

In the preceding example, a link is rendered with the text approve and an approve icon. If the record in a row is already approved, some of the properties of the link are overridden:

- The text is changed.
- The icon is removed, and it is disabled.
- The other properties are inherited from the outer link.

You can provide several overrides using the <ElseOverride> element. If you use multiple overrides, the first override (if any) whose condition evaluates to **True** is applied.

Attribute Name	Type	Usage
condition	Expression	The condition that triggers the override.
default	Boolean	Default value if the condition fails to evaluate due to an error. Defaults to **False**.

<Grid>

A grid is used to render a table-like structure on a form. The cells of a grid can be either title- or value-blocks, and the properties **open** and **mandatory** can be controlled for each individual value cell. Grids do not attempt to align with anything in their scope. Internally, cells are aligned, but externally the grid is not aligned with anything. Grids include headers, rows, and footers, which all correspond to rows in the grid.

Availability Overview - grid version

	Employee Information		Planning Time		Load %		
	No.	Name	Week	Total{In total}	Week	Total{In total}	Keep
First	12	Anders Hanse	42,00	0,00	0,00	0,00	No
Second			0,00	0,00	0,00	0,00	No

A grid can be specified using two different syntaxes:

- **A short form** — The short form is intended as an easy and simple way to create common forms of grids. The syntax also lends itself well to a form where the MDML source code visually resembles the layout of the grid.
- **A verbose form** — The verbose form is an alternative for more complex grids. This syntax follows a more classic XML paradigm, where the visual structure of the grid is not immediately obvious from reading the MDML source code.

The example grid shown above can be specified in the short form in the following manner:

```
<Grid>
```

```
<Header style="boldHeader"
  cells="||Employee Information||
         ||Planning Time||
         ||Load %||
         _"/>
<Header style="simpleHeader"
  cells="|No.|  |Name|
         PlanningUnitVar  |Total{In total}|
         PlanningUnitVar  |Total{In total}|
         |Keep|"/>
<Row title="First"
  cells="Availability1EmployeeNumber  Employee1NameVar
         PlanningTimeFirstUnit1Var
PlannedInShownUnit1Var
         LoadPercentageFirstUnit1Var
LoadPercentageInShownUnit1Var
         Keep1"/>
<Row title="Second"
  cells="Availability2EmployeeNumber  Employee2NameVar
         PlanningTimeFirstUnit2Var
PlannedInShownUnit2Var
         LoadPercentageFirstUnit2Var
LoadPercentageInShownUnit2Var
         Keep2"/>
</Grid>
```

Attribute Name	Type	Usage
-	-	-

<Header>, <Row>, <Footer> (in the Context of <Grid>)

Use a grid header, row, or footer to represent a row in the grid. You specify the configuration of cells or columns in the row using a special short-hand in the cells-attribute. The cell definition contains a set of cells that are values, titles, or empty.

You can annotate a value cell with a mandatory (*) or a closed (-) marker. Furthermore, you can annotate Boolean fields with # to indicate that a check box, rather than a drop-down, should be used. You can attach a secondary source to a value cell by appending (+) and the name of the secondary source. This is primarily used for unit fields. Title cells can span multiple columns by

surrounding them with multiple vertical bars. The complete syntax for the cell definition is displayed in the following figure:

```
definition  =     ws* ( cell ( ws+ cell )* )?

cell        =     primary ( '+' secondary )?

primary           =     field | label | empty

secondary         =     field | label

field       =     ( closed | mandatory )? qualifiedid checkbox?

closed            =     '-'

mandatory         =     '*'

checkbox          =     '#'

label       =     quote+ title quote+

empty       =     '_'

qualifiedid =     a valid id

title       =     any string, escape sequences are \| and \'

quote       =     '|' | ''

ws          =     whitespace
```

Attribute Name	Type	Usage
title	Display	Indicates the title. This attribute is used for static titles.
titleValue	Key	Indicates the source of the title. This attribute is used for titles that are derived from the value of a referred field.
titleSource	Key	Indicates the source of the title. This attribute is used for titles that are derived from the title of a referred field.
cells	(Grid Cells Type)	*See explanation in the section introduction paragraph.*
frame	Boolean	Indicates whether the descendent elements of this container should, by default, be rendered as framed.

<Field> (in the Context of Grid <Header>, <Row>, or <Footer>)

A field is a cell with a title- and a value-block. Use it to render a single source.

Attribute Name	Type	Usage
source	Id	The source of the value-block for this element (for example, a database field).
shadowTitle	Display	The title that is displayed in the value-block when no value is visible. Use Shadow titles as hints for data entry.
title	Display	Indicates the title. This attribute is used for static titles.
titleValue	Key	Indicates the source of the title. This attribute is used for titles that are derived from the value of a referred field.
titleSource	Key	Indicates the source of the title. This attribute is used for titles that are derived from the title of a referred field.
name	Key	The unique identifier of this tag.
mandatory	Boolean Expression	Indicates whether this element is mandatory, that is, whether you must supply a value before submitting.
open	(Open Type) or Boolean Expression	The open state of this tag and all its descendants. Valid values are: ▪ False ▪ Never ▪ Create ▪ Update ▪ Always ▪ True In addition, you can specify any Boolean expression that can be evaluated in the relevant context. In this way, you can make the open state dependent on the context.
size	(Size Type)	The size hint that should be applied to the element. The size hints affect the width of the entire element. Valid values are: ▪ Standard

Attribute Name	Type	Usage
		▪ Tiny ▪ Small ▪ Medium ▪ Large ▪ Huge ▪ Vast
autoSubmit	Boolean	Indicates whether the value that you enter into this element should be automatically submitted when the element loses focus.
searchLayout	Key	The layout that is used if this element launches a foreign key (CTRL+G) search. If nothing is specified, a default search layout (for example, **Find_X.mdml.xml**) is retrieved from the server.
searchView	Key	The layout view that is used if this element launches a foreign key (CTRL+G) search. The view should be part of the layout specified in the searchLayout-attribute. If nothing is specified, the first view from the search layout is chosen.
searchOption	Key	Indicates the name of the search option to be applied when doing “lightweight” searches from within the field. The name must match the name of the filter layout/view implicitly or explicitly stated.
suggestions	Suggestions	Indicates the suggestion strategy for fields based on a foreign key. Valid values are: ▪ Standard ▪ None ▪ Automatic ▪ onDemand
type	(Data Type)	Defines the type to which this value should be rendered. For example, a field of type **timeDuration** can be rendered as **real**.

<Boolean> (in the Context of Grid <Header>, <Row>, or <Footer>)

Use this cell to render a Boolean type source.

Attribute Name	Type	Usage
source	Id	The source of the value-block for this element (for example, a database field).
title	Display	Indicates the title. This attribute is used for static titles.
titleValue	Key	Indicates the source of the title. This attribute is used for titles that are derived from the value of a referred field.
titleSource	Key	Indicates the source of the title. This attribute is used for titles that are derived from the title of a referred field.
name	Key	The unique identifier of this tag.
open	(Open Type) or Boolean Expression	The open state of this tag and all its descendants. Valid values are: ▪ False ▪ Never ▪ Create ▪ Update ▪ Always ▪ True In addition, you can specify any Boolean expression that can be evaluated in the relevant context. In this way, you can make the open state dependent on the context.
autoSubmit	Boolean	Indicates whether the value that you enter into this element should be automatically submitted when the element loses focus.

<UnitField> (in the Context of Grid <Header>, <Row>, or <Footer>)

Use this cell to render a value with a unit. The unit can either be retrieved from a source (for example, a database field) or specified as a static title. The value and unit are visually rendered by a single widget, which makes the value-blocks of the <UnitField>-tag appear as a single block.

Attribute Name	Type	Usage
title	Display	Indicates the title. This attribute is used for static titles.
titleValue	Key	Indicates the source of the title. This attribute is used for titles that are derived from the value of a referred field.
titleSource	Key	Indicates the source of the title. This attribute is used for titles that are derived from the title of a referred field.
name	Key	The unique identifier of this tag.
source, unitSource	Key	Indicates the source of the value (for example, a database field).
shadowTitle, unitShadowTitle	Display	The title that is displayed in the value-block when no value is visible. Use Shadow titles as hints for data entry.
valueMandatory, unitMandatory	Boolean Expression	Indicates whether this element is mandatory, that is, whether you must supply a value before submitting.
valueOpen, unitOpen	(Open Type) or Boolean Expression	The open state of this tag and all its descendants. Valid values are: ▪ False ▪ Never ▪ Create ▪ Update ▪ Always ▪ True In addition, you can specify any Boolean expression that can be evaluated in the relevant context. In this way, you can make the open state dependent on the context.
valueSize, unitSize	(Size Type)	The size hint that should be applied to the element. The size hints affect the width of the entire element. Valid values are:

Attribute Name	Type	Usage
		▪ Standard ▪ Tiny ▪ Small ▪ Medium ▪ Large ▪ Huge ▪ Vast
unitTitle	Display	Indicates the unit in the form of a static title. This is, for instance, used for units such as **%**.
unitPosition	(Unit Position Type)	Indicates the position of the unit in relation to the value. You can explicitly pre- or postfix the unit, or alternatively rely on the currency setting of the operating system to determine the position. Valid values are: ▪ Standard ▪ Prefix ▪ Postfix ▪ Currency
autoSubmit	Boolean	Indicates whether the value that you enter into this element should be automatically submitted when the element loses focus.
unitType	(Data Type)	Defines the type to which this value should rendered. For example, a field of type **timeDuration** can be rendered as **real**.

<Label>, <EmptyLabel> (in the Context of Grid <Header>, <Row>, or <Footer>)

A label is a single block rendering a title. The <EmptyLabel> is equivalent to a <Label>-tag with an empty string as title. Do not use **titleValue** or **titleSource** in the <EmptyLabel>-tag.

Attribute Name	Type	Usage
title	Display	Indicates the title. This attribute is used for static titles.
titleValue	Key	Indicates the source of the title. This attribute is used for titles that are derived from the value of a referred field.

Attribute Name	Type	Usage
titleSource	Key	Indicates the source of the title. This attribute is used for titles that are derived from the title of a referred field.
name	Key	The unique identifier of this tag.

<BooleanGroup>

A Boolean group is a special grouping construct that appears within normal form groups. Use this element to group a set of <Boolean>-tags together and give them a common title.

Attribute Name	Type	Usage
name	Key	The unique identifier of this tag.
mandatory	Boolean Expression	Indicates whether this element is mandatory, that is, whether you must supply a value before submitting.
open	(Open Type) or Boolean Expression	The open state of this tag and all its descendants. Valid values are: ▪ False ▪ Never ▪ Create ▪ Update ▪ Always ▪ True In addition, you can specify any Boolean expression that can be evaluated in the relevant context. In this way, you can make the open state dependent on the context.
autoSubmit	Boolean	Indicates whether the value that you enter into this element should be automatically submitted when the element loses focus.
name	Key	The unique identifier of this tag.
title	Display	Indicates the title. This attribute is used for static titles.
titleValue	Key	Indicates the source of the title. This attribute is used for titles that are derived from the value of a referred field.

Attribute Name	Type	Usage
titleSource	Key	Indicates the source of the title. This attribute is used for titles that are derived from the title of a referred field.

<Boolean>

Use this element to render a Boolean source. The element is rendered as a check box. As opposed to other form elements, the title block of a <Boolean> is placed after the value-block. If you specify this field outside the scope of a <Field> (in the Context of Grid <Header>, <Row>, or <Footer>).

A field is a cell with a title- and a value-block. Use it to render a single source.

Attribute Name	Type	Usage
source	Id	The source of the value-block for this element (for example, a database field).
shadowTitle	Display	The title that is displayed in the value-block when no value is visible. Use Shadow titles as hints for data entry.
title	Display	Indicates the title. This attribute is used for static titles.
titleValue	Key	Indicates the source of the title. This attribute is used for titles that are derived from the value of a referred field.
titleSource	Key	Indicates the source of the title. This attribute is used for titles that are derived from the title of a referred field.
name	Key	The unique identifier of this tag.
mandatory	Boolean Expression	Indicates whether this element is mandatory, that is, whether you must supply a value before submitting.
open	(Open Type) or Boolean Expression	The open state of this tag and all its descendants. Valid values are: ▪ False ▪ Never ▪ Create ▪ Update ▪ Always ▪ True

Attribute Name	Type	Usage
		In addition, you can specify any Boolean expression that can be evaluated in the relevant context. In this way, you can make the open state dependent on the context.
size	(Size Type)	The size hint that should be applied to the element. The size hints affect the width of the entire element. Valid values are: ▪ Standard ▪ Tiny ▪ Small ▪ Medium ▪ Large ▪ Huge ▪ Vast
autoSubmit	Boolean	Indicates whether the value that you enter into this element should be automatically submitted when the element loses focus.
searchLayout	Key	The layout that is used if this element launches a foreign key (CTRL+G) search. If nothing is specified, a default search layout (for example, **Find_X.mdml.xml**) is retrieved from the server.
searchView	Key	The layout view that is used if this element launches a foreign key (CTRL+G) search. The view should be part of the layout specified in the searchLayout-attribute. If nothing is specified, the first view from the search layout is chosen.
searchOption	Key	Indicates the name of the search option to be applied when doing “lightweight” searches from within the field. The name must match the name of the filter layout/view implicitly or explicitly stated.
suggestions	Suggestions	Indicates the suggestion strategy for fields based on a foreign key. Valid values are: ▪ Standard ▪ None ▪ Automatic ▪ onDemand
type	(Data Type)	Defines the type to which this value should be rendered. For example, a field of type **timeDuration** can be rendered as **real**.

<Field> (in the Context of Grid <Header>, <Row>, or <Footer>)

A field is a cell with a title- and a value-block. Use it to render a single source.

Attribute Name	Type	Usage
source	Id	The source of the value-block for this element (for example, a database field).
shadowTitle	Display	The title that is displayed in the value-block when no value is visible. Use Shadow titles as hints for data entry.
title	Display	Indicates the title. This attribute is used for static titles.
titleValue	Key	Indicates the source of the title. This attribute is used for titles that are derived from the value of a referred field.
titleSource	Key	Indicates the source of the title. This attribute is used for titles that are derived from the title of a referred field.
name	Key	The unique identifier of this tag.
mandatory	Boolean Expression	Indicates whether this element is mandatory, that is, whether you must supply a value before submitting.
open	(Open Type) or Boolean Expression	The open state of this tag and all its descendants. Valid values are: ▪ False ▪ Never ▪ Create ▪ Update ▪ Always ▪ True In addition, you can specify any Boolean expression that can be evaluated in the relevant context. In this way, you can make the open state dependent on the context.
size	(Size Type)	The size hint that should be applied to the element. The size hints affect the width of the entire element. Valid values are: ▪ Standard

Attribute Name	Type	Usage
		▪ Tiny ▪ Small ▪ Medium ▪ Large ▪ Huge ▪ Vast
autoSubmit	Boolean	Indicates whether the value that you enter into this element should be automatically submitted when the element loses focus.
searchLayout	Key	The layout that is used if this element launches a foreign key (CTRL+G) search. If nothing is specified, a default search layout (for example, **Find_X.mdml.xml**) is retrieved from the server.
searchView	Key	The layout view that is used if this element launches a foreign key (CTRL+G) search. The view should be part of the layout specified in the searchLayout-attribute. If nothing is specified, the first view from the search layout is chosen.
searchOption	Key	Indicates the name of the search option to be applied when doing “lightweight” searches from within the field. The name must match the name of the filter layout/view implicitly or explicitly stated.
suggestions	Suggestions	Indicates the suggestion strategy for fields based on a foreign key. Valid values are: ▪ Standard ▪ None ▪ Automatic ▪ onDemand
type	(Data Type)	Defines the type to which this value should be rendered. For example, a field of type **timeDuration** can be rendered as **real**.

<Boolean> (in the Context of Grid <Header>, <Row>, or <Footer>)

Use this cell to render a Boolean type source.

Attribute Name	Type	Usage
source	Id	The source of the value-block for this element (for example, a database field).
title	Display	Indicates the title. This attribute is used for static titles.
titleValue	Key	Indicates the source of the title. This attribute is used for titles that are derived from the value of a referred field.
titleSource	Key	Indicates the source of the title. This attribute is used for titles that are derived from the title of a referred field.
name	Key	The unique identifier of this tag.
open	(Open Type) or Boolean Expression	The open state of this tag and all its descendants. Valid values are: ▪ False ▪ Never ▪ Create ▪ Update ▪ Always ▪ True In addition, you can specify any Boolean expression that can be evaluated in the relevant context. In this way, you can make the open state dependent on the context.
autoSubmit	Boolean	Indicates whether the value that you enter into this element should be automatically submitted when the element loses focus.

<UnitField> (in the Context of Grid <Header>, <Row>, or <Footer>)

Use this cell to render a value with a unit. The unit can either be retrieved from a source (for example, a database field) or specified as a static title. The value and unit are visually rendered by a single widget, which makes the value-blocks of the <UnitField>-tag appear as a single block.

Attribute Name	Type	Usage
title	Display	Indicates the title. This attribute is used for static titles.
titleValue	Key	Indicates the source of the title. This attribute is used for titles that are derived from the value of a referred field.
titleSource	Key	Indicates the source of the title. This attribute is used for titles that are derived from the title of a referred field.
name	Key	The unique identifier of this tag.
source, unitSource	Key	Indicates the source of the value (for example, a database field).
shadowTitle, unitShadowTitle	Display	The title that is displayed in the value-block when no value is visible. Use Shadow titles as hints for data entry.
valueMandatory, unitMandatory	Boolean Expression	Indicates whether this element is mandatory, that is, whether you must supply a value before submitting.
valueOpen, unitOpen	(Open Type) or Boolean Expression	The open state of this tag and all its descendants. Valid values are: ▪ False ▪ Never ▪ Create ▪ Update ▪ Always ▪ True In addition, you can specify any Boolean expression that can be evaluated in the relevant context. In this way, you can make the open state dependent on the context.
valueSize, unitSize	(Size Type)	The size hint that should be applied to the element. The size hints affect the width of the entire element. Valid values are: ▪ Standard ▪ Tiny ▪ Small ▪ Medium ▪ Large

Attribute Name	Type	Usage
		▪ Huge ▪ Vast
unitTitle	Display	Indicates the unit in the form of a static title. This is, for instance, used for units such as **%**.
unitPosition	(Unit Position Type)	Indicates the position of the unit in relation to the value. You can explicitly pre- or postfix the unit, or alternatively rely on the currency setting of the operating system to determine the position. Valid values are: ▪ Standard ▪ Prefix ▪ Postfix ▪ Currency
autoSubmit	Boolean	Indicates whether the value that you enter into this element should be automatically submitted when the element loses focus.
unitType	(Data Type)	Defines the type to which this value should rendered. For example, a field of type **timeDuration** can be rendered as **real**.

<Label>, <EmptyLabel> (in the Context of Grid <Header>, <Row>, or <Footer>)

A label is a single block rendering a title. The <EmptyLabel> is equivalent to a <Label>-tag with an empty string as title. Do not use **titleValue** or **titleSource** in the <EmptyLabel> tag.

Attribute Name	Type	Usage
title	Display	Indicates the title. This attribute is used for static titles.
titleValue	Key	Indicates the source of the title. This attribute is used for titles that are derived from the value of a referred field.
titleSource	Key	Indicates the source of the title. This attribute is used for titles that are derived from the title of a referred field.
name	Key	The unique identifier of this tag.

<Calendar>

Use a calendar element as an elaborate rendering of a date source. The date value is plotted into a full calendar, and you have the option to display it in an input field above the calendar. Calendars support a more advanced form of styles as described in the section about the <Chart> tag (Chart Style).

The chart style is specified using a <Chart> tag nested within a <Style> tag. Use this to specify a style specifically for the chart widgets (pie/bar/dial).

Attribute Name	Type	Usage
chartOrientation	(Orientation)	(Bar chart only) Indicates chart orientation: ▪ Vertical (bars are vertical) ▪ Horizontal
textOrientation	(Orientation)	(Bar chart only) Indicates text orientation for (Y) axis values: ▪ Vertical or Horizontal ▪ Decimal values from 90 to 90
showLabels	Boolean	(Bar and pie charts) Indicates whether to display text labels with values on the chart.
showLegend	Boolean	(Bar and pie charts) Indicates whether to display a legend block on the chart.
showGridLines	Boolean	(Bar chart only) Indicates whether to display grid lines on the chart area.
gridLinesColor	(Color)	(Bar chart only) Indicates the color of grid lines. Can be a named color, a valid RGB color, or the client default (**Standard**).
palette	Key	(Bar and pie charts) References a palette to be used for styling individual bars/pie sectors.

<Palette>-tag>

Palette is a sequence of styles to be applied to bars in a bar chart and to pie sectors in pie charts. The <Palette> tag is nested within a <Chart> tag. You can add styles to palette as inner <Style> tags or by referencing via style attributes.

Attribute Name	Type	Usage
name	Key	The name that is used to refer to this palette. The name must be unique in the evaluation context.
ref	Key	Apply the values from a referred palette to the current <Palette> tag, except for those that are explicitly defined in the local context.

Attribute Name	Type	Usage
style	Key List	A list of style names separated by semicolons.

<Switch>-tag

Attribute Name	Type	Usage
source	Id	The source of the value-block for this element (for example, a database field).
title	Display	Indicates the title. Use this attribute for static titles.
titleValue	Key	Indicates the source of the title. Use this attribute for titles that are derived from the value of a referred field.
titleSource	Key	Indicates the source of the title. Use this attribute for titles that are derived from the title of a referred field.
name	Key	The unique identifier of this tag.
mandatory	Boolean Expression	Indicates whether this element is mandatory, that is, whether you must supply a value before submitting.
open	(Open Type) or Boolean Expression	The open state of this tag and all its descendants. Valid values are: ▪ False ▪ Never ▪ Create ▪ Update ▪ Always ▪ True In addition, you can specify any Boolean expression that can be evaluated in the relevant context. In this way, you can make the open state dependent on the context.
autoSubmit	Boolean	Indicates whether the value that you enter into this element should be automatically submitted when the element loses focus.
style	Styles	The style that applies to this tag and its descendants.

Attribute Name	Type	Usage
appearance	(Element Appearance Type)	Indicates the appearance of this element.  See the explanation in the Forms (Elements) section. Valid values are: ▪ Standard ▪ Normal ▪ Maximized
inputField	Boolean Expression	Indicates whether an input field should be displayed above the calendar. Use the input field to emphasize the current selection of the calendar.
selectionSource	Key	Indicates the field used to determine the range of dates within the current week. In other words, this attribute is used to configure week selection for the calendar. If you set **selectionPeriod** to **fullWeek**, the value of this attribute is ignored.
autoSelection	Boolean Expression	Indicates that each navigation in the calendar updates the selection and thereby the inputField. The difference between auto-submit and auto-selection is that auto-selection is a client-side feature that does not involve any server-communication.
selectionPeriod	(Selection Type)	Indicates the basic unit of selection in the calendar. The default is **Day**. Other possible values are: ▪ splitWeek ▪ fullWeek If the selectionSource attribute is also present, that determines the actual selection.

<Element>

Use the <Element>-tag to insert custom elements with block configurations that are not covered by the predefined elements. You can create a custom block configuration by nesting a <Label>-tag, and one or more of the following block tags:

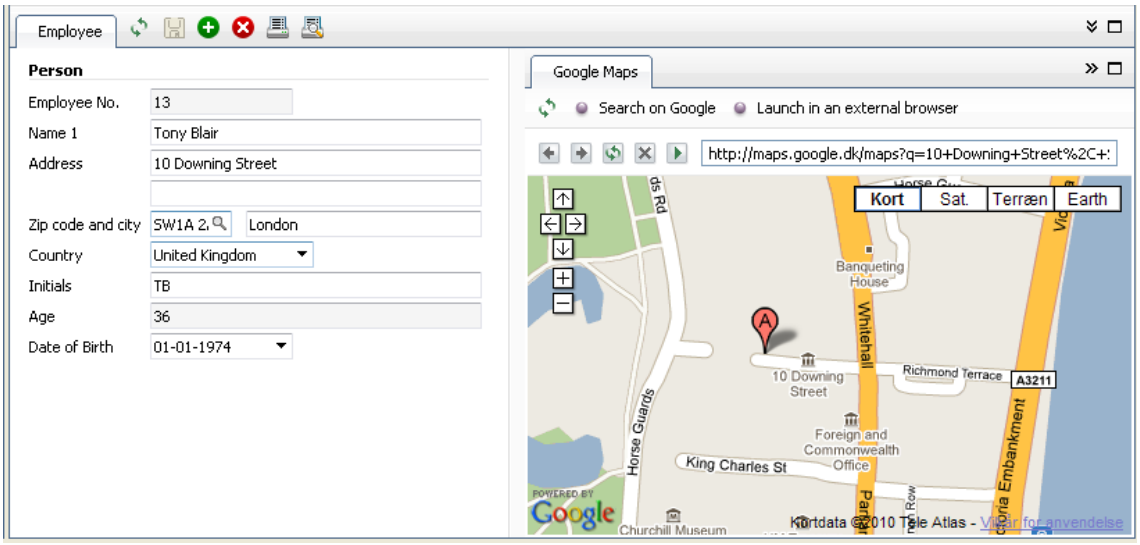
- <FieldBlock>
- <PeriodYearBlock>
- <ComboFieldBlock>
- <LabelBlock>
- <ZipCityBlock>

Use of custom elements is **strongly discouraged** because alignment cannot be guaranteed.

Attribute Name	Type	Usage
name	Key	The unique identifier of this tag.
mandatory	Boolean Expression	Indicates whether this element is mandatory, that is, you must supply a value before submitting.
open	(Open Type) or Boolean Expression	The open state of this tag and all its descendants. Valid values are: ▪ False ▪ Never ▪ Create ▪ Update ▪ Always ▪ True In addition, you can specify any Boolean expression that can be evaluated in the relevant context. In this way, you can make the open state dependent on the context.
size	(Size Type)	The size hint that should be applied to the element. The size hints affect the width of the entire element. Valid values are: ▪ Standard ▪ Tiny ▪ Small ▪ Medium ▪ Large ▪ Huge ▪ Vast
autoSubmit	Boolean	Indicates whether the value that you enter into this element should be automatically submitted when the element loses focus.

Browser

The following is a screenshot of a browser view as rendered in the client.



<Browser>-tag

A browser is a specialized view to render the MWSL component card using an embedded Web browser. A browser view can accept parameters similar to a filter view.

Attribute Name	Type	Usage
name	Key	The name of this view. The name should uniquely identify this view within the scope of the containing MDML pane.
title	Display	Use the view title to override the title that is defined on the pane. This does, however, not take effect if a title is defined on the corresponding MWSL component.
style	Styles	The style that applies to this tag and its descendants.
open	(Open Type) or Boolean Expression	The open state of this tag and all its descendants. Valid values are: ▪ False ▪ Never ▪ Create ▪ Update ▪ Always ▪ True

Attribute Name	Type	Usage
		In addition, you can specify any Boolean expression that can be evaluated in the relevant context. In this way, you can make the open state dependent on the context.
JavaScript	Expression (Boolean)	Indicates whether JavaScript should be enabled in the embedded browser.
loginRules	Expression List	Indicates additional login rules that are applicable for this view.
autoRefresh	Boolean	Indicates that the browser widget should refresh whenever the pane is refreshed. When you turn autoRefresh off, refresh the browser widget by clicking the Refresh button in the pane toolbar or the dedicated browser refresh in the widgets toolbar.

<Parameters>-tag

Any view can accept parameters from the corresponding MWSL specification. Use the <Parameters>-tag to hold these parameter declarations. If a view with parameters is invoked from MWSL without arguments, an exception occurs.

Attribute Name	Type	Usage
-	-	-

<Parameter>-tag

Each view parameter has a mandatory name and type. The name is used to bind variables in the view during evaluation of conditionals and expressions. It should therefore be unique and not overlap any existing identifiers in the evaluation context.

Attribute Name	Type	Usage
name	Key	The name of the parameter. Use this name in the conditionals and expressions of this view as a valid identifier. The name is required.
type	(Field Type)	The type of the parameter. Failure to comply with the type restriction imposed by this declaration causes an exception when invoking the view from a MWSL specification. Valid values are: ▪ Boolean ▪ Integer ▪ Real ▪ Amount

Attribute Name	Type	Usage
		▪ String ▪ Date ▪ Time ▪ Enum

<ControlBar>-tag (in the Context of a <Browser>)

The control bar of a browser corresponds to the control bar of a regular browser like Internet Explorer or Firefox.

Attribute Name	Type	Usage
name	Key	The unique identifier of this tag.
navigation	Expression (Boolean)	Indicates whether the browser navigation buttons should be visible.
address	Expression (Boolean)	Indicates whether the address or location field showing the current URL should be visible.
progress	Expression (Boolean)	Indicates whether a progress indicator should be displayed when performing a request.
status	Expression (Boolean)	Indicates whether a status message explaining when the browser page was last updated should be visible.

<Url>-tag

The URL displayed in the browser is specified by the value attribute on this tag. URLs are dynamically generated so the type of the attribute is a string expression. This means that even for static URLs, the contents of the value attribute must be quoted using single-quotes (‘). As an alternative to the value attribute, you can specify a nested <Value>-tag that allows a full CDATA section. The value attribute and the nested value tag are mutually exclusive.

Attribute Name	Type	Usage
name	Key	The unique identifier of this tag.
value	Expression (String)	A string expression that forms the basis for the URL supplied to the embedded Web browser.

<Query>-tag (in the Context of a <Url>-tag)

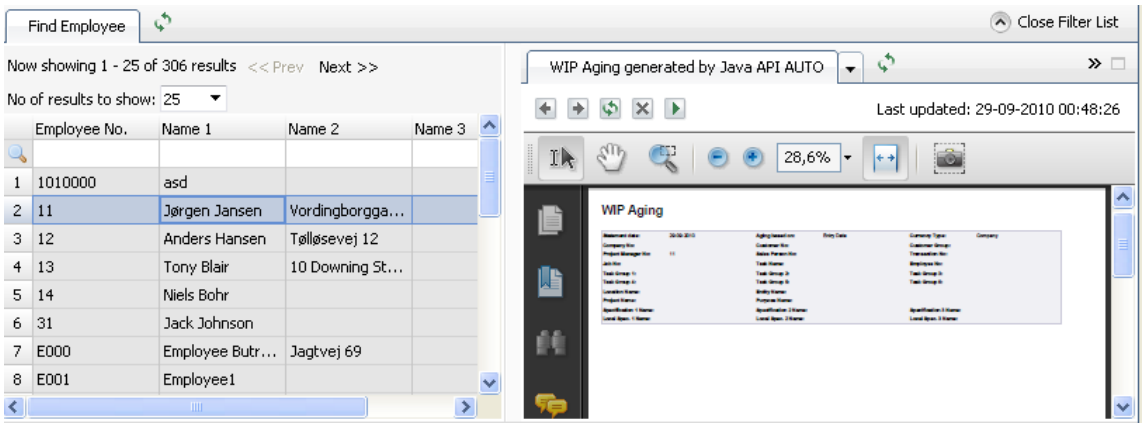
You can specify URL query parameters using the <Query>-tag. This follows the standard HTTP query string format. The actual fields and values differ from Web service to Web service. The value is resolved in the same manner as a <Description>-tag by means of placeholder substitution. The value-attribute is the template, and the arguments are expressions. The main

difference from the <Description>-tag is that the value does not use locale-specific formatting; that is, it is universal/canonical, and it is URL-encoded before being entered in the browser.

Attribute Name	Type	Usage
field	Id	The HTTP query field.
value	(Placeholder String)	The template string that is used as the basis for placeholder substitution. This attribute is required. The format for placeholders is ^ and an integer representing the index of the argument in the arguments list.
arguments	Expression List	A semicolon-separated list of arguments.

Report

The following is a screen shot of a report view as rendered in the client.



<Report>-tag

A report is a specialized view that uses an external report generator to render the MWSL component card. A report view can accept parameters similar to a filter view.

Attribute Name	Type	Usage
name	Key	The name of this view. The name should uniquely identify this view within the scope of the containing MDML pane.
title	Display	Use the view title to override the title that is defined on the pane. This does, however, not take effect if a title is defined on the corresponding MWSL component.
style	Styles	The style that applies to this tag and its descendants.
open	(Open Type) or Boolean Expression	The open state of this tag and all its descendants. Valid values are: ▪ False ▪ Never ▪ Create ▪ Update ▪ Always ▪ True In addition, you can specify any Boolean expression that can be evaluated in the relevant context. In this way, you can make the open state dependent on the context.

Attribute Name	Type	Usage
source	Expression (String)	Indicates the source of the report. The precise definition of the source depends on the third-party system that produces the report. This attribute is required.
view	Expression (String)	Indicates the view to be used from the source of the report. The precise definition of the view depends on the third-party system that produces the report. The default is the empty string.
output	(Output Type)	The output format of the report. A report can either open a report inline or in an external window. The default format for the report is HTML. Valid values are: ▪ html ▪ pdf ▪ xls
autorun	Expression (Boolean)	Indicates whether the report should run automatically whenever new data is received. The default value is **True**; however, for time-consuming reports, consider turning this setting off.
loginRules	Expression List	Indicates additional login rules that are applicable to this view.
engine	(Engine Type)	Specifies the backend engine delivering this report. The default value is “businessObjects” which is the primary report provider in Maconomy. To use existing Analyzer reports as known from the Java Client, specify “analyzer” (see separate documentation for more on this). In principle, you can specify other report providers here but that must be complemented by an extension that leverages such a report provider.

<Parameters>-tag

Any view can accept parameters from the corresponding MWSL specification. Use the <Parameters>-tag to hold these parameter declarations. If a view with parameters is invoked from MWSL without arguments, an exception occurs.

Attribute Name	Type	Usage
-	-	-

<Parameter>-tag

Each view parameter has a mandatory name and type. Use the name to bind variables in the view during evaluation of conditionals and expressions. It should be unique and not overlap any existing identifiers in the evaluation context.

Attribute Name	Type	Usage
name	Key	The name of the parameter. Use this name in the conditionals and expressions of this view as a valid identifier. The name is required.
type	(Field Type)	The parameter type. Failure to comply with the type restriction imposed by this declaration causes an exception when invoking the view from a MWSL specification. Valid values are: ▪ Boolean ▪ Integer ▪ Real ▪ Amount ▪ String ▪ Date ▪ Time ▪ Enum

<ControlBar>-tag (in the Context of a <Report>)

The control bar of a report is similar to the control bar (except with regard to default configuration).

Attribute Name	Type	Usage
name	Key	The unique identifier of this tag.
navigation	Expression (Boolean)	Indicates whether the browser navigation buttons should be visible. The default is **False**.

Attribute Name	Type	Usage
address	Expression (Boolean)	Indicates whether the address or location field showing the current URL should be visible. The default is **False**.
progress	Expression (Boolean)	Indicates whether a progress indicator should be displayed when performing a request. The default is **True**.
status	Expression (Boolean)	Indicates whether a status message explaining when the browser page was last updated should be visible. The default is **True**.

<Query>-tag (in the Context of a <Report>-tag)

The container tag for query arguments.

Attribute Name	Type	Usage
-	-	-

<Argument>-tag

An argument is a dynamically generated value that is supplied to the underlying source of the report. The precise format differs between report providers.

Attribute Name	Type	Usage
field	Id	The unique identifier of this tag.
value	(Placeholder String)	The template string that is used as the basis for placeholder substitution. This attribute is required. The format for placeholders is ^ and an integer representing the index of the argument in the arguments list.
arguments	Expression List	A semicolon-separated list of arguments.

Introduction

The goal of MDML is to deliver a new way of specifying and rendering dialog layouts in the Bellatrix project. This implies that MDML specifications are read, parsed, formatted, and finally rendered by the MDML engine as part of the new Bellatrix client.

The most important feature of MDML is the focus on *what* rather than on *how*. In other words, the MDML author should primarily concern himself with describing what data should be displayed rather than how it should be displayed. The underlying engine facilitates this by deciding how to present and align the different elements.

This section contains the full reference of the expression language and its use in MDML. It discusses MDML specifications, elements, expressions, style properties, and other related functionalities.

Panes and Views

An MDML specification contains the layout of a dialog box. The layout describes different presentations of the dialog box, depending on the type of pane that it is embedded in. Pane types include filter pane, upper pane, lower pane, or simply pane. For each pane type, the layout specification can contain several different views for the dialog box, such as filter, form, and table. Each pane type can also be associated with auxiliary information such as definitions of styles, functions, wizards, and actions that are shared among the views.

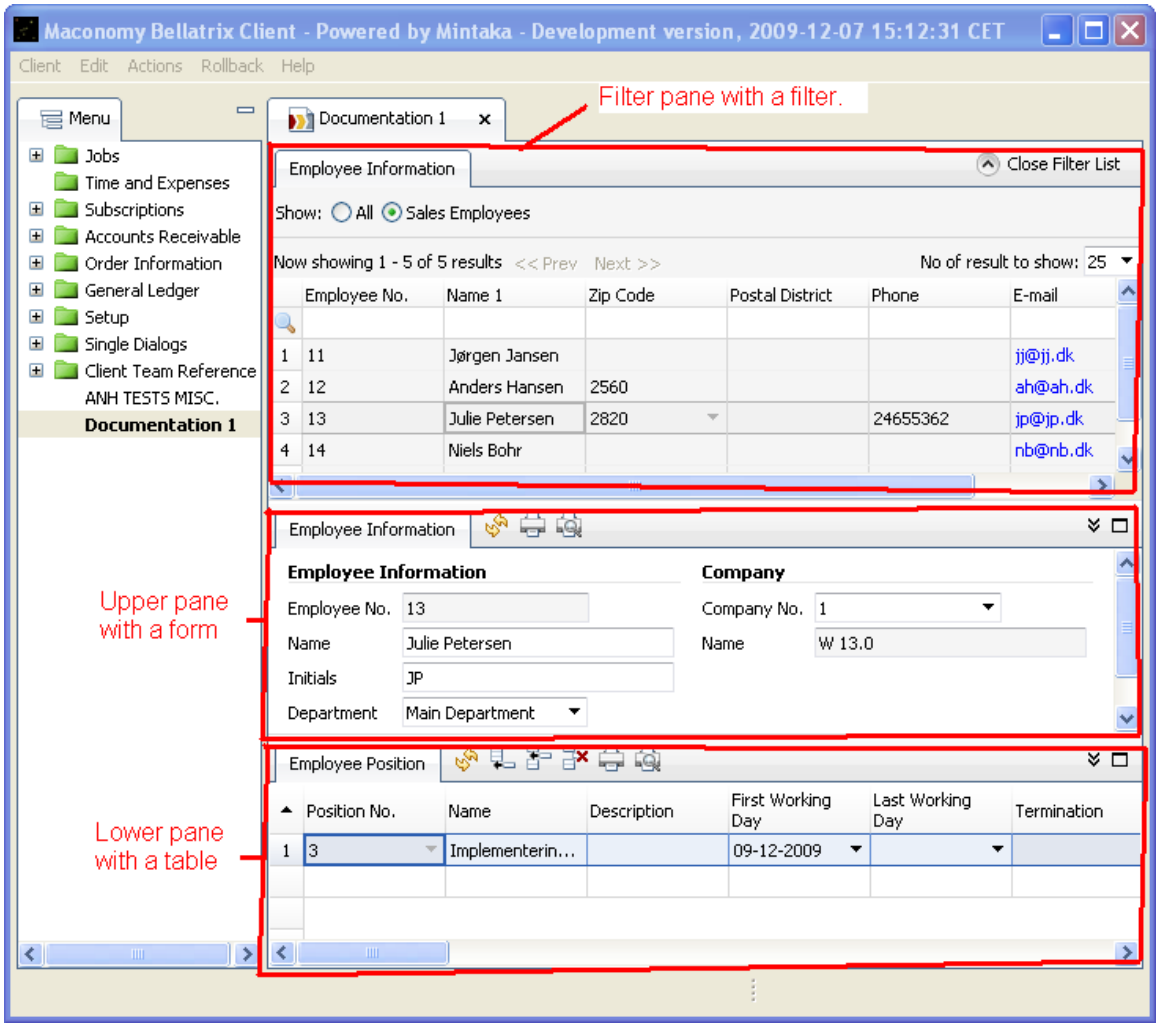
More specifically, a layout may contain a filter pane, followed by either a pane, or by an upper pane and a lower pane. Each pane can be rendered in different views such as a form, table, and filter. A filter pane contains filters. A pane contains forms. An upper pane also contains forms, and a lower pane contains both tables and forms.

```
<?xml version="1.0" encoding="UTF-8"?>
<MDML version="0.1" xmlns="http://www.maconomy.com/ns/mdml">
  <Layout name="c_documentation2" dialog="EmployeePositions">
    <FilterPane>
      <!-- PUT FILTERS HERE -->
    </FilterPane>
    <UpperPane>
      <!-- PUT FORMS HERE -->
    </UpperPane>
    <LowerPane>
      <!-- PUT TABLES AND FORMS HERE -->
    </LowerPane>
  </Layout>
</MDML>
```

Forms present one set of related data (typically one record in the database), whereas tables present several sets of related data (several records in the database). Filters are similar to normal tables, but they have an additional control bar that can contain search restrictions.

The following figure illustrates three different renderings of the *EmployeePositions* dialog box. At the top, it is rendered as a filter in a filter pane. In the middle, it is rendered as a form in an upper pane. Finally, at the bottom, it is rendered as a table in a lower pane.

The concrete composition of these three panes in a single workspace is described in *MWSL* and is therefore not of concern to MDML. MDML solely describes the presentation of the dialog box in the context of each individual pane.



Views

This section describes the three kinds of views. Because tables and filters are fairly similar and relatively easy to understand, they are described first. Forms, which are more conceptually challenging, are described next.

Tables

The table view presents a set of records in a relation. The layout specifies the fields that should be shown and the order in which they are shown. The table header can be customized by overriding the default titles of the fields. Furthermore, it is possible to specify which fields should always be visible in the table and which fields can be added or removed by the user using the *columns chooser*. The layout for the table in the lower pane in the preceding figure is shown here.

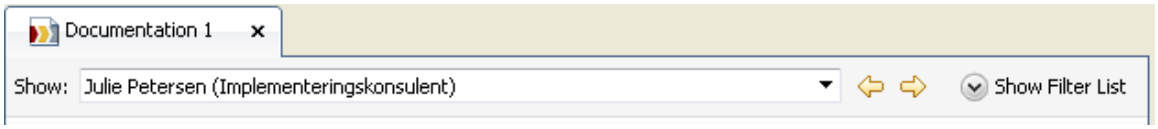
```
<Table>
  <Actions all="true"/>
  <Columns>
    <Field field="PositionNumber" required="true"/>
    <Field field="Name" required="true"/>
    <Field field="Description"/>
    <Field field="EmploymentDate"/>
    <Field field="DateEndEmployment"/>
    <Field field="LastWorkingDay"
defaultHidden="true"/>
    <Field field="LastCompensationPayday"
defaultHidden="true"/>
    <Field field="EndEmploymentVar"
title="Termination"/>
  </Columns>
</Table>
```

Filters

The main difference between a table and a filter is the control bar. Both views contain a table that can be customized as described above. The first row of the filter table is special, since the cells are used for user-defined search criteria. The filter table is also optionally equipped with a control bar that allows the user to restrict the records shown in the table. Each restriction of the filter is implemented as an expression in the Expression Language. The filter in the filter pane of the preceding figure is specified in the following manner.

```
<Filter>
  <ControlBar>
    <Selection default="currentemployees">
      <Option name="all" title="All"/>
      <Option name="salesemployees"
              title="Sales Employees"
              restriction="SalesEmployee = true"/>
      <Option name="currentemployees"
              title="Current Employees">
        <Restriction <![CDATA[
          LastWorkingDay inrange [date(nulldate) .. today()]
        ]]></Restriction>
      </Option>
    </Selection>
  </ControlBar>
  <Compact>
    <Description value="^1 (^2)" arguments="Name1 Position"/>
  </Compact>
  <Columns>
    <Field field="EmployeeNumber" required="true"/>
    <Field field="Name1"/>
    <Field field="Name2" defaultHidden="true"/>
    <Field field="Name3" defaultHidden="true"/>
    <Field field="ZipCode"/>
    <Field field="PostalDistrict"/>
    <Field field="Telephone"/>
    <Field field="ElectronicMailAddress"/>
  </Columns>
</Filter>
```

with the values of the fields in the *arguments* attribute during evaluation, as shown in the following figure.



Incidentally, the compact section of a filter is also used to specify the table layout of value pickers. You can enforce a certain “minimal” search restriction on a filter by directly specifying a restriction on the `<Selection>`-element placeholder text.

```
<Filter title="Find Job">
  <ControlBar>
    <Selection>
      <Restriction> <![CDATA[
        not Template
        ]]>
      </Restriction>
      <Option title="All" name="all"></Option>
      <Option title="My Jobs" name="my">
        <Restriction> <![CDATA[
          ProjectManagerNumber = userEmployeeNumber()
          ]]>
        </Restriction>
      </Option>
    </Selection>
  </ControlBar>
```

This specifies that only non-template jobs can be shown in the filter, regardless of the option chosen by the user. This restriction, which results from choosing an option, is the conjunction of the general restriction and the restriction pertaining to the selected option.

When you invoke searching from fields (“value pickers”), the general restriction is always implicitly applied. In addition, you can specify that the restriction of a named search option must be applied when presenting search suggestions in the value picker.

You can apply default sorting to the filter using the <OrderBy>-element. The source of the sorted field and the order type must be defined when initializing default sorting.

```
  <Columns>
    <OrderBy source="EmployeeNumber" order="ascending"/>
    <Field field="EmployeeNumber" required="true"/>
    <Field field="Name1"/>
    <Field field="Name2" defaultHidden="true"/>
    <Field field="Name3" defaultHidden="true"/>
    <Field field="ZipCode"/>
    <Field field="PostalDistrict"/>
    <Field field="Telephone"/>
    <Field field="ElectronicMailAddress"/>
  </Columns>
```

In certain conditions, you can define override rules to override the attributes of an order by an element. For example, in the following case, the filter is sorted by “PostalDistrict” in ascending order if the “conditionOne()” function is true, while it is sorted by “ZipCode” in descending order if the “conditionOne()” function is false, and the “conditionTwo()” function is true. Otherwise, the filter is sorted by “EmployeeNumber” in ascending order.

```
<Columns>
    <OrderBy source="EmployeeNumber" order="ascending">
      <Override condition="conditionOne()">
        <OrderBy source="PostalDistrict" order="ascending"/>
      </Override>
      <ElseOverride condition="conditionTwo()">
        <OrderBy source="ZipCode" order="descending"/>
      </ElseOverride>
```

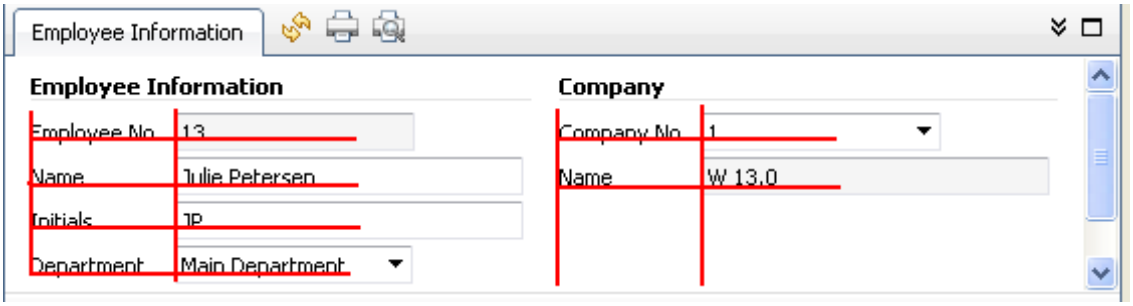
Forms

Use the form view to display a single record. This view facilitates a much greater degree of customization than tables and filters at the cost of added complexity. Ideally, the MDML author should only focus on enumerating the data fields in the form and the logical grouping of these fields. The underlying engine then ensures reasonable layout and alignment of the data. However, MDML allows for significantly more subtle customization and tweaking of the way things align. For more information, see Best Practices.

The content of a form is arranged in rows, columns, and groups. The content of a group is arranged in rows and columns. The primary purpose of a group is to combine related data (optionally, with a common title). Each piece of data in a form is represented by elements, such as *Field*, *Interval*, and *Grid*. The layout of the form in the upper pane in the first figure is specified in the following manner.

```
<Form>
  <Actions all="true"/>
  <Column>
    <Group title="Employee Information">
      <Field field="EmployeeNumber"/>
      <Field field="Name1" title="Name"/>
      <Field field="Initials"/>
      <Field field="DepartmentNumber"/>
      <Field field="Position"/>
    </Group>
  </Column>
  <Column>
    <Group title="Company">
      <Field field="CompanyNumber"/>
      <Field field="CompanyNameVar" title="Name"/>
    </Group>
```

When a dialog box is displayed in a form, the MDML engine computes the vertical and horizontal positions of each visible part (groups and elements). The engine performs this computation instead of simply delegating the work to the underlying graphical library because MDML guarantees a certain degree of alignment between elements. The following figure illustrates this alignment. The vertical lines show how blocks (the parts of an element) align across elements when a set of elements are laid out vertically. This kind of alignment is called *horizontal alignment*. The horizontal lines show the *baseline alignment* of blocks within a single element.



The MDML engine also computes the minimum, preferred, and maximum size of each block. The layout manager uses this information to enable a pleasant visual experience when the window shrinks or grows. The widths of the different fields in the preceding vary because they depend on the type of the underlying data field, as well as whether the field is a key field or not.

Horizontal Alignment in Forms

The MDML engine ensures alignment between the different components of a form. The components include visible parts such as elements and groups, as well as invisible parts such as rows, columns, and scopes.

Elements

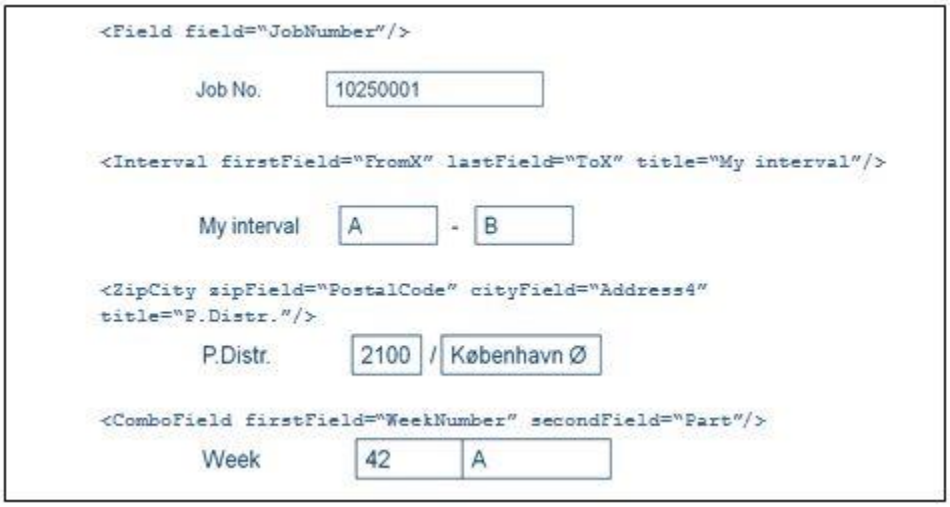
MDML provides the capability to express any combination of fields and labels. However, some combinations that are very common are provided as built-in elements. Thus, an element is an abstraction of a number of fields and labels.

Elements do not only specify which labels and fields are involved in their construction. They also specify how they are aligned against each other. Elements are defined in terms of rulers. For more information, see Rulers and Alignment.

Some of the elements can appear in a *slim* form. They are intended for aligning non-interval elements nicely with interval elements, and for aligning intervals with each other.

Some elements also exist in a *full* version, where the label is left out. This is typically used when one label is common for a group of fields, such as in an address specification, where only the first line has a label to provide as much space as possible.

The following figure shows examples of MDML elements.

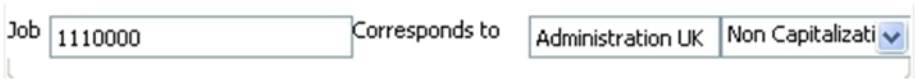


Custom Elements

If the special form elements do not suffice, custom elements can be used instead. They provide the option of controlling alignment more explicitly.

Custom elements are composed of blocks, such as the basic blocks Label and Field, and the combined forms ComboField, ZipCity, UnitField, and PeriodYear. The following figure shows an example of a custom element.

```
<Element>
   <Label title="Job"/>
       <FieldBlock field="JobNumber"/>
       <LabelBlock title="Corresponds to"/>
       <ComboFieldBlock firstField="JobName"
secondField="JobGroup"/>
</Element>
```



Custom elements should be used with caution, as the engine does not centrally control their layout and alignment.

If the first block of a custom element is a label tag, the label will be aligned with the labels of the rest of the elements. Use label blocks to prevent this kind of alignment. However, you cannot use label tags in a position different from the first one.

Rulers and Alignment

Rulers ensure alignment in MDML. A ruler is part of every container and element. It specifies how their content is arranged, and how it should be aligned with other elements outside and inside the container.

Rulers organize the content of elements in 12 columns. The columns do not need to have the same width, and some columns can (and will often) have 0 width. The columns are separated by 13 tab stops, numbered from 0 to 12.

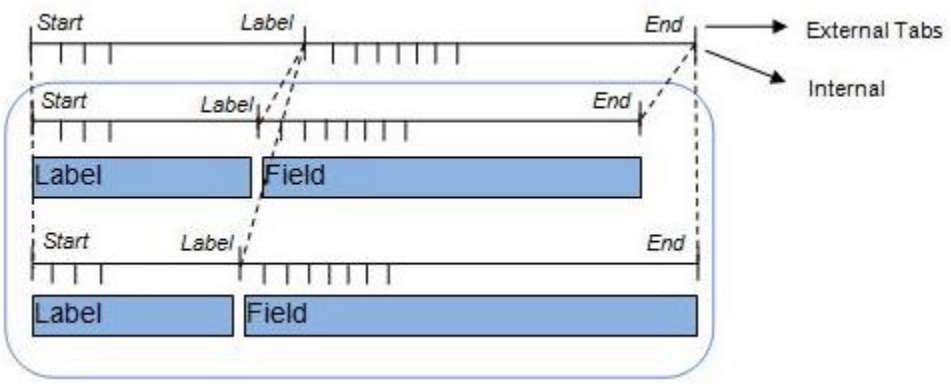
Every component within the element must specify which tab stop it begins at and which tab stop it ends at. In many cases, this specification is implicit, since either it is predefined, like in elements, or given a the combination of rulers.

The 13 tab stops of a ruler are referred to as inner tab stops. The exposed tab stops are the tab stops that define the alignment with the components outside the container. They are always a subset of the inner tab stops.

Each exposed tab stop is formed by a position and a type. Elements and containers that expose a certain tab will be aligned with other elements that expose the same tab (in the same position and with the same type).

For example, assume that a group contains two field elements, as shown in the following figure. Each element specifies that the label starts at tab stop 0 and ends at tab stop 4, and that the input field starts at tab stop 4 and ends at tab stop 12. The ruler of each element always exposes tab stops 0 and 12, as well as tab stop 4, to align the ends of the labels. The exposed tab stops of the elements are propagated to the ruler of the group, as the figure shows.

To sum it up, the start and the end of the elements are always aligned. Labels always try to align with labels, intervals with intervals, and so on. In the case of custom elements, which are user-defined, the alignment depends on the number of elements in each row, since both tab stop and tab type must match.

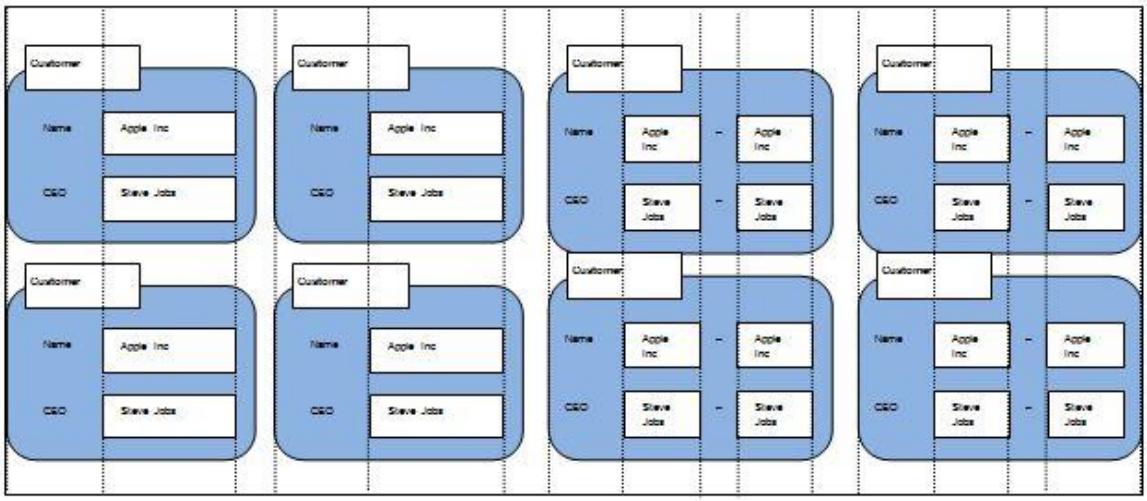


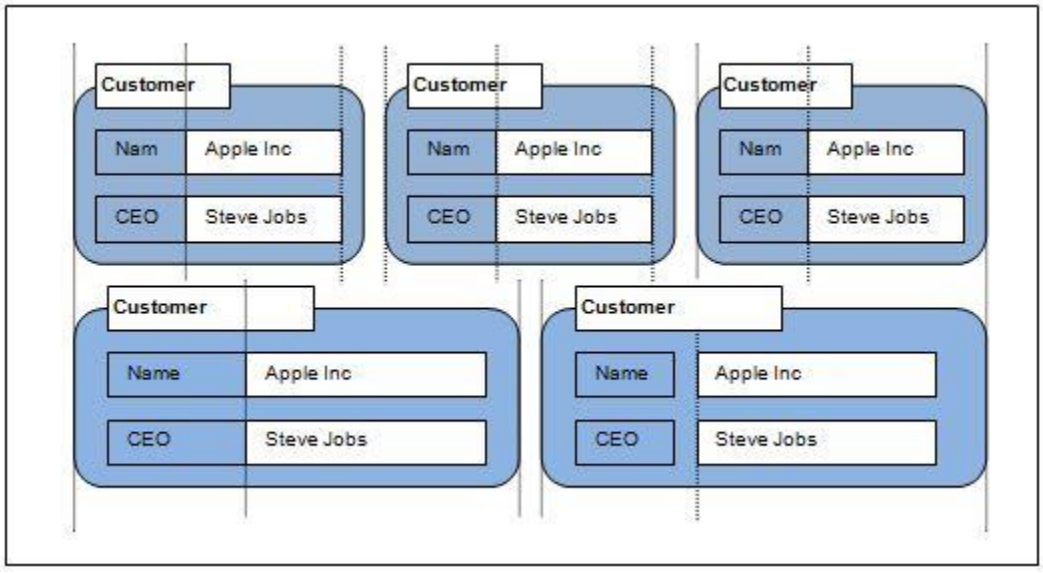
Tab Stop Types

The following table shows the types of tab stops:

Tab Stop	Description
Start	The first tab—always 0.
End	The last tab—always 12.
Label	The end of a label.
IntervalStart	Intervals consist of a label followed by two fields separated by a dash (“-“). The first field ends at the abstract tab stop intervalStart (the first field is between label and intervalStart).
IntervalEnd	The beginning of the second field in an interval.
Custom	Used to mark the different blocks in custom elements.
Published	User-defined tab stop.

The rule of alignment also applies to elements across containers, but only for rows that have the same number of containers or elements. For example, if there are two rows, and each row contains two groups, the elements inside the groups are aligned according to the rule of alignment. However, if there is only one group in the second row, the rule does not apply. The next two figures illustrate this.



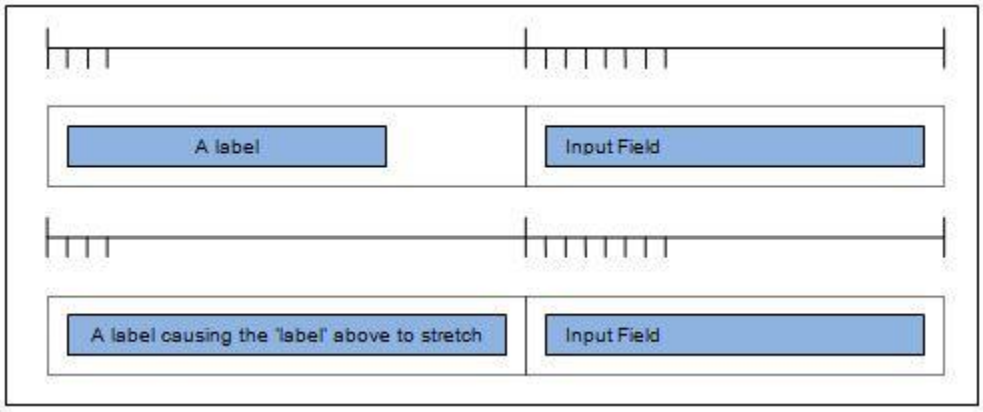


Exposing a tab stop means that the outside is affected by the sizes from the inside, but it also means that the outside can influence the sizes inside. For example, a label in a field may cause another label in another element to become wider, as shown in the next figure. The MDML layout algorithm always tries to find the optimal sizes for all of the elements in the dialog box.

Scope and Ruler Tags

You can modify the predefined alignment of containers and elements using the scope and ruler tags, which have the ability to expose or unexpose tabs.

Use scopes to create scopes inside the specification. This is very useful if you want to apply a property to a set of elements or containers. All scope tags accept other scope tags as content (which inherit the context of the parent scope).



Currently, you can only use rulers inside scopes, but in the future, other properties, such as styles, will be available.

A ruler tag specifies which tab stops of the contained elements should be exposed. However, ruler tags can also appear inside containers, or even inside elements. If they appear in a container, their properties are applied to the ruler of the container. If they are inside elements, the properties modify only that specific element.

Functions and Expressions

Layouts can depend on data and changes to the data as data is updated. This effect is achieved using the embedded Expression Language.

The Expression Language

The Expression Language is a little language used to express predicates, string functions, and other types of simple expressions.

Types and Literal Values

The Expression Language uses the Maconomy type system with syntax for defining literals for each type of value.

Type Name	Examples of Literal Value
Amount	amount(10.35) amount(-5)
Boolean	true false
Date	nulldate date(2010, 12, 24)
Decimal	10.3505 -2.5
Integer	15 -15
Popup	StyleAssociationType'Enterprise_Defined
String	'flaf' "flaf" "fl'af" 'fl"af' 'fl\'af' "fl\"af"
Time	nulltime time(13, 55) time(13, 55, 05)

You can use the backslash in string literals to escape the string delimiter. Date and Time values are constructed using the `date()` and `time()` constructor functions.

Variables

In addition to specifying literal values, expressions can refer to variables, such as `x` and `height`. The meanings of variables are determined by the context of the expression. It can be a function parameter or a field reference.

The Expression Language supports a special syntax for obtaining field references in containing (parent) panes. Such field references are prefixed with a number of instances of the `parent` keyword, separated with a dot:

```
parent.parent.someField
```

In this example, the expression refers to the value of the field `someField`, which is located in the parent of the parent of the current pane.

Unary Operators

The Expression Language supports a unary minus to indicate a negative numeric value, and the keyword `not` to indicate negation.

Operator Name	Operators	Operand Type
Negation	not	Boolean
Unary minus	–	Amount Decimal Integer

Binary Operators

The Expression Language supports the following binary operators, in order of precedence.

Operators	Operator Class	Operand Types	Type Coercion Applied
or	boolean	Boolean	No
and			
=	equality	Any	No
!=			
< <= > >= lt le gt ge	relational	Amount Date Decimal Integer String	Yes
inrange			

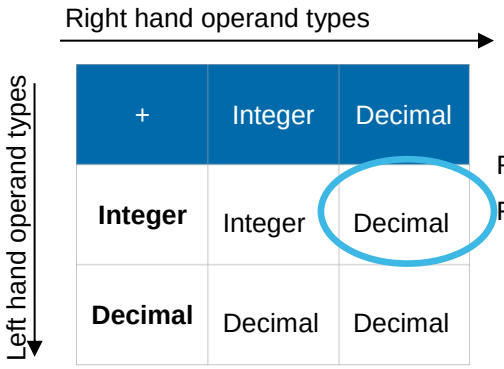
Operators	Operator Class	Operand Types	Type Coercion Applied
		Time	
like	string	String	Yes
+ -	arithmetic	Amount Decimal Integer String (only +)	Yes
* /			

For operators to which type coercion is not applied, the operands of operators must be of the same type. For operators to which type coercion is applied, type coercion follows the rules listed in the following tables.

Type Coercions

Some of the binary operators can be applied to operands of unequal types. This is governed by type coercion rules. The rules are listed in the following tables.

The tables are read in the following manner.



Operators: < <= > >=						
	Integer	**Decimal**	**Amount**	**String**	**Date**	**Time**
Integer	Boolean	Boolean	Boolean	-	-	-
Decimal	Boolean	Boolean	Boolean	-	-	-
Amount	Boolean	Boolean	Boolean	-	-	-
String	-	-	-	Boolean	-	-
Date	-	-	-	-	Boolean	-
Time	-	-	-	-	-	Boolean

Operators: + –					
	Integer	**Decimal**	**Amount**	**String (only +)**	**Any Other (only +)**
Integer	Integer	Decimal	Amount	String	-
Decimal	Decimal	Decimal	Amount	String	-
Amount	Amount	Amount	Amount	String	-
String (only +)	String	String	String	String	String
Any Other (only +)	-	-	-	-	String

Operator: *			
	Integer	**Decimal**	**Amount**
Integer	Integer	Decimal	Amount
Decimal	Decimal	Decimal	Amount
Amount	Amount	Amount	-

It is an error to multiply two amount values.

Operator: /			
	Integer	**Decimal**	**Amount**
Integer	Decimal	Decimal	-
Decimal	Decimal	Decimal	-
Amount	Amount	Amount	Decimal

No type of value other than amount can be divided by an amount.

Like Operator

The like operator is treated specially in the Expression Language. The second string operand defines a pattern, and the first string operand is matched against this pattern. The pattern can use the following wildcard characters.

Wildcard Character	Matches
*	Zero or more of any character
?	Exactly one of any character

You can use the backslash to escape the wildcard characters in patterns. The following are examples of valid like expressions, all of which evaluate to true:

```
'universe' like 'universe'
'universe' like '*verse'
'universe' like '???verse'
'universe' like '*ver*'
'universe?' like '*ver*\?'
```

Inrange Operator

The inrange operator is used with a square bracket range syntax, of the form:

```
value inrange [lower .. upper]
```

The expression evaluates to true if the value is between the lower and upper inclusive. For example:

```
5 inrange [1 .. 10]
```

An open endpoint is declared using the default value for the type.

Type Name	Default Value
Amount	amount(0)
Date	nulldate
Decimal	0.0
Integer	0
String	''
Time	nulltime

`zero` is the default value for numeric types. This means that the range `[0 .. 10]` is open-ended, such that the following expression evaluates to true:

-10 inrange [0 .. 10]

Upon evaluation, the inrange expression is expanded to this equivalent expression:

```
(value >= lower or lower = default) and (value <= upper or
upper = default)
```

default is the default value for the type of **value**.

If Expressions

Use the `if` expression to let an expression evaluate to two possible results, depending on a condition.

The `if` expression has the form:

`if` *expr1* `then` *expr2* `else` *expr3*

The `if` expression evaluates to expr2 if expr1 evaluates to true, and to expr3 if expr1 evaluates to false. If expr1 does not evaluate to a Boolean value, the `if` expression fails.

Unlike in many other computer languages, the `if` expression is an expression, so it can be combined with other expressions:

```
baseFee + (if isWeekend(jobDate) then weekendExtraCharge else
0)
```

Function Calls

The Expression Language supports function calls. You can define functions directly in layouts or using an extension point.

```
namespace:functionName(x, y, z)
```

You can omit the namespace from a function call, in which case the Maconomy Standard Functions namespace `maconomy` is assumed. Layout-defined functions are automatically added to the default namespace, and need not be qualified with a namespace. The following two expressions are equivalent:

```
dateField > currentDate()
dateField > maconomy:currentDate()
```

Error Handling with DefaultTo

Sometimes an expression cannot be evaluated. This can happen if:

- A variable or function is not defined.
- An operator is applied to operands of the wrong types.
- A function call fails.

To handle these situations, you can provide a fallback expression using the defaultTo operator in the following form:

expr `defaultTo` fallback

If the expression expr evaluates to a value without causing an error, that value is the value of the whole expression. If there is an error, the expression fallback is evaluated. For example:

```
phoneNumberFromWebService() defaultTo ''
```

You can provide a chain of fallbacks. The left-most successful expression is the value of the whole chain:

```
postCodeFromDirectory() defaultTo
postCodeFromLocalAddressBook() defaultTo ''
```

Expression End-User Localization

String literals in expressions can be localized to the end user's language. This is only useful if the evaluated expression will be shown somewhere in the user interface, for example, in a <Description> element in MDML or something similar. A string literal that should be translated is marked by a t-modifier:

```
  t'hello world' > (da_DK) 'hej verden'

                   (assuming that this term is in the
dictionary)
```

Enterprise Localization

A Maconomy system can be localized at the time of installation. This kind of localization is referred to as enterprise localization and happens only once in the lifetime of a system. Several parts of the database are localized during this localization.

Some pop-up literals are, for instance, localized from W to the enterprise locale, for example, on a da_DK system, the JobPopupType3'Work_Order literal will be localized to JobPopType3'Arbejdsordre. Similarly, several key string values are translated. The field values in the TheGroup field from SystemParameters are, for instance, translated, for example, on an en_US_MCS system the term "Job Cost" is translated to "Project Cost." Precise details about what is converted are described in the Application documentation.

The expression language supports enterprise localization by adding modifiers that ensure that specific strings and pop-ups are localized according to the system's enterprise locale. The following shows how the two cases mentioned above are specified and evaluated:

```
  JobPopupType'Work_Order'e > (da_DK)
JobPopupType'Arbejdsordre

  e'Job Cost' > (en_US_MCS) 'Project Cost'
```

A literal is, in other words, marked by an e-modifier to indicate that it should be enterprise localized. This feature is primarily intended for developers who create specifications spanning multiple installations in different languages.

The Expression Language in MDML

When expressions are used in an XML-based language such as MDML, you must consider restrictions imposed by the XML syntax. Due to certain questionable design decisions in the XML standard, characters such as '<', '>', '"', and '&' must be escaped in the following way: '<', '>', '"', and '&' when used as Parsed Character Data (PCDATA). To avoid these escape sequences, you can also escape the entire expression by putting it in a Character Data (CDATA) section, for example, '*<![CDATA[some expression]]>*'.

Expressions are either named or anonymous when used in MDML. The MDML Function element makes it possible to define a named expression that optionally takes a list of parameters. You define functions in the Define element of either a pane or a global definitions file. A named expression can be invoked by another expression. Anonymous expressions can be used on

various MDML elements, such as the condition attribute of a form conditional or a style. See the next page for a more complex example.

An important thing to note about expressions is that they are only evaluated when the pane receives new data. This means that the performance overhead of evaluating expressions is not incurred during editing, but rather on load, when the user submits or performs other actions that require data communication with the server. The only exception to this rule is in wizards, where the expressions are evaluated on every page shift. For more information about wizards, see Wizards.

Evaluating the expressions of a pane can have three possible outcomes: no change, a minor change, or a major change. The first two options have a relatively low performance overhead, since the GUI is reused. An example of a minor layout change is changing the foreground color of a label. The third option, which is a major layout change, occurs when a conditional evaluates differently from the previous evaluation, or when a font property is changed. These kinds of changes require a recomputation of the alignment; hence the whole GUI is replaced. For large panes, this could result in a noticeable delay in the GUI. MDML authors should take these performance considerations into account when designing data-dependent layouts and traffic lighting.

The following table provides an overview of the impact that different style properties have on a layout (for more information, see Style Properties).

Style Property	Impact	Comment
size	major	
anchor	major	
fontName	major	
fontSize	major	
foregroundColor	minor	
backgroundColor	minor	
bold	major	Except in switch-styles for the Calendar element (see Switch Styles). In that case, it is a minor change.
italics	major	Except in switch-styles for the Calendar element (see Switch Styles). In that case, it is a minor change.
underline	major	Except in switch-styles for the Calendar element (see Switch Styles). In that case, it is a minor change.
justify	minor	
decimalCount	minor	
zeroSuppression	minor	

Style Property	Impact	Comment
negativeNumberFormat	minor	

The following figure illustrates the two forms of expressions, as well as some applications. The parts marked by callouts are described in greater detail.


```
<?xml version="1.0" encoding="UTF-8"?>
<MDML version="0.1" xmlns="http://www.maconomy.com/ns/mdml">
 <Layout name="EmployeeControl" dialog="EmployeeControl">
  <UpperPane>
   <Define>
    <Function name="isZero" parameters="x" value="x = 0.00" />
    <Function name="billingPriceLessThanCost">
     <Value>
      <![CDATA[BillingPriceBaseCurrency < CostBaseCurrency]]>
     </Value>
    </Function>
   </Define>
   <Form name="aForm">
    <Group title="State of the Matter">
     <If condition=" billingPriceLessThanCost()">
      <FullLabel title="Bugger! We are losing money!" />
     </If>
     <ElseIf condition="isZero(CostPrice)
                        and BillingPriceCurrency > 0.00">
      <FullLabel title="YES! There's NO cost!" />
     </ElseIf>
     <Else>
      <FullLabel title="It's not free, but we earn money!" />
     </Else>
    </Group>
   </Form>
  </UpperPane>
 </Layout>
</MDML>
```

1. The Function element defines a named expression called *isZero, which* takes one parameter *x* and evaluates to *x = 0.00*. Functions that take more than one parameter should use a space-separated list in the parameters attribute. Parameters are not explicitly typed, and expressions that take parameters are therefore dynamically typed. However, functions are, by convention, of the Boolean type, so if the expression it contains evaluates to anything else, a null value is returned. The function is available in the scope associated with the pane.

   Currently, parameters are not supported, but they will be supported in the future.

2. The Function element defines a named expression called *billingPriceLessThanCost*. In this case, the value is defined in a nested Value element, rather than using the value attribute on the Function element. Since the expression contains a '<' character, the whole expression is escaped as CDATA. The names *BillingPriceBaseCurrency* and *CostBaseCurrency* refer to fields that must be accessible from this pane (they must be part of the environment when evaluating the expression).
3. The If-branch of the conditional introduces an anonymous expression that simply invokes the *billingPriceLessThanCost* expression. The name of the function is followed by parentheses to indicate that this is an invocation-named expression, rather than a misspelled fieldreference. Since all functions (at least in the foreseeable future) contain

an expression that evaluates to a Boolean value, they can safely be used as conditions in both conditionals and in conditional styles (an example of an expression that evaluates to a string type is *placeholder substitution* in the Description element).

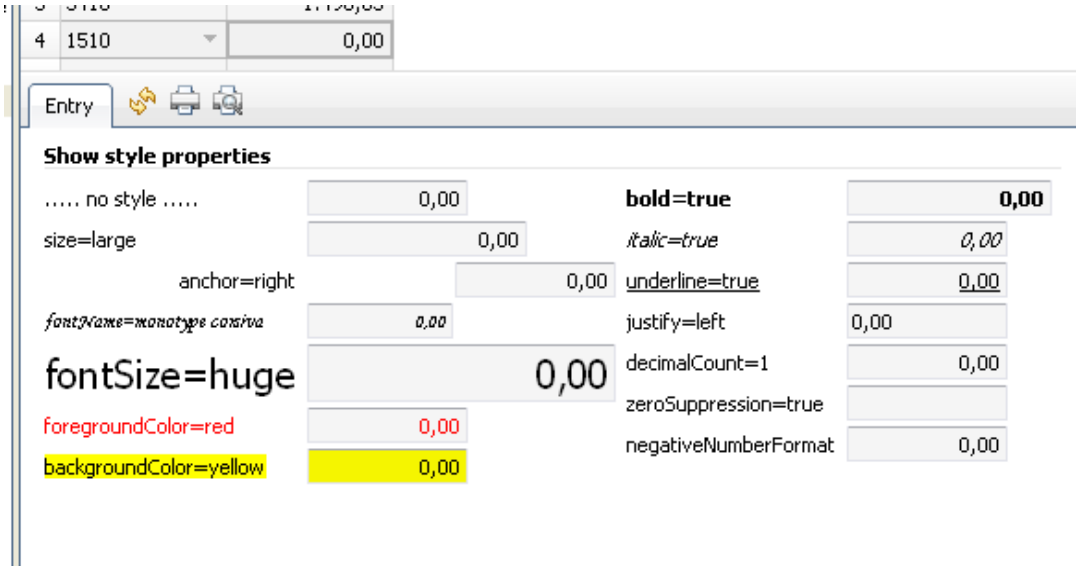
4. The ElseIf-branch of the conditional introduces an anonymous expression that internally invokes the named expression *isZero* function on the field *CostPrice, among others*. Take note that if a data change causes this conditional to evaluate to another branch, the whole layout must be recomputed. The contents of each branch may produce a different constraint set for the solver. This is called a *major* layout change. A minor layout change, on the other hand, is a less dramatic change, such as changing the font foreground color, and does not alter the internal constraint set.

Styles

You can style a layout in a number of different ways. You can create a general style for a whole layout and then create customized styles locally. You can also create customized styles for specific kinds of blocks or field types (a special style for all amount fields). Finally, you can specify styles as part of the pane definitions section, in which case they can be referred to by other styles (and customized).

Style Properties

Most style properties are supported by the different block types in elements. In time, groups will also support some style properties. Figure 10 shows the available style properties. Note that decimalCount and negativeNumberFormat are not yet supported by the widget library.



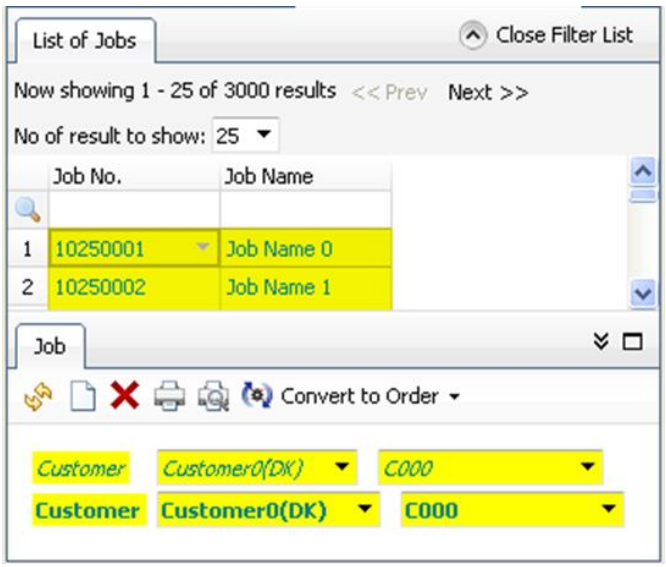
You can apply styles locally to a single element by attaching a nested style element. The style can apply to all blocks of the element or only to a certain kind of blocks, such as field or label, as shown in the following example.

```
<Group title="Applying styles">
    <!-- applying a style to all blocks of an element -->
    <Reference foreignKey="CustomerNumber_Customer">
      <Style bold="true"/>
    </Reference>
    <!-- applying a style to all Field blocks of an element -->
    <Reference foreignKey="CustomerNumber_Customer">
      <Style>
        <Field bold="true"/>
      </Style>
    </Reference>
</Group>
```

Applying styles in this manner can quickly become tedious, so other options exist. You can either refer to named styles defined on the pane level or attach a style higher up in the hierarchy (for a group, form, or even the entire MDML specification). This last option has a cascading effect, which means that if a style is defined for a group, all children of that group (its contents) inherit the group style. You can always override inherited style settings at any point in the hierarchy.

In the following example, all of the children of the Layout inherit a style with teal letters on a yellow background. This means that both filter and form and their contents will inherit this style. The Group in the form enriches this style by specifying that children of the group should be displayed in italics. Finally, the second Reference element overrides these settings by turning italics off and turning bold face on. This is done by referring to a reusable, named style that is defined in the Define section of the pane. Using the ref attribute on styles is equivalent to statically inserting the named style at all of the sites from which it is referred. The results are shown the following screen image.

```
<?xml version="1.0" encoding="UTF-8"?>
<MDML version="0.1" xmlns="http://www.maconomy.com/ns/mdml">
  <Layout dialog="Jobs" name="c_documentation4">
    <Style foregroundColor="teal"
           backgroundColor="yellow"/>
    <FilterPane>
      <Filter>
        <Columns>
          <Field field="JobNumber" />
          <Field field="JobName" />
        </Columns>
      </Filter>
    </FilterPane>
    <UpperPane>
      <Define>
        <Style name="NotItalicsButBold" italic="false" bold="true"/>
      </Define>
      <Form>
        <Actions all="true" />
        <Group>
          <Style italic="true"/>
          <Reference foreignKey="CustomerNumber_Customer"/>
          <Reference foreignKey="CustomerNumber_Customer">
            <Style ref="NotItalicsButBold"/>
          </Reference>
        </Group>
```

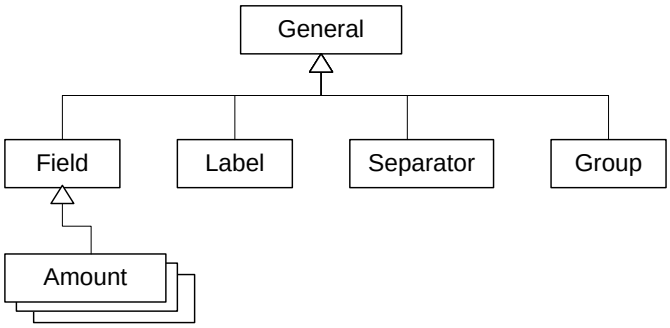


In addition, you can reference not only a single style, but a list of styles, which can be applied to each cell in a Grid. The following shows an example of a list style reference.

```
<UpperPane>
  <Define>
    <Style name="style1" backgroundColor="blue"/>
    <Style name="style1" backgroundColor="red"/>
    <Style name="style1" backgroundColor="green"/>
  </Define>
  <Form>
    <Group>
      <Grid>
        <Row title="Row" cells="Cell1 Cell2 Cell3"/>
          <Style ref="style1;style2;style3"/>
      </Row>
    </Group>
  </Form>
</ UpperPane>
```

Resolution

The styles of a layout are resolved on load and every time that the pane receives data. The resolution algorithm is invoked on every block in every element in the layout. The resolved style of a block depends on two main parameters, context and hierarchy. First, the resolution algorithm attempts to find a style definition that matches the context of a block. Contexts are arranged in a hierarchy, as shown in the following figure. The purpose of contexts is to allow the MDML author to define special styles for fields, labels, or even specific field types. Hence, a field block of type Boolean will first attempt to find a style defined for Boolean fields. If that fails, it will search for a style defined for the Field context, and finally for the general context.



Because a layout is a tree structure, the algorithm first looks for styles defined on the local element node, and then proceeds up the tree until it reaches the root node (the *MDML* element). A local style definition takes precedence over styles that are defined higher up in the hierarchy. However, you must note that context also takes precedence over the hierarchy.

To explain this concept in more detail, consider the following example:

The following layout has a whole set of “competing” style definitions at various levels in the hierarchy, which are directed at different contexts. The *UpperPane* defines a style with red foreground color for the general context and blue foreground color for the field context. The *Group* overrides this setting by stating that the foreground color should be yellow in the general context. Both labels of the *Field* and the *Reference* elements inherit this setting.

```
<UpperPane>
  <Style foregroundColor="red">
    <Field foregroundColor="blue" />
  </Style>
  <Form>
    <Actions all="true" />
    <Group>
      <Style foregroundColor="yellow" />
      <Field field="JobName" />
      <Reference foreignKey="CustomerNumber_Customer">
        <Style>
          <Field foregroundColor="purple" />
        </Style>
      </Reference>
    </Group>
  </Form>
```

The foreground color of the field block of the *Reference* element resolves to purple because the style for the field context is defined on the element. The field block of the *Field* element is a bit more complicated because it does not define a style for the field context. Because context takes precedence over hierarchy, the foreground color resolves to blue because the style of the *UpperPane* defines a style for the field context. The style definition on the *Group* is ignored, even though it specifies a foreground color, because the resolution algorithm attempts to find a style for the relevant context first.

All styles can also be attributed with a condition. If the condition of a style evaluates to false, that is equivalent to not having a style at all. Conditions on styles are evaluated on load and every time that the pane receives data.

Switch Styles

Switch styles are a special kind of style that is suited for more complex styling of elements. Currently, MDML supports only a *calendar switch style* for the Calendar element and fields that are of the date type. Use the calendar switch style to style individual cells in a calendar.

```
<Calendar field="DateVar">
  <Style>
    <Switch field="TimeSheetStatusVar">
      <Case char="0">
        <ToolTip text="Approved" />
        <Style foregroundColor="green" />
      </Case>
      <Case char="1">
        <ToolTip text="Due" />
        <Style ref="redStyle" />
      </Case>
    </Switch>
  </Style>
</Calendar>
```

This style relies on a status field that supplies a set of characters that correspond to individual dates. The switch style corresponds to the switch statement known in traditional programming languages—it defines a dynamic behavior, depending on the character given for each date. In the preceding example, the character “0” from the TimeSheetStatusVar results in the tooltip “Approved” and a green foreground color. The character “1” results in the tooltip “Due” and the named style “redStyle” defined elsewhere in the layout. If no status information is supplied for a given date (or if no switch style is defined for the calendar), every cell defaults to the style of the entire calendar element.

The value of the char attribute and the number of case branches in a switch statement are customizable. The contents just need to be aligned to whatever the application delivers.

Actions

Use action tags to specify which actions should be presented in each pane. Action tags can be alone, or they can be grouped. You can also group them as the available space decreases. In addition, you can present them as an icon, a title, a title plus an icon, or as a title with an icon that disappears if there is not enough space available.

There are two types of actions, standard actions and application actions. Standard actions include Create, Update, Delete, Refresh, Move, Print, and Print This. The other actions are the ones that are related only to the pane, such as “Submit Time Sheet,” “Convert to Quote,” or “Enable Employee Control.”

The outer-most tag for specifying actions is <Actions>. By setting its attribute “all” to true, all of the default pane actions are included. Otherwise, the desired actions must be declared manually.

After that, a <Exclude> tag may follow. This tag allows the exclusion of certain actions by specifying the actions to leave out. If they are standard actions, the specific tag for that action can be used. Otherwise, a normal action tag must be stated.

You can declare the <StandardActions> tag subsequently. Like the <Actions> tag, it offers the attribute “all,” but in this case, it refers to all possible standard actions for the pane. If <StandardActions> is not declared, and <Actions> has the attribute “all” set to true, the standard actions are included. Otherwise, if the <StandardActions> tag is specified, the attribute “all” also must be set to true to present all of the standard actions. In addition, this tag also offers some attributes that allow the modification of some of the default properties of the standard actions in a compact way, without having to declare it specifically in the actions.

You can group standard actions by putting them inside the tag <Group>. Otherwise, the actions are contained in a default group.

The next tags are the <Action> and <Group> tags. Use these tags to manually specify the groups, the actions to show, and their properties.

If you use either standard actions or normal action tags, the MDML engine interprets that as “the user takes control of which actions to present.” This means that the declared actions are presented with their specified properties first, followed by the rest, if the attribute “all” is true.

Wizards

Wizards are modal windows that can be associated with actions. When the action is invoked, a modal dialog window pops up, and the user is directed through a series of pages. Each page is specified as a normal form layout. The wizard is defined at the page level and can be referred to from various actions in the views of the pane. The following example shows the specifications for a wizard. Notice how the wizard is launched by invoking the *Create* action on the *Employee_with_wizard* form.

```
<!-- Create wizard -->
<Wizard name="CreateWizard" title="Wizard for creating a project manager">
  <Description value="EmployeeNumber=^1 and Initials=^2"
               arguments="EmployeeNumber Initials" />
  <Page form="EmployeeInfoPage" title="Personal Details" />
  <If condition="Country='US'">
    <Page form="EmployeeUSSpecificPage"
          title="Enter your favorite NBA team"/>
  </If>
  <Page form="EmployeePositionInfoPage"
        title="Enter details about your position"/>
</Wizard>

<Form name="Employee_with_wizard">
  <Actions all="true">
    <StandardActions all="true" createWizard="CreateWizard" />
  </Actions>
```

Wizards facilitate to a level of dynamic behavior: First, the individual pages are regular forms and can contain data-dependent expressions in the form of form conditionals and conditional styles. Second, the sequence of pages in a wizard can also be data-dependent. In the preceding wizard, the *EmployeeUSSpecificPage* is displayed only if the user's country is the United States. The expressions on both forms and in the wizard definitions are evaluated every time the user switches wizard pages. There is one exception to this: the history of previously visited wizard pages does not change. For example, if the user has reached the *EmployeePositionInfoPage* and changed his country from the United States to France, clicking **Back** will still take him back to the *EmployeeUSSpecificPage* because that was the last page that he visited.

By default, all pages on a given path of pages are considered mandatory. This means that the user must visit each page before he can complete the wizard. You can override this for individual pages by specifying the value of the `mandatory` attribute. The user can then skip a page that is not considered mandatory and still complete the wizard.

For example, you have a three-step wizard where page three is specified as *not* mandatory. This means that after the user has viewed page one and page two, which are mandatory pages, he can complete the wizard without viewing page three. If page three is mandatory, the user must complete this page before he can complete the wizard. After the user has viewed all mandatory pages, he can view previous pages and still complete the wizard.

You can tag pages as mandatory by default on the wizard tag by using the `mandatory` attribute. If you leave this attribute out, the tag is considered to have the value `true`. You can include arbitrary Boolean expressions in the `mandatory` attributes.

Triggers

Triggers are a specific functionality of links, actions, wizards, and wizard pages that is used to set the default values for fields. You can define triggers in a <Define> block of the pane in a <Trigger>-tag, and they can include any number of assignments, validations, or refresh steps in any order. For example, when a trigger is assigned to a link, clicking that link runs and performs all of the trigger steps. If the trigger fails, an error message is displayed, and the link does not open. If the trigger runs successfully, the link opens. The following is an example of a trigger definition.

```
<!-- Trigegr definition -->
<Trigger name="TestTrigger" condition="true">
  <Assign source="WeekNumber" value="45" />
  <Check condition="WeekNumber=45">
      <Error template="The week number is incorrect"
             focusField="WeekNumber" />
  </Check>
  <Refresh type="workspace" />
<Trigger>
```

Grids

Availability Overview - grid version

	Employee Information		Planning Time		Load %		
	No.	Name	Week	Total{In total}	Week	Total{In total}	Keep
First	12	Anders Hanse	42,00	0,00	0,00	0,00	No
Second			0,00	0,00	0,00	0,00	No

A grid is a table structure that can appear inside a form group. It has three different kinds of rows: headers, regular rows, and footers. Each of these rows can have a number of cells, similar to a normal table. The first cell in each row is a label that stores the title for the row. The following is a simple example based on the preceding DetailedPlanning layout.

```
<Grid>
  <Header style="boldHeader"
    cells="||Employee Information||
           ||Planning Time||
           ||Load %||
           _"/>
  <Header style="simpleHeader"
    cells="|No.|  |Name|
           PlanningUnitVar  |Total{In total}|
           PlanningUnitVar  |Total{In total}|
           |Keep|"/>
  <Row title="First"
    cells="Availability1EmployeeNumber  Employee1NameVar
           PlanningTimeFirstUnit1Var    PlannedInShownUnit1Var
           LoadPercentageFirstUnit1Var  LoadPercentageInShownUnit1Var
           Keep1"/>
  <Row title="Second"
    cells="Availability2EmployeeNumber  Employee2NameVar
           PlanningTimeFirstUnit2Var    PlannedInShownUnit2Var
           LoadPercentageFirstUnit2Var  LoadPercentageInShownUnit2Var
           Keep2"/>
</Grid>
```

Each row can have a title (which can be based on a value or a field) and a mandatory cell definition. The cell definition in the attribute *cells* is a concise way of specifying the structure of the grid.

The cell definition contains a set of cells that are either fields, labels, or empty. Fields can be annotated with a mandatory (*) or a closed (-) marker. You can cause labels to span multiple columns by surrounding them with multiple vertical bars. Empty cells are simply labels with no titles. Although it is currently supported only in the syntax, a cell will have an optional secondary field in the future, such as adding a unit to a field.

The following shows the eBNF syntax for the cell definitions.

```
definition  =    ws* ( cell ( ws+ cell )* )?
cell        =    primary ( '+' secondary )?
primary     =    field | label | empty
secondary   =    field | label
field       =    ( closed | mandatory )? qualifiedid
closed      =    '-'
mandatory   =    '*'
label       =    quote+ title quote+
empty       =    '_'
qualifiedid =    a valid id
title       =    any string, escape sequences are \| and \'
quote       =    '|' | '"'
ws          =    whitespace
```

Given the preceding syntax, the following are some examples of cell definitions and the way in which they should be interpreted:

"||Employee Information|| ||Planning Time|| ||Load %|| _"

means three labels, each of which spans two columns, and an empty cell that spans one column at the end.

*"-EmployeeNumber *EmployeeName |||Comments||| |Keep|"*

means a closed field (EmployeeNumber), a mandatory field (EmployeeName), a label that spans three columns, and a label that spans a single column.

You can style grids in the same manner as other elements. In the preceding example, individual rows are styled.

Currently, it is not possible to specify a style for individual columns or cells.

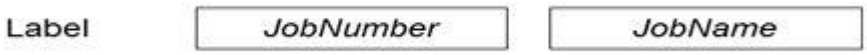
Best Practices

This section contains a brief list of some best practices that MDML authors are encouraged to follow.

Send an e-mail if you think that some of these practices are wrong, misleading, or simply brilliant.

General Tips

- Avoid using custom elements. Alignment is not guaranteed for such constructs.
- Never abuse an element if the semantics do not fit. For instance, the <Unit> element consists of a title and two fields, but one of them is intended for unit values. Do not use this for the following situation, because neither JobNumber nor JobName are measured in units.



- Provide explicit titles only if the default title does not fit and cannot be made to fit.

If possible, avoid making detail adjustments. If you find that an adjustment or a custom element configuration is needed several times, let your Maconomy representative know. There may be a need to create elements or improve the engine.

Improving MDML Filtering Capabilities

When working with large data sets, such as GL or Job Entry transactions, you might trigger long-running queries. For instance, if you select the “All” filter option and view all transactions in GL, then that query can in the worst case take minutes to complete (if indices are not setup properly). If you decide to abort such a query, you may decide to close the workspace or perhaps even the entire WSC. This is a poor “solution” if metadata is enabled since the next time the WSC is started, it attempts to re-establish the last query by inspecting what was the last piece of metadata about this user. This means that you end up in the same situation you were trying to escape when aborting earlier.

To address this problem, we have introduced a set of filter syntax extensions. These extensions do not improve performance per se but strive to avoid the situation described above. Specifically, the new metadata attribute allows the MDML author to specify that certain filter options should not be stored in metadata, such as, do not store the “All” filter option. A more fine-grained level is provided with the sorting and filtering attributes which allow the MDML author to disable sorting and filtering on certain columns. This is particularly useful for non-indexed columns where operations such as filtering and/or changing the sort order can trigger very long running queries.

This feature consists of three attributes that are added to many of the existing MDML elements, detailed below. Customize your MDML layout as needed with the following attributes:

- **metadata** – Used to control whether metadata about user operations such as selecting a particular filter option is stored in metadata.
- **sorting** – Used for disabling sorting on column.
- **filtering** – Used for disabling filtering on column.

These attributes are expressions.

Metadata

Set **metadata** to an expression that resolves to *false* to disable the use of metadata. With this restriction, metadata will not be used to store the state of selected user elements.

Example

In the following example, we disable metadata for users who do not have a Sales role:

```
<Option metadata=" hasRole('Sales')" … />
```

Additionally, you can disable the restoration of the selected filter option and column sorting when reopening the workspace. This means that when the metadata attribute resolves to *false* for **<Option>** and **<Field>**, metadata is disabled. Specify this directly on the elements, or on the parent elements. Since child elements inherit values from the parent, if you disable elements in the parent, you must enable the selected child elements or disable only in the selected child elements .

Attribute **metadata** is available for:

- disabling storing filter option: <Layout>, <FilterPane>, <Filter>, <ControlBar>, <Selection>, <Option>
- disabling storing sorting: : <Layout>, <FilterPane>, <Filter>, <Columns>, <Scope> (in <Filter> and <Columns>), <Field> (in <Columns>), <UnitField> (in <Columns>)

Attribute name	Type	Usage
metadata	Expression (Boolean)	Controls if the metadata should be disabled for this element or children of this element.

Sorting

Set **sortable** to *false* to disable sorting for the filter column.

- Use the **sortable** attribute on <Columns> (parent) to disable for all (child) columns
- Use the **sortable** attribute on <Field> or <UnitField>> to disable for a single column, or conversely, to enable a single column if it was disabled via the <Columns> specification.

Example

In the following example, we disable sorting for all columns (sortable="false") and then enable it only on JobNumber column:

```
<Columns sortable="false">
…
<Field source="JobNumber"      name="NotInvoicing_JobNumber"   sortable="true" />
…
</Columns>
```

After you set **sortable** to *false*, clicking on a column header no longer initiates sorting, and now there is no action.

Currently, there are no indicators to let you know the sorting is disabled for a column.

Attribute **sortable** is available for: <Columns>, <Scope> (in <Columns>), <Field> (in <Columns>), <UnitField> (in <Columns>)

Attribute name	Type	Usage
sortable	Expression (Boolean)	Controls if sorting should be disabled for this element or children of this element.

Filtering

Set **filterable** to *false* to disable filtering for the filter column.

- Use the **filterable** attribute on <Columns> (parent) to disable for all (child) columns
- Use the **filterable** restriction on <Field> or <UnitField>> to disable for a single column, or conversely, to enable a single column if it was disabled via the <Columns> specification.

Example

In the following example, we disable filtering for all columns (filterable=”false”) and then enable it only on JobNumber column:

```
<Columns filterable ="false">
…
<Field source="JobNumber"      name="NotInvoicing_JobNumber"   filterable ="true" />
…
</Columns>
```

Disabled columns display a greyed filtering field so that the user cannot enter filter restriction.

Attribute **filterable** is available for: <Columns>, <Scope> (in <Columns>), <Field> (in <Columns>), <UnitField> (in <Columns>)

Attribute name	Type	Usage
filterable	Expression (Boolean)	Controls if filtering should be disabled for this element or children of this element.

MMSL

Introduction

Use this document to learn about tags and associated attributes related to MMSL.

Attributes of a given tag are described using a table like the following. An empty table means that there are no attributes.

Attribute Name	Type	Usage
Attr1	*Type of attr1*	This attribute is used to…
Attr2	*Type of attr2*	Indicates…

Attributes refer to any of the types listed in the following table.

Attribute	Description
Expression	An expression that evaluates to either true or false (a predicate). This expression can use the Maconomy standard functions, such as envVar('user.info.username').
Boolean	A Boolean attribute (true or false).
Key	A string that is case-insensitive and NEVER exposed to end users. Use this for references (internally or from other parts in the spec/other specs).
Display	A string that is meant to be displayed to an end user (and which is therefore localized). Can never be used for any reference.
Id	A case-sensitive string for referencing items in environments that are not controlled by Maconomy. This is NEVER exposed to an end user.
Version	A number of the format “xx.yy,” where xx is the major version, and yy is the minor version.

Example Menu

The following code shows an example of MMSL specification.

```
<?xml version="1.0" encoding="UTF-8"?>
<MMSL xmlns="http://www.maconomy.com/ns/mmsl" version="0.4">
  <Menu>
    <Group title="My Workspaces" icon="iconFolder.png">
      <Workspace source="DraftInvoiceEditing" title="My Drafts"
                 icon="workspace1.png" />
    </Group>
    <Group title="Jobs">
      <Workspace title="Jobs" source="Jobs" />
      <Workspace title="Job Batch" source="JobBatch" />
      <Workspace title="Opportunities" source="Opportunities" />
    </Group>
    <Group title="Time &amp;amp; Expenses">
      <Workspace name="registerTime" title="Time and Expense Registration"
                 source="TimeAndExpenseRegistration" />
    </Group>
  </Menu>
  <Auto>
    <!-- Open "Time & Expenses" automatically every Monday
         with a custom "Time to register"-title.           -->
    <If condition="intWeekDay( currentDate() ) = 1">
      <Workspace ref="registerTime" title="Time to register" />
    </If>
  </Auto>
</MMSL>
```

General Tags

<MMSL>-tag

This is the general container tag for the menu specification. It has two main parts.

- The Menu-section of workspaces, which can be divided and subdivided into groups.
- The Auto-section, which lists workspaces that should be opened at startup.

Attribute Name	Type	Usage
version	Version	Indicates the current version of this menu specification.

<Menu>-tag

This is the container for the workspaces that are part of the menu. These workspaces can be conditionally included using the <If><ElseIf><Else>-tags and divided into groups using the <Group>-tag.

Attribute Name	Type	Usage
-	-	-

<Auto>-tag

This is the container for the workspaces that are launched during startup. These workspaces can be conditionally included using the <If><ElseIf><Else>-tags. You can only start a workspace automatically if it is included in the actual menu. If a workspace is included in the menu, and it is unconditionally referred to in the Auto-section, it only starts automatically if condition X is satisfied.

Attribute Name	Type	Usage
-	-	-

<Group>-tag

This is the primary visual grouping construct. Groups can contain workspaces and nested groups. The members of a group can be included conditionally using the <If><ElseIf><Else>-tags.

Attribute Name	Type	Usage
title	Display	The title indicates the common feature of the group's members. This attribute is required and should not be left empty.
icon	Id	The resource identifier of the icon. If no icon is specified, a default icon is used.

<If><ElseIf><Else>-tags

This is a non-visual grouping construct that can be used to either include or exclude the members of this tag. The condition attribute can be simple true/false literals or complex expressions, such as username()='Administrator'. The evaluation of conditions takes place once during startup. The user must sign out and restart the client to reevaluate these conditions.

Attribute Name	Type	Usage
condition	Expression	The condition that must be satisfied for the children of this tag to be included. This attribute is required on the <If>- and <ElseIf>-tags and prohibited on the <Else>-tag.
default	Boolean	The default attribute indicates a fallback value if the condition cannot be evaluated without errors. This can be caused by either a syntactically invalid condition or an error in the client environment, such as invoking a function that is not available.

Menu-Section

<Workspace>-tag

This tag specifies the presence of a workspace item in the menu. A workspace item represents a workspace as defined in MWSL. However, the appearance of a workspace in the menu does not necessarily imply that this workspace can be opened. You must also take the access specification in MCSL into account to fully determine if a workspace item in the menu is actually accessible.

Attribute Name	Type	Usage
title	Display	The visible title of this workspace. Since the same workspace can appear several times in a single menu, the title is required and should be given some thought.
icon	Id	An optional icon reference to an icon that is rendered next to the workspace title. If no icon is specified, a default icon is used.
source	Key	The workspace that is represented by this menu item. This refers to a corresponding MWSL file. The file extension (‘.mwsl.xml’) should be left out.
name	Key	An optional name for the item. The name can be referred from the Auto-section. This name should be unique in the menu. If nothing is specified, the value of the source attribute is used.
pluginId	Id	The id of the plug-in implementing the workspace implementation that is used to render this workspace. The default is to render a workspace as an Eclipse editor.

Auto-Section

<Workspace>-tag

This indicates that a workspace item in the menu should be opened automatically when the client starts. For more information, see <Workspace>-tag.

Attribute Name	Type	Usage
title	Display	The visible title of this workspace. The title overrides the title specified in the menu. If nothing is specified here, the title from the menu is used.
icon	Id	An optional icon reference to an icon that is rendered next to the workspace title. If no icon is specified, nothing is shown.
source	Key	The workspace that is represented by this menu item. This refers to a corresponding MWSL file. The file extension (‘.mwsl.xml’) should be left out.
ref	Key	Reference to a named workspace from the menu-section.
pluginId	Id	The id of the plug-in implementing the workspace implementation that is used to render this workspace. Currently, the default is to render a workspace as an Eclipse editor.

MNSL

Quick Reference

This section is a quick reference to Deltek Maconomy MNSL. The MNSL language is used to define a specification of a notification. The notifications are calculated and presented to a user when he or she starts the client.

A notification specification can be seen as an extension to a Notification Type specified in the application. You use the application to set up Notification Types and assign them to users. The notification specification extends this setup with information about how the notification should be used in the Maconomy client.

This document briefly describes all of the tags and associated attributes and includes an example.

Attributes of a given tag are described using a table like the following.

Attribute Name	Type	Usage
Attr1	Type of attr1	This attribute is used to…
Attr2	Type of attr2	Indicates…

The referred types of the attributes can be one of the following:

Attribute Type	Description
Expression	An expression in the general Expression Language (EL).
Key	A string that is case-insensitive and never exposed to any end user. Used for references (internally or from other parts in the spec/other specs).
Display	A string that is meant to be displayed to an end user (and that is therefore localized). Can never be used for any reference.
Entity	A string that is case-insensitive and can contain ":" for separation of a universe name.

Notification Tags

<MNSL>

This refers to the general container tag for the notification specification.

Attribute Name	Type	Usage
Version	(Version Type)	Indicates the current version of this layout specification. A number of the format **xx.yy** where **xx** is the major version and **yy** is the minor version.

<Notifications>

This specifies a notification, which consists of a query and a link. The query defines how data that the user should be notified about should be selected and restricted. The link defines how data that the user should be notified about is opened in the client.

In Maconomy 2.5.1 (MNSL version 0.8) it is also possible to specify one or more remove-with rules which specify when notifications of this type should be automatically removed.

Attribute Name	Type	Usage
name	Key	Indicates the name of the notification. This name must match a notification type created in the application. Notifications are only calculated for users who have a given notification type assigned to them. Can include namespace prefix.
title	Display	An optional value that indicates the title of the notification shown in the Notification view. The title specified on the notification type is used as the default.

<Query>

This specifies the query that is used for calculating the notification. The query that finds notifications for a user is based on a list of fields that are selected on the entity, the entity, and the restriction on the entity. The list of fields is derived from information in the <Link>-tag.

> **Note:** Omitting this part from the specification will exclude the relevant notification from asynchronous calculation.

Attribute Name	Type	Usage
entity	Entity	Indicates a name of an entity or a universe.

<Restriction>

This specifies a restriction on the preceding entity.

Attribute Name	Type	Usage
condition	Expression	An optional one-line expression that restricts the data selected on the entity.

<Condition>

This specifies an optional multi-line expression for the restriction on the preceding entity.

<Link>

This specifies how a notification should be shown to a user and opened in the client. It represents a link to a workspace, a `<Description>` that defines how the link is presented in the client and, if necessary, additional parameters such as waypoints and target.

`<Waypoint>` is a pane in a path that is taken to navigate to the final `<Target>` pane in a link. Both `<Waypoint>` and `<Target>` represent an item in the link path and have the same attributes and elements.

Attribute Name	Type	Usage
workspace	Key	Indicates a name of a workspace. When a user clicks on the notification this workspace is opened. Can include namespace prefix.
icon	Resource	An optional value that specifies an icon for the notification shown in the notification view. Also appears in the tab for the workspace that is opened when a user clicks the notification.

<Description>

This specifies how a notification should be shown to a user and opened in the client.

Attribute Name	Type	Usage
template	(Placeholder String)	The template string that is used as the basis for placeholder substitution. This attribute is required. The format for placeholders is **^** followed by an integer that represents the index of the argument in the arguments list.
arguments	Expression List	A whitespace delimited list of arguments. The arguments can consist of fields from the universe / query.

<Waypoint>, <Target>

<Waypoint> is a pane in a path that is taken to navigate to the final `<Target>` pane in a link. Both `<Waypoint>` and `<Target>` represent an item in the link path and have the same attributes and elements.

Attribute Name	Type	Usage
pane	Key	Pane name for waypoint or target.
field	Key	Field name of the field that should acquire focus when opening the link.

<Restriction>

The <Restriction> restricts the record in the opened <Waypoint> or <Target> pane based on the <Match>. If the <Waypoint> or <Target> is a filter, a radio button is added to the filter with a matched restriction.

Attribute Name	Type	Usage
Title	Display	Indicates the title of the option that defines the restriction in the filter.

<Match>

The match indicates a record that satisfies the field and value selection and is used to set a focus on that record.

Attribute Name	Type	Usage
queryField	Key	The name of a field from the universe / query.
Field	Key	The name of a field from Waypoint or Target pane.

<RemoveWith>

The remove-with rule defines a selection criterion for removing already created notification records. The notifications will be automatically removed when a record is created in the specified database entity with field values matching the values stored in the notification.

The field values to match against are extracted from the <Query> entity record when the notification is created, and are stored together with the <Description> template arguments.

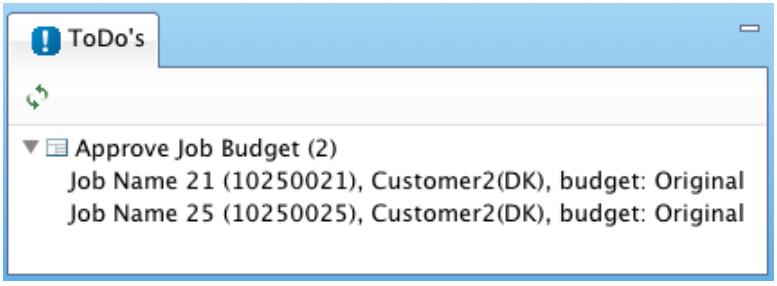
Attribute Name	Type	Usage
entity	Key	The name of a database entity.
fields	Key	A whitespace delimited list of field names from the entity.

Example

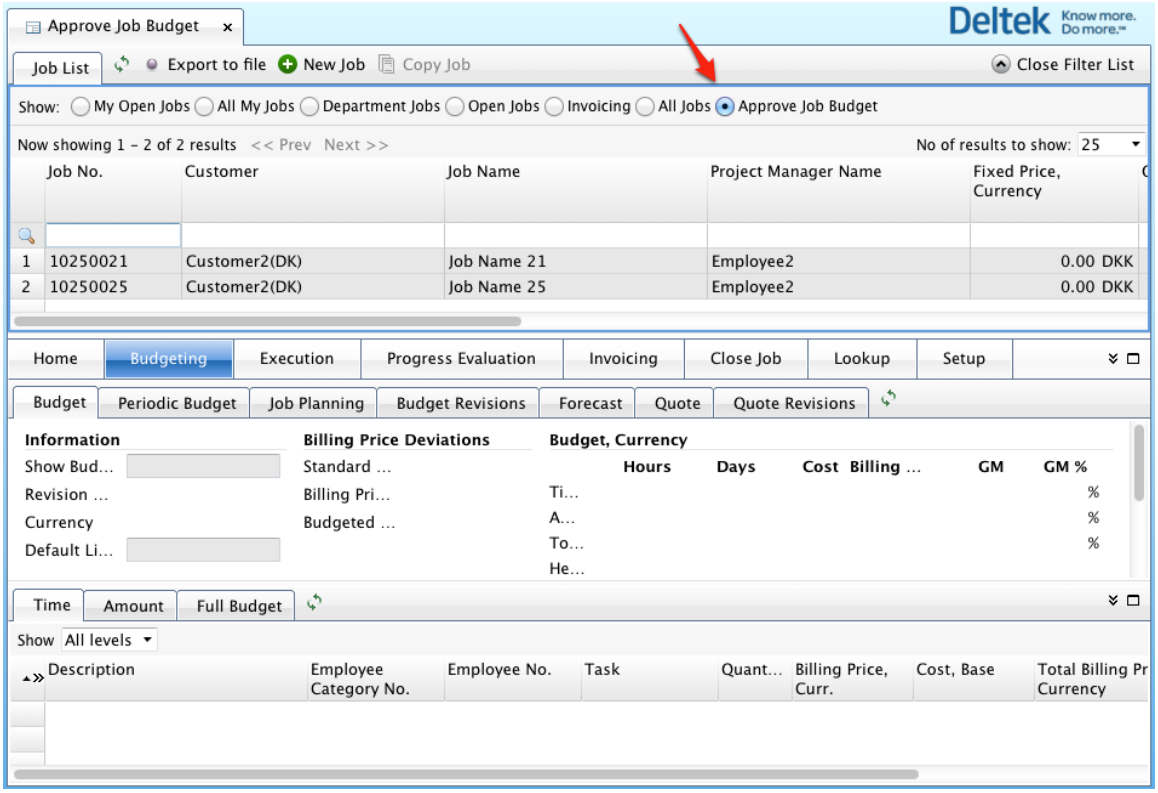
The following example shows a notification specification for the notification “ApproveJobBudget.” The notification selects data from the universe “NotificationUniverses::JobCost::ApproveJobBudget” and restricts the query to return only jobs for which the current user is the project manager.

```
<?xml version="1.0" encoding="UTF-8"?>
<MNSL xmlns="http://www.deltek.com/ns/mnsl" version="0.2">
  <Notifications name="ApproveJobBudget">
    <Query entity="NotificationUniverses::JobCost::ApproveJobBudget">
      <Restriction condition="userEmployeeNumber() = JobHeader.ProjectManagerNumber" />
    </Query>

    <Link workspace="Jobs">
      <Description template="^3 (^1), ^4, budget: ^2" arguments="JobHeader.JobNumber
                                                JobBudget.BudgetType
                                                JobHeader.JobName
                                                JobHeader.Name1" />

      <Waypoint pane="Jobs_Filter">
        <Restriction title="Approve Job Budget">
          <Match queryField="JobHeader.JobNumber" field="JobNumber" />
        </Restriction>
      </Waypoint>

      <Target pane="JobBudgets_Card">
        <Restriction title="Job Budget">
          <Match queryField="JobHeader.InstanceKey" field="InstanceKey" />
        </Restriction>
      </Target>

    </Link>
  </Notifications>
</MNSL>
```

When the notification returns data an entry appears in the ToDo’s view in the client as seen in the following figure.



When a user invokes one of the links, the workspace is opened, and a new option appears in the filter (indicated by the arrow in the following screenshot).



Installation

The notification specifications (mnsl files) must be installed on the Maconomy server in **CustomizationDir » Custom » Notifications**. A notification type with the same name is created in Maconomy and assigned to a user.

MWSL

Quick Reference

This section is a quick reference to MWSL. It briefly describes all of the tags and associated attributes.

Attributes of a given tag are described using a table like the following.

Attribute Name	Type	Usage
Attr1	*Type of attr1*	This attribute is used to…
Attr2	*Type of attr2*	Indicates…

The referred types of the attributes can be one of the following.

Attribute	Description
Boolean	A Boolean attribute is true or false.
Expression	An expression in the general Expression Language (EL).
Key	A string that is case-insensitive and never exposed to an end user. Use it for references (internally or from other parts in the spec/other specs). A key must be in the form: (_a-zA-Z)(_a-zA-Z0-9)*, for example: “myName1.”
NsKey	A name-spaced string that is case -insensitive and never exposed to any end user. Use it for reference (internally or from other parts in the spec/other specs). An NsKey must be in the form: (Key:)?(Key), for example: “myNS:myName2” or “myName3.”
Display	A string that is meant to be displayed to an end user (and that is therefore localized). It can never be used for reference.
Id	A case-sensitive string that is used to reference items in environments that are not controlled by Maconomy. It is never exposed to an end user.
Enumeration	An enumerated set of string values.

Preamble Tags

<Definitions>-tag

Use the <Definitions>-tag to contain various forms of definitions that are available when parsing the workspace. Only functions can be specified.

There are no attributes for the <Definitions>-tag.

<Function>-tag

The <Function>-tag is embedded within the <Definitions>-tag. This tag declares a named function that can be accessed in expressions throughout the workspace.

Attribute Name	Type	Usage
name	Key	The name of the function. Use this name to refer to the function from within expressions.
type	Enumeration	This is the *return type* of the function. The default type of expression is Boolean, but functions of other types can be created.
value	Expression	This is the *function body*. The function body is itself an expression and may refer other functions that are declared above or functions that are available as “built-in” functions or functions that are “plugged in.”
Parameters	Key list	Here you specify the formal names of the parameters, (the *input parameters)* for the function. When invoking a function, you must invoke it with as many parameters as are declared in this list.

Example

```
<Function name=”mult”
          type=”real”
          value=”x * y”
          parameters=”x y”/>
```

This defines a “mult” function that returns a value of the type real. It takes two parameters, “x” and “y,” and returns these two values multiplied.

Examples of invoking this function are:

```
2 + mult(2, 17.3)

mult(mult(6,4), mult(2,6))
```

Component Tags

<Filter>-tag

The <Filter>-tag specifies the presence of a filter pane at this level in the workspace. The filter pane can contain bindings to other panes, and these can be organized in different compositional structures (Expansions, Assistants). “Initial filters” are compactable. An initial filter is a filter that either has no parents, or only has initial filters as parents. Also, an initial filter must have either no siblings or filter-pane siblings only. In this case, the filter is treated as part of “the initial data finding.” Such filters (tab rows) can be “compacted,” thereby only showing the selected filter in a very compacted mode. Compacting is not supported by the client.

A filter that is not bound by any other pane can show any record that is referred by the where clause of the default cursor and corresponding dialog.

Attribute Name	Type	Usage
source *Not available if filter is bound using a <With>-binding*	NsKey	Indicates the name of the dialog that is used to define the content of the filter. For example, if the dialog is “InvoiceSelection,” the filter shows jobs (this is the entity that is shown in the upper part of that dialog). It also unconditionally applies the filtering that is defined by the where-clause of the upper pane cursor of that dialog. (In this case, meaning that it is not possible to see jobs that are not able to be invoiced.)
Title	Display	The title of the tab that represents access to the filter pane. If not specified, the title of the associated MDML layout is used, or if that title is undefined the default title that is specified for that FilterPane in DDL is applied.
Layout	NsKey	Name of the MDML file that contains the layout that should be applied for this pane (excluding “.mdml.xml”). If this value is an empty string, no layout is applied, and a blank pane is displayed. (This is useful to quickly make a mockup workspace.) If this attribute is not specified, the layout called *dialogName*.mdml.xml is used. If the pane is bound by a <With>-binding, the layout value from its parent is used as the default when no value is explicitly specified.
view	Key/Expression	Name of the layout view to apply. In a layout file, each type of pane can have many different layout *views* defined. Laying out a pane requires *one* view. The first view that has the name specified by this attribute is applied. If no value is specified, the first view (for the relevant pane) is applied. It is possible to give *arguments* to a view (provided that the view is declared with corresponding parameters in MDML). In this case, you can add a

Attribute Name	Type	Usage
		number of expressions in a comma-separated list, for example: `view="myView(true, 33 - 2, currentDate())"` If the pane is bound by a <With>-binding, the view value from its parent is the default if no value is explicitly specified.
Name	NsKey	The internal name of the pane in *this* workspace. Currently unused, but may eventually be used to address a specific pane in a workspace from, for example, links, notifications, and so on.
pluginId	Id	The ID of the plug-in that implements the filter pane to apply. If left unspecified, the default filter-implementation is used.

<Card>-tag

The <Card>-tag specifies the presence of a card pane at this level in the workspace. The card pane can contain bindings to other panes, and these can be organized in different compositional structures (for example, Expansions and Assistants).

A card that is not bound by any other pane shows a *random record* from the set of records identified by the where clause of the default cursor of the corresponding dialog. This is useful for dialogs that are capable of showing only one record, such as SystemInformation. The Card tag is used to reference normal card-parts as well as "action cards."

Attribute name	Type	Usage
source *Not available if card is bound using a <With>-binding*	NsKey	Indicates the name of the dialog that is used to define the content of the card. For example, if the dialog is "TimeSheets," the card may show time sheet headers plus variables that are defined for the card part of the dialog "TimeSheets."
Title	Display	The title of the tab that represents access to the card pane. If not specified, the title of the associated MDML layout is used, or if that title is undefined, the default title specified for that CardPane in DDL is applied.
layout	NsKey	Name of the MDML file that contains the layout that should be applied for this pane (excluding ".mdml.xml"). If this value is the empty string, no layout is applied and a blank pane is displayed. (This is useful to quickly make a mockup workspace.) If this attribute is not specified, the layout called *dialogName*.mdml.xml is used.

Attribute name	Type	Usage
		If the pane is bound by a <With>-binding, the layout value from its parent is the default if no value is explicitly specified.
View	Key/Expression	Name of the layout view to apply. In a layout file, each type of pane can have many different layout *views* defined. Laying out a pane requires *one* view. The first view that has the name specified by this attribute is applied. If no value is specified, the first view (for the relevant pane) is applied. It is possible to give *arguments* to a view (provided that the view is declared with corresponding parameters in MDML). In this case, you can add a number of expression in a comma-separated list, for example: `view="myView(true, 33 - 2, currentDate())"` If the pane is bound by a <With>-binding, the view value from its parent is the default if no value is explicitly specified.
Name	NsKey	The internal name of the pane in *this* workspace. It is currently unused, but may eventually be used to address a specific pane in a workspace from, for example, links, notifications, and so on.
pluginId	Id	The ID of the plug-in that implements the card pane to apply in this instance. If left unspecified, the default filter-implementation is used.
hidden	Expression	If the expression evaluates to true, this card (and all of its children) are not visible in the workspace; they are hidden. This means they may as well not be there. The expression is evaluated in the context of the current card. This means that all fields, functions, and expressions that are exposed to the card can be used for calculation on the visibility of the card. For this particular attribute, three special functions are available: hasNoSeed() The `hasNoSeed()` function is true if the pane does not have any data, *not even "0 rows."* Basically, this means that the underlying dialog pane cannot be read using the foreign key, for example if the foreign key is not enabled, if the dialog cannot show the designated record, or if the record does not exist in the database.

Attribute name	Type	Usage
		hasNoRecords() The `hasNoRecords()` function is true if the pane shows 0 rows, *including the case* where the dialog pane (a table) *is* associated with an existing record (from the card pane), but where there are just no rows in the table. Thus, the `hasNoRecords()` function should be used with great caution, because it might prevent a user from populating the table (because the "insert" or "add" row are not accessible when the pane is not shown). Thus, the `hasNoRecords()` function should only be used in cases where it is never possible to add new rows to a table. `hasNoRecords()` covers *all* cases that are covered by `hasNoSeed()` *plus* the case where data has been read; there just happen to be 0 rows in the result. hasNoChildren() The `hasNoChildren()` function is true if the pane does not have any children. The number of children is dynamically calculated from the hidden-attribute of all of the children of the current pane. This means that if the current pane has a number of children, which are *all* dynamically hidden from their state, the parent (current) pane is also hidden.

<Table>-tag

The <Table>-tag specifies the presence of a table pane at this level in the workspace. A table pane always corresponds to the table part of a card-table dialog. The table pane can contain bindings to other panes, and these can be organized in different compositional structures (for example, Expansions, Assistants). Technically, a table pane is always bound by referring the record in the *upper pane* of the same dialog.

A table that is not bound by any other pane shows the table part of a *random upper-pane record* that is identified by the where clause that defines the upper-pane cursor in the corresponding dialog. This can be useful for card/table-dialogs that are only capable of showing one upper-pane record.

Attribute Name	Type	Usage
source *Not available if table is bound using a <With>-binding*	NsKey	Indicates the name of the dialog that is used to define the content of the table. For example, if the dialog is "TimeSheets," the table shows time sheet lines plus variables defined for the table part of the dialog "TimeSheets."
title	Display	Represents access to the table pane. If not specified, the title of the associated MDML layout is

Attribute Name	Type	Usage
		used, or, if that title is undefined, the default title specified for that TablePane in DDL is applied.
layout	NsKey	Name of the MDML file that contains the layout that should be applied for this pane (excluding ".mdml.xml"). If this value is the empty string, no layout is applied. A blank pane is displayed. (This is useful to quickly make a mockup workspace.) If this attribute is not specified, the layout called *dialogName*.mdml.xml is used. If the pane is bound by a <With>-binding, the layout value from its parent is the default if no value is explicitly specified.
view	Key/Expression	Name of the layout view to apply. In a layout file, each type of pane can have many different layout *views* defined. Laying out a pane requires *one* view. The first view that has the name specified by this attribute is applied. If no value is specified, the first view (for the relevant pane) is applied. It is possible to give *arguments* to a view (provided that the view is declared with corresponding parameters in MDML). In this case, you can add a number of expressions in a comma-separated list, for example: `view="myView(true, 33 - 2, currentDate())"` If the pane is bound by a <With>-binding, the view value from its parent is the default if no value is explicitly specified.
name	NsKey	The internal name of the pane in *this* workspace. This is currently unused, but may eventually be used to address a specific pane in a workspace from, for example, links, notifications, and so on.
pluginId	Id	The ID of the plug-in that implements the table pane to apply in this instance. If left unspecified, the default filter-implementation is used.
hidden	Expression	If the expression evaluates to true, this table (and all of its children) are not visible in the workspace; they are hidden. This means that they may as well not be there. The expression is evaluated in the context of the current table. This means that all fields, functions, and expressions that are exposed to the table can be used for calculation of the visibility of the table. For this particular attribute, three special functions are available.

Attribute Name	Type	Usage
		hasNoSeed() The `hasNoSeed()` function is true if the pane does not have any data, *not even "0 rows."* Basically, this means that the underlying dialog pane cannot be read using the foreign key, for example if the foreign key is not enabled, if the dialog cannot show the designated record, or if the record does not exist in the database. hasNoRecords() The `hasNoRecords()` function is true if the pane shows 0 rows, *including the case* where the dialog pane (a table) *is* associated with an existing record (from the card pane), but where there are just no rows in the table. Thus, the `hasNoRecords()` function should be used with great caution, because it might prevent a user from populating the table (because the "insert" or "add" row is not accessible when the pane is not shown).The `hasNoRecords()` function should only be used in cases where it is not possible to add new rows to a table. `hasNoRecords()` thus covers *all* cases that are covered by `hasNoSeed()` *plus* the case where data has been read; there just happen to be 0 rows in the result. hasNoChildren() The `hasNoChildren()` function is true if the pane does not have any children. The number of children is dynamically calculated from the hidden-attribute of all of the children of the current pane. This means that if the current pane has a number of children, *all* dynamically hidden from there state, the parent (current) pane is also hidden.

<Hidden>-tag

A hidden tag corresponds to a card pane that is *not* shown to a user. This can be handy to "secretly" bind two dialog-panes that are not directly related. For example, if entity E1 points to a record in entity E2, and E2 points to a record in entity E3. Then suppose that there are three dialogs: D1 (containing E1), D2 (containing E2), and D3 (containing E3). You want a workspace that contains a pane that shows D1 followed directly by one that shows D3 (where the record that is displayed is the one that is identified by the record in D2, which was referred by the record in D1). In this case, a workspace like the following can be used:

```
<Card dialog="D1">
  <Bind foreignKey="E1toE2">
    <Hidden dialog="D2">
```

```
            <Bind foreignKey="E2toE3">
              <Card dialog="D3"/>
            </Bind>
          </Hidden>
        </Bind>
      </Card>
```

This yields a workspace that shows a card pane at the top (D1) followed by another card pane (D3).

A hidden pane that is not bound by any other pane behaves just like a corresponding <Card> would behave, except that it is not shown.

Attribute Name	Type	Usage
Source	NsKey	Indicates the name of the dialog that is used to define the content of the hidden card. For example, if the dialog is "TimeSheets," nothing is shown, but you may create bindings from this pane using all foreign keys that are defined for the card part of the Time Sheets dialog.
Name	NsKey	The internal name of the pane in *this* workspace. Currently unused, eventually it may be used to address a specific pane in a workspace from, for example, links, notifications, and so on.

<Workspace>-tag (embedded)

The <Workspace>-tag, in embedded form, is used to "include" the contents of another workspace specification into this one. It can be used in situations where a pane can be referred. At that position, the content of the referred workspace specification (excluding its outer-most workspace tag) is inserted into this workspace. The embedded workspace can recursively embed other workspace specs.

Attribute Name	Type	Usage
source	NsKey	The name of the workspace spec to "include" at this place.
Name	NsKey	Currently unused, but may be used in the future to reference this workspace inclusion.

<Section>-tag

The <Section>-tag marks a "titled" button/tab that has no direct content (for example, its content is defined by its inner tags; alone it has no content). See the <Formation>-tag for more details.

Because a segment does not hold any data by itself, it is bypassed in the data binding. Therefore, a segment must contain either a formation or a binding; the binding relates to the nearest pane that precedes the segment. For example, consider the following specification:

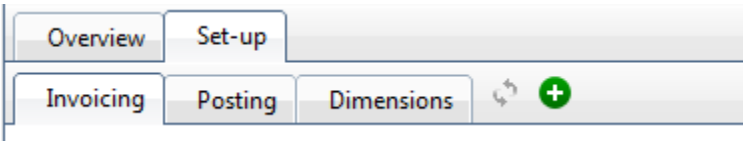
```
<Filter dialog="Jobs">
  <Formation>
    <Segment title="Home">
```

```
      <Bind> … </Bind>
      <Bind foreignKey="MainJob"> … </Bind>
    </Segment>

    <Segment title="Budget">
      <Bind> … </Bind>
    </Segment>
  </Formation>
</Filter>
```

Here the implicit "primary" bindings in segments "Home" and "Budget" refer to the current record in the Jobs-filter. Similarly, the "MainJob" binding refers to the current record in the Jobs filter.

It is also possible to have "sections" (empty tabs) beside normal panes (for example, outside of <Formation>s). In this case, the section is represented as an empty tab with its expansion following directly.

In the following, you can see a section "Set-up" where tabs follow directly below.



Attribute Name	Type	Usage
name	Key	The internal name of the section in *this* workspace. This is currently unused, but may eventually be used to address a specific segment in a workspace from, for example, links, notifications, and so on.
title	Display	The title of the section displayed in the button/tab.
hidden	Expression	If the expression evaluates to true, this section (and all of its children) are not visible in the workspace; they are hidden. This means that they may as well not be there. The expression is evaluated in the context of the current section. This means that all fields, functions, and expressions that are exposed to the section can be used for calculation on the visibility of the section. For this particular attribute, three special functions are available: hasNoSeed() The `hasNoSeed()` function is true if the pane does not have any data, *not even "0 rows"*. Basically, this means that the underlying dialog pane cannot be read using the foreign key, for example if the foreign key is not enabled, if the dialog cannot show the designated record, or if the record does not exist in the database.

Attribute Name	Type	Usage
		hasNoRecords() The `hasNoRecords()` function is true if the pane shows 0 rows, *including the case* where the dialog pane (a table) *is* associated with an existing record (from the card pane), but where there are just no rows in the table. Thus, the `hasNoRecords()` function should be used with great caution, because it might prevent a user from populating the table (because the "insert" or "add" row is not accessible when the pane is not shown). Thus, the `hasNoRecords()` function should only be used in cases where it is never possible to add new rows to a table. `hasNoRecords()` thus covers *all* cases that are covered by `hasNoSeed()` *plus* the case where data has been read; there just happen to be 0 rows in the result. hasNoChildren() The `hasNoChildren()` function is true if the pane do not have any children. The number of children is dynamically calculated from the hidden-attribute of all of the children of the current pane. This means that if the current pane has a number of children, which are *all* dynamically hidden from there state, the parent (current) pane is also hidden.

Container Tags

<Workspace>-tag (outermost level)

The <Workspace>-tag marks the contents of the workspace.

Attribute Name	Type	Usage
name	NsKey	Indicates the name of the workspace. It must be identical to the file name (leaving out ".mwsl.xml"). A namespace defines a sub-folder in the workspace-folder. (For example, MyNamespace:MyFile must be stored in: mynamespace/myfile.mwsl.xml.)
title	Display	The title of the workspace. This tag is largely unused. (The title of the workspace is taken from the menu spec.) If a workspace is loaded by other means than the menu spec., this title is used. In time, the menu spec. could be compiled from the titles that are defined by the workspace specs.

<Expansions>-tag

The <Expansions>-tag groups together a number of panes. They are shown as a number of tabs. The content of the expansion direction is assumed to indicate the primary purpose of a workspace, which is reflected by current and future defaults.

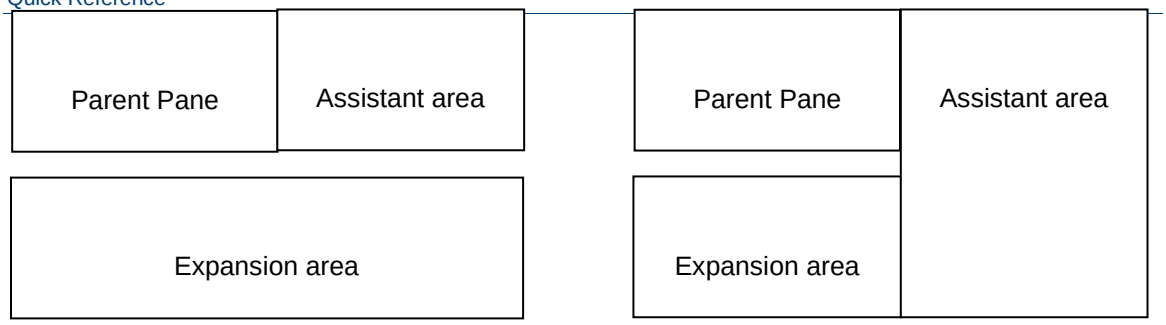
Attribute Name	Type	Usage
name	Key	Indicates the name of the expansion collection. This is currently unused, but may be used in the future to reference a specific expansions collection.
horizontal	Boolean	The default is false. (This is currently true for expansions following an initial filter. This will be changed in the future.) If true, the expansion tabs are displayed to the *right* of the parent pane. Otherwise, the expansion tabs are displayed *below* the parent panes.
minimized	Boolean	The default is false (except for the first expansion that does not contain an initial filter). If true, the expansions are minimized by default (so not directly visible). Instead a reveal button is displayed.
parentSize	Enumeration	This attribute specifies the ratio between this expansion and its parent pane by defining the relative size of the *parent pane*. Possible values are:

Attribute Name	Type	Usage
		`tiny, small, medium, large, huge, vast`
hidden	Expression	Determines the default expression for the hidden attribute for panes in this expansion.

<Assistants>-tag

Assistants are used to group a number of panes. They are shown as a number of tabs. The content of the assistant direction is assumed to contain information that is “occasionally needed” (in contrast to Expansions, which are considered “prime content”). This fact is reflected by current and future defaults. Apart from the defaults, there are no differences between what can be done with assistants and what can be done with expansions.

Attribute Name	Type	Usage
name	Key	Indicates the name of the assistant collection. This is currently unused, but may be used in the future to reference a specific assistants collection.
horizontal	Boolean	The default is true. (Currently, false is for assistants that follow an initial filter; this will be changed in the future.) If true, the assistant tabs are displayed to the *right* of the parent pane. Otherwise, the expansion tabs are displayed *below* the parent panes.
minimized	Boolean	The default is true. If true, the assistants are minimized by default (that is, not directly visible). Instead a reveal button is displayed.
enlarged	Boolean	The default is false. This attribute has effect *only* when the parent pane has *both* assistants and expansions. If true, the assistants are space-prioritized over the expansions. If false, the expansions are space-prioritized.
parentSize	Enumeration	This attribute specifies the ratio between this assistant and its parent pane by defining the relative size of the *parent pane*. Possible values are: `tiny, small, medium, large, huge, vast`
hidden	Expression	Determines the default expression for the hidden attribute for panes in this assistant.

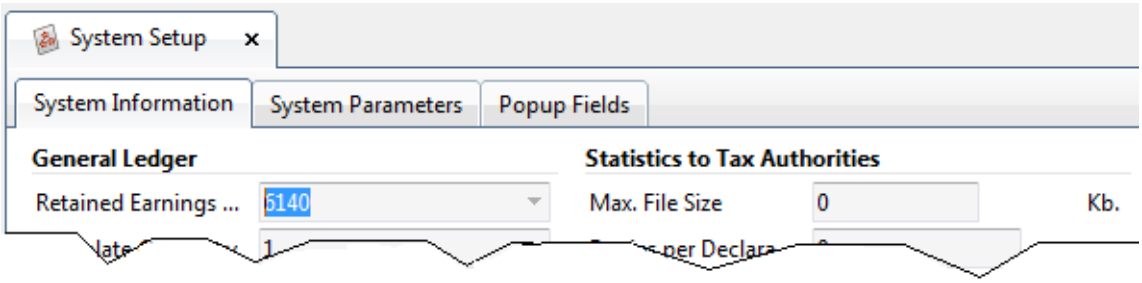


Expansion is space prioritized

Assistant is space prioritized

Initial Collection of Tabs

The “root” of a workspace can be thought of as “an initial expansion.” Thus, it is possible to initially specify a number of parallel structures. Notice the difference between initial parallel structures and a formation: A formation is a collection of “title buttons” (that is, no pane content) rendered as “boxes,” whereas an initial parallel structure is a collection of panes (each of which has a tab that represents access to that pane).



<Formation>-tag

A formation represents a collection of “title buttons” or “empty tabs.” Visually, formations are rendered as a “ribbon that contains rectangular titled buttons” (specified by <Section>-tags). There is *no* content that is directly associated with any of these title buttons. The only thing that can follow is an Expansions-collection (for example, a collection of panes, each represented with a tab), another formation, or a *Mount*-connection. (See <Mount>-tag.) Formations can be used to offer a selection of what appears to be a “sub workspace” or can be seen as a “sub menu.”

Attribute Name	Type	Usage
name	Key	Indicates the name of the formation collection. This is currently unused, but may be used in the future to reference a specific formation collection.

Connection Tags

Connection tags are used to define how two consecutive panes are bound together by their data, in other words, how the parent pane defines the content of the following pane.

<Mount>-tag

The mount connection defines that the parent binding has *no influence* on the contents of the following pane. This implies that anything following a <Mount>-tag is populated in the same way as it would be if it were the first pane in a workspace. A workspace always implicitly starts with a <Mount>-binding. <Mount> is often used to specify the “beginning” in segments inside an initial formation, or to connect to panes that do not have a formalized key (such as action cards). Also, it is possible to start “anew” inside a workspace by using the <Mount>-connection.

There are currently no attributes for the <Mount>-tag.

<Bind>-tag

The bind connection is used to bind a parent pane to a *single record*. The content of the following pane is defined by *one single* record.

Attribute Name	Type	Usage
name	Key	Indicates the name of the connection. It is currently unused, but may be used in the future to reference fields from child panes. If unspecified, the name defaults to the name of the foreign key specified by the foreignKey attribute.
foreignKey	Key	The name of the foreign key to use for this binding. The referred name must exist in the parent pane, and it must point to the entity that is contained by the child pane. If you just want the “record itself,” use the foreign key that is by convention named “primary.”

<Restrict>-tag

The restrict connection is used to connect filter panes. The restriction denotes a set of records that can be shown in the child filter pane based on the record that has focus in the parent filter pane.

For the casual eye, there are two different uses of the restrict connection: one where a foreign key from *both* the parent and the child pane is specified, and one where only a foreign key from the child filter pane is specified.

Example: Foreign Key Specified on Child Pane *only*

The parent pane shows customers, and the child pane shows a job. The foreign key from the *child pane* (referred to as the “reversed foreign key” because it points in the “reverse” direction of the data flow) is “customer referred by a job.” This means that any record that is shown in the *child* pane must point out (using that foreign key) exactly the record that has focus in the parent pane. In this case, it means that all jobs that are shown in the child pane must point to the customer that is selected in the parent pane.

Example: Foreign Key Specified on *both* Child Pane and Parent Pane

The parent pane shows “customer,” and the child pane shows “employees.” The foreign key from the *child pane* (in other words, the “reversed foreign key”) points out a zip code. The foreign key from the *parent pane* (in other words, the “foreign key”) also points out a zip code. Both foreign keys must point to the same entity. This means that any record that is shown in the *child* pane must point out the *same* zip code that is pointed out by the selected record in the *parent pane*. The “zip code” is used as a “stepping stone” between the parent and the child pane entities. In this case, it means that selecting a customer, the second pane shows employees that live close to that customer (in other words, in the same zip code area).

The two uses are actually the same: in the first case (in other words, only specifying a “reversed foreign key”), the parent pane always implicitly points to itself.

Attribute Name	Type	Usage
name	Key	Indicates the name of the connection. This is currently unused, but may be used in the future to reference fields from child panes. If unspecified, the name defaults to the name of the foreign key specified by the reversedForeignKey attribute.
reversedForeignKey	Key	The name of the foreign key that points from the child pane to the record/entity pointed out by the foreign key from the parent pane. If the foreignKey attribute is left unspecified, it defaults to “the current record in the parent pane,” meaning that the reversedForeignKey should point out the current record in the parent pane in this case.
foreignKey	Key	The default is primary (in other words, the same record as is found in the parent pane). The name of the foreign key to use for this binding. The referred name must exist in the parent pane, and it must point to the same entity that is pointed to by the reversed foreign key.

<With>-binding

<With>-bindings are used to connect panes from the same source container (for example, the same dialog). Basically it means that whatever connection is used internally in the dialog to populate data in several panes is used. It also means that panes that are bound together using the <With>-binding share variable state space. This is most easily understandable for card/table dialogs: the content of the *table* part is determined intrinsically by the application source code for the dialog. In particular, the content of the table pane may depend on the value of *variables* that are held by the *card pane*. Thus, to obtain the table that intrinsically belongs with a particular card-record, you must use the <With>-binding. In many ways, a <With> binding is similar to <Bind foreignKey=”primary”>, but the very important difference is that using the <With> binding, you are guaranteed to get the pane that belongs with the one to which it is <With>-bound.

As an oddity, binding a card and a table pane from the same dialog together using <Bind foreignKey=”primary”> guarantees that if the table content depends on *variables* that are manipulated in the card, the table does *not* reflect these values.

You can *only* bind panes together using <With> if they originate from the same container source (dialog). In fact, the syntax makes certain of that. Thus, if you want to bind two panes of different

dialogs together you *cannot* use <With>. You must use <Bind foreignKey="…"> or one of the other bindings.

Typically, when you have panes from the same dialog, you want to bind them together using <With>.

For dialogs, it is supported to connect Card, Table, and Filter using <With>.

There are currently no attributes for the <With>-binding.

<Through>-binding

The <Through>-binding is used when you insert a <Section> (empty tab) in the middle of a chain of panes that data-wise depend on each other. Basically it means that the data context that is immediately *above* is passed "through" the section and, thus, is available for the children of the section.

An example is:

```
<MWSL xmlns="http://www.deltek.com/ns/mwsl" version="0.24">
  <Workspace name="MyWorkspace">
    <Filter source="Jobs">
      <With>
        <Card/>
      </With>
      <Through>
        <Section title="This is an empty tab">
          <With>
            <Card/>
          </With>
          <Bind foreignKey="ProjectManagerNumber_Employee">
            <Card source="Employees"/>
          </Bind>
        </Section>
      </Through>
    </Filter>
  </Workspace>
</MWSL>
```

In this example, you start out with a filter of jobs. In the following, there is first the card of the dialog "Jobs" that shows the job that is selected in the filter. Also below the filter, there is an empty tab. And below that empty tab there *again* is access to the data that is selected in the filter. This enables the showing of that job card-pane again, as well as an employee card that shows the project manager of the selected job.

There are currently no attributes on the <Through>-binding.

MOL

Introduction

Maconomy Object Language (MOL) is a language for defining custom database objects to be integrated in a standard Maconomy installation. Using MOL, new database objects are defined in Maconomy terminology.

This documentation reference describes the MOL language syntax.

For further information about the installation and use of MOL objects, see *Deltek Maconomy MBuilder Reference* and *MUL Language Reference* section in this document.

Language Definition

This section describes the current version of MOL. The formal syntax of MOL is presented in BNF (Bachus Naur Form). The syntax of MOL is tag-based, and it consists of elements and attributes just like XML. Unlike XML, every attribute in MOL has an associated type, and can have a short form.

Note that MOL is not case-sensitive.

```
mol ::=
  <MOL 1.0>
  object
```

Object

The core element of MOL is the object element. Exactly one instance of the object element must be defined. Inside the object element are field definitions, and so on.

```
object ::=
  <object (name = id | id) [(title = string | string)] >
    fieldDefinition+
  <end object>
```

Attributes of the object element are described in the following table.

Attribute	Description
name	The unique name identifier of the object. The name must be prefixed by a three-letter namespace followed by '_', for example, "abc_MySalary" where "abc" is the namespace.
[title]	The title of the object.

In the following example, a simple MOL object is defined. The namespace identifier is "abc", and the resulting object name is "abc_MySalary."

```
<MOL 1.0>
<Object abc_MySalary>
 .Componentid    :String :Key+
 .Target         :Integer
 .Salary         :Amount
 .SalaryGroup    :String :"Salary level"
<End Object>
```

Field Definitions

All field definitions reside inside the object element. A field consists of a field name, a field type, and possibly a field title and a key indicator attribute.

```
fieldDefinition ::=
  .id :(type = id | id) [:key(+ | -)])
  |
  (<field ((name = id | id) :(type = id) [:key(+ | -)]) >)
```

Attributes of the field element are as follows.

Attribute	Description
name	The name identifier of the field. The name must be unique within the object.
type	The type identifier attribute is mandatory. Available types are listed in “Maconomy Types”.
[key]	The optional key indicator attribute is used to indicate key fields of the object. The set of all key fields must identify the object entries uniquely.

In the previous example, the single field `Componentid` of type String is the only key field.

```
.Componentid  :String :Key+
```

Thus, the field Componentid must uniquely identify the object entries.

Maconomy Types

A set of predefined types in the Maconomy environment is available for use in MOL. Note that the database implementation of all Maconomy types is fixed and highly database-specific. See the “MBuilder Reference” documentation for further details on type conversions.

The types are divided into two groups: basic types and enumeration pop-up types.

Basic Types

The basic types available from MOL are listed in the following table.

Type	Description
`INTEGER`	32-bit integer value
`REAL`	Floating point value
`AMOUNT`	Fixed Maconomy-style amount type
`BOOLEAN`	Boolean type
`STRING`	String type of max. 255 characters

Type	Description
DATE	Fixed Maconomy style date type
TIME	Fixed Maconomy style time type

Popup Types

A large set of Maconomy version-specific enumeration types (“pop-up types”) are available in MOL. The set of available types and their corresponding values is Maconomy version-specific, and can be viewed using a Maconomy client in the Popup Fields dialog in the Setup module.

MQL

Introduction

The Maconomy Query Language (MQL) is a language for interacting with the Maconomy database. Currently only data selection is supported by MQL.

MQL, Universes, and SQL

To reduce the MQL learning curve, MQL looks similar to SQL. However, this resemblance is only on the surface, because MQL is a language on its own.

When writing a SQL command, the developer must know the join structure of the relations in the database. The join structure and semantic information about fields in the database, also called the data model, is very complex in the Maconomy system.

The main difference between MQL and SQL is the separation of the data model from the command. In SQL, the data model is specified in each command; in MQL, the data model is specified in a Universe. This separation enables the reuse of the data model in multiple commands, and it enables the MQL command to be developed without knowledge of the data model. For further information on Universes, see the *Deltek Maconomy Language Reference MUL.*

Unlike SQL, MQL is aware of types, and especially the Maconomy type system. The types of all expressions in an MQL command are validated before execution. This sort of validation can reveal many errors during development.

Unlike SQL, MQL is database-independent. The Maconomy Server knows which database is actually used, and performs a runtime translation of MQL into SQL in accordance with the requirements of the specific database.

Reading this Manual

The formal syntax of MQL is presented in BNF (Bachus Naur Form).

Where to Use MQL

MQL is designed to be the Maconomy language for direct database access. Currently, MQL is only used for reporting purposes, and is only available in M-Script and in MRL.

M-Script

MQL can be used from M-Script with the maconomy::mql* set of functions.

The version of the MQL command executed is dependent on the M-Script version, unless explicitly defined in the MQL command.

In the M-Script context, actual values for parameters can be given through an M-Script object, and the result of a command is returned as an M-Script object. Cursor definitions and result structuring features are ignored in this context.

For further information on MQL command integration in M-Script, see the *Deltek M-Script Maconomy API Reference*.

MRL (Maconomy Report Language)

MRL is used for specifying Universe Reports. A Universe Report is installed and executed on the Maconomy Server. MQL is used in MRL for query specification.

The version of the MQL query executed is dependent on the MRL version used.

In the MRL context, formal parameters are defined outside the MQL query, but are implicitly available inside the query. The MRL runtime framework handles actual values for the parameters. Cursor definitions and result structuring features are available in this context.

For further information on MQL command integration in MRL, see the *Deltek Maconomy Language Reference MRL* or the introduction to Universe reporting, *Getting Started with Universe Reports*.

Commands

This section describes the current version of MQL. A version history of MQL is supplied in the "Version History" section. Unlike SQL, an MQL command contains version identification, used for controlling syntax and/or semantic changes. The version identification can be implicit, given by the context of the MQL command. See Where to Use MQL for more information.

MQL commands are not case-sensitive; for instance, the command mselect equals MSelect.

```
mql ::=
  (<MQL 1.5> | MQL 1.5)
  mselect
```

MSelect Command

The `mselect` command retrieves rows, columns, and derived values from a Maconomy Universe. The syntax for the command is as follows.

```
mselect ::=
   MSELECT [DISTINCT] (fieldlist|cursor)
   [AGGREGATE [(ALL|SUM|MINMAX),] [aggregatedef (,aggregatedef)*] ]
   FROM module [INTERFACE id]
   [WHERE expressionshort]
   [ORDER BY qualifiedid [ASC|DESC] (, qualifiedid [ASC|DESC])* ]
   [USING PARAMETERS parmfield (, parmfield)* ]

fieldlist ::=
   field (, field)*

field ::=
   qualifiedfieldid |
   expressionshort AS id [TITLE string]

cursor ::=
[ fieldlist [, cursor] ] AS CURSOR id

aggregatedef ::=
   aggregateexp [AS id [TITLE string]] [ON idOrQualifiedid]

aggregateexp ::=
   SUM() |
   MIN() |
   MAX() |
   PCT(idOrFunctionfieldid, idOrFunctionfieldid) |
   MUL(idOrFunctionfieldid, idOrFunctionfieldid) |
   DIV(idOrFunctionfieldid, idOrFunctionfieldid) |
   constExpressionShort

parmfield ::=

   id (TYPE|:) typeid [TITLE string] [DEFAULT constExpressionShort]
```

Universe and Field Selection

The basic functionality of the `mselect` command is the selection of fields from a Universe. The Universe is selected in the from clause, and fields used elsewhere in the query are taken from this Universe.

Unlike SQL, Universes cannot be joined in the from clause. If a join of Universes is required, then a new Universe must be defined. Note that all Maconomy relations are also available as Universes.

Column Selection

The fields in the select clause define the order of the columns in the result. The fields also determine the root-selection of the used Universe. The root-selection is quite complex, but normally you do not need to be concerned with this issue. See the *Deltek Maconomy Language Reference MUL* for further information.

The following example shows the selection of two fields from the 'Employee' Universe, which is also a Maconomy relation:

```
mselect EmployeeNumber, Name1 from Employee
```

EmployeeNumber **EmployeeNumber** **String**	**Name1** **(Name)** **Name 1** **(Title)** **String** **(Type)**
1011	Hansa Mujaf
1012	Joe Daniels

Unlike SQL, there is no group-by clause in MQL where explicit grouping of rows can be defined. If a field in the select clause is defined using one of the row group functions (see the "Row Group Functions" section), implicit row grouping is done on the selected fields not defined using a row group function.

In the following example, the field 'EmployeeNumberCOUNT' is defined using a row group function, and therefore an implicit row grouping is done on the field 'Country'.

```
mselect Country,
        COUNT(EmployeeNumber) as EmployeeNumberCOUNT
from Employee
order by Country
```

Country **Country** **CountryType**	**EmployeeNumberCOUNT** **(Name)** **EmployeeNumberCOUNT** **(Title)** **Integer** **(Type)**
UK	2
USA	10

Universe Interface Option

Different interfaces can be defined in a Universe, allowing different views of the data model defined in the Universe. The interface used in the query is selected in the from clause. If no interface is specified, the default interface is used.

Distinct Option

The distinct option in the select clause specifies how to handle duplicate rows in the result. If the option is selected, duplicate rows are filtered out. By default, the distinct option is not selected, that is, all rows in the result are shown.

Field Definition

New fields can be defined in the select clause using the Maconomy functions defined in the "Maconomy Functions" section. Unlike SQL, all fields have an associated title. The title of a new field is defined by the title option. If no title is defined, the name of the field is used as the title.

In the following example, the Boolean field "Employed2003" is defined, indicating if the employee was employed in the year 2003. Also, the Boolean field "Employed" is defined, indicating if the employee is currently employed:

```
mselect EmployeeNumber,
        Name1,
        DateEmployed inrange [2003.01.01 .. 2003.12.31] as
Employed2003,
        DateEndEmployment = date'nil as Employed title "Employed ?"
from Employee
```

EmployeeNumber EmployeeNumber String	Name1 Name 1 String	Employed2003 Employed2003 Boolean	Employed Employed ? Boolean
1011	Hansa Mujaf	Yes	Yes
1012	Joe Daniels	No	No
...			

Structuring the Result

Unlike SQL, Mselect contains features for structuring the result of the query. The result structure might be ignored if the location where the command is executed does not support this feature. See "Where to Use SQL."

Aggregate Definition

With an aggregate definition, the sum, minimum, and maximum of values in a result column can be calculated. This kind of aggregate is called column aggregates. Further aggregates can be defined using the column aggregates as input for some simple calculations. This kind of aggregate is called aggregate aggregates.

Column Aggregates

Column aggregates can be defined explicitly for each column or defined implicitly for all integer, real, and amount columns for which a row group function is used. With the explicit definition, the

aggregate can be assigned to a name and given a title, whereas with the implicit definition, names are constructed from the column name. The names of aggregates can be used when referring to the aggregates, for example in Universe Reports, where the aggregates can be used in the layout file.

Implicit column aggregates are performed on all integer, real, and amount columns defined using a row group function. Valid values for the implicit column aggregate definition are as follows.

ALL	Sum, minimum, and maximum of all columns defined using a row group function are calculated.
SUM	Sum of all columns defined using a row group function is calculated.
MINMAX	Minimum and maximum of all columns defined using a row group function are calculated.

If no name is specified in an explicit column aggregate definition, a name is assigned to the aggregate exactly as if it was implicitly defined.

In the following example, aggregate calculation is performed on the real column 'NumberOfWeekSUM', which is a field in the Universe "JobUniverse" defined using the row group function SUM. Aggregate calculation is also performed on the integer column "cTrans," defined in the query using the row group function COUNT. Note that aggregate calculation is not performed on the integer column "integerField" because it is not defined using a row group function. Notice also the names assigned to the aggregates.

```
mselect Employee.EmployeeNumber,
        Employee.Name,
        NumberOfWeekSUM as hours,
        1 as iField,
        count(Employee.EmployeeNumber) as cTrans
aggregate all
from JobUniverse
where TimeSheet.PeriodStart inrange [2003.01.01 .. 2003.12.31]
order by Employee.EmployeeNumber
```

Employee.EmployeeNumber EmployeeNumber String	Employee.Name Name 1 String	hours ??? Real	iField ??? Integer	cTrans ??? Integer
1011	Hansa Mujaf	70.0	1	2
1012	Joe Daniels	47.5	1	7
		117.5 *(hours$SUM)*		*9* *(cTrans$SUM)*
		47.5 *(hours$MIN)*		*2* *(cTrans$MIN)*
		70.0 *(hours$MAX)*		*7* *(cTrans$MAX)*

In the following example, an explicit column aggregate is defined to calculate the sum of the "iField" column, which is not included in the implicit column aggregations because the column is not defined using a row group function. Note the name "iFieldMin" assigned to the "min" explicit-aggregate definition of the "iField" column.

```
mselect Employee.EmployeeNumber,
        Employee.Name,
        NumberOfWeekSUM as hours,
        1 as iField,
        count(Employee.EmployeeNumber) as cTrans
aggregate all,
          sum() on iField,
          min() as iFieldMin on iField
from JobUniverse
where TimeSheet.PeriodStart inrange [2003.01.01 .. 2003.12.31]
order by Employee.EmployeeNumber
```

Employee.EmployeeNumber **EmployeeNumber** **String**	**Employee.Name** **Name 1** **String**	**hours** **???** **Real**	**iField** **???** **Integer**	**cTrans** **???** **Integer**
1011	Hansa Mujaf	70.0	1	2
1012	Joe Daniels	47.5	1	7
		117.5 *(hours$SUM)*	*2* *(iField$SUM)*	*9* *(cTrans$SUM)*
		47.5 *(hours$MIN)*	*1* *(iFieldMin)*	*2* *(cTrans$MIN)*
		70.0 *(hours$MAX)*		*7* *(cTrans$MAX)*

Aggregate Aggregates

Aggregate aggregates can be defined using column aggregates and constants as input for some simple calculations. This can, for example, be used for calculation of overall averages.

The simple calculations available are division using "div," multiplication using "mul," and %-calculation using "pct." An aggregate aggregate cannot be associated with a column; that is, the "on" option is not valid.

In the following example, implicit column aggregates are defined as in the previous examples. Additionally, the aggregate aggregate "aaOverallAvg" is defined using the column aggregates "hours$SUM" and "cTrans$SUM."

```
mselect Employee.EmployeeNumber,
        Employee.Name,
        NumberOfWeekSUM as hours,
        count(Employee.EmployeeNumber) as cTrans
aggregate all,
          div(hours$SUM, cTrans$SUM) as aaOverallAvg
from JobUniverse
where TimeSheet.PeriodStart inrange [2003.01.01 .. 2003.12.31]
order by Employee.EmployeeNumber
```

Employee.EmployeeNumber **EmployeeNumber** **String**	**Employee.Name** **Name 1** **String**	**hours** **???** **Real**	**cTrans** **???** **Integer**
1011	Hansa Mujaf	70.0	2
1012	Joe Daniels	47.5	7
		117.5 *(hours$SUM)*	*9* *(cTrans$SUM)*
		47.5 *(hours$MIN)*	*2* *(cTrans$MIN)*
		70.0 *(hours$MAX)*	*7* *(cTrans$MAX)*
13.056 (aaOverallAvg)			

Cursor Definition

A cursor is a named group of result columns. Naming a group of columns is needed when references are made to multiple queries. This is the case in MRL Reports where multiple queries can be defined and referred to in the layout of the report.

A cursor can contain a subcursor. A subcursor specifies that for each different value of the result row outside the subcursor, a list of rows with values inside the subcursor is to be generated. A subcursor is needed in connection with the aggregate option, when a subtotal is needed for a group of columns.

The cursor definitions are ignored if the location where the command is executed does not support this feature. See Where to Use MQL.

In the following example, a subcursor named “JobCursor” is defined, grouping the job related columns. The effect is that for each employee row, a list of rows (one row for each job), and a subtotal for each employee is generated.

```
mselect [ Employee.EmployeeNumber,
          Employee.Name,
          [Job.JobNumber, NumberOfWeekSUM] as cursor JobCursor
        ] as cursor EmployeeCursor
        aggregate sum
from JobUniverse
where TimeSheet.PeriodStart inrange [2003.01.01 .. 2003.12.31]
order by Employee.EmployeeNumber, Job.JobNumber
```

Employee.EmployeeNumber EmployeeNumber String	Employee.Name Name 1 String	Job.JobNumber ??? String	NumberOfWeekSUM ??? Real
1011	Hansa Mujaf	10	40.0
		20	30.0
			70.0 (NumberOfWeekSUM$SUM)
1012	Joe Daniels	10	37.5
		30	10.0
			47.5 (NumberOfWeekSUM$SUM)
			117.5 (NumberOfWeekSUM$SUM)

Restriction

Restrictions on the result rows can be specified in the where clause of the `mselect` command. A large number of Maconomy functions are available for restriction specifications. See Maconomy Functions.

In the following example, all employees who are employed later than the beginning of this month are listed:

```
mselect EmployeeNumber, Name1
from Employee
where DateEmployed >= getfirstofmonth( getdate() )
order by EmployeeNumber
```

Unlike SQL, the `mselect` command does not have a having clause. In SQL, the having clause is used for restrictions that contain a row group function. In Mselect, such restrictions are also specified in the where clause.

In the following example, all countries with fewer than 5 employees are shown.

```
mselect Country
from Employee
where COUNT(EmployeeNumber) < 5
```

```
order by Country
```

Unlike in SQL, in MQL subqueries cannot be used in a restriction. A subquery is a query used in the where clause. Subqueries will be introduced in a future version of MQL. Note that subqueries can be used in a restriction defined in the Universe.

Ordering

The order of the result rows can be specified in the order-by clause. Unlike in SQL, in MQL only fields selected in the select clause can be used for ordering. It is always a good idea to specify the order-by clause, because if it is not specified, the order is undefined.

The result rows may be sorted in ascending or descending order. Valid values for the order option are as follows.

ASC	Ascending sorting, this is the default value.
DESC	Descending sorting.

In the following example, the result rows are ordered ascending on the field "Employee.EmployeeNumber" and then descending on the field "Job.JobNumber."

```
mselect Employee.EmployeeNumber,
        Employee.Name,

        Job.JobNumber,

               NumberOfWeekSUM
        from JobUniverse
        order by Employee.EmployeeNumber asc,

                 Job.JobNumber desc
```

Employee.EmployeeNumber EmployeeNumber String	Employee.Name Name 1 String	Job.JobNumber ??? String	NumberOfWeekSUM ??? Real
1011	Hansa Mujaf	20	30.0
1011	Hansa Mujaf	10	40.0
1012	Joe Daniels	30	10.0
1012	Joe Daniels	10	37.5

Parameter Definition

The `mselect` command can be parameterized using the parameter section of the command. Parameterization of an `mselect` command can be used in parts of the Maconomy system that support this feature. See Where to Use MQL.

The advantage of formal parameter specification is that type-safe values can be assigned at execution time, and that a command can be validated without being executed (static validation).

A parameter defined in the parameter section can be used as any other field reference. The type of the parameter must be specified, whereas the title and default value are optional. If a default value is specified, this value is used if no actual value is given at execution time.

In the following example, all employees employed within a given date range are listed. If “parmDateBegin” equals 2003.01.01 at execution time, all employees employed after the first of January 2003 are listed. If no actual values for the parameters are given at execution time, employees employed after today are listed.

Example:

```
mselect EmployeeNumber, Name1
from Employee
where DateEmployed inrange [parmDateBegin .. parmDateEnd]
order by EmployeeNumber
using parameters parmDateBegin type date default date’today,
                 parmDateEnd   type date
```

Common Syntax

This section describes common elements of the MQL language.

Expressions

Expressions are used in restrictions and in field definitions. Every expression has an associated type, which is inferred from the explicit type given by field reference, by type given by constant, or by type given by function type schema. A type error occurs if the inferred type does not match the expected type of an expression. If for instance an expression is used as a restriction, the type of the expression must be the type Boolean. Note that implicit type conversions are not performed. Functions are available for doing explicit type conversions.

```
expressionShort ::=
  subExpressionShort       |
  fieldExpressionShort     |
  constExpressionShort     |
  functionExpressionShort

subExpressionShort :: =
  ( expressionShort )

fieldExpressionShort :: =
  qualifiedfieldid | .id | functionfieldid

functionExpressionShort :: =
  functionExpressionShortInfix   |
  functionExpressionShortPrefix

functionExpressionShortInfix :: =
  expressionShort  *  expressionShort   |
  expressionShort  /  expressionShort   |
  expressionShort DIV expressionShort   |
  expressionShort  +  expressionShort   |
  expressionShort  -  expressionShort   |
  expressionShort  <  expressionShort   |
  expressionShort  <= expressionShort   |
  expressionShort  >  expressionShort   |
  expressionShort  >= expressionShort   |
  expressionShort  =  expressionShort   |
  expressionShort  != expressionShort   |
  expressionShort  <> expressionShort   |
  expressionShort AND expressionShort   |
  expressionShort OR  expressionShort   |
  expressionShort INRANGE [ expressionShort .. expressionShort ] |
  expressionShort IN expressionShort           Special function

FunctionExpressionShortPrefix ::=
  - expressionShort                          |
  ! expressionShort                          |
  NOT expressionShort                        |
  id ( expressionShort {,expressionShort}* ) |

  id ()
```

Identifiers

```
id ::=
  char (char | num)*

module ::=
  id(::id)*                                   Job::EmployeeReport

qualifiedid ::=
  id(.id)*                                    Jobheader.JobNumber

functionfieldid ::=
  qualifiedid $ id                            Jobheader.JobNumber$SUM

idOrQualifiedid ::=
  id | qualifiedid

idOrFunctionfieldid ::=
  id | functionfieldid
```

Types

```
typeid ::=
  STRING | INTEGER | REAL | AMOUNT | BOOLEAN |
  id                                                        CountryType
```

Apart from the basis types, the Maconomy pop-up types are also available.

Literals

```
constExpressionShort ::=
  kernelString | templateString | rawString   |
  integer      | real           | amountValue |
  booleanValue | dateValue      | timeValue   |
  typedValue

templateString ::=
  "(char)*"                                                    "Template
string"

kernelString ::=
  @(char)*@                                                     @Kernel
string@

rawString ::=
  '(char)*'                                                          'Raw
string'

integer ::=
  [-]num+

real ::=
  [-]num+ . num+
0.5434
                                                                      -
1010.999

amountValue ::=
  [-]num+ . num num A
100.50A
                                                                      -
2000.50A

booleanValue ::=
  true | false

dateValue ::=
  num num num num . num num . num num
2001.12.31

timeValue ::=
  num num : num num : num num [AM|PM]
23:50:01

10:03:00AM

typedValue ::=
  typeid '__ null |
  typeid '__ id

num ::=
  1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0

char ::=
  A | .. | Z | Æ | Ø | Å | a | .. | å | ø | å
```

A template string is localized dynamically before execution of the command. This means that the string is translated using the dictionary currently selected by the user. Kernel and raw strings are **not** dynamically localized.

Maconomy Functions

A wide range of functions is available for use in expressions. For each function, a type scheme is defined. If the types of the arguments to a function do not match the type scheme, a type error is given.

Note that even if the database is known, database functions are not available.

Predefined Functions

A list of functions exists for every released TPU.

Special Functions

Function	Type schema(s) Description
in (infix operator)	(α, string) → Boolean Returns true if the first argument is in the second argument. The second argument must be a constant, which is either a literal or a parameter reference. The format of the second argument is a user interface range, for example, "1..10", "1<" for integers, and "T*" for strings. See elsewhere for format specification.

Row Group Functions

The row group functions create one row from a number of rows using the corresponding database row group functions.

Function	Type schema(s) Description
sum	integer → integer real → real amount → amount Computes the total sum of all values in a group of rows.
min	integer → integer real → real amount → amount string → string Computes the minimum of all values in a group of rows.

Function	Type schema(s) Description
max	integer → integer real → real amount → amount string → string Computes the maximum of all values in a group of rows.
avg	integer → integer real → real amount → amount Computes the average of all values in a group of rows.
count	α → integer Count the number of rows in a group. Note that unlike sql, null values appearing from outer joins are included in the count.

Version History

MQL 1.5

The aggregation feature has been extended with explicit column aggregation and aggregate aggregation. This extension can be used for overall average calculation. See Aggregate Definition.

MQL 1.4

Internal release, no changes relevant for documentation.

MQL 1.3

Support for raw strings added.

Support for template strings added—that is, dynamic translation of strings.

The special in-operator added, supporting in-range functionality on user interface values.

MQL 1.2

Initial version.

MUL

Introduction

The Maconomy Universe Language (MUL) is used when defining a Maconomy universe. A Maconomy universe is a specification of a data model or part of a data model. A universe is used when interacting with the Maconomy database through the Maconomy Query Language (MQL).

Separating Data Models from Queries

When writing an SQL query, the developer must know the join structure of the relations in the database. The join structure and semantic information about fields in the database, also called the data model, is very complex in the Maconomy system.

An SQL query contains an implicit data model through the selection of relations and join restrictions. But the query also contains a selection of fields, a definition of fields, and other restrictions. The consequence of hiding the data model in one query is that the data model is difficult to reuse and maintain. It is also impossible to create a query if you do not know the data model very well.

The following example shows a simple SQL query, where it is difficult to see the data model.

```
select activity.activitynumber,
       activity.activitytext,
       employee.Name1,
       SUM(timesheetline.numberofday1+ ...
+timesheetline.numberofday7) as regtime
from   timesheetline, activity, employee
where  timesheetline.employeenumber = employee.employeenumber
  and  timesheetline.periodestart   = '2001.01.01'
  and  activity.activitynumber = timesheetline.activitynumber
  and  SUM(timesheetline.numberofday1 + ... +
timesheetline.numberofday7) > 0
group by activity.activitynumber, activity.activitytext,

employee.Name1
```

By separating the data model from the query, you can reuse the data model in more than one query. The separation also makes the data model visible and therefore easier to explore and maintain. In addition, it enables a person who is not familiar with the data model to express a query using an already defined data model.

In the following example, the data model implicitly defined in the preceding SQL query is explicitly defined using a universe. The defined universe is called `timesheetUniverse`. Two different queries are defined using this universe. The first query is the same as the SQL query:

```
Universe:
<mul 1.3>
<universe timesheetUniverse>
  <join (TimeSheetLine, Employee)>
    .EmployeeNumber
  <end Join>
  <join (TimeSheetLine, Activity)>
    .ActivityNumber
  <End Join>

  <field regtime>

  SUM( timesheetline.numberofday1 + ... + timesheetline.numberofday7
```

```
)
  <end field>
<end universe>
```

Query 1:

```
mql 1.3
mselect Activity.ActivityNumber, Activity.ActivityText,
Employee.Name1, Regtime
from    timesheetUniverse
where   regtime > 0.0
  and   timesheetline.periodstart = 2001.01.01
order by Activity.ActivityNumber
```

Query 2:

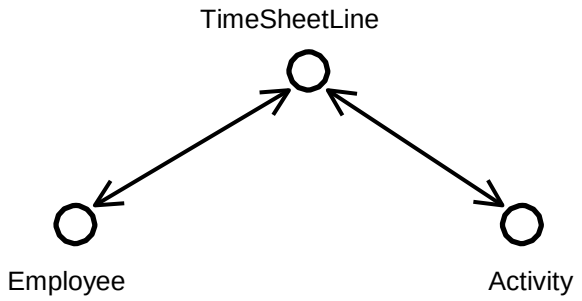
```
mql 1.3
mselect Employee.EmployeeNumber, Regtime
from    timesheetUniverse
where   timesheetline.periodstart inrange [ 2003.01.01 .. 2003.01.31
]
order by Employee.EmployeeNumber
```

How Data Models Work

In an SQL query where the data model and the query are mixed, you can adapt the data model so that it fits the given situation exactly. When separating the data model and the query, this kind of exact fit is no longer possible, because the data model is to be used in many different queries that are not available when designing the data model. When designing the data model, it is therefore necessary to have an idea of how the data model and query are combined.

The data model can be seen as a graph, where objects are vertexes, and joins are bidirectional edges. The graph defined by a universe may not contain cycles.

In the previous example, a universe was defined. The universe joins the three objects TimeSheetLine, Employee, and Activity. These joins can be illustrated as the following tree.



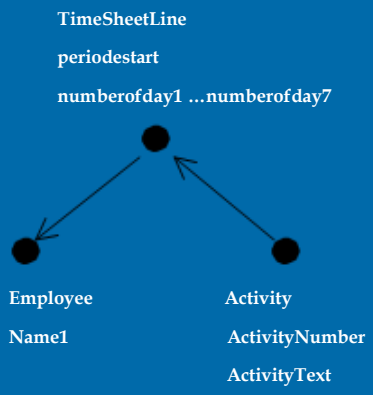
A query selects fields from the objects used in the universe, thereby selecting vertexes in the graph. The first field selected in the query selects the root of the tree. Now the least spanning tree marks the objects needed in the query.

Note the implicit root selection specified in the first selected field of the query, because it might result in different query results and unexpected behavior if two columns are switched around. In particular, this can happen when outer joins are used. Additionally, the Workspace Client does not keep the first column constant. The “first selected field” is the first field in the query, unless the first field is a constant.

After finding the least spanning tree, objects not selected are removed from the tree. The effect is that only objects needed in the query are used in the generated SQL query.

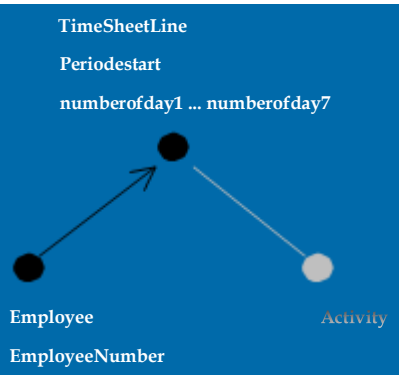
In the following examples, the previously defined universe is used in two queries. In the first query, the first selected field is ActivityNumber from the object Activity, and the least spanning tree contains all of the objects in the universe. Note that the field Regtime is defined using the fields numberofday1, ..., numberofday7 from the object TimeSheetLine.

```
mql 1.3
mselect Activity.ActivityNumber,
        Activity.ActivityText,
        Employee.Name1,
        Regtime
from    timesheetUniverse
where   regtime > 0.0
  and   timesheetline.periodstart = 2001.01.01

order by Activity.ActivityNumber
```



In the second query, the first selected field is EmployeeNumber from the object Employee. The least spanning tree does not contain the Activity object, indicating that the object is not needed in the generated SQL query.

```
mql 1.3
mselect Employee.EmployeeNumber,
        Regtime
from    timesheetUniverse
where   timesheetline.periodstart inrange
        [ 2003.01.01 .. 2003.01.31 ]

order by Employee.EmployeeNumber
```



Reading this Section

The formal syntax of MUL is presented in BNF (Bachus Naur Form).

Definition

This section describes the current version of MUL. For a version history of MUL, see Version History.

Effort has been made to turn MUL into a programmer-friendly language. The syntax of MUL is a tag-based language, and it consists of elements and attributes just like XML. Unlike XML, every attribute in MUL has an associated type and can have a short form.

```
mul ::=
 (<MUL 1.3> | MUL 1.3)
 universe
```

Universe

The universe element defines a universe. The universe element can be used inside another universe element, also known as an internal universe.

```
universe ::=
 <universe (name = module|module) [(title = string|string)] >
   [help]

   object*
   join*
   parameter*
   field*
   restriction*
   interface*

 <end universe>
```

The following are attributes of the universe element, when defining a new universe.

Attribute	Description
name	The name of the universe. The universe name is used when referring to the universe.
[title]	The title of the universe.

In the following example, a very simple universe is defined. The universe name is reference::U001, and uses the relation employee. Users of this universe can only access the fields EmployeeNumber and EmployeeName. All of the other fields from the employee relation cannot be accessed because they are not part of the universe interface.

```
<MUL 1.3>
<universe reference::U001 "Reference Manual Universe 001">
   <object employee>
   <interface>
      .EmployeeNumber :employee.EmployeeNumber
```

```
      .EmployeeName    :employee.Name1
   <end interface>

<end universe>
```

Help

With the help element, external help information regarding the universe can be specified. Help for a universe can be specified using five levels.

1. **name** is the identifier of the universe, and is used for referring to the universe. The name is defined in the universe element.
2. **title** is a descriptive name for the universe, and is used in listings shown to the user. The title is defined in the universe element.
3. **description** is a very short description of what the universe can be used for. The description is for use in universe listings or mouse-over help.
4. **summary** is a short version of the help text.
5. **text** is an in-depth description of what the universe can be used for and how it works.

```
help ::=
 <help [(description =
   string|string)] > [summary]
   [text]
 <help end>

summary ::=
  <summary>
    [text]
  <end summary>

text ::=
  <text (text = text|text) /> | text
```

The following example defines a universe help element where all levels of help are defined.

```
...
<help "MUL Reference Manual, Universe">
  <summary>
    #This Universe is for the MUL Reference Manual#
  <end summary>
    #
    This Universe illustrates the help element, and is used as an
example
    in the MUL Reference Manual.
    #
<end help>

...
```

Objects

An object is a relation in the Maconomy database. An object element in a universe refers to an existing object, refers to an external universe, or creates a “new” object using one of the standard database set operators.

```
object ::=
  objectRef | universeRef | objectUnion
```

Object Reference

An object is a relation in the Maconomy database, defined by Maconomy or added to the system using MOL (Maconomy Object Language). See the *Deltek Maconomy MOL Language Reference*.

```
objectRef ::=
  <object (name = id|id) [basis = id] >
```

The following are attributes of the object element.

Attribute	Description
name	The name of the object used when referring to the object in the universe definition. If the name is an existing object in the Maconomy database, the basis attribute can be omitted.
[basis]	The name of an existing object in the Maconomy database.

In the following example, two references to the Maconomy object Employee are defined and named Employee and SeniorEmployee. This is useful when using the same object in different join situations.

```
...
<object Employee>
<object SeniorEmployee basis=Employee>
<join (Employee, SeniorEmployee) OuterNormal>
  .SeniorEmployee <-> .EmployeeNumber
<end join>
```

Universe Reference

The universe reference element defines an alias for an existing universe. By using the alias, an existing universe can be used just like any other object.

Using an external universe as an object can be used for modularization of a complex universe, and for reuse of already defined universes.

```
universeRef ::=
  <universe (name = id|id) basis = id />
```

The following are attributes of the universe element, when used for referring to an existing universe.

Attribute	Description
name	The alias for the external universe.
basis	The name of an existing universe

In the following example, the external universe reference::U001 is referred and given the name SeniorEmployee. After the alias definition, the fields from the universe can be used just like any other objects.

```
...
<object Employee>
<universe SeniorEmployee basis=reference::U001>
<join (Employee, SeniorEmployee) OuterNormal>
  .SeniorEmployee <-> .EmployeeNumber
<end join>

...
```

Set Operator, Union

The objectUnion element defines a “new” object using the standard database union operator on two objects. The effect of the union operator is that a new object with the union of all rows in the two objects is created.

The fields available from the object defined are the union of all of the fields defined in the two objects. If a field only exists in one of the objects, a constant null value field is automatically created.

```
objectUnion
    ::=
   <objectUnion (name = id|id) [unionAll(+|-)] > object

  object
   <end objectUnion>
```

The following are attributes of the objectUnion.

Attribute	Description
name	The alias for “new” object created.
[unionAll]	Controls the handling of duplicate rows. If unionAll- (default), duplicate rows are eliminated. If unionAll+, duplicate rows are retained.

The following example defines the union object MyUnionObject from two internally defined universes. The object MyUnionObject has four fields: employeeNumber, employeeName, fieldX, and fieldY. Note that the names of the objects X and Y are not used when referring to a field from the object MyUnionObject.

```
...
<objectUnion MyUnionObject>

  <universe X>
    <object Employee>
    <interface>
      .employeeNumber : Employee.EmployeeNumber
      .employeeName   : Employee.Name1
      .fieldX         : type=Boolean :value=true
    <end interface>
  <end universe>

  <universe Y>
    <object Employee>
    <interface>
      .employeeNumber : Employee.EmployeeNumber
      .employeeName   : Employee.Name2
      .fieldY         : type=Boolean :value=true
    <end interface>
  <end universe>

<end objectUnion>

...
```

Join Definition

The join element defines how to join two objects. Together with the objects defined by the object elements, the joins define a graph without cycles. That is, every object defined by an object element must be joined to at least one other object. It is not necessary to define an object element for a Maconomy object before using it in a join.

```
join ::=
  <join (id, id) [(joinType = joinTypeEnum |
   joinTypeEnum)] > (.id <-> .id | .id)*

  <end join>


joinTypeEnum ::=
  Normal | Required | Outer | NormalOuter | OuterNormal
```

A join between two objects contains a list of pairs of join fields. The first field in a pair must exist in the first object, and the second field must exist in the second object. The two fields in the pair must have the same type.

In the following example, the three objects MyEmployee, TimesheetLine, and JobRegistrationBudgetRelation have been joined in a graph without cycles. In the join between the objects TimeSheetLine and MyEmployee, one pair of join fields has been defined using the field .EmployeeNumber as both the first and the second element of the pair. In the join between the objects TimesheetLine and JobRegistrationBudgetRelation, two pairs of join fields are used. Note that no object element is necessary for the objects TimesheetLine and JobRegistrationBudgetRelation.

```
...
<object MyEmployee basis=Employee>

<join (TimeSheetLine, MyEmployee)>
  .EmployeeNumber
```

```
<end Join>

<join (TimeSheetLine, JobRegistrationBudgetRelation)>
  .EmployeeNumber
  .LineNumber <-> .TimeSheetLineNumber
<end join>

...
```

A join is a bidirectional edge between the two objects, and for each direction a join type is associated.

In the examples used for each join type, the following objects and contents are used.

TimeSheetLine Object

EmployeeNumber	LineNumber	ActivityNumber
100	1	200
100	2	201
100	3	201
102	1	

Activity Object

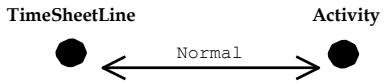
ActivityNumber	ActivityText
200	Design
201	Programming
202	Verification

Join Type, Normal

The join type Normal between the two objects X and Y defines a database normal join from X to Y and a database normal join from Y to X. A database normal join indicates that only rows from an object that has at least one matching row in the other object is selected. The join is only supplied when fields from both objects are selected. The join type Normal is the default value.

In the following example, the Normal join type is used in a universe to define the join between the two objects TimeSheetLine and Activity. The first query selects the object TimeSheetLine as root, and the second query selects the object Activity as root.

```
<mul 1.3>
<universe reference::U002 "Reference Manual Universe 002">
   <join (TimeSheetLine, Activity)>
      .ActivityNumber
   <end join>
<end universe>
```

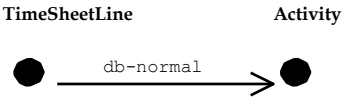


```
<mul 1.3>
<universe reference::U002 "Reference Manual Universe 002">
   <join (TimeSheetLine, Activity)>
```

Query 1:

```
mselect Timesheetline.EmployeeNumber,
        Timesheetline.ActivityNumber,
        Activity.ActivityText
from reference::U002
order by Timesheetline.EmployeeNumber
```

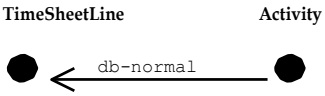
EmployeeNumber	ActivityNumber	ActivityText
100	200	Design
100	201	Programming
100	201	Programming



Query 2:

```
mselect Activity.ActivityNumber,
        Activity.ActivityText,
        Timesheetline.EmployeeNumber,
        Timesheetline.LineNumber
from reference::U002
order by Activity.ActivityNumber
```

ActivityNumber	ActivityText	EmployeeNumber	LineNumber
200	Design	100	1
201	Programming	100	2
201	Programming	100	3



Join Type, Required

The join type Required is the same as the join type Normal, but the join is always supplied, even if no fields from the objects are selected. This join type is to be used if the Normal join has a semantic meaning. This join type reduces the optimization possibilities, so use it with care.

In the following example, the Required join type is used in a universe to define the join between the two objects TimeSheetLine and Activity. The query only selects fields from the object Activity, but as the result reveals, the join of the Object TimeSheetLine is enforced.

```
<mul 1.3>
<universe reference::U003 "Reference Manual Universe 003">
   <join (TimeSheetLine, Activity) Required>
     .ActivityNumber
   <end join>

<end universe>
```

```
mselect Activity.ActivityNumber,
        Activity.ActivityText
from reference::U003
order by Activity.ActivityNumber
```

ActivityNumber	ActivityText
200	Design
201	Programming
201	Programming

Join Type, Outer

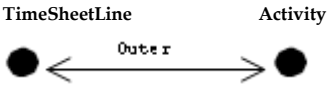
The join type Outer between the two objects X and Y defines a database outer join from X to Y and a database outer join from Y to X. A database outer join indicates that all rows from an object are selected, even if no matching rows are found in the other object. The join is only supplied when fields from both objects are selected.

If rows are found in the joined object, the database outer join behaves just like a database normal join. If no rows are found in the joined object, fields from this object are filled with the Maconomy null value matching the type of each field. Note that the null value is **not** the database null value.

Note that if one of the key fields of the outer joined object is the Maconomy null value, no rows were found in the joined object.

In the following example, the Outer join type is used in a universe to define the join between the two objects TimeSheetLine and Activity. The first query selects the object TimeSheetLine as root, and the second query selects the object Activity as root. A new field outerExists has been defined in the first query, indicating whether or not rows were found in the outer joined object.

```
<mul 1.3>
<universe reference::U004 "Reference Manual Universe 004">
   <join (TimeSheetLine, Activity) Outer>
      .ActivityNumber
   <end join>

<end universe>
```

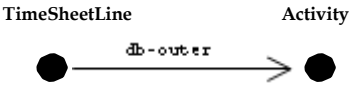


```
<mul 1.3>
<universe reference::U004 "Reference Manual Universe 004">
   <join (TimeSheetLine, Activity) Outer>
```

Query 1:

```
mselect Timesheetline.EmployeeNumber,
        Timesheetline.ActivityNumber,
        Activity.ActivityText,
        Activity.ActivityNumber!=string'null as
outerExists
from reference::U004
order by Timesheetline.EmployeeNumber
```

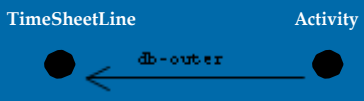
Employee Number	ActivityNumber	ActivityText	outerExists
100	200	Design	true
100	201	Programming	true
100	201	Programming	true
102	string'null	string'null	false



Query 2:

```
mselect Activity.ActivityNumber,
        Activity.ActivityText,
        Timesheetline.EmployeeNumber,
        Timesheetline.LineNumber
from reference::U004
order by Activity.ActivityNumber
```

Activity Number	ActivityText	EmployeeNumber	LineNumber
200	Design	100	1
201	Programming	100	2
201	Programming	100	3
202	Verification	string'null	integer'null

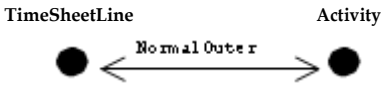


Join Type, NormalOuter

The join type **NormalOuter** is an asymmetric join between two objects X and Y. It defines a database normal join from X to Y, but a database outer join from Y to X. The join is only supplied when fields from both objects are selected.

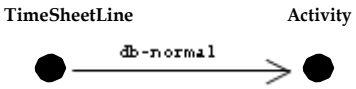
In the following example, the NormalOuter join type is used in a universe to define the join between the two objects TimeSheetLine and Activity. The first query selects the object TimeSheetLine as root, and the second query selects the object Activity as root.

```
<mul 1.3>
<universe reference::U005 "Reference Manual Universe 005">
    <join (TimeSheetLine, Activity) NormalOuter>
      .ActivityNumber
    <end join>
<end universe>
```



Query 1:

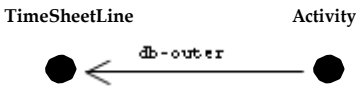
```
mselect Timesheetline.EmployeeNumber,
        Timesheetline.ActivityNumber,
        Activity.ActivityText
from reference::U005
order by Timesheetline.EmployeeNumber
```



Employee Number	ActivityNumber	ActivityText	outerExists
100	200	Design	true
100	201	Programming	true
100	201	Programming	true

Query 2:

```
mselect Activity.ActivityNumber,
        Activity.ActivityText,
        Timesheetline.EmployeeNumber,
        Timesheetline.LineNumber
from reference::U005
order by Activity.ActivityNumber
```



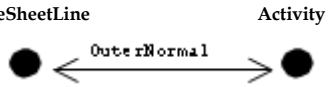
Activity Number	ActivityText	EmployeeNumber	LineNumber
200	Design	100	1
201	Programming	100	2
201	Programming	100	3
202	Verification	string'null	integer'null

Join Type, OuterNormal

The join type OuterNormal is an asymmetric join between two objects X and Y. It defines a database outer join from X to Y, but a database normal join from Y to X. The join is only supplied when fields from both objects are selected.

In the following example, the OuterNormal join type is used in a universe to define the join between the two objects TimeSheetLine and Activity. The first query selects the object TimeSheetLine as root, and the second query selects the object Activity as root.

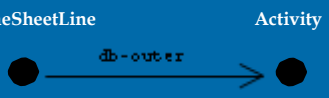
```
<mul 1.3>
<universe reference::U006 "Reference Manual Universe 006">
   <join (TimeSheetLine, Activity) OuterNormal>
      .ActivityNumber
   <end join>
<end universe>
```



Query 1:

```
Mselect Timesheetline.EmployeeNumber,
        Timesheetline.ActivityNumber,
        Activity.ActivityText
From reference::U006
Order by Timesheetline.EmployeeNumber
```

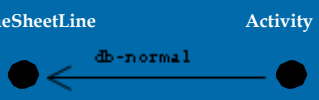
EmployeeNumber	ActivityNumber	ActivityText
100	200	Design
100	201	Programming
100	201	Programming
102	string’ null	string’ null



Query 2:

```
Mselect Activity.ActivityNumber,
        Activity.ActivityText,
        Timesheetline.EmployeeNumber,
        Timesheetline.LineNumber
From reference::U006
Order by Activity.ActivityNumber
```

Activity Number	ActivityText	EmployeeNumber	LineNumber
200	Design	100	1
201	Programming	100	2
201	Programming	100	3



Parameter Definition

A universe can be parameterized, allowing fields and restrictions to be dependent on values supplied at run time. The current use of parameters is subject to change and should be used with care.

```
parameter ::=
  (
     <parameter (name = id|id) (type = typeid|typeid) [(title = string|string)]
           [value = constExpressionShort|constExpressionShort]
     />
  ) |
  (    id :(type = typeid|typeid) [:(title = string|string)]
          [:(value = constExpressionShort|constExpressionShort)]
  )
```

When a formal parameter is defined, it can be used in expressions just like any other field. If no value is given to the parameter at run time, the default value is used.

Parameters defined in a universe are considered global. This means that if a universe uses an external universe, and the external universe defines a parameter, that parameter is known in the universe.

In the following example, the parameter parmEmployeeNumber is defined and used for parameterization of a restriction.

```
<mul 1.3>
<universe reference::U007 "Reference Manual Universe 007">
  <join (TimeSheetLine, Activity)>
    .ActivityNumber
  <end join>

  <parameter parmEmployeeNumber type=string>

  <restriction>
    TimeSheetLine.EmployeeNumber = parmEmployeeNumber
  <end restriction>

<end universe>
```

Field Definition

New fields can be defined in a universe using the field definition element.

```
field ::=
  fieldExp | fieldValue | fieldBasis | fieldHidden

fieldExp ::=
  <field (name = id|id) [title = string|string] [description =
    string] > expressionShort
  <end field>

fieldValue ::= (
     <field (name = id|id) [title = string|string] [description =
           string] (value =
           constExpressionShort|constExpressionShort)
     />
  ) |
 (   .id [:(title = string|string)] [:description = string]
         (:(value = constExpressionShort|constExpressionShort))
  )

fieldBa
  sis
  ::= (
     <field (name = id|id) [title = string|string] [description =
            string] (basis = qualifiedfieldid|qualifiedfieldid)
     />
  ) |
  (
     .id [:(title = string|string)] [:description =
         string] (:(basis =
         qualifiedfieldid|qualifiedfieldid))
```

The following are attributes of the field definition element.

Attribute	Description
name	The name of the new field.
[title]	The title of the new field.
[description]	A short description of the new field.

Attribute	Description
[value]	The new field is defined as a constant with this value. Used in connection with the short version of the field definition element.
[basis]	The new field is defined as an alias for this field. Used in connection with the short version of the field definition element.

A field is defined by an expression, and the type of the field is inferred from this expression. The Maconomy functions available in the expression are listed in Maconomy Functions.

Two short versions of a field definition are available, one defining a constant, and one defining an alias for another field.

In the following example, three fields are defined in a universe. The first field is defined using the Maconomy function if, the second field is defined using the constant 42, and the third field is defined as an alias for the field Employee.Name1.

```
<mul 1.3>
<universe reference::U008 "Reference Manual Universe 008">
  <object Employee>

  <field fieldFunction>
     if(Employee.SalesEmployee, "Sales person", "Not a sales person")
  <end field>
  .fieldConstant :value=42
  .fieldAlias    :Employee.Name1

<end universe>
```

Restriction Definition

A restriction can be defined in a universe using the restriction definition element.

```
restictionDefinition ::=
  <restriction [required+|required-] >
    expressionShort

  <end restriction>
```

A restriction specified in the universe is only in effect if a field from a selected object is used in the restriction. If the attribute required+ is used, then the restriction is always in effect.

In the following example, a restriction using only fields from the object Activity is defined. The first query only selects fields from the object TimeSheetLine. Because no fields are selected from the object Activity, the restriction is not in effect. The second query selects fields from the object Activity, and therefore the restriction is in effect. The data used in this example is defined in Join Definition.

```
<mul 1.3>
<universe reference::U009 "Reference Manual Universe 009">
  <join (TimeSheetLine, Activity) OuterNormal>
    .ActivityNumber
  <end join>

  <restriction>
```

```
    like( "%mm%", Activity.ActivityText )
  <end restriction>
<end universe>
```

Query 1:

```
mselect Timesheetline.EmployeeNumber,
        Timesheetline.ActivityNumber
from reference::U009
order by Timesheetline.EmployeeNumber
```

EmployeeNumber	ActivityNumber
100	200
100	201
100	201
102	string′null

Query 2:

```
mselect Timesheetline.EmployeeNumber,
        Timesheetline.ActivityNumber,
        Activity.ActivityText
from reference::U009
order by Timesheetline.EmployeeNumber
```

EmployeeNumber	ActivityNumber	ActivityText
100	201	Programming
100	201	Programming

In the following example, a required restriction using only fields from the object Activity is defined. The query only selects fields from the object TimeSheetLine, but because the restriction is required, the join to the Activity is forced, and the restriction is in effect. The data used in this example is defined in Join Definition.

```
<mul 1.3>
<universe reference::U010 "Reference Manual Universe 010">
  <join (TimeSheetLine, Activity)>
    .ActivityNumber
  <end join>

  <restriction required+>
    like( "%mm%", Activity.ActivityText )
  <end restriction>

<end universe>
```

Query 1:

```
mselect Timesheetline.EmployeeNumber,
        Timesheetline.ActivityNumber
from reference::U010
order by Timesheetline.EmployeeNumber
```

EmployeeNumber	ActivityNumber
100	201
100	201

Interface Definition

An interface is a specification of how users are to see the universe. An interface can be seen as a long list of fields, where the fields can be divided into groups.

```
interface ::=
  <interface [name = id|id] [basis = qualifiedfieldidList] [sumOnAmount(+|-)] /> |
  <interface [name = id|id] [basis = qualifiedfieldidList] [sumOnAmount(+|-)] >
    field | fieldHide | interfaceGroup | interfacegroupHide
  <end interface>

fieldHide ::=
  <field (name = id|id) hidden+ />   |
  .id : hidden+

qualifiedfieldidList ::=
  qualifiedfieldid | [qualifiedfieldid (, qualifiedfieldid)* ]
```

The following are attributes of the interface elements.

Attribute	Description
[name]	The name of the interface. If a name is not assigned to an interface, the interface is considered the default interface.
[basis]	A list of object/interface group names. Fields and groups from each element in the list are added to this interface.
[sumOnAmount]	Indicates that amount fields added to the interface with the basis attribute are wrapped with the group operator SUM. The default value is sumOnAmount- indicating that amount fields are not wrapped.

A universe can contain more than one interface, but a query using a universe can only select fields from one interface at a time. If no interface is specified, the objects and their fields are visible and can be used directly in a query.

All of the fields and subgroups from an object/object-interfacegroup can be copied to the interface with the basis attribute. If two items with the same name are added to the interface, the item from the first-mentioned element in the basis attribute has precedence. If a field is defined using the group operator SUM, the field added to the interface is also defined using the group operator SUM.

Fields and subgroups can be hidden using the special hidden element, which hides a previously defined field/subgroup.

In the following example, an interface is defined taking all of the fields from the objects TimeSheetLine and Activity, except for the VATCode field, which is hidden. The field ActivityNumber exists in both objects, but is taken from TimeSheetLine because it is mentioned first in the list of basis objects.

```
<MUL 1.3>
<universe reference::U011 "Reference Manual Universe 011">
   <join (TimeSheetLine, Activity) OuterNormal>
     .ActivityNumber
   <end join>
   <interface basis = [TimeSheetLine, Activity] >
      .VATCode :hidden+

   <end interface>

   <end universe>
```

In the following example, an interface is defined taking all of the fields from the objects JobHeader and JobEntry. Because the attribute SumOnAmount+ is used, all amount fields in the two objects are added to the interface using the group operator SUM, for example, the field CostPriceInvoiced available in the universe is defined by SUM(JobEntry.CostPriceInvoiced). The field NumberHoursRegistered has the type real, and to turn the field into a SUM –field, an explicit definition is used.

```
<MUL 1.3>
<universe reference::U012 "Reference Manual Universe 012">
   <join (JobHeader, JobEntry) OuterNormal>
     .JobNumber
   <end join>
   <interface basis = [JobHeader, JobEntry] SumOnAmount+>
     <field NumberHoursRegistered>
       SUM(JobEntry.NumberHoursRegistered)
     <end field>
   <end interface>
<end universe>
```

In the following example, an interface is defined taking all of the fields from the previously defined universe reference::U012. All fields are defined using a group operator SUM, that is, all amount fields and the field NumberHoursRegistered are defined in the interface using the group operator SUM.

```
<MUL 1.3>
<universe reference::U013 "Reference Manual Universe 013">
   <Universe MyUniverse basis=reference::U012 />
   ...
   <interface basis=[MyUniverse]/>
<end universe>
```

Interfacegroup Definition

Fields in the interface can be grouped using the interfacegroup element. This can be used for logical grouping of fields.

```
interfaceGroup ::=
  <interfacegroup [name = id|id]
                  [basis = qualifiedfielddidList] [sumOnAmount(+|-)] [title
                  = string|string]
  >
    field | interfaceGroup | interfaceGroupHide
  <end interfacegroup>

interfaceGroupHide ::=
  <interfacegroup [name = id|id] (hidden+|hidden-) />
```

The following are attributes of the interfacegroup elements.

Attribute	Description
name	The name of the group.
[title]	The title of the group.
[basis]	A list of object/interface group names. Fields and groups from each element in the list are added to this group.
[sumOnAmount]	Indicates that amount fields added to the interface with the basis attribute are wrapped with the group operator SUM. The default value is sumOnAmount- indicating that amount fields are not wrapped.

Fields can be copied to the interfacegroup with the basis attribute exactly as it is done in the interface element.

In the following example, three interfaces are given to the same universe. The first interface is a long list of fields, the second simulates the same grouping of fields as defined by the objects, and the third defines a limited interface.

```
<MUL 1.3>
<universe reference::U014 "Reference Manual Universe 014">
   <join (TimeSheetLine, Activity) OuterNormal>
     .ActivityNumber
   <end join>
   <field regtime>
     SUM(timesheetline.numberofday1 + timesheetline.numberofday2)
   <end field>

   -- First interface, long list of fields
   <interface basis = [TimeSheetLine, Activity] >
     .regtime :.regtime
   <end interface>

   -- Second interface, Object grouping of fields
   <interface objectView>
     <interfacegroup TimeSheetLine basis=TimeSheetLine />
     <interfacegroup Activity basis=Activity />
     .regtime :.regtime
```

```
    <end interface>

    -- Second interface, limited interface
    <interface limitView>
      <interfacegroup TimeKeys>
        .Empoloyee :TimesheetLine.EmployeeNumber
        .Activity :Activity.ActivityNumber
      <end interfacegroup>

      <interfacegroup TimeNumbers>
        .Monday :timesheetline.numberofday1
        .Thursday :timesheetline.numberofday2
        .weekSum :.regtime
      <end interfacegroup>
    <end interface>
<end universe>
```

Common Syntax Elements

See the *Deltek Maconomy Language Reference MQL*.

Maconomy Functions

A wide range of functions is available for use in expressions. For each function, a type scheme is defined. If the types of the arguments to a function do not match the type scheme, a type error is given.

Note that even if the database is known, database functions are not available.

See the *Deltek Maconomy Language Reference MQL* for a list of functions.

Version History

MUL 1.3

Inheritance of SUM fields from sub-universe when using basis attribute on interface and interfacegroup elements.

New SumOnAmount attribute on the interface and interfacegroup tags, for automatic generation of SUM fields on all amount fields inherited using the basis attribute.

If no interface is defined in a universe, an interface is created to inherit SUM fields. Available objects are created as matching interfacegroups and fields at the top level in the interface.

MUL 1.2

Initial version.

Workspace Performance Boost

Introduction

Deltek Maconomy 2.2.x introduces the concept of *workspaces*. Workspaces are designed to present users with the information that they need to perform specific tasks—when they need it. Previous versions of Maconomy are built on a *dialog* concept. Each dialog presents access to one kind of business entity (such as a job, a time sheet, or an invoice). The Deltek Maconomy 2.2.x application contains several hundreds of such dialogs.

Workspaces can include large portions of the business functionality. Workspaces are built using panels, which are structured in a hierarchical manner to guide a user toward more dedicated functionality. Each of these panels presents a part of a dialog. Sometimes a single workspace may consist of more than hundred dialogs and several hundred panels. For this reason, opening a dialog in the former clients cannot really be compared with opening a workspace in the Maconomy 2.2.x workspace client.

A poorly planned and built workspace may perform inadequately, compared to a well-planned and constructed workspace and the former dialogs.

This document explains:

- Mechanisms that are involved in loading data for a workspace
- Constructing well planned workspaces
- Tips for optimizing workspace performance
- Modifying workspaces that do not perform satisfactorily

This document is only a guideline. It cannot guarantee that any workspace will perform satisfactorily, even if these guidelines are met. Many factors can influence the performance, including client machine OS, client machine hardware, database content, database indexes, server hardware, server activity, network latency, and network bandwidth.

In addition, this document addresses areas in constructing workspaces that can inhibit efficiency, and thus need special attention, including:

- Workspaces that use cross-references (fields and/or actions)
- Workspaces that contain statically <Hidden>-panels
- Dynamic workspaces (parts of the workspace may appear/disappear dynamically, depending on the data in the workspace)

Workspace Data Fundamentals

Before exploring the advanced aspects of performance tuning workspaces, it is important to understand how data is fundamentally distributed in a workspace.

A workspace is organized as a tree of panels. At the top, you have one panel. Below that one, you may have several panels, each depending on the data that is shown or selected in the top panel. Each of these panels may again have panels below them, which depends on the data. This structure may be repeated any number of times.

Example 1

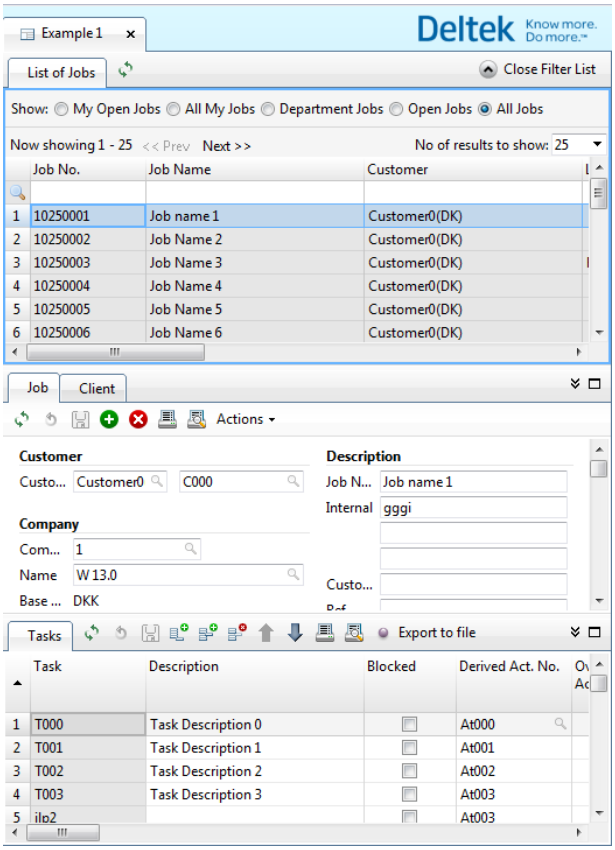
The following is a simple workspace with four panels:

- A filter for selecting a job
- A tab that displays detailed information about the job
- A tab that displays information about the client of the selected job
- An additional panel that displays the list of tasks that are associated with that particular job when you select the tab to display job details

The workspace appears as follows:

```
<Filter source="Jobs" title="List of Jobs">
  <With>
    <Card title="Job">
      <Bind foreignKey="primary">
        <Table source="JobTasks" title="Tasks"/>
      </Bind>
    </Card>
  </With>
  <Bind foreignKey="CustomerNumber_Customer">
    <Card source="CustomerCard" title="Client"/>
  </Bind>
</Filter>
```

The related screen is displayed as follows:



When data is loaded in this workspace, the filter panel contains a list of all jobs (limited by the number of search results selected by the user), as follows:

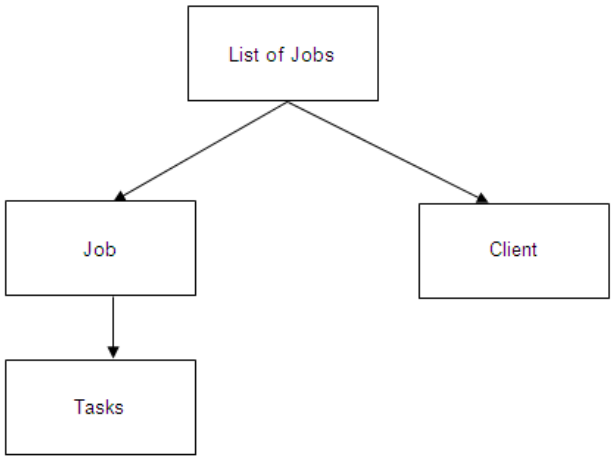
- The Jobs panel is loaded with the job data of the job that is selected in the filter.
- The Tasks panel is loaded with the list of tasks that are associated with the selected job.
- The Client panel is loaded with the information about the client of the selected job.

As a user browses among the jobs, the contents of the Job, Tasks, and Client panels are continuously updated with the appropriate information.

In the former Maconomy clients, you needed to open three different dialogs to accomplish the same thing: The Jobs dialog, the JobTasks dialog. and the CustomerCard dialog.

Workspace Structure

The workspace in Example 1 has a tree-like structure, as follows:



The arrows indicate the direction of data dependency. Panels always depend on data in the panel(s) that are above them in the tree structure. Panels are described as follows:

- Parent panel — The panel that is immediately above a given panel. In the preceding example, List of Jobs is the parent.
- Child panel — The panel(s) below a given panel. In the preceding example, Job, Client, and Tasks are the children.

> In this, the children are displayed below their parent. However, workspaces may show child panels below their parent, to the right-hand side of their parent, or both. This is only for display and does not alter behavior. When drawing a workspace tree-structure, you may choose either display.

Example 2

You can change the workspace from Example 1 to make the Client panel appear to the right-hand side of the filter. Drawing the workspace as a tree structure would give the same result as previously shown, because the Job and Client panels still have the same parent, the List of Jobs filter.

The specification for such a workspace is:

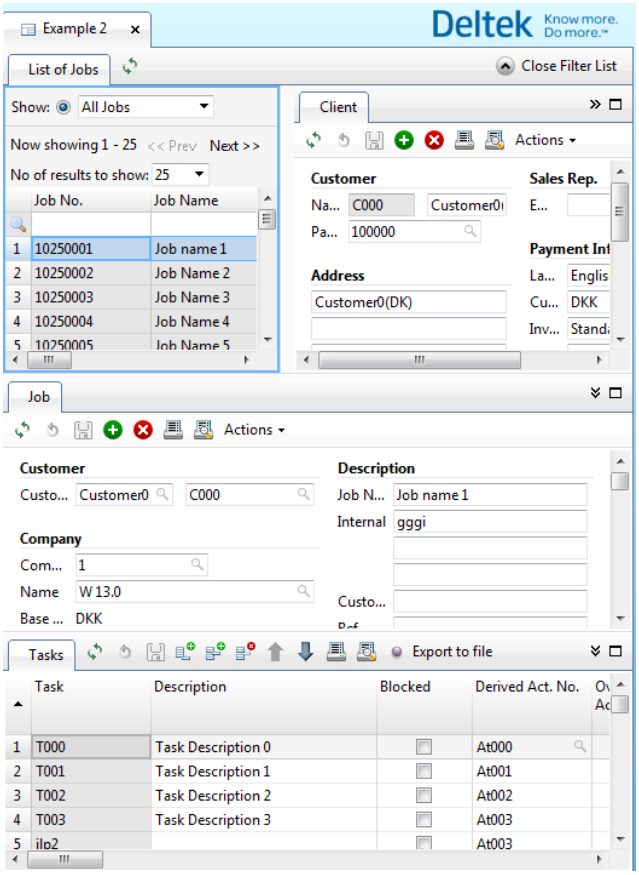
```
<Filter source="Jobs" title="List of Jobs">
  <Expansions>
    <With>
      <Card title="Job">
        <Bind foreignKey="primary">
          <Table source="JobTasks" title="Tasks"/>
        </Bind>
      </Card>
    </With>
  </Expansions>
  <Assistants>
    <Bind foreignKey="CustomerNumber_Customer">
      <Card source="CustomerCard" title="Client"/>
    </Bind>
  </Assistants>
</Filter>
```

Example 2 differs from Example 1 in that the Expansions and Assistant tags are added. These tags are used to indicate how child panels are visually organized relative to the parent pane, as follows:

- Assistants — An assistant is displayed on the right-hand side.
- Expansions — An expansion is displayed below.

For both, the panels (including new expansions and assistants) can be recursively embedded.

The data structure of the Example 2 workspace is displayed differently, as shown in the following figure.



Now *both* the Job panel *and* the Client panel are displayed at the same time. In Example 1, you must select one or the other.

In the following, it does not matter whether panels are shown to the right-hand side (assistants) or below (expansions). This does not affect the performance of a workspace. However, showing data in an expansion as well as an assistant means that because more data is visible at once, more data must be loaded (as detailed in the following).

Working with Larger Workspaces

Examples 1 and 2 show examples of a simple workspace. However, it is not uncommon to have hundreds of panels in a workspace. Some workspaces exceed 1000 panels. Loading data for all of the panels all of the time would severely impact the performance of such workspaces. For this reason, only the necessary parts of a workspace are loaded. When a user reveals a new part of the workspace, the additional data is automatically loaded on demand.

For the workspace screen shot in Example 1, data is loaded for:

- The filter panel (List of Jobs)
- The Job panel
- The Task list panel

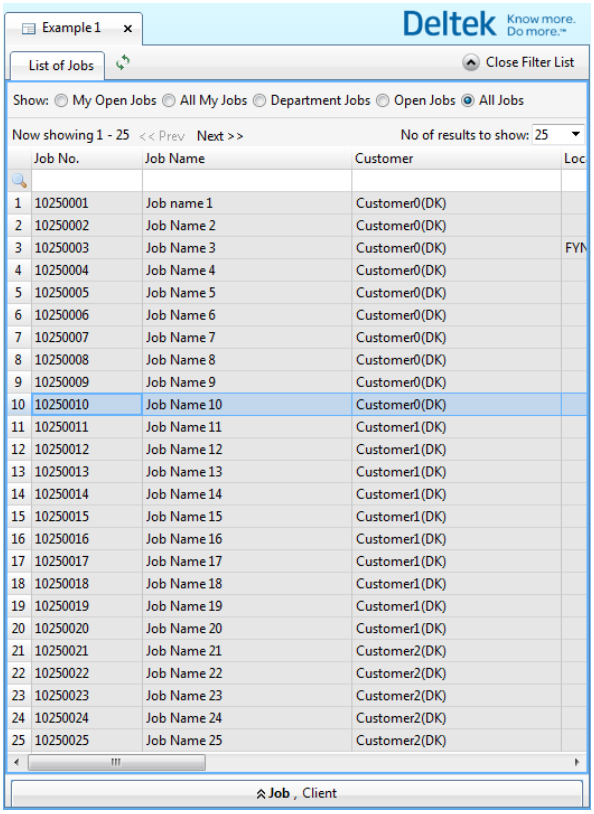
Data is *not* loaded for the Client panel, because that panel is not currently visible. Instead, it loads as the user selects to reveal that portion of the workspace. This allows you to avoid the performance penalty of loading data for panels for which the content is not visible. If the user does not reveal the Client tab in a specific flow, that data is never loaded, thereby saving time. This is particularly important with larger workspaces.

The workspace data structure is the same as for Examples 1 and 2, with one important difference: in Example 2, all of the panels are visible, and in Example 1, they are not. This means that for the workspace in Example 2, you must load data for all of the panels. Still, if the workspace contained several hundreds of panels, you would only need to load data for the four visible panels, again saving a huge amount of time.

Minimizing Expansions and Assistants

To keep unnecessary data from loading, minimize expansions and assistants. This keeps corresponding panels from being displayed.

The following screen shot uses the workspace from Example 1, with only the filter visible. The remaining portion of the workspace is minimized, revealing only the titles of the immediate children. The result is that only data for the filter is loaded. Selecting different records in the filter is faster because no additional data is loaded.



A workspace designer cannot know which parts a user will choose to expand or reveal, and thus it is impossible to determine the exact performance of a workspace. However, it helps in design to remember that the data-loading engine always loads only what is needed.

Advanced Workspace Concepts

The previous section explained that workspaces always automatically behave optimally, because only the needed data is loaded. However, many workspaces require features that are more advanced. When using advanced features, always consider performance. Workspaces can be designed in slightly different ways, enabling some to perform better than others.

Using Glue (Hidden) Panels

Workspaces can contain panels that are statically hidden. This feature is used when information in these panels is used for cross referencing, or when they are needed as glue between panels.

Example 3

You want to design a workspace that shows a list of clients that can be filtered. For the selected client, you want to show which of your employees live in the same postal district as that client, so that you can determine which employees can easily get to a particular client in a very short period of time.

You could design such a workspace as:

```
<Filter source="CustomerCard" title="List of Clients">
  <Bind foreignKey="PostalDistrict_1">
    <Card source="PostalDistricts">
      <Restrict reverseForeignKey="PostalDistrict_1">
        <Filter source="Employees" title="Nearby Employees"/>
      </Restrict>
    </Card>
  </Bind>
</Filter>
```

This shows an inefficient workspace. While it does show the relevant employees for each selected client, it also shows the middle panel that specifies details about the postal district. You do not need that information. However, because the employee table does not contain a foreign key that maps to customers who have the same postal district, you do need this. The middle panel acts as glue between the client and the employees. However, the workspace would be displayed more efficiently if the glue panel was not displayed.

Hiding Glue Panels

There are two ways to hide glue panels:

- Designate the panel as statically hidden (always loads)
- Use stepping-stone panels (does not load)

Using Statically Hidden Panels

You can achieve the required visual appearance by designating the Postal district panel as statically hidden. This is displayed as follows:

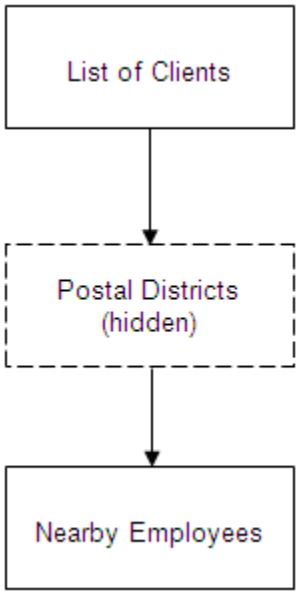
```
<Filter source="CustomerCard" title="List of Clients">
  <Bind foreignKey="PostalDistrict_1">
    <Hidden source="PostalDistricts">
      <Restrict reverseForeignKey="PostalDistrict_1">
        <Filter source="Employees" title="Nearby Employees"/>
      </Restrict>
    </Hidden>
```

```
  </Bind>
</Filter>
```

Now the workspace visually displays only two panels, but it still includes the glue data.

Remember that although the Postal District panel is not visible, it is still loaded. It is always loaded when the Nearby Employees panel is loaded, because the Nearby Employees panel depends on the data in the hidden panel.

The following shows the workspace data dependency tree.



This graphic demonstrates the dependency of the Postal Districts panel on the Nearby Employees panel.

Using Stepping-Stone Panels

In Example 3, you need to load a panel that you only need for technical reasons. Sometimes, it is possible to get rid of such panels. When you want to connect a filter panel, use a stepping-stone table in the database. The two panels to link together must both have a foreign key reference to that stepping-stone database table.

In the previous example, you need to link the List of Clients with the Nearby Employees. The Postal District is the stepping stone. Therefore, you can optimize the preceding workspace by specifying this indirect relationship between the two panels in the `Restrict`-binding.

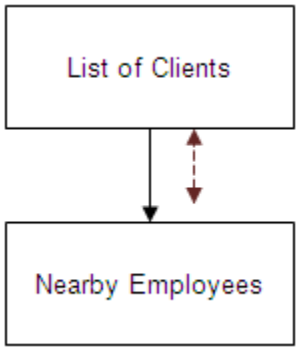
Specify the workspace of Example 3 as:

```
<Filter source="CustomerCard" title="List of Clients">
  <Restrict reverseForeignKey="PostalDistrict_1" foreignKey="PostalDistrict_1">
    <Filter source="Employees" title="Nearby Employees"/>
  </Restrict>
</Filter>
```

The `Restrict`-specification has a reverse foreign key and a (forward) foreign key. The reverse foreign key is defined on the lower panel. The (forward) foreign key is defined on the upper panel. The reverse and forward foreign keys must point to the same entity table—in this case, the postal district.

The benefit of designing the workspace like this is that you avoid loading the postal district (glue) panel. There is no extra cost associated with applying both a reverse and a forward foreign key. Thus, you need to consider and potentially load only two panels instead of three.

You can draw the workspace data dependency tree as:



The dashed arrows that indicate that the link between the two panels is performed via a stepping-stone that acts as glue, but that no additional data is loaded on this account.

Dynamic Workspaces

Parts of the workspace can be hidden or revealed based on such things as the role of current user and the department where the user works. Workspaces can be more powerful and act dynamically based on the data that is currently shown.

For example, you might want to add a panel to a job-related workspace showing the list of sub-jobs pertaining to the selected job—but only if the job is not already a sub-job itself (in which case it can never have sub-jobs). In addition, if the job is itself a sub-job, you might want to reveal a panel that shows information about the corresponding main job—but if the job is not associated with a main job, you do not want this panel to be visible. Thereby, the workspace dynamically hides and reveals various parts of a workspace, depending on the data that is being shown.

You can design this by specifying an expression that determines when to hide the given panel (and all of its children and further descendants). These expressions can refer to any data in the panel itself, the data in the parent, the parent's parent, and so on. In other words, data in the pane itself and all of its direct ancestors can be used to specify this condition.

In addition to this, you can refer to a number of functions that more indirectly relate to data in the current pane:

- `hasNoRecords()` — This evaluates as true if the pane contains no records for the current data selection.
- `hasNoSeed()` — This evaluates as true if the binding into the panel is ill-defined. For instance, if you have a job without a specified project manager, the foreign key that links the project manager employee is ill-defined. Any panel that follows a binding with that foreign key would in this case be un-seeded. Because some dialogs may be able to show records for empty keys, this condition must be evaluated based on the data in the pane itself. The difference between this function and `hasNoRecords()` is subtle, and is not detailed in this document.
- `hasNoChildren()` — This evaluates to true if this panel has no (visible) children.

While these functions may be convenient, they may also result in poor performance, for the reasons in this scenario:

Assume that you have a number of panels, at a certain level (visualize each as a row of tabs). One of them, tab *H*, is specified as hidden if it contains no records. If one of the other tabs is selected, you do not show data for *H*, so normally, you would not load data for this panel. However, to determine whether the tab *H* should be displayed, you must evaluate the hide condition. Because the hide condition refers to data in *H*, you must to load the data in *H*, no matter which of the tabs is selected. This incurs a performance overhead. Do not assume that data loading is more efficient because the pane is hidden. Data-loading is actually less efficient.

Example 4

Consider the case where you want a filter of jobs, followed by a panel that shows the details of the selected job.

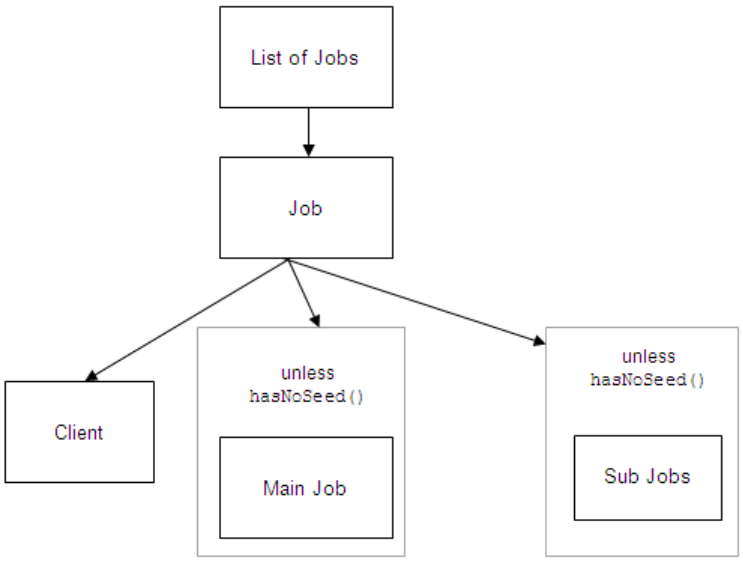
As child panels, you want:

- A panel that shows details about the client of the job
- A panel that shows the main job of the selected job (if it has any)
- A panel that shows the list of sub-jobs of the selected job (if it has any)

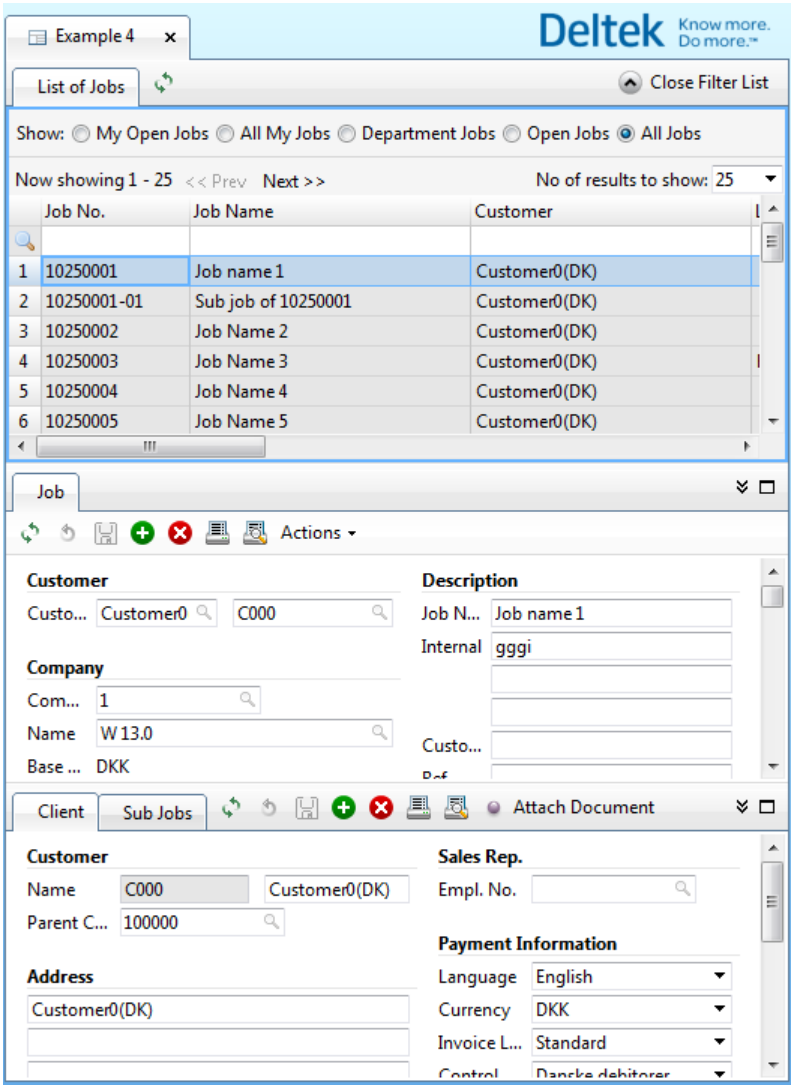
You specify this workspace as follows:

```
<Filter source="Jobs" title="List of Jobs">
  <Expansions>
    <With>
      <Card view="original" title="Job">
        <Bind foreignKey="CustomerNumber_Customer">
          <Card source="CustomerCard" title="Client"/>
        </Bind>
        <Bind foreignKey="MainJobNumber_JobHeader">
          <Card source="MainJobs" title="Main Job" hidden="hasNoSeed()"/>
        </Bind>
        <Bind foreignKey="primary">
          <Table source="MainJobs" title="Sub Jobs" hidden="hasNoSeed()"/>
        </Bind>
      </Card>
    </With>
  </Expansions>
</Filter>
```

This is the tree:

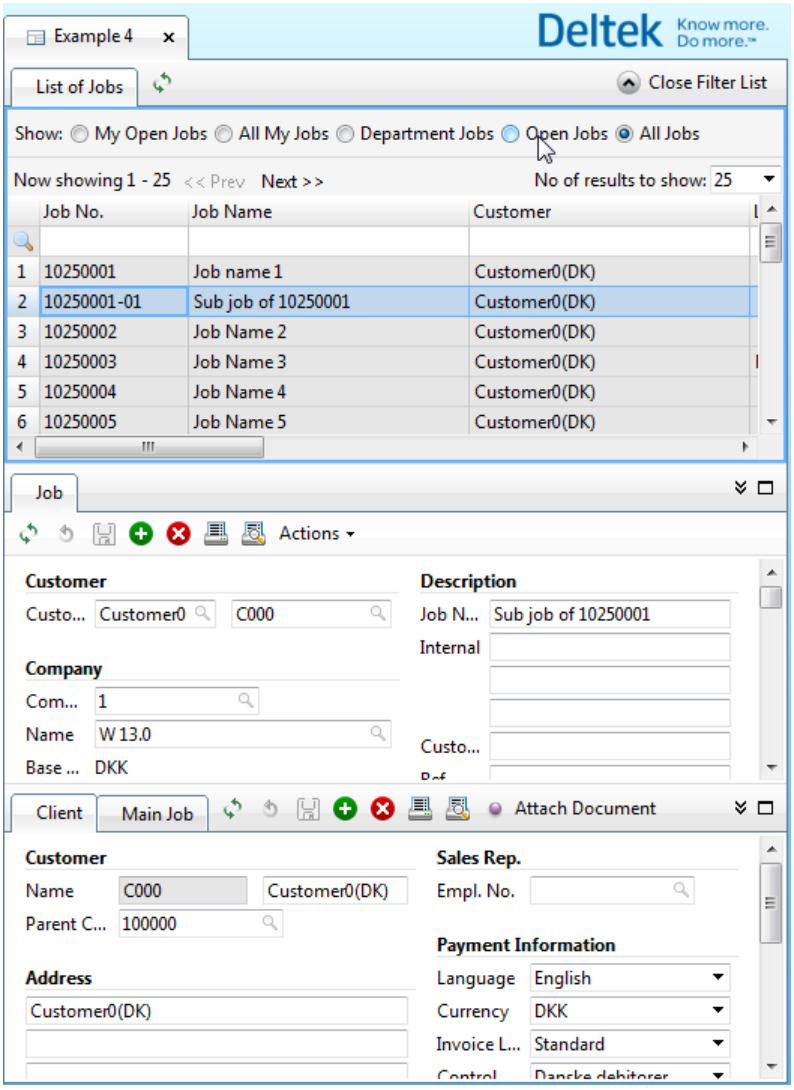


For this workspace, this screen is displayed:



Notice that in this case, the tab *Sub Jobs* is visible, and *Main Job* is not.

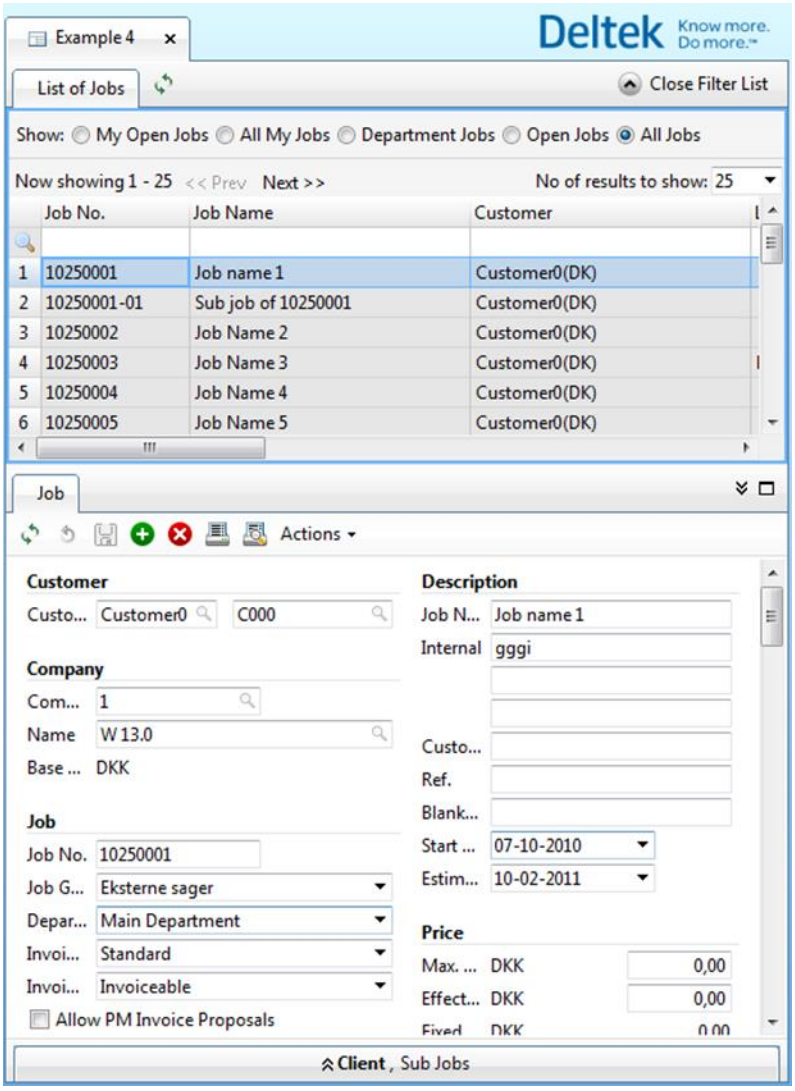
For sub-jobs (jobs that have a main job), the same workspace would be displayed as:



In this case, the *Main Job* tab is visible, and the *Sub Job* tab is not.

You have achieved what you want, but with an incurred cost: the Main Job and Sub Jobs panels are loaded all the time. Even in the case where a user minimizes the corresponding part, it is necessary to load the data for these two panels, because the appearance of the minimized part displays all of the available tab titles.

Because the visibility of the tab titles that pertains to these two panels is dynamic, and because the dynamics are based on the data in those panels, it is necessary to load the data in the panels.



Even in this case where the content of the Sub Jobs and Main Job panels is not visible, it is necessary to load data for those panels because of the way in which the workspace is specified.

To make this workspace perform better, you must consider the kind of data that is available in the workspace, and what is needed. It is possible to change the hide expressions so that the additional data load can be avoided.

To do this, determine a similar hide expression that is based on expressions that refer to data in parent panels. You must determine whether certain data field values in the parent panels is correlated with the condition you are interested in, and you can express the hide condition based on that.

Because in this case, the data of the dynamic panel is not referred, there is no need to load data for it, unless it is explicitly revealed by an end-user—similar to the non-dynamic case.

Example 5

This example, shows how the workspace in Example 4 can be changed to avoid the additional data load that is incurred by that workspace. First, determine if the Job panel contains a field, `MainJobNumber`,

that is blank for jobs that have or can have sub-jobs associated with them, and that is non-blank for jobs that are associated with a main job.

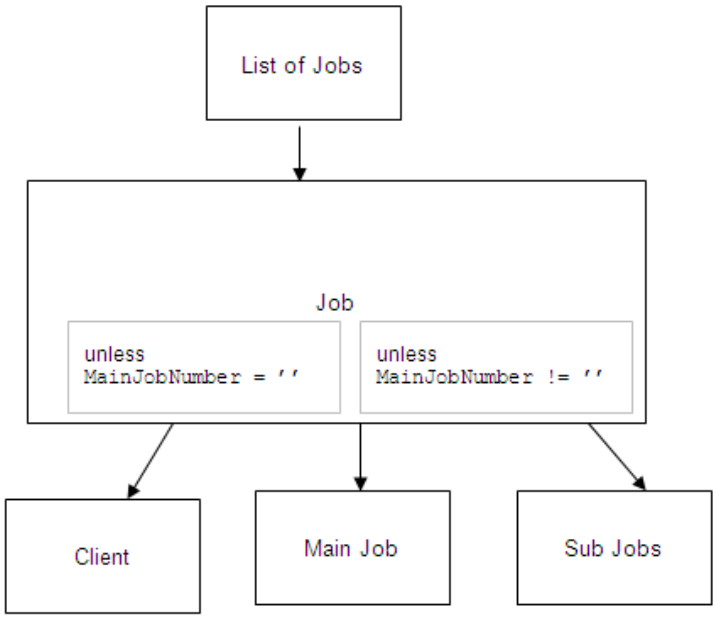
The workspace is designed as:

```
<Filter source="Jobs" title="List of Jobs">
  <Expansions>
    <With>
      <Card view="original" title="Job">
        <Bind foreignKey="CustomerNumber_Customer">
          <Card source="CustomerCard" title="Client"/>
        </Bind>
        <Bind foreignKey="MainJobNumber_JobHeader">
          <Card source="MainJobs" title="Main Job"
                hidden="parent.MainJobNumber = ''"/>
        </Bind>
        <Bind foreignKey="primary">
          <Table source="MainJobs" title="Sub Jobs"
                 hidden="parent.MainJobNumber != ''"/>
        </Bind>
      </Card>
    </With>
  </Expansions>
</Filter>
```

The `parent`-prefix indicates that the field is found in the immediate parent panel, compared to the one in which the hide-expression is defined. You can refer to the parent's parent by specifying `parent.parent` and so on.

The workspace looks and behaves the same as the workspace in Example 4, but loads less data and, consequently, performs better.

Here is the related tree:



Using the functions `hasNoRecords()` and `hasNoSeed()` leads to a higher data load. Avoid this by using the hide condition expression in the parent panes. If the hide expression refers to data in the pane itself, that pane again must be loaded.

Sometimes it is not possible to fully express the hide condition solely from the parent context. In such cases, it is still more efficient to express as much as possible from the parent context.

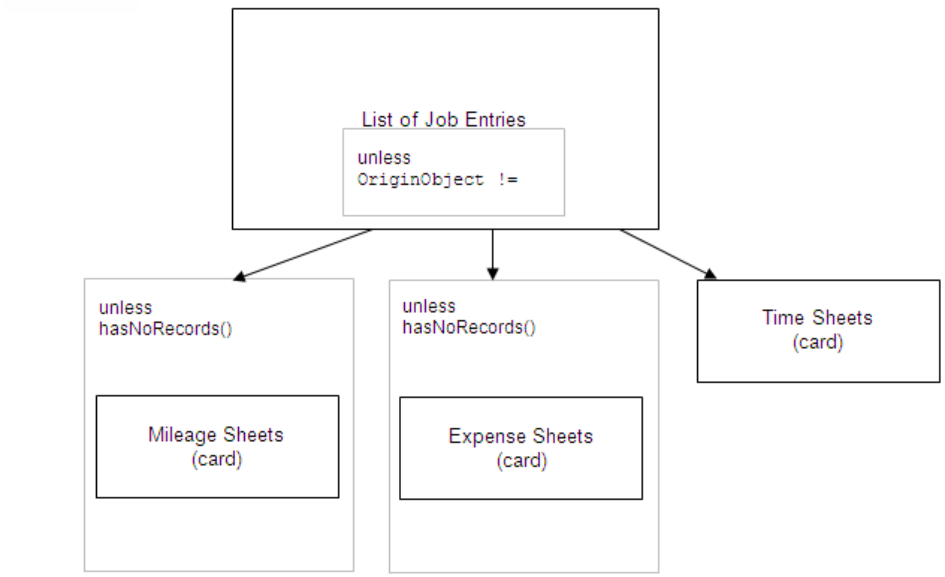
Example 6

You must construct a workspace that shows a list of job entries. Based on the origin of the job entry, you must show whatever has led to that entity, including the time sheet, the expense sheet, the mileage sheet, and so on.

In this case, the information on the job entry is not enough to distinguish between expense sheets and mileage sheets. However, attempting to load the mileage sheet dialog with the given transaction number leads to 0 (zero) records, if either the entry is not based on an expense/mileage sheet, or if the entry is based on a non-mileage expense sheet and vice versa for expense sheets.

Thus, you might consider designing the workspace as follows:

```
<Filter source="JobEntryOverview" title="List of Job Entries">
  <Bind forigenKey="OriginObject_ExpenseSheetHeader">
    <Card source="MileageSheets" title="Mileage Sheet"
          hidden="hasNoRecords()">
      <With>
        <Table/>
      </With>
    </Card>
    <Card source="ExpenseSheets" title="Expense Sheet"
          hidden="hasNoRecords()">
      <With>
        <Table/>
      </With>
    </Card>
  </Bind>
  <Bind foreignKey="OriginObject_TimeSheetHeader">
    <Card source="Time Sheets" title="Time Sheet"
          hidden="parent.OriginObject != RelationNameType'TimeSheetHeader">
      <With>
        <Table/>
      </With>
    </Card>
  </Bind>
</Filter>
```

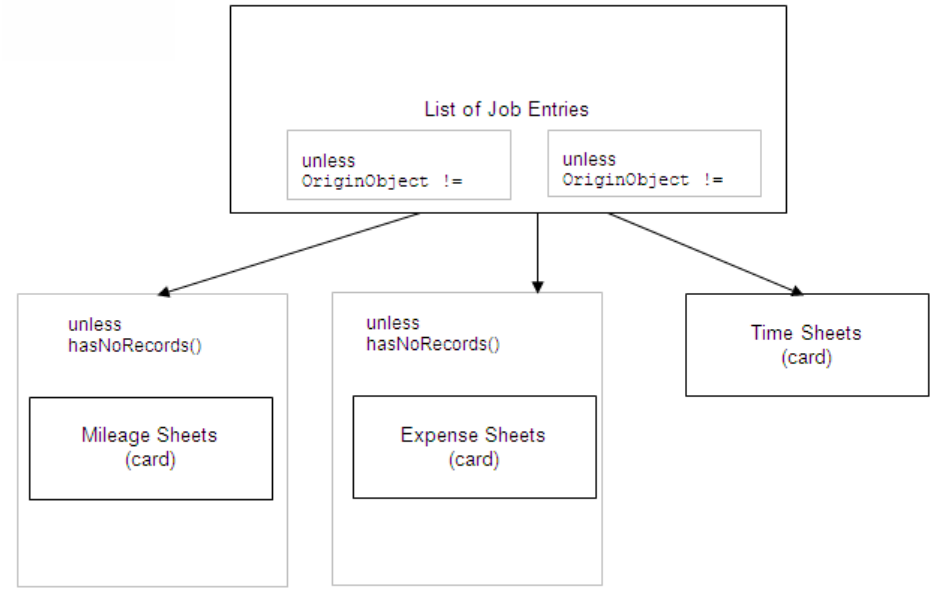


In this example, you only show related time sheets, expense sheets, and mileage sheets.

The visibility of the time sheet is controlled solely by the parent properties. However, this does not work for the expense/mileage sheet because the value of the field `parent.OriginObject` is the same for mileage and expense sheets.

However, in many cases, the selected job entry is not an expense or mileage sheet. In this case, you can avoid even loading data for these panels by elaborating the hide expressions, as follows:

```
<Filter source="JobEntryOverview" title="List of Job Entries">
  <Bind forigenKey="OriginObject_ExpenseSheetHeader">
    <Card source="MileageSheets" title="Mileage Sheet"
          hidden="parent.OriginObject != RelationNameType'ExpenseSheetHeader
                  or hasNoRecords()">
      <With>
        <Table/>
      </With>
    </Card>
    <Card source="ExpenseSheets" title="Expense Sheet"
          hidden="parent.OriginObject != RelationNameType'ExpenseSheetHeader
                  or hasNoRecords()">
      <With>
        <Table/>
      </With>
    </Card>
  </Bind>
  <Bind foreignKey="OriginObject_TimeSheetHeader">
    <Card source="Time Sheets" title="Time Sheet"
          hidden="parent.OriginObject != RelationNameType'TimeSheetHeader">
      <With>
        <Table/>
      </With>
    </Card>
  </Bind>
</Filter>
```



In this version of the workspace specification, it checks whether the selected entry is completely out of scope for mileage and expense sheets. If so, these two panels are hidden. If not, you must examine

whether the panels contain data or not. This implies that data is not loaded for these panels if the selected job entry refers to something other than an expense/mileage sheet.

Cross-Referencing Workspaces

Another powerful feature of workspaces is the ability to have cross-references. Cross-references are specified in the layouts of the panels.

A cross-reference means that one panel refers to data in another panel.

For example, a panel that shows detailed information about the job could show the e-mail address of the project manager. This information is not found as a field in the Jobs dialog, but it can be found on the Employee dialog. Using layout cross-references, you can show the value of the e-mail address field of the project manager of the job. In this case, the Job panel has a cross-reference to the Project Manager panel. This implies that when the Job panel is shown, it is necessary to load the data for the Project Manager panel as well.

Example 6

```
<Filter source="Jobs">
  <With>
    <Card>
      <Bind foreignKey="ProjectManagerNumber_Employee">
        <Hidden source="Employees"/>
      </Bind>
    </Card>
  </With>
</Filter>
```

This workspace shows a list of jobs and shows detailed information about the selected job. It also contains a statically `<Hidden>`-panel. Because there is no content that depends on the Hidden panel, it is not loaded.

However, consider the following layout for the Job card panel:

```
...
<Field source="ProjectManagerNameVar"/>
<Field source="ProjectManagerNumber_Employee.ElectronicMailAddress"/>
...
```

The layout displays the field ElectronicMailAddress from the panel that is referenced through the project manager foreign key binding.

This can be efficient, although you must consider that every time that data needs to be loaded for the Jobs panel, data must also be loaded for the Project Manager panel, because a field from that panel is referenced.

Another use of cross-references is in predefined filter restrictions. When specifying a filter layout, you can specify a static restriction that is always applied to the filter, and you can specify a selection of predefined restrictions that a user can choose from.

Such restrictions may depend on data in other panels. However, it is only supported to depend on data in parent panels. From a data-load standpoint, it is generally acceptable to add such restrictions, because it is not necessary to load additional panels because the parent panels are always needed anyway.

However, it may hinder performance when you reference information in filter panels. Filter panels are different from card and table panels in that not all potentially available fields are retrieved when data is obtained; only the fields that are strictly needed are retrieved. The advantage of this approach is that it increases the underlying database performance. Typically, retrieving data for all potentially available fields is considerably slower than retrieving only a portion of the data, such as 10-20 fields. In addition, if some of the fields that are retrieved require database joins, performance may be further impacted.

Example 7

You want to create a workspace that shows a list of jobs. For each job for which open (non-invoiced) hours are entered, you want to show a panel for selecting invoices. If there are no open hours, this panel should be hidden.

In this case, you can create the workspace as:

```
<Filter source="Jobs" title="List of Jobs">
  <Bind foreignKey="primary">
    <Card source="InvoiceSelection" title="Invoice Selection"
          hidden="parent.HoursOpen = 0">
      <With>
        <Table/>
      </With>
    </Card>
  </Bind>
</Filter>
```

This seems to be an appropriate setup. You only have the panels that are potentially needed, and the hidden constraint of the Invoice Selection panel is guarded by information in the parent pane only.

However, in this case it turns out that the calculation of the HoursOpen field is inefficient because it is based on information that is based on a join with an aggregated view. The workspace automatically discovers that it needs to retrieve the data for this field in the filter panel, but it is retrieved for all rows.

Depending on the data in your database and the number of rows in your filter, this may lead to degraded performance.

In this case, the following implementation might work better:

```
<Filter source="Jobs" title="List of Jobs">
  <Bind foreignKey="primary">
    <Hidden source="JobOverview">
      <Bind foreignKey="primary">
        <Card source="InvoiceSelection" title="Invoice Selection"
              hidden="parent.TimeNumberOpenVar = 0">
          <With>
            <Table/>
          </With>
        </Card>
      </Bind>
    </Hidden>
  </Bind>
</Filter>
```

This workspace has the additional cost of loading data for one more panel, although *only for the selected job*. In contrast, the initial version did not do this, but *for each line in the filter*, a corresponding value was calculated based on a join with aggregated data.

Thus, in some cases, it is worth adding a hidden panel instead of expanding the number of columns that are retrieved in the filter. There are no hard rules, but suggested parameters. Experiment to find which solution works better for you.

Cross-references can also be used to refer to actions in other panels. For example, from a filter panel you can refer to the create action of a corresponding card panel. In other cases, you can expose actions from a hidden card in the corresponding table panel.

As with fields, it is preferable for performance reasons to refer to actions of parent panels. You can refer to child panels, too, but data must then be loaded for these child panels. The data is loaded for the corresponding panels to determine the correct enabled state of the actions.

Example 8

You want to create a workspace that shows a filterable list of time sheets. From the filter, you need to create a time sheet (using a wizard), and to submit and approve the selected time sheet. In addition, you want to be able to review the selected time sheet.

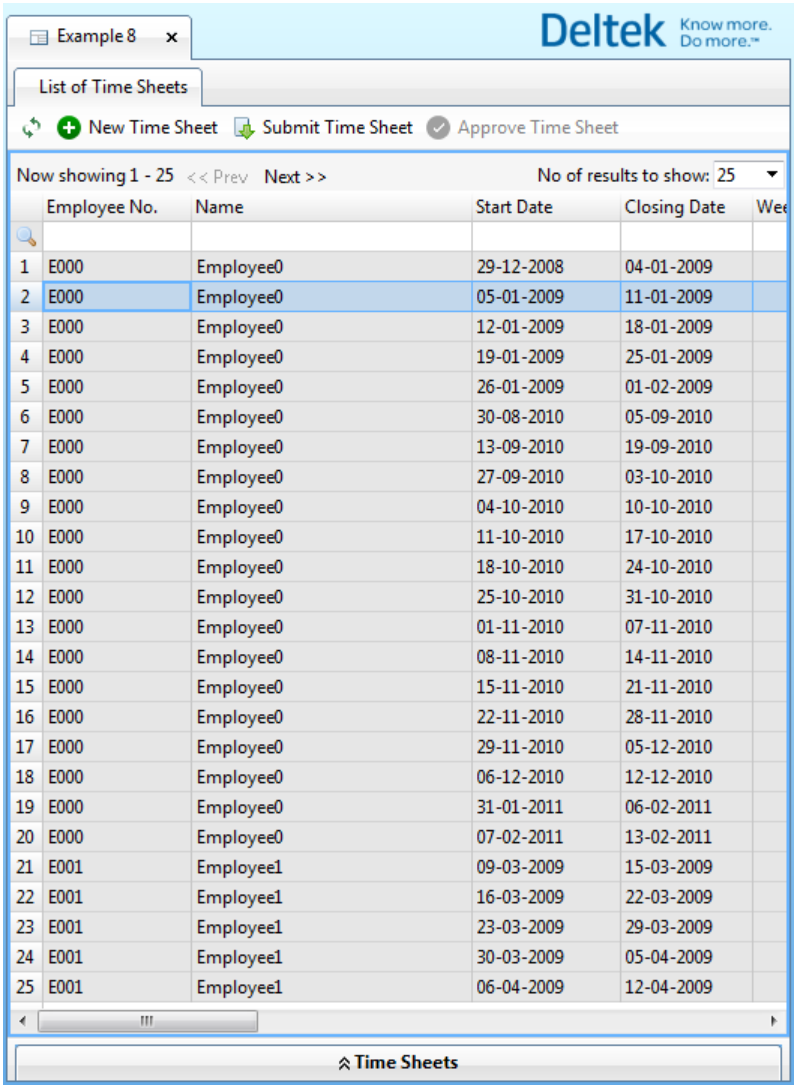
You could implement that workspace in the following way:

```
<Filter source="TimeSheets" title="List of Time Sheets"
        layout="mytimesheetlayout">
  <With>
    <Card>
      <With>
        <Table/>
      </With>
    </Card>
  </With>
</Filter>
```

The filter layout in the file `mytimesheetslayout.mdml.xml` is displayed as follows:

```
<FilterPane>
  <Filter>
    <Actions>
      <Action source="with.create" wizard="createWizard"/>
      <Action source="with.SubmitTimeSheet" />
      <Action source="with.ApproveTimeSheet" />
    </Actions>
    ...
</FilterPane>
```

The related screen is displayed as follows.



	Employee No.	Name	Start Date	Closing Date	Wee
1	E000	Employee0	29-12-2008	04-01-2009	
2	E000	Employee0	05-01-2009	11-01-2009	
3	E000	Employee0	12-01-2009	18-01-2009	
4	E000	Employee0	19-01-2009	25-01-2009	
5	E000	Employee0	26-01-2009	01-02-2009	
6	E000	Employee0	30-08-2010	05-09-2010	
7	E000	Employee0	13-09-2010	19-09-2010	
8	E000	Employee0	27-09-2010	03-10-2010	
9	E000	Employee0	04-10-2010	10-10-2010	
10	E000	Employee0	11-10-2010	17-10-2010	
11	E000	Employee0	18-10-2010	24-10-2010	
12	E000	Employee0	25-10-2010	31-10-2010	
13	E000	Employee0	01-11-2010	07-11-2010	
14	E000	Employee0	08-11-2010	14-11-2010	
15	E000	Employee0	15-11-2010	21-11-2010	
16	E000	Employee0	22-11-2010	28-11-2010	
17	E000	Employee0	29-11-2010	05-12-2010	
18	E000	Employee0	06-12-2010	12-12-2010	
19	E000	Employee0	31-01-2011	06-02-2011	
20	E000	Employee0	07-02-2011	13-02-2011	
21	E001	Employee1	09-03-2009	15-03-2009	
22	E001	Employee1	16-03-2009	22-03-2009	
23	E001	Employee1	23-03-2009	29-03-2009	
24	E001	Employee1	30-03-2009	05-04-2009	
25	E001	Employee1	06-04-2009	12-04-2009	

The buttons are enabled as follows:

- The **New Time Sheet** button is always enabled.
- The **Submit Time Sheet** button is enabled if the selected time sheet has not already been submitted.
- The **Approve Time Sheet** button is enabled if the selected time sheet has been submitted and not approved.

To determine the enabledness state of the action as specified by the time sheets dialog, the time sheet card panel must be loaded every time that a line is selected. Consequently, navigating up and down in the filter is slow (especially because the client delays the server-roundtrip for a while if the user navigates to another time sheet), because a server-roundtrip must be done for each new line.

Improving Performance with Cross-Reference Actions

You can improve the performance of workspaces that use cross-reference actions by taking over the control of the enabledness state. In this case, because the enabledness state no longer depends on data in the panel, the panel is not loaded until it is shown, or until the user activates the corresponding action.

Example 9

To improve the workspace of Example 8 you determine that what slows this workspace down is the cross-referencing actions. By taking control of the enabledness state of these actions, the workspace does not need to load data for that panel. Thus, navigating the table is much faster (provided that the time sheet panel is minimized) because no server roundtrip is needed when navigating.

First, consider whether actions are enabled or disabled:

- The New Time Sheet action should never be disabled.
- The Submit Time Sheet action should be disabled if the selected time sheet has not been submitted, or if there is no selected time sheet.
- The Approve Time Sheet action should be disabled if the selected time sheet has either not been submitted or already approved. Also, if there is no selected time sheet, the action should be disabled.

When no record is selected, it means that expressions that are based on the data of the selected record are ill-defined, and therefore such expressions cannot be evaluated. In this case, they default to false. Unless you change the design, the disabledness expressions default to false, leaving the actions enabled.

Using the `defaultTo` construct in the expressions makes the workspace perform better by changing the cross-referencing layout, as follows:

```
<FilterPane>
  <Filter>
    <Actions>
      <Action source="with.create" wizard="createWizard"
              disabled="false"/>
      <Action source="with.SubmitTimeSheet"
              disabled="submitted defaultTo true"/>
      <Action source="with.ApproveTimeSheet"
              disabled="not submitted or approved defaultTo true"/>
    </Actions>
    ...
</FilterPane>
```

With this change, the workspace looks and behaves exactly like that in Example 8, only it performs better because the data does not have to be loaded into the card panel every time that you change the line in the filter panel. This is loaded only when needed.

Identifying Top Considerations

This section explains how to optimization performance, including enabling an explanatory log, and a table that details why specific data is loaded, and how you can optimize the workspace, if applicable.

Enabling an Explanatory Log

To begin, enable an explanatory log, which, for each data request, logs an explanation of which workspace panes are considered for data loading, and why.

To enable logging, complete the following steps:

1. Edit the log description file that is associated with your client (typically `<Client-directory>/configuration/logback.xml`). In this file, add the following:

```
<configuration>
...
  <logger
name="com.maconomy.client.workspace.model.local.model.McWorkspaceRequestTreeResolver”>
    <level value="DEBUG"/>
  </logger>
</configuration>
```

2. Open the client and then the workspace.
3. After the workspace is loaded, refresh data in the topmost panel in the workspace.
4. The log output now contains an explanation of which parts of the workspace require data when refreshing the top-most panel.
5. Select the various parts/tabs in the workspace to evaluate data in the explanation log.

Example 10

This reexamines the workspace from Example 1.

When this workspace is just opened, only the filter panel is visible. The log output is displayed as follows:

```
REQUEST OF TYPE: REFRESH

The LIKELY REQUEST ROOT is Pane <workspace root> because there are no mounted
non-filter parent panes above the requesting pane even though the request
does not have side-effects

Pane <workspace root> is included because it INITIATES THE REQUEST

*Pane Jobs ('List of Jobs') (FILTER) is included because it is VISIBLE
Pane <workspace root> is included because it's ON THE ROOT PATH from *Pane
Jobs ('List of Jobs') (FILTER)

The ROOT of the request is: Pane <workspace root>

----------------------------
```

`REQUEST OF TYPE` — Disregard this line. This line says that the data request is *Refresh* type, meaning that data is refreshed. This document does not detail this value.

`LIKELY REQUEST ROOT` — Disregard this section. This indicates the likely request root, and shows which panel (pane) is considered the top-most panel to communicate to the server. This is often—but not always—the workspace root. This document does not detail these lines.

`INITIATES THE REQUEST` — Begin evaluation at this portion. The following is a list of panels that are included in the request to the server, with an explanation of why. An asterisk (*) indicates that the corresponding panel carries data. Other panels (for example, sections and the root) do not carry any data and have no performance impact whatsoever.

In this example, one data-carrying panel is included because it is visible and therefore needs data. Data is only loaded for the filter. No data is loaded for the Job and Client panels.

If a user double-clicks an entry in the filter, this makes the filter appear compacted, and reveals the remaining portion of the workspace. By default, the Job panel is selected and is therefore visible. The resulting output is:

```
REQUEST OF TYPE: REFRESH

The LIKELY REQUEST ROOT is Pane <workspace root> because there are no mounted
non-filter parent panes above the requesting pane even though the request
does not have side-effects

*Pane Jobs#2 ('Job') (CARD) is included because it INITIATES THE REQUEST

*Pane Jobs ('List of Jobs') (FILTER) is included because it is VISIBLE

*Pane JobTasks ('Tasks') (TABLE) is included because it is VISIBLE

Pane <workspace root> is included because it's ON THE ROOT PATH from *Pane
Jobs ('List of Jobs') (FILTER)

The ROOT of the request is: Pane <workspace root>

----------------------------
```

In this case, you can see that three data-carrying panels are retrieved: the filter, the Job panel, and the Tasks panel—which is a child of the Job panel and by default visible.

The Job panel gets data because it initiates the request, meaning that either the user executed some action (such as update, delete, or close) in this panel or, as in this case, because the client automatically determines to refresh this panel. It is refreshed because it is revealed in a case where it does not have any data (or more specifically, it does not have updated data).

In addition, the Tasks panel has data because it is now also visible.

Similarly, the List of Jobs (filter) panel must have data because it is visible.

You may wonder why the explanation of the Job panel is that it *initiates the request* rather than that *it is visible*. This is because for each panel, only one reason is specified.

In the preceding example, in the second round of data retrieving the filter panel is included even though it apparently already contains data. This is because all panes between a panel generally must have updated data as well as the root of the request must be considered to determine exactly which data must be loaded. **I**t is *not necessarily the same as actually reloading data for these panels*. Analysis occurs during data retrieval.

This analysis may imply that data is not reloaded to a panel. It is beyond the scope of this document to go into details about this. However, remember that a full reload of all of the mentioned panels does not necessarily take place.

The coupling service (which is responsible for retrieving data) always behaves optimally. Still, the number of panels that are written in the log reflects the potential total amount of data that must be (re)loaded and why.

Data Loading Explanation Table

The following table examines why data loading is considered.

Logged Explanation	Data Loads Because…
`<panel> is included because it INITIATES THE REQUEST`	This panel triggers the data retrieval. The data retrieval may be triggered by user action, such as a delete or update, or a refresh that occurs when part of the workspace is being revealed, and that part does not have fully updated data.
`<panel> is included because it is VISIBLE`	This occurs because any panel that is currently visible is considered for reloading when data changes occur. Thus, if you open many panels simultaneously in a workspace, more data must be loaded.
`<panel> is included because it's title is (potentially) visible and its appearance is DEPENDTANT OF IT'S OWN DATA (and/or data of children)`	This occurs due to a hide expression that is dependent on data in itself (for example, `hasNoRecords () )` or if the hide expression references values of fields in the panel itself (for example, `not Approved`). It may also be loaded if a panel applies the function `hasNoChildren()`, or if visible child panels depend on data in those child panels. **To optimize the workspace**, change the hide expression to instead depend on the parent data.
`<panel> is included because the title of <other panel> is potentially visible DEPENDING ON DATA of this branch`	This occurs when the hide expression of a panel depends on the data in a hidden panel. Thus, the hide expression of the `<other panel>` is dependent on data in another parent panel (of `<other panel>`), but that parent panel is a `<Hidden>`-panel. **To optimize this workspace**, let the hide expression of the `<other panel>` depend on the parent data of that `<Hidden>` -panel.

Logged Explanation	Data Loads Because…
`<panel> is included because its DYNAMIC FOREIGN KEY uses same switch field as that of <other panel>`	This occurs because data is always retrieved to siblings of a panel that is bound by a dynamic foreign key, when those siblings are also bound by a dynamic foreign key that is switched by the same switch field. This is automatically selected to ensure that the corresponding panel is included when one dynamic foreign key suddenly becomes out of scope. For example, when the current panel becomes out of scope, it does not contain any data. But if one of the corresponding siblings does, this sibling should take its place.  A *dynamic foreign key* is a foreign key that has a validity that is conditioned by the value of a field (the *switch field*). If that switch field has a specific value, this foreign key is considered relevant. Dynamic foreign keys that have a common switch field are mutually exclusive. If there are many such dynamic siblings, the list of panels may be substantial. However, this does not compromise performance because only *one* of these is active at any given time. The inactive panels have no data, and this is efficiently handled by the coupling service.
`<panel> is included because IT'S PARENT'S VISUAL STATE DEPENDS ON THIS CHILD.`	This occurs when a panel has a hide expression with `hasNoChildren()`. Data is retrieved for those children where the visibility is dependent on data in themselves (or their children), and where it cannot be statically determined how many children are visible in a given case. **To optimize your workspace**, avoid the use of `hasNoChildren()`.
`<panel> is included because IT IS REFERRED BY <other panel>`	This occurs when a field or an action is borrowed from another panel. If the panel that borrows a field/action is visible, data is retrieved for the `<other panel>`. **To optimize your workspace when actions are borrowed**, express the enabledness of the actions based on data in `<panel>` or one of its parents. **To optimize your workspace when fields are borrowed**, obtain the borrowed value from the panel context, potentially expressed as an expression. In this case, use the `<Description>`-element in the layout.
`<panel> is included because it's ON THE ROOT PATH from <other panel>`	This occurs because any panel that needs data needs context from parent panels to determine which data to load into a given panel.

Example 11

Take another look at the workspace from Example 4. Here you have built a workspace with two panels that may be dynamically hidden, depending on whether a selected job is a sub-job or a main job:

The workspace is:

```
<Filter source="Jobs" title="List of Jobs">
  <Expansions>
    <With>
      <Card view="original" title="Job">
        <Bind foreignKey="CustomerNumber_Customer">
          <Card source="CustomerCard" title="Client"/>
        </Bind>
        <Bind foreignKey="MainJobNumber_JobHeader">
          <Card source="MainJobs" title="Main Job" hidden="hasNoSeed()"/>
        </Bind>
        <Bind foreignKey="primary">
          <Table source="MainJobs" title="Sub Jobs"
                 hidden="hasNoSeed()"/>
        </Bind>
      </Card>
    </With>
  </Expansions>
</Filter>
```

If the workspace is opened and a user double-clicks a job in the filter, the log output is displayed as follows:

```
REQUEST OF TYPE: REFRESH

The LIKELY REQUEST ROOT is Pane <workspace root> because there are no mounted
non-filter parent panes above the requesting pane even though the request
does not have side-effects

*Pane Jobs#2 ('Job') (CARD) is included because it INITIATES THE REQUEST

*Pane Jobs ('List of Jobs') (FILTER) is included because it is VISIBLE

*Pane CustomerCard ('Client') (CARD) is included because it is VISIBLE

*Pane MainJobs ('Main Job') (CARD) is included because its title is
(potentially) visible and its appearance is DEPENDTANT OF IT'S OWN DATA
(and/or data of children)
*Pane MainJobs#2 ('Sub Jobs') (TABLE) is included because its title is
(potentially) visible and its appearance is DEPENDTANT OF IT'S OWN DATA
(and/or data of children)

Pane <workspace root> is included because it's ON THE ROOT PATH from *Pane
Jobs ('List of Jobs') (FILTER)

The ROOT of the request is: Pane <workspace root>

---------------------------
```

Closely review the two boldfaced lines to find an opportunity to optimize your workspace. Notice that the panes are loaded not because the data is going to be shown to the user, but because it is necessary to determine whether showing the corresponding tab makes sense or not.

Consider whether you can determine whether it makes sense to show the Main Job tab without loading data for it, as well as whether you can determine whether it makes sense to show the Sub Jobs tab without loading data for it.

To optimize this workspace:

- Hide this panel when there is no main job for the job in question.

  It only makes sense to show the Main Job tab for jobs that have a main job. This information is found in the **MainJobNumber** field on the job. The job is found in the Job panel that is the immediate parent of the Main Job tab.

  Include the hide expression as follows:

  ```
  parent.MainJobNumber = ''
  ```

- Hide this panel when there is already a main job that is associated with the job in question.

  It only makes sense to show the Sub Jobs tab for jobs that either have or could have sub-jobs. A job that is itself a sub-job cannot have sub-jobs, while other jobs can.

  Include the hide expressions as follows:

  ```
  parent.MainJobNumber != ''
  ```

Change the workspace specification into the one listed in Example 5:

```
<Filter source="Jobs" title="List of Jobs">
  <Expansions>
    <With>
      <Card view="original" title="Job">
        <Bind foreignKey="CustomerNumber_Customer">
          <Card source="CustomerCard" title="Client"/>
        </Bind>
        <Bind foreignKey="MainJobNumber_JobHeader">
          <Card source="MainJobs" title="Main Job"
                hidden="parent.MainJobNumber = ''"/>
        </Bind>
        <Bind foreignKey="primary">
          <Table source="MainJobs" title="Sub Jobs"
                 hidden="parent.MainJobNumber != ''"/>
        </Bind>
      </Card>
    </With>
  </Expansions>
</Filter>
```

This results in a different log output when selecting a job in the filter by double-clicking. The following output is displayed:

```
REQUEST OF TYPE: REFRESH

The LIKELY REQUEST ROOT is Pane <workspace root> because there are no mounted
non-filter parent panes above the requesting pane even though the request
does not have side-effects

*Pane Jobs#2 ('Job') (CARD) is included because it INITIATES THE REQUEST

*Pane Jobs ('List of Jobs') (FILTER) is included because it is VISIBLE

*Pane CustomerCard ('Client') (CARD) is included because it is VISIBLE

Pane <workspace root> is included because it's ON THE ROOT PATH from *Pane
Jobs ('List of Jobs')

The ROOT of the request is: Pane <workspace root>

---------------------------
```

The two boldfaced lines are now gone, and data is no longer loaded for those panels. Since the behavior of the two workspace definitions is identical, you have created a more performance-efficient version of the workspace.

Tips and Tricks

This document has described which workspace constructs may result in excessive data loading, and how to alter specifications to create more efficient workspaces. While there are no set rules for specifications, follow the best practice guidelines and tips and tricks that are provided in the following sections to achieve better workspace performance.

Avoid Borrowing Fields

Try to avoid borrowing fields from other panels. If you must, consider whether the information that you want can be included from parent panels or can be found by applying values of the panel itself to a function value. If this is not possible, consider whether you can obtain the values from a panel that will typically need data for other reasons.

Take Care Borrowing Actions

If you borrow actions from other panels, consider whether you can control the enabledness state of every borrowed action from the layout if at all possible. Notice that if you can only control the enabledness state of two out of three actions borrowed from the same panel, there will be no benefit.

Borrow from Card Dialogs

When you borrow fields or actions from other panels, try to borrow from card dialogs rather than card/table dialogs. Card dialogs typically perform better (on the server) than card/table dialogs, even if only the card part is needed.

Borrow Few Panels

If you borrow fields and/or actions from other panels, attempt to borrow from as few panels as possible (not counting parent panels).

Borrow from Nearest Non-`<Hidden>` Panel

When you refer fields or actions from parent panels, try to borrow from the nearest non-`<Hidden>` panel possible. The reason for choosing the nearest is not important from a performance perspective, but you may benefit from this because it makes your cross-references slightly easier to manage. Performance-wise, it makes sense to avoid `<Hidden>`-panels, because this may allow the client to retrieve less data in some cases, thereby improving performance.

Avoid Certain Hide Expressions Functions

In hide expressions, avoid the use of the functions `hasNoSeed()`, `hasNoRecords()`, and `hasNoChildren()`. Likewise, avoid the use of references to data in the panel itself.

If you cannot avoid the use of these functions, attempt to rule out as many situations as possible based on the value of parent data.

For example, rather than using:

```
hasNoRecords()
```

Instead, use the following:

```
parent.Field != SomeValue or hasNoRecords()
```

This is more efficient even if you know that the panel will never contain any record when the first part of the expression is fulfilled.

About Deltek

Better software means better projects. Deltek is the leading global provider of enterprise software and information solutions for project-based businesses. More than 23,000 organizations and millions of users in over 80 countries around the world rely on Deltek for superior levels of project intelligence, management and collaboration. Our industry-focused expertise powers project success by helping firms achieve performance that maximizes productivity and revenue. www.deltek.com