The Deltek Maconomy Extension Framework

Programmer's Guide

Edited by

Jakob Lyng Petersen

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 November 2016.

Contents

Chapter 1

Introduction

The Extension Framework for Deltek Maconomy is a programming framework that enables programmers to extend the functionality of the Deltek Maconomy ERP software. Extensions made with this framework are *safe* in that it is not possible to compromise the data integrity of the software.

This book describes the concepts of the Extension Framework, what kind of extensions can be made and how to program such extensions. The expected audience of this book is software developers with a thorough understanding of the Deltek Maconomy ERP system and experience with Java.

1.1 The Scope of Extensions

The Extension Framework is meant to extend the functionality of the Deltek Maconomy ERP system version 2.0 service pack 4 or higher. In version 2.0, a new server-side component, the *Coupling Service* was introduced. This component is a central component for users of the Maconomy *Workspace Client*.

Extensions to the workspace client have many facets. Some are "simple" ranging from changing the content and presentation of the menu, changing or defining new workspaces, changing the layout, to altering to do's. Making such extensions are done declaratively by modifying or adding special-purpose XML files using the Maconomy Extender IDE. Extensions of this kind are documented elsewhere [QMm, QMw, QMd, QMn, QMc, CMd] and are beyond the scope of this book.

Other kinds of extensions are closer related to extending the *logic* of the software application. Such extensions are written as so-called *plug-in's* to the Coupling Service and must be programmed in Java™ [GJSB05]. The scope of this book is such Java-based extensions. The functionality offered thereby are available to the Workspace Client. They

will not be available to users of the "Jaconomy" client, the Maconomy Portal or mobile applications such as "Touch Time"[1].

1.2 General Coding Conventions

The Extension Framework follows a number of standards and conventions. It is important that you know of them. This section will take you through these. Also, this section will highlight certain coding conventions that you are highly encouraged to follow in your own code.

1.2.1 General Naming Convention for Framework Types

Generally, all types provided and used by the framework are prefixed with `Mi` (for interfaces), `Mc` (for regular classes) and `Me` (for `enum`-classes.)

In fact, this naming convention is not a generally recommended naming convention for Java types. It is, however, practical for extension developers to know that types provided by the framework follow this naming convention. So, if a type is named like this, it is provided by the Extension Framework. If you need to search for a type provided by the framework, you can easily differentiate it from types offered by other libraries. Apart from this little oddity, the remaining part of the type names follow widely accepted naming patterns for Java: starting with an upper-case letter, using camel-casing to separate words (not underscores '_'), and only upper-case acronyms less than three characters. Examples:

- `MiKey`: the actual type name "key" starts with an upper case character. *Not* ~~`MiKEY`~~, ~~`Mikey`~~ or ~~`MiKeY`~~.
- `McStringDataValue`: different words are separated by a change in casing. *Not* ~~`McstringDataValue`~~ or ~~`McString_Data_Value`~~.
- `MiDatabaseApi`: API is an acronym, but it is not upper-cased because it has three characters. *Not* ~~`MiDatabaseapi`~~ or ~~`MiDatabaseAPI`~~.

We do not *recommend that you follow the same naming convention for your classes and interfaces.* Instead, we encourage you to choose a widely accepted naming convention such as the naming convention used by Extension Framework-types (except the `Mc`, `Mi` or `Me` prefix.) Also, it is a common naming convention to prefix all interface types with a capital 'I' (like `IJobCostDimensions` which could be a name for an interface allowing access to Maconomy Job Cost dimensions.)

[1] In the future, extensions may affect other end-user interfaces as well.

1.2.2 `null` is *Never* Used

The general-purpose Java constant `null` is *never* used within the Extension Framework. Period. The reason for this is that `null` is a valid value for any Java class- or interface-type. This means that it is very easy to inadvertently make a programming error, because the compiler cannot check for it.

For example, suppose that some method `f(SomeClass argument)` is invoked by `f(null)`. The Java compiler will accept this, because it is valid Java code. However, if the implementer of the method '`f`' attempts to invoke any method on the argument, a run-time error will occur. Even if the implementer of the method '`f`' attempts to handle the possibility of receiving `null`, that method may invoke other methods directly or indirectly, and some of them may fail on receiving `null`. Experience shows that such run-time errors[2] are very hard to avoid, and often such errors are hard to debug because the error might not occur where the error was really made.

Obviously, `null` was invented for a reason. Sometimes it is practical to have the ability to signal that "you don't care" or that you have no sensible value to provide. Instead of using `null`, the Extension Framework uses a generic type called `MiOpt<T>` indicating an optional value of value type `T`. This type provides only a few methods:

Method	Remarks
`isDefined`	This method returns `true` if an actual value is available through the `get` method.
`isNone`	This method returns `true` if there is no value associated with this object. Using such a value corresponds exactly to using `null`, except that you cannot invoke methods of the `T`-type without getting compile-time errors.
`get`	This method returns the value of type `T` contained by this object. This will only succeed if `isDefined()` yields `true`. But the programmer has the chance to test for this before doing so. If the `get()` method is invoked when there is no value this will not be discovered at compile-time. However, a run-time error will be thrown *immediately* where the programming error is, leading to much easier debugging.

We strongly encourage you to follow this way of coding even "internally" in your own code. We expect *that you do it when working with the Extension Framework.* Hence, documentation will never explicitly state that `null` is not allowed, because it never is. When methods take arguments of type `MiOpt<...>`, this means that the argument may be a value that is not defined. Similarly, methods *never* state whether the returned

[2] Known as `NullPointerExceptions`

value may be `null`, because it never is. If the declared return type is `MiOpt<...>`, it means that the returned value may or may not be defined. Naturally, 3rd-party libraries may make use of `null`, and when using such libraries, `null` is a possibility you will have to consider. In such cases, we encourage you to convert the potential `null`-values into `MiOpt` just before/after invoking such libraries.

The Extension Framework provides methods for creating values of type `MiOpt` by using the utility class `McOpt`. In this way, `McOpt.none()` creates a "none" value, and `McOpt.opt(obj)`[3] converts `obj` into a "defined" value, letting `get()` return `obj`.

1.2.3 The Java `List`, `Set` and `Map` are never used directly

Java has interfaces for lists (`List`), sets (`Set`) and maps (`Map`.) With the use of generics introduced in Java 5, it has become much more type-safe to work with these interfaces and related classes, compared to Java 1.4. For example, the compiler will yield an error if you try to store an "apple" in a list containing "postal codes." However, there are still some operations that are not adequately type-safe. For example, with Java sets, it is allowed to ask "is the apple, `myApple`, found in this set of postal-codes?" While you may argue that this makes perfectly sense mathematically (the answer will always be "no" since an apple is not a postal code), chances are that the programmer made a mistake if the program ever needs to ask this question.

In order to remedy this, the Extension Framework uses and offers similar interface-types which *are* adequately type safe. With these types, you will get a compile-time error, if your program asks whether an apple is found in a list of postal codes. The types offered by the Extension Framework are otherwise similar to the standard Java types. In fact, they are instances of these types, so you can pass them to 3rd-party libraries which does not know of the Extension Framework types. The types are called `MiList`, `MiSet` and `MiMap`. And just like the peer standard Java interfaces, they use generics, so you can declare the type of objects contained by a specific instance of these types. Since these types are also working as the standard Java equivalents, they have all the same methods. So, technically, you *can* ask whether a specific apple is found in a list of postal codes. However, that method is marked as "deprecated" which will make the Maconomy Extender strike out the code and the compiler will make a warning, making it immediately clear that something is wrong. Instead, the framework types introduces methods with the same name, except that the name has the suffix `TS`[4]. This means that

- Use `containsTS(...)` instead of ~~`contains(...)`~~
- Use `removeTS(...)` instead of ~~`remove(...)`~~

[3] This can be shortened even further by using "static imports" introduced in Java 5, which will allow you to simply write: `opt(obj)`

[4] For "*T*ype*S*afe.

- Use `containsAllTS(...)` instead of ~~`containsAll(...)`~~
- Use `removeAllTS(...)` instead of ~~`removeAll(...)`~~
- Use `retainAllTS(...)` instead of ~~`retainAll(...)`~~
- Use `indexOfTS(...)` instead of ~~`indexOf(...)`~~
- Use `lastIndexOfTS(...)` instead of ~~`lastIndexOf(...)`~~
- Use `subListTS(...)` instead of ~~`subList(...)`~~
- Use `equalsTS(...)` instead of ~~`equals(...)`~~
- Use `containsKeyTS(...)` instead of ~~`containsKey(...)`~~
- Use `containsValueTS(...)` instead of ~~`containsValue(...)`~~
- Use `getTS(...)` instead of ~~`get(...)`~~
- Use `putTS(...)` instead of ~~`put(...)`~~; although the standard `put()` method only accepts arguments of the right type, it may return `null`. The type-safe variant converts the result into an `MiOpt`.
- Use `entrySetTS()` instead of ~~`entrySet()`~~
- Use `valuesTS()` instead of ~~`values()`~~
- Use `keySetTS()` instead of ~~`keySet()`~~
- Use `depositTS(...)` instead of ~~`deposit(...)`~~

In addition to providing type-safe variants of the corresponding standard methods, these interfaces offer a few convenience methods, e.g., `getOptTS(...)` and `getElseTS(...)`.

By using the utility class `McTypeSafe` you can easily construct sets, maps and lists of the types `MiSet`, `MiMap` and `MiList`. Using this class offers the following factory methods:

Method	Remarks
`arrayList`	Returns an initially empty `MiList` which is based on an array-list implementation.
`collection`	Returns an empty `MiCollection`.
`convertCollection`	Returns a `MiCollection` which is based on a given input `Collection`, i.e., transforming the input-collection to be of type `MiCollection`. In this way, you can turn any Java-collection into a `MiCollection`.
`convertDeque`	Returns a `MiDeque` from a specified `Deque` object.
`convertIterable`	Returns a `MiCollection` based on the contents of the specified `Iterable`.

Method	Remarks
`convertList`	Returns a `MiList` from a specified `List`. The returned list is based on the implementation of the provided list. It merely wraps that `List` into being a `MiList`.
`convertMap`	Returns a `MiMap` from a specified `Map`. The returned map is based on the implementation of the provided map. It merely wraps that `Map` into being a `MiMap`.
`convertSet`	Returns a `MiSet` from a specified `Set`. The returned set is based on the implementation of the provided set. It merely wraps that `Set` into being a `MiSet`.
`convertSortedMap`	Returns a `MiSortedMap` from a specified `SortedMap`. The returned sorted map is based on the implementation of the provided sorted map. It merely wraps that `SortedMap` into being a `MiSortedMap`.
`convertSortedSet`	Returns a `MiSortedSet` from a specified `SortedSet`. The returned sorted set is based on the implementation of the provided sorted set. It merely wraps that `SortedSet` into being a `MiSortedSet`.
`createArrayDeque`	Returns an initially empty `MiDeque` based on an `ArrayDeque`.
`createArrayList`	Returns an initially empty `MiList` based on an `ArrayList`. An optional argument allows you to specify the initial capacity of the list. Other variants of this method allows you to prepopulate the list with values from any `Iterable` or by specifying a comma-separated list of values.
`createCollection`	Returns a `MiCollection` optionally containing specified values.
`createConcurrentHashMap`	Returns a `MiMap` based on a concurrent `HashMap`.
`createEnumMap`	Returns a `MiMap` based on an `EnumMap`.
`createEnumSet`	Returns a `MiSet` based on an `EnumSet`.
`createHashMap`	Returns a `MiMap` based on a `HashMap`. The map may initially be empty or based on some provided key/value elements.
`createHashSet`	Returns a `MiSet` based on a `HashSet`. The set may be empty or initially populated with specified values.
`createLinkedDeque`	Returns a `MiDeque` based on a `LinkedDeque` implementation.
`createLinkedHashMap`	Returns a `MiMap` based on a `LinkedHashMap` implementation.
`createLinkedHashSet`	Returns a `MiSet` based on a `LinkedHashSet` implementation.

Method	Remarks
`createLinkedList`	Returns a `MiList` based on a `LinkedList` implementation.
`createSingletonList`	Returns a singleton `MiList`.
`createStack`	Returns a `MiStack` which is a wrapping of a `Stack` implementation.
`createTreeMap`	Returns a `MiMap` based on a `TreeMap` implementation.
`createTreeSet`	Returns a `MiSet` based on a `TreeSet` optionally with a specified comparator.
`emptyList`	Returns an empty `MiList`. The list cannot be modified.
`emptySet`	Returns an empty `MiSet`. The set cannot be modified.
`emptyMap`	Returns an empty `MiMap`. The map cannot be modified.
`enumSetAllOf`	Returns a `MiSet` based on an `EnumSet` with specified contents.
`enumSetOf`	Returns a `MiSet` based on an `EnumSet` with specified content.
`singletonList`	Returns a singleton `MiList`.
`singletonMap`	Returns a singleton `MiMap`.
`singletonSet`	Returns a singleton `MiSet`.
`unmodifiableList`	Returns an unmodifiable `MiList` based on a provided `List`.
`unmodifiableListCopy`	Returns an unmodifiable `MiList` which contains a copy of a provided `List`.
`unmodifiableMap`	Returns an unmodifiable `MiMap` based on a provided `Map`.
`unmodifiableMapCopy`	Returns an unmodifiable `MiMap` which contains a copy of a provided `Map`.
`unmodifiableSet`	Returns an unmodifiable `MiSet` based on a provided `Set`.
`unmodifiableSortedMap`	Returns an unmodifiable `MiSortedMap` based on a provided `SortedMap`.

1.2.4 On the Use of `Strings`

When programming against the Extension Framework and the Maconomy API's, you need to use textual identifiers for a lot of different purposes. For example, when referring to a specific container, you need to specify the name of the container. When looking up field values of some record, you need to refer to the name of that field. Also, when displaying a message to the end-user, you need to describe the text to be shown.

Java has a generic type for handling strings: the `String` type. When you use `String`s, sometimes,the casing of the string-content matters, other times, it doesn't. For example, for any identifier (e.g., data-base field, container name, field name), the casing is irrelevant: the framework will always ensure this. Other times, casing is highly relevant (e.g, for text shown to the end-user.)

If you think of the purpose of specifying the name of a field, and the purpose of specifying a text which should eventually be presented to an end-user, there's a huge difference! A text presented to an end-user should be nice and easy to read. The name of the field is some internal reference that has the purpose of uniquely identifying some value. There is nothing fundamental in having a textual name—a field might as well have been identified by a number. So the meaning and purpose of these usages are significantly different. You never want to mix these two. And you don't need things like advanced string manipulation for identifies like field names. For example, why would you want to slice out a small part of an id of some field?

Consequently, the framework internally[5] completely separates the two uses of strings. The identifiers are represented by a type called `MiKey` whereas texts that are meant to be presented to an end-user are represented by a type called `MiText`. Although both are implemented by internally keeping a `String`, they can't be mixed up without leading to compiler errors.

It is possible as an extension programmer using the Extension Framework to strictly follow this pattern. However we acknowledge that it is sometimes slightly cumbersome to do this rigorously. For example looking up a field with a certain name, or specifying a message to the end user, it is sometimes a bit annoying to being forced into converting `String`s into the proper type. Therefore, the extension programmer is in many cases offered the choice of being rigorous (always using either `MiKey` or `MiText`) or to use a more sloppy (but more readable) approach that uses `String`s directly[6].

The two interfaces have companion factory classes, `McKey` and `McText`, with factory methods for producing objects of the corresponding interface-type. By using Java's static import feature, the clutter can be minimized.

For example, `record.getStr(key("EmployeeNumber"))` is the rigorous way of obtaining the value of the String-field "`EmployeeNumber`" from a record. `record.getStr("EmployeeNumber")` is the more straight-forward (but less 'safe') way of expressing the same thing. For example:

```
// The rigorous approach (without static imports)
containerRunner.error(McText.text("Negative amounts are not
    allowed"));
// The rigorous approach (using static imports)
containerRunner.error(text("Negative amounts are not allowed"));
// The slightly more readable (but sloppy) approach
```

[5] And to some degree also externally

[6] The Extension Framework will then immediately convert the `String` into the proper type


```
containerRunner.error("Negative amounts are not allowed")
```

Although the compiler cannot check it, you should use the "sloppy" variant only when referring to literal `String`s. Whenever your argument is represented by some kind of variable, you should use the rigorous approach. For example:

```
// Literal strings are fine to use
record.getInt("Quantity");
//! Not recommended: storing Strings in variables for later use
String e = "EmployeeNumber";
String m = "Message for the end-user";
// ... (lots of code) ...
//! Oops! We refer to a String which is not really a field name!
   The compiler is unable to detect this!
record.getStr(m);

// In this case, the following is better
MiKey e = key("EmployeeNumber");
MiText m = text("Message for the end-user");
// ... (lots of code) ...
// The compiler WILL give an error. No harm done!
record.getStr(m);
// This will compile just fine; the types are correct!
record.getStr(e);
```

We recommend that you avoid using `String`s except as literal `String` arguments. We also recommend that you don't use `String`s to represent structures that might just as well (or more appropriately) be represented by a new class or interface. Since `String`s can be used to encode a lot of different types of information, it generally means that they are less type-safe.

1.2.5 Working with Data Types

The Maconomy ERP system supports a number of different data types:

Amount for working with monetary units, e.g., 14200.50. Amounts are always represented with 2 decimals.

Boolean for working with boolean values.

Date for working with date values. Notice that in Maconomy, Date and Time are separated. A special "date" value represents "no date", visualized as a blank.

Integer for working with integer numbers.

Popup for working with enumerated types. The "Popup" type is not meaningful by itself. The specific enumeration type is needed to represent a specific value.

Real For working with Real (decimal) numbers.

String For working with text strings.

Time For working with time value. Notice that in Maconmoy, Date and Time are separated.

All values of the above types are internally represented by a Java class called `McDataValue`. A value of this type may represent *any* of the above concrete types. This type is needed to obtain a generic interface to data at large. However, for every value instance, that value will have a concrete type which is more specific that merely `McDataValue` which is an abstract super-type for all the concrete types. The concrete classes used to represent the above values are: `McAmountDataValue`, `McBooleanDataValue`, `McDateDataValue`, `McIntegerDataValue`, `McPopupDataValue`, `McRealDataValue`, `McStringDataValue` and `McTimeDataValue`.

You should *never use class casting* to convert from a `McDataValue` to one of the concrete sub-types. As we shall see in Chapter 4 the Extension Framework provides ways to conveniently pick up values of a specific type.

Some times, however, you would want to convert a given `McDataValue` into a specific type, and as an extension programmer, you *know* what that type is. In this case, the framework provides a number of utility methods for converting a generic data value to a specifically typed data value. In case of a programming error (meaning that the value is not an instance of that type) the framework will throw an exception at run-time.

Often, however, working with specific Maconomy-encoded values is not exactly what you want either: sometimes, you would prefer to use a corresponding Java type instead. This is useful in order to utilize one of the many Java libraries or when performing trivial things such as `if`-statements. Again, the framework provides a mechanism to convert between standard Java types and the internal Maconomy-encoded types.

In general these methods are made available by the framework through utility classes (classes with static methods) listed below.

`McAmount` which is used to convert between `McDataValue`/`McAmountDataValue` and `BigDecimal` which is the Java class chosen for representing amounts in Java. Notice that floating-point `double` is inadequate for monetary units due to imprecision and rounding issues!

`McBool` which is used to convert between `McDataValue`/`McBooleanDataValue` and `boolean` which is the primitive Java type chosen to represent booleans.

`McDate` which is used to convert between `McDataValue`/`McDateDataValue` and `GregorianCalendar` which is the Java class chosen for representing dates. Usually the `McDateDataValue` is easier to work with than the `GregorianCalendar`. Only `GregorianCalendar` is a standard Java type, `McDateDataValue` is not.

`McInt` which is used to convert between `McDataValue`/`McIntegerDataValue` and `int` which is the primitive Java type chosen to represent integers.

`McPopup` which is used to convert between `McDataValue` and `McIntegerDataValue`. As there is no obvious choice for representing a "popup value" as a standard Java class this is not directly possible. You can choose an *aspect* of a popup value that you want to convert into some Java type. If you need the *ordinal value* you can convert into a Java `int`, if you need the *literal value* you can convert into a `MiKey`. Notice that `MiKey` is not a standard Java type. It represents a case-insensitive String which does not exist as a standard Java type.

`McReal` which is used to convert between `McDataValue`/`McRealDataValue` and `BigDecimal` which is the Java class chosen for representing real numbers in Java. Notice that floating-point `double` is inadequate for this purpose due to imprecision and rounding issues!

`McStr` which is used to convert between `McDataValue`/`McRealDataValue` and `String` which is the Java class chosen for representing text strings in Java.

`McTime` which is used to convert between `McDataValue`/`McDateDataValue` and `GregorianCalendar` which is the Java class chosen for representing time. Usually the `McTimeDataValue` is easier to work with than the `GregorianCalendar`. Only `GregorianCalendar` is a standard Java type, `McTimeDataValue` is not.

Notice that these classes are not representing the actual values; they are only used as library method name-spaces! Hence, you cannot create a value of type ~~McDate~~, but using the `McDate` class you can create objects of type `McDateDataValue`.

In general, the above mentioned utility classes provide two methods:

Method	Remarks
`val`	This method converts either a generic `McDataValue` or the specific Java-type into the specific `McDataValue` variant, e.g., `McDateDataValue` or `McStringDataValue`.
`of`	This method converts generic (or specific) `McDataValue` into the Java type associated with the type.

For example, the following code shows how to construct a generic data value, and how to convert it back to a Java type.

```
// Construct a boolean value with the internal encoding
McBooleanDataValue boolVal = McBool.val(true);
if (McBool.of(boolVal)) {
  ...
}

// Obtain some generic data value, known to be of
// type String from somewhere
McDataValue genericVal = ...;
```

```
// The generic value can be converted to a specific sub-type
McStringDataValue sdv = McStr.val(genericVal);
// And it can be converted directly to the Java representation
String s = McStr.of(genericVal);
```

Apart from the possibility to convert between various formats representing values, the above mentioned utility classes have a number of convenience methods for working with the values.

Amount-related utilities: `McAmount`

Method	Remarks
`add`	Returns a `McAmountDataValue` having the value corresponding to adding the two argument values.
`compare`	Returns an `int` that indicates how the two arguments, a and b, compare to each other: $\texttt{compare}(a,b)$ is $\begin{cases} < 0 & \text{if } a < b \\ > 0 & \text{if } a > b \\ = 0 & \text{if } a = b \end{cases}$
`divide`	Returns a `McAmountDataValue` having the value corresponding to dividing the two argument values.
`multiply`	Returns a `McAmountDataValue` having the value corresponding to multiplying the two argument values.
`negate`	Returns a `McAmountDataValue` having the negated value of the provided argument.
`subtract`	Returns a `McAmountDataValue` having the value corresponding to subtracting the two argument values.

Boolean-related utilities: `McBool`

Method	Remarks
`TRUE`	A constant representing the value `true`.
`FALSE`	A constant representing the value `false`.
`and`	Returns a `McBooleanDataValue` having the value corresponding to the value $a \wedge b$ where a and b are the provided arguments.
`or`	Returns a `McBooleanDataValue` having the value corresponding to the value $a \vee b$ where a and b are the provided arguments.

Method	Remarks
`not`	Returns a `McBooleanDataValue` having the value corresponding to the value $\neg a$ where a is the provided argument.

Date-related utilities: `McDate`

Method	Remarks
`addDays`	Returns a `McDateDataValue` which is a specified number of dates after a given date.
`addMonths`	Returns a `McDateDataValue` which is a specified number of months after a given date.
`addYears`	Returns a `McDateDataValue` which is a specified number of years after a given date.
`compare`	Returns an `int` that indicates how the two date arguments, a and b, compare to each other: $$\texttt{compare}(a,b) \text{ is } \begin{cases} < 0 & \text{if } a < b \\ > 0 & \text{if } a > b \\ = 0 & \text{if } a = b \end{cases}$$ The "null date" is considered less than all other non-null dates.
`day`	Returns an integer representing the day of the argument date. For example, if the date is July 2, 2010, the day would return 2. If the date is the null date, an exception is thrown.
`month`	Returns an integer representing the month of the argument date. For example, if the date is July 2, 2010, the month would return 7. If the date is the null date, an exception is thrown.
`year`	Returns an integer representing the year of the argument date. For example, if the date is July 2, 2010, the year would return 2010. If the date is the null date, an exception is thrown.
`isNull`	Returns `true` if the argument date is the null date, `false` otherwise.
`nullDate`	Returns a `McDateDataValue` which has the null-date value.
`today`	Returns a `McDateDataValue` which has the value of the current date.
`val`	The `val` method is found in a version that takes three integer arguments representing the day, month and year. Hence, this method returns a `McDateDataValue` that corresponds to the specified date.

Integer-related utilities: `McInt`

Method	Remarks
`add`	Returns a `McIntegerDataValue` having the value corresponding to adding the two argument values.
`compare`	Returns an `int` that indicates how the two arguments, a and b, compare to each other: $\texttt{compare}(a,b)\text{ is }\begin{cases} <0 & \text{if } a<b \\ >0 & \text{if } a>b \\ =0 & \text{if } a=b \end{cases}$
`div`	Returns a `McIntegerDataValue` having the value corresponding to the integer part of dividing the two argument values.
`multiply`	Returns a `McIntegerDataValue` having the value corresponding to multiplying the two argument values.
`negate`	Returns a `McIntegerDataValue` having the negated value of the provided argument.
`subtract`	Returns a `McIntegerDataValue` having the value corresponding to subtracting the two argument values.

Popup-related utilities: `McPopup`

Method	Remarks
`isNil`	Returns true if the provided value represents a blank (“nil”) popup value.nil value|see popups, nil value
`isRaw`	Returns true if the provided value represents a “raw” popup value. I.e., a popup value where the ordinal is currently unknown.
`of`	Returns the literal value of the provided popup value as a `MiKey`.
`ordinalOf`	Returns an `int` representing the ordinal value of the provided popup value. If the provided value is a “raw” popup value, this method throws an exception.
`nil`	Returns a `McPopupDataValue` having the blank (“nil”) value for the specified concrete type.

Real-number-related utilities: `McReal`

Method	Remarks
`add`	Returns a `McRealDataValue` having the value corresponding to adding the two argument values.
`compare`	Returns an `int` that indicates how the two arguments, a and b, compare to each other: $\texttt{compare}(a,b)\text{ is }\begin{cases} <0 & \text{if } a<b \\ >0 & \text{if } a>b \\ =0 & \text{if } a=b \end{cases}$
`divide`	Returns a `McRealDataValue` having the value corresponding to dividing the two argument values.
`multiply`	Returns a `McRealDataValue` having the value corresponding to multiplying the two argument values.
`negate`	Returns a `McRealDataValue` having the negated value of the provided argument.
`subtract`	Returns a `McRealDataValue` having the value corresponding to subtracting the two argument values.

String-related utilities: McStr

Method	Remarks
`EMPTY`	Returns a `McStringDataValue` representing the empty string.
`isLike`	Returns a boolean indicating whether a given `McStringDataValue` represents the same value as a given `String`, *not taking casing into account*.
`text`	Returns a `McStringDataValue` having the content of a specified `String` argument. The returned value does not have a maximum length!
`trunc`	Returns a `McStringDataValue` having the content of a specified `String` argument, truncated to the maximum length handled by the Maconomy server.
`val`	Similar to `trunc`.

Time-related utilities: `McTime`

Method	Remarks
`compare`	Returns an `int` that indicates how the two time arguments, a and b, compare to each other: $\texttt{compare}(a,b) \text{ is } \begin{cases} < 0 & \text{if } a < b \\ > 0 & \text{if } a > b \\ = 0 & \text{if } a = b \end{cases}$ The "null time" is considered less than all other non-null time values.
`hours`	Returns an integer representing the hour of the argument time value. For example, if the time is 14:05:27, the hours would return 14. Notice the 24-hour notation. If the time is the null time, an exception is thrown.
`minutes`	Returns an integer representing the minutes of the argument time value. For example, if the time is 14:05:27, the minutes would return 5. If the time is the null time, an exception is thrown.
`seconds`	Returns an integer representing the seconds of the argument time value. For example, if the time is 14:05:27, the seconds would return 27. If the time is the null time, an exception is thrown.
`isNull`	Returns `true` if the argument time is the null time, `false` otherwise.
`nullTime`	Returns a `McTimeDataValue` which has the null-time value.
`now`	Returns a `McTimeDataValue` which has the value of the current time of the day.
`val`	The `val` method is found in a version that takes three integer arguments representing the hours, minutes and seconds. Hence, this method returns a `McTimeDataValue` that corresponds to the specified date. The hour argument must be provided using 24 hours notation. Hence 0 is midnight, 12 is noon and 23 is 11 P.M.

1.3 Coding Conventions Used in this Manual

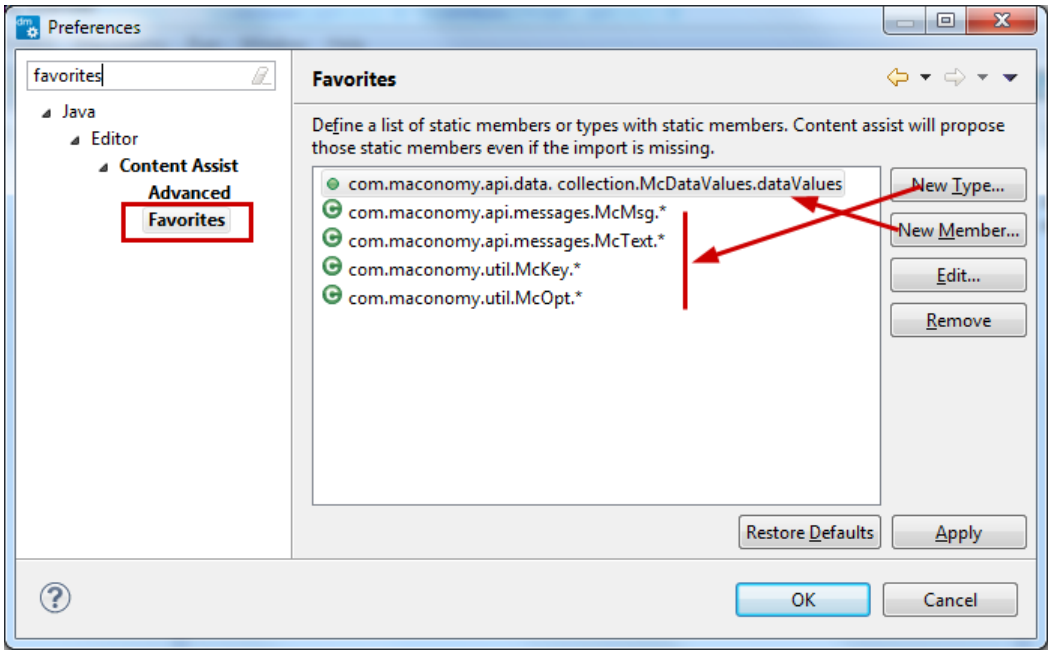
Throughout this manual, a number of code snippets are shown. In most cases, the entire listing is not included. For example, import statements (which are most frequently figured out by the Maconomy Extender automatically) are not shown.

However, in many listings, a number of "static imports" are assumed. The following

table shows the methods used, and the counterpart to be used if no static import is made:

Method	Corresponds to	Required static import
`opt(x)`	`McOpt.opt(x)`	`com.maconomy.util.McOpt.*`
`key(name)`	`McKey.key(name)`	`com.maconomy.util.McKey.*`
`msg(str)`	`McMsg.msg(str)`	`com.maconomy.api.messages.McMsg.*`
`msg(str, f)`	`McMsg.msg(str, f)`	`com.maconomy.api.messages.McMsg.*`
`text(str)`	`McText.text(str)`	`com.maconomy.api.messages.McText.*`
`dataValues()`	`McDataValues.dataValues()`	`com.maconomy.api.data.collection.McDataValues.dataValues`

It is possible to configure the Maconomy Extender to automatically recognize the above method names, automatically inserting the static imports when the above method names are entered.



The screen shot shows how to configure the preferences in the Maconomy Extender.

1.4 A Note for PDM Developers

This section is targeted specifically at readers used to extending the Maconomy Portal using PDM [Pdm] and M-Script [MSca]. If you are not used to making extensions to Maconomy in this way, you may safely skip this section.

Although the Extension Framework described in this book is in many ways comparable to PDM and the M-Script Maconomy API [Mscb], there are some significant differences.

- PDM-extensions are *client-side*, whereas the Extension Framework offers *server-side* extensions.
- The Extension Framework is based on a compositional architecture whereas PDM has a monolithic architecture.
- With the Extension Framework, all contributions are first-class citizens of the framework, and can be used within the framework on equal terms with any other contributions. In fact, the implementation of the standard Maconomy containers[7] is nothing but an extension written in the framework itself.

1.4.1 Server-Side versus Client-Side

So, why would you care whether an extension is server-side or client-side? There are several reasons for you should:

- Client-side extensions—by definition—can be leveraged only through one particular client interface. A server-side extension may be used for several different client interfaces[8].
- Separation of concerns: with server-side extensions, you are completely spared from worries concerning the GUI and user-interactions. In other words, you can concentrate on the business logic provided by your extension. You don't have to worry about various user-interactions and GUI presentations. This make your code easier to write and maintain.
- Since the presentational layer is completely separated from your extensions, the end-users will experience a consistent user-interface, leading to higher usability.
- All client-side functionality automatically becomes available for your extensions. The client(s) cannot distinguish your extensions from others (or from the "standard.") You don't have to manually support things like drag'n'drop of table rows, making your functionality available through wizards, offering traffic lighting, or worrying about how to react to various workspace constellations. If you fulfill the contract required of extensions, you get it!

Experience shows that solutions offered through the workspace client are often thought in a different way compared to how a solution to the same need would be envisioned in the Maconomy portal. Working with the workspace client requires a change of mind set

[7] Formerly known as Maconomy dialogs

[8] As of the present version, the extensions to the coupling service are only available though the workspace client

compared to working with the portal. One of the central reasons for this is exactly the difference between client-side extensions and server-side extensions, combined with the powerful features of MDML [CMd] and MWSL [QMw]—these things change how you do various things, and how an effective solution for end-users is built.

1.4.2 Compositional Architecture

The compositional architecture of the Extension Framework means that you can extend other extensions. And others can extend yours. Since all functionality is offered through extensions[9], your contributions are indistinguishable from others. And since you can extend others' work, your contributions can be extended—by you or by others. This fact makes it easier to separate concerns, add/remove extensions at a more fine-grained level, and increases the clarity of each individual contribution. It also lowers the risk of code-duplication, since you avoid re-implementing what might otherwise be overshadowed.

With PDM, you need to explicitly program a portal component offering the functionality of a Maconomy dialog, or whatever custom logic you choose to offer. In addition, such a portal component needs to be combined with whatever extension logic you want. If you wish to extend this component further, you need to either modify the source code of that particular portal component, or you need to re-program it from scratch, adding the extensions there. Although M-Script code can be modularized there is no framework support for re-using extensions of semantically equivalent entities.

1.4.3 All Extensions are First-Class Citizens

Another consequence of the compositional architecture of the Extension Framework is that once an extension is made and installed, it will automatically be invoked when *other* extensions programmatically interacts with whatever container you extend. For example, suppose your task is to make an extension that automatically ensures that when a job is created, it is blocked for registration, and that the job has status "Quote." And that whenever the job is converted to status "Order", the job is automatically unblocked for registration. The customer wants the block-for-registration field to be removed from the user-interface. Now suppose that your extension is going to be installed in an environment where another extension is already installed: that extension has the capability of creating jobs linked with sales orders in certain cases. The architecture of the Extension Framework means that the only thing you need to do is to focus on *your* task: making an extension that extends the "Create" operation on the Jobs container, and the "ConvertToOrder" action on the Jobs container. Once your extension is installed, other extensions accessing the "Jobs" container programmatically will automatically

[9]Even the standard Maconomy functionality is offered through an extension that is prepackaged together with the coupling service

invoke your extension, without knowing it. In a PDM context, you would have had to modify or re-write the extension that someone else did, in order to ensure that the functionality provided by your extension is obeyed whenever a job is created or converted into "Order" status, no matter the origin of the event.

Chapter 2

Overview

This chapter gives an overview of the Extension Framework and how it fits into the Maconomy 2.0 architecture. It also introduces the main concepts that extension programmers will need to know and work with.

2.1 The Maconomy 2.0 Architecture

Maconomy 2.0 is very similar to previous versions of Maconomy. Indeed, it is still possible to use the same technologies and front-ends as in previous versions. For example, when upgrading from Maconomy X1, clients may continue to use a portal installation, Jaconomy client etc. Just as usual. A high-level overview of the components in the Maconomy 2.0 architecture stack is shown in Figure 2.1.

The main new contribution of Maconomy 2.0 is a new platform which uses the *Workspace Client* as the user front end. Rather than presenting Maconomy "dialogs" (like the Jaconomy client) or specially coded components (like the Portal), it presents *workspaces*. Workspaces are specified declaratively using the MWSL language [QMw]. A workspace defines a hierarchical structure in which data is presented, and the system will ensure that data is always updated relative to the specification. You may think of a workspace as a tree of nodes, where each node contains data (one or more records) and where the edges in the tree are annotated to indicate which data should be shown in the next node. The edges primarily use foreign keys to specify how to derive the key values for data shown in the next node. This value is always derived from the current record in the node which is a parent to the node from which a key value is being derived.

Interpretation of the data and what data to present in each node is managed by a workspace engine. The workspace engine is a part of a server-side component introduced in Maconomy 2.0: the *Coupling Service*. Figure 2.2 shows a very small workspace where three nodes are tied together in a workspace. The workspace engine will ensure that

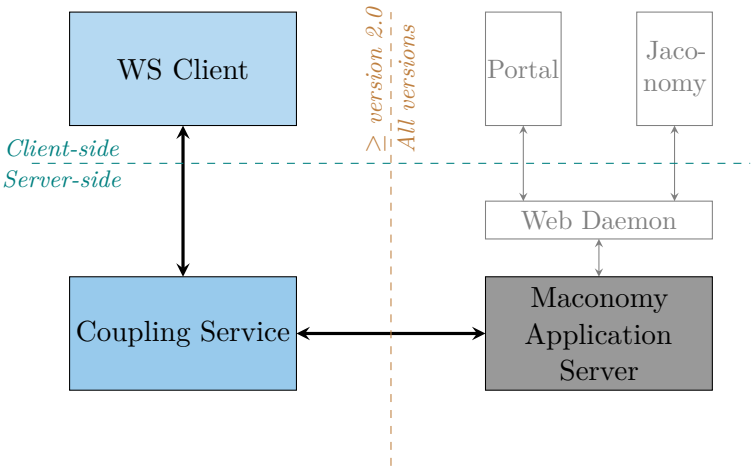
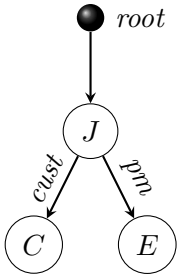


Figure 2.1: The Coupling Service and the Workspace client are in scope only for version 2.0 and higher. The Extension Framework described in this document concerns only the coupling service and whatever clients may interact with the coupling service (currently only the workspace client.) The Portal, the Jaconomy client and Web Daemon components are still functional but are not affected by the Extension Framework. In this figure they are shown in a dimmed shade because they are not discussed in this document. The Maconomy Application Service is (as usual) responsible for executing the "core" Maconomy application logic. This logic is made accessible to the coupling service by a specific extension that handles Maconomy containers by interacting with the Maconomy Application Server. Notice the dashed line which indicates the client-side and server-side. Since all extensions are made in the coupling service, such extensions are server-side extensions.)



J:	`Jobs`
C:	`Customers`
E:	`Employees`
cust:	`CustomerNumber_Customer`
pm:	`ProjectManager_Employee`

Figure 2.2: A small workspace. The root note is just a starting point tying together parallel nodes at the top. The top node J contains data from the `Jobs` container. Using the foreign key `CustomerNumber_Customer` which determines a customer based on the `CustomerNumber` field in the Jobs container, it is declared that data in the node C must be found in the `Customers` container. Also, using the foreign key `ProjectManagerNumber_Employee` which determines an employee based on the `ProjectManagerNumber` field in the Jobs container, it is declared that data in the node E must be in the `Employees` container.

the job having focus in the jobs node will be used to derive the data contained by the two other nodes. Hence, the content of the job will contain a customer number, and that customer number will be used to determine the data presented in the `Customers` container. Similarly, the workspace engine ensures that no matter which employee is specified as the project manager of the job, that employee is shown in the E node, and that the data will be taken from the `Employees` container. The *visual appearance* of this will be managed by the workspace client. Data from a given node is presented in a "panel" using either a form layout, a table layout or a filter layout. This depends on what part of the container the data is taken from, which is also specified in the workspace declaration. This part is referred to as the *pane*. The fact that each individual pane can be referred on its own is what makes it possible to configure the workspaces from building blocks that are more fine-grained than entire containers.

2.2 Containers and Panes

We have been referring to the term *container* several times already. As explained above, data in a workspace, shown by the workspace client, is taken from a container. So what is a container? In earlier versions of Maconomy, the term has been "dialogs." In fact, a Maconomy dialog is just a special case of a container. Since the Extension Framework is in principle independent of Maconomy, we have chosen to generalize the dialog concept. The result is a container. Therefore, throughout this document, we shall refer to the term "container" rather than "dialog."

A container comprises a number of *panes*. Each pane of a given container has a specific

Pane Type	*Layout Type*
Card	Form
	Browser
	Report
Table	Table
Filter	Filter

Table 2.1: Overview of how pane type and rendering option are tied together

type, and a specific *name*. Each workspace node contains data from one pane. Visually, panes are presented in so-called "panels" in the workspace client. Three types of panes exist:

Card panes which contain a single record. Examples of card panes are numerous for Maconomy containers, for example the "Jobs" container and the (card part) of the "Time Sheets" container.

Table panes which contain zero or more records. Examples of table panes are numerous for Maconomy containers, for example, the (table part) of the "Job Budgets" container and the (table part) of the "Time Sheets" container.

Filter panes which, like tables, contain zero or more records. For filters, arbitrary sub-sets of the potential content may be shown depending on the possible filter options applied by the user and/or layout designer. Filters also support things like paging (e.g., "Showing records 1 to 25 of 3200.") In former versions of Maconomy, filter panes have been there, but have appeared to be "second class" members of the family. In Maconomy 2.0, filter panes are treated the same as card and table panes. Examples of filters are: the pane which appears when you press "`Ctrl+F`" in Jaconomy. In the Jobs container, it will present a filterable list of jobs. Also the search panes that pops up when you press "`Ctrl+G`" are instances of filter panes.

Table 2.1 shows how different pane types influence the layout options in MDML and the workspace client. Maconomy containers are found in one of these configurations:

- Filter
- Filter – Card
- Filter – Card – Table
- Card
- Card – Table

Other configurations might exist. With the Extension Framework you can make other configurations, although there is only easy support for the above configuration plus "Filter – Table" and "Table" in the current version.

2.3 Extension Principles

By now, it should be clear that containers play a very central role with the Extension Framework. In fact, the only thing you can do is:

- Extend existing containers
- Contribute new containers

When contributing a new container, you must obviously specify which configuration your container has, i.e., what panes are found in the container, and what their names[1] are. In addition to this, you specify which *events*[2] are supported by each pane in your container. The following events exist:

Initialize The first step in the two-step process of creating a new record. Upon creating a new record, an end-user is always presented with a template for the new record. This template record may or may not be modified before it is finally created. If the user chooses to cancel/revert the operation at this stage, no record will be created.

Create The second step in the two-step process of creating a new record. This event is run after the end-user has had a chance of modifying a template record. After this event is done, the new record is expected to be persisted in the persistence layer, typically the database. There is no guarantee that all fields will have the value entered by the end-user. The business logic of your container may choose to change some or all of the values.

Read This event occurs whenever data needs to be refreshed. For example, initially when opening a workspace, the visible panes will be refreshed. Whenever some event takes place in a pane of a workspace tree, the workspace engine will automatically request this event for all panes that may need to be refreshed.

Update This event occurs when a change is made to some record. For example, when an end-user changes the numbers of hours registered on a time sheet line. After this operation successfully completes, the record is expected to be updated in the underlying persistence layer, if needed. There is no guarantee that all fields will have the value entered by the end-user. The business logic of your container may choose to change some or all of the values.

Delete This event occurs when a record is being deleted. If this operation completes successfully, it is expected that the record no longer exists in the underlying persistence layer.

Print This event occurs when a "Print This" is requested on a given pane. Exactly what that means is entirely up to the business logic of the container. Typically it

[1] Currently, there is only "easy" support for the configurations mentioned in 2.2, and only support for panes with certain standardized names. Except for quite extreme cases, this should suffice.

[2] Basically a user-operation, or an operation requested externally

is expected that some document/report will be produced showing data that relates to the data represented by the pane in question. This operation *might* change the record as a side-effect, e.g, a `LastPrintedDate` field.

Move This event occurs when a record in being "moved." Moving can be one of the following user operations

- Move current line "up." Possible in some tables (or tree-tables)
- Move current line "down." Possible in some tree (or tree-tables)
- "Indent" the current line. Possible in some tree-structured tables.
- "Outdent" the current line. Possible in some tree-structured tables.
- "Drag'n'drop" the current line to a new position in the table (or tree-table.) This operation is possible in some tables and some tree tables.

Action This event occurs whenever a "named action" is invoked on a pane in a container. You can declare which (if any) named actions are available in each pane of a container. Examples of such actions are "Submit Time Sheet", "Convert to Order" and "Post."

Lock This event occurs when the system determines that it should be attempted to lock a specific record. Your container may or may not support locking.

Unlock This event occurs when the system determines that an acquired lock should be released.

When extending a container, you can (for each pane) choose which of the events supported by that pane in the container, you wish to extend. Meaning that you have a choice of having your code executed whenever one of these events occur. You may also add support for new events, typically adding new named actions. In principle, you can add support for other events as well. Finally, you may *remove* support for operations that would otherwise be supported by the container you extend. For example, if you don't want end-users to, e.g., delete jobs, you may do so by entirely removing the support for deleting jobs.

As mentioned in the introduction, the Extension Framework has a compositional approach to extending containers. This means that several different extensions for the same container may co-exist. Every container has at least *one* extension: the defining instance[3]. We call the defining instance the *root*. For any specific container, there will be exactly one such root. An attempt to declare several roots for the same container will yield a run-time error.

It is possible to extend an existing container. This is done by contributing an implementation that declares itself an extension to a container. There can be many such extensions. The Extension Framework will ensure that these are ordered in some way.

[3]Remember that contributing a new container is considered "an extension." So in order for some container to be known, such an extension must exist

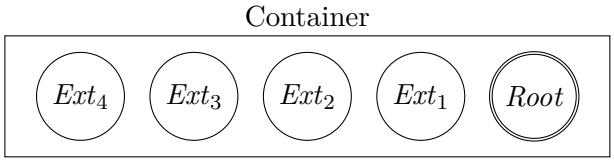


Figure 2.3: A container is made up of a number of extension contributions. *Root* marks the the root, and must be present. Zero or more extensions may be installed on top of the root. In this figure, the contributions $Ext_1 \ldots Ext_4$ are present. From the outside, it is indistinguishable how many extensions are present. The framework will organize the ordering of the contributions, always ensuring that the root is last.

The root will always be the "last." From the outside, it is impossible to determine how many different contributions are present. Externally, a given container will have a certain set of capabilities, and a certain behavior. Whether this is all contributed by the root or by numerous extensions is irrelevant externally. Figure 2.3 shows an example of a container comprising a number of extension contributions.

Extensions to an existing container can contribute to the behavior of the container being executed. This may be done in several ways:

- Additional behavior for the supported events may be contributed
- Support for events not supported by the container prior to extending it may be added. This covers standard actions (create, update, delete, print) as well as named actions.
- Events supported by the container prior to extending it may be removed. For example, it is possible to remove support for deleting, or the support for certain named actions.
- New persisted fields may be contributed.
- New calculated fields, known as *variables* may be contributed.

2.3.1 More on Container Events

The core of the Extension Framework are the container events: without container events, no information could be provided to end-users, and no interaction could take place. You may argue that everything is *about* the container events. Everything else is merely a matter of supporting the handling of these. Container events can be divided into two kinds: *data-carrying events* and *supportive events*. The data-carrying events are events used to interact with an end-user. They may operate on certain portions of the data, and they alter the data presented to the user, as well as on the persisted data entities. The supportive events are not expected to have side-effects to the persisted data, and are used to support the framework in interacting with containers. Some containers may not

support all data-carrying events, but every container must support the supportive events. The supportive events are:

Open "Opens" a container. This operation may not have any effect at all. A given container implementation can use this to initialize various data structures, if needed. The framework will invoke this event prior to other events made on a container.

Close The opposite of "Open." When the framework is done with some container, "Close" will be invoked. There is no guarantee how long the container is left open by the framework. Also, several events may be invoked between Open and Close.

Specify This event is used by the framework to describe the capabilities of a container. This description include the pane configuration of the container, the name and type of the fields in each pane, which fields are open or mandatory, which foreign keys are known by which pane, and which event actions are supported by a certain pane, i.e., which data-carrying events are supported by each pane of the container.

Restrict This event is invoked when a search (e.g., a "Ctrl+G"-search) is being executed from a field in this container. A search really means executing a "Read" event in some *other* container. The restrict event gives the container from where the search was started a chance to define certain restrictions that must be applied to that search. For example, when searching for Employees in some container, the container from where the search is performed may want to dictate that only employees that are not "blocked" should be comprised by the search result. Or that the employees should only be those employees belonging to the company specified in some field in the container from where the search is taking place. Such restrictions are sometimes called *foreign-key conditions*.

The data-carrying events have been mentioned in Section 2.3. All of the data event have similar characteristics: the input is a description of the actual data/constraints under which the event should be executed, and the output is a *container value*. A container value represents resulting values for zero or more panes of the container. Usually, a value for the pane in which the event is executed is expected, although this is only strictly required for Read-events. Each of the pane values in the resulting container value may be a *partial* value. A partial value means that the data is an addendum to a previous Read-result and the partial responses of other events having occurred since the latest full[4] pane value occurred.

Why may an event result in a value of *other* panes? The answer is performance. Depending on the actual container-implementation, results for several pane values may be known anyway. Making these values a part of an event-response-value makes the workspace engine capable of avoiding unnecessary re-reads. As an example, consider the `TimeRegistration`[5] container of the Maconomy application. This container comprises a card pane (indicating the user and the period for which the time registrations covers)

[4] I.e., non-partial
[5] Externally titled "SpeedSheet"

and a table pane (containing each of the specific registrations made in the specified period of time.) Whenever a line in the table is created, updated or deleted, the contents of the card part may change accordingly. For example, the card is capable of showing the total number of invoiceable hours registered in the specified period of time. The Maconomy Server calculates the changes to this number effectively by adjusting the existing value by a delta calculated from the exact event made in the table. Doing this is more efficient than re-calculating the value based on all lines in the table. So, when the resulting (partial) pane value is calculated for the table, the corresponding value is also made available for the card. Rather than throwing this value away, this value is made available as part of the response. depending on the actual needs, the workspace-engine may determine to use that card part (rather than having to re-read it), or it may decide that there is no use for the card part at all (in which case no harm is done.)

2.3.2 Using Data-Models

This all seems a little complex, right? The truth is that dealing with container values (comprising values for several panes) and dealing with pane values (comprising the value of the event record as well as possibly values for other records, either as partial or full pane values) is a somewhat complex task. For this reason, the Extension Framework has been built in a way that abstracts away most of this, letting the programmer focus on the important stuff (the business logic) rather than manipulating and handling container values in the right way. This is achieved by using a concept called *data models*. The purpose of data models is to let the extension programmer focus on the business logic, and to offer abstractions that lets the programmer re-use semantically similar behaviors in different containers. This includes abstracting away the tedious management of multiple pane values in container values as well as the management of partial or full pane values.

In the Extension Framework a container is an integral part: the central concept within the framework. This is not so for the data-model. A container doesn't strictly *need* an implementation based on data models. However, a lot of effort has been put into making it easy to provide new containers or to extend existing containers. This is done by providing abstract container implementations. In order to use these abstract implementations, the programmer is forced to use data models. And so, data models becomes a de-facto integral part of the Extension Framework. If "containers" are the heart of the framework, "data models" are the spirit. Our goal is that virtually all extension logic is contributed using data models.

The idea is that the data models describe the core business logic. Since events are tied to one record, the events of the data models are likewise associated with one record. For example, when a user updates a record, the data model will be invoked with information about that one record, and the task of the extension programmer is to implement validation and updates pertaining to that one record. Similarly, when a user deletes a record, the data-model is invoked in a context of the record being

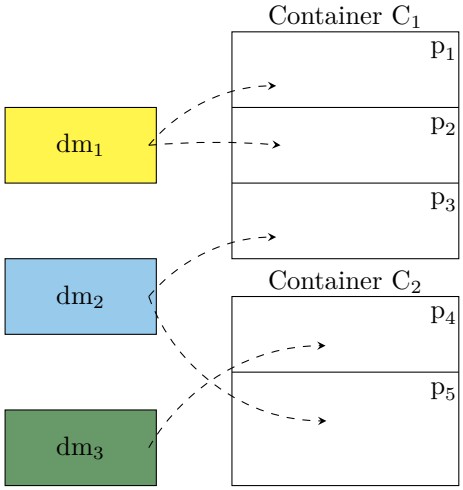


Figure 2.4: Each pane of a container is associated with a data model. The data models may be re-used for several panes and for different containers. In this example, the data model dm_1 is used for panes p_1 and p_2 of container C_1, dm_2 is used for pane p_3 in C_1 and pane p_5 in C_2. The data model dm_3 is used only for pane p_4 in C_2.

deleted. The abstract container implementations provided by the Extension Framework will automatically invoke the data models at the right time with the right information and tie the information together in the right format. Figure 2.4 shows how the same data model may be used for different panes and different containers.

2.3.3 "Mixing" Container Behaviors

Sometimes, the business logic of a certain action is not just about data in the container in question. Occasionally, you may want to programmatically interact with other containers on behalf of the user.

As an example, consider the case when a new employee is created. Assume you want to create an extension that will automatically create a new User and associate that user to the created employee. Obviously, this can be done manually by the end-user. But the purpose of the extension is to ease this work flow and prevent situations where the user-creation is forgotten, or the employee is not associated with the user. In the standard Maconomy system, there is no container that does all of this in one container: you need the `Employees` container to create the employee and the `UserInformation` container to create the user.

The Extension Framework allows "mixing" container behaviors by offering an API that gives access to other containers. Of course, you can also get programmatic access to new instances of the container of the current event. When interacting with containers through this API, it is important to notice that you get access to the container including

all extensions that may have been made to this container. This is because the behavior of a container is defined by *all* contributions (as visualized in Figure 2.3.)

2.3.4 Names and Name Spaces

In the Extension Framework, containers are referred to by their name. Often, we think of a name of a container as, e.g., `Jobs` og `TimeSheets`. Such names refer to the internal dialog names of the Maconomy system. However, this way of naming is a little informal. More formally, the name of a container is

namespace:*name*

Notice the character '`:`' separating the name space and the name. Whenever a container is referenced or defined in the framework, the name is expected to be in this form, although the name space can be left out. If the name space is not explicitly given, `maconomy` will be assumed.

The purpose of the name space is to avoid name clashing. All containers provided by default by Maconomy will have the name space `maconomy`. This name space is reserved for Deltek Engineering to use. When creating new containers as an extension programmer, you *must not* use this name space! Instead, you should choose a name space specific for your organization (e.g., `deltekUk`) or a name space specific to a given customer (e.g., `trifolium`.) The use of name spaces avoid current *and future* name clashes. For example, if you develop a container named `myCustomer:budgetControl`, then that container will never clash with and never overshadow a `maconomy:budgetControl` container if such a container is ever released in a future version of Maconomy.

2.4 OSGi

The Extension Framework and the coupling service are built using a run-time framework called OSGi [OSG08, MVA10]. OSGi is a component run-time for Java. Basically, this fact is not extremely important for an extension programmer. However, some knowledge of OSGi is required in order to understand what your extension can access, and what others can use from your extension.

The OSGi run-time manages components known as *bundles*. Basically, a bundle is a Java `.jar`-file, i.e., a number of Java packages containing Java classes and interfaces. A bundle may also include other kinds of file resources. Each bundle contains a manifest file, called `MANIFEST.MF` which declares which packages may be accessed from outside the bundle itself. Any java class or interface in a package that is *not* declared accessible from the outside cannot be referred from outside that bundle. Likewise, in the manifest file, each bundle must specify which other bundles or packages it depends on. This makes the OSGi run-time capable of managing these dependencies by refusing to start a bundle,

unless its dependencies can be fulfilled. Bundles can be dynamically installed, started and stopped, i.e., while the application is running.

Each bundle identifies itself with an ID, which is basically a string. It is recommended to have a naming convention corresponding to that used for Java packages (i.e., `com.mycompany.some.name`.) In addition to the ID, each bundle also has a version number (such as `1.0.0`.) When declaring dependencies to other bundles, it is possible to specify a range of version numbers that your bundle is compatible with. Again, the OSGi run-time will ensure that your bundle cannot be started unless these constraints are fulfilled. In principle, it is possible to have several versions of a bundle with a given name installed. If needed, both of these may be running; bundles requiring a specific version will refer to the matching instance.

The coupling service comprises a number of bundles. Some of these are relevant for extension programmers, others are not. You must specify which parts you wish to depend on. Fortunately, the Maconomy Extender IDE will help you with this. Most of the time, you don't have to think about it.

It is absolutely possible to make use of 3rd-party Java libraries. However, you need to expose these as OSGi bundles. Some libraries are already available as OSGi bundles, others are not. If you need a library that isn't wrapped as an OSGi bundle, all you really need to do is to make a `MANIFEST.MF` that describes the properties of the OSGi bundle.

Chapter 3

Getting Started

This chapter guides you through how to make your very first extensions using the Extension Framework. We shall do this by making the mandatory "Hello World" example, and then modify this slightly into a real extension. The overall concepts and ideas will be briefly explained. For a more in-depth explanation of the various parts, you are referred to the following chapters.

3.1 A "Hello World" Extension

In this section we shall develop an extension that simply displays the text "Hello World" in a field in a container. To do so, we must extend a container. We choose the `maconomy:Jobs` container for this purpose.

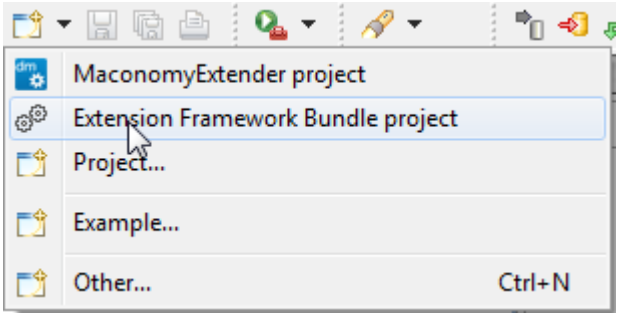
As previously explained, extensions are merely java bundles installed in the Coupling Service, which—in turn—is just a standard OSGi run-time. So, in principle, you can use any IDE or text editor to develop extensions. However, it is highly recommended to use the Maconomy Extender [Del13]. In fact, developing extensions is only officially supported by Deltek if you *do* use the Maconomy Extender. Fortunately, there are many good reasons for doing so. The Maconomy Extender is based on the state-of-the-art Java IDE, Eclipse, augmented with tooling that makes it particularly easy and useful to develop and deploy extensions on a running coupling service.

The Maconomy Extender is already used for configuring other aspects of a Maconomy installation, and it is assumed that the reader of this book already knows about it.

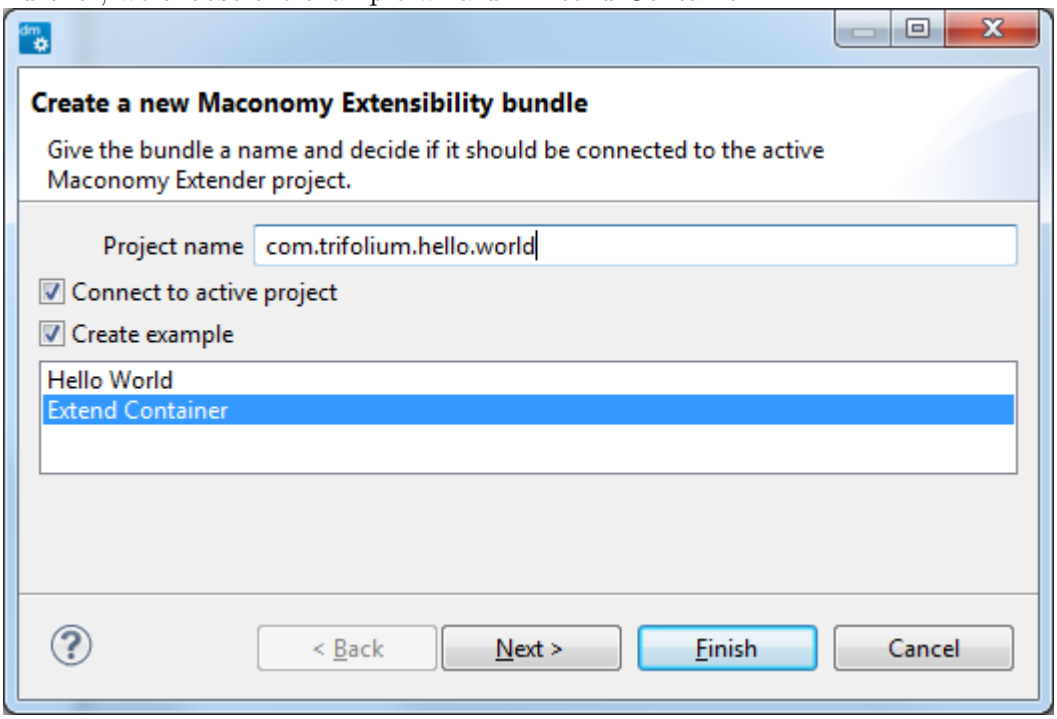
In order to implement an extension, you must have an *extension bundle project*. The bundle project must be associated with the relevant "Maconomy Extender project." Such a Maconomy Extender project comprises information about the Maconomy system being extended. As you may have several such Maconomy Extender projects, you need to specify which one your extension bundle project belongs to. You may have one or

more extension bundle projects associated to a given Maconomy Extender project. It is generally recommended that each “extension” is made in a separate bundle. This makes it easier to enable/disable each extension separately.

1. Activate the relevant Maconomy Extension project
2. Select the New → Extension Framework Bundle project



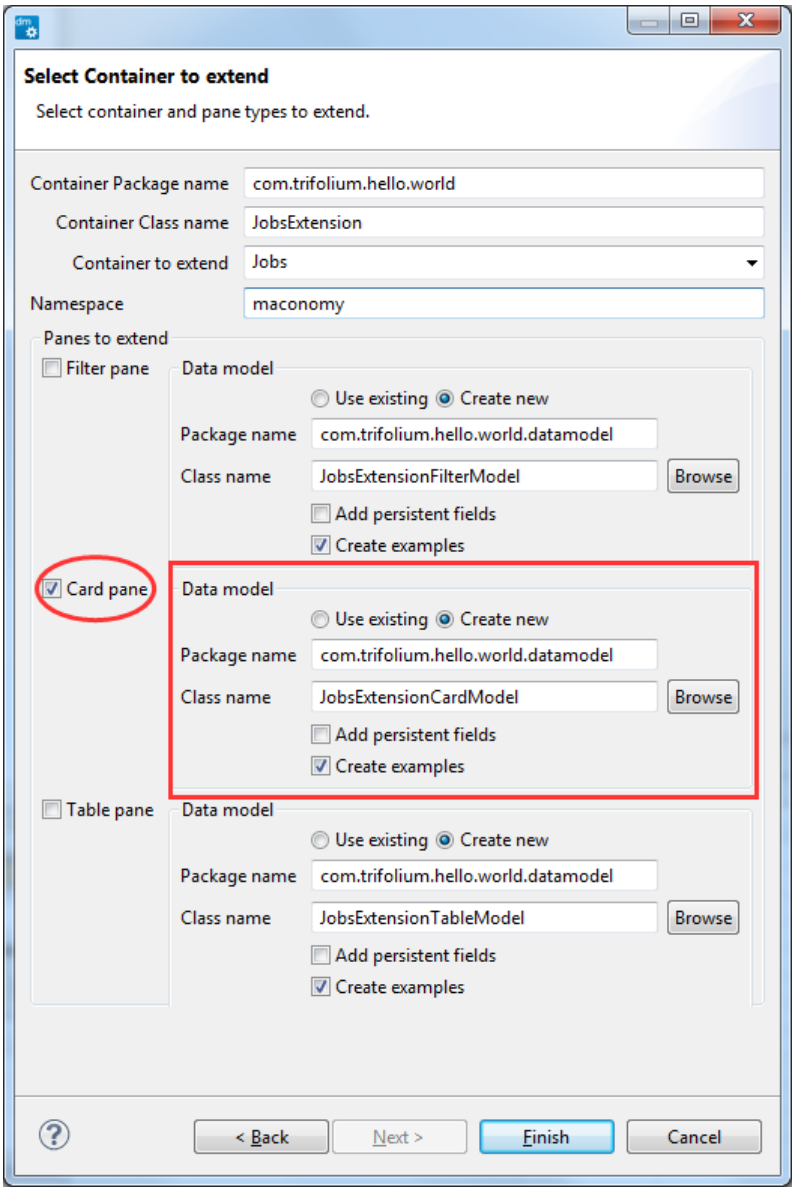
3. Now a wizard is launched. In this first page, make sure that the project is associated your active Maconomy Extender project. We also select that an example is generated.
4. Further, we choose the example wizard “Extend Container[1]”



5. In the next step of the wizard, we specify the package and class name for our

[1]Don’t be confused by the “Hello World” example; this will generate a (different) Hello World example as a new container. This is not what we choose in this example.

container extension. In this case `com.trifolium.hello.world` as package name and `JobsExtension` as the class name. Furthermore, we specify that the extended container is `maconomy:Jobs`. Further, we choose generation of a data model for the Card pane of the Jobs container, and we ask for a generated example.

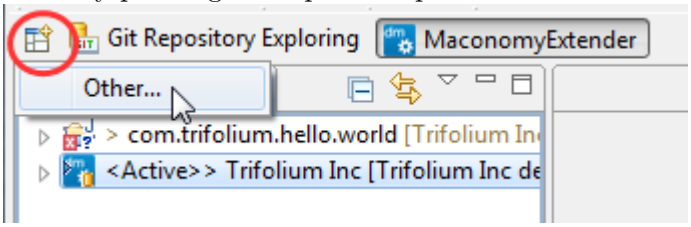


6. Upon pressing the "Finish" button, the Maconomy Extender will generate

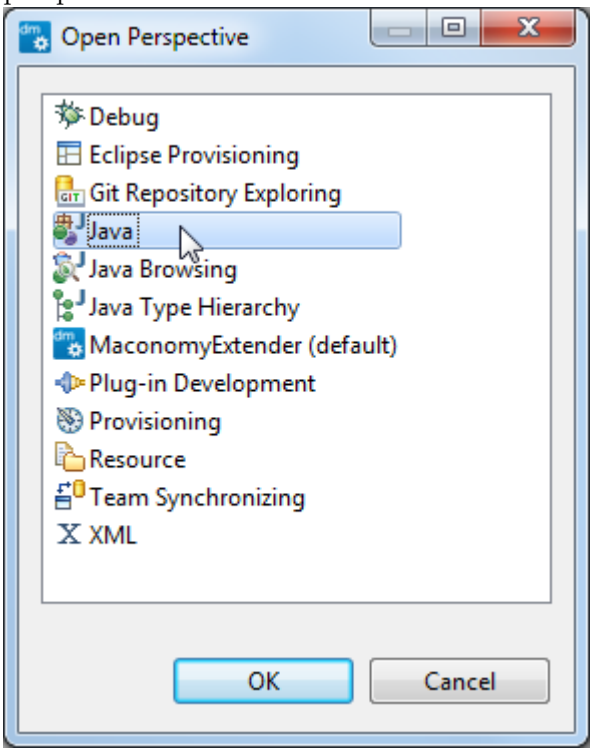
   - An extension framework bundle project, associated with the active Maconomy Extender project.
   - Two Java class files: `JobsExtension` and `JobsExtensionCardModel`.

- A `MANIFEST.MF` file specifying the OSGi properties of your bundle.
- A `plugin.xml` file declaring that this is an extension of the `maconomy:Jobs` container.

7. Naturally, you can inspect these files in the “Project Explorer” view in the Maconomy Extender. However, when developing Java, it is convenient to change the layout of the Maconomy Extender. You can do so by switching to the *Java perspective*. You do so by pressing the Open-Perspective button



8. If you haven't selected this perspective before, you select the Other... → Java perspective



9. Now, the layout of the Maconomy Extender will change to show views that are better suited for Java development. You can always switch back to the original layout by selecting the “MaconomyExtender” perspective.

3.1.1 Starting, Stopping and Debugging the Generated Code

Because we opted to let the wizard generate example code, the code is now ready to run! All the generated code does is registering a new action, tentatively titled "My New Action." When this action is invoked, a message is shown to the end user, saying "Hello World."

First, let's try to run this, just to verify that everything works. Below, we shall examine the code in order to understand how this is done.

In order to run the code, open the "Servers" view in Maconomy Extender. This is done from the menu Window → Show View → Other... and from there choosing Maconomy Extender → Servers. From this view, you can either add a coupling service (if you haven't done so already), or—while developing—you can specify the path to the coupling service you are developing against in the Properties of the main Maconomy Extender project. You do so by right-clicking the project and choosing "Properties." Then, in the opened dialog, you navigate to MaconomyExtender → Extensibility Framework and enter the path to the coupling service in the field labeled "Target platform path." If you have specified the path to the coupling service in this way, you can launch the coupling service *including the Java extensions you are developing* simply by clicking the "Run the coupling service"-button ( .) When you do so, the coupling service will be started. The Java extensions made in projects that are associated with the "Active" project will be installed in the running instance of the coupling service. Notice that the coupling service will run *locally on your machine*. Usually the Console view opens when the coupling service is started in this way[2]. It will display an OSGi console prompt

```
osgi>
```

From this OSGi console, we can verify that the coupling service is in good shape and that our new extension OSGi bundle is installed. To do so, type the following command:

```
osgi> ss com.trifolium
```

The "`ss`" is a console command meaning "short status." The optional "`com.trifolium`" means that we only want the status of bundles having an id that contains that string. You will see something like:

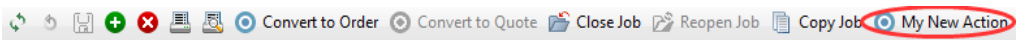
```
osgi> ss com.trifolium
"Framework is launched."


id  State       Bundle
67  ACTIVE      com.trifolium.hello.world_1.0.0.qualifier
osgi>
```

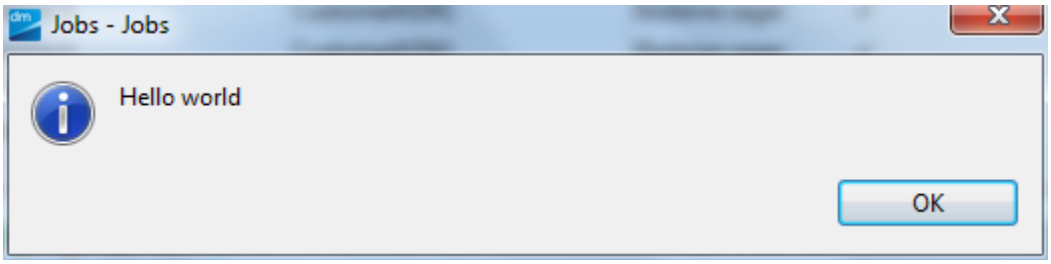
[2]You may have configured the behavior differently in your Maconomy Extender. If this is the case, you'll probably have no difficulty opening the Console view anyway.

This tells us that our bundle `com.trifolium.hello.world` is really installed and is “active”, meaning that it is started and the code in it can be invoked.

Now, start a matching client and connect to the coupling service. Then open the workspace found under Single Dialogs → Job Cost → Creation → Jobs. This workspace contains nothing but the `maconomy:Jobs` container. The layout for the card pane will show “all available actions,” so our new action should be present as well. And indeed, selecting a job in the filter will populate the card with a job as well. In the action buttons bar, it shows:



Let’s try to invoke that action: click it using the mouse. Now our extension bundle will be invoked, and will execute the code for that action. In this case, you can see the result in the client GUI:

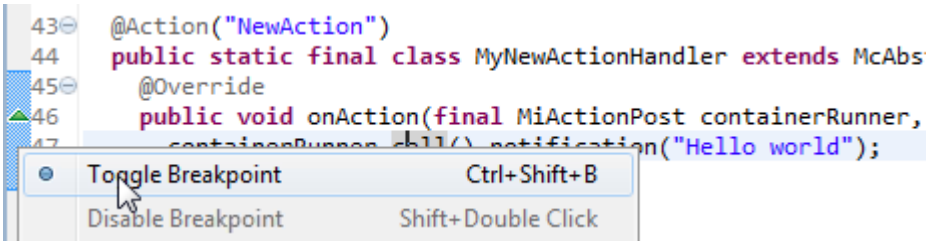


Congratulations—you just successfully invoked your first extension!

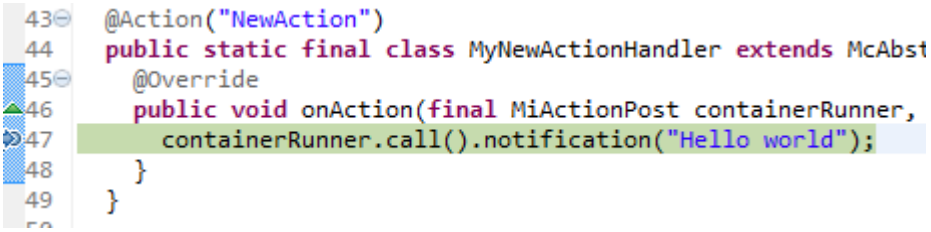
Terminate the coupling service program which you just launched from within the Maconomy Extender. You can do so by pressing the Terminate button (.)

For the purpose of demonstrating how to debug our program, open the generated Java class file: `JobsExtensionCardModel.java`[3]. Once this file is opened, you can get a “quick outline” of the file. Press `Ctrl+O` and start typing “`MyNewActionHandler`” (casing is not important here.) After typing a few characters, there’s only one possibility. Click on that. The Maconomy Extender will now navigate directly that the code that handles the action. An action handler is an inner class, which we shall not be concerned with right now. Inside that class definition, there’s a method called `onActionPost`. This method contains one line of code. In the Maconomy Extender, double click the left-hand-side margin or right-click the margin next to the line and select “Toggle Breakpoint.” Now a small bullet appears in the margin indicating that you have a break point at the specified line.

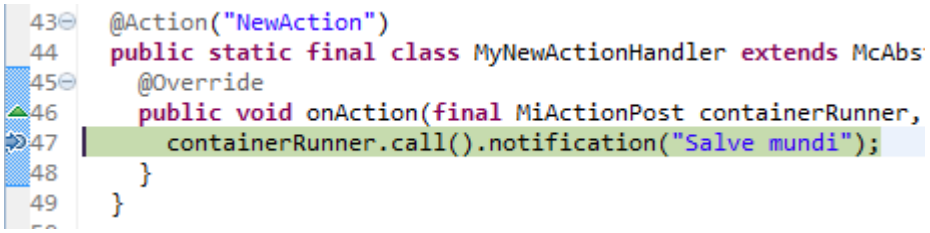
[3]Do this by pressing `Ctrl+Shift+R` and start typing the name of the file.



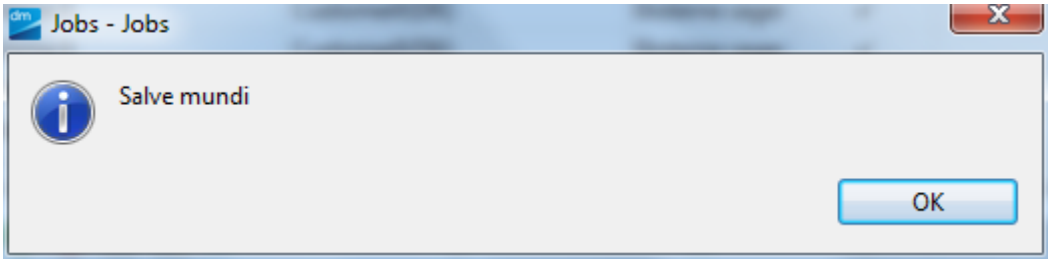
Now let's start the coupling service again, this time in *debug mode*. Like before, you do this from the "Servers" view. But this time you press the "Debug the coupling service" ( ) button. From the client, again try to invoke the "My New Action" button. This time, the code will be suspended at the break point. Upon doing so, the Maconomy Extender will ask whether you want to switch to the "Debug perspective." Like the Java perspective, the Debug perspective is an organization of views that is particularly useful when debugging Java code. Accept the switch to the debug perspective. The Maconomy Extender will now highlight the line in the code where execution is suspended.



At this point you have the ability to examine the value of various variables, single-step through the code etc. You *also* have the ability to change the code, and continue execution *using the changed code*. Let's try to modify the code slightly, just to verify that it works: we decide to greet the user in Latin rather than in English, so we change the String value into `Salve mundi` and save the file.



The code is still suspended, so we resume the code by pressing the Resume ( ) button. And indeed, the client will now greet the user in Latin:



3.1.2 A closer look at the code

Without going into too much detail, let us have a closer look at the code that does this. The generated files of interest are

- The `plugin.xml` file which declares that we extend the container `maconomy:Jobs`
- The Java class-file that implements the *container extension*: `JobsExtension.java`
- The Java class-file that implements the data model responsible for the actual logic: `JobsExtensionCardModel.java`

The `plugin.xml` file

The `plugin.xml` file defines which container-extensions are present, and which classes implement the behavior. In this case, we extend an existing container: the `maconomy:Jobs`.

When we want to extend a container (or contribute a new container), this is done by hooking into a specific *extension point*. An extension point is a named entry that is used to declare the addition of functionality within some given area. The extension framework provides the extension point

```
com.maconomy.api.container
```

This is the extension point you should use when extending containers or when providing new containers. Inside the scope of this extension point, you must declare *which* contributions you have. In this case, we have one contribution: the extension of the container `maconomy:Jobs`. The generated `plugin.xml` file looks like

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <?eclipse version="3.4"?>
3 <plugin>
4   <extension point="com.maconomy.api.container">
5     <extend container="maconomy:Jobs"
6         id="com.trifolium.hello.world:ExtendedJobs">
7         <factory class="com.trifolium.hello.world.JobsExtension$Factory"
              />
8     </extend>
```



```
9     </extension>
10  </plugin>
```

The interesting part starts with the `<extend container="maconomy:Jobs"`. This is where it is declared that we are extending the Jobs container. The `id` is not important for now. When a container is extended, you must provide a *factory* to the container contribution. A factory is a class capable of producing instances of some type. The specified class *must* implement the interface `MiContainerFactory`, and it must have a publicly accessible constructor with no arguments. The factory class is automatically generated by the wizard. The odd "$" notation is the syntax used to refer to nested Java classes. So, the relevant class in this example is the class called "`Factory`" which is declared inside the class "`JobsExtension`" in the package "`com.trifolium.hello.world`."

The container class: `JobsExtension`

So, the `plugin.xml` declares *how* to construct the container contribution: by using the specified factory class. The factory classes are typically very simple and non-interesting boiler-plate code, that will construct a class implementing an interface called `MiContainerEvents`. Implementing this interface from scratch is very complex and definitely not recommended! Instead, you should write a class that extends an abstract class that does all the boiler-plate code for you. In cases where you *extend* a container, you want to extend the class called `McAbstractExtendedContainer`. The abstract class takes care of virtually everything. The only thing you need to do is to configure it by specifying which data models should be used for which panes. In our case, we only want to extend the `card` pane. The configuration is done by implementing the method `defineConfiguration` (see Section 4.1.1.) Again, you reference a data-model *factory* rather than the actual data model. That's it. There's nothing more to do with the container class. The resulting code looks like this:

```
2   public final class JobsExtension extends
        McAbstractExtendedContainer {
3
4     private JobsExtension(final MiContainerFactory.MiResources
          resources) {
5       super(resources);
6     }
7
8     /** {@inheritDoc} */
9     @Override
10    protected MiExtended defineConfiguration() {
11      final MiContainerConfiguration.MiExtended configuration =
            McContainerConfiguration.McExtended.card(
            JobsExtensionCardModel.FACTORY);
12
13
14      return configuration;
```

```
15    }
16
17    public static final class Factory implements MiContainerFactory
          {
18      public MiContainerEvents createContainer(final
            MiContainerFactory.MiResources resources) {
19        final MiContainerEvents container = new JobsExtension(
              resources);
20        return container;
21      }
22    }
23  }
```

The data-model class: `JobsExtensionCardModel`

So, the container class isn't very interesting, nor is the `plugin.xml`. Then what is? As explained in Chapter 2, the actual semantics of a container takes place in the data-model. In this example, the data-model is implemented by the class `JobsExtensionCardModel`. In this simple “Hello world” example, the data model does the following:

- Declares the data-model factory class and a constant `FACTORY` (which is referred by the container.) The factory class is defined at the bottom of the file.
- Declares the existence of a new action with the internal name `MyNewAction`. This happens in the method `defineDomesticSpec()`. In general, this method is used to declare the contributions to the pane declared *at this level*—hence the term “domestic.”
- Defines the semantics of the `MyNewAction`. Actions are implemented by declaring an internal class which has an `Action`-annotation with a value that corresponds to the internal name of the action. For each action that are introduced (“added”) by this data-model, that action-handler class must extend an abstract class `McAbstractDataModelRootAction`. And then the method `onAction(...)` will be invoked by the framework when the action is run. In our case, all the action does is produce a notification to the end-user with the message “Hello world.”

The code of the data-model can be seen here:

```
2  public class JobsExtensionCardModel extends
       McAbstractExtendedDataModel {
3    private JobsExtensionCardModel(final MiDataModelFactory.
         MiResources resources) {
4      super(resources);
5    }
6
7    @Override
```



```
8     public MiExtended defineDomesticSpec(final MiDefine
          containerRunner) {
9       return McPaneSpec.McExtended.pane()
10          .addAction(key("NewAction"), "My New Action")
11          .end();
12    }
13
14    @Action("NewAction")
15    public static final class MyNewActionHandler extends
          McAbstractDataModelRootAction {
16      @Override
17      public void onAction(final MiActionPost containerRunner, final
            MiAction eventData) throws Exception {
18        containerRunner.call().notification("Hello world");
19      }
20    }
21
22    public static final Factory FACTORY = new Factory();
23    public static final class Factory implements
          MiExtendedDataModelFactory {
24      /** {@inheritDoc} */
25      @Override
26      public MiExtendedDataModel create(final MiDataModelFactory.
            MiResources resources) {
27        return new JobsExtensionCardModel(resources);
28      }
29    }
30  }
```


Chapter 4

Container Events

In this chapter we shall go into detail with the container and data-model event handling. From the outside, you can interact with a container by performing *container events* on that container. Hence, a container event is some operation performed on a container. Basically, there are two kinds of container events: *data-carrying* events and *supportive* events. Most of the "common events" are data carrying. The characteristics of these events are that they produce or consume data (or both.) The supportive events are either procedural in nature, or they represent functions that are basically data independent. Before you can implement the functionality for events, however, you need to define your container and bind data models to the container.

4.1 Implementing a Container (Contribution)

As hinted earlier, you can contribute to a containers behavior in two ways: Either you can *create* a container (i.e., define its existence) or you can *extend* an existing container. If you create a container you are obviously responsible for the "core behavior" of that container; there's nothing to rely on. Once you have created a container, others can (with or without your knowledge) extend that container.

If you extend a container, it means that someone else has defined the existence of the container you extend. Your contribution can be seen as an "addendum" to that container, resulting in an augmented container. From the outside, it is never possible to see or know whether one or more contributions are defining the behavior of the container. Figure 2.3 visualizes this behavior. What is the reason for this behavior? Why don't we just override the entire container or always insist of creating new containers that may then programmatically delegate to already existing containers? The answer to this question is that we believe the current design gives the right balance between ease of use (ease of programming), separation of concerns, management of configurations and code robustness. Obviously, there may be cases where this approach is slightly more difficult

to work with than other designs. On average, however, we believe that this design is a good compromise. It has been a key design goal to make it as simple as possible to write new containers and to extend existing containers, and to ensure that the extension programmer need only consider the business logic relevant to his current tasks. While at the same time making it easy to get access to extensions in all relevant workspaces without having to modify these.

In the remaining part of this book, it is most often irrelevant whether some particular explanation has to do with a *new container* or an *extension to* an existing container. In such cases, we shall refer to the part that is implemented as a *container contribution*.

When implementing a container contribution, the Extension Framework will expect you to provide a class that implements the interface `MiContainerEvents`. It is, however, quite difficult and complex to implement this interface from scratch. And it is not expected that you will do so. Doing so requires a lot of boiler-plate code and there will be a high risk of making errors. For this reason, the Extension Framework provides two abstract classes that implement this interface:

- `McAbstractRootContainer`: this class must be used when you create a new container.
- `McAbstractExtendedContainer`: this class must be used when you extend an existing container.

Using either abstract class will enforce that you use data-models to implement the actual behavior of your container. Your actual container class will therefore typically consist of a few lines of code. The abstract implementation will take care of all the boiler plate, calling your data-models in the right way when needed, and will produce result values in the right format.

In this chapter, we shall primarily be concerned with data-models. The overall concept behind container and data-models (as well as container-level events and data-model events) are very similar. The data-models are, however, simpler and easier to work with. But there are a few things that you need to know about the container class.

4.1.1 Binding data-models to the container

So, we claim that a container implementation based on either `McAbstractRootContainer` or `McAbstractExtendedContainer` will typically consist of only a few lines of code. This is so because the abstract classes already implement *all events*. So how can a generic abstract implementation know what you want to do? The answer is that it can't. It can, however, do so by proxy: These abstract classes will depend on that the business logic is implemented by a data-model.

In fact, the two classes have *one method* that isn't implemented. Hence, it is up to the extension programmer to implement that method. The purpose of this method is define

which data-model to use for which pane. This method is called `defineConfiguration`. Notice, that you don't have to specify data-models for all panes: just the ones where you have something to contribute. Depending of whether you create or extend a container, the return type is slightly different. However, to produce values of the required return type, you should in both cases use the factory methods found in the types: `McContainerConfiguration.McRoot` or `McContainerConfiguration.McExtended`.

The factory methods are:

Method	Remarks
`filter`	Defines that the container contains a filter pane and declares the data model for this pane. The result object allows you to specify data models for other panes.
`card`	Defines that the container contains a card pane and declares the data model for this pane. The result object allows you to specify data models for the table, but not for the filter. In case you need a filter, you should start with the `filter` method.
`table`	Defines that the container contains a table pane and declares the data model for this pane. It is not possible to specify data-models for other panes.
`withCard`	Following a `filter` method, this declares the data model for the card. The result object allows you to specify data models for the table, but not for the filter.
`withTable`	Following a `filter`, `card` or `withCard` method, this declares the data model for the table. It is not possible to specify data-models for other panes.
`filterWithCard`	This is a short-hand for `filter(dm).withCard(dm)`. Hence, you can use this when you have the *same* data-model for the filter and for the card.

Method	Remarks
`filterWithCardWithTable`	This is a short hand for `filter(dm).withCard(dm).withTable(dm)`. This convenience method is only possible for extensions, since it is very rare that a *new* three-pane container has the same data-model for all panes. However, for certain type of extensions, it might be common. For example, the prepackaged extension that enables export to spread-sheets, merely adds the same extension data-model to all panes in a container.

Remember that in general, a container comprises n panes, where $n > 0$. In the current version, the Extension Framework only has built-in support for creating panes comprising one *filter pane* and/or one *card pane* and/or one *table pane*. You will recognize that this is very similar to the configuration of Maconomy containers. The Extension Framework will, however enable you to make a table-only container or a filter/table-container. These configurations are never offered by Maconomy containers. When containers are tied together using workspaces, the possible configuration gives you the possibility to model virtually anything. The cases where you would really *need* other pane-configurations in *one* container may exist, but we don't consider them frequent. In such cases, the Extension Framework currently does not aid you. In fact, we state that such configurations are not supported by the Extension Framework.

4.2 Specifying the capabilities of a container

When the outside world (such as a client application) communicates with the coupling service/container API, it is necessary to know what the capabilities of a given container are. The capabilities include information for each pane like:

- Which fields are present?
- What are the type of the fields?
- Are the fields open for editing and are they mandatory?
- Which actions are available (standard actions as well as named actions) such as `Create`, `Update`, `Delete`, `SubmitTimeSheet` etc?
- Which fields are searchable and how?
- Which foreign keys are present?

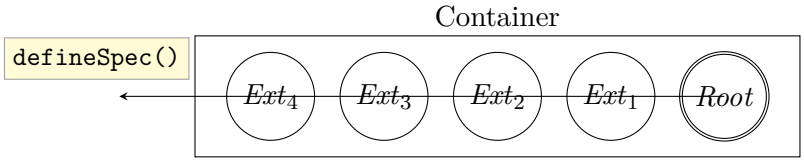


Figure 4.1: The `defineSpec` container-event method is a supportive event. The container must be able to produce a specification declaring its capabilities from nothing. Notice that *each* container contribution must return a fully valid specification.

Each container will declare such things for each pane via a declarative specification XML-dialect called MDSL. We refer to this as the *container specification*. When a container specification is needed, a supportive event will be invoked on the container. This event is called `defineSpec()`. On the container-level it results in a result type called `MiContainerSpec` which encompasses the MDSL content. This means that all container contributions must return a full-fledged and valid MDSL from this event. Figure 4.1 shows this concept. If we take a look at Figure 4.1, the following will happen when the `defineSpec` is called:

1. `defineSpec` will be called on the root, *Root*.
2. `defineSpec` will be called on Ext_1, and the result from *Root* can be accessed. Ext_1 produces a common result for *Root* and Ext_1 combined.
3. `defineSpec` will be called on Ext_2, and the result from Ext_1 can be accessed. Ext_2 produces a common result for the output of Ext_1 and Ext_2 combined. Notice that Ext_2 sees the result from Ext_1 as though it was the first extension on top of a root that had produced a result similar to the one produced by Ext_1.
4. `defineSpec` will be called on Ext_3, and the result from Ext_2 can be accessed. Ext_3 produces a common result for the output of Ext_2 and Ext_3 combined.
5. `defineSpec` will be called on Ext_4, and the result from Ext_3 can be accessed. Ext_4 produces a common result for the output of Ext_3 and Ext_4 combined.
6. Since there are no more container contributions, the final result of invoking `defineSpec` on the container is the result produced by the last contribution, in this case the result of Ext_4.

Notice that each contribution is invoked in a similar way: it will have access to the result of whatever is underneath it. But a contribution cannot and should not be concerned with whether the underlying result is the result of invoking one or many container contributions. Likewise, every container contribution should consider itself the "final" specification, and must therefore output a result which may be considered valid and complete.

Although no container event can occur without the `defineSpec` event having been run at least once, *there is no guarantee* that it will be called immediately before any operation. Because the resulting MDSL specifications may be cached by the coupling service and/or by clients, there is no guarantee that the method will be called more than once. You should therefore never make code that relies on that the `defineSpec` method is called: the `defineSpec` method may only be called *once* and never again.

Although writing MDSL files from scratch can be done relatively straight-forward, there are some cross-references that needs to be taken care of in the right way. It becomes much more complicated when you need to declare additions or deltas to an existing MDSL. When using data-models, the data model has a similar event: `defineDomesticSpec`. This method is used to declare the capabilities for whatever pane is associated with that data model. The term *domestic* refers to that the spec declares what is *new* and therefore introduced by this data model. In other words, a data model need not and should not describe the resulting capabilities of a pane—it should only describe what *its* contributions are. The framework will automatically invoke the data-models when necessary and then produce a resulting MDSL. It is absolutely possible to have a data model that does not alter the capabilities of a container. For example, an extension that wishes to modify the `maconomy:ExpenseSheets` by adding a check that no expense registrations less than 10 EUR are ever registered, does not alter the capabilities of an expense sheet container: it alters the *semantics* of creating and updating using that container, but it *doesn't add* the capability of creating or updating, since the container is already capable of doing that. Thus, you only need to implement this method if you change the capabilities of the container.

Let us take a look at some examples

Listing 4.1: You can add *and* change capabilities using the `defineDomesticSpec` method

```
2   public MiPaneSpec.MiExtended defineDomesticSpec(final MiDefine
        containerRunner) {
3     return McPaneSpec.McExtended.pane()
4       .changeField("Text5").title("Rejection Justification").
            noSearch().then()
5       .addStringVariable(PM_EMAIL_VAR, "Proj. Mgr. E-Mail").then()
6       .changeForeignKey("ProjectManagerNumber_Employee")
7         .supplementLink(PM_EMAIL_VAR, key("ElectronicMailAddress")
              )
8       .end();
9   }
```

In Listing 4.1 three things are done:

1. Some properties of the field `Text5` are *changed*: the default title is changed to "Rejection Justification," and if there are searches related to this field, these will by default not be enabled (although this can be overridden in the layout.) Changing the default title means that any MDML layout element making use of the default title of this field will render "Rejection Justification" rather than, e.g., "Text 5."

2. A new calculated field[1] is *added* to the pane. By doing so, the *internal name* must be defined. The internal name is the name used to refer to that field in layouts. Here, the internal name of the field is determined by some constant defined elsewhere. It is good practice to define constants for such added fields. In addition, a default title must be provided. In this case, the title is "Proj. Mgr. E-Mail."

3. A foreign key is *changed*. Foreign keys are used to bind panes together in workspaces. They may also be used to specify search behaviors. In this case, the foreign key is given an additional *supplement link*. A supplement link is used to enable searches based on supplementary information rather than on formal data keys. In this case, the supplement link will make it possible to search for project managers by entering the e-mail address of a project manager from the (just added) calculated field.

From the code you can see that the structure of the `defineDomesticSpec` method has the following form: each line starts with a declaration indicating a pane-level change. E.g., the *changing* of a field, the *changing* of a foreign key or the *addition* of a field. The resulting object is a builder-style object that allows the programmer to alter the properties of that part. For example, the properties of a field or the properties of a foreign key. Each of these methods again return that same builder object, allowing the programmer to "chain" all relevant property alterations. When everything has been specified for a certain part, the `then` method is invoked. This method will now allow new changes or additions to the pane level. Finally the `end` method declares that there are nothing more to describe, and the return value has a form that can be returned from the `defineDomesticSpec` method. Obviously, you can structure your code in whatever way the compiler allows. It is good practice, however, to keep everything relevant for the same part (field, foreign key, action etc.) on the same line, letting all lines end with the `then()` method. It is also good practice to align the pane-level methods. E.g.,

```
...
  .addField(...).property1().property2().then()
  .addField(...).property1().property2().then()
  .changeField(...).property1().property2().then()
  .addAction(...).property1().property2().then()
  .changeForeignKey(...).property1().property2().then()
  .changeForeignKey(...).property1().property2().then()
  .addForeignKey(...).property1().property2().then()
  .end();
```

In this way, each line represents one pane-level property. Or, if there are many/long property declarations, align with the property methods at the same level:

```
...
  .addField(...).property1().property2().then()
  .addField(...).property1().property2().then()
```

[1] Calculated fields are called "variables" in the extension framework

```
    .changeField(...).property1()
                     .property2()
                     .property3().then()
    .addAction(...).property1().property2().then()
    .changeForeignKey(...).property1().property2().then()
    .changeForeignKey(...).property1()
                          .property2()
                          .property3()
                          .property4().then()
    .addForeignKey(...).property1().property2().then()
    .end();
```

You may not declare changes for the same field twice, even if the properties differ. The exact behavior of the Extension Framework is undefined. In the future, a run-time error may occur!

Listing 4.2: Changing several capabilities

```
2   public MiPaneSpec.MiExtended defineDomesticSpec(final MiDefine
        containerRunner) {
3     return McPaneSpec.McExtended.pane()
4       // Add fields and variables
5       .addStringField(MY_FIELD1, "My Manager No.").open().
            mandatory().multiLine().then()
6       .addBooleanField(MY_FIELD2, "My Checkmark").open().
            autoSubmit().then()
7       .addDateField(MY_FIELD3, "My Date").open(MeOpenness.
            OPEN_UDPATE).then()
8       .addAmountField(MY_FIELD4, "My Amount").open().then()
9       .addPopupField(MY_FIELD5, "My Currency", key("CurrencyType")
            ).open().then()
10      .addLimitedStringField(MY_FIELD6, "My Longer Text", 4096).
            open().then()
11      .addLimitedStringField(MY_FIELD7, "My shorter Text", 8).open
            ().then()
12      .addStringVariable(MY_VAR, "My Manager Name").then()
13
14      // Add foreign keys
15      .addForeignKey(MY_FOREIGN_KEY, "Manager", McContainerName.
            create("Find_Employees")).link(MY_FIELD1, key("
            EmployeeNumber")).supplementLink(MY_VAR, key("Name1")).
            then()
16
17      // Add and remove actions
18      .addAction(MY_ACTION, "My Action").icon("myActionIcon").then
            ()
19      .removeStandardAction(MeAction.DELETE)
20      .end();
21  }
```


Listing 4.2 shows another example implementation of the `defineDomesticSpec` method. Here, a number of fields of different types are added, a single variable (calculated field) is added, a foreign key is added, and a named action is added. At the same time, the deletion capability is removed from the pane.

So, how exactly can the domestic specs of the data-models be defined? The relevant pane-level factory can be produced by addressing a factory method in the class `McPaneSpec`. Or rather, if your data-model is intended for a *root container*, you must use one of the factory method in `McPaneSpec.McRoot`, otherwise, you must use the factory method in `McPaneSpec.McExtended` (see Listing 4.2.)

Method	Remarks
`pane`	This method is present for extended containers and for root containers. In the case of a root container, you must specify a few properties of that pane. For container extensions, there is nothing to specify.
`autoPositionPane`	This method is present for root containers only. It defines that the pane content should be treated as *auto-position* content. Auto-position content may also be known as "automatic line numbering." Panes declared in this way must specify which field indicates the position ("line number.") The Extension Framework will then automatically maintain the line numbers. For example, when a line is deleted, then all following lines should have their position (or line number) decreased. When a line is inserted, the lines following the inserted line should have their position increased. Similarly, when moving lines up and down in a table, the positions of the lines should be maintained. These position numbers will be maintained as integers numbered starting from 1. Line-numbering is common in the Maconomy application. For example, the order of time sheet lines and expense sheet lines is the one determined by the end-user. In the database, this ordering is managed by such a position field. It makes little sense to number *all* time sheet lines starting from 1. Indeed, time sheet lines belonging to the same time sheet should be organized relative to each other only. This means that each time sheet may have a "line 1." In order to specify the scope of the ordering, the extension programmer must also specify an auto-position *context*. For time sheets lines, the context would be the fields pointing to the time sheet header. You declare which fields are considered auto-position context fields by invoking the method `autoPosContext` on the fields which make up the auto-position context.

Method	Remarks
`treePane`	This method is present for root containers only. It defines that the pane content may be organized in a tree-structure. Tree-panes are by definition also auto-position panes. In addition to the auto-position properties, a tree-structured pane must specify how data is tree-structured. Since all panes are data homogeneous (i.e., only records of one type may be shown in a pane) there must be some way of indicating whether a given record in a pane is considered a child record of some other record in that pane. To do this, you indicate which foreign key is used to point from a record to its parent record.

Each of the *pane*-type methods return a pane builder object. For pane builders you can do the following:

Method	Remarks
`add`*`T`*`ypeField`	Specifies the addition of a field with the type matching the exact method name. Fields, by definition, are expected to be *persisted* (e.g., permanently stored) somewhere, typically in the database. For example, `addStringField` adds a field of type `String`, and `addAmountField` adds a field of type `Amount`. The possible types are: `Amount`, `Boolean`, `Integer`, `Real`, `String`, `Date`, `Time`, `Popup` and `TimeDuration`. The `TimeDuration` is basically a Real, but it is declared that this field should be considered a duration of time. Client applications may use this to write, e.g., `0:30` (30 minutes) rather than `0.5` ($\frac{1}{2}$ hour.) For the `String` type, there are variant methods that specify the maximum content length of the String. If the length is left out, the default will be assumed (see below.) Setting the maximum length ensures that the clients will restrict the number of characters[2] that can be entered in the field. A value of 8 will mean that no more than 8 characters can be entered. Likewise, there is a variant where the "type" is `UnlimitedString`. This can be used to declare Strings with no content limit. Obviously, *you* have the responsibility to ensure that data can be properly persisted.

[2]Actually the number of bytes. Using some characters such as Cyrillic, Chinese or other unicode characters take up more than one byte. The use of unicode characters is not possible in all versions of Deltek Maconomy.

Method	Remarks
`addField`	This method is similar to the typed variants above, except that the type is given as an argument. The use of this method is generally discouraged: you should use the explicit method names where possible, because it increases the clarity of the code. In some cases, however, you could have very generic code that must take the concrete type as an argument.
`add`*`T`*`ypeVariable`	This method is similar to adding a field, except that a variable, by definition, is considered a *calculated field* and therefore should *not* be persisted. Also, the Extension Framework enforces variables to be read-only.
`addVariable`	This method is similar to the typed variants above, except that the type is given as an argument. The use of this method is generally discouraged: you should use the explicit method names where possible, because it increases the clarity of the code. In some cases, however, you could have very generic code that must take the concrete type as an argument.
`addAction`	`MeAction`-typed argument. This method declares the addition of one of the "standard" CRUD-actions: Insert/Add, Create, Refresh, Update, Delete, Print This, Move (Up/Down/Indent/Outdent.) There are certain restrictions that will be enforced: for example, it is not possible to have an "Insert" without a corresponding "Create," and the framework will automatically manage some of these things for you. You are referred to the JavaDoc for details on this. The JavaDoc is accessible from within the Maconomy Extender.
`addAction`	`MiKey`/`String` arguments. This method declares the addition of a *named* action. Named actions are all actions that are not one of the above CRUD-actions. As an example, the Maconomy application contains a lot of named actions, for example the "SubmitTimeSheet" and "ApproveTimeSheet" actions. The name must be unique. The internal name is the name used to reference this action in MDML layouts.
`addPrintAction`	This "action" is not really an action that is invoked on the container. Instead, it specifies a container that will be launched when a "Print..." action is invoked. Many containers from the Maconomy application have such Print-selection containers associated. Using this method, you can add such behavior on your own. Beware that this "Print..." differs from the "Print (This)" action, which *will* be invoked on this container.

Method	Remarks
`addForeignKey`	This method declares the addition of a foreign key. A foreign key is a reference from fields in a record in this pane to data somewhere else. Typically such foreign keys are used to construct workspaces. A foreign key pointing to data identified by the key fields (K_1, ..., K_n) must specify which fields in this pane refers to each of those key fields. For example: $$\begin{array}{lcl} F_1 & \rightarrow & K_1 \\ F_2 & \rightarrow & K_2 \\ & \vdots & \\ F_n & \rightarrow & K_n \end{array}$$ specifies that the field F_1 in this pane references the key field K_1, the field F_2 in this pane references the key field K_2 etc. We refer to this specification as the *link*. In addition to the link, a foreign key may specify *supplement links*. A supplement link defines fields or variables that contain supplementary information that relates to the data of the link. For example, an employee name or a task description. We write this by reversing the direction of the arrow: $$\begin{array}{lcl} S_1 & \leftarrow & D_1 \\ S_2 & \leftarrow & D_2 \\ & \vdots & \\ S_n & \leftarrow & D_n \end{array}$$ Which specifies that the supplement field S_n in this pane will obtain the value kept in the descriptive field D_n in the data being referred to. Supplement fields will allow the end-user to search by entering text in them—when a search entry is selected, the actual keys will be transferred into the foreign-key fields. See Section 4.2.3 for a more thorough description on foreign keys.

Method	Remarks
`addSearchKey`	This method declares the addition of a *search key*. A search key is in many ways like a foreign key. But it isn't a foreign key, and cannot be used to bind panes together in workspaces. It will allow searching from the involved fields, though. A search key may occur in cases where it isn't possible to define a mapping to *all* the key fields of the data being referred. For example, the key of the database entity called `LocalSpec1` is `(LocalSpec1Name, LocalSpec1List)`. But in many cases, if you know the value of a Local Spec1 Name, you only know the associated company, not the `LocalSpec1List` used for that company. In such cases, a foreign key cannot be defined. If you want to allow the user to search for local spec 1 names, you need to define a search key. You can dynamically let your code apply relevant search restrictions when searches do take place. See Section 4.2.3 for a more thorough description on search keys.
`changeField`	This method is only available for extensions, not for roots. It declares changes to properties of a field. This could be things like the default title, the openness properties or changes to whether this field is considered mandatory. You may also use this method to change which foreign key/search key to use for searching from within this field.
`changeVariable`	This method is only available for extensions, not for roots. It declares changes to properties of a variables. Notice that all fields from the Maconomy application will be exposed as "fields", event though they may not be stored in the database.
`changeAction`	This method is only available for extensions, not for roots. It declares changes of properties of an action. Depending on the argument type, this may be the name of a named action, or a enum referencing a standard CRUD-action. Using this method, you can change the default title or the default icon of an action.
`changePrintAction`	This method is only available for extensions, not for roots. It is used to change the properties of the "Print. . . "-action (not to be confused with the "Print (This)" action.)
`changeForeignKey`	This method is only available for extensions, not for roots. It is used to change the properties of a foreign key. Using this field, you can add supplement links (see description of the `addForeignKey` method above), or change the default title.

Method	Remarks
`changeSearchKey`	This method is only available for extensions, not for roots. It is similar to the `changeForeignKey` method, except that it refers to search keys.
`removeAction`	This method is only available for extensions, not for roots. It may be used to remove an action from a container. For example, if you wish to remove the action `SubmitTimeSheet-Temporarily` action from the `maconomy:TimeSheets` container, you can remove it from the container by using this method. It means that the container will not externally describe that this method exists. Consequently, it will not be shown in the client, and cannot be referred by MDML layouts.
`removeStandardAction`	This method is only available for extensions, not for roots. It may be used to remove a standard action from a container. For example, if you wish to remove the possibility to delete certain entries, you can remove the standard action `Delete`.
~~`supportLocking`~~	This method may be used for root containers to determine that locking is supported. Notice that there is no framework handling of locking events. You will have to implement locking support yourself. If locking is supported, lock events will occur when a user starts to edit a record, and an unlock event will occur when the user reverts or submits the changes. NB! As of Maconomy version 2.3, locking events no longer occur and this method is consequently deprecated.

In addition to the above methods, there are a few control methods that control the behavior of the builder:

Method	Remarks
`getDefaultStringMaxLength`	This method returns the current default max string length associated with the pane spec builder. The default string max length is used by the `addStringField`, and `addStringVariable` methods.
`setDefaultStringMaxLength`	This method sets the default string max length of this builder. Hence, all subsequent calls to `addStringField` or `addStringVariable` will have this max length unless an explicit max length is provided. By default, the default string max length is 255.

Method	Remarks
`setDefaultStringUnlimited`	This method is similar to `setDefaultStringMaxLength` except that it will treat the default length as "unlimited."
`setLenient`	This method puts the pane spec builder in "lenient" mode. For example, if you add or change an action and then also remove it, the builder will complain with a run-time error if it is not put in lenient mode. It is highly recommended that the builder is kept in non-lenient mode. By default, pane spec builders are non-lenient.

4.2.1 Field and Variable Properties

As hinted above, you can add or change field properties by one of the `add`*`Type`*`Field` or `-Variable` or one of the `change`*`Type`*`Field` or `-Variable` methods. The result is a field/variable property builder, and the subsequent method calls are used to specify properties of the specified field.

For the `add`-methods, you must specify a mandatory *name* (which will be the internal id of the field or variable.) In layouts, this is the name to use when you want to refer to this name. When referring to the field or variable in the business logic, you must likewise refer to this name. In addition to the name, you must specify the default title of the field if you *add* it.

The list of methods of the field/variable builders are

Method	Remarks
`key`	This method is appropriate for *root panes*. The method must be invoked on any field which is a key field. For example, an Employee would have a field, `EmployeeNumber` and declare this as a key, indicating that this field can be used as a key. You must specify one or more key fields when creating a root pane: `.addField(EMPL_NO, "Empl. No").key()`
`autoPosContext`	This method is appropriate for auto-positionable (or tree-structured) root panes. This method must be applied to all fields comprising the auto-position context. See Page 53 for more information on auto-positionable panes.

Method	Remarks
`title`	This method is available for changes. When adding fields or variable, the title is specified as a mandatory argument. This method is used to specify the default title. The default title is the title which is used in layouts where no explicit title is used, for example `<Field source="SomeField" />` This layout element will render a label for the field `SomeField` using the specified default title for that field. Changing the title of a field may be handy in cases where you want to use an existing remark or text field with a specific semantic. In such cases, the default title should be changed.
`open`	This method can be used to open a field. It is not available for variables which are always closed. There are two variants of this method: one that takes no arguments, and one taking an `MeOpenness` enum. The first declares the field as open for editing when the pane is both *init* and *exists* state. A pane is in *init*-state when an initialize-event has occurred, presenting a template record which the user can edit before finally creating it. When a pane shows an existing record, it is in *exists*-state. For field additions, fields are considered closed if this method is not invoked. Fields can be closed in layouts, but cannot be opened if declared as closed.
`mandatory`	This method is used to declare that a field is considered mandatory. Declaring a field as mandatory means that the client-side will validate that the field contains a non-blank value in states where the field is open for editing. Hence, declaring a closed field as mandatory has no effect. When changing a field, there are two variants of this method: one with no parameters (meaning that mandatory is set) and one with a boolean indicating the whether to consider the field as mandatory. Notice that you should be extremely careful if you remove the mandatoryness state of a field: in such case your code *must ensure* that the field is always filled out by your extension! Mandatoryness can be enabled in layouts, but it cannot be removed by layouts.

Method	Remarks
`autoSubmit`	This method is used to declare that a field by default should be "auto-submittable." If a field is auto-submittable, it means that when the user is "done" editing the field, the pane in which the field resides will automatically be submitted for update. Although this works for any field type, it is usually used with "single-click" field editing such as dates, booleans and pop-ups. For field changes, there are two variants of this method: one with no arguments (which enables autoSubmit) and one that takes a boolean argument indicating whether or not autoSubmit should be set. This setting is merely a default setting. It can be overruled by layouts.
`hidden`	This method specifies the field as "hidden." A hidden field cannot be shown in a layout. Also, it is not possible for end-users to add such a field using the "Customize Columns..."-option. It *is* possible to use the value of this field in expressions. Hidden fields are very rare, and should be used judiciously.
`filterable`	This method specifies whether this field can be filtered/sorted if it occurs in a filter pane. If this mehtod is not invoked, fields will be considered non-filterable.
`multiline`	This method is applicable for fields/variables of type `String`. It is used to indicate that the field may contain new-line characters. By default, fields are not allowed to contain new-line characters.
`autoSearch`	This method defines the way searching is applied to a field that is searchable. Invoking this method implies that • Searching will take place as the user types. • A small magnifying glass will be rendered inside the field, allowing the user to invoke a search window by activating it. • Ctrl+G-search can be invoked from this field. auto-Search is the default way of searching, so if nothing else has been specified, the behavior will be auto-search.
`onDemandSearch`	This method defines the way searching is applied to a field that is searchable. The search-behavior implied by invoking this method is similar to `autoSearch`, except that a drop-down list is shown. Ctrl+G search and search-as-you-type also works in this mode.
`noSearch`	This method disables search for a field. If this method is invoked, the field will, by default, not show any visible signs of searching capabilities, even if the field is in principle searchable.

Method	Remarks
`searchBehavior`	This method takes a single enum-argument defining the search behavior of this field. This method may be used instead of either `autoSearch`, `onDemandSearch` or `noSearch` in cases where it is more convenient to calculate the search-capabilities rather than invoking one of these methods, depending on some logic.
`foreignKeyOrder`	This method specifies the order of foreign keys of a field. The order[3] of foreign keys is important for searchable fields: the first foreign key is the one used for searching. A list may be applicable in cases where the foreign key is dynamic, i.e., only enabled depending on the value of other fields. It is not necessary to specify all foreign keys for the field: any foreign key not mentioned will be inserted in the list after the ones explicitly specified.
`properties`	This method takes a set of enum-properties encompassing the following properties: mandatoryness, hidden, multi-line, auto-submit and filterable. By invoking this method, you can specify a set of properties for this field in one method call. It may be convenient in situations where the capabilities are based on non-simple logic.

4.2.2 Action Properties

As hinted above, you can add or change action properties by one of the `addAction` or `changeAction` methods. The result is an action property builder, and the subsequent method calls are used to specify properties of the specified action.

For the `add`-methods, you must specify a mandatory *name* (which will be the internal id of the field or variable.) In layouts, this is the name to use when you want to refer to this name. When referring to the action in the business logic, you must likewise refer to this name. In addition to the name, you must specify the default title of the action if you *add* it.

The list of methods of the action builders are:

[3] i.e., prioritization order

Method	Remarks

`title`	This method can be used when *changing* an action. It is used to specify the default title of the action. The default title is used in layouts if no other title has been explicitly specified. As an example, suppose you wish to change the title of the action `ConvertToOrder` in the container `maconomy:Jobs` into "Set as Order." You can do this by changing the default title, thereby avoiding changing the layouts:
	`changeAction(CONV_ORDER)`
	`.title("Set as Order")`

`icon`	This method is used to specify the default-icon of an action. When adding an action, there is no default-icon specified. This will imply that the workspace client uses whatever icon is used as a default default icon. The icon can in all cases be overridden in a layout.

`availableWhen`	This method is used to specify if an action will be included in layouts specifying that "all" action should be shown, using:
	`<Actions all="true">`
	`...`
	`</Actions>`
	
	The availability is specified by an enumeration argument:
	`ALWAYS` meaning that the action will be included in the layout when the `all` attribute is `true`.
	`REFERRED` meaning that the action *will not automatically* be included in the layout, even when the `all` attribute is `true`. With this setting, the action must *explicitly* be included. For example, the `ExportDataSet` ("Export-to-Excel") action is added to card-panes in this way, whereas it is added to filter/table panes using the `ALWAYS` option. The `REFERRED` option is particularly useful for actions that *must* be parameterized.
	`DEFAULT` This will apply the default availability configured for the system. At present this is `ALWAYS`.

In addition to adding or changing actions, you can also entirely remove an action that would otherwise be available. For example, suppose that a given customer installation does not wish to use the `SubmitTimeSheetTemporarily`-action in the `maconomy:TimeRegistration` container. Of course, the action can be removed from the layouts. But it is a bit cumbersome to do this, if this is the *only* change, and you *never* want to offer

this functionality. In this case, you can remove the action by:

```
removeAction("SubmitTimeSheetTemporarily")
```

Similarly, you can remove support for the "standard" CRUD actions. For example, if you want to entirely remove the possibility of deleting jobs, you can invoke the following method:

```
removeStandardAction(MiPaneSpec.MeAction.DELETE)
```

4.2.3 Foreign-Key and Search Properties

Foreign keys are a very central concept for the Maconomy Workspace Client: the foreign keys are used to specify how data is tied together in a workspace. The general rule is that the content of a pane is always determined from its parent pane. Let us have an example. Suppose you have a workspace showing a list of Jobs. The panes below this list will be able to show data related to whatever job has focus. So, how do we know what is related? The answer is: from the foreign keys specified in the List-of-Jobs pane. Each declared foreign key specifies a mapping from one or more fields in *this* pane into some other data. As the next pane, you can show panes from any container that has such data as its key. In the filter pane of the `maconomy:Jobs` container, there are 70+ different foreign keys, enabling you to show a large amount of related data in subsequent panes. Some of these are:

`primary` The foreign key `primary` always implicitly exists in a pane. You should not declare it. It represents the identity mapping, in this case mapping from the job to itself, enabling you to show panes using the selected job as the key.

`SalesPersonNumber_Employee` This foreign key maps to the employee specified in the field `SalesPersonNumber`

`Team1Number_Team` This foreign key maps to the team specified by the `Team1Number` field.

`CustomerNumber_Customer` This foreign key maps to the customer specified in the `CustomerNumber` field.

`CompanyCustomer` This foreign key maps to the company-customer specified by the fields: `CustomerNumber` and `CompanyNumber`.

From the above examples, we can see that a field can be used in several foreign keys. For example, the field `CustomerNumber` is used in the foreign keys `CustomerNumber_Customer` and `CompanyCustomer`. We can also see that, depending on the data, more than one field must be used to properly identify a key in the foreign data.

Apart from being used to specify relationships between data, foreign keys may also be used to *search* for data. In the above example, the field `SalesPersonNumber` must represent an employee. Therefore, if this field is open for editing, the end-user should get

help to enter a valid employee number. The foreign key `SalesPersonNumber_Employee` is used for this purpose: based on the specification of this foreign key, the client can launch a search-pane searching for employees when the user activates searching.

Sometimes a record in a pane does not have enough information to reference a certain set of foreign data. And still, you want to aid the user by offering searching for specific kinds of data. In this case, you can use what is called a *search key*. A search key is quite similar to a foreign key, only it cannot be used to bind panes together in a workspace because the specification is not a full valid foreign key. A prominent example of where a search key is used is in the table-part of the `maconomy:TimeSheets` container. In a time sheet line, there is a field, `TaskName`, which is supposed to indicate a specific task name in a task list. So, obviously, you want to aid the user by being able to search from within this field. However, the data hosting a task (or technically a *task-list line*) has *two key fields*: the name of the task *list* that it belongs to, and the name of a task in that list. The time sheet line, however, has no reference to the task *list*. Instead, the task list is indirectly specified by the associated job. For this reason, it is impossible to declare a foreign key on a time-sheet line that references the task. Instead, a *search key* is declared, specifying that the only valid results must be found as a task name of some task-list line. When the user eventually invokes the search, a search condition is being applied based on the value of the job specified; in this way, it is possible to restrict the tasks displayed in the search result such that only tasks related to the specified job are shown.

Foreign-Key Basics

So, how does the system know which search-container to invoke when searching takes place? This must be specified on the foreign key/search key! Let us have a look at how a search key is specified in final MDSL (XML) format:

```
1 <ForeignKey name="SalesPersonNumber_Employee"
2             title="Sales Person"
3             source="maconomy:Find_Employee">
4   <Field ref="SalesPersonNumber" foreignKeyField="EmployeeNumber"/>
5 </ForeignKey>
```

This is interpreted in the following way:

`name` The `name`-attribute denotes the internal name of this foreign key. This is the name to use when binding panes together in a workspace.

`title` The `title`-attribute indicates the title of the foreign key. This title is the default-title used by `<Reference>`-elements in MDML.

`source` The `source`-attribute specifies the name of the container that is invoked when a search is initiated. In this case it is the `maconomy:Find_Employee` container. The pane used is the *filter* pane of the specified container! The `maconomy:Find_Employee` container is a filter-only container that show lists of employees. By default, this

container applies no conditions to which employees can be shown. It is absolutely possible to specify a *different* container than the `Find_`-container. However, you need a container that contains all the key fields being referenced by the foreign key. As an example, you could instead use the filter-pane of the `maconomy:Employees` container. In this case, the foreign-key definition would look like:

```
<ForeignKey name="SalesPersonNumber_Employee"
            title="Sales Person"
            source="maconomy:Employees">
  <Field ref="SalesPersonNumber" foreignKeyField="EmployeeNumber"/>
</ForeignKey>
```

Maconomy offers a `Find_`-container for all database tables in the Maconomy-database. These find containers have no intrinsic where-clause attached. This is in contrast to the filter-panes of the "normal" card containers: these may have intrinsic where-clauses. Sometimes, you may benefit from such intrinsic where-clauses. For example, if you want to have a field that points to an invoiceable job and you want to enable searching, you can declare a foreign key that uses the `maconomy:InvoiceSelection` as the `source`: in this way, the search window will only show jobs that can be displayed by the `maconomy:InvoiceSelection`-container.

`ref` The `ref`-attribute is found for all nested `<Field>`-tags. It indicates a field in *this* pane that maps to a specific key field in the foreign data.

`foreignKeyField` The `foreignKeyField`-attribute is found for all nested `<Field>`-tags. It is used to specify which key field in the *foreign data* is being referred by a field in *this* pane.

The embedded `<Field>`-tags are called the "links" because they specify how data in this pane is related to or linked to data somewhere else.

Supplement links

In addition to link-fields, foreign keys and search keys can have optional *supplement links*. A supplement link is a de facto relationship between data in this pane and the foreign pane. Usually, it is used to declare derived information that depends on the specified foreign key. For example, in the card pane of the `maconomy:Jobs`-container, the following foreign key may be defined:

```
<ForeignKey name="ProjectManagerNumber_Employee"
            title="Project Manager"
            source="Find_Employee">
   <Field ref="ProjectManagerNumber" foreignKeyField="EmployeeNumber"/>
   <SupplementField ref="ProjectManagerNameVar" foreignField="Name1"/>
</ForeignKey>
```

Compared to the previous example, the new thing is the presence of the `<SupplementField>`-tag. The attributes are the same as for the `<Field>`-tag, but the meaning is slightly

different. It indicates that the value of the field/variable `ProjectManagerNameVar` can be expected to contain the same value as the field `Name1` of the foreign data that is being referred by the `<Field>`-tags. This kind of specification will enable searching from the supplement field (here `ProjectManagerNameVar`) *even if it is closed for editing*. When searching is made from this field, it will attempt to match the *name* rather than the *number*. When a value is selected from a search-result, the corresponding *key value* (in this case `EmployeeNumber`) will be transferred into the corresponding foreign-key field (here `ProjectManagerNumber`) regardless of whether that field is shown in the layout or not. The fact that it is possible to search (and apparently edit) such supplement fields that are usually specified as being closed for editing has no real impact: the workspace client will treat these fields as closed, and the updated value of closed fields will *not* be communicated to the server-side. The supplement fields are there to give better feedback to the end-user and to allow the end-user to search from within these de facto related fields.

Conditional Foreign Keys

Sometimes a foreign key/search key is *conditional*. This means that—depending on the value of some other field—the foreign key should be considered enabled or disabled. If a foreign key is disabled, data will *not* be distributed through it in a workspace. Also, searching will not take place using this foreign key.

Conditional foreign keys are used in several places in the Maconomy application. A prominent place is in the `maconomy:GeneralJournal` container's table pane. In that pane, there is a field called `AccountNumber`. Depending on the value of the field `TypeOfEntry`, this field references *either* an account, *or* a customer, *or* a vendor. In the MDSL, this is represented in the following way:

```
<ForeignKeySwitch field="TypeOfEntry">
   <Case value="GRPType'\G\ ">
      <ForeignKey name="AccountNumber_Account"
                  title="Account"
                  source="Find_Account">
         <Field ref="AccountNumber"
                foreignKeyField="AccountNumber"/>
      </ForeignKey>
   </Case>
   <Case value="GRPType'\R\ ">
      <ForeignKey name="AccountNumber_Customer"
                  title="Customer"
                  source="Find_Customer">
         <Field ref="AccountNumber"
                foreignKeyField="CustomerNumber"/>
      </ForeignKey>
   </Case>
   <Case value="GRPType'\P\ ">
      <ForeignKey name="AccountNumber_Vendor"
                  title="Vendor"
```

```
21                    source="Find_Vendor">
22            <Field ref="AccountNumber"
23                   foreignKeyField="VendorNumber"/>
24        </ForeignKey>
25    </Case>
26 </ForeignKeySwitch>
```

This meaning of this specification is: depending on the value of the field `TypeOfEntry` (called the *switch field*,) the following conditional foreign keys are enabled/disabled:

`AccountNumber_Account` this foreign key is enabled if the value of `TypeOfEntry` is `GRPType'G`. The foreign key references an account. Hence, if searching is invoked in a situation where `TypeOfEntry` has this value, the `maconomy:Find_Account` will be invoked. Also, a workspace pane bound using this foreign key will be used in this case. If the `TypeOfEntry` has any other value, workspace panes following this foreign key will be empty.

`AccountNumber_Customer` this foreign key is enabled if the value of `TypeOfEntry` is `GRPType'R`. The foreign key references a customer. Hence, if searching is invoked in a situation where `TypeOfEntry` has this value, the `maconomy:Find_Customer` will be invoked. Also, a workspace pane bound using this foreign key will be used in this case. If the `TypeOfEntry` has any other value, workspace panes following this foreign key will be empty.

`AccountNumber_Vendor` this foreign key is enabled if the value of `TypeOfEntry` is `GRPType'P`. The foreign key references a vendor. Hence, if searching is invoked in a situation where `TypeOfEntry` has this value, the `maconomy:Find_Vendor` will be invoked. Also, a workspace pane bound using this foreign key will be used in this case. If the `TypeOfEntry` has any other value, workspace panes following this foreign key will be empty.

The switch fields must be of some enum (popup) type.

Method Overview

In order to make contributions to the set of foreign keys and search keys, you must use one of the following methods: `addForeignKey`, `changeForeignKey`, `addSearchKey` or `changeSearchKey`. The result of these methods is a foreign-key property builder, and the subsequent method calls are used to specify properties of the specified foreign- or search key.

Method	Remarks
`addForeignKey`	*Arguments:* • `name` • `title` • `searchContainerName` Using this method, you add a foreign key. If a search based on this foreign key is invoked, the *filter*-pane named `filter` of this container will be used.
`addForeignKey`	*Arguments:* • `name` • `title` • `searchContainerPaneName` Using this method, you add a foreign key. If a search based on this foreign key is invoked, the specified container and pane name will be used. This method should be used in cases where the filter pane is different from the standard name.
`addForeignKey`	*Arguments:* • `name` • `title` Using this method, you add a foreign key. But in contrast to the above methods, *it will not be possible to search using this foreign key.* Hence, searching can only be enabled for the referenced fields if these fields are part of another foreign- or search key.
`addForeignKeyBy-Copying`	*Arguments:* • `name` • `foreignOrSearchKeyToCopy` This method adds a new foreign key which, initially, has exactly the same content as an already existing foreign key. This can be useful in cases where you want to augment an existing foreign key, while keeping the original foreign key untouched. For example, you could use this to change a search-key into a full-fledged foreign key by adding the missing link-field specifications. The original search key will always be treated as a search key, and cannot be used to bind panes together in a workspace.
`addSearchKey`	*Arguments:* • `name` • `title` • `searchContainerName` Using this method, you add a search key. If a search based on this search key is invoked, the *filter*-pane named `filter` of the specified search-container will be used.

Method	Remarks
`addSearchKey`	*Arguments:* • `name` • `title` • `searchContainerPaneName` Using this method, you add a search key. If a search based on this search key is invoked, the specified container and pane name will be used. This method should be used in cases where the filter pane is different from the standard name.
`addSearchKeyBy-Copying`	*Arguments:* • `name` • `foreignOrSearchKeyToCopy` This method is similar to `addForeignKeyByCopying`, except that it introduces a new search key rather than a foreign key.
`changeForeignKey`	This method is used to alter the properties of an foreign key. For example, you can use this to change the default title of the foreign key, or to add supplement links. Notice that it is not supported to *add* a foreign key and then—for the same pane, at the same level—also *change* the key. The change refers to changing a foreign key that has been introduced by a container contribution closer to the root!
`changeSearchKey`	This method is similar to `changeForeignKey`, except that it is used to modify an existing search key rather than a foreign key.

The above methods result in a foreign key/search key property builder. Using these builders, you can alter the properties of the foreign key/search key in question. The list of methods of the foreign key builders are:

Method	Remarks
`title`	This method is available when changing or copying. The method can be used to change the default title of a foreign key/search key. The default title is used by the MDML `<Reference>`-element.

Method	Remarks

`link`	The link method is used to specify a link (foreign key field) mapping from a field in *this* pane to a *key field in some foreign data*. The `link` method takes one or two arguments. The one-argument version declares a link that maps the field in *this* pane into a key field *having the same name*. For example, a job would typically have a `CustomerNumber` field referencing a customer. And the key field of a customer would typically be named in the same way. By using this method, the code can be kept shorter and easier to read.
	`link("CustomerNumber")`
	The two-argument version can be used to map a field in *this* pane into a key field in some foreign data which is named differently. For example, a `ProjectManagerNumber` field would typically be linked to `EmployeeNumber`.
	`link("ProjectManagerNumber", "EmployeeNumber")`

`supplementLink`	This method is similar to the `link` method above, except that it declares supplement links. As it is expected to be very rare that the fields are called the same in this case, there is only the two-argument version of this method: `supplementLink("ProjectManagerNameVar", "Name1")`
`searchContainer`	This method is available when changing or copying foreign keys and search keys. The method is used to change the associated search container.
`searchContainerPane`	This method is available when changing or copying foreign keys and search keys. The method is used to change the associated search container as well as the pane used by the search. This can be used in cases where the used pane is not called "filter."
`enabledBy`	This method is used to specify a *conditional foreign key or search key*. When this method is invoked, the current foreign key/search key will be considered conditional. It will be considered enabled when the specified switch field has the corresponding specified value. An example use is `enabledBy("TypeOfEntry", McPopup.val(key("GRPType"), "G");`

4.2.4 Using Name Spaces

Suppose your contribution adds a variable to container. When doing so, you must define a name. This name is used to reference the variable in other contexts (either through the layout or programmatically.) Naturally, in order to avoid ambiguity, the name must be unique. If you specify a name that already exists, the framework will issue an error at run-time.

So, eventually, you find a good name[4] that doesn't clash with already existing fields or variables. Now you may think everything is good, but it isn't! Your extension may not be *forward compatible*. This means that your extension may not work when a future version of Maconomy is eventually installed.

Let us have a look at an example. Suppose your client wants you to add a variable that calculates the utilization % for time sheets. So, you add a variable to the `maconomy:TimeSheets` container called `UtilizationPctVar`. Your code works wonderfully, and the customer is happy. Although "utilization" tends to be calculated using slightly different rules at different customers, having some kind of calculated utilization shown for time sheets seems like a generic feature. For this reason, it is far from unthinkable that in a future version, the core Maconomy application will add a variable showing the utilization for a time sheet (calculated using some more or less generically applicable rules.) In this case, it is very thinkable that the name of such a variable would be `UtilizationPctVar`. This means that when your client upgrades to that version of Maconomy, *your* extension will break! In fact, as long as it is installed, time sheet functionality is completely broken!

In order to reduce the risk of this, you could start to invent strange naming conventions, like `__myCustomer__UtilizationPctVar`. The thing is just that such naming conventions tend to be unstructured[5] and they will only *reduce* the risk of name clashing. The risk will not be eliminated.

For this reason, the extension framework supports the concept of *name spaces* for fields, variables, actions, foreign keys and search keys. Hence, basically everything that you can *add* to a container. By using name spaces, your extension is *guaranteed* to be forward compatible with future versions of the core Maconomy application. The core Maconomy application always uses the "implicit" name-space `maconomy`. This means that when referencing such names programmatically, you must leave out that implicit name-space. From layouts, you may optionally specify the implicit name space.

In order to declare the name space you want for your extension, you must implement the data-model method `defineNameSpace`. Every name that is *added* to the container (using the methods described above to declare the capabilities returned by `defineDomesticSpec`,) will automatically prepend the name space followed by a colon

[4] Finding a good name can be surprisingly difficult, but is worth the effort! A variable with a misleading or unclear name can cause a lot of confusion over time. It can even lead to programming errors.

[5] Two different programmers choose two different ways of doing it

(the name-space separator) to the names. If the names explicitly contain the name space, the name space will not be duplicated. And if some other name space is specified, an error will be issued at run-time.

In addition to declaring your things like fields and variables, you frequently need to reference these name in your code. For example looking up a value or assigning a value. When doing such referenced, you *must* prepend the name space. Otherwise, the framework has no chance of knowing whether you refer to the name with or without a name space. One way to ensure this is to declare all the referenced names as constants in your class. This, however, requires that you explicitly state the name space in each of these declarations. Declaring this as `static` constants implies that the initialization happens once, the very first time the data model is ever referenced.

As an alternative (which one is right for your is largely a matter of taste) is to declare the "constants" as non-`static` but `final` member variables that are initialized upon object construction. To do this, you can use the utility method `ns` that is available in all data-models. This method automatically prepends the name space returned by `defineNameSpace` to a name that is not already having a name space. The only down-side is that the initialization needs to be done every time a data model object is constructed (which happens for each individual event.) Alternatively, you can declare the names *without* name space, and remember to apply the `ns` method every time you reference that name in your code. We recommend to either declare naming constants including the name space, or declaring the names as member variables that are initialized using the `ns` method.

Listing 4.3: Using a Name Space

```
2
3     private static final MiKey NAME_SPACE = key("Trifolium");
4     private static final MiKey UTILIZATION_PCT =
5         NAME_SPACE.concat(":UtilizationPctVar");
6
7     // Notice the absence of the "static" key word,
8     // and the use of ns().
9     private final MiKey UTILIZATION_FACTOR =
10        ns("UtilizationFactorVar");
11
12    /** {@inheritDoc} */
13    @Override
14    public MiKey defineNamespace() {
15      return NAME_SPACE;
16    }
17
18    /** {@inheritDoc} */
19    @Override
20    public MiExtended defineDomesticSpec(final MiDefine
          containerRunner) throws Exception {
21      return McPaneSpec.McExtended.pane()
```

```
22          // UTILIZATION_PCT already contains the name space.
23          // The framework will check that it is the same as
24          // the one returned by defineNameSpace()
25          .addRealVariable(UTILIZATION_PCT,
26                           "Utilization %").then()
27
28          .addRealVariable(UTILIZATION_FACTOR,
29                           "Utilization Factor").then()
30
31          // Here, the name space is explicitly stated.
32          // The framework will check that it's the right one
33          .addAction(key("Trifolium:UpdateFromPlan"),
34                     "Update from Plan").then()
35
36          // References to core application names
37          // must not have a name space
38          .changeAction(key("SubmitTimeSheet"))
39              .icon(key("MySubmitIcon")).then()
40
41          // You can add new actions that are named the same as core
42          // application action (except for the name space)
43          // If the added name does not already contain a name space,
44          // the current name space will automatically be added
45          // (in this case Trifolium:SubmitTimeSheet)
46          .addAction(key("SubmitTimeSheet"),
47                     "My Custom Submit Time Sheet").then()
48
49          .end();
50      }
```

Listing 4.3 shows how name spaces are declared. The method `defineNameSpace` is implemented in line 14. By implementing this method, you declare that you want to add a name space to added capabilities (added fields, variables, actions, foreign keys and search keys.) It's implementation is quite simple, just returning the constant defined in line 3.

The definition of the added and changed capabilities is handled by implementing the method `defineDomesticSpec` in line 20. In this example, the domestic specification adds one variable and two actions. Furthermore, it changes the icon property of the an action defined by the core Maconomy application. It is worth noticing that:

- In line 25 a variable is added. The name is given by the constant `UTILIZATION_PCT` declared in line 4. So, it is declared that the added variable has the name `Trifolium:UtilizationPctVar`. Since this name contains a name-space prefix (`Trifolium:`), the Extension Framework will check that it is the same as the name space specified by the method `defineNameSpace`. If this was not the case, an error would be issued at run-time: you are not allowed to add several name spaces in the same data-model!

- In line 28 another variable is added; the `Trifolium:UtilizationFactorVar`. Again its name is defined using a reference to a *de-facto constant*[6]. This de-facto constant, defined in line 9, isn't (and cannot be) declared `static` due to the use of the `ns` method.

- In line 33 a new named action is declared. This time, the name is in-lined (i.e., not specified by a constant.) Since the declared name already contains a name-space prefix, it is checked that it matches the name space declared by `defineNameSpace`.

- In line 38 we *change* an action, rather than adding it. In this case, we change the icon on the action `SubmitTimeSheet` which is a core Maconomy application action. Therefore, no name space is specified, as this action has the implicit Maconomy name space.

- In line 46 we add yet another action. This time, the action added is seemingly just called `SubmitTimeSheet`. But since it is *added*, the Extension Framework will automatically prepend the current name space. This means that the name of the new action is `Trifolium:SubmitTimeSheet`.

In your code, when you reference the added (name-spaced) names, you *must* prepend the name space. Otherwise, the framework has no chance of knowing whether you refer to the name with or without a name space. One way to ensure this is to declare all the referenced names as constants in your class (just as it is done in line 4) in Listing 4.3.) or as de-facto constants using the `ns` method (just as it is done in line 9 in Listing 4.3.

When you reference the name-spaced names in layouts, you must also include the name space in the reference. For example:

```
<Group title="Key Figures">
  <Field source="Trifolium:UtilizationPctVar" />
  <Field source="Trifolium:UtilizationFactorVar" />
  <Field source="InvoiceablePercentageOfWeekVar" />
</Group>
```

This layout snippet shows a group "Key Figures" that shows three variables: the first tow are `Trifolium:UtilizationPctVar` and `Trifolium:UtilizationFactorVar` (which are added by our extension,) and the third is the field `InvoiceablePercentageOfWeekVar` which is contributed by the standard Maconomy application.

For backwards compatibility reasons, it is technically possible *not* to use name spaces for field etc. *However, we strongly encourage that you* always *use name spaces for every extension you do.*

[6] It is technically not a constant because it must be re-initialized each time a data model is initialized. A `static` constant will only be initialized *once* no matter how many object instances are created.

4.3 Implementing Data-Carrying Events

In this section, we shall have a closer look at the data-carrying events. Although these events are different, they have many things in common from an implementation point of view.

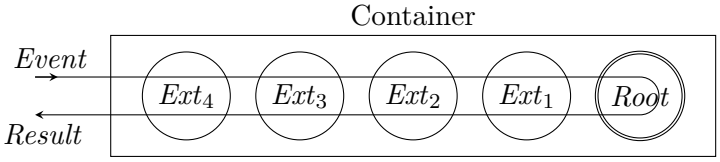


Figure 4.2: The data-carrying events all have a similar life-cycle in a container. This figure shows the life cycle of some event, *Event*. The result of executing that event is whatever result comes out of the outer-most container contribution.

Figure 4.2 shows the general life-cycle of data-carrying events. When having a chain of container contributions, we shall refer to the root contribution as "last", "bottom" or "inner-most", and the contribution at the opposite end of the chain as the "first", "top" or "outer-most." In cases where there is only the root, the root will obviously be both first and last.

1. The first container contribution will be invoked. In the example, the first contribution is the Ext_4. This is done by invoking a method called `on`*`Event`*`Pre`[7]. Hence, this method is executed prior to the next contribution in the chain.

2. The next container contribution will be invoked. In the example, this is Ext_3. Again, it is the `on`*`Event`*`Pre` that is invoked.

3. This goes on until only the root is left. The root container contribution will also have the method `on`*`Event`*`Pre` invoked, and then immediately after, a method called `on`*`Event`*`Post`. The "Pre" and "Post" can be though of as prior to nothing and after nothing. The root container contribution is now completely done, and will not be invoked again. As far as it is concerned, the event operation is finished. The "Post" method will get an input corresponding to an empty result value.

4. After the post-method has been executed in the root contribution, the method `on`*`Event`*`Post` will be executed on the contribution immediately preceding the root, in this case the Ext_1 extension. The "Post" method will receive as input whatever value was produced by the root contribution. When the result is returned from Ext_1, it is completely done and will not be invoked again. As far as Ext_1 is concerned, the event operation is finished.

[7] For each specific event, the *Event* will be substituted with a term specific to that event, e.g., "Update" or "Delete." Hence, `onUpdatePre` or `onDeletePre` respectively.

5. The `on`*`Event`*`Post` will be invoked on the immediately following execution, in this case Ext_2. The "Post" method will receive whatever input was produced by the "Post" method of the following contribution, i.e., the result produced by Ext_1. Once done, Ext_2 is completely done and will not be invoked again. As far as Ext_2 is concerned, the event operation is finished.

6. This goes on until the first container contribution is reached. The `on`*`Event`*`Post` method will be invoked on that contribution, in this case in Ext_4. This method will receive as input whatever result was produced by the "Post" method of the following contribution. When this is done, this container contribution is completely done. As there are no preceding contributions, *the container is completely done*, and the resulting value for the entire container is whatever value is returned from the post-method of the first container contribution.

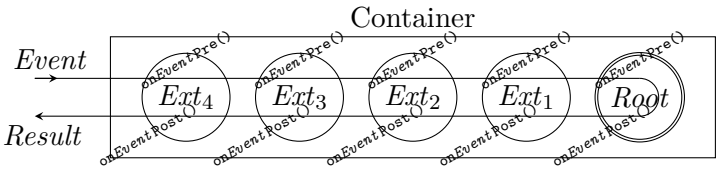


Figure 4.3: This shows the order in which the "Pre" and "Post" methods are called. The order is defined by the direction of the Event/Result arrow going through all container contributions!

Figure 4.3 shows the order in which the `on`*`Event`*`Pre` and `on`*`Event`*`Post` methods are called.

Because of this life-cycle, it means that—seen from a particular container contribution—the behavior is always exactly the same, no matter the number of container contributions, and no matter if one or more of them happen to be removed or re-ordered: *every container-contribution must produce a fully valid return value that is a candidate for the final result of the container*. Figure 4.4 illustrates this.

Hence, each individual container contribution must act as though it is responsible for producing the final result of a container event. *Maybe* this result will be altered by extensions on top, but that is of no concern for the specific container contribution. All it should be concerned with is performing *it's own logic* in the `on`*`Event`*`Pre` and `on`*`Event`*`Post` methods. Figure 4.5 illustrates this.

4.3.1 Working with Data-Models

As mentioned earlier, implementing functionality directly on the container level is somewhat difficult. As always, you should make use of data models instead. Once a data model has been bound to a container, the default container implementation will automatically invoke an event script on a given data model when applicable.

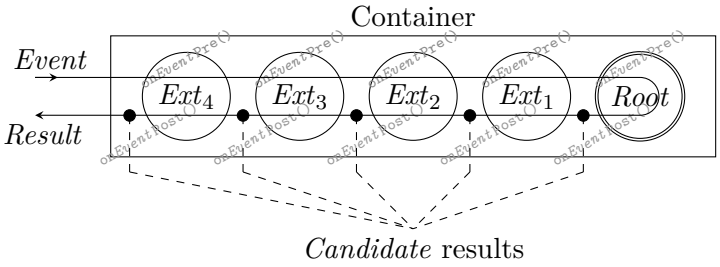


Figure 4.4: Each container contribution must produce a fully functioning result value from the `onEventPost` methods. Hence, the result of a given `onEventPost` method is a *candidate* result. The last produced candidate result will be the final result for the container. In this way, each container contribution need not worry about whether or not there are other contributions and they cannot (and certainly *should not*) depend on whether there are preceding contributions!

For *root data models* (i.e., a data model used with a root container), the `onEventPre` and `onEventPost` scripts are basically two sides of the same thing: the "Pre"-script is immediately followed by the "Post" script. In order to simplify the data model, there is just *one* method to implement for each script. This script is called `onEvent`.

For *extension data models* (i.e., a data model used with a non-root container), the "Pre" and "Post" scripts are both meaningful since they can do something either *before* the remaining contributions are invoked, or *after* the remaining contributions have been invoked. Hence, for extension data models, two scripts may be implemented: `onEventPre` and `onEventPost`.

The method signatures for `onEvent`, `onEventPre` and `onEventPost` are similar:

```
void onEvent(MiContainerRunner.MiEventPost containerRunner,
             MiEventData.MiEvent           eventData)

void onEventPre(MiContainerRunner.MiEventPre containerRunner,
                MiEventData.MiEvent          eventData)

void onEventPost(MiContainerRunner.MiEventPost containerRunner,
                 MiEventData.MiEvent           eventData)
```

The first thing to notice is that these methods do not return any values. The container-level "Post"-event methods, however, must produce a valid value for this event. How is that possible? It is important to understand that *Data-models are record-centric and containers are container-value-centric*. This means that the data-models are only concerned with the actual event record. Nothing else. The underlying container implementations provided by the framework will automatically gather information via the data-models and at the end it will tie this information together in the right way and provide a container value

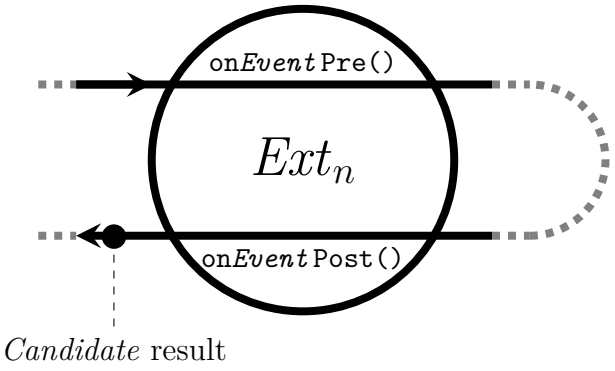


Figure 4.5: Each container contribution must be able to generate a fully valid result value for some event, *Event*. A specific contribution cannot and should not know or be concerned with whether there are contributions on top of it. Also, it should not be concerned with whether there is one or more extensions between itself and the root. The root can rely on that it is last, and any non-root contributions can rely on that there is at least the root extension after it.

comprising a pane value for the necessary panes. This is exactly one of the facts that makes data-models so much easier to work with:

1. A data-model event-method need *only* consider the event record in question. Everything else can be left to the framework.
2. In case of multiple record panes, the framework will sort out which record is being used, so all information is provided to the data-model. No work needs to be done.
3. Since the data-models are not supposed to produce neither pane values nor container values, all the nasty details related to this can be forgotten and left to the framework.

So, if the methods do not produce a result value, how is it possible to contribute? Such contributions must be made though the `eventData` parameter which is passed on the any event-method. The event data may comprise several things, including:

- The "original data", i.e., the data that the event is being invoked on. Hence, the data that was seen by the end-user prior to executing the event. This can be obtained through the method `getOriginalData`.
- The "user data." User-data is only used for "Create" and "Update" events. It represents an edited record. For example, if the user changes a field `CustomerNumber` from "A" to "B", then the user-data will know that the field `CustomerNumber` was changed, and the it's new (but not yet committed) value is "B." The original data (see above) will present the `CustomerNumber` value as "A." The user-data can be obtained by using the `getUserData` method.
- The "result data." The result data contains the final result values of the event-record.

So, if you wish to change the value of a field (introduced by *your* contribution) as a result of running some event, then you can do this by simply modifying the result-data object. The result-data can be obtained by using the method `getResultData`.

Event/method	`getOriginalData`	`getUserData`	`getResultData`
`Read`	✗	✗	✓
`Initialize`	✗	✗	✓
`Create`	✓	✓	✓
`Update`	✓	✓	✓
`Delete`	✓	✗	✗
`Action`	✓	✗	✓
`Print`	✓	✗	✓
`Move`	✓	✗	✓
~~`Lock`~~	✓	✗	✓
~~`Unlock`~~	✓	✗	✓

Table 4.10: An overview of which events give access to what kind of data in the `eventData` object.

It may be a surprise that none of the above methods of the event-data object are available for all events. Table 4.10 shows which methods are available in which event context. For example, notice that upon `Delete`-events, there is no access to the result data. This is because a delete event *removes* a record. For that reason, it makes no sense to specify values on that record as a result of deleting it. Similarly, the `Initialize` and `Read` events have no original data. This is because these events are supposed to *produce* a record by reading a database, or by "inventing" a template record. This must be possible also in cases where there *were* no "original" records in the first place. And finally, only the `Create` and `Update` methods have access to the user-data.

It may come as a surprise that, e.g., actions do not have access to user-data. For example, using a wizard, the user can edit a record and immediately execute an action. However, under the hood this is implemented as a sequence of events: an `Update`-event followed by an `Action` event. So, when the `Action` event occurs, there *is* no user-data. Only the "original data" which, in turn, corresponds to the result of the preceding `Update` event.

Working with Original Data

As outlined in Table 4.10, it is possible to obtain information about the record on which some event is executed. The original data is exposed through a *read-only* interface, since

it makes no sense to claim that the original data is something other than it was. Since the Extension Framework is quite generic, it is not possible to expose the information using specific types for each particular record. Instead, record-like values are exposed using a generic interface that resembles a key/value look-up map. The basic interface is called `MiValueInspector`. Through this, you can get access to values of particular fields. Since different fields have values of different types, *values* are represented using classes of subtype `McDataValue`. Such raw data-values are, however, often a little cumbersome to work with. If you *know* that the value of a given field is a `String`, and you just want to obtain that String, then going via an abstract `McDataValue` is tedious. For this reason, the `MiValueInspector` interface has methods that makes it much easier to obtain values in a number of ways:

- It is possible to obtain the value of a specific field as a Java type. For example, if you know that the type of the field `CustomerNumber` is `String`, then you can obtain this as a Java String by

  ```
  originalData.getStr("FieldName")
  ```

  For fields of type `Popup`, you must choose whether you want to access the *ordinal value* or the *literal value* by selecting either `getPopupOrdinal` or `getPopup` respectively.

- It is possible to obtain the value of a specific field as a Maconomy-encoded data value of a specific type. For example, a `McStringDataValue` or a `McPopupDataValue`. These data values are sub-types of the generic `McDataValue` type. It is sometimes useful to extract the values as typed data values. Either because they may be used by framework functions, or because that representation gives easy access to a number of properties such as popup literal name, popup ordinal value and popup title value. Especially for popup-values, the `McPopupDataValue` provides a way to encompass all aspects of a popup value in one place. E.g., the ordinal value, the literal value, the title and the type name. Examples are:

  ```
  originalData.getStrVal("StringFieldName")
  originalData.getPopupVal("PopupFieldName")
  ```

- It is possible to obtain the value of a specific field as a generic Maconomy-encoded data value. You cannot directly access the underlying semantic value from that generic type. It is sometimes useful for framework functions, or if you don't know/care about the specific field type.

  ```
  originalData.getVal("FieldName")
  ```

Listing 4.4 shows examples of how values are extracted from original data and how the values can be further used by the business logic. The various methods for extracting values of different types are:

Method	Remarks
`getAmount`	Returns the value of a field of type `Amount`. If the field does not have this type or if the field does not exist, an exception is thrown. The returned value is the Java `BigDecimal`. Although the `BigDecimal` is somewhat more cumbersome to use that, e.g., `float` or `double`, it is a deliberate decision to use this type: with the binary floating point types `float` and `double`, imprecision will occur, and there is no controlled way to handle it. For `BigDecimal` you need to be absolutely aware of what should happen, for instance in case of infinite decimal expansion (such as $\frac{1}{3} = 0.33333333\ldots$), i.e., how rounding should be applied. Such issues are very important when dealing with monetary units and should be dealt with cautiously. You are referred to the API documentation for `BigDecimal`: http://docs.oracle.com/javase/7/docs/api/java/math/BigDecimal.html.
`getBool`	Returns the value of a field of type `Boolean`. If the field does not have this type or if the field does not exist, an exception is thrown. The returned value is represented as a Java native `boolean`.
`getDate`	Returns the value of a field of type `Date`. If the field does not have this type or if the field does not exist, an exception is thrown. The returned value is represented as a Java `GregorianCalendar`. Although a `GregorianCalendar` abstracts both a date and a time of the day, the returned value is only significant with respect to the *date* portion. If needed, you can add time information to the returned value.
`getInt`	Returns the value of a field of type `Integer`. If the field does not have this type or if the field does not exist, an exception is thrown. The returned value is represented as a Java native `int`.

Method	Remarks
`getPopup`	Returns the value of a field of type `Popup`. If the field does not have this type or if the field does not exist, an exception is thrown. The returned value represents the *literal value* of the popup value. Since literal values are not case sensitive, the returned value is represented as a `MiKey` value. Note that the literal value is not the same as the title. The ordinal value will never be localized!
`getPopupOrdinal`	Returns the value of a field of type `Popup`. If the field does not have this type or if the field does not exist, an exception is thrown. The returned value represents the *ordinal value* of the popup value. The ordinal value is returned as a Java native `int`.
`getReal`	Returns the value of a field of type `Real`. If the field does not have this type or if the field does not exist, an exception is thrown. The returned value is the Java `BigDecimal`. Please see the description related to `Amounts` above.
`getStr`	Returns the value of a field of type `String`. If the field does not have this type or if the field does not exist, an exception is thrown. The returned value is the Java `String` type.
`getTime`	Returns the value of a field of type `Time`. If the field does not have this type or if the field does not exist, an exception is thrown. As for `Date`, the return type for this method is a Java `GregorianCalendar`. Although `GregorianCalendar` values comprise a date as well as a time of the day, the value returned from this method is only significant for the *time part*. If needed, you can add date information to the returned value.
`get`*`Type`*`Val`	Similar to `get`*`Type`* except that the returned value is of the appropriate `McDataValue` sub-type, e.g., `McIntegerDataValue` or `McAmountDataValue`. Hence, there is one method for each type, e.g., `getPopupVal` and `getStrVal` etc.

Method	Remarks
`get`*`Type`*`OrElse`	Similar to the `get`*`Type`* except that if the specified field is not found, a default value of the appropriate type is returned. The default value is passed as an argument to the method. Hence, there is one method for each type, e.g., `getBoolOrElse` and `getStrOrElse` etc.

Listing 4.4: Extracting Values from Original Data

```
2       final MiValueInspector originalData = eventData.
            getOriginalData();
3
4       // Obtaining the value as a built-in Java type
5       // makes it possible to use the value with standard
6       // Java libraries and methods
7       final String customerNumber = originalData.getStr("
            CustomerNumber");
8       if (customerNumber.matches("t[0-9]+")) {
9         containerRunner.call()
10          .error("You are not allowed to use a template customer");
11      }
12
13      // You can also obtain the value as Maconomy-encoded
14      // representation.
15      final McStringDataValue customerNumberSdv = originalData.
            getStrVal("CustomerNumber");
16      //  From there you can retrieve internal representations
17      customerNumberSdv.stringValue();
18
19
20      // It is mostly used with popup-values where the "value" may
21      // have multiple aspects
22      final McPopupDataValue popupVal = originalData.getPopupVal("
            Statistics1");
23      final MiKey literalValue = popupVal.getLiteralValue();
24      final Integer ordinalValue = popupVal.getValue();
25
26      if (literalValue.isLike("Foreign") || ordinalValue == 4) {
27        doSomething();
28      }
29
30      // Finally, it is possible to obtain values as
31      // generic data values
32      final McDataValue val = originalData.getVal("SomeField");
33      if (val.getType().isType(MiDataType.MeType.AMOUNT)) {
```



```
34          doSomethingElse();
35      }
```

Apart from the methods mentioned above, the `MiValueInspector` contains other methods that can be used for various purposes. For example, copying the value inspector and presenting it as a modifiable record type, asking whether a certain field is found in the inspector etc. Some of the most interesting methods are listed below. For further information on the available methods, you are referred to the IDE content help for more details.

Method	Remarks
`copyValues`	This method comes in a number of flavours. • No arguments: the method returns a `MiRecordValues` object where the value is initially a copy of the current value inspector. As the return type is not a value inspector, you can modify the returned value. • An argument which is an iterable of field names: this method works similar to the no-argument version except that only the fields contained by the iterable will be included in the returned record. If the iterable contains a field name that does not exist in the current value inspector, an exception will be thrown. • A comma-separated list of `MiKeys` or `Strings`. This method is identical to the version above, except that the fields are specified directly. • An argument which is a value inspector. This method works similar to the above, except that only the fields defined by the argument value-inspector are taken into account.
`copyValuesIfExist`	This method comes in a number of flavours. They are all equivalent to `copyValues` above, except that it is allowed to specify fields that are not defined by the current value inspector: these fields will be ignored.
`copyValuesOpt`	This method comes in a number of flavors. They are all equivalent to the above methods, except that the return value is an *optional* record value. If one or more of the argument fields are not found in the current value inspector a `McOpt.none` object will be returned. Otherwise the returned value will be defined and contain a value corresponding to `copyValues`.

Method	Remarks
`setAllCopy`	This method takes a value inspector as an argument. The returned record value which is a "copy" (i.e., a new record) can be thought of as the set "union" of the current value inspector and the argument value inspector. The value of a given field will have the value specified the argument value inspector if it is defined in that. Otherwise it will have the value as found in the current value inspector.
`retainAllCopy`	This method takes a value inspector as an argument. The returned record value which is a "copy" (i.e., a new record) can be thought of as the set "intersection" of the current value inspector and the argument value inspector. The returned record will only contain fields names that are found in both the current and the argument value inspector. The values will be the ones defined by the argument value inspector. See also `retainAllValuesCopy` below.
`retainAllValuesCopy`	This method takes a value inspector as an argument. The returned record value which is a "copy" (i.e., a new record) can be thought of as the set "intersection" of the current value inspector and the argument value. The difference between this method and `retainAllCopy` is that `retainAllValuesCopy` takes both field name and value into account. Hence, the returned record will contain only fields that occur with identical values in the current value inspector and the argument value inspector.
`removeAllCopy`	This method takes a value inspector as an argument. The returned record value which is a "copy" (i.e., a new record) can be thought of as the set "complement" of the current value inspector and the argument value. The returned record therefore only contains field names that are defined in the current value inspector, but *not* defined in the argument value inspector.
`removeAllValuesCopy`	This method is similar to `removeAllCopy` except that it takes both field name *and* associated *value* into account. Hence, the returned record will contain all field/values from the current value inspector where a similar field with the same value does not exist in the argument value inspector.

Method	Remarks
`containsAll`	This method takes a value inspector as an argument. It returns `true` if the current value inspector contains all field names defined by the argument value inspector. Hence, this corresponds to asking whether the argument is a sub-set of the current value inspector.
`containsAllValues`	This method is similar to `containsAll` except that both field names and the associated values are taken into account. Hence, the method returns `true` only if all field of the argument value inspector are defined and associated with the same value in the current value inspector.
`contains`	This method asks whether a specified field (name) is defined in the current value inspector. If so, the method returns `true`, otherwise `false`.
`containsValue`	This method asks whether a specified field (name) is defined in the current value inspector with a specified value. If so, the method returns `true`, otherwise `false`.
`equalsTS`	This method takes a value inspector as argument. It returns `true` if the argument is equal to the current value inspector for all fields. Hence, the two value inspectors define exactly the same field names with exactly the same values.

Working with User Data

As outlined in Table 4.10, for events that allows user-changes (`Create` and `Update`), it is possible to obtain information about the values changed by the user. As explained, this information is provided via the `eventData` parameter using the method `getUserData` which returns an interface of type `MiUserData`. At the core, the `MiUserData` interface is just a `MiValueInspector` (see above), with a few additional methods. The user-data object represents the values the user sees when the event is invoked. *The user-data does not* only *represent the fields that were actually changed. You can ask for any field value!*

Suppose that a user sees the following record (for brevity, only three fields are shown):

JobNumber	Description	StartingDate	Remark1
10250001	Importat Job	⟨*blank*⟩	An important job!

Now, the user discovers the spelling error in the description field: It should spell

"Important Job." So, the user changes the value of the field `Description` to this, and while he's at it, also specifies a starting date: `01-09-2013`. Then he submits the record, which will lead to an `Update` event. The event data will contain a user-data part with the following values:

JobNumber	Description	StartingDate	Remark1
10250001	Important Job	01-09-2013	An important job!

Hence, querying about the value of the field `StartDate` will give a date representing: 01-09-2013. This is *different* from the original data which is also provided with the event, as can be seen below:

	JobNumber	Description	StartingDate	Remark1
Original data	10250001	Importat Job	⟨*blank*⟩	An important job!
User-data	10250001	Important Job	01-09-2013	An important job!

Hence, for the values that are unchanged, asking for that value from the user-data will return the same value as asking the original data for the same field value.

Often you need to do some checks or do some logic if a certain field is changed by the user. So how do you know whether or not a field value has been changed? Of course, you can extract the values from the user-data and from the original data and compare them. This is, however, somewhat cumbersome and obfuscates the business logic. Instead, the `MiUserData` type offers a few methods that makes it easier to check things like that.

Method	Remarks
`changed`	Tests whether a specified field is changed by the user, i.e., whether the value in user-data is the same as the one found in original data.
`unchanged`	The opposite of `changed`. I.e., tests whether a specific field is *not* changed.
`getUserChange`	Returns a `MiValueInspector` interface comprising all fields that were changed. If the resulting value inspector is empty, i.e., contains no field values, it means that nothing was changed. Please be aware, that this method is "expensive" performance-wise, and should therefore be used infrequently and judiciously. For instance, it should not be used inside a loop.

In addition to looking at changed (or unchanged) values in the user-data object, *it is*

possible to modify *the values in the user-data object*. What does this mean, and why is it relevant? Modifying the user-data object is the only way to attempt to modify fields belonging to *other* container contributions!

As an example, suppose you want to make an extension to the container `maconomy:Jobs`. The extension you would like to have is this: whenever the `ProjectManagerNumber` field is changed, you wish to update the `LocationName` field of the job so that it matches the `LocationName` value of the employee assigned as project manager. The problem is that the `LocationName` field is not your ownership! You cannot dictate the value of this field. To solve the task, you have two possibilities:

- In the "Pre"-script, you can change the value of the field `LocationName` *on behalf of the user*. Technically, you do this by modifying the user-data values for fields belonging to some following container contribution, e.g., the root.
- In the "Post"-script, you can programmatically do another update based on this container, thereby—acting as an end-user—invoke an update of the field.

If the first option is possible, that is the best choice because it leads to no additional update events, and therefore performs better. The latter approach will result in two update events, and it may even be necessary to enforce a re-read of the container as well. However, since user-data is only available in `Create` and `Update` events, these are the only events offering this possibility. You may wonder why we cannot just change the user-value in the "Post" method. In order to understand this, let us take a look at how container contributions are invoked in an event life-cycle. Figure 4.6 illustrates why

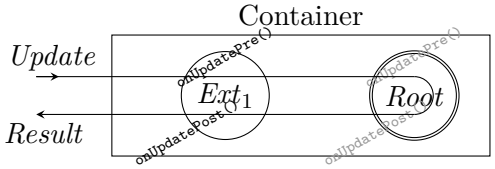


Figure 4.6: The `onUpdatePre` method is invoked before the root container is even made aware of this event. Hence, any modifications to the user-data structure is—seen from the root—the same as if the "real" user had made similar changes. When the `onUpdatePost` event is invoked, the root is done and will not be notified about any changes to the user-data. Which is why is doesn't work to modify user-data in the "Post"-method.

it makes sense to alter the user-data in a "Pre"-script but not in a "Post"-script: the `onUpdatePre` is executed *before* any of the event-scripts in the root. Therefore, modifying user-data in the `onUpdatePre` of the Ext_1 implies that the user-data—as seen from the root—is seen as having the values that resulted from the modifications made to the user-data in Ext_1. The root doesn't know (or care!) whether the user-data was modified by the *actual end-user* or programmatically by some preceding container contribution. It will react identically. Hence, updating the user-data in Ext_1, corresponds to emulating that the user did the same update.

Also, it is very important to understand why it doesn't work to update the user-data in the `onUpdatePost` script in Ext_1. As you can see from the figure, *the root has already been run, and will* not *be invoked again!* It corresponds giving your son an old note reading "Buy salt, please," and then—when he returns with the salt—take the note and change "salt" into "pepper," expecting him to have bought pepper. Just like your son has no chance of figuring out that you really wanted, the root contribution will have no chance of reacting to user-data-modifications done in the "Post"-script.

In order to make it possible to modify the values of the user-data, the `MiUserData` type extends an interface called `MiValueAdmission`. The `MiValueAdmission` interface is an extension of the interface `MiValueInspector`, and therefore allows to to efficiently look up field values of different types. In addition, it allows you to change the values in an efficient way as well, i.e., by allowing you to give values using standard Java types. The section below, dealing with *result-data*, will explain this in more detail.

Listing 4.5 shows an example involving user data. In the example, if a field called `HundredsNumberOf` is changed, we wish to emulate that the user has changed the field `NumberOf` to one hundred times that value. Also, in case either the field `LocationName` or the field `EntityName` is changed, some logic is performed.

Listing 4.5: Checking and Modifying User-Data

```
2   final MiUserData userData = eventData.getUserData();
3
4   if (userData.changed("HundredsNumberOf")) {
5     userData.setReal("NumberOf",
6                      userData.getReal("HundredsNumberOf").
                           multiply(new BigDecimal("100")));
7   }
8
9   if (userData.changed("LocationName")
10      || userData.changed("EntityName")) {
11    doSomething();
12  }
```

Working with Result Data

Above, we have seen how to access the original data and the user-data. However, we haven't yet seen how we can actually affect the *result value*. For this purpose, the `eventData` parameter gives access to the *result data*. The only event that does not give access to the result data is the `Delete`-event, as it makes no sense to update the value of a record that is going to be deleted anyway.

The result-data is made available to the programmer via the `eventData` parameter through the `getResultData` method. By modifying the values of the result data, you communicate to the framework what the field-values of the event record are as an effect

of running the event in question. There are some very important things to notice related to this:

- You must only change the values of fields and variables that are declared in the *domestic specification* of your container contribution! *You have* no business *setting the result values of fields that are not domestic to your extension!*
- The Extension Framework will, if required, automatically persist the values that need to be persisted. The framework will do so when it sees fit. Hence, setting the value of field `myField` to 4 and then setting it to 2 immediately after, will not lead to two updates in the underlying database. When your logic is done, the framework will persist your fields for you! Hence, *you should not manage persisting of values yourself.*
- You are allowed to modify the value of *any domestic field or variable*. It does not matter whether the field is declared as being "open" or not. Since *your* extension contribution has declared the fields or variables, *your* extension contribution has full right (and responsibility) of setting the fields/variables to whatever value you want.

Some default-behavior related to the result-data of the domestic fields is implemented automatically by the Extension Framework:

- By default, the result-data will be set to the original values, if they exist. For `Read` events, the record will be read using the persistence strategy instead. For `Initialize` events, all fields will be blanked using the "empty"/"zero" value for the corresponding type.
- In addition, in case of the presence of user-data (see above), the user-data values will be accepted as result-data.
- If the user-data contains a modification to a field value that has been declared as "closed," an exception will be thrown. Therefore, *you are not allowed to programmatically modify the user-data of closed fields belonging to other contributions!*
- The above will be done prior to your logic being invoked (i.e., the "Pre"-script in case of a container-extension contribution, and the event script in case of a root contribution.)
- For data-models used with root containers, and for data-models extending the `McAbstractPersistingExtendedDataModel` class, the result values for the domestic fields will automatically be persisted or deleted if needed. This will happen after the execution of the "Post" method (or after the execution of the event method for roots.)

This means that if you just wanted to provide an interface to some database fields (applying no logic at all), you don't need to program anything in the data-model, except

defining the domestic spec and the persistence strategy to be used. The rest will happen automatically.

It is very important to understand why you must only make changes to fields that are *domestic* to your contribution! The first thing you must know is that the result-data abstraction is introduced by the data-models. The data-models are introduced at each contribution level, so the result-data structure is *not* passed on between extension contributions. While you may modify the result-data of your own domestic fields and variables in the "Pre"-script, this only means that these will automatically be suggested as result-values by the Extension Framework when the "Post"-script is invoked. So, every time a "Post" event-script is invoked, the Extension Framework will ensure that the domestic values at that level are assigned whatever value they had when the "Pre"-script terminated. Obviously, it is possible to make further modifications to the result data in the "Post"-script. So, why can't you just make changes to fields belonging to following contributions, e.g., the root? As explained in Listing 4.6, the main reason is that once the "Post" method of a container contribution is invoked, the following contributions are completed, and will *not be invoked again*. Hence, they will even have the possibility of persisting the fields. Also, remember that the way to *request* a value change of some field is to emulate a user-edit by modifying user-data. Hence, the result data can and must only be used to set the value of domestic fields! If this was not the case, data-integrity could be severely damaged. Imagine that you introduce two fields, and imagine that it is vital to your logic that these two fields relate to each other in a certain way. Then, if other extensions could just change your fields behind your back, your logic would be ruined.

At present, it is technically possible to assign values to result-data for fields belonging to other contributions. In the future, this may result in a run-time error. In any case, presently, it has no other effect that that the data eventually being presented to the end-user is wrong, and will likely lead to a "data change by another user" error the next time the user tries to do anything. The lesson is: *only modify result-data of fields and variables that are introduced by the current container contribution!*

As with the original data, field values are implemented using sub-types of the generic `McDataValue` method. As already explained, working with these values is often quite tedious. Often, you would much rather work with standard Java types. Since the result-data has the type `MiDataValues`, which is an extension of the general `MiValueAdmission` type, a number of methods are available to make this easily done. All the `get`*`Type`*-methods listed in the orignal-data section are also available on the result-data object.

Method	Remarks
`setAmount`	Sets the value of some field of type `Amount`. If the field does not have this type, an exception is thrown. If the field does not already exist, it is added to the underlying record-value. This method comes in a number of flavours: the value argument can be of a number of different formats: **`BigDecimal`** which should be considered the preferred Java-type for representing amounts. See http://docs.oracle.com/javase/7/docs/api/java/math/BigDecimal.html for more documentation on this type. **`int`** which allows you to conveniently write code such as `resultData.setAmount("CostPrice", 0)` **`double`** which allows you to conveniently write code such as `resultData.setAmount("CostPrice", 100.50)`. For internal calculations you are strongly encouraged to use `BigDecimal`. **`McAmountDataValue`** allowing you to assign the value to values obtained from other framework-related data without having to convert from and to `McAmountDataValues`. For example, `resultData.setAmount("Cost", emp.getAmountVal("Cost"))`
`setBool`	Sets the value of some field of type `Boolean`. If the field does not have this type, an exception is thrown. If the field does not already exist, it is added to the underlying record-value. This method comes in a couple of flavours: the value argument can be of a number of different formats: **`boolean`** which is the natural Java type to use with boolean logic. **`McBooleanDataValue`** allowing you to assign the value to values obtained from other framework-related data without having to convert from and to `McBooleanDataValues`. For example, `resultData.setBool("Blocked", job.getBoolVal("Blocked"))`

Method	Remarks

`setDate`	Sets the value of some field of type `Date`. If the field does not have this type, an exception is thrown. If the field does not already exist, it is added to the underlying record-value. This method comes in a number of flavours: the value argument can be of a number of different formats:
	`GregorianCalendar` which is considered the preferred type for working with date-related information.
	`int, int, int` which makes it possible to conveniently specify a date by providing the year, moth and day, such as: `setDate("EndDate", 2013, 12, 31)`
	`McDateDataValue` allowing you to assign the value to values obtained from other framework-related data without having to convert from and to `McDateDataValue`s. For example,
	`resultData.setDate("StartDate",`
	`job.getDateVal("StartDate"))`

`setInt`	Sets the value of some field of type `Integer`. If the field does not have this type, an exception is thrown. If the field does not already exist, it is added to the underlying record-value. This method comes in a couple of flavours: the value argument can be of a number of different formats:
	`int` which is the natural Java type to use for integer values.
	`McIntegerDataValue` allowing you to assign the value to values obtained from other framework-related data without having to convert from and to `McIntegerDataValue`s. For example,
	`resultData.setInt("Duration",`
	`calLine.getIntVal("Duration"))`

Method	Remarks
`setPopup`	Sets the value of some field of type `Popup`. If the field does not have this type, an exception is thrown. If the field does not already exist, it is added to the underlying record-value. This method comes in a couple of flavours: the value argument can be of a number of different formats: **`McPopupDataValue`** This variant allows you to reassign the value to values obtained from other framework-related data without having to convert. This is the only form to provide all information (e.g., ordinal value as well as literal value) related to a popup value. This should be done with result-data! You can construct such values by using the `McPopup` utility class which provides a number of factory methods for producing values of type `McPopupDataValue`. **`MiKey, MiKey`** This variant allows you to set a popup value of a specified type and a specified literal value. You should *not* use this to assign values to result-data, because the ordinal value is not resolved! You may use this variant to set values in user-data, since the Maconomy root implementation can resolve the ordinal values if missing. **`String, String`** This variant is similar to the variant above, except that it allows you to specify the type name and literal value using `String`s for convenience.

Method	Remarks
`setReal`	Sets the value of some field of type `Real`. If the field does not have this type, an exception is thrown. If the field does not already exist, it is added to the underlying record-value. This method comes in a number of flavours: the value argument can be of a number of different formats: **`BigDecimal`** which should be considered the preferred Java-type for representing reals. See http://docs.oracle.com/javase/7/docs/api/java/math/BigDecimal.html for more documentation on this type. **`int`** which allows you to conveniently write code such as `resultData.setReal("NumberOf", 0)` **`double`** which allows you to conveniently write code such as `resultData.setReal("NumberOf", 100.2)`. For internal calculations you are strongly encouraged to use `BigDecimal`. **`McRealDataValue`** allowing you to assign the value to values obtained from other framework-related data without having to convert from and to `McRealDataValue`s. For example, `resultData.setReal("NumberOf", entry.getRealVal("NumberOf"))`

Method	Remarks

`setStr`	Sets the value of some field of type **`String`**, possibly truncating the input. If the field does not have this type, an exception is thrown. If the field does not already exist, it is added to the underlying record-value.
	This method comes in a number of flavours: the value argument can be of a number of different formats:
	`String` which is the natural Java-type for representing strings. *The string will be truncated to 255 bytes, if it is longer than that!* Hence, the abbreviated form: **`set`***`Str`* (which truncates the term "**`String`**."
	`String, int` which is similar to the variant above, except that the integer argument specifies the number of bytes the value may contain before truncating.
	`McStringDataValue` allowing you to assign the value to values obtained from other framework-related data without having to convert from and to `McStringDataValue`s. For example,
	`resultData.setStr("Name",`
	`emp.getStr("Name1"))`
	The maximum length is that specified by the argument value.

`setString`	Sets the value of some field of type **`String`**, without truncating the input. If the field does not have this type, an exception is thrown. If the field does not already exist, it is added to the underlying record-value. Hence, this method is very similar to **`setStr`** above, except that the non-abbreviated name, **`set`***`String`* indicates that "long strings" are supported, i.e., with to truncation.

Method	Remarks

`setTime`	Sets the value of some field of type `Time`. If the field does not have this type, an exception is thrown. If the field does not already exist, it is added to the underlying record-value. This method comes in a number of flavours: the value argument can be of a number of different formats:
	`GregorianCalendar` which is considered the preferred type for working with time-related information.
	`int, int, int` which makes it possible to conveniently specify a time by providing the hour, minute and second, such as: `setDate("CheckIn", 13, 00, 00)`. Notice that the hour-part is 24-hour based, so 0 is midnight and 12 is noon.
	`McTimeDataValue` allowing you to assign the value to values obtained from other framework-related data without having to convert from and to `McTimeDataValue`s. For example,
	`resultData.setTime("CheckIn",`
	`ts.getTimeVal("TimeCheckedInDay1"))`

`setAll`	This method takes a `MiValueInspector` and for each comprised value, sets the corresponding value in this record. So, if you have built a sub-record with values, you can set these as values in the result data by doing: `resultData.setAll(subRecord)`.
`setVal`	Sets the value of some field to the value of a generic `McDataValue`. This method comes in a couple of flavours: **`MiOpt<McDataValue>`** if the optional data-value is defined, this specified field will get the defined value. If the optional data-value is undefined (i.e., is a `None` value,) this method does nothing and leaves the value of the field unchanged. **`McDataValue`** Changes the value of the field to one provided in the argument. There is no check that the provided value is really a sensible value!
`getValues`	Returns this record as an `MiRecordValues` type. Modifying the elements on the returned value will also modify `this` (the current object.)
`asDataValues`	Returns this record as an `MiDataValues` type. Modifying the elements on the returned value will also modify `this` (the current object.)

Method	Remarks
`asKeyValues`	Returns this record as an `MiKeyValues` type. Modifying the elements on the returned value will also modify `this` (the current object.)
`asUnmodifiable`	Returns this record through an object that will throw a run-time exception if an update of the values is attempted. Any changes to `this` (the current object) will be reflected by the returned object, though.

Listing 4.6 shows an example where result-data is modified. Two things are done: an amount field called `ThirdOfTotal` is set to $\frac{1}{3}$ of the value of the field `Total`. Also, a field called `NumberOfDays` is set to the number of days between two dates represented by the fields `MyStart` and `MyEnd`.

Listing 4.6: Modifying Result Data

```
2   final MiDataValues resultData = eventData.getResultData();
3
4   final MathContext MATH_CONTEXT = new MathContext(12,
        RoundingMode.HALF_UP);
5   resultData.setAmount("ThirdOfTotal",
6                        resultData.getReal("Total").divide(new
                             BigDecimal("3"), MATH_CONTEXT));
7
8   if (!resultData.getDateVal("MyStart").isNull()
9       && !resultData.getDateVal("MyEnd").isNull()) {
10    final int numberOfDays =
11      (int)((resultData.getDate("MyEnd").getTimeInMillis()
12             - resultData.getDate("MyStart").getTimeInMillis())
13            / 86400000L); // mili-seconds per day
14    resultData.setInt("NumberOfDays", numberOfDays);
15  } else {
16    resultData.setInt("NumberOfDays", 0);
17  }
```

4.4 Implementing Initialize Events

In this section, we shall have a closer look at `Initialize` events. The `Initialize` event is a data-carrying event, and consequently has a "Pre"- and a "Post"-event script: `onInitializePre` and `onInitializePost`. When you make a root container and use a data model, the data model offers you the possibility to implement a method called `onInitialize`. In general, the event life-cycle for the `Initialize` event follows the same pattern as shown for general data-carrying events. Figure 4.7 shows this.

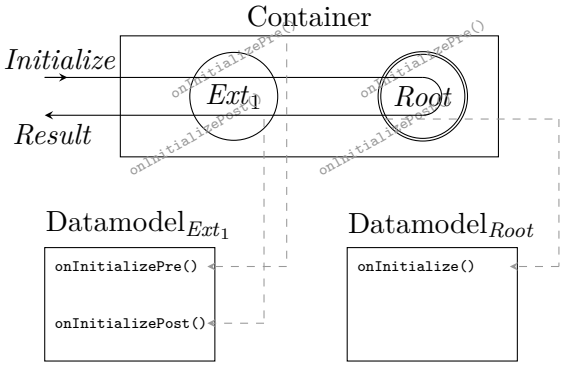


Figure 4.7: The life-cycle for the `Initialize` event implemented using data-models. For a root contribution, the `onInitialize` method is invoked from the container's `onInitializePost` method. For extensions, the `onInitializePre` and `onInitializePost` methods invoke similarly named methods in the data model.

If your container has the ability to create new records, it must be prepared to handle the following two events: `Initialize` and `Create`. The reason for this is that "creating" a new record is a two-step process:

1. The user runs an action which is typically labeled something like "New *Entity*" (for card panes/wizards,) "Insert *Entity*" or "Add *Entity*" (for table panes.) When one of these actions are invoked, a *template* record is presented to the end-user. The presented template-record is the result of the `Initialize` event!

2. The user can optionally edit fields from this record and if the creating of the record is really desired, the "Save *Entity*" action can be run. This event is the `Create` event which will be covered later.

As shown in Table 4.10, the event-data object given to a data model's `onInitializePre`, `onInitializePost` or `onInitialize` methods only provides access to result data. Providing information about original data makes no sense: the record to be created might even be the very first record ever.

Typically, a template record merely presents "empty" or "zero" fields. By default, the Extension Framework will populate all fields with such values. You are free to change one or more of the field values. For example, you may wish to provide the employee number related to the current user, today's date or something similar.

4.4.1 Automatic Management of Line Positions

For root containers, the Extension Framework can be asked to automatically manage the ordering of lines. This feature is known under several names such as "Line-Number Control" and "Auto-Position." Let us start out by looking at an example. Suppose you

want to create a container that can be used to list a number of personal development activities (PDA) for a given employee. This would be modeled by a card/table container displaying employee information in the card and containing PDA entries for that employee in the table part. When entering the PDA's you want the users to add and insert each line just where they want. It means that it should be possible to insert and add lines, and you would like the users to be able to move the lines after creation. Thereby, the order is only known by the user. This kind of container is found many places in the core Maconomy application. Examples include `maconomy:TimeSheets`, `maconomy:JobBudets`, `maconomy:PurchaseOrders` and many more. The way this is achieved in the Maconomy application is to let each record contain a field that denotes the position of that record in its "natural" context. This field is an `Integer`-typed field, by convention called `LineNumber`. When an initialize event happens, meta-data tells whether the line should be inserted before some other line (and if so, which line), or appended at the end. A set of records could look like:

EmpNo	LineNumber	Description	Completed	InstanceKey
11221	1	PRINCE2	false	IK00001
11221	2	Extend Customer Network	false	IK00002
11221	3	Improve Communication Skills	false	IK00003

The field `LineNumber` indicates the preferred display order[8]. Now suppose that the user wants to insert a new line between the current line 1 and 2. The desired outcome, once that record is eventually created is the following:

EmpNo	LineNumber	Description	Completed	InstanceKey
11221	1	PRINCE2	false	IK00001
11221	2	Turning Conflicts into Opportunities	false	IK00004
11221	3	Extend Customer Network	false	IK00002
11221	4	Improve Communication Skills	false	IK00003

Notice that the lines with Instance Keys "IK00002" and "IK00003" have had the line number changed: when the inserted line was inserted in the table, this must happen in order to maintain the ordering defined by the end-user. Correspondingly, if a line is deleted, subsequent lines need to have their line number decreased.

The Extension Framework offers automatic support for this kind of behavior. This is done by letting your table data-model extend one of the abstract classes

[8]The end-user can always change the presentation sorting in the client

- `McAbstractAutoPositionRootDataModel` which enables automatic management of auto-position field (e.g., `LineNumber`.) The actual field to use for this purpose is specified by the domestic spec in the method `defineDomesticSpec`.
- `McAbstractTreeStructuredRootDataModel` which additionally enables tree-structured data and management thereof.

The latter will add the capability of visualizing data in tree-structures. If you do so, the framework will automatically override the value of the field used to denote the "auto-position." In the example above, the `LineNumber` field. This happens *after* the call to the `onInitialize` event method. No renumbering of the auto-position field will take place until the `Create` event is executed. This will all be taken care of by the framework. Consequently you don't need to manage re-numbering the auto-position field manually. So, by extending the proper abstract data-model, the framework will do this for you. *You don't need to initialize the value of the auto-position field in the* ***`onInitialize`*** *method: that value will be overridden by the framework.*

Listing 4.7 shows an example implementation of a root data-model that handles auto-position. The example shows the implementation of the `onInitialize` method as well as the methods `definePersistenceStrategy` (which declares that data should be persisted in a custom table) and `defineDomesticSpec` which defines which fields are present, the auto-position field and the auto-position *context*. The context is necessary in order to maintain the line numbers. Here, the auto-position context is specified as a singleton set: {`EmpNo`}. This implies that the numbering sequence of `LineNumber` will be maintained for all collections having the same value in the field `EmpNo`. It is possible to specify several fields as auto-position context fields.

Listing 4.7: Implementing an Initialize event.

```
2   private final static MiKey ENTITY_NAME = key("TRI_PDATable");
3   private final static MiKey FIELD_LINENUMBER = key("LineNumber");
4   private final static MiKey FIELD_EMP_NO = key("EmpNo");
5   private final static MiKey FIELD_DESCRIPTION = key("Description"
        );
6   private final static MiKey FIELD_COMPLETED = key("Completed");
7   private final static MiKey FIELD_INSTANCEKEY = key("InstanceKey"
        );
8
9   /**
10   * Defines the the fields introduced by this pane.
11   * Also, the pane is declared as an auto-position pane.
12   * The field LineNumber is specified as the auto-position field.
13   * The field EmpNo is specified as the auto-position "context".
14   */
15  @Override
16  public MiRoot defineDomesticSpec(final MiDefine containerRunner)
        {
17    return McPaneSpec.McRoot.autoPositionPane(
```



```
18          text("Personal Development Actions"),
19          ENTITY_NAME,
20          text("Personal Development Action"),
21          // LineNumber is the auto-position field
22          FIELD_LINENUMBER)
23
24        // Employee No. is declared as auto-position context
25        .addStringField(FIELD_EMP_NO, "Employee No.").autoPosContext
              ().open().then()
26        .addIntegerField(FIELD_LINENUMBER, "Line No.").then()
27        .addStringField(FIELD_DESCRIPTION, "Description").open().
              mandatory().then()
28        .addBooleanField(FIELD_COMPLETED, "Completed").open().then()
29        .addStringField(FIELD_INSTANCEKEY, "Instance Key").key()
30        .end();
31    }
32
33    /**
34     * Sets the value of initialized records.
35     * Here we set a template description and a
36     * value for the key field.
37     */
38    @Override
39    public void onInitialize(final MiInitializePost containerRunner,
40                             final MiInitialize eventData) throws
                                 Exception {
41      final MiDataValues resultData = eventData.getResultData();
42      final MiValueInspector cardData = eventData.getContext().
            getPaneData().get().getResultData().get();
43
44      resultData.setVal(FIELD_EMP_NO, cardData.getVal("
            EmployeeNumber"))
45              .setStr(FIELD_DESCRIPTION, "Enter the description")
46              .setStr(FIELD_INSTANCEKEY, getGloballyUniqueKey());
47    }
```

4.5 Implementing Create Events

In this section, we shall have a closer look at `Create` events. The `Create` event is a data-carrying event, and consequently has a "Pre"- and a "Post"-event script: `onCreatePre` and `onCreatePost`. When data-models are being used[9], the logic related to `Create` is implemented by *two* kinds of event-scripts:

- A "change" script, which is meant to contain all logic that is common for `Create` and `Update` events, i.e., all events that involves some kind of user-change. Very

[9] Which you should always do

often, the core logic of these evens are identical. For example, suppose you have a field for which the content should be an employee number. Hence, if it is filled out, it *must* be identical to an existing employee number. This employee number may be entered during creation or during updating. The code validating this, however, is identical for both cases. Likewise, if you have three fields: `Quantity`, `UnitPrice` and `TotalPrice`, the user may enter the value of the these fields during creation or during updating. The code to ensure that they all relate correctly to each other, i.e., that `Quantity` × `UnitPrice` = `TotalPrice`, is identical for both cases.

- A "create" script, which is meant to contain logic pertaining exclusively to the creation step. For example, it might be the case, that upon creating a record, you need to derive information that should not be re-derived for every update. You can implement such logic in this script.

Hence, for root contributions, you can implement two methods: `onChange` and `onCreate`. `onChange` will be invoked by the framework and immediately followed by an invocation of `onCreate`. For extension contributions, the `onCreatePre` of the container will first invoke `onChangePre` immediately followed by an invocation of `onCreatePre` in the data model. Similarly, the `onCreatePost` method of the container will first invoke `onChangePost` immediately followed by `onCreatePost` in the data model.

In general, the event life-cycle for the `Create` event follows the same pattern as shown for general data-carrying events. Figure 4.8 shows this.

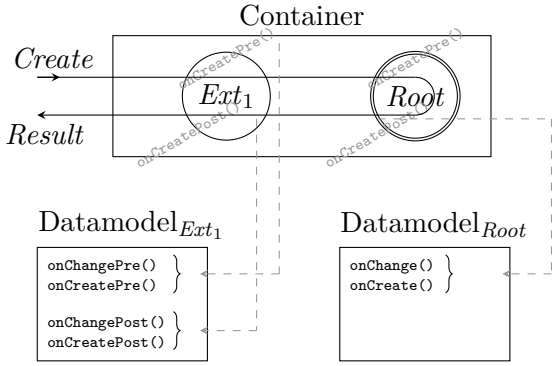


Figure 4.8: The life-cycle for the `Create` event implemented using data-models. For a root contribution, the methods `onChange` followed by `onCreate` method are invoked from the container's `onCreatePost` method. For extensions, the `onCreatePre` invokes the `onChangePre` followed by `onCreatePre` in the data model. `onCreatePost` methods invoke the `onChangePost` followed by `onCreatePost` in the data model.

As pointed out in Section 4.4, the `Create` event is the second phase of the "Creation life cycle." The first phase, the `Initialize` event, provides a template record which the end user may edit before it is eventually persisted. Once the user has edited the record, and the "Save" action is run, a `Create` event takes place. For this reason, the

`Create` event-data gives access to: original data (the template record created by the `Initialize` event), user-data (because the user may have edited some of the fields) and the result-data (because a resulting record needs to be made and the result presented to the end-user.)

Suppose we are building on top of the example given in Listing 4.7. In this example, we implemented an `Initialize` event. The corresponding `Create` event implementation is shown in Listing 4.8.

Obviously, when a record is created, it needs to be persisted. Typically, data is persisted in a database table. It is important to notice, that *the Extension Framework will automatically persist the data to be created. You should* not *do this manually!* The framework knows how to persist the data by invoking the method `definePersistenceStrategy`. By implementing this method, you declare how to persist and fetch data, e.g., by persisting data in a specific database table. In the example given in Listing 4.8, this method declares that data should be persisted in the MOL table `TRI_PDATable` in the Maconomy database. This is all declared in *one* statement (lines 34–35), and that's all you need to do with regards to persistence! The `databaseApi` is an instance variable initialized in the constructor of the data-model. This will be covered later.

Listing 4.8: Implementing a Create event.

```
1
2   /**
3    * Do logic pertaining specifically to "Create".
4    * Check that the Description field was changed by the
5    * user (we don't allow the default template text
6    * provided by the onInitialize script).
7    */
8   @Override
9   public void onCreate(final MiCreatePost containerRunner,
10                       final MiCreate eventData) throws Exception
                            {
11    final MiUserData userData = eventData.getUserData();
12    containerRunner.check(userData.changed(FIELD_DESCRIPTION))
13                   .error("You must specify a description");
14  }
15
16  /**
17   * Here we can place any logic which is common for Create and
18   * Update events.
19   */
20  @Override
21  public void onChange(final MiChange containerRunner,
22                       final MiUserChange eventData) throws
                            Exception {
23    // logic common to Create and Update
24  }
25
```

```
26    /**
27     * Defines the persistence strategy to use with this data-model.
28     * Data is stored in the custom table
29     *   TRI_PDATable
30     * in the Maconomy database.
31     */
32    @Override
33    public MiAutoPositionPersistenceStrategy
          definePersistenceStrategy(final MiContainerRunner.MiDefine
          containerRunner) {
34      return McMolAutoPositionablePersistenceStrategy
35          .create(ENTITY_NAME, FIELD_LINENUMBER, getApiProvider());
36    }
37
```

4.6 Implementing Update Events

In this section, we shall have a closer look at `Update` events. The `Update` event is a data-carrying event, and consequently has a "Pre"- and a "Post"-event script: `onUpdatePre` and `onUpdatePost`. When data-models are being used, the logic related to `Update` is implemented by *two* kinds of event-scripts: a "change" script and an "update" script. This is similar to `Create` events (see Section 4.5.)

Hence, for root contributions, you can implement two methods: `onChange` and `onUpdate`. `onChange` will be invoked by the framework and immediately followed by an invocation of `onUpdate`. For extension contributions, the `onUpdatePre` of the container will first invoke `onChangePre` immediately followed by an invocation of `onUpdatePre` in the data model. Similarly, the `onUpdatePost` method of the container will first invoke `onChangePost` immediately followed by `onUpdatePost` in the data model.

In general, the event life-cycle for the `Update` event follows the same pattern as shown for general data-carrying events. Figure 4.9 shows this.

So, exactly when does an `Update` event occur? It happens when the user edits data that already exists (i.e., not a template record as offered by the `Initialize` event, see Section 4.4.) The usual case would be that the user looks up some record, changes one or more fields, and presses the "Save" button.

However, in some cases, the Update event may occur *implicitly*. This happens in cases where the user edits fields of a record and then does one of the following things:

- Starts typing in some *other* pane. The workspace client will in this case ask the user whether data should be saved or not. If the user opts to save the data, an update event takes place.

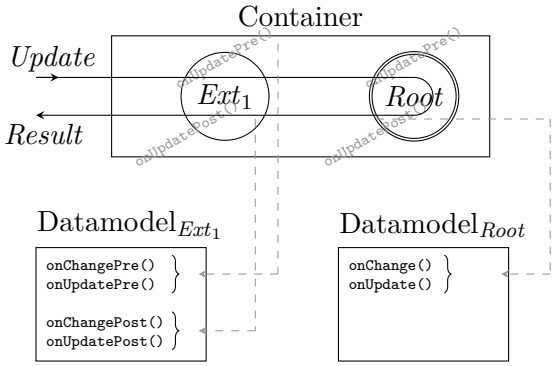


Figure 4.9: The life-cycle for the `Update` event implemented using data-models. For a root contribution, the methods `onChange` followed by `onUpdate` method are invoked from the container's `onUpdatePost` method. For extensions, the `onUpdatePre` invokes the `onChangePre` followed by `onUpdatePre` in the data model. `onUpdatePost` methods invoke the `onChangePost` followed by `onUpdatePost` in the data model.

- Switches to a tab that will imply that the pane being edited is hidden. Again, the workspace client will prompt the user whether the changes should be saved or not.
- Closes the client. Again the workspace client will ask the user whether the data should be saved.
- The user runs an action in the pane being edited. In this case, the workspace client does *not* ask the user whether data should be saved: that will be done implicitly. Once the update has been successfully completed, the selected action will be run *on the result of the* `Update` *event*. Notice that wizards with editable fields will be treated in this way as well[10].

Especially the last case is important to understand: it implies that when an action is run, the action-event *never* has knowledge about the data that was updated "at the same time" (or so the user may believe.) Similarly, the `Update` event will *not* know that an `Action` event will be executed afterwards. Such two events are always completely de-coupled, and the logic around them must work without any assumptions regarding this.

Obviously, when a record is updated, the updated record needs to be persisted, typically in a database table. It is important to notice, that *the Extension Framework will automatically persist the data being updated. You should* not *do this manually!* As explained for `Create` events (see Section 4.5,) the framework knows how to persist the data by invoking the method `definePersistenceStrategy`.

Listing 4.9 shows an example of an extension to the `TimeSheets` container. The extension adds two editable fields

[10]Except for `Create`-wizards which will result in a `Create` event as expected.

Trifolium:Locked which is a boolean field meant to indicate that a specific time sheet line must not be deleted.

Trifolium:DefaultHours which is a real field. When the user changes the value of that field, the entered value is automatically set for all of the days of this time sheet line. Unless the user—at the same time—changed such a value. So, for example, if the user enters the following values

$$\texttt{Trifolium:DefaultHours} = 4, \texttt{NumberDay2} = 1$$

This should correspond to entering "4" for all days except Tuesday (day 2) where the value is "1." Once entered, the **DefaultHours** should be reset, allowing the user to change it again.

The listed code does basically three things:

1. The existence of the new fields is declared. This happens in lines 18–23 in the method `defineDomesticSpec`.
2. It is declared that the new fields should be persisted using a database table called **TRI_TimeSheetLine** by implementing `definePersistenceStrategy`. In this case, the method is a one-liner (line 33.) This allows the framework to automatically handle persistence of the new fields.
3. The actual business logic pertaining to editing the **DefaultHours** field is implemented using the method `onChangePre` in lines 44–70.

   Using the **onChangePre** rather than **onUpdatePre** means that the same logic will be applied to **Create** events.

   The **onChangePre** method calls the private method **resolveHours** which does something interesting in line 78: here the user-data is modified on behalf of the user. This is possible because the code is run in the "Pre" phase. Doing the same thing in the "Post" phase would not work, as explained in Section 4.3.

Notice that the code does *nothing* about the value of the field **Locked**. Since the field is declared as open, it can be edited by the end-user. The framework automatically persists the value entered by the user, since there is no logic changing the value of the field, and since user-input is by default accepted as the result-value.

Listing 4.9: Implementing an Update event.

```
2   private final static MiKey ENTITY_NAME = key("TRI_TimeSheetLine"
        );
3   private final static MiKey NS = key("Trifolium");
4   private final static MiKey FIELD_LOCKED = NS.concat(":Locked");
5   private final static MiKey FIELD_DEFAULT_HOURS = NS.concat(":
        DefaultHours");
6
7   @Override
```



```
8   public MiKey defineNamespace() { return NS; }
9
10  /**
11   * Add two fields:
12   *   Locked (bool): the line cannot be deleted while locked
13   *   DefaultHours (real): editing this field will set this
14   *       number of hours for all days that are not edited at
15   *       the same time by the user.
16   */
17  @Override
18  public MiExtended defineDomesticSpec(final MiDefine
        containerRunner) {
19    return McPaneSpec.McExtended.pane()
20      .addBooleanField(FIELD_LOCKED, "Lock Line").open().then()
21      .addRealField(FIELD_DEFAULT_HOURS, "Default Hrs.").open()
22      .end();
23  }
24
25  /**
26   * Defines the persistence strategy to use with this data-model.
27   * Data is stored in the custom table
28   *   TRI_TimeSheetLine
29   * in the Maconomy database.
30   */
31  @Override
32  public MiPersistenceStrategy definePersistenceStrategy(final
        MiContainerRunner.MiDefine containerRunner) {
33    return McMolPersistenceStrategy.create(ENTITY_NAME,
        getApiProvider());
34  }
35
36  /**
37   * Used for both Update and Create!
38   * If the user changes the field "DefaultHours", that
39   * value is transferred to any quantity-day field
40   * that the user has not edited at the same time.
41   * Then the DefaultHours field is reset.
42   */
43  @Override
44  public void onChangePre(final MiChangePre containerRunner,
45                          final MiUserChange eventData) throws
                               Exception {
46    final MiUserData userData = eventData.getUserData();
47    final MiDataValues resultData = eventData.getResultData();
48
49    if (userData.changed(FIELD_DEFAULT_HOURS)) {
50      final BigDecimal defaultHours = userData.getReal(
          FIELD_DEFAULT_HOURS);
51
```

```
52        // error if default hours < 0, i.e., check that hours >= 0
53        containerRunner.check(defaultHours.compareTo(BigDecimal.ZERO
              ) >= 0)
54                        .error("Default hours must be positive");
55
56        // On behalf of the user, set the quantity for each day to
57        // that indicated by the "default hours" field.
58        resolveHours(userData, key("QuantityDay1"), defaultHours);
59        resolveHours(userData, key("QuantityDay2"), defaultHours);
60        resolveHours(userData, key("QuantityDay3"), defaultHours);
61        resolveHours(userData, key("QuantityDay4"), defaultHours);
62        resolveHours(userData, key("QuantityDay5"), defaultHours);
63        resolveHours(userData, key("QuantityDay6"), defaultHours);
64        resolveHours(userData, key("QuantityDay7"), defaultHours);
65
66        // Reset the field such that it changes back to zero again
67        // (the field is used only to edit several fields by proxy).
68        resultData.setReal(FIELD_DEFAULT_HOURS, BigDecimal.ZERO);
69      }
70    }
71
72    private void resolveHours(final MiUserData userData,
73                              final MiKey dayQuantityField,
74                              final BigDecimal defaultHours) {
75      // only update a field if it was unchanged by the user;
76      // otherwise, let the user's own value win
77      if (userData.unchanged(dayQuantityField)) {
78        userData.setReal(dayQuantityField, defaultHours);
79      }
80    }
```

4.7 Implementing Delete Events

In this section, we shall have a closer look at `Delete` events. The `Delete` event is a data-carrying event, and consequently has a “Pre”- and a “Post”-event script: `onDeletePre` and `onDeletePost`. When you make a root container and use a data model, the data model offers you the possibility to implement a method called `onDelete`, whereas you may implement `onDeletePre` and `onDeletePost` methods for extension data models. In general, the event life-cycle for the `Delete` event follows the same pattern as shown for general data-carrying events. Figure 4.10 shows this.

One thing makes the `Delete` event slightly different from other data-carrying events: since the event-record is eventually being deleted, it makes no sense to provide a “result” value for that record! Consequently, the `eventData` object does not offer access to any result data. Only the original data is available.

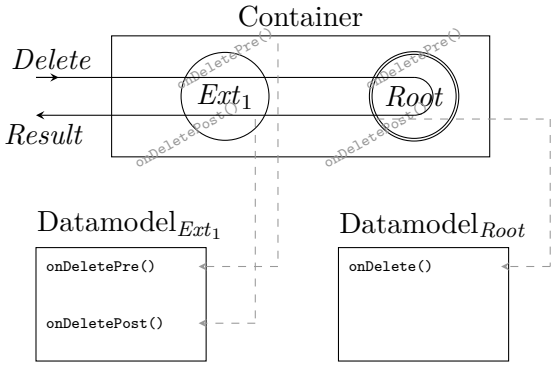


Figure 4.10: The life-cycle for the `Delete` event implemented using data-models. For a root contribution, the `onDelete` method is invoked from the container's `onDeletePost` method. For extensions, the `onDeletePre` and `onDeletePost` methods invoke similarly named methods in the data model.

Also, remember that in the `onDeletePost` method, all following contributions have already been executed and will not be invoked again. So, if you extend, say, the `maconomy:TimeSheets` container, the Time Sheet Line of the event has already been deleted when the `onDeletePost` method is invoked in your data model! Therefore, you will not be able to look up the corresponding `TimeSheetLine` in the Maconomy database at this time. If you have added custom fields to time sheet lines, and these are stored in a custom table, that custom-table record will not have been deleted when the `onDeletePost` method is invoked.

Obviously, the idea about `Delete` events is that a record should be deleted. *You should not delete the record yourself: the Extension Framework will automatically do this.* This is true for the "main" record. If you have additional related records, that the framework knows nothing about, you will have to delete these yourself. As explained in Section 4.5, how a record is deleted is declared by the `definePersistenceStrategy` method.

If your code determines that for some reason, the deletion cannot be accepted, you can show an error-message to the end-user. Doing so will automatically roll back the current transaction. Notice that *all contributions in the container* are being executed in the same transaction. So in the example above, even if the root-contribution for `maconomy:TimeSheets` (the "core application") may have already deleted a specific time sheet line in the database, giving an error in an extension's `onDeletePost`-script will imply that the deletion in the Maconomy-database is being rolled back.

Continuing the example from Listing 4.9, let us show how the logic around deletion would be implemented. Remember that lines where the custom field `Locked` has the value true, must not be deleted. The code is shown in Listing 4.10.

Notice that there is *no* code that *actually* deletes any records: this is all automatically handled by the Extension Framework.

Listing 4.10: Implementing a Delete event.

```
1
2   /**
3    * In order to fail as quickly as possible, we implement our
4    * guard in the Pre-script. The behavior would be exactly
5    * the same, if implemented as onDeletePost.
6    */
7   @Override
8   public void onDeletePre(final MiDeletePre containerRunner,
9                           final MiDelete eventData) throws
                               Exception {
10    final MiValueInspector originalData = eventData.
          getOriginalData();
11
12    containerRunner.check(!originalData.getBool("Locked"))
13                   .error("The line cannot be deleted");
14  }
```

4.8 Implementing Action Events

In this section, we shall have a closer look at `Action` events. The `Action` event is a data-carrying event, and consequently has a "Pre"- and a "Post"-event script: `onActionPre` and `onActionPost`. The general life-cycle of `Action`-events is shown in Figure 4.11. Compared to other data-carrying events, the `Action` event is slightly different because there is really not just *one* action event. The "`Action`" event is an entire *class of events*. A specific `Action` event is associated with a *name* that makes it unique compared to other `Action`-events. For example, in the `maconomy:TimeSheets` dialog, there are a number of different actions, e.g., `SubmitTimeSheet`, `ApproveTimeSheet` and `ReopenTimeSheet`. Invoking the `SubmitTimeSheet` action is quite different from invoking the `ApproveTimeSheet` action, and they should not be mixed up! Because there are infinitely many possible action names, it is not possible to have dedicated event-methods for each specific action. Instead, invoking some action, will result in a generic `Action` event. The event-information, however, informs about the name of the action.

When using data-models, it has been chosen *not* to have one generic "`onAction`" method that will be invoked for all actions. This more easily leads to errors, and requires the programmer to have switch on the action name in order to handle the event correctly. It would be a pity, if invoking the `SubmitTimeSheet` action accidentally implemented the `ApproveTimeSheet` behavior instead!

Instead, it has been decided that each action must have *its own dedicated event methods*. This is achieved by using Java *inner classes* and Java *annotations*. For each action you want to contribute to, you must do the following:

Declare an inner class inside the data model class. The class can be a `static` class or

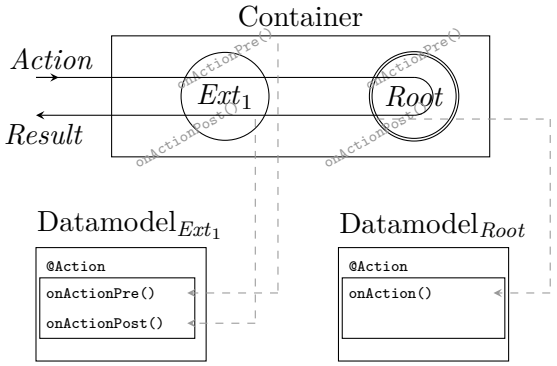


Figure 4.11: The life-cycle for the `Action` event implemented using data-models. For a root contribution, the `onAction` method is invoked from the container's `onActionPost` method. For extensions, the `onActionPre` and `onActionPost` methods invoke similarly named methods in the data model. Notice that the invoked event methods are not implemented directly in the data model. Instead they are implemented in `@Action`-annotated inner action-handler classes.

a non-`static` class. A non-`static` class is necessary if the inner class *must* access non-static information contained in the data-model class. Otherwise, a `static` class should be declared.

This inner class, which we refer to as an *action-handler class*, must extend one of the following abstract classes:

`McAbstractDataModelExtendedAction` if your data model is an *extension* contribution *and* the action in question is *not declared by your extension.*

`McAbstractDataModelRootAction` if your data model is a *root* contribution *or* the action in question *is introduced by* your *extension.*

Annotate the class with an `@Action`-annotation[11], declaring the name of the action that is being implemented by that code. If the name of the action is `SubmitTimeSheet`, you make an annotation like:

```
@Action("SubmitTimeSheet")
```

If the action name is `CalculateProfit` the annotation would look like:

```
@Action("CalculateProfit")
```

Although it is good style to use the same casing all over, the name of an action is *case-insensitive*, so the following two annotations are considered the same: `@Action("abc")` and `@Action("AbC")`.

[11] Formally the annotation class: `com.maconomy.toolkit.panes.datamodels.Action`

It is important to notice that for actions that you *don't want* to contribute to, *you don't have to do anything!*

In order to get a better understanding of all of this, let us have an example: suppose you want to make an extension to the `maconomy:TimeSheets` container. Your client has requested that you modify the `SubmitTimeSheet` action such that if no hours have been registered, the time sheet can't be submitted. Also, unless there are more than 2 hours registered, a warning should be issued upon approval.

So, what do you need to do? Obviously, you aim is to extend an existing container, extending two existing actions. Since you want to modify the behavior of *two different actions*, you need to declare *two inner classes* and annotate both of them using the `@Action` annotation. Since the actions are not introduced by your extension[12], you declare the inner classes as extensions of the abstract class `McAbstractDataModelExtendedAction`.

This may seem complicated, but it really isn't. Listing 4.11 shows the implementation of an entire data-model class handling the above functionality. There are two inner classes, each extending `McAbstractDataModelExtendedAction`. The class implementing the extended behavior of the `SubmitTimeSheet` action is found in lines 8–19. The event-method is found in line 10. By the `@Action`-annotation in line 7, the framework knows that this code belongs to the `SubmitTimeSheet` action.

The other action-handler class, implementing the extension to the `ApproveTimeSheet` action, is implemented in lines 22-34. The annotation telling the framework which action this is meant for is found in line 21. The actual event handling method starts in line 24.

Listing 4.11: Implementing Action events.

```
2  public class ActionDataModel extends McAbstractExtendedDataModel {
3    public ActionDataModel(final MiDataModelFactory.MiResources
         resources) {
4      super(resources);
5    }
6
7    @Action("SubmitTimeSheet")
8    private static final class SubmitTimeSheetHandler extends
         McAbstractDataModelExtendedAction {
9      @Override
10     public void onActionPre(final MiActionPre containerRunner,
11                             final MiAction eventData) throws
                                   Exception {
12       final MiValueInspector originalData = eventData.
             getOriginalData();
13
14       final BigDecimal totalTime = originalData.getReal("
             TotalNumberOfWeekVar");
```

[12] They are introduced by the root contribution in this case.


```
15        containerRunner
16          .check(totalTime.compareTo(BigDecimal.ZERO) == 0)
17          .error("Please register time before submitting!");
18      };
19    }
20
21    @Action("ApproveTimeSheet")
22    private static final class ApproveTimeSheetHandler extends
          McAbstractDataModelExtendedAction {
23      @Override
24      public void onActionPre(final MiActionPre containerRunner,
25                              final MiAction eventData) throws
                                    Exception {
26        final MiValueInspector originalData = eventData.
              getOriginalData();
27
28        final BigDecimal totalTime = originalData.getReal("
              TotalNumberOfWeekVar");
29        containerRunner
30          .check(totalTime.compareTo(new BigDecimal("2")) <= 0)
31          .warning(String.format("Only %2.1f hours registered",
32                                 totalTime.doubleValue()));
33      };
34    }
```

The annotations help the programmer and the run-time to easily identify the code implementing an action. As for other data-carrying events, you have the possibility of implementing a "Pre"-method and/or a "Post"-method.

Sometimes, you may desire to add an entirely new action to a container. Naturally, you can do nothing else than that when implementing a root contribution. But sometimes, an extension contribution may invent entirely new actions. Such actions are regarded as *root-level actions*, and the framework will automatically ensure that the container in which the action is added, is treated as the root. An event-flow as shown in Figure 4.12 is taking place. Notice that contributions following the one in which the action is executed *are not invoked*[13]. There's a really good reason for this: a contribution following the introducing one, cannot know anything about the particular action. First of all, it has not invented it itself. Secondly, it can only rely on behaviors of contributions that it extends itself. As it is further towards the root than the one introducing the actions, it cannot know that this actions exists. So what should it do, when asked to execute some action? Absolutely nothing! For this reason, the Extension Framework treats the introducing container as the root in this particular case!

It is also important to notice that contributions extending the contribution which introduces an action, will be invoked as usually. Such extensions can know about the

[13]This is the behavior enforced when using data-models. If you write a container directly using containers, this behavior is not guaranteed. As usual, you should always use data-models.

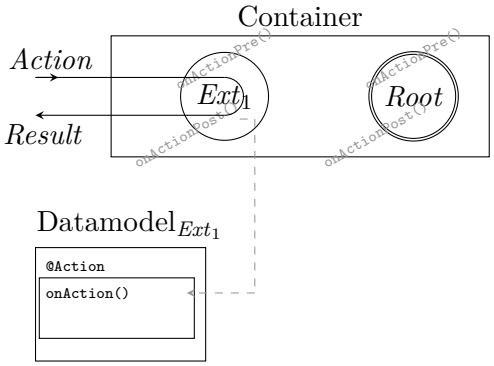


Figure 4.12: The life-cycle for the `Action` event in the case where the executed action is introduced in an *extension contribution*. When such an action is executed, the contribution that introduces the action is treated as the root *for that action-execution*.

existence of the action, since they extend the contribution that defines it. Again, let us clarify this by looking at an example. Suppose we want to introduce two actions in the `maconomy:Jobs`: one that marks the job as being "Blocked" for time registrations, amount registration, invoicing and budgeting. And another one that similarly "Unblocks" the job. Thus, we need to introduce two new actions. Since they are introduced by our extension contribution, the action-handler classes must extend the `McAbstractDataModelRootAction`. Extending this class let us only implement the method `onAction`. Apart from that, there's really no big difference from extending an existing action. Listing 4.12 shows an entire data-model implementation that contributes two new actions: `Trifolium:Block` and `Trifolium:Unblock`. The `Trifolium:Block` action (annotated in line 22) is implemented by the inner action-handler class in lines 23–35. Notice the method name `onAction` in line 25 (*not* ~~`onActionPre`~~ or ~~`onActionPost`~~) which does the actual implementation.

Similarly, the action `Trifolium:Unblock` (annotated in line 37) is implemented by the action-handler class in lines 38–50. Again, the name of the event-method is `onAction` which is found in line 40.

Listing 4.12: Implementing Action events for *new* actions.

```
2  public class RootActionDataModel extends
       McAbstractExtendedDataModel {
3    public RootActionDataModel(final MiDataModelFactory.MiResources
       resources) {
4      super(resources);
5    }
6
7    private static final MiKey NS = key("Trifolium");
8    private static final MiKey ACTION_BLOCK = NS.concat(":Block");
9    private static final MiKey ACTION_UNBLOCK = NS.concat(":Unblock"
       );
```



```
10
11    @Override
12    public MiKey defineNamespace() { return NS; }
13
14    @Override
15    public MiPaneSpec.MiExtended defineDomesticSpec(final MiDefine
          containerRunner) {
16      return McPaneSpec.McExtended.pane()
17        .addAction(ACTION_BLOCK, "Block Job").then()
18        .addAction(ACTION_UNBLOCK, "Unblock Job")
19        .end();
20    };
21
22    @Action("Trifolium:Block")
23    private static final class BlockHandler extends
          McAbstractDataModelRootAction {
24      @Override
25      public void onAction(final MiActionPost containerRunner,
26                           final MiAction eventData) throws
                                 Exception {
27        final MiContainerExecutor jobs = containerRunner.executor().
              construct();
28        jobs.update(dataValues()
29                    .setBool("BlockedForTimeRegistrations", true)
30                    .setBool("BlockedForInvoicing", true)
31                    .setBool("BlockedForBudgeting", true)
32                    .setBool("BlockedForAmountRegistrations", true))
                         ;
33        containerRunner.fullRefresh();
34      }
35    }
36
37    @Action("Trifolium:Unblock")
38    private static final class UnBlockHandler extends
          McAbstractDataModelRootAction {
39      @Override
40      public void onAction(final MiActionPost containerRunner,
41                           final MiAction eventData) throws
                                 Exception {
42        final MiContainerExecutor jobs = containerRunner.executor().
              construct();
43        jobs.update(dataValues()
44                    .setBool("BlockedForTimeRegistrations", false)
45                    .setBool("BlockedForInvoicing", false)
46                    .setBool("BlockedForBudgeting", false)
47                    .setBool("BlockedForAmountRegistrations", false)
                         );
48        containerRunner.fullRefresh();
49      }
```

```
50    }
```

For the sake of clarity, the shown implementation almost duplicates the code of the two actions. In a real-life implementation, an abstract class would be made, and the action-handler for the two classes would extend that class. Listing 4.13 shows this: the behavior is identical to the one shown above, but here we let our annotated action-handlers extend a custom class. The annotations are used by the framework to identify the class implementing the behavior of a given action. As long as this class is a subtype of the right abstract class, it is of no importance whether it extends the required sub class directly or by extending some other abstract class. Here, the common behavior is implemented by the abstract inner class in line 16, whereas the actual action implementations now merely extend this class, which can be seen in lines 3 and 10.

Listing 4.13: Implementing Action events using an abstract class.

```
2   @Action("Block")
3   private static final class BlockHandler extends AbstractHandler{
4     BlockHandler() {
5       super(true);
6     }
7   }
8
9   @Action("Unblock")
10  private static final class UnblockHandler extends
        AbstractHandler{
11    UnblockHandler() {
12      super(false);
13    }
14  }
15
16  private static abstract class AbstractHandler extends
        McAbstractDataModelRootAction {
17    private final boolean value;
18    AbstractHandler(final boolean blockValue) {
19      this.value = blockValue;
20    }
21    @Override
22    public void onAction(final MiActionPost containerRunner,
23                         final MiAction eventData) throws
                             Exception {
24      final MiContainerExecutor jobs = containerRunner.executor().
            construct();
25      jobs.update(dataValues()
26                  .setBool("BlockedForTimeRegistrations", value)
27                  .setBool("BlockedForInvoicing", value)
28                  .setBool("BlockedForBudgeting", value)
29                  .setBool("BlockedForAmountRegistrations", value)
                        );
30      containerRunner.fullRefresh();
```



```
31        }
32    }
```

When adding actions, these will by default be exposed as "enabled" whenever it makes sense. However, it is not always desired that an action is exposed as enabled. An enabled action will be rendered as "clickable" in the GUI, whereas a disabled action will be rendered in a grayed-out manner.

In order to control the enabledness of actions (including the "standard actions" such as `Initialize`, `Update`, `Delete` and `Print`), the Extension Framework will invoke the method `refreshActions` in the data model. Notice that this action is not (only) invoked for action events; it is called for each event (including reads), thereby ensuring that the enabledness reflect the data being presented. Please refer to Section 4.12 for more information on this topic.

4.9 Implementing Print Events

In this section, we shall have a closer look at `Print` events. The `Print` event is a data-carrying event, and consequently has a "Pre"- and a "Post"-event script: `onPrintPre` and `onPrintPost`. When you make a root container and use a data model, the data model offers you the possibility to implement a method called `onPrint`, whereas you may implement `onPrintPre` and `onPrintPost` methods for extension data models. In general, the event life-cycle for the `Print` event follows the same pattern as shown for general data-carrying events. Figure 4.13 shows this.

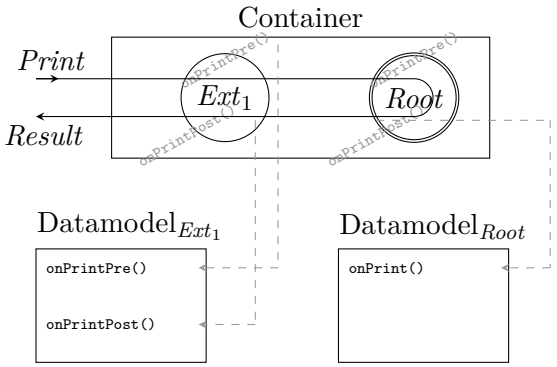


Figure 4.13: The life-cycle for the `Print` event implemented using data-models. For a root contribution, the `onPrint` method is invoked from the container's `onPrintPost` method. For extensions, the `onPrintPre` and `onPrintPost` methods invoke similarly named methods in the data model.

It may come as a surprise that the `Print` event is data carrying? Why doesn't it just show a print-out of the record in question? The truth is that the `Print` very much

corresponds to an `Action`-event. Only this particular action has a pre-defined name and consequently a dedicated event. *By custom*, this event generates a "print out." The truth is, however, that nothing *requires* this event to generate print-outs.

The event *does* support updates of data. For instance, you might want to implement a `PrintOutDateAndTime` that stores a time stamp for when a print was last generated. Doing so requires updating the data. For this reason, you can address both original data as well as result data from the event.

The coupling service comes bundled with a PDF-library called "iText" [iTe08, Low10]. This 3rd-party library is documented elsewhere, and going into how to use it is therefore not in the scope of this document.

Just as your print action (or any other action!) can result in some PDF-document, you are allowed to produce one or more documents of any type. A document is returned to the user through the call-back mechanism. If the client-machine knows about the specific document type, it can be opened on the client machine. For example, the output could be a text (`.txt`) document, a Microsoft Word (`.doc`) or Microsoft Excel (`.xls`) document instead of a PDF. Callbacks are covered in Section 5.4.

4.10 Implementing Move Events

In this section, we shall have a closer look at `Move` events. The `Move` event is a data-carrying event, and consequently has a "Pre"- and a "Post"-event script: `onMovePre` and `onMovePost`. When you make a root container and use a data model, the data model offers you the possibility to implement a method called `onMove`, whereas you may implement `onMovePre` and `onMovePost` methods for extension data models. In general, the event life-cycle for the `Move` event follows the same pattern as shown for general data-carrying events. Figure 4.14 shows this.

The `Move`-event, like the `Action`-event, is a class of events. It may occur as a result of one of the following user-events:

Move Up This occurs when the user "moves a line upwards," e.g., by pressing the button ⬆.

Move Down This occurs when the user "moves a line downwards," e.g., by pressing the button ⬇.

Indent This happens when the user "indents a line," e.g., makes a line a child-line of another line. This may happen by pressing the button ➡.

Outdent This happens when the user "outdents a line," e.g., makes a child-level-line move into the same level as its parent. This may happen, e.g., by pressing the button ⬅.

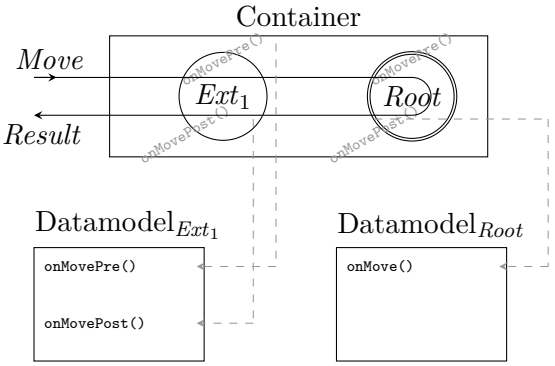


Figure 4.14: The life-cycle for the `Move` event implemented using data-models. For a root contribution, the `onMove` method is invoked from the container's `onMovePost` method. For extensions, the `onMovePre` and `onMovePost` methods invoke similarly named methods in the data model.

"Drag'n'Drop" This happens when the user, using the mouse, drags a line from one position in a table and drops it at some other location. This may be equivalent to one of the above operations, or to a series of such operations. Only *one* event will occur.

In the Extension Framework, each move-event provides additional information that is needed to interpret the event, including what kind of move operation has been performed. Information is given about the *source row*—the row being moved—as well as a *target row*—the row relative to which the move operation must be interpreted. This information can be obtained from the `eventData` parameter by invoking the methods `getSourceRowNumber` and `getTargetRowNumber` respectively. The source and target row numbers are interpreted in the following way:

1. The indexes are 0-based. So the first line is line number 0.
2. If there is a tree-structure, the line indexes are relative to the *flattened* list of lines organized using the default server-side sorting order.

For example, consider Table 4.20. This tables shows an a number of records organized in a tree-structure. The lines are ordered with respect to the Line-numbers of records at each level, an such that the tree-structure is respected, i.e., that top-level nodes are placed first, and child nodes follow their parent node. The table then shows how the index used for the `getSourceRowNumber` and `getTargetRowIndex` methods. For example, the top-most line (the first line not having any parents, P1) has index 0. The first child of that line has index 1 etc.

Index	`Description`	`LineNumber`	`Key`	`ParentKey`
0	Parent 1	1	P1	⟨*blank*⟩
1	Child(P1) 1	1	C1P1	P1

Index	Description	LineNumber	Key	ParentKey
2	Child(P1) 2	2	C2P1	P1
3	Child(C2P1) 1	1	C1C2P1	C2P1
4	Parent 2	2	P2	⟨*blank*⟩
5	Child(P2) 1	1	C1P2	P2
6	Child(P2) 2	2	C2P2	P2

Table 4.20: The row indexes correspond to the 0-based position of a line in the "visual order" of the default server-side ordering. This is the "flattened list."

In addition to the source and target row index, the `getMoveOperation` method on the `containerRunner` object, returns an `enum`-type that reflects *how* to interpret the source row relative to the target row. In combination this information specifies what the move-operation is all about. The possible values of the `getMoveOperation` are:

`MOVE_AFTER` Which means that the source row is moved immediately after the target row, having the same parent[14] as the target row.

`MOVE_BEFORE` Which means that the source row is moved immediately before the target row, having the same parent as the target row.

`MOVE_INTO` Which means that the source row becomes the *first* child of the target row.

If you make a root-contribution allowing tree-operations, you should note that maintaining the tree-structure is handled by the Extension Framework. The aim of event-methods is to do additional logic pertaining to moving data. This could include validating if a specific move is allowed or should lead to an error message, or maintaining the values of fields that relate to moving. Usually, the logic around moving data is virtually non-existing, in which case there is no reason to implement data. If your data-structure contains aggregated data, you will need to implement logic that maintains this when `Move` events occur.

4.11 Implementing Lock and Unlock Events

As of Maconomy 2.3, `Lock` and `Unlock` events will not occur. Implementing support for these is consequently deprecated. As the "locks" have always been "convenience locks" (i.e., not necessarily guaranteed), there is no replacement for these.)

In this section, we shall have a brief look at ~~`Lock`~~ and ~~`Unlock`~~ events. The `Lock`/`Unlock` events are data-carrying events, and consequently have a "Pre"- and a "Post"-event

[14] If the table is not a tree-structured table, all lines will be considered having an "empty" parent.

script: onLockPre/onUnlockPre and onLockPost/onUnlockPost. When you make a root contribution you must implement the Lock/Unlock behavior directly in the container. For extension contributions you may implement onLockPre/onUnlockPre and onLockPost/onUnlockPost methods for extension data models. In general, the event life-cycle for the Lock/Unlock events follow the same pattern as shown for general data-carrying events. Figure 4.15 shows this for the Lock event. The Unlock event is similar.

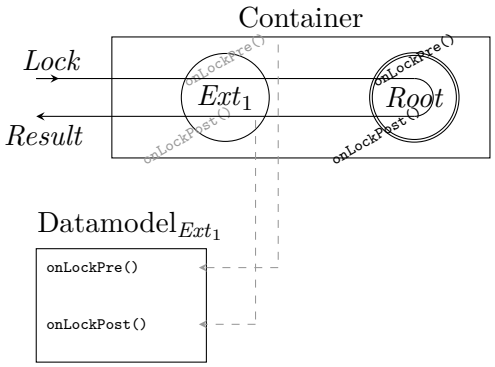


Figure 4.15: The life-cycle for the Lock event implemented using data-models. For extensions, the onLockPre and onLockPost methods invoke similarly named methods in the data model. For a root contribution, the locking mechanism must be implemented directly in the container. It is expected that data-models may be used for this in the future.

It is considered very rare, that you should want to implement functionality for lock and unlock events in an extension contribution. For root contributions, this may be more likely. How you implement this is entirely up to you. The Extension Framework has no utilities supporting this, however.

4.12 Implementing Read Events

In this section, we shall have a closer look at Read events. The Read event is a data-carrying event, and consequently has a "Pre"- and a "Post"-event script: onReadPre and onReadPost. When you make a root container and use a data model, the data model offers you the possibility to implement a method called onRead. In general, the event life-cycle for the Read event follows the same pattern as shown for general data-carrying events. Figure 4.16 shows this. The support for implementing "read events" is mostly for the sake of completeness. Actually implementing logic pertaining to merely *reading* data is probably very rare. An example could be record-level access control on custom data.

Most often, you would *not* need to implement anything specifically for Read events. Most

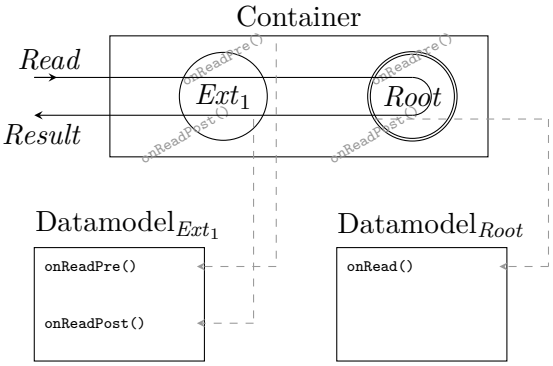


Figure 4.16: The life-cycle for the `Read` event implemented using data-models. For a root contribution, the `onRead` method is invoked from the container's `onReadPost` method. For extensions, the `onReadPre` and `onReadPost` methods invoke similarly named methods in the data model.

of what could reasonably take place for reads is handled by the Extension Framework or by other means. The following could be considered tasks for a `Read` event:

1. Fetching the desired data from the database. The Extension Framework automatically handles this, and *you should not do this yourself.* For root-level data-models, obviously some data should be read, for instance from a database. For extension contributions one of two scenarios may occur:

   (a) The extension adds new (persisted) fields. In this case, the framework will automatically gather the needed data. The framework can even do this more efficiently, because it does so prior to invoking the data model.

   (b) The extension does not add any (persisted) fields. In this case, there's no need to read any data whatsoever.

   By invoking the method `definePersistenceStrategy`, the framework uses knows how reading is performed, e.g., by looking up data from a custom database table in the Maconomy database.

2. Calculating values for added variables ("non-persisted fields.") While this observation is true, you should remember that a similar task should be carried out *for any data-carrying event!* Because of this, calculating variable values is maintained by the `refreshVariables` method which will automatically be invoked by the framework whenever needed. In general, this method may be needed for all records in a data-response, not only the event-record of the actual event. The bottom line is: your implementation of `Read` should *not* carry out this task!

3. Calculate the action-enabledness status. Each pane has a set of attributes indicating which actions are enabled and which are not. This information is used by the workspace client to show actions/operations as enabled or disabled. Whereas it

is true, this should be carried out *for any data-carrying event!* Because of this, calculating the action enabledness state is maintained by the `refreshActions` method which will automatically be invoked by the framework whenever needed. The bottom line is: your implementation of `Read` should *not* carry out this task.

4. Imposing a built-in restriction for data-selection for root contributions. Suppose you are building a root container, and suppose that you want a table-pane to contain records each corresponding to some piece of "equipment." Your equipment records may be "blocked." Suppose that you want a particular container to only show non-blocked equipment. In this case, you should ensure that a particular condition (where-clause) is always applied. This is not done directly by the data-models `onRead` method. In fact, the `onRead` is invoked *for each record* that is being fetched by the framework. This situation is handled by letting your data model implement the method `defineAdditionalReadCondition`. See Section 4.12.1.

5. Imposing a specific default ordering on data in a pane. Again, suppose you are building a root container, and suppose that you want a table-pane to contain records each corresponding to some piece of "equipment." Among other things, an equipment record contains a field `DataOfPurchase`. Suppose that you want the default-ordering of data in this pane to show the most recently purchased equipment at the top. You can control this by applying a default ordering. Like above, this is not a task for the `onRead` method which is invoked for each record being read. Instead, you provide this information to the framework by letting your data-model implement the method `defineDefaultSortOrder`. See Section 4.12.1.

As you can see from the above, all the "obvious" tasks related to `Read`-events are automatically handled by the framework.

4.12.1 Controlling Restrictions and Sorting

When building a root contribution, sometimes you may want to impose certain built-in restrictions to data shows in a pane. And sometimes you may wish to control the default-sorting. *Notice that this makes sense only for root contributions*: if you extend a container, you cannot dictate that some data is not part of a given pane. It *is* because the root said so! The same goes for the default sorting[15].

The methods `defineAdditonalReadCondition` and `defineReadCondition` are used to restrict the data potentially being read. You are encouraged to implement the `defineAdditionalReadCondition` if possible.

The way reading works is that eventually, the framework needs to read data. Based on the *key* (for card and table panes) or *requested restriction* (for filter panes), the framework deduces a *suggested restriction*. The method `defineReadCondition` is invoked, and it the suggested restriction is given as input in the form of an expression.

[15]The user can always change the client-side sorting.

The default implementation simply invokes `defineAdditionReadCondition` and builds an expression corresponding to

`suggestedRestriction` ∧ `defineAdditionReadCondition(...)`

Hence, by implementing the `defineAdditionReadCondition` method, you don't have to remember to add the suggested restriction.

In rare cases, you might *need* to override the implementation of `defineReadCondition`. This could be the case, if the suggested condition does not match your data. For example, suppose your container is meant to take an `EmployeNumber` as a key, thus allowing you to `<Bind>` your container in a workspace using an Employee-related foreign key. For some reason, *does not contain an* ***`EmployeeNumber`*** *field*. The suggested restriction will match the binding key, e.g. something like `EmployeeNumber = '100992'`. In this case, you need to override the implementation of `defineReadCondition` so that it transforms the suggested restriction into an expression that makes sense for your data, e.g., `EmployeeID = '100992'` or even `EmployeeNumber = '777228'`! Unless your implementation explicitly invokes and conjugates the result of `defineAdditionalReadCondition`, that method will now no longer work as intended!

Most often, you can rely on the suggested condition and you will want only to implement `defineAdditionReadCondition`.

The default sorting of items in a table-pane will be defined (more or less arbitrarily) by the database if nothing is specified. However, if your data-model contains auto-positioned data and, therefore, your data-model extends `McAbstractAutoPositionRootDataModel`, the default sort-order will (and must) be ordering by the auto-position field. The framework will ensure this. Similarly, the framework will enforce an auto-positional ordering respecting tree structures when the pane contains tree-structured data, thereby extending `McAbstractTreeStructuredRootDataModel`. For plain table panes (i.e., without auto-positionable content and without tree-structured data), you can specify the default sort-order by implementing the method `defineDefaultSortOrder`. This method should return a list of `McSortOrder`s. A sort order comprises a field name and an ordering (ascending or descending.) The returned list is interpreted such that the first element in the list is the most significant sort-order. For items with equal values, the next sort order will be applied and so on.

Listing 4.14 shows an example that specifies an additional where clause that is dynamically changed depending on settings in an associated card pane: Equipment marked as being "blocked" will be excluded from the list, unless a field, `IncludeBlocked`, contained in the card pane is set to `true`. Furthermore, the list of equipment will be shown using a default sorting that sorts the equipment by the `CompanyNumber` owning the piece of equipment, and—secondary—by `PurchaseDate`, showing the most recently purchased equipment first.

Listing 4.14: Custom data-restrictions and sorting.

1


```
2    /**
3     * Exclude blocked items, unless the associated card pane
4     * specifies that blocked fields should be included
5     */
6    @Override
7    public MiExpression<McBooleanDataValue>
        defineAdditionalReadCondition(
8            final MiContainerRunner.MiDefine containerRunner,
9            final MiContextData contextData) throws Exception {
10     final MiContextPaneData cardPane = contextData.getPaneData().
           get();
11     // don't include blocked unless "IncludeBlocked"
12     // is set in the card pane (assumed to be present!)
13     if (!cardPane.getResultData().get().getBool("IncludeBlocked"))
           {
14       return dataValues().setBool("Blocked", false).asExpression()
           ;
15     } else {
16       return McExpressionUtil.TRUE;
17     }
18   }
19
20   /**
21    * The default sort order is:
22    *   Company No. (ascending): primary
23    *   Purchase Date (descending): secondary
24    */
25   @Override
26   public MiList<McSortOrder> defineDefaultSortOrder(
27           final MiContainerRunner.MiDefine containerRunner,
28           final MiEventData eventData) {
29     final McSortOrder companyOrder =
30         new McSortOrder(FIELD_COMPANY_NO,
31                         MeSortType.ASCENDING);
32     final McSortOrder newestFirst =
33         new McSortOrder(FIELD_PURCHASE_DATE,
34                         MeSortType.DESCENDING);
35     return McTypeSafe.createArrayList(companyOrder, newestFirst);
36   }
37
```

4.12.2 Refreshing Variable Values

As indicated on page 125, calculated fields (a.k.a. variables) should be calculated when data is read. Or when any other data-carrying event is executed, for that matter!

The Extension Framework will automatically invoke the method `refreshVariables` for

each record in a response[16]. In this way, it is ensured that calculated values are always correctly updated relative to each specific record.

As an example, suppose your client wants to include the "job description" and the "popup1" of the job for each line in a time sheet. Each line already has a `JobNumber` field, and the job *name* is shown in a calculated field. However, the job *description* and *popup1* are not available. By introducing two new variables in an extension to the `maconomy:TimeSheets` container, we can add information about the job description for each line. Listing 4.15 shows the implementation of such a data model.

In the example, there are a few things to notice: the implementation makes use of a utility class called `McCachedDataHost`, see Section 6.2.6. This is declared in lines 17–22. This class makes it possible to abstract fields from various database tables into a `Map`-like structure. You can look up the value of fields based on a key. This class internally implements a cached buffer between the calling code and the database: if information concerning a specific key has already been obtained, the database will not be queried. By declaring the set of all fields you might be interested in through this mechanism, it is ensured that *when* you make a query against some table, you fetch *all* the relevant fields. This allows your code to be more clear, because you don't have to manually fetch fields "too early", without compromising performance. So, if your code looks up information based on the same key several times, only one database query will be made. It is important that such `McCachedDataHost`s are *short-lived!* You shouldn't keep such object for a long time. It is generally adequate to keep such objects as member variables, since each event will create new data-model instances.

The addition of the two variables is done by the method `defineDomesticSpec` starting in line 38. The values of these variables are calculated in the method `refreshVariables` in lines 48–72. The "calculation" is done by looking up the `JobDescription` and the `Popup1` fields of the `JobHeader` table, based on the current value of the `JobNumber` field of the time sheet line. Of course, it may happen that there is not yet a job specified on the time sheet line in question. This case is automatically handled by the `dataHost`, because the interface to the `JobHeader` table is asked to "ignore" blank job numbers. In this case, a look-up will default to a specified default value, in this case, an empty string/nil value. These look-ups happen in lines 56 and 64.

Listing 4.15: Calculating Variables.

```
2   // lazily initialized dataHost
3   private MiOpt<McCachedDataHost> dataHost = McOpt.none();
4   public ExtendedTimeSheetsTableDataModel(final MiDataModelFactory
        .MiResources resources) {
5     super(resources);
6   }
7
8   /**
```

[16]Remember that this is not necessarily the same as *all* records in a pane, since pane-level responses may be partial!


```
9       * Returns the data-host in use for this data model.
10      * The data host must not be initialized in the constructor.
11      * Therefore, we introduce this method to lazily initialize it
            on demand.
12      * @return The data host.
13      */
14    private McCachedDataHost getDataHost() {
15      if (this.dataHost.isNone()) {
16        McCachedDataHost dh;
17        dh = new McCachedDataHost(getApiProvider());
18        dh.installCache("JobHeader", // table name
19                        "JobNumber", // name of key field
20                        opt(McStr.val("")), // ignore ""
21                        "JobDescription", // wanted fields
22                        "Popup1");
23        this.dataHost = opt(dh);
24      }
25      return this.dataHost.get();
26    }
27
28    private static final MiKey NS = key("Trifolium");
29    private static final MiKey VAR_JOB_DESCRIPTION =
30        NS.concat(":JobDescriptionVar");
31    private static final MiKey VAR_JOB_POPUP1 =
32        NS.concat(":JobPopup1Var");
33
34    @Override
35    public MiKey defineNamespace() { return NS; }
36
37    @Override
38    public MiExtended defineDomesticSpec(final MiDefine
          containerRunner) {
39      return McPaneSpec.McExtended.pane()
40        .addStringVariable(VAR_JOB_DESCRIPTION, "Job Descr.").then()
41        .addPopupVariable(VAR_JOB_POPUP1,
42                          "Job Popup 1",
43                          key("JobPopupType1")).then()
44        .end();
45    }
46
47    @Override
48    public void refreshVariables(final MiContainerRunner.MiDataPost
          containerRunner,
49                                 final MiResult eventData) throws
                                       Exception {
50      final MiDataValues resultData = eventData.getResultData();
51      final McCachedData jobHeader = getDataHost().getCache("
          JobHeader");
52      // retrieve the JobDescription from the job with the
```

```
53      // specified job number. Return an empty string if the
54      // job is blank.
55      final McDataValue jobNumber = resultData.getVal("JobNumber");
56      final McDataValue jobDescription =
57          jobHeader.getValue(jobNumber,
58                             "JobDescription",
59                             McStr.val(""));
60
61      // retrieve the Popup1 from the job with the specified
62      // job number. Return a nil value if the job
63      // is blank.
64      final McDataValue jobPopup1 =
65          jobHeader.getValue(jobNumber,
66                             "Popup1",
67                             McPopup.nil("JobPopupType1"));
68
69      // Assign the variable values in the result
70      resultData.setVal(VAR_JOB_DESCRIPTION, jobDescription)
71                .setVal(VAR_JOB_POPUP1, jobPopup1);
72  }
```

The implementation shown above will work. But it as a slight problem: if there are many lines in a specific time sheet, then potentially, a new database query will be made for each line. Although—because of the `McCachedDataHost`—only one query will be made for each unique job number.

In order to give the programmer a chance to do something about this, you may optionally implement the method `refreshVariablesPrepare`. This method *may* be invoked by the Extension Framework, although this is not guaranteed: for example, if there is only one record in a result set, this method will not be called.

`refreshVariablesPrepare` gives information to the data model stating something like: "in a short while, you can expect `refreshVariables` to be invoked for each record in a given set of records. The idea is, that an implementation of `refreshVariablesPrepare` may perform an optimized query, extracting all the needed information, and cache this in a short-living cache, that may be used by `refreshVariables`. The implementation of `refreshVariables` must be robust against the case that the cache does not contain the relevant information.

By using the `McCachedDataHost` such behavior comes almost for free. The interface used to look up data *is* robust against a situation where the desired information has not already been pre-fetched, while using a cached version if information is already available.

Listing 4.16 shows an implementation of `refreshVariablesPrepare` that works with the example shown in Listing 4.15 above. In line 9 the implementation makes use of a utility class `McDataModelUtil` and a method `extractValues` that is capable of extracting a set of unique field values from a set of records. In the example, we extract all unique `JobNumber` values occurring in the set of time sheet lines. In line 15 these values are used

to populate the `dataHost` cache with respect to these values. This performs better than looping over the records and fetching data corresponding to each record, because one single database query is being built. Thereby, the cache is populated so that when the `refreshVariables` method is invoked for each record shortly after, the data is present, and no further database queries are needed.

Applying this pattern for extracting related information from the database is strongly encouraged. At present, the `McCachedDataHost` only supports handling records with *one* key field. If you have a case that requires look up in data structures with more than one key field, you will need to make a similar mechanism (hard-coded or generic.) In any case, *it is strongly advised that you implement the* `refreshVariablesPrepare` *method for multi-record panes!*

Listing 4.16: Preparing Variable Calculation.

```
2   @Override
3   public void refreshVariablesPrepare(
4       final MiContainerRunner.MiDataPost containerRunner,
5       final MiCollection<MiRecordInspector> records) throws
            Exception {
6     final MiKey jobNumberField = key("JobNumber");
7     // Extract all distinct values of the field "JobNumber"
8     // in the set of records
9     final MiMap<MiKey, MiSet<McDataValue>> occuringValues =
10        McDataModelUtil.extractValues(records, jobNumberField);
11
12    // Given that set of job numbers, populate the data
13    // cache "JobHeader".
14    getDataHost().getCache("JobHeader")
15      .populate(occuringValues.getTS(jobNumberField));
16  }
```

With the combination of Listings 4.15 and 4.16, we have a full implementation of a data model extending the time sheet table to include two read-only fields each reflecting the job description and the popup1 associated with the job that might be specified on each time sheet line. Because the framework automatically invokes the `refreshVariables` method whenever needed, the implementation automatically works for all cases: creations of new time sheet lines, updating a time sheet line, refreshing etc.

4.12.3 Refreshing Action States

Just as refreshing variables is needed in a number of cases (reading, maybe, being the most obvious) the action state needs to be updated for each data-carrying event. The action-states always relate to the *current record*. Hence, actions in a table-pane always related to the *current line* in the table. However, the action-states are assumed to be the same for *all records* in a pane. This means that it must be possible to compute the action states for any given record in a pane.

The action state is basically a mapping that defines whether a specific action is enabled ("clickable") or not ("grayed out.") Sometimes it can enhance the usability to enable and disable certain actions depending on the context. In other cases, better usability may be obtained by letting an action appear as enabled, and then (if invoked in a state that makes no sense) give an error-message explaining the user why this action cannot currently be executed.

The action-enabledness state covers *all* actions—not just the named actions. This means that you may control the enabledness of whether or not it is possible to `Initialize` (start creating a new record), `Update` (edit an existing record), `Delete` (deleting existing records) and `Print` (printing the current record.) Move operations are considered enabled if `Update` is enabled. Of course, only actions that have been declared as available in the specification for the container (i.e., declared in the `defineDomesticSpec` for some data model) will be considered potentially enabled. So, if you have a container that doesn't even allow `Delete`, it makes no sense to attempt to "enable" deletion: this will have no effect!

By default, the Extension Framework will consider all actions as enabled when relevant[17]. By letting your data model implement the method `refreshActions`, you can dynamically enable or disable actions.

Like many other data-model methods, the `refreshActions` receives two parameters: a `containerRunner` and *eventData*. For this method, the `eventData` has a `getResult`. However, unlike the normal event-methods, the result is not record-level data, but rather a structure defining the enabledness of the actions of this pane. The result type for `getResult` is an `MiActionStates` object. This object can be modified and queried in a "builder-style" manner. The framework will use the resulting value of this object to build the adequate return values from the container.

The `eventData` offers the following methods that are special for `refreshActions`:

Method	Remarks
`getCurrentRecord`	This method returns an optional current record: if the pane for which action states are being calculated *has* a current record (i.e., is non-empty), then the returned optional (`MiOpt`) will contain a `MiValueInspector` that represents the current record value. Otherwise a `none` object (`McOpt.none()`) will be returned. Typically, you will use the current record values to determine the enabledness of your actions.
`getResult`	Returns the result data which—in this case—is an `MiActionStates` object. By modifying this object, you can change the action state.

[17] If a pane in in the "new" state, i.e., an `Initialize` has just occurred, then attempting to enable deletion will be disregarded as nonsense: the only thing that makes sense in this state is to allow `Create`.

The `MiActionStates` type resulting from the `getResult` method offers a number of methods:

Method	Remarks
`defaultAll`	Sets *all* actions to the default enabledness state relative to the state, the pane is currently in. For example, if the pane is in INIT-mode, `Create` should be enabled, but `Update`, `Delete` and named `Actions` should be disabled. This method returns the `MiActionStates` object itself, allowing method chaining.
`defaultAdded`	This method is similar to `defaultAll`, except that it involves only actions introduced by this data-model!

`enable`	This method returns an `MiActionEnabler` object. This is itself a builder-style object, that can be used to enable one or more actions. For example:
	`MiActionStates result = eventData.getResult();`
	`result.enable().update()`
	`.delete()`
	`.actions("Submit", "RemoveAll");`
	will enable `Update`, `Delete` and the two actions `Submit` and `RemoveAll`. See below for more details.

`disable`	This method is similar to `enable` above, but returns an `MiActionDisabler` object. This is itself a builder-style object, that can be used to disable one or more actions. For example:
	`MiActionStates result = eventData.getResult();`
	`result.disable().update()`
	`.delete()`
	`.actions("Submit", "RemoveAll");`
	will disable `Update`, `Delete` and the two actions `Submit` and `RemoveAll`. See below for more details.

`initialize`	This method returns a `MiEnablednessChecker` that can be used to determine whether `Initialize` is currently enabled or not. For example:
	`MiActionStates result = eventData.getResult();`
	`if (result.initialize().isEnabled()) {`
	`// Initialize is enabled`
	`} else {`
	`// Initialize is disabled`
	`}`
	See below for more details.

Method	Remarks
`create`	Similarly to the `initialize`-method above, this method returns a `MiEnablednessChecker` that can be used to determine whether `Create` is currently enabled or not.
`read`	Similarly to the `initialize`-method above, this method returns a `MiEnablednessChecker` that can be used to determine whether `Read` is currently enabled or not.
`update`	Similarly to the `initialize`-method above, this method returns a `MiEnablednessChecker` that can be used to determine whether `Update` is currently enabled or not.
`delete`	Similarly to the `initialize`-method above, this method returns a `MiEnablednessChecker` that can be used to determine whether `Delete` is currently enabled or not.
`print`	Similarly to the `initialize`-method above, this method returns a `MiEnablednessChecker` that can be used to determine whether the "Print..." action is currently enabled or not. *This method should not be confused with* ***`printThis`*** *(see below.) It involves the* ***`Print...`****-"action", which merely may re-direct to a card-style pane offering the possibility of doing batch printing. The* ***`Print`****-event relates to the* ***`printThis`*** *method!*
`printThis`	Similarly to the `initialize`-method above, this method returns a `MiEnablednessChecker` that can be used to determine whether `Print` is currently enabled or not. *Notice the difference between* ***`print`*** *(above) and* ***`printThis`****!*

`action`	Similarly to the `initialize`-method above, this method returns a `MiEnablednessChecker` that can be used to determine whether a specifically named `Action` is currently enabled or not. For example:
	`MiActionStates result = eventData.getResult();`
	`if (result.action("SubmitTimeSheet").isEnabled()) {`
	`  // submit time sheet is enabled.`
	`} else {`
	`  // submit time sheet is disabled`
	`}`

The types `MiActionEnabler` and `MiActionDisabler` have similarly named methods. The only difference is the actual effect of that method (whether it *enables* or *disables* a specific method.) The list of common methods is

Method	Remarks
`initialize`	This method sets the enabledness of `Initialize`. If the object is an *en*abler, it becomes enabled. If it's a *dis*abler, it becomes disabled.

Method	Remarks
`create`	This method sets the enabledness of `Create`. If the object is an *en*abler, it becomes enabled. If it's a *dis*abler, it becomes disabled.
`read`	This method sets the enabledness of `Read`. If the object is an *en*abler, it becomes enabled. If it's a *dis*abler, it becomes disabled.
`update`	This method sets the enabledness of `Update`. If the object is an *en*abler, it becomes enabled. If it's a *dis*abler, it becomes disabled. Update also controls those `Move` operations that may be applicable.
`delete`	This method sets the enabledness of `Delete`. If the object is an *en*abler, it becomes enabled. If it's a *dis*abler, it becomes disabled.
`print`	This method sets the enabledness of the `Print...`-"action." If the object is an *en*abler, it becomes enabled. If it's a *dis*abler, it becomes disabled. This should not be confused with the `printThis` method (see below!)
`printThis`	This method sets the enabledness of `Print`. If the object is an *en*abler, it becomes enabled. If it's a *dis*abler, it becomes disabled. This should not be confused with the `print` method (see above!)

`action`	This method sets the enabledness of one or more `Action`s. If the object is an *en*abler, the specified actions become enabled. If it's a *dis*abler, they become disabled. For example:
	`myEnabler.action("Act1", "Act2", "Act3");`
	`myDisabler.action("Act4", "Act5", "Act6");`

`all`	This method sets the enabledness of *all* actions. If the object is an *en*abler, they become enabled. If it's a *dis*abler, they become disabled.

Listing 4.17 shows an example of a data-model that adds a "Submit→Approve/Reject" metaphor on top of Jobs. The idea here is that `Text20` is being used as a status-field. The actions should only be enabled if it makes sense. Hence, a Job that has not yet been submitted, or has been rejected, should be able to be submitted. A submitted job should be able to be either approved or rejected. If a job can be submitted, it cannot be approved or rejected then. The `refreshActions` is implemented in line 13. Initially, the three new actions are all marked as being disabled. This is done by the statement in line 17. Then, if a current record exists, we read the status (`Text20`) field and enable the relevant actions, depending on the status. This takes places in lines 31 and 34.

Listing 4.17: Setting Action Enabledness.

```
2   @Override
3   public MiExtended defineDomesticSpec(final MiDefine
        containerRunner) {
4     return McPaneSpec.McExtended.pane()
5       .addAction("Trifolium:SubmitJob", "Submit Job").then()
6       .addAction("Trifolium:ApproveJob", "Approve Job").then()
```

```
7         .addAction("Trifolium:RejectJob", "Reject Job").then()
8         .changeField("Text20").title("Approval Status").closed()
9         .end();
10    }
11
12    @Override
13    public void refreshActions(final MiContainerRunner.MiData
          containerRunner,
14                               final MiActionState eventData) throws
                                     Exception {
15      final MiOpt<MiValueInspector> currentRecord = eventData.
          getCurrentRecord();
16      final MiActionStates result = eventData.getResult();
17      result.disable().actions("Trifolium:SubmitJob",
18                               "Trifolium:ApproveJob",
19                               "Trifolium:RejectJob");
20      if (currentRecord.isDefined()) {
21        final MiValueInspector currentJob = currentRecord.get();
22
23        final String status = currentJob.getStr("Text20");
24        final boolean isSubmitted = status.equals("Submitted");
25        final boolean isRejected = status.equals("Rejected");
26        final boolean isApproved = status.equals("Approved");
27
28
29        if (isRejected
30            || (!isSubmitted && !isApproved)) {
31          result.enable().actions("SubmitJob");
32        }
33        if (isSubmitted) {
34          result.enable().actions("ApproveJob", "RejectJob");
35        }
36      }
37    }
```

4.12.4 Pane-Level Read Data

When working with data-models, the event methods described in this chapter are *record-centric*. This means that the event methods have the event record as the focal point. It is not possible to control other records in the pane, or the value of other panes in the container! There is, however, one exception to this: when a `Read`-event occurs, the data-model is, as usually, invoked from the framework container implementation. This implementation builds a *container value* which is the result type for events at the container level. *Just before* this container value is returned, the data-model responsible for the event in question is invoked *one last time*. The invoked method is `onReadPane`. Here, the data-model may contribute to all aspects of the *container value*.

Usually, this is not needed. Sometimes, however, you might want to make your data-model *enforce* certain conditions on the shown data. Or maybe your data-model is so advanced, that it significantly changes the data. For instance, it could merge several lines in a table into one logical line (so called "emulated long texts.")

In such cases, the `onReadPane` method comes into action. You may think of the `onReadPane` as a convenient (and controlled!) way of operating at the container-level rather than at the data-model level. A lot of the boiler-plate code that would be needed in a container-level implementation has just been taken care of.

Listing 4.18 shows a complete listing of a data model making use of the `onReadPane` method to ensure that when a job budget is read, the budget type will automatically be set to a budget type specified in the constructor. In lines 56–69 two data-model factories are defined. These can then be used by a containers `defineConfiguration` method. The implementation of `onReadPane` is found in lines 20–54. The current value of the field `ShowBudgetTypeVar` is looked up. If it is not the desired one, the container is programmatically updated by setting this field to the desired value (lines 38–44), and the resulting value is merged into the current container response (line 49), overriding the pane values for the card and the table.

Listing 4.18: Using `onReadPane` to fix budget type.

```
2  public class FixedBudgetDataModel extends
       McAbstractExtendedDataModel {
3    private static final MiKey SHOW_BUDGET_TYPE_FIELD = key("
       ShowBudgetTypeVar");
4    private final String fixedBudgetType;
5    private static final MiKey doNotUpdateParameter = key(
       FixedBudgetDataModel.class.getName() + ":do_not_update");
6
7    protected FixedBudgetDataModel(final MiDataModelFactory.
       MiResources resources,
8                                   final String fixedBudgetType) {
9      super(resources);
10     this.fixedBudgetType = fixedBudgetType;
11   }
12
13   @Override
14   public MiPaneSpec.MiExtended defineDomesticSpec(final MiDefine
       containerRunner) {
15     return McPaneSpec.McExtended.pane()
16         .changeField(SHOW_BUDGET_TYPE_FIELD).closed().end();
17   }
18
19   @Override
20   public void onReadPane(final MiReadPost containerRunner,
21                          final MiEventData.MiReadPane eventData)
                              throws Exception {
22     // Unless the doNotUpdateParameter is specifically set to true
```

```
23      if (!McBool.of(containerRunner.getParameters().getValues().
           getElseTS(doNotUpdateParameter, McBool.FALSE))) {
24        final MiContainerAdmission resultContainer = eventData.
             getResult();
25        final MiOpt<MiPaneInspector> cardPane =
26            resultContainer.getPaneInspectorOpt(MePaneType.CARD.
                 getPaneName());
27        if (cardPane.isDefined()) {
28          final MiOpt<MiRecordInspector> cardCurrentRecordValue =
29              cardPane.get().getCurrentRecordInspector();
30          final boolean cardDefined = cardCurrentRecordValue.
               isDefined();
31          if (cardDefined) {
32            final MiRecordInspector currentCardValue =
                 cardCurrentRecordValue.get();
33            final MiKey popup = currentCardValue.getPopupVal(
                 SHOW_BUDGET_TYPE_FIELD).getLiteralValue();
34            if (!popup.isLike(fixedBudgetType)) {
35
36              // programmatically update the container to show
37              // the right budget type
38              final MiContainerExecutor jobBudgetCard =
                   containerRunner.executor().construct(MePaneType.
                   CARD);
39              final MiParameters doNotUpdateParameters =
                   McParameters.create().setBool(doNotUpdateParameter,
                    true);
40              final MiDataValues updateBudgetType =
41                  dataValues().setPopup("ShowBudgetTypeVar",
42                                        "JobBudgetTypeType",
43                                        fixedBudgetType);
44              jobBudgetCard.read(doNotUpdateParameters).update(
                   updateBudgetType);
45
46              // merge the update-result into the current result,
47              // using the card and table occurring from the update
48              // instead of value originally read
49              resultContainer.merge(jobBudgetCard.getContainerValue
                   ());
50            }
51          }
52        }
53      }
54    }
55
56    public static final MxFactory FACTORY_WORKING = new MxFactory("
         Original");
57    public static final MxFactory FACOTRY_CONTRACT = new MxFactory("
         Reference");
```



```
58
59    public static final class MxFactory implements
          MiExtendedDataModelFactory {
60      private final String fixedBudgetType;
61      public MxFactory(final String fixedBudgetType) {
62        this.fixedBudgetType = fixedBudgetType;
63      }
64
65      @Override
66      public MiExtendedDataModel create(final MiDataModelFactory.
            MiResources resources) {
67        return new FixedBudgetDataModel(resources, fixedBudgetType);
68      }
69    }
```

So, why would you want to change, e.g., the `maconomy:JobBudgets` container to *always* show a *fixed* job budget type? The answer is, you wouldn't. You want *some* instances to always show a fixed budget type. To achieve this, we do the following:

1. We define two new containers, `Trifolium:WorkingBudget` and `Trifolium:ContractBudget` that are both *identical copies of the* ***`maconomy:JobBudgets`*** *container, including any extension!*

2. Next, we define an extension contribution for `Trifolium:WorkingBudget` that makes use of the data-model above, enforcing that container to show only working budgets.

3. Similarly, we define an extension contribution for `Trifolium:ContractBudget` that uses the same data-model, but this time instantiated to show only contract budgets.

Listing 4.19 shows the implementation of the extension contribution for `Trifolium:WorkingBudget`: it merely refers the factory constant `FixedBudgetDataModel.FACTORY_WORKING` defined in the data-model above. The implementation of the container contribution for `Trifolium:ContractBudget` is not shown; it is left as an exercise to the reader to figure out how to make that.

Listing 4.19: A Fixed-Budget Container.

```
2  public class WorkingBudgetContainer extends
       McAbstractExtendedContainer {
3    protected WorkingBudgetContainer(final MiContainerFactory.
         MiResources resources) {
4      super(resources);
5    }
6
7    @Override
8    protected MiContainerConfiguration.MiExtended
         defineConfiguration() {
9      final MiContainerConfiguration.MiExtended configuration =
```

```
10          McContainerConfiguration.McExtended.card(
                FixedBudgetDataModel.FACTORY_WORKING);
11      return configuration;
12    }
13
14    public static final class Factory implements MiContainerFactory
          {
15      public MiContainerEvents createContainer(final
            MiContainerFactory.MiResources resources) {
16        final MiContainerEvents container = new
              WorkingBudgetContainer(resources);
17        return container;
18      }
19    }
```

Finally, let us see how these container contributions are declared in the `plugin.xml` file. Notice the first two contributions that defines the existence of the containers `Trifolium:WorkingBudget` and `Trifolium:ContractBudget`. This is done simply by "cloning" the `maconomy:JobBudgets` container! Once this is done, we can contribute the extensions to these two *specific* containers, without influencing the `maconomy:JobBudgets` at all!

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <?eclipse version="3.2"?>
3  <plugin>
4     <extension name="Extending standard containers"
5                point="com.maconomy.api.container">
6        <create container="Trifolium:WorkingBudget" id="com.trifolium.
             budgets.cloned:WorkingBudget">
7          <clone container="Maconomy:JobBudgets"/>
8        </create>
9        <create container="Trifolium:ContractBudget" id="com.trifolium.
             budgets.cloned:ContractBudget">
10         <clone container="Maconomy:JobBudgets"/>
11       </create>
12       <extend container="Trifolium:WorkingBudget" id="com.trifolium.
             Budgets.WorkingBudget" >
13          <factory class="com.trifolium.examples.spec.containers.
                WorkingBudgetContainer$Factory" />
14       </extend>
15       <extend container="Trifolium:ContractBudget" id="com.trifolium.
             Budgets.ContactBudget" >
16          <factory class="com.trifolium.examples.spec.containers.
                ContractBudgetContainer$Factory" />
17       </extend>
18    </extension>
19 </plugin>
```


4.13 Parameterizing Events

In the following sections, you have learned how to implement various events of a container. The events may react slightly differently depending on the data (e.g., deleting a "Sum/Text"-type of job budget line might be slightly different from deleting a "Time"-type of line.) But what if you want some more significant variation to your actions? And what if you want to signal that some event has been called programmatically, rather than "regularly" by an end-user?

In order to address such issues, the concept of *parametrized events* has been introduced. Any event may be associated a set of *parameters*. A parameter can be seen as a "meta-data" value. Or at least, it is a value that is not tied to the data of the event-record.

Usually, the parameter-set of an event is the empty set. But this need not be the case. Parameters may occur for two reasons:

1. The parameters has been declared in the *layout*. Currently, only `Action`-events can be parametrized from the layout.
2. An event may be generated programmatically, and thereby have one or more associated parameters.

4.13.1 Parameters from the Layout

From an MDML-layout [CMd], you can specify one or more parameters for `<Action>` references. Currently, it is only supported to parametrize `Actions` from the layout—other types of events cannot be parametrized. A parameter is a *name* and an associated *value*. In the layout, the value may be given using the `valueString` attribute, meaning that the type of the parameter value is a `String` and that the value is the literal string specified[18]. Or you may use the attribute `value` meaning that the type will be that of the specified expression. The expressions will be evaluated on the client-side using the information present in the relevant pane, and *other* panes (e.g., parent panes) if needed. The expressions will be *partially evaluated*, meaning that if there are some variable references or expressions that cannot be resolved on the client side, the expression will merely be "reduced" as much as possible. The partially evaluated expression (which is often a constant-value-expression) will be associated with the parameter of the specified name. When the action in question is invoked, the specified parameters will be associated with the event, and the event-handling methods may react based on that information. This is used in several situations, for example:

- The `ExportDataSet` ("Export to Excel") action heavily relies on parameters in order to work. The parameters specify things such as which fields should be

[18] Allowing expression using the `^{}`-syntax.

exported, what the column titles should be, and what format to export to: the action is capable of exporting to both plain text and Excel format.

- The `EmailOnAction` action heavily relies on parameters to work. The actions are used to specify things such as: what is the IP-address of the mail server? What is the mail-server port? *Which* action should *really* be run? Should output documents be sent as mail only, or should they (also) be passed on the the end-user? Should the mail be sent from the server-side, or should a mail template be created in `.eml`-format, so that it can be opened by a mail-application on the client-machine? And several other things.

- The `RunReport` action (run a Business Objects report) relies on parameters to work: it uses parameters to indicate which report to run and which report-parameters to apply to the report, as well as the output format.

Parameters are especially practical, if there is some degree of configuration to the actions. Also remember, that you can add the same action multiple times in the same layout *with different parameters*. This may allow you to let the end-user decide to use one or another variant of the same action. The result—as far as the end-user is concerned—is two different things. As an example, the `EmailOnAction` may be used to associate two different actions with the e-mail functionality. Technically, the implemented action may be the same, but the result is very much different from a user point of view!

4.13.2 Programmatic Event Parameters

When container events are made programmatically, it is also possible to associate the event with a number of parameters. And you can do so for *any* container event, not only `Action`-events.

This is especially used in cases, where your action may need to invoke itself programmatically: in order to avoid infinite loops, you can programmatically add a parameter indicating that the action should *not* call itself again. For example, suppose you have a Requisition. When the Requisition is approved0, you client wants the "Remark1" field updated with a time stamp indicating exactly when the approval took place. You can do this, but you need to programmatically update the Requisition. When the Requisition is updated, the approval is broken. If you update *prior* to approving, the Requisition is no longer submitted, and needs to be re-submitted. And if you do this, you must also do the actual approval programmatically in order to avoid a "data changed by another user"-error. Which leads to that the implementation of the "Approve Requisition" action must invoke itself programmatically. By introducing a parameter when the approval is invoked programmatically, we can avoid doing the extra update (assuming this has just been done) by examining whether a specific parameter has indeed been specified. This pattern is common in real-life extensions at customer installations.

4.13.3 Using Parameters

The parameters can be obtained from the `containerRunner` parameter, that is passed to all events. This happens through the method `getParameters`. This method returns a type, `MiParameters`. This type is basically a sub-type of `MiValueAdmission` which is explained in Section 4.3.1. This means that you can `get` and `set` parameters with specific types. For example

```
final MiParameters parameters = containerRunner.getParameters();
final int portNo = parameters.getInt("MailPort");
final int exportFormat = parameters.getStr("Format");
parameters.setBool("MyParameter", true);
```

It is important to notice, however, that treating the parameters as constant values (using the `get`*`Type`*)-methods *only works if the parameters are truly constant expressions!*. This may obviously be a requirement from your extension. If a given parameter may be a (non-constant) expression, you can access the value of the parameter, but you must give a so-called *evaluation context*. An evaluation context may contribute values for variables in the expression, and may event contribute functions.

For this reason, the `MiParameters` type contains a number of additional methods.

Method	Remarks
`get`*`Type`*	All the typed `get`-variants are found in an overloaded method that takes an evaluation context as an argument. The returned value is the value of the underlying expression evaluated in this evaluation context.
`get`*`Type`*`Opt`	All the typed `get-Opt`-variants are found in an overloaded method that takes an evaluation context as an argument. The returned value is the value of the underlying expression evaluated in this evaluation context, or `none` if no such parameter exists.
`get`*`Type`*`OrElse`	All the typed `get-OrElse`-variants are found in an overloaded method that takes an evaluation context as an argument. The returned value is the value of the underlying expression evaluated in this evaluation context, or the specified default value if no such parameter exists.
`getExpr`	Returns the value of the parameter with a specified name, as an `MiExpression` rather than a value.

Method	Remarks
`getExprOpt`	Returns the value of the parameter with a specified name, as an `MiExpression` rather than a value. If no parameter exists with the specified name, a `none` value is returned.
`setExpr`	Sets the value of a parameter with a specified name to a specific expression. If the parameter already exists, its value is overridden.
`asParametersByNameSpaceCopy`	Returns a new parameter set where all parameters that begins with a specified name-space are found with the same value as in `this` parameter set. Only, the specified name-space is stripped off! A name-space is assumed to be of the form `[a-zA-Z][a-zA-Z0-9_]*:`, for example: `SubmitTimeSheet:`, `MyNameSpace:` or `Trifolium:`. Parameters with several nested name-spaces will only be stripped of the top name-space. Notice that the returned parameters object contains *copies* of the specified parameters.

If you need to finally evaluate expression in some context (e.g., evaluate some parameters after having done a certain part of an action), you must provide a `MiEvaluationContext`. Such a context can be built using the `McEvaluationContext` class and the builder method that it provides. There is, however, a convenient way of creating an evaluation context that matches what you typically want when it comes to evaluation parameters. This is the type `McParameters.McContextBuilder`. This has a factory method, `create` which is found in two flavors:

- A variant without parameters. This just returns a builder for an initially empty evaluation context, i.e., an evaluation context that knows the value of nothing.
- A variant that takes a `MiValueInspector`, such as a record value. This method will create an evaluation-context builder which is capable of resolve the value of any field that exists in the provided value inspector with its corresponding value. Only, the variable that can be resolved is prefixed with "`result.`." For example, if the record contains

  $$\texttt{PurchaseOrderNumber} = 123456, \texttt{SupplierNumber} = \texttt{'201008'}$$

  then the evaluation context being created can resolve the following values:

  $$\texttt{result.PurchaseOrderNumber} = 123456, \texttt{result.SupplierNumber} = \texttt{'201008'}$$

This is useful if you need to bind expression variables that cannot be evaluated finally on the client side, must must be evaluated based on some kind of "result" during the action execution.

The returned evaluation context builder has a number of methods that you may use:

Method	Remarks
`add`	Which takes a prefix and a `MiValueInspector`. The evaluation context will be augmented to contain the fields/values contained by the value inspector, but—like in the case with the `create` method above— with a specific prefix. Instead of the default `result`-prefix, you can specify your own. This may be needed either if the name `result` for some reason seems off, or in case you need to have parametrizations from several different sources.
`build`	This returns a `MiEvaluationContext` with the capabilities specified for the builder until now.

4.14 Other Container Events

By now, we have covered all data-carrying events as well as the supportive event `defineSpec`. There are, a few more supportive events.

4.14.1 Open and Close Events

When a container is going to be used, it must first be "opened." After using the container, it must be "closed." For normal operations, the Extension Framework will automatically open and close the containers it needs to address. When programmatically accessing containers, you will need to `Open` and `Close` these.

The `Open`-event exists to notify a container that it's going to be used for one or more operations. The `Close`-event similarly exists to let the container know that it is no longer needed. The `Open`-event may be used to allocate needed resources, setting up connections etc. For exactly this reason, it is vital that if a container is `Open`ed, it is also `Closed`; the allocated resources can be released on the Close event.

Currently, implementing the `Open` and `Close` events is not optimally supported, and it is expected to be changed in the near future. If you *must* implement these methods, please consult Deltek Engineering first!

4.14.2 Restrict Events; Modifying Searches

When using the workspace client, searching frequently takes place: every time you type something in a field decorated with a magnifying glass or a drop-down button, a search is invoked as soon as you pause for a while. This feature is known as "search-as-you-type." Generally, this is just a way of doing what is generally called a "foreign-key search" or a "Ctrl+G" search.

In a naïve implementation, such a foreign-key search for something, would just find all "somethings." Often, this is far from desirable. What is frequently needed is a certain *foreign-key condition*. A foreign-key condition is a specific search-restriction that is being applied when searching, without the user explicitly being aware of this. For example, a foreign-key condition could be "`JobHeaders` where `Blocked` = `false`." While this is a perfectly valid and possible foreign-key condition, often you want something even more elaborate. You need the foreign-key condition to *adapt* to the data you are currently editing.

For example, suppose you want to enforce that the only employees who can be appointed `ProjectManager` for a job are those associated with the same `CompanyNumber` as the job. This means that when you edit a job belonging to `CompanyNumber` "1," only employees of company 1 may be assigned as project manager. But when you have a job belonging to `CompanyNumber` "2," only employees of company 2 may be assigned as project manager. This cannot be expressed with a static foreign-key condition—you need dynamically generated foreign-key conditions.

This is where the `Restrict` event occurs. When a search is taking place, this is basically a `Read` event of some filter pane in some container. Notice, that it is usually a completely different container than the one you are making the search from! For example, searching for a project manager in the `maconomy:Jobs` container will lead to a `Read` event in the container `maconomy:Find_Employees`! When you add or change foreign-keys using the `defineDomesticSpec`, you can specify the name of an associated container; *that container* is the one that is invoked when a search is made using that foreign-key (see Section 4.2.3.) *Before* the `Read` event occurs, *the container from where you are searching* (called the *host*) will be invoked: the `Restrict` event occurs. This gives the host the opportunity to identify a restriction that must be applied to the search. Part of the event-data is information about the *uncommitted* data which currently resides in the host pane!

Figure 4.17 shows the steps involved when a search is taking place. Notice that the search may have some restricitons imposed by the user (e.g., "`CustomerName` starts with 'Bri'" or `Blocked=false`.) A `Restrict` event then takes place in the *host* container. The resultin foreign-key condition (which may simply be `true` to indicate that no further restrictions is applied) will be "merged" with the user restrictions. Merging means that the foreign-key condition is "and'ed" with the user restrictions. Using the combined restriction, a `Read` event takes place on the *search* container, and the result from that

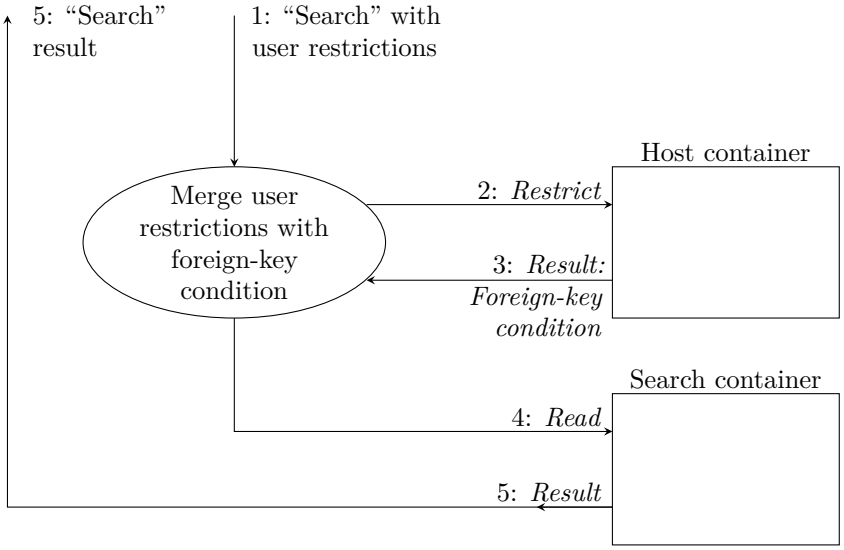


Figure 4.17: When a search is taking place, the restrictions imposed by the user is merged with the foreign-key condition obtained from the "host" container before the "search" container is read.

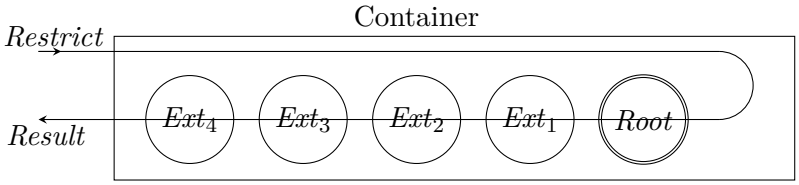


Figure 4.18: The `Restrict` container-event method is a supportive event. The container must be able to produce an expression, known as a foreign-key condition. The default foreign-key condition is `true` (i.e., all records for which `true` is true, that is: all records.) Notice that the event starts from the Root and goes through the extensions from there.

event is the search result.

By now, we know about `Read` events. Let us take a close look at the `Restrict` event. The `Restrict` event *is not data-carrying*, and it does not have "Pre" and "Post" scripts. This is shown in Figure 4.18. Notice that the process starts at the root level.

The `Restrict` event is supported by data models. Figure 4.19 shows the life-cycle of `Restrict` events using data-models. Like other events, the event method `onRestrict` takes two parameters: `containerRunner` and `eventData`. The `eventData` method offers a number of methods that are specific for this particular event:

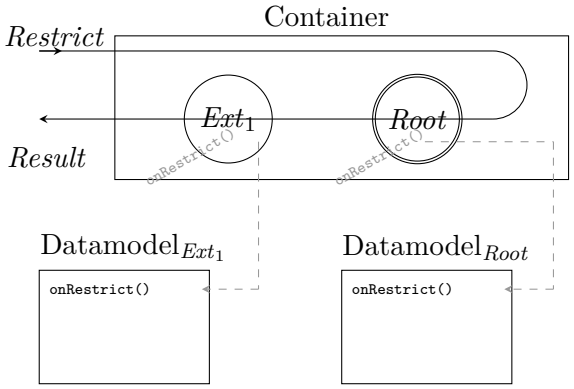


Figure 4.19: The life-cycle for the `Restrict` event implemented using data-models. For both root and extension contributions, the method `onRestrict` method is invoked from the container's `onRestrict` method.

Method	Remarks
`getForeignKeyName`	This method returns the name of the foreign-key for which a foreign-key condition should be returned.
`getRestrictionValues`	This method returns a `MiValueInspector` specifying the value of the current *uncommitted* fields in the host pane. Hence, if the user has modified two fields (but not "Saved",) and then invokes some search, the modified values will be reflected. Fields that have not been edited by the user will reflect the original value.
`getQueryExpressionAdmission`	This method returns an object of type `MiQueryExpressionAdmission`. This object reflects the current foreign-key condition. You can modify and inspect the foreign-key condition through this object.

The `MiQueryExpressionAdmission` type contains a few interesting methods:

Method	Remarks
`and`	This method can be used to "and" an additional condition to the foreign-key condition. Usually, you will want to use this.
`clear`	Clears the entire foreign-key condition, leaving it as a `true` expression ("show all.")
`andContainerSpecific`	This method is for advanced use only, and should be used only for root contributions. It adds a condition in a form that isn't represented in a normal expression. This might be used to communicate with back-ends that are not able to communicate enforced foreign-key conditions in a way that can be translated to an expression, i.e., by using some kind of token.
`toQueryExpression`	This method converts the current content of this object to a `MiQueryExpression`. Using this object, you can extract an `MiExpression` for further inspection, as well

dedicate a specific option list to contain the value of expense reasons. The field `Remark1` on the expense sheet lines is used to contain the reason.

The `onRestrict` event can be implemented like this:

```
public void onRestrict(final MiRestrict containerRunner, final
    MiEventData.MiRestrict eventData) {
  if (eventData.getForeignKeyName().equalsTS(
      FK_EXPENSE_REASON_OPTION)) {
    eventData.getQueryExpressionAdmission().and(
        EXPR_OPTIONLIST_EXPENSE_REASON);
  }
}
```

Notice the check concerning which foreign key/search key is in scope. Only if it is the one matching the constant `maconomy:FK_EXPENSE_REASON_OPTION`, do we apply the desired restriction. It is an important point, that searches are not done *from a field, but rather via a foreign-key or a search key!* If a given foreign key contains more than one field, then a specific foreign-key restriction should be applied *no matter* which *field the search was initiated from*. Sometimes, a field may be associated with *several* foreign keys/search keys. In this case, it makes little sense to know that a search was initiated from that field: we need to know *which* search. You may affect the priority of which foreign key/search key is being used from a specific field. Please refer to Section 4.2.1 for more information.

The code snippet above assumes that the field `Remark1` has been search-enabled, and that the `maconomy:Find_TheOption` is used for searching from within this field. Listing 4.20 shows the entire implementation of all the things that should be done:

- The field `Remark1` should have its default title changed, and should be specified as having "on-demand" (drop-down-like) search. This is done in the `defineDomestic-Spec` method.
- A new foreign key or search key involving the field `Remark1` should be specified. It should reference `TheOption`, using the `maconomy:Find_TheOption` container as search container. In this case, we must use a *search key* because we cannot reference all key fields from the option list. The foreign-key condition produced by `onRestrict` will ensure that the search is adequately delimited. In the example, this takes place in line 17.
- The `onRestrict` (lines 37–41) method must be implemented, restricting the searches for option lists such (made using the new search-key) that only options in the dedicated option list is presented.
- The `onChange` (lines 24–34) event should be implemented in order to ensure that the user does not "manually" (i.e., without selecting an option from a search-result) enter a reason that is not specified as an option in the expense-reason option list.

In the example, lines 3–7 declares a number of constants that makes the code more easily

maintainable and readable.

Listing 4.20: Restricting Values Using an Option List.

```
2  public class OptionListDataModel extends
       McAbstractExtendedDataModel {
3    private static final MiKey FIELD_EXPENSE_REASON = key("Remark1")
       ;
4    private static final MiKey FK_EXPENSE_REASON_OPTION = key(
       FIELD_EXPENSE_REASON.asString() + "_TheOption");
5    private static final McStringDataValue
       EXPENSE_REASON_SETUP_OPTION_LIST = McStr.val("
       Trifolium_Expense_Reason_Setup");
6    private static final MiValueInspector OPTIONLIST_NAME =
       dataValues().setVal("OptionListNumber",
       EXPENSE_REASON_SETUP_OPTION_LIST);
7    private static final MiExpression<McBooleanDataValue>
       EXPR_OPTIONLIST_EXPENSE_REASON = OPTIONLIST_NAME.asExpression
       ();
8
9    public OptionListDataModel(final MiResources resources) {
10     super(resources);
11   }
12
13   @Override
14   public MiExtended defineDomesticSpec(final MiDefine
       containerRunner) {
15     return McPaneSpec.McExtended.pane()
16       .changeField(FIELD_EXPENSE_REASON).title("Reason").
           onDemandSearch().then()
17       .addSearchKey(FK_EXPENSE_REASON_OPTION,
18                     "Expense Reason",
19                     "maconomy:Find_TheOption").link(
                         FIELD_EXPENSE_REASON, key("Name"))
20       .end();
21   }
22
23   @Override
24   public void onChangePost(final MiChangePost containerRunner,
       final MiUserChange eventData) throws Exception {
25     final MiDataValues resultData = eventData.getResultData();
26     if (eventData.getUserData().changed(FIELD_EXPENSE_REASON)) {
27       final MiExpression<McBooleanDataValue> checkValidOption =
28           McExpressionUtil.and(EXPR_OPTIONLIST_EXPENSE_REASON,
               dataValues().setVal("Name", resultData.getVal(
               FIELD_EXPENSE_REASON)).asExpression());
29       containerRunner
30         .check(getDatabaseApi().mcount("TheOption").where(
             checkValidOption).getResult() > 0)
31         .error(McMsg.msg("Please choose a value from the list.",
```



```
32                          FIELD_EXPENSE_REASON));
33      }
34    }
35
36    @Override
37    public void onRestrict(final MiRestrict containerRunner, final
          MiEventData.MiRestrict eventData) {
38      if (eventData.getForeignKeyName().equalsTS(
            FK_EXPENSE_REASON_OPTION)) {
39        eventData.getQueryExpressionAdmission().and(
              EXPR_OPTIONLIST_EXPENSE_REASON);
40      }
41    }
42
```

The above example statically applies a specific foreign-key condition to a specific foreign-key search. Sometimes, the foreign-key conditions need to be *dynamic*, i.e., dependent of the data in the host pane (the pane from where the search is initiated.) Suppose that your client wants you to make the following: when selecting a project manager for a job, it must only be possible to enter employees belonging to the same company as the job belongs to.

In this case, we need to build the foreign-key condition *depending on the uncommitted job data*. Hence, if the user has entered company "2" then the search for project managers should *only* show employees of company 2. However, changing the company to "1" should imply that searches for employees should only reveal employees of company 1. In this case, the implementation of `onRestrict` could be something like:

```
public void onRestrict(final MiRestrict containerRunner,
                       final MiEventData.MiRestrict eventData)
                          throws Exception {
  final MiValueInspector restrictionValues =
      eventData.getRestrictionValues();
  if (eventData.getForeignKeyName().isLike("
      ProjectManagerNumber_Employee")) {
    eventData.getQueryExpressionAdmission()
      .and(restrictionValues.copyValues("CompanyNumber").
          asExpression());
  }
```

Notice the use of the `McExpressionUtil`.`createExpression`. This method makes an expression from a set of values. As argument, it gets the sub-set of fields from the uncommitted data, comprising only the field `CompanyNumber`.

In a real-life example, we might additionally want to implement a check in `onChange` ensuring that the user does not manually enter an employee that is not allowed.

Chapter 5

Container Call-backs

In this chapter we shall have a look at the *call-back events* that can occur from a container. Call-back events are events that are initiated programmatically with the purpose of interacting with the client-side/user.

In the previous chapter, a few examples has made use of call-backs. Until now, this has been unexplained. In this chapter, you will learn how to initiate a call-back, what the run-time model for call-backs is and how to make extensions that interact with the call-backs on behalf of the user. This includes modifying, changing and "ignoring" call-backs.

5.1 The Call-Back Mechanism

Let us start out by asking: "why do we need call-backs at all?" The main reason is increased usability. The secondary reason is that sometimes, call-backs are really *needed*. Let us have a look at the different *kinds* of call-backs that can be made:

Message call-backs give some kind of message to the end-user. There are three kinds of message call-backs: Errors, Warnings and Notifications.

Progress call-backs are used to communicate progress to the end-user. For long-running tasks, it can be highly desirable to communicate that work is being done, is progressing and giving the user some kind of idea of when the operation can be expected to be finished. Visually, progress call-backs will display a progress bar and a status message in the workspace client.

Document call-backs are used to communicate documents to and from the end-user. For example, presenting a PDF with a printed invoice is implemented by making a document call-back, requesting the client to show a given document. In some

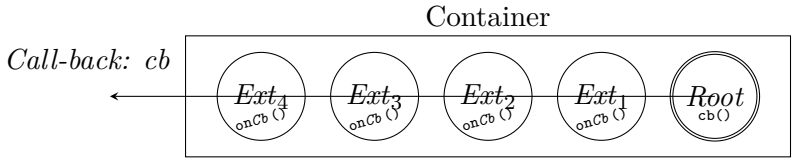


Figure 5.1: The event flow of procedural call-back events. In this example, the call-back, `cb`, is initiated in the Root contribution. It might just as well be initiated in any of the extension contributions.

cases, the logic needs some kind of input file that the user selects. This is also a document call-back, this time requesting a file from the user.

Miscellaneous call-backs In the current version, a specific call-back instructs the client to render itself in "test-mode."

5.1.1 The General Call-Back Event Flow

When a call-back is initiated, it is initiated in some container contribution. This may be the root contribution or some extension contribution. Most call-back events are *procedural*. This means that there is no "return value" from the call-back. The call-back is made. Period. A few call-backs are *functional*, which means that the call-back results in a return value. The flow is a little different in the two cases. Figure 5.1 shows the general flow for procedural call-backs. In the shown example, the call-back is initiated in the *Root* contribution. It might as well be initiated in any of the extension contributions. Only contributions between the initiating one and the user will be notified about the call-back. Once a call-back, `Cb`, is initiated the "next" contribution (i.e., the contribution towards the client side/user) will be notified. This happens by invoking the method `onCb` in that container. This goes on until there are no more extension contributions. At that time, the resulting call-back will be invoked in the client. As usual, this event will by default delegate to a corresponding method in the data-model. This is shown in Figure 5.2. Notice that the contribution from where the call-back is initiated *does not* have its call-back handling method (`onCb`) invoked.

5.2 Message Call-Backs

The message call-backs cover the following specific call-backs:

`Error` call-backs which provides a nice error message to the end-user indicating that some condition is not fulfilled. For example that the cost price cannot be negative, or that a reference to a non-existing employee is attempted. The `Error` call-back is probably the most widely used.

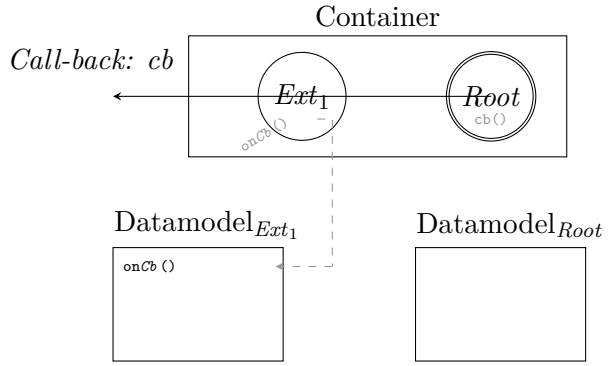


Figure 5.2: The event flow of procedural call-back events. In this example, the call-back, `cb`, is initiated in the Root contribution. The call-back event-handling method, `onCb` is only invoked in the "next" contributions. I.e., the contribution that initiates a call-back does not have its `onCb` method invoked.

`Warning` call-backs which provides a message to the end user. The user may either choose to cancel the operation (in which case the transaction is rolled back and the control flow is terminated—just like for error call-backs.) Or the user may choose to continue, in which case the transaction continues without further ado. The `Warning` call-back should be used with great thought. The reason is that while the use decides whether to continue or cancel the operation, the entire transaction is paused. If some data-base operations have been initiated and locks have been taken, these locks will be maintained until the user takes a decision. This may lead to performance issues for other users. For exactly this reason, `Warnings` will automatically time-out after 5 minutes (defaulting to Cancel) in the workspace client.

`Notification` call-backs which provides a nice informative message to the end-user. The information is shown on the client side, and the control flow continues.

`Fatal` call-backs which provides no-so-nice error messages to the end-user. Basically, fatal errors should never occur! If they do, you should be prepared to receive an error report; it means that your code is wrong or that some database invariant has been violated! The message shown to the end-user is therefore more of an internal "debug-like" nature, and will include a stack-trace that helps you identify the source of the error.

For `Error` and `Warning` call-backs, you may optionally indicate the name of a field which should take focus. In case of `Warning`, the field focus is only taken into account if the user cancels the operation.

In many cases, requesting field-focus in relation to error-messages or warnings highly improves usability. For example, suppose that you specify an error message saying "The amount cannot be negative." The user gets this error and wonders which amount is being

referred to. Of course, a better error message would help. But even a message like "The cost price cannot be negative" may be annoying to the end user. Suppose the user has edited a number of fields. The field currently having focus is probably not the "Cost Price" field. Now when the message appears, the user will have to identify the Cost Price field, put focus in it and change the value. It is much more convenient if the client automatically puts focus in the field in question.

5.2.1 Invoking Message Call-Backs

All message-like call backs are triggered based on some condition. The general way of thinking is:

> Check that some condition is true. If this is *not* the case, I want a specific message/error/warning.

In the Extension Framework the way to invoke a message is basically a two-step process: first you invoke a check. This returns an object that lets you do an error, a warning or a notification. The error/message/warning is, however, only processed if the condition was true. Otherwise, nothing happens.

```
containerRunner.check(!userData.getStr("JobNumber").isEmpty())
               .error("You must specify a Job");
```

Hence, fist the method `check` is invoked on the `containerRunner`. This returns an object on which you can specify an error or another message call-back. If the condition passed to the `check` evaluated to `true`, then doing so does nothing at all. Otherwise, an error (or whatever message call-back is requested) occurs. You may store the result of the `check` method in a local variable if you wish. The expression passed to the `check` method is evaluated *immediately* when the check-method is applied—not when the corresponding call-back is requested! The return type of the `check` method is `MiContainerChecker`. Usually, you don't store this type in a local value. Instead you directly invoke one of the following methods:

Method	Remarks
`error`	This error displays a (nice) error-message to the end-user, rolls back the transaction and terminates the control flow. This method comes in a number of flavours: the value argument can be of a number of different formats: **`String`** The string content will be displayed in the error message. **`MiMsg`** The `MiMsg` type comprises a text message and a focus field. If a field with the specified name is visible in the pane in which the error occurs, that focus will get focus. Otherwise, a focus change is not performed. You can construct a `MiMsg` by using the `McMsg` class, which provides a static method, `msg`. By using `static` imports, this may let you simply write `msg(...)` rahter than `McMsg.msg(...)`. **`MiText`** The `MiText` type embeds textual information that should be presented to an end-user. This is similar to the `String` variant. Unless you inline your messages, you should use this variant or the `MiMsg` variant. In order to obtain a localized message, you should use a `term` method to obtain the text, see Section 7.10. **Parameterized** versions of the above methods. Just like you can invoke container-events with parameters, you may invoke the call-backs using parameters. E.g., `error("Message", parameters)`. If you do not explicitly specify a parameter set, the call-back will by default be associated with the parameters in scope (i.e., the parameters for the current event or call-back.)
`warning`	This error displays a (nice) warning-message to the end-user, letting the user decide whether to cancel (which rolls back the transaction and terminates the control flow) or continue the current operation, in which case the control flow continues as if nothing had happened. This method comes in variant similar to `error` as shown above.
`notification`	This error displays a (nice) information-message to the end-user. The control flow continues as if nothing had happened. This method comes in variant similar to `error` as shown above. No focus changes will be applied for notifications though.

Method	Remarks
`fatal`	This error displays a (non-nice) error-message to the end-user. The message is not intended for the end-user but for the programmer. If a `fatal` error triggers, it corresponds to an assertion violation. The error dialog shown to the user will contain a "Details" button which reveals a stack trace. The `fatal` method comes in a number of flavours corresponding to those for `error` as shown above. In addition, you may optionally specify any number of Java-objects for which you wish to see the content in case the fatal error is triggered. The content is generated by invoking the `toString` method on each object.

The examples of Chapter 4 contain several uses of error invocations. In Listing 5.1 more examples are shown. Notice that the pattern is roughly the same for `Error`, `Warning`, `Notification` and `Fatal`. `Fatal`, however, is different in that the arguments are merely just raw information-objects that we'd like to see in case the assertion doesn't hold. In the example, the assertion is that if an employee has a `LocationName` associated, then that location can be looked up. This should certainly hold! If not, we need to see the value of the `LocationName` as well as the `EmployeeNumber` in question. Notice that the information objects can be arbitrary complex data-structures. In order to gain any benefit, though, the associated objects must have a `toString` implementation that shows an adequate amount of details in a readable format. For brevity, the listing assumes that `McMsg` has been statically imported. This allows the programmer to write `msg(..., ...)` instead of `McMsg.msg(..., ...)`.

Listing 5.1: Invoking Message Call-Backs.

```
2   public void onUpdatePost(final MiUpdatePost containerRunner,
3                            final MiUpdate eventData) throws
                                Exception {
4
5     // Demonstration of various ways of doing message
6     // call-backs
7     final MiUserData userData = eventData.getUserData();
8     final MiDataValues resultData = eventData.getResultData();
9
10    // If the line is locked, and MyField is
11    // changed, give an error, putting
12    // focus in MyField.
13    containerRunner
14      .check(userData.unchanged("MyField")
15             || !userData.getBool("Locked"))
16      .error(msg("Field cannot be changed when line is locked",
17                 "MyField"));
18
```



```
19      final BigDecimal billingPrice = userData.getAmount("
            BillingPrice");
20      final BigDecimal cost = userData.getAmount("CostPrice");
21      // If the billing price is lower than the
22      // cost, give a warning. If user
23      // cancels, put focus in BillingPrice field.
24      containerRunner
25        .check(userData.unchanged("BillingPrice")
26               || billingPrice.compareTo(cost) < 0)
27        .warning(msg("Billing Price is lower than Cost Price",
28                     "BillingPrice"));
29
30      if (userData.changed("EmployeeNumber")) {
31        final McCachedData employee = dataHost.getCache("Employee");
32        final McDataValue employeeNo = userData.getVal("
              EmployeeNumber");
33        // If the user enters an employee number that
34        // does not exist, give an error with
35        // focus in EmployeeNumber field
36        containerRunner
37          .check(employee.exists(employeeNo))
38          .error(msg("Employee does not exist",
39                     "EmployeeNumber"));
40
41        final McDataValue employeeLocation =
42          employee.getValue(employeeNo, "LocationName", McStr.val(""
                ));
43        if (!employeeLocation.isNull()) {
44          // look up the description of the employee's location
45          // It is assumed that the location MUST exist
46          final McCachedData location = dataHost.getCache("Location"
                );
47          containerRunner
48            .check(location.exists(employeeLocation))
49            .fatal("Location missing", employeeLocation, employeeNo)
                  ;
50
51          resultData.setVal("LocationName", employeeLocation);
52
53          // If the derived location is different from
54          // what the user has just seen, notify
55          // the user about that.
56          containerRunner
57            .check(employeeLocation
58                   .compareTo(userData.getVal("LocationName")) == 0)
59            .notification("Location has been changed to "
60                          + McStr.of(employeeLocation));
61        }
62      }
```

Sometimes you want to *unconditionally* do some message call-back. Maybe, the code structure is such that if the control flow passes a certain statement, then an error/warning condition has occurred. Or maybe you want to unconditionally show some message to the user, e.g., "The job has been copied."

In such cases, you can obviously write

```
containerRunner.check(false)
               .notification("The job has been copied");
```

However, it feels awkward to do so: you specify `false` as a condition because you *want* the message. In order to make such code more readable, you may obtain an object that will unconditionally lead to a given message-callback. This method is called `call`. Using this method, the above example could be written:

```
containerRunner.call().notification("The job has been copied");
```

Which is less awkward, more readable and less error prone. In this way you can invoke any message call-back unconditionally.

5.2.2 Reacting on Message Call-Backs

You may choose to react when a given message call-back occurs, although you can only react to call-backs that occur in contributions nearer to the root; you are never notified of call-backs occurring in contributions "before" your contribution.

Why would you want to react to a message call-back? There are several use-cases, including:

- Discard/suppress warnings. When reacting on a warning call-back, you may choose to discard the warning, continuing as if nothing has happened. This is particularly useful if you make operations that programmatically invokes a number of events of which some may invoke warnings that you *don't* want to propagate to the end-user in particular cases.
- Discard/suppress notifications. Some notifications are seen as annoying and irrelevant in some customer installations. Such notifications may be discarded.
- Turn warnings into errors. You may choose to do an `Error` callback instead of a given `Warning` callback. You can change the warning message or reuse it for the error.
- You may wish to change the message or the field focus.

Notice that you can discard `Warning` and `Notification` call-backs, but *you cannot discard `Error` and `Fatal` call-backs*: if an error occurs in some contribution, it occurs, and the operation must be aborted! Figure 5.3 shows the life-cycle of message call-backs exemplified by the `Warning` call-back.

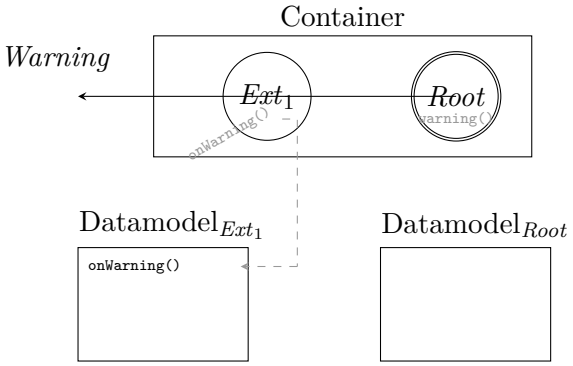


Figure 5.3: All message call-backs are procedural: even the warning! The warning does *not* return, e.g., `false` if the user cancels. Instead the control-flow is aborted. The life-cycle of the other message call-backs are similar to the one shown for `Warning`.

Listing 5.2 shows an example reacting on `Warning`s and `Error`s. Notice, that *any* warning invoked in contributions closer to the root in the container in question will result in the `onWarning` method being invoked. It is frequently the case, that a certain call-back behavior is only *sometimes* needed. You may use parameters to communicate such information. In the shown example, we check for certain parameters in order to determine whether to: discard/suppress/skip the warning, continuing as if nothing has happened. Or to make the warning into an error, using the same message. Or to change the warning message entirely. Notice the use of the method `skip` in line 13. Invoking `skip` implies that the next contributions (e.g., between this contribution and the client) are not notified about this call-back, and after the callback-handling method terminates, the control flow continues as if nothing has happened. It is not syntactically possible to invoke `skip` for `Error` and `Fatal` events.

Line 19 shows how to turn a warning into an error: you simply start a *new* call-back, in this case an `Error` call-back. Since the control-flow cannot continue after an error, there is no need to also `skip` the warning.

Line 24 is interesting: it shows how to continue the current call-back but *with modified arguments*. This is done by invoking the `next` method on the `containerRunner`. This is *different* from merely invoking a *new* warning call-back. Doing so could lead to two consecutive warnings to the user: if the first is accepted, the current warning would continue. The current call-back continues after the call-back-handling method terminates unless either `next` or `skip` have been called.

The `next` method is also used in the implementation of `onError` in line 36: here the task is to always set focus to some field, `MyField`, when an error occurs. This is done by modifying arguments of the current `Error` event.

Listing 5.2: Reacting on Message Call-Backs.

```
2     @Override
```

```
3   public void onWarning(final MiWarning containerRunner,
4                         final MiMsg message) throws Exception {
5     final MiParameters parameters = containerRunner.getParameters
          ();
6
7     if (parameters.getBoolOrElse("SkipWarn", false)) {
8       // discard the warning; the "next" contribution
9       // is not notified about the warning at all and
10      // control flow continues from where the warning
11      // was made. This corresponds to the user
12      // selecting "OK".
13      containerRunner.skip();
14    }
15
16    if (parameters.getBoolOrElse("WarnAsError", false)) {
17      // Make the warning into an error by
18      // invoking an error call-back
19      containerRunner.call().error(message);
20    }
21
22    if (parameters.getBoolOrElse("ChangeMsg", false)) {
23      // Completely change the warning message
24      containerRunner.next(msg("This is a changed warning message"
            ));
25    }
26
27  }
28
29  @Override
30  public void onError(final MiError containerRunner,
31                      final MiMsg message) throws Exception {
32    // Always put focus in "MyField" if an error occurs
33    final MiMsg myFieldFocusMsg =
34      msg(message.asText(), "MyField");
35
36    containerRunner.next(myFieldFocusMsg);
37  }
```

5.3 Progress Call-Backs

The progress call-backs are meant to control various aspects of progress indication to end-users. Progress-indication is highly useful for long-running operations. Experience shows that an operation that is perceived as “slow” by the users can be seen as “ok” by the same users, just by adding progress indication! With the progress callbacks you can:

Initialize a progress bar Hence presenting a no-progressed progress-bar with a certain initial message and the estimated number of steps to process. Currently, only one progress indicator is supported.

Update the status of a progress bar This will change the status of the progress bar, e.g., increasing the progress. You may also change the message displayed in the progress bar. You must not update the status of a progress bar unless one has been initialized and has not been closed yet.

End a progress bar Hence, "closing" the progress indication.

5.3.1 Invoking Progress Information

In order to do invoke progress information, you must do the following steps:

1. Initiate a progress bar
2. Do zero or more updates to the progress status
3. End the progress bar

In order to get a handle to communicate progress-information, you must invoke the method `progress` on the `containerRunner`. By doing this, you get access to the following methods:

Method	Remarks
`start`	Starts a progress bar. This method must be invoked prior to invoking `step`. Only one open progress bar is supported. This method comes in a number of flavours, but common for all is that a status message (in the form of a `String` or a `MiText`) must be provided. In addition, you may optionally provide the estimated total number of steps that will be undertaken during progress feedback. Finally, you may provide a set of parameters. If you do not explicitly specify a parameter set, the call-back will by default be assoicated with the parameters in scope (i.e., the parameters for the current event or call-back.) This method returns an object of the type `MiContainerProgresser.MiProperties` which may be used to control the progress behavior in a slightly advanced way. Typically, however, the easiest is to let this object alone. See page 166 below for more information on advanced progress management.

Method	Remarks
`step`	When this object is called, the progress indicator is updated. In normal use-cases, the `step` method will increase the progress with one step out of the total amount of steps declared for the `start` callback. If the total number of steps has not been defined, the progress indicator will only be advanced if you use advanced progress management. If the total number of steps has been defined, the progress indicator will be advanced by one step, unless you have been managing the progress properties manually. You may opt to change the status message upon stepping. Doing so will simply update the displayed message in the progress indicator. You may choose to specify a set of parameters to be associated with this call-back. If you don't, the parameters currently in scope (i.e., the parameters for the current event or call-back) will be used.
`end`	This method terminates the progress indicator. You should not start more than one progress indicator at a time, and you must `end` on that has been started!

The typical usage for progress indications is when your logic iterates over something in cases where each step might take some time. Or it could be a combined operation comprising a number of steps, and where each step takes a noticeable amount of time.

Listing 5.3 shows an example implementation of an action that submits and approves a number of time sheets. The selection of the time sheets and the actual submitting and approval is not shown, since this is not the important part here. What *is* important is how progress indication is used to give feedback to the end-user that the operation is running and progressing. In line 9 the progress indicator is started with a status message. Also notice that the estimated number of total steps is provided as information. In this case, the number of time sheets that has been selected for submission and approval. Following that, we loop over every selected time sheet and submits & approves each of them. For each of these time sheets, the progress indicator should be updated with one step. This is done in line 12. Since no status message is provided, the current status message is left unchanged. Finally, after the loop, the progress indicator is removed in line 15 by invoking the `end` method. Notice that the `step` method is invoked before doing the main operation. Whether to do this before or after is a matter of taste. You should notice a couple of things:

- If you update the progress *after* an operation (e.g., indicating what has already

been accomplished) the end-user will rarely experience the progress-bar going to 100, since it is closed immediately after the last (100%) step.

- The current progress-indicator implementation in the workspace client attempts to "animate" the progress bar. And this takes a while. So, for example, if you have two steps, then step 1 will lead to an animation from 0% to 50%. The second step will lead to an animation from 50% to 100%. Since the animation takes some time, the progress indicator will likely be closed in good time before the animation is done if the update is the last thing happening before the progress indicator is ended.

Listing 5.3: Invoking Progress Call-Backs.

```
2   public void onAction(final MiActionPost containerRunner,
3                        final MiAction eventData) throws
                            Exception {
4     final MiSet<MiKeyValues> selectedTimeSheets =
5         getTimeSheetsToSubmitAndApprove(containerRunner);
6
7     final int totalSteps = selectedTimeSheets.size();
8     if (totalSteps > 0) {
9       containerRunner.progress().start("Automatically submitting
            and approving time sheets",
10                                       totalSteps);
11      for (final MiKeyValues timeSheet : selectedTimeSheets) {
12        containerRunner.progress().step();
13        submitAndApprove(containerRunner, timeSheet);
14      }
15      containerRunner.progress().end();
16    }
```

In the example above, the status message displayed by the progress indicator was static. You may want to update the status message to indicate either a "phase" or some kind of counter. Listing 5.4 shows an implementation that is almost identical to that in Listing 5.3, except that the status message is updated to let the user know exactly how many time sheets have been processed. The difference is found how the `step` method is invoked in line 13: this time it provides a status message rather than keeping the current message.

Listing 5.4: Updating Status Messages During Progress Indication.

```
2   public void onAction(final MiActionPost containerRunner,
3                        final MiAction eventData) throws
                            Exception {
4     final MiSet<MiKeyValues> selectedTimeSheets =
5         getTimeSheetsToSubmitAndApprove(containerRunner);
6
7     final int totalSteps = selectedTimeSheets.size();
8     if (totalSteps > 0) {
9       int currentStep = 0;
```

```
10          containerRunner.progress().start(getCurrentStatus(
                containerRunner, totalSteps, currentStep),
11                                              totalSteps);
12          for (final MiKeyValues timeSheet : selectedTimeSheets) {
13            containerRunner.progress().step(getCurrentStatus(
                  containerRunner, totalSteps, ++currentStep));
14            submitAndApprove(containerRunner, timeSheet);
15          }
16          containerRunner.progress().end();
17        }
18      }
19
20      private String getCurrentStatus(final MiActionPost
            containerRunner,
21                                      final int totalSteps,
22                                      final int currentStep) {
23        return String.format("Automatically submitting and approving
              time sheets: %d of %d",
24                             currentStep,
25                             totalSteps);
```

You should notice, that the end-user may cancel an operation by "cancelling" the progress indication. If the user does so, an `McInterruptedException` is thrown. If uncaught, the operation will be aborted, and a roll-back will be made. Normally, you should not catch such an exception, and you don't have to `end` the progress indicator in this case: that will be taken care of automatically.

Advanced Progress Management

On rare occasions, you may not find it easy to estimate the number of steps or determine exactly when a full "step" has been accomplished. In such cases, you can control the progress indication more manually. In order to do this, you must access the `MiContainerProgresser`.`MiProperties` which is returned from the `start` call-back. By modifying this object, you can control in a more fine-grained way how to update the progress indicator. Modifying this object does not in itself lead to any progress indication update. However, upon the next invocation of `step`, the update takes place. If you have manually changed the *progress* part (i.e., the "percentage", not just the status message), the `step` method will not automatically change the progress "percentage"; instead the values having been provided by you will be used.

The properties object gives access to a number of methods:

Method	Remarks
`getDecimalPos`	Returns the current decimal-valued representation of the current progress. A value of 1 corresponds to 100%.

Method	Remarks
`getDefaultStepSize`	Returns the amount with which the decimal position (see above) will be increased when a normal `step` is invoked.
`getMessage`	Returns the current status message.
`getNextPos`	Returns the next step-wise position of the "next" step. I.e, the progress as an integer (compared to the number of total steps.)
`getPos`	Return the current step-wise position of the "next" step. I.e, the progress as an integer (compared to the number of total steps.)
`getTotalSteps`	Returns the currently specified total number of steps.
`setDescription`	Changes the description associated with the progress indicator. No call-back is actually made until the `step` method is invoked.
`setPct`	Changes the current progress indication value to a certain amount, expressed as a percentage (0%–100%.) The client may not be able to visualize a decreasing completion percentage! No call-back is actually made until the `step` method is invoked.
`setPosition`	Changes the current progress indication value to a certain "position", i.e., a certain number of absolute steps (compared to the currently specified total number of steps.) No call-back is actually made until the `step` method is invoked.
`setTotalSteps`	Changes the current progress indication value by changing the estimated number of total steps to be carried out. No call-back is actually made until the `step` method is invoked.

`getSubNextPos`
`getSubPos`
`getSubTotalSteps`
	These methods are intended to be called by the framework, and are intended to support a notion of "sub-steps." This mechanism is experimental and subject to change. *You should not use these method unless you have discussed this with Deltek Engineering!*

By invoking the progress-property object using the methods listed above, you are able to manually calculate, e.g., a certain percentage-of-completion, which can be used when the step-size is non-linear or if the number of progress steps cannot be easily determined.

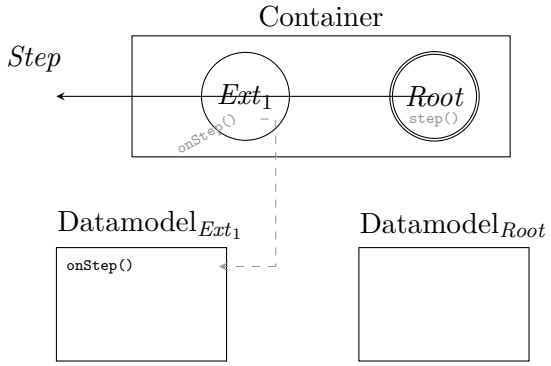


Figure 5.4: All progress call-backs are procedural. The life-cycle of the other progress call-backs are shown here, exemplified by the `Step` call-back. The other progress call-backs are similar.

5.3.2 Reacting on Progress Call-Backs

Just like you can react on message call-backs, you can react on progress call-backs. As for message call-backs, only contributions between the one invoking the call-back and the end-user will be notified. The framework will invoke the `onStart`, `onStep` and `onEnd` automatically. Figure 5.4 shows the life-cycle of progress call-backs exemplified by the `Step` call-back.

In such call-back-handling methods, you may invoke the `skip` method on the container-runner. Doing so will effectively ignore the call-back in question. Hence, to get rid of progress indication in a give situation, you can implement the above mentioned methods and merely invoke `skip`.

You may also change the progress-properties; either the description or the progress indicator.

If you merely want to change the properties of a given call-back, you should ensure that you invoke `next` rather than making a new similar call-back: doing so will keep the current call-back "alive," and it will continue once your explicit call-back has been carried out. For `onEnd` there is really nothing you can do except let the call-back through or `skip` it. For this reason, `next` is not available in this case. For `onStart`, it is only possible to modify the message, not the total number of steps; if you need to do this, you will need to either invoke a completely new `start` call-back (and `skip` the current one), or you need to use advanced progress management during `onStep`.

Listing 5.5 shows a short example where the progress behavior is being modified in this way depending on certain parameters.

Listing 5.5: Reacting on Progress Call-Backs.

```
2     @Override
```



```
3   public void onStart(final MiStart containerRunner,
4                       final MiText description) throws Exception {
5     final MiParameters parameters = containerRunner.getParameters
          ();
6     // If the SkipProgress parameter is set, don't show progress
7     if (parameters.getBoolOrElse("SkipProgress", false)) {
8       containerRunner.skip();
9     }
10
11    // Use the message communicated in ProgressText parameter
12    // if specified.
13    if (!parameters.getStrOrElse("ProgressText", "").isEmpty()) {
14      containerRunner.next(text(parameters.getStr("ProgressText"))
            );
15    }
16  }
17
18  @Override
19  public void onStep(final MiStep containerRunner,
20                     final MiProperties properties) throws
                           Exception {
21    final MiParameters parameters = containerRunner.getParameters
          ();
22    // If the SkipProgress parameter is set, don't show progress
23    if (parameters.getBoolOrElse("SkipProgress", false)) {
24      containerRunner.skip();
25    }
26
27    // If the parameter UpdateEvery10 is set then only
28    // propagate progress steps for every 10th
29    // progress position.
30    if (parameters.getBoolOrElse("UpdateEvery10", false)) {
31      if (properties.getPos() % 10 == 0) {
32        containerRunner.next(properties.getMessage(),
33                             properties.getPos(),
34                             properties.getTotalSteps());
35      } else {
36        containerRunner.skip();
37      }
38    }
39  }
40
41  @Override
42  public void onEnd(final MiEnd containerRunner) throws Exception
        {
43    final MiParameters parameters = containerRunner.getParameters
          ();
44    // If the SkipProgress parameter is set, don't show progress
45    if (parameters.getBoolOrElse("SkipProgress", false)) {
```

```
46          containerRunner.skip();
47      }
```

5.4 Document Call-Backs

Document call-backs allow you to communicate files with the user/client machine. There are three different call-backs of this type:

`Show` which allows you to show a document to the end-user. When a document is uploaded to the client-machine in this way, the operating system on the client machine is asked to "open" the document using the default application for the kind of file in question. If there is no known default programs, the OS will likely prompt the user about which application to use.

This mechanism is used, for example, for showing prints: a PDF document is generated and using this call-back, the document is opened on the user's machine.

`Save` which is similar to `Save`, except that the client will prompt the user where the document should be saved. Hence, the document is not opened, only saved.

`Load` which asks the user to provide a document of some kind. This document could be needed by the business logic (for example as some kind of import file), or it could be an image or other document (which should be placed in the Maconomy document archive.) This call-back can be configured such that the user is able to provide several documents.

5.4.1 Invoking Document Call-Backs

In order to initiate a document call-back, you must access the method `document` on the `containerRunner`. By doing so, you get access to the following methods:

Method	Remarks
`show`	This method takes a `McFileResource` and an optional parameters object as arguments. If the parameters are left out, the call-back will be associated with whatever parameters are in scope (for the event or the current call-back.) See below for an overview of `McFileResource`. The result is that a `Show` call-back occurs. When such a call-back is received in the workspace client, the client will ask the operating system to open the actual file, just as if was opened by the end-user. Hence, showing an "`.xls`"-document will likely open Excel, whereas showing a "`.pdf`" document will open the document using some PDF-reader such as Acrobat Reader.
`save`	This method is similar to `show` above, except that it results in a `Save` call-back. The workspace client will prompt the user where the file should be saved.
`load`	This method initiates a `Load` call-back. This call-back is slightly different from other call-backs in that it has a *return value*. Therefore, this particular call-back—unlike the procedural call-backs—has a "Pre" and a "Post" handling mechanism!

In order to create files that can be used with the `Show` or `Save` call-backs, you must create an instance of the class `McFileResource`. There are numerous constructors available. Common to all of them is that they represent a name, a content, a content-type and an a origin-descriptor which has no semantic use, but may be used to clarify the source of the file resource.

Constructor Arguments	Remarks
`String`	Creates a file resource from the file indicated by the `String` file name. The content of the file resource will be the content of that file , the origin-descriptor will be the absolute file path. The content type is attempted "guessed" from the extension of the file.
`File`	Similar to the constuctor above, except that you provide the `File` directly.
`String`, `byte[]`	Similar to the `String` variant above, except that the specified file *need not exist*. The actual content used is given in the `byte`-array.
`File`, `byte[]`	Similar to the `File` variant above, except that the specified file *need not exist*. The actual content used is given in the `byte`-array.

Constructor Arguments	Remarks
`String`, `byte[]`	Similar to the `String` variant above, except that the specified file *need not exist*. The actual content used is given in the `byte`-array.
`MiKey`, `MiKey`, `String`, `String`, `byte[]`	Creates a file resource with a specified base-name (`MiKey`), an specified extension (`MiKey`), the MIME content-type of the file resource (`String`), and an origin description (`String`.) The actual content used is given in the `byte`-array.
`MiKey`, `MiKey`, `String`, `String`, `String`	Similar to the constructor above except that the content is provided as a `String` rather than a `byte`-array.
`MiKey`, `McFileResource.MeType`, `String`, `String`	Creates a file resource with a specified base-name (`MiKey`.) The file type is indicated by the `McFileResource.MeType` argument, which is used to derive a standardized extension and content-type. The origin description is provided by the first (`String`.) The actual content used is given in the last `String` argument.
`MiKey`, `McFileResource.MeType`, `String`, `byte[]`	Similar to the constructor above except that the actual content used is given in the `byte`-array.

In addition to the above mentioned constructors, a number of variants exist which use a `McFileDescriptor` to provide information about the name, extension and content-type. The constructors for this class are similar to the ones stated above, except that the origin-descriptor and content is left out.

Once you have a file-resource of some sort, you can use it with the `Show` or `Save` call-backs. Listing 5.6 shows an example of doing `Show` and `Save` callbacks. The code is a snippet from a data-model introducing some kind of "Equipment" data. There are two events implemented: the `Print`-event and an `Action` event for an action called "Save Print." The implementation of the `Print` event calls a private method which generates some PDF-document based on the data in the pane (using the iText [iTe08] library.) The resulting document is then passed to an invocation of the `Show` call-back using the `show` method in line 6.

The action "Save Print" similarly invokes the PDF-generating method, but instead of showing it, we request that it is being saved by invoking the `Save` call-back using the

`save` method. This happens in line 15.

The method generating the PDF is implemented in lines 19–61. Notice how the file-resource is generated in lines 55–59.

Listing 5.6: Show and Save Call-Backs.

```
2   @Override
3   public void onPrint(final MiPrintPost containerRunner,
4                       final MiPrint eventData) throws Exception {
5     final McFileResource printOutput = generatePdf(eventData);
6     containerRunner.document().show(printOutput);
7   }
8
9   @Action("SavePrint")
10  private final class SavePrintHandler extends
        McAbstractDataModelRootAction {
11    @Override
12    public void onAction(final MiActionPost containerRunner,
13                         final MiAction eventData) throws
                             Exception {
14      final McFileResource printOutput = generatePdf(eventData);
15      containerRunner.document().save(printOutput);
16    }
17  }
18
19  private McFileResource generatePdf(final MiTransform eventData)
        throws DocumentException {
20    final MiValueInspector originalData = eventData.
          getOriginalData();
21
22    final Document document = new Document(PageSize.A4);
23    final ByteArrayOutputStream bas = new ByteArrayOutputStream();
24    PdfWriter.getInstance(document, bas);
25    document.open();
26
27
28    final String headlineText =
29      String.format("Equipment %s (%s)",
30                    originalData.getStr(FIELD_EQUIPMENT_NO),
31                    originalData.getStr(FIELD_DESCRIPTION));
32    final Paragraph headline =
33      new Paragraph(headlineText,
34                    new Font(Font.HELVETICA, 14, Font.BOLD));
35
36    document.add(headline);
37    document.add(new LineSeparator(2.0f, 100.0f, Color.BLACK, 0,
          -4.0f));
38    final StringBuilder printContent = new StringBuilder();
```

```
39      final McDateDataValue purchaseDate = originalData.getDateVal(
            FIELD_PURCHASE_DATE);
40      final String prettyPurchaseDate =
41          String.format("%02d-%02d-%04d",
42                        purchaseDate.getDay(),
43                        purchaseDate.getMonth(),
44                        purchaseDate.getYear());
45      printContent
46        .append("Purchase Date: ")
47        .append(prettyPurchaseDate)
48        .append('\n')
49        .append("Belongs to Company No. ")
50        .append(originalData.getStr(FIELD_COMPANY_NUMBER));
51
52      final Font fontNormal = new Font(Font.HELVETICA, 10, Font.
            NORMAL);
53      document.add(new Paragraph(printContent.toString(), fontNormal
            ));
54      document.close();
55      final McFileResource printOutput =
56        new McFileResource(key("Equipment"),
57                           McFileResource.MeType.PDF,
58                           "Generated by " + this.getClass().getName
                                ().
59                           bas.toByteArray());
60      return printOutput;
61    }
```

The `Load` call-back is slightly different from the other document call-backs in that it is associated with a return value: the result of this call-back is a list of documents. The call-back is configured using a `MiFileSelector` type. This type allows configuration of a default file name and/or extension as well as an option specifying whether the user is allowed to provide a multiple documents or not. The user may choose to cancel the selection of a file, in which case the result is an empty list. In order to create an instance of a `MiFileSelector`, you can use the factory methods provided by `McFileSelector`:

Method	Remarks
`create`	This factory method returns a new file-descriptor. Several flavours exist: • A zero-argument variant which creates a file-selector with no specifics regarding which kind of document to load, and which provides the capability for selecting only one document (i.e., not multiple documents.) • A variant which takes a name. The name represents a suggested file name. If the name contains a wild-card such as `*.txt`, the file-selector will attempt to show only files with this extension. It can be overruled by the end-user, though! • A variant which takes an existing file-selector and a `boolean` argument indicating whether or not multiple files may be selected. The resulting file-selector is identical to the one provided, except that the ability of allowing multiple files is as specified in this factory.

Listing 5.7 shows an example of using the `Load` call-back. Imagine an action that allows the user to select a number of documents, and upload these to some file server. In lines 8–10 a file selector is created. This file selector accepts multiple documents, and by default suggests that `.png`-documents are being selected by the user. The call-back is invoked in lines 12–13 by invoking the `load` method. If the resulting list is empty, the user is notified that no files were provided (and therefore nothing is uploaded.) The code iterates over all the provided files and calls a method that is meant to upload the file to some file-server.

Listing 5.7: Obtaining Documents From the User.

```
2   @Action("Upload")
3   private final class UploadHandler extends
        McAbstractDataModelRootAction {
4     @Override
5     public void onAction(final MiActionPost containerRunner,
6                          final MiAction eventData) throws
                             Exception {
7
8       final MiFileSelector fileSelector =
9         McFileSelector.create(McFileSelector.create("*.png"),
10                              true);
11
12      final MiList<McFileResource> files =
13        containerRunner.document().load(fileSelector);
14
15      containerRunner.check(files.size() > 0)
16                     .notification("No files selected");
```

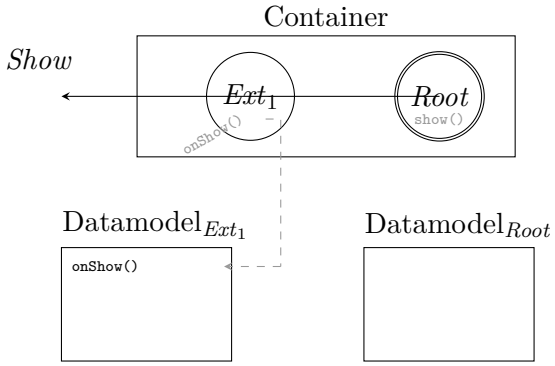


Figure 5.5: The `Show` and `Save` call-backs are procedural. The life-cycle of these two call-backs are shown here, exemplified by the `Show` call-back.

```
17
18          for (final McFileResource file : files) {
19            uploadFileToFileServer(file);
20          }
21        }
```

5.4.2 Reacting on Document Call-Backs

Just like you can react on message and progress call-backs, you can react on document call-backs. As for the other kinds of call-backs, only contributions between the one invoking the call-back and the end-user will be notified. The framework will invoke the `onShow` and `onSave` methods automatically. These two are procedural call-backs. Figure 5.5 shows the life-cycle of the `Show` and `Save` call-backs exemplified by the `Show` call-back.

The `Load` call-back is not procedural—it is functional since it provides a return value. This means that the life-cycle is different from the others: there are `onLoadPre` and `onLoadPost` methods that will be invoked prior to prompting the user for files, and *after* receiving the list of files from the user. Figure 5.6 illustrates the life-cycle of the `Load` call-back. Roughly the following takes place:

1. The contribution in front of the one invoking the `Load` call-back is notified that a `Load` call-back occurs. This implies that the `onLoadPre` is called. The container will delegate to the relevant data-model's `onLoadPre` method. This method has two parameters: `containerRunner` and `fileSelector`. The file-selector is used to specify the kind of file(s) are requested by the code. For example, it may state that the user should look for "`*.txt`" files, and whether or not multiple file-selection is enabled or not. The method *returns* the resulting file selector. The default implementation of this method (i.e., if you don't implement anything) is to return

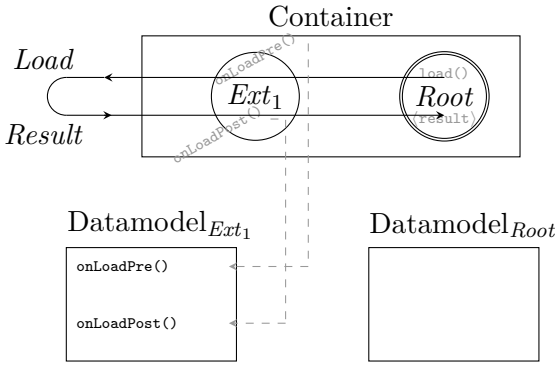


Figure 5.6: The `Load` call-back is functional and consquently has both a "Pre" and a "Post" call-back handling method. The life-cycle of the call-back is shown here.

whatever file-selector is given in the argument. You have the possibility of returning a `McOpt.none()` object, which corresponds to a file-selector with single-file selection an no specific file-selection properties.

In the `onLoadPre`-script, you may also choose to invoke the `skip` method on the `containerRunner`. Doing so will *abort* further processing of the load-event! This means that the end-user will *not* be presented with a file prompt. Likewise, contributions further to the top will also not be notified that a `Load` event is taking place. The framework will then invoke the `onLoadPost` for your contribution and the following ones until the place where `load` was invoked. In this way, your contribution can take control of `Load` call-backs and provide files on behalf of the end-user!

2. When the `onLoadPre` event has been processed for all relevant container contributions, a file-chooser dialog is presented to the end-user. The end-user can then either "Cancel" the operation (corresponding to selecting no files) or choose a number of files. It is only possible to choose more than one file if the final file-selector enables this. There is, however, nothing that prevents the user from choosing a file of some "unexpected" type!

3. After the user has selected 0 or more files, the `onLoadPost` call-back event method will be called in the first contribution. This method again has two arguments: a `containerRunner` and a `fileResource` argument, which is really a *list* of file resources. The `onLoadPost` method should in turn *return* a list of selected files. The default implementation will just return whatever list of files was given in the `fileResource` argument. You may choose to create an entirely new list of files, remove some files, add files etc. The list you return will be considered the list delivered by the end-user in the following contributions.

4. In this way, the `onLoadPost` methods will be invoked in the container contributions until the place where `load` was invoked. The resulting list will be returned by the

`load` method and the code continues as usual.

Listing 5.8 shows an example of reacting on a `Show` event. This is done by implementing `onShow`. The code checks whether the *container event*[1] in scope is the action `PrintInvoice`. If so, the invoice document is grabbed and passed on to a method that is supposed to somehow modify the invoice doucument/replacing it. And the resulting document is being passed on instead of the original invoice document.

Listing 5.8: Reacting on Documents Being Shown.

```
2   @Override
3   public void onShow(final MiShow containerRunner,
4                      final McFileResource document) throws
                           Exception {
5     if (containerRunner.getEventInfo().getActionName().isLike("
          PrintInvoice")) {
6       containerRunner.next(modifyInvoice(document));
7     }
8   }
```

Listing 5.9 shows an example of reacting on `Load` events. We want to provide an import file programmatically rather than asking the user for it. So, in line 7 in the `onLoadPre` method we check if the action is the `Import` action. If so, `skip` is invoked in line 8. By `skip`ing we basically shortcut the load event, and the `onLoadPost` method will be called by the framework immediately after. Here, we again check whether the action in scope is `Import`, and if so, we generate an import file and return that file as the result. This happens in line 20.

Listing 5.9: Reacting on Documents Being Shown.

```
2   @Override
3   public MiOpt<MiFileSelector> onLoadPre(
4       final MiLoadPre containerRunner,
5       final MiFileSelector fileSelector) throws Exception {
6     final MiKey actionName = containerRunner.getEventInfo().
          getActionName();
7     if (actionName.isLike("Import")) {
8       containerRunner.skip();
9     }
10    return super.onLoadPre(containerRunner, fileSelector);
11  }
12
13  @Override
14  public MiList<McFileResource> onLoadPost(
15      final MiLoadPost containerRunner,
16      final MiList<McFileResource> fileResource) throws Exception
            {
```

[1] Not call-back!


```
17      final MiKey actionName = containerRunner.getEventInfo().
            getActionName();
18      if (actionName.isLike("Import")) {
19        final McFileResource importFile = generateImportFile();
20        return McTypeSafe.createArrayList(importFile);
21      } else {
22        return super.onLoadPost(containerRunner, fileResource);
23      }
24    };
```


Chapter 6

Programmatic Data Interaction

Until now, we have mainly been concerned with implementing container events, relying on the Extension Framework to make the necessary database updates and container invocations. While avoiding manually accessing the database and manually controlling container invocations, it is frequently necessary to do so. For example, it is often necessary to look up information of "secondary data" (i.e., data which is not directly related to an event-record) or to programmatically perform container operations (i.e., doing *other* container events than the event which is currently running.)

In this chapter we shall have a closer look at such programmatic data-interaction. More specifically, we shall learn how to read and manipulate data in the database, and how to programmatically interact with containers.

6.1 Accessing Containers

Sometimes, you may need to programmatically invoke operations on a container during an operation in another container.

For example, suppose that you need to introduce an action in the `maconomy:Employees` container: you wish to make an action that lets you replace the current employee with some other employee. The implication of executing the action would be that for all open jobs where the current employee is specified as project manager, that job will be assigned a new project manager (the replacement employee.) Since the project manager information is a part of the core Maconomy logic, it is not possible to manipulate the data directly using database operations. Even if it was, you shouldn't do that! Instead, you should leave updating logic to the contribution which owns this field.

In order to do so, we need to emulate that a user does this. This is what is mean by "programmatically accessing a container." The Extension Framework contains an API

for doing such things. Through this API you can do everything that can be done by an end-user.

6.1.1 Obtaining access to a container

The Extension Framework gives the possibility of interacting programmatically with containers. This is done by using an object called a *container executor*. A container executor represents a container. In addition, a container executor is tied to a specific pane of the container. Hence, any events (like `Update`) that is being invoked on a container executor will be performed on that particular pane.

In order to get access to a container executor, you must invoke the `executor` method. This method is available in two situations:

- For any `containerRunner`, i.e., from event methods. In this case, the `executor` method comes in two flavours: a zero-argument version, and a version that takes a container name as an argument. The zero-argument version indicates, that "this" container (i.e., the same container as the current event) should be accessed.
- From the data-model resources object obtained from the data model's `getResources` method. In this case, you *must* specify an explicit container name.

The `executor` doesn't give a container executor directly, instead, it provides the access point of doing so. It contains the following methods of interest:

Method	Remarks
`construct`	This method constructs a new container executor. The method comes in a number of flavours • A zero-argument version. This means that the container executor will be targeted at the *same pane* as the current event. Obviously, this variant is only available in contexts where this makes sense. • A one-argument version that takes the name of the pane (or the type of the pane) as an argument. Hence, in this way, you can specify whether to target the `filter`, the `card` or the `table` pane. When the container executor is accessed from a `containerRunner` of data-model event,

Method	Remarks
`initiate`	This container results in a *container-executor provider* which is an instance of the type: `MiContainerExecutor.MiProvider`. This class lets you `open` and `close` the container executor provider. `open`ing will return an object of type `MiContainerExecutor`. This corresponds to what you get from the `construct` method above. In this case you *must* remember to explicitly `close` the provider again! *Usually you should prefer using the* `construct` *method mentioned above instead if possible (see below.)*
`specify`	Returns the specification of the container referenced by the `executor` method.

Let us have a closer look at the `MiContainerExecutor.MiProvider`, i.e., the type that results from the `initiate` method. This type has the following methods of interest:

Method	Remarks
`specify`	Returns the specification of the container in question. Notice that since a container is being invoked, the result is taking *all contributions into account*. This method returns a `MiContainerSpec`. Using this, you can inspect the properties of the container by examining the `container` method. Please refer to Section 8.2 for more information.
`open`	This method "opens" a container, and returns a `MiContainerExecutor` which you really use to make operations on the container. By doing so, you must specify the name or the type of a pane in the container. The operations you are performing will be made on that pane! So, if you specify the type of a card pane (i.e., `MePaneType.CARD`), any operations made on that container executor will be made on the card. If you specify the table pane type instead (i.e., `MePaneType.TABLE`), the operation will be made on the table. You can specify any pane type or name known by the container in question. As an optional argument, you may specify whether `close` should automatically be invoked if an exception occurs during an operation. If you don't specify this argument, closing will automatically take place during exceptions. Thereby you don't have to wrap your code inside a `try-finally` structure[1].

[1]The automatic close-upon-exceptions are only done in operations on the contianer executor. So if your code does other things as well, you *must* wrap your code in a `try-finally` structure.

Method	Remarks
`close`	Closes the container. *It is important that you always close a container that has been opened!* A given container implementation may allocate scarce resources upon opening. If the container isn't closed, these resources may never be released!

Once you have `open`ed the container executor, you can start operating it. However, you may need to notify the framework about *which record instance* you are working with. That is: you need to specify which *container key* you relate to. A container key dictates which record instance is in scope. For example, just performing a `SubmitTimeSheet` makes little sense unless you first specify *which* time sheet is being submitted. On the other hand, if you want to initialize and create a new record in a card-pane, you don't need container key—none may exist.

A container executor can be used to invoke consecutive operations. Each operation will work on the result of the previous operation. So, for example, you can submit *and then* approve a time sheet using a container executor by:

1. Specifying the container key in scope (the key fields of the time sheet in question)
2. Submitting the time sheet. The framework will automatically do a read first, if this has not explicitly been done.
3. Approving the time sheet. The framework will automatically use the result of the previous operation (i.e., the result occurring from submitting the time sheet) as a foundation for the `ApproveAction` operation.

6.1.2 Obtaining Access to a Container with Automatic Management of Open/Close

The Extension Framework provides a way of managing the `open`ing and `close`ing of container-executor providers. This is possible for any container, and is especially easy for the current container. This is done by letting the framework automatically `open` a container-executor provider, thus directly returning a container-executor ready to be used. In addition, the framework will automatically `close` the underlying provider. Usually, you will let the framework mange this automatically. This behavior is available by invoking the method `construct` in one of its forms (instead of the `initiate` method.) The method is invoked on the result of the `executor` method. Most frequently, this is done by "chaining" the method calls like:

```
containerRunner.executor("TimeSheets").construct(MePaneType.CARD);
```

Often you wish to access whatever container is currently in scope for some container operation. For example, suppose you wish to implement an action that submits and

approves a job budget. You could do this by adding an action `SubmitAndApprove` to the `maconomy:JobBudgets` container. Whenever the `SubmitAndApprove` action is invoked, you need to do the following:

1. Create an instance of the `maconomy:JobBudgets` container (executor provider).
2. Open that instance, focusing on the card pane.
3. Set the container key to whatever key relates to the current `SumitAndApprove`-action.
4. Invoke the `SubmitJobBudget` action.
5. Invoke the `ApproveJobBudget` action.
6. Close the container (executor provider) instance.

By making use of the `construct` method, you can avoid most of these steps. Since you want access to the *current container* you don't specify the name of a container when invoking the `executor` method. From there you can invoke the `construct` method in a way that means "this pane" (i.e., the pane in which the current event occurs) or a specific pane. The result of doing that is, thus, an object of type `MiContainerExecutor`.

There are several ways of invoking `executor-construct`:

1. No arguments provided for the `executor` (meaning a container executor for *this* container) followed by the `construct` with no arguments (meaning the pane of the current event). Initially, such a container executor will be instantiated to work on the same key as the current event. This is the case when the `executor()` is invoked.
2. No arguments provided for the `executor` (meaning a container executor for *this* container) followed by a one-argument version of the `construct` method which takes either a pane name or a pane type as argument. This leads to a container executor which will work on the specified pane. For example, if the current event is related to the card pane, you can open an executor which is targeted on the table pane. Initially, such container executors will be instantiated to work on the same key as the current event (since the `executor` was invoked with no arguments.)
3. One argument provided for the `executor` method (either a `MiContainerName` or a `String`.) Following that, you can invoke the one-argument version of `construct`, specifying the pane of that container (either as a name or as a pane type.) Since this container-executor is related to another container, it will *not* initially have any key-restriction associated.

In the above example, a `SubmitAndApprove` action would be associated the card pane. Therefore `containerRunner.executor().construct()` would yield a container executor for the card pane in `maconomy:JobBudgets` container, and by default associated with the same key as the `SubmitAndApprove` event works on.

Another way of invoking the `construct` method is to pass the name of a pane, which will be the pane of choice rather than the "current pane." For example, if you want to introduce an action that automatically updates the prices of several records in a table pane, then that action would typically be associated with the card pane, but updating records in the table would happen using a container executor. In such a case, you can still benefit from the `construct` method, only you would explicitly obtain it with respect to the table rather than the card. Meaning that performed operations would be made on the table rather than the card. Hence `containerRunner.executor().construct(MePaneType.TABLE)`.

And finally, you could invoke the `executor` method so that the container executor returned by a subsequent `construct` refers to a pane in an entirely different container.

Apart from the fact that you don't have to `open`/`close` and that the container may be associated with a default container key[2], there is no difference between working with container executors instantiated using

```
executor(...).construct(...)
```

and

```
executor(...).initiate()
```

6.1.3 Controlling the Scope of Container Operations

As mentioned above, you may need to specify the container key, i.e., determining the scope of an operation. This corresponds to determining what data is being read by the container executor when a read is invoked. And similarly, what the scope of some other operation is.

You may change the key at any time. Doing so will automatically make the container executor forget about the data it may currently have. The next operations will concern the key just specified.

In order to change the key, or the *restriction* of the container executor, you must invoke the `control` method. The control method contains a number of methods that can be used to set the restriction. Notice that a restriction used for *filter* panes are somewhat different compared to card and table panes. Filters are less rigorous in that the exact columns being read is not necessarily all possible columns. And the rows read are not necessarily all rows: it may be based on some expression-based query, and it may read only a sub-set of all potentially available rows (e.g., a "page" such as records 51–100.) Finally, filters may be subject to a specified ordering. This is not the case for card and table panes: here a specific key value (a value for each key field of the container) must be provided. The object returned by the `control` is of type `MiContainerExecutor.MiControl` and lets you change all of these aspects. Only some aspects relate to filter panes only,

[2]When the opened container is implicitly the same as the current container

other aspects relate to non-filter panes. The list of methods offered by the control object is:

Method	Remarks
`restrictBySome`	This method applies a restriction corresponding to a top-level pane in a workspace, or a pane that is bound using a `<Mount>`-binding. Usually, there is no guarantee what this exactly means. For some containers, however, only one key value exists. Some containers, e.g., `maconomy:BatchJobReallocation` and `maconomy:CalculateWorkHours`, don't even *allow* a specific key. For such containers, you should use this method to set the restriction. This method applies to card and table panes only.
`restrictBy`	This method is used to restrict a container key by a specific set of values (for card or tables) or by a specific filter-expression (for filters.) For non-filter panes, you can restrict by using a `MiKeyValues` argument which is basically a specific typed flavour of `MiDataValues`. Alternatively, you may in-line the field names an corresponding values in the method call. For filters, you may specify the restriction as an `MiExpression` or a `MiQuery` which—in addition to an expression—may contain information about sort-order and paging. Alternatively, you may specify such properties separately, see below.
`select`	This method applies to filter panes only. It is used to specify the columns to be read when a read is next time invoked on the container executor. You may either provide an `Iterable` of names (e.g., a list or a set) or you may provided a comma-separated list of field names. When this method is invoked, the restriction, paging and ordering information is cleared!
`allRows`	This method applies to filter panes only. It resets any paging information previously given. Any filtering-expressions or sort-order already given will be maintained.
`rowRange`	This method applies to filter panes only. It can be used to specify the range of rows (i.e., the "page") to be read when a read is next time invoked on the container executor.
`orderBy`	This method applies to filter panes only. It can be used to specify a specific ordering of one or more fields. This ordering will be applied when a read is next performed by the container executor.

All the methods of the control object returns the control object itself, allowing you to chain the methods.

6.1.4 Invoking Operations using the Container Executor

So, once the container executor has been obtained and once you have specified its scope, you can start invoking operations on the container executor. You may change the scope via the control object at any time. When you do so, the internal data of the container runner is cleared, and a re-read will automatically be performed the next time you invoke an action that requires present data.

The operations that you perform *do not return any data directly*. If you need the resulting data, you can query the container executor for that, see below. Instead the container-executor operations returns the container executor itself, thereby allowing you to chain multiple operations, for example:

```
MiContainerExecutor timeSheets = ...;
timeSheets.control().restrictBy(...);
timeSheets.action("SubmitTimeSheet").action("ApproveTimeSheet");
```

The list of operations on a `MiContainerExecutor` is:

Method	Remarks
`add`	This method invokes a `Initialize` event on the container. For table-panes, this event will be instantiated such that it corresponds to appending the line to the end of the table.
`insert`	This method invokes a `Initialize` event on the container. For table-panes, this event will be instantiated such that it corresponds to inserting the line above whatever line is considered the "current line." See below how to determine and control the current line.
`create`	This method invokes a `Create` event on the container. As an argument you must provide a `MiDataValues` object which corresponds to the fields that might have been changed by a user compared to the template record returned from the `add` or `insert` methods. Since you may chain the operation-methods, you can easily do a full creation process by `myContainer.insert().create(...);`

Method	Remarks
`read`	This method invokes a `Read` event on the container. Since read-events are generally automatically performed when needed, you may occasionally want to explicitly perform a read. This may happen if you perform operations (possibly in other containers) and you know a refresh is needed.
`update`	This method invokes a `Update` event on the container. If the current pane is a table pane, the update will happen of whatever record is the "current line." See below how to determine and control the current line. As an argument, you must provide a `MiDataValues` object corresponding to values that are being changed by a user.
`delete`	This method invokes a `Delete` event on the container. If the current pane is a table pane, the deleted row will be whatever record is the "current line." See below how to determine and control the current line.
`action`	This method invokes a `Action` event for an action with a specific name on the container. If the current pane is a table pane, the action event will relate the whatever record is the "current line." See below how to determine and control the current line. You must provide the name of the action to execute.
`print`	This method invokes a `Print` event on the container. If the current pane is a table pane, the print event will relate the whatever record is the "current line." See below how to determine and control the current line. You must provide the name of the action to execute.
`move`	This method invokes a `Move` event on the container. The move event is relevant for table panes only. The moved row will be whatever record is the "current line." See below how to determine and control the current line. You must provide the name of the action to execute. You must provide a move operation as well as the row relative to which the move operation must be interpreted. See Section 4.10 for more information on move operations.
`lock`	This method invokes a `Lock` event on the container. If the current pane is a table pane, the lock event will relate the whatever record is the "current line." See below how to determine and control the current line. You must provide the name of the action to execute. Most often, the locking will work relative to an entire container. This is up to the concrete implementation.

Method	Remarks
`unlock`	This method invokes a `Unlock` event on the container. If the current pane is a table pane, the unlock event will relate the whatever record is the "current line." See below how to determine and control the current line. You must provide the name of the action to execute. Most often, the locking/unlocking will work relative to an entire container. This is up to the concrete implementation.
`inspect`	This method returns a `MiPaneSpecInspector` representing the specification of the pane of this container executor. Using this, you can inspect the properties of the container by examining the `container` method. Please refer to Section 8.2 for more information.

All the above methods are found in two variants: either with or without parameters. If no parameter-object is specified, the parameter-object associated with the event in question will be the empty parameter set. In this way, you can programmatically associate parameters with any container event.

Listing 6.1 shows an example of an action implemented in some container containing the fields `ID` and `NAME`. The action creates an employee with `EmployeeNumber` = `ID` and `Name1` = `NAME`. This is done by creating a provider of the `Employees` container. This happens in line 11. In line 13 the container is then opened, focusing on the card pane. In order to create a new employee, it is necessary to obtain a template record by invoking an `Initialize` event, followed by a `Create` event specifying changes compared to the template record. This all happens in line 23. Finally, the container is closed in line 24.

Listing 6.1: Operating on a New Container.

```
2   @Action("CreateAsEmployee")
3   private static final class CreateAsEmployeeHandler extends
        McAbstractDataModelRootAction {
4     private static final McPopupDataValue US = McPopup.val("
          CountryType", "United_States", 0, "United States");
5     @Override
6     public void onAction(final MiActionPost containerRunner,
7                          final MiAction eventData) throws
                               Exception {
8       final MiValueInspector origData = eventData.getOriginalData
            ();
9       final MiContainerName employeeContainerName =
          McContainerName.create("Employees");
10      final MiProvider employeesContainer =
11        containerRunner.executor(employeeContainerName).initiate()
              ;
```



```
12        final MiContainerExecutor employeeCard =
13          employeesContainer.open(MePaneType.CARD);
14
15        // create a new employee by using
16        // the ID field as employee number
17        // the NAME field as Name1 and always use
18        // United States (defined in constant US) as Country.
19        final MiDataValues creationValues =
20          dataValues().setVal("EmployeeNumber", origData.getVal("ID"
              ))
21                      .setVal("Name1", origData.getVal("NAME"))
22                      .setPopup("Country", US);
23        employeeCard.insert().create(creationValues);
24        employeesContainer.close();
25        containerRunner.call().notification("Employee has been
              created");
26    }
```

Using the `executor-construct` method instead of the `executor-initiate` implies that opening and closing is managed by the framework. In this case, the example from Listing 6.1 can be expressed slightly easier, as shown in Listing 6.2. The main difference is found in line 4 where a container executor is directly obtained from a named container and a pane on that container.

Listing 6.2: Operating on a New Container Without Explicit Open/Close.

```
2         final MiValueInspector origData = eventData.getOriginalData
              ();
3         final MiContainerExecutor employeeCard =
4           containerRunner.executor("Employees").construct(MePaneType
              .CARD);
5
6         // create a new employee by using
7         // the ID field as employee number
8         // the NAME field as Name1 and always use
9         // United States (defined in constant US) as Country.
10        final MiDataValues creationValues =
11          dataValues().setVal("EmployeeNumber", origData.getVal("ID"
              ))
12                      .setVal("Name1", origData.getVal("NAME"))
13                      .setPopup("Country", US);
14        employeeCard.insert().create(creationValues);
15        containerRunner.call().notification("Employee has been
              created");
```

Listing 6.3 shows an example of an action implemented as an extension to the `Jobs` container. The action is supposed to mark all "blocked" fields for the current job. The implementation creates a container executor for the `Jobs` container by invoking the `executor-construct` method. This automatically focuses on the same pane which is by

default targeted for the same job as the action event—which is exactly what we want. In lines 9–13 the four "blocked" fields are updated by given an data-values structure associating all these fields with the value `true` to the `update` method.

Listing 6.3: Operating on the Current Container.

```
2   @Action("BlockAll")
3   private static final class BlockAll extends
        McAbstractDataModelRootAction {
4     @Override
5     public void onAction(final MiActionPost containerRunner,
6                          final MiAction eventData) throws
                               Exception {
7       // Set all "block"-like fields on Job to true
8       final MiContainerExecutor job = containerRunner.executor().
            construct();
9       job.update(dataValues()
10                    .setBool("BlockedForTimeRegistrations", true)
11                    .setBool("BlockedForInvoicing", true)
12                    .setBool("BlockedForBudgeting", true)
13                    .setBool("BlockedForAmountRegistrations", true));
14    }
```

6.1.5 Inspecting Data of a Container Executor

As mentioned in Section 6.1.4, the event-invocation methods of the container executor don't return the data resulting from the given event. Instead, the container executor itself is returned. This allows the programmer to conveniently invoke a series of events against a container in a "one-liner."

Of course, it must be possible to inspect the data resulting from events. This is possible because the container executor maintains a relevant set of values of a container. For example, after performing a `Read` on a pane, you can retrieve the resulting data. Then if you run an `Action`, an `Update` or both, you can again inspect the resulting data. You can even inspect data in other panes that happen to get data for specific events. For example, reading the card pane in the `maconomy:TimeSheets` container will also provide a result for the corresponding table pane and vice versa.

The pane which is in scope (the pane used when constructing the container executor) has a "current row pointer" which indicates which of the result rows is considered the current. Any operation will be made with respect to that current row. For cards, there is generally one row (or zero rows) and the "current row pointer" is therefor not of much interest in this case. For table panes, however, the current row is highly relevant.

In general, you can obtain the current row index, the current row, the current pane value and the current container value (i.e., a value comprising potentially several panes, e.g.,

a card and a table pane.) The methods used to obtain information about the current values of a container executor are:

Method	Remarks
`getRowCount`	Returns the number of rows contained in the current pane.
`getRecord`	Returns a `MiValueInspector` representing the current record value in the current pane. NB! If there are no records in this pane, this method will throw an exception. See `getRecordOpt` below.
`getRecordOpt`	This method corresponds to `getRecord` (see above) except that the result is wrapped in a `MiOpt` value. If there is no current record, a `McOpt.none` object will be returned. If there is a possibility that there is no current record, you should use this method and check whether the result `isDefined()` or `isNone()`!
`getPaneValue`	Returns a `MiPaneInspector` which gives read-only access to various pane-value properties. Apart from allowing access to any record value in the pane, it also gives paging information, action-enabledness state etc. NB! If no pane value has been obtained for the current pane, this method will throw an exception. See `getPaneValueOpt` below.
`getPaneValueOpt`	This method corresponds to `getPaneValue` (see above) except that the result is wrapped in a `MiOpt` value. If the current pane value is undefined, this method will return a `McOpt.none` object. If there is a possibility that the pane value is undefined, you should use this method and check whether the result `isDefined()` or `isNone()`.
`getContainerValue`	This method returns a container value object for the current container value, i.e., a structure giving information about all pane values that have received data. This method always yields a well-defined result, since a container value may be empty.
`getRowIndex`	This method returns the row index of the current row. Notice that indexes are 0-based. NB! If there is not a well-defined current row (e.g., in case there is no data), this method will throw an exception! See `getRowIndexOpt` below.
`getRowIndexOpt`	This method corresponds to `getRowIndex` (see above) except that the result is wrapped in a `MiOpt` value. If there is no current row, a `McOpt.none` object will be returned. If there is a possibility that there is no current row, you should use this method and check whether the result `isDefined()` or `isNone()`!

Method	Remarks
`getKeyValues`	This method returns the formal key values of the current record. Hence, it is not the *container key* that is returned, but the key values used to identify the record having focus.

Listing 6.4 shows how to look up data in the container executor. The code snippet illustrates a case where a job is created (in the example for a hard-coded customer.) The creation takes pace in line 6. In line 14 we check the value of the `Status` field of the current record (which is the record resulting from creating the job.) If the status is *not* `Quote`, we execute the action `ConvertToQuote` (line 15.) Finally, in line 19 we generate a message to the end-user displaying the job number that was created, this time by looking up the value of the field `JobNumber` in the current record.

Listing 6.4: Inspect Data of a Container Executor.

```
2   final MiContainerExecutor job = containerRunner.executor("Jobs
        ").construct(MePaneType.CARD);
3
4   // Create a job for some (hard-coded)
5   // customer number
6   job.insert().create(dataValues().setStr("CustomerNumber", "
        2888-3883"));
7   // set the key to the newly created job number
8   final MiKeyValues newJob = job.getRecord().asKeyValuesCopy("
        JobNumber");
9   job.control().restrictBy(newJob);
10
11
12  // check if the job status is Quote. If not
13  // convert it into Quote status.
14  if (!job.getRecord().getPopup("Status").isLike("Quote")) {
15    job.action("ConvertToQuote");
16  }
17  final String message =
18    String.format("Job No. %s has been created",
19                  job.getRecord().getStr("JobNumber"));
20  containerRunner.call().notification(message);
```

6.1.6 Navigating and Iterating through Records

As pointed out in Section 6.1.5, a container executor has a concept of a "current record." When you invoke some operation, it is related to the current record.

Just as you can ask what the current row index is, you can change the current row index.

Doing so corresponds to a user bringing focus to a particular line: in itself it has no effect on the container data, but the next event will be relative to that row.

For this reason it is possible to move the value of the current row using the following methods:

Method	Remarks
`setRowIndex`	This method is used to set the row index to some arbitrary 0-indexed value. If this row index does not exist, `false` will be returned. Otherwise `true` is returned. If `false` is returned, the current row will not be affected.
`incRowIndex`	This method increases the row index by one. If it is not possible to increase the current row index, `false` is returned. Otherwise `true` is returned. If `false` is returned, the current row index will not be affected.
`decRowIndex`	This method decreases the row index by one. If it is not possible to decrease the current row index, `false` is returned. Otherwise `true` is returned. If `false` is returned, the current row index will not be affected.

Using the above methods it is possible to iterate over the rows or to put focus into a specific row.

Hence, by using these methods, you can navigate to specific rows of a container executor. Listing 6.5 shows an example where a given expense sheet is opened. If it contains 2 lines or more, the last line is moved to the top. This example exemplifies how to put focus on a specific row in a container executor: in line 5 we read a specific expense sheet table. If there are more than 1 records in the resulting table, the focus is put onto the last row. This is done in line 8 by using the `setRowIndex` method. This implies that further actions will relate to that record. In line 10 a `Move` operation is made, moving the current row before row index 0 (i.e., to the top.)

Listing 6.5: Navigating to a Specific Record.

```
2     final MiContainerExecutor expenseSheetsTable = containerRunner
          .executor("ExpenseSheets").construct(MePaneType.TABLE);
3     expenseSheetsTable.control().restrictBy(key("
          ExpenseSheetNumber"), McInt.val(expSheetNumber));
4
5     expenseSheetsTable.read();
6     if (expenseSheetsTable.getRowCount() > 1) {
7       // navigate to last line
8       expenseSheetsTable.setRowIndex(expenseSheetsTable.
            getRowCount() - 1);
9       // move that line above the first line (line 0)
```

```
10        expenseSheetsTable.move(MeMoveOperation.MOVE_BEFORE, 0);
11      }
```

In addition, you can alter the focus row to the next (`incRowIndex`) or the previous (`decRowIndex`). All of the focus-positioning methods return a boolean indicating if the row index is out of bounds (`false`) or not. Using these methods and the return value, you can iterate over the records in a pane. Listing 6.6 shows an example where we iterate through all records in the table of the `maconomy:TimeSheets` container. Initially we just read some hard-coded time sheet, just as an example. Then in line 18 we put focus on the first row. If there are no rows, this method will return `false`. The result is stored in the local variable `hasMoreRows`. In lines 19–22 we iterate over the all the table rows. The point is that we in line 21 increase the row number and update the variable `hasMoreRows`: once we reach the last line, the `while`-loop will terminate.

Listing 6.6: Iterating Over All Rows in a Table.

```
2       final MiContainerExecutor timeSheetTable = containerRunner.
           executor("TimeSheets").construct(MePaneType.TABLE);
3       final MiKeyValues tsKey = McKeyValues.create()
4         .setStr("EmployeeNumber", "Emp001")
5         .setDate("PeriodStart", 2013, 7, 22);
6       timeSheetTable.control().restrictBy(tsKey);
7
8       final MiRecordValue zeroAllDays = dataValues()
9         .setReal("NumberOfDay1", BigDecimal.ZERO)
10        .setReal("NumberOfDay2", BigDecimal.ZERO)
11        .setReal("NumberOfDay3", BigDecimal.ZERO)
12        .setReal("NumberOfDay4", BigDecimal.ZERO)
13        .setReal("NumberOfDay5", BigDecimal.ZERO)
14        .setReal("NumberOfDay6", BigDecimal.ZERO)
15        .setReal("NumberOfDay7", BigDecimal.ZERO);
16
17      timeSheetTable.read();
18      boolean hasMoreRows = timeSheetTable.setRowIndex(0);
19      while(hasMoreRows) {
20        timeSheetTable.update(zeroAllDays);
21        hasMoreRows = timeSheetTable.incRowIndex();
22      }
```

Depending on your needs, iterating over a number of records can typically be done more easily and elegantly by using the concept of *record executors*, as explained in Section 6.1.7.

6.1.7 Record Executors

A record executor is obtained from a container executor. It offers the possibility to execute events, although the events are locked to a specific record. From a container

executor, you can obtain a record executor for the current record (i.e., the record of the focus row specified by `getRowIndex()` of the container executor). As long as that record exists in the container executor, you can do operations on that record, regardless of whether the focus row index of the container executor is changed and regardless of whether the record has moved to a new index position (i.e., as a side-effect of doing operations on the container executor.) A container executor and any record executor obtained from it, will remain in sync. Hence, updating data in one can immediately be seen in the other.

The `MiContainerExecutor` interface gives access to a couple of methods that gives you access to a record executor:

Method	Remarks
`getRecordExecutor`	This method returns a record executor which is locked to the row currently having focus in the container executor, i.e., the row having the index returned by `getRowIndex()`. This record executor will forever be locked to that particular *semantic* row. Essentially the row having that particular unique key. So, even if the row is moved to some other row position in the underlying container executor, the record executor will *still* be locked to the same record. If there's no current row (i.e., if the pane is empty) an error is issued.
`getRecordExecutorOpt`	This method is similar to `getRecordExecutor` except that the result is returned as an `MiOpt`. Hence, if there is no current record, a `none` value is returned.

Method	Remarks
`matchBy`	This method is used to iterate over a number of rows in the container executor: it returns an `Iterable` of `MiRecordExectuor`s that are tied to the records that fulfill the condition specified as an argument to the `matchBy` method. The iteration order will go from lower indexes towards higher indexes. This can, however, be changed by invoking the method `reverse()` on the iterable: this will return an iterable with the same elements, only the iteration order is reversed compared to the original. This method is found in a number of flavours: • A variant that takes an expression as input argument. Only records for which the expression evaluates to true will be part of the iterable. • A variant that takes a value inspector. Only records for which the specified fields have the corresponding values will be part of the iterable. This corresponds to transforming the value inspector into an expression by use of the `asExpression()` method. • A variant that takes a `Predicate` as argument. This variant lets you make use of Java 8's lambda expression notation. In short, you provide a function that—given a record—decides whether or not to include the record in the returned `Iterable`.
`matchAll`	This method is similar to `matchBy` above, except that it matches all records. This is useful for iterating over all records in a table.

Listing 6.7 shows how to iterate over all time sheet lines in a time sheet, updating each of them. The program does the same thing as the one shown in Listing 6.6. The difference is how to loop over the various lines. In this case, we make use of the `matchAll` method. This happens in line 17. The result of that method is an `Iterable` of all rows, each represented as a record executor. Since each record executor is locked to a specific record, we don't need to explicitly change the focus row of the container executor using `setRowIndex`. Instead, we directly invoke the `update` method on the record executor. After the loop is done, the current row index of the container executor is left unchanged.

Listing 6.7: Iterating Over All Rows in a Table Using Record Executors.

```
1  final MiContainerExecutor timeSheetTable = containerRunner.
       executor("TimeSheets").construct(MePaneType.TABLE);
2  final MiKeyValues tsKey = McKeyValues.create()
```



```
3           .setStr("EmployeeNumber", "Emp001")
4           .setDate("PeriodStart", 2013, 7, 22);
5       timeSheetTable.control().restrictBy(tsKey);
6
7       final MiRecordValue zeroAllDays = dataValues()
8           .setReal("NumberOfDay1", BigDecimal.ZERO)
9           .setReal("NumberOfDay2", BigDecimal.ZERO)
10          .setReal("NumberOfDay3", BigDecimal.ZERO)
11          .setReal("NumberOfDay4", BigDecimal.ZERO)
12          .setReal("NumberOfDay5", BigDecimal.ZERO)
13          .setReal("NumberOfDay6", BigDecimal.ZERO)
14          .setReal("NumberOfDay7", BigDecimal.ZERO);
15
16      timeSheetTable.read();
17      for (final MiRecordExecutor timeSheetLine : timeSheetTable.
            matchAll()) {
18        timeSheetLine.update(zeroAllDays);
19      }
```

Suppose that we want to apply a small optimization: instead of updating *all* rows with zero values, we want to ignore rows that already have zero values. This can be easily achieved by making use of the `matchBy` method, as shown in Listing 6.8. In line 18, we simply state that we only want to iterate over rows where the rows do *not* contain all zero values.

Listing 6.8: Iterating Over All Rows in a Table Using Record Executors.

```
17      // Only update lines that are not already zero-lines
18      for (final MiRecordExecutor timeSheetLine : timeSheetTable.
            matchBy(not(zeroAllDays.asExpression()))) {
19        timeSheetLine.update(zeroAllDays);
20      }
```

A record executor pretty much corresponds to a container executor, but it's locked to a particular record. The following methods are available on a record executor:

Method	Remarks
`add`	This method executes an `Initialize` event, adding (appending) a new row at the end of a table pane. This is step one of the two step creation process. The result is *a different* record executor which is now tied to the template record obtained as an effect of the `Initialize` event. Thus chaining the create method will in fact invoke creation of that template record.

Method	Remarks
`insert`	Like the `add` method, this method executes an `Initialize` event, but *inserts* the template record above the record represented by the current record executor. The result is a *a different* record executor which is now tied to the template record obtained as an effect of the `Initialize` event. Thus chaining the create method will in fact invoke creation of that template record.
`create`	This method executes a `Create` event on the current record executor. The change values that should be applied are passed as an argument. The result is the record executor corresponding to the created record.
`update`	This method executes an `Update` event on the current record executor. The change values that should be applied are passed as an argument. The result is this record executor, allowing for method chaining, e.g., first update, then run an action.
`delete`	This method executes a `Delete` event on the current record executor. Since this leads to that the corresponding record is *deleted*, you can no longer work with this record executor. Any attempt to do so, will result in a run-time error. For this reason, the return value of this method is a value inspector reflecting the value of the record that was deleted, not a record executor.
`print`	This method executes a `Print` event on the current record executor. The result is this record executor, allowing for method chaining, e.g., first print, then update.
`action`	This method executes an `Action` event of some named action on the current record executor. The result is this record executor, allowing for method chaining, e.g., first run one action, then run another.
`move`	This method executes a `Move` event on the current record executor. The result is this record executor, allowing for method chaining, e.g., first move the record, then update it.
`lock`	This method executes a `Lock` event on the current record executor. The result is this record executor, allowing for method chaining.
`unlock`	This method executes an `Unlock` event on the current record executor. The result is this record executor, allowing for method chaining.
`getContainerName`	This method returns the name of the container to which this record executor is tied.

Method	Remarks
`getPaneName`	This method returns the pane name of the pane to which this record executor is tied.
`getPaneState`	This method returns the current pane state of the pane to which this record executor is tied.
`getRowIndex`	This method returns the row index to which this record excutor is tied. Note that this might differ over time. For instance if you have a record executor linked with record index 1 in a table pane and the record with index 0 is deleted, then *this* record executor becomes linked to the record with index 0, since it is tied to the same semantic record over time.
`getRowIndexOpt`	This method is similar to `getRowIndex` execpt that the result is wrapped in a `MiOpt`.
`inspect`	This method returns a spec inspector of the corresponding pane in the corresponding container, allowing programmatic access to things like field declarations, foreign key declarations, available actions etc.
`isActionEnabled`	This method can be used to find out whether a given action is enabled for this record executor.
Value Inspector methods	A record executor can be treated simply as a record containing the values of the corresponding record. For this reason, all methods available on the interface `MiValueInspector` are available on a record executor. For example, `getBool`, `getStr`, `getVal`, `fieldCount`, `containsField`, `copyValues` etc. Please refer to Section 4.3.1 for more on this.

In Listing 6.7 and Listing 6.8 we saw an example of invoking the `update` method on a record executor. Since a record executor is locked to a *semantic record* (not its row index), we can iterate over collections of record executors while performing operations that make changes to the row index. Deleting records is an obvious example: suppose we want to delete all records in a table. This can be done by iterating over all of the lines, deleting each of them. Listing 6.9 shows an example of this: here we merely iterate over all lines, and delete each of them in line 10. *A warning should be issued here.* If you do a forward iteration over the lines, as in Listing 6.9, all the remaining lines will be re-numbered (depending on the underlying application logic) every time you delete. Although the functionality will work, this may lead to poor performance for no reason at all.

Listing 6.9: Deleting Lines Using Record Executors.

```
1  final MiContainerExecutor timeSheetTable = containerRunner.
       executor("TimeSheets").construct(MePaneType.TABLE);
2  final MiKeyValues tsKey = McKeyValues.create()
3    .setStr("EmployeeNumber", "Emp001")
4    .setDate("PeriodStart", 2013, 7, 22);
5  timeSheetTable.control().restrictBy(tsKey).read();
6
7  //! Warning: forward iteration may lead to poor performance
8  //! since lines need to be renumbered for every deletion.
9  for (final MiRecordExecutor timeSheetLine : timeSheetTable.
       matchAll()) {
10   timeSheetLine.delete();
11 }
```

An alternative to doing a forward iteration, is a *reverse iteration*. Such a reversed iteration order can be obtained by invoking the `reverse` on the record executor iterable returned by `matchAll` and `matchBy` methods. An example can be seen in Listing 6.10. Here the iteration order is reversed in line 9. Thereby avoiding the need for re-numbering of any other time sheet line.

Listing 6.10: Deleting Lines Using Record Executors and Reverse Iteration.

```
7  // By reversing the iteration order, performance is
8  // improved since re-numbering is not needed.
9  for (final MiRecordExecutor timeSheetLine : timeSheetTable.
       matchAll().reverse()) {
10   timeSheetLine.delete();
11 }
```

The fact that record executors are locked to a semantic record (not its index) means that container executors and record executors can be used jointly: performing an operation on a record executor will (if possible) preserve the focus row index of the container executor. To illustrate this, suppose we have some time sheet containing the following three lines:

Row index	Remark	NumberDay1
0	On-sight Assessment	4.5
1	Final Report	3.0
2	Pre-analysis	1.5

Consider Listing 6.11 which shows how container executors and record executors can be operated one after the other. First the row focus of the container executor is set to 1 (i.e, the 2nd line). This happens in line 3. This information, and the Remark field of the related line is printed to the console. Next, in line 12, we iterate over all the lines in the table using record executors. For each iteration, we print the focus index of the *container*

executor to the console (along with the currently associated Remark value of that line). The Remark of that line is updated appending a 'C' at the end. This happens in lines 13–21. Next, we print the row index currently associated with the record executor in scope, as well as its Remark value. And the Remark of that record is updated, appending an 'R' at the end. This takes place in lines 23–31. After the loop we again investigate the row focus index of the container executor and the corresponding value of the Remark field (lines 36–39.) And we loop over all of the lines, printing the row index and the Remark field value for all the lines (lines 42–45.) The resulting output is:

```
BEFORE LOOP
Container executor focus index:1
Remark of focus line          :Final Report
------------------------------
LOOP WITH UPDATES
Container executor focus index:1
Remark of focus line          :Final Report
Record executor focus index   :0
Remark of record executor     :On-sight Assessment

Container executor focus index:1
Remark of focus line          :Final ReportC
Record executor focus index   :1
Remark of record executor     :Final ReportCC

Container executor focus index:1
Remark of focus line          :Final ReportCCR
Record executor focus index   :2
Remark of record executor     :Pre-analysis

------------------------------
AFTER LOOP
Container executor focus index:1
Remark of focus line          :Final ReportCCRC
Remarks of all lines:
Row index                     :0
Remark has value              :On-sight AssessmentR
Row index                     :1
Remark has value              :Final ReportCCRC
Row index                     :2
Remark has value              :Pre-analysisR
```

Notice that the row focus index for the container executor remains unchanged throughout the process, also while the record executors of various row indexes are being operated on.

Also notice that the effect of doing an update in a container executor is immediately seen in the record executor and vice versa. Specifically, we notice that the row with index 1 (the row that the container executor has its focus on) is updated four times: first by the container executor (when the record executor is tied to index 0). Then again by the container executor (when the record executor is tied to index 1), and immediately after by the record executor. And finally, it is once again updated by the container executor while the record executor is tied to index 2. Hence, yielding a Remark value of `ReportCCRC`.

Listing 6.11: Mixing Container Executors and Record Executors.

```
1  final MiContainerExecutor tsTable = containerRunner.executor
       ("TimeSheets").construct(MePaneType.TABLE);
2  tsTable.control().restrictBy(timeSheetKey);
3  tsTable.setRowIndex(1);
4  print("BEFORE LOOP");
5  print("Container executor focus index",
6        tsTable.getRowIndex());
7  print("Remark of focus line",
8        tsTable.getRecord().getStr(REMARK));
9  print("--------------------------------");
10 print("LOOP WITH UPDATES");
11
12 for(final MiRecordExecutor tsLine : tsTable.matchAll()) {
13   print("Container executor focus index",
14         tsTable.getRowIndex());
15   final String remarkContainerFocus = tsTable.getRecord().
       getStr(REMARK);
16   print("Remark of focus line",
17         remarkContainerFocus);
18   // Update the container focus line appending a 'C' at the
       end of the remark
19   tsTable.update(dataValues()
20                  .setStr(REMARK,
21                          remarkContainerFocus + "C"));
22
23   print("Record executor focus index",
24         tsLine.getRowIndex());
25   final String remarkRecordFocus = tsLine.getStr(REMARK);
26   print("Remark of record executor",
27         remarkRecordFocus);
28   // Update the container focus line appending a 'R' at the
       end of the remark
29   tsLine.update(dataValues()
30                 .setStr(REMARK,
31                         remarkRecordFocus + "R"));
32   print("");
33 }
34 print("--------------------------------");
```



```
35        print("AFTER LOOP");
36        print("Container executor focus index",
37              tsTable.getRowIndex());
38        print("Remark of focus line",
39              tsTable.getRecord().getStr(REMARK));
40        print("Remarks of all lines:");
41        for (final MiRecordExecutor tsLine : tsTable.matchAll()) {
42          print("Row index",
43                tsLine.getRowIndex());
44          print("Remark has value",
45                tsLine.getStr(REMARK));
46        }
```

6.1.8 Working with Multiple Container Panes

Sometimes, you may wish to access information and/or operate several panes of some container programmatically. For example, assume that you wish to programmatically create a time sheet *and* create a number of time sheet lines. Doing so would require you to work with the card pane (in order to create the time sheet) and work with the table pane (in order to create time sheet lines.) However, a container executor is always locked to a specific pane in order to avoid confusion.

Obviously, you could create *two* different container executors (one focusing on the card, and one focusing on the table.) But this is overkill and gives unnecessary overhead, when they are meant to relate to the same time sheet.

The Extension Framework has a solution to this: from a container executor, you can *derive* other container executors for other panes of the same container. The original container executor is the *master*: this instance controls how the container is restricted (e.g., what key is being referred to.) The derived container executors are *slaves*: they automatically follow the key defined by the master, and it is not possible to derive further container executors from these. They do, however, allow access to other panes in the container.

Method	Remarks
`derive`	This method derives a container executor with focus on some other pane, which is provided as an argument. The derived container executor lets you invoke operations on records in that other pane. Its content will always be derived from whatever key/restriction is specified for the master container executor. The data in the derived executor and the one from where it's derived will be synchronized. For example, imagine that you have a container executor for the `maconomy:TimeSheets` card pane and from there derive an executor for the table pane. Then changing data in the table pane which affects the card pane (or vice versa) will be immediately visible in related executor.

Listing 6.12 shows an example using this mechanism. In the example, the card pane of the `maconomy:TimeSheets` is used as the master. The table pane is derived from it in line 4. In line 6 the card pane is used as usually to create a new time sheet. Then a time sheet line is created using the table-part container executor in line 18. Since the table pane is derived from the card, the created lines will automatically be made in the time sheet just created using the card pane. Notice the use of the `add` method which is used to ensure that the new line is appended to the end, no matter the number of lines that may have been created as a side-effect of creating the time sheet. The creation of the time sheet line specifies a number of hours for Monday. This will be reflected in the card pane. We can directly see this effect by looking up data using the card pane. This is so because the table was derived from the card—in this way the two panes are linked together.

Listing 6.12: Working With Multiple Panes.

```
2     final MiContainerExecutor timeSheetCard = containerRunner.
          executor("TimeSheets").construct(MePaneType.CARD);
3     final MiContainerExecutor.MiDerived timeSheetTable =
4         timeSheetCard.derive(MePaneType.TABLE);
5
6     timeSheetCard.insert().create(
7       dataValues().setStr("EmployeeNumber", "Emp001")
8                   .setInt("WeekNumber", 30));
9
10    // set the key to the time sheet just created
11    final MiKeyValues currentTS = timeSheetCard.getRecord().
          asKeyValuesCopy("EmployeeNumber", "PeriodStart");
12    timeSheetCard.control().restrictBy(currentTS);
13
14    // now the time sheet has been created.
```



```
15      // Add a line at the end of the table.
16      // Hence if lines already have been created,
17      // it's added at the bottom.
18      timeSheetTable.add().create(dataValues().setReal("NumberDay1",
            8));
19
20      // after adding the line, the content of the card is
21      // automatically updated since the two container
22      // executors are linked since one is derived from
23      // the other!
24      final BigDecimal totalDay1 = timeSheetCard.getRecord().getReal
            ("TotalNumberDay1Var");
25      containerRunner.call().notification("Number of hours on Monday
            : " + totalDay1);
```

6.2 Accessing System and Database Information

Although the Extension Framework in many cases takes care of fetching and persisting custom data, it is still frequently needed to look up stuff in the Maconomy database. And sometimes, you need access to other user or server related parts of the Maconomy system. The Extension Framework has an API for that.

In general, access to the database is obtained from a `resources` object that is given as an argument to the factory/constructor methods of data-models and containers. The resources object should not be used directly from the constructor. Instead, the abstract data-model implementation provides a method for obtaining access to the resources: `getResources` This method should only be used outside of the constructor!

This `getResources` returns an object of type `MiDataModel`.`MiResources`. This type offers the following interesting methods:

Method	Remarks
`executor`	This provides the ability to create a container executor as explained above. Like above, you invoke this method using a container name. By subsequently invoking `initiate`, you get a container-executor provider which you can then `open`. This may be used, for example, to implement a `MiPersistenceStrategy` which persists using a specific container rather than directly using the database.
`getEnvironmentInfo`	This method returns a `MiEnvironmentInfo` object. Using this, it is possible to get information about the current user session, for example the current user name.

Method	Remarks
`getApiProvider`	This method returns a type which can give access to a "(Maconomy) server API" and a "(Maconomy) database API." Of these two, the database API is the one that is most frequently used. The database API of type `MiDatabaseApi` is obtained through the method `getDatabaseApi`.
`getCallbacks`	This method returns an object which provides access to certain out-of-context callbacks. Currently, only one such call-back is supported: a call-back which will enforce the workspace client to render itself in test-mode. This call-back may be useful to invoke if you make "temporary hacks" that are not supposed to ever work in a production system!

The abstract data-model implementations offers convenience methods for obtaining the general API-provider and the database API. These are obtained through the data-model methods `getApiProvider` and `getDatabaseApi` respectively.

6.2.1 Accessing Environment Information

The environment information is obtained from the `resources` object by invoking the method `getEnvironmentInfo`. The returned type, `MiEnvironmentInfo` provides the following methods:

Method	Remarks
`getDatabaseShortName`	This method returns the database short name to which the user is connected.
`getLanguage`	This method returns a `MiKey` representing a language code. This language code is the language used by the client-side.
`getUserName`	Returns the Maconomy user-name of the current user as a `String`.
`getUserNameVal`	Returns the Maconomy user-name of the current user as a `McStringDataValue`.
`getUserLoginName`	Returns the Maconomy log-in name of the current user as a `String`.
`getUserLoginNameVal`	Returns the Maconomy log-in name of the current user as a `McStringDataValue`.
`getRoleInstanceKey`	Returns the instance key of the Maconomy user-role of the current log-in's user-role as a `String`.
`getRoleInstanceKeyVal`	Returns the instance key of the Maconomy user-role of the current log-in's user-role as a `McStringDataValue`.

Method	Remarks
`getRoleName`	Returns the name of the Maconomy user-role of the current log-in's user-role as a `String`.
`getRoleNameVal`	Returns the name of the Maconomy user-role of the current log-in's user-role as a `McStringDataValue`.

Listing 6.13 shows an example of an extension that adds an `Approve` action to some container. In addition, three fields are added. These fields indicate whether or not an approval has taken place, *who* did the approval and what date the approval was made. In order to do this, we need access to the current user name. This is done by accessing the current user name from the environment information which is made available through the container runner in line 43: the field `ApprovedBy` is assigned the value of the current user in the handler of the `Approve` action.

Listing 6.13: Accessing the Curernt User Name.

```
2   private static final MiKey NS = key("Trifolium");
3   private static final MiKey APPROVED =
4       NS.concat(":Approved");
5   private static final MiKey APPROVED_BY =
6       NS.concat(":ApprovedBy");
7   private static final MiKey APPROVAL_DATE =
8       NS.concat(":ApprovalDate");
9   public EnvironmentInfoExample(final MiResources resources) {
10    super(resources);
11  }
12
13  @Override
14  public MiKey defineNamespace() { return NS; }
15
16  @Override
17  public MiPersistenceStrategy definePersistenceStrategy(final
        MiContainerRunner.MiDefine containerRunner) {
18    return McMolPersistenceStrategy.create(key("TRI_MyTable"),
          getApiProvider());
19  }
20
21  @Override
22  public MiExtended defineDomesticSpec(final MiDefine
        containerRunner) {
23    return McPaneSpec.McExtended.pane()
24      .addBooleanField(APPROVED, "Approved").then()
25      .addStringField(APPROVED_BY, "Approved By").then()
26      .addDateField(APPROVAL_DATE, "Date Approved").then()
27
28      .addAction("Approve", "Approve")
```

```
29          .end();
30      }
31
32
33      @Action("Trifolium:Approve")
34      final class ApproveHandler extends McAbstractDataModelRootAction
            {
35        @Override
36        public void onAction(final MiActionPost containerRunner,
37                             final MiAction eventData) throws
                                Exception {
38          final MiDataValues resultData = eventData.getResultData();
39
40          resultData
41            .setBool(APPROVED, true)
42            .setStr(APPROVED_BY,
43                    containerRunner.getEnvironmentInfo().getUserName()
                        )
44            .setDate(APPROVAL_DATE, McDate.today());
45        }
46      }
```

6.2.2 Accessing the Maconomy Database

Using the method `getApiProvider` on the `resources` object, you can invoke another method called `getDatabaseApi`. Or you may obtain the database API directly by invoking the data-model method `getDatabaseApi`. Finally, the `containerRunner` parameter has a method, `getDatabaseApi`, that returns the database API. This method returns an object of type `MiDatabaseApi`, and it represents an interface to the Maconomy database. This type offers access to a number of methods:

Method	Remarks
`mselect`	This method offers the possibility of fetching data from the Maconomy database using either normal database tables or Maconomy universes using MQL [MQL]. `mselect` should be preferred over `select` and `sql` if at all possible. By doing so, all data fields are returned with the correct Maconomy types. When `mselect` is *not* used, popups are returned as integers, and amounts are returned as reals on SQL-Server. Also, in the where-clause, Maconomy types can be used, rather than having to use the type used internally in the database! However, if your table (e.g., a custom MOL table) contains `String` fields longer than 255 characters, `mselect` *cannot be used*! The `mselect` method is available in a number of variants, including variants that make it possible to write syntax more or less resembling SQL syntax. We encourage making use of these variants since they are easier to read and write, and since the compiler can help ensuring that your expressions are well-formatted. See below for examples.
`mcount`	This method is used to count the number of entries of database tables and/or universes. Like `mselect` should be preferred over `select`, `mcount` should be preferred over `count`.
`mexists`	This method is used to check whether records with certain properties exist. Like `mselect` should be preferred over `select`, `mexists` should be preferred over `exists`.
`count`	This method is similar to `mcount` except that it uses the native database types rather than Maconomy types.
`exists`	This method is similar to `mexists` except that it uses the native database types rather than Maconomy types.
`insert`	This method is used to do an `insert into` operation in the database. While it is not possible to modify the standard application database tables, you can use this method to add records to custom MOL tables. Remember that you should *not* manually insert/update/delete tables used as primary data-keepers when implementing data-models! The `insert` method is available in a number of variants, including variants that make it possible to write syntax more or less resembling SQL syntax. We encourage making use of these variants since they are easier to read and write.

Method	Remarks
`update`	This method is used to do an `update` operation in the database. While it is not possible to modify the standard application database tables, you can use this method to add records to custom MOL tables. Remember that you should *not* manually insert/update/delete tables used as primary data-keepers when implementing data-models! The `update` method is available in a number of variants, including variants that make it possible to write syntax more or less resembling SQL syntax. We encourage making use of these variants since they are easier to read and write, and since the compiler can help ensuring that your expressions are well-formatted.
`delete`	This method is used to do a `delete from` operation in the database. While it is not possible to modify the standard application database tables, you can use this method to add records to custom MOL tables. Remember that you should *not* manually insert/update/delete tables used as primary data-keepers when implementing data-models! The `delete` method is available in a number of variants, including variants that make it possible to write syntax more or less resembling SQL syntax. We encourage making use of these variants since they are easier to read and write, and since the compiler can help ensuring that your expressions are well-formatted.
`getDatabaseType`	This method returns en enum indicating the type of database you are connected to. At present `ORACLE` or `SQL_SERVER`. Sometimes such information may be needed, especially when doing "raw" sql.
`select`	This method is similar to `mselect` except that it does not take Maconomy data-types into account. Some types in the result are represented as Maconomy types rather than the type used internally. This goes for `String`, `Integer`, `Real`, `Bool`, `Date`, `Time`. However, `Popup` fields are returned as integers representing the ordinal value and `Amount` fields may be returned as reals on some databases. The `select` method is available in a number of variants, including variants that make it possible to write syntax more or less resembling SQL syntax. We encourage making use of these variants since they are easier to read and write, and since the compiler can help ensuring that your expressions are well-formatted. See below for examples.

Method	Remarks
`sqlBuilder`	This method is used to construct SQL based on a textual value. The SQL builder allows the programmer to build SQL which is safe with respect to SQL injection. This method returns a builder object which lets you declare values using placeholders. Such values will be properly formatted and escaped according to the actual type of each value. In addition, the SQL builder contains a wide variety of methods that can make it much easier to build value lists such as `in`-expressions, field lists, `like`-patterns etc.
`sql`	This method takes an `McSql` object as argument. Such objects are constructed by using the `sqlBuilder` method (see above.)
`getPopupValues`	This method returns a list of all popup values of a given Maconomy popup type.
~~`commit`~~ ~~`rollback`~~ ~~`getAutoCommit`~~ ~~`setAutoCommit`~~	These methods must not be used by extension programmers!

In earlier version of the Extension Framework writing MQL [MQL] was somewhat cumbersome, because you had to provide all information needed to execute a query as arguments to the `mselect` method. While you can still do that (see Listing 6.15 below), it is possible to specify the database queries in a way that is easier to both read and write. And which lets the compiler help ensuring that where-clauses are meaningful.

This is done by having the `mselect` method return a "builder-style" object. That is, an object that you can operate on to declare your query. Once you are done, all you need to do is to invoke the `getResult` method to execute the query and obtain the result.

Listing 6.14 shows an examples that uses this "builder-style" `mselect` query. The examples illustrates the implementation of a custom action associated with the `maconomy:CompanyInfo` container: upon invoking the action `ShowEmployeeStatistics`, a small "report" (text document) will be generated. This report will show the number of male and female employees currently working in the specific company. Also, a list of zip codes will be presented, and for each zip code, it will show how many of the company's active employees are associated with that zip code. In order to do that, we want to query the database to extract employees that:

- Are associated with this company
- Are not blocked

- Are having an employment date that is at most today's date. Hence, employees that are scheduled to be employed "in the future" will not be considered.
- Are still working for this company, i.e., where the employment ending date is either not specified, or is later than today's date.

In the code, such employees are obtained by the "`mselect` statement" in lines 15–22. Notice that the syntax is pretty close to that of "obvious SQL." Obviously, the exact same syntax cannot be achieved because we must adhere to general Java syntax as well. Also, notice that methods must be used for relational operators such as `<=` (`le`) and `>` (`gt`). Please see the method list below. Also notice that it is possible to create quite complex embedded expression structures (e.g., `or`) embedded into the expressions. In line 15 the field list and the table from where data is fetched is being declared. In line 16 the where-clause is started by the method `where`. Subsequent additions to the where clause are declared using the `and` method, first time in line 17. It is possible to optionally declare order by declarations after the last part of the where clause. This is done by the `orderBy` method, including a method (`asc`) declaring the ordering direction. Finally, the query is declared as "done," and executed by invoking the `getResult` method. The result of an `mselect` is a type called `MiQueryInspector`. Such an inspector provides a number of methods to query the result. The type itself makes it possible to iterate through the records (each represented as a `MiValueInspector`). This is particularly nice when using Javas `for-each` construct. This is shown in line 29. The remaining part of the code shown is just plain Java producing the report and presenting a document to the end-user.

Listing 6.14: Accessing the Database Using MQL "Builder"

```
2   @Action("ShowEmployeeStatistics")
3   private final class ShowContactInfoHandler extends
        McAbstractDataModelRootAction {
4     private static final String NL = "\r\n";
5
6     /** {@inheritDoc} */
7     @Override
8     public void onAction(final MiActionPost containerRunner,
9                          final MiAction eventData) throws
                               Exception {
10
11      final MiValueInspector originalData = eventData.
            getOriginalData();
12
13      final McDateDataValue today = McDate.today();
14      final MiQueryInspector currentEmployees = getDatabaseApi()
15        .mselect("Gender", "ZipCode").from("Employee")
16        .where(originalData.copyValues("CompanyNumber"))
17        .and().not("Blocked")
18        .and().le("DateEmployed").date(today)
19        .and(or(gt("DateEndEmployment").date(today),
```



```
20                  eq("DateEndEmployment").nullDate()))
21          .orderBy("ZipCode").asc()
22          .getResult();
23
24        // Now process the result of the query
25        final MiMap<String, Integer> zipCodeCounts = McTypeSafe.
              createLinkedHashMap();
26        int females = 0;
27        // First iterate over all employees, thereby obtaining a
              count
28        // for each ZipCode and a count of females
29        for (final MiValueInspector currentEmpl : currentEmployees)
30        {
31          final String zipCode = currentEmpl.getStr("ZipCode");
32          zipCodeCounts.putTS(zipCode,
33                              1 + zipCodeCounts.getElseTS(zipCode,
                                    0));
34          // ordinal 1 is Female
35          if (currentEmpl.getPopupOrdinal("Gender") == 1) {
36            ++females;
37          }
38        }
39        // The males are assumed to be the total number minus
40        // the number of females
41        final int males = currentEmployees.getRowCount() - females;
42
43        // produce a report document
44        final StringBuilder output = new StringBuilder()
45          .append("Statistics for Employees of Company ")
46          .append(originalData.getStr("CompanyNumber"))
47          .append(NL).append(NL)
48          .append("No. of male employees: ").append(males)
49          .append(NL)
50          .append("No. of females: ").append(females)
51          .append(NL).append(NL)
52          .append("No. of Employees in Zip Codes")
53          .append(NL);
54        // iterate over all the ZipCodes found and add a
55        // line to the report
56        for (final Map.Entry<String, Integer> entry : zipCodeCounts.
              entrySetTS()) {
57          final String zipCode = entry.getKey();
58          final int count = entry.getValue();
59          output.append(zipCode)
60                .append(": ")
61                .append(count)
62                .append(NL);
63        }
64
```

```
65        // generate a file resource and show the contents
66        final McFileResource outputFile =
67          new McFileResource(key("Stats"),
68                             McFileResource.MeType.TEXT,
69                             "Generated by " + this.getClass(),
70                             output.toString());
71        containerRunner.document().show(outputFile);
72      }
73    }
```

Constructing `select` statements (i.e., non-MQL queries) is done using an interface identical to that of `mselect`. The `mselect`/`select` methods takes a list of fields that should be comprised by the query result. These can be provided in a number of ways:

- As a comma-separated list of `MiKey` arguments
- As a comma-separated list of `String` arguments. This is a convenient way if the field names are inlined (i.e., not defined as constants).
- As an `Iterable` of `MiKeys`, such as a list, a set or any other `Collection` structure.
- As a `MiValueInspector`, meaning all fields comprised by that value inspector should be selected.

Basic Query-Builder Methods

The result-type of invoking the `mselect`/`select` provides access to the following methods:

Method	Remarks
`mselect`	Using this method, you can specify additional fields to be selected. You can invoke this method as many times as you like. This may be useful in cases where the fields are not found in *one* collection, or in cases where some fields are defined as `MiKeys` and others are simply inlined as `Strings`. For example: `mselect(F, G).mselect("xyz")` which selects the fields represented by the constants `F` and `G` as well as the field `xyz`.
`from`	This method specifies the table/universe on which the query is based. You can provide the name as a `MiKey` or as a `String`. Once this method has been invoked, you can no longer specify field-selections. Instead, you must specify the `where`-part.

Method	Remarks
`where`	After having specified the `from` table, you may optionally specify a `where`-clause. If you need to specify an ordering or a pageing interval, you must specify a `where`-clause (possibly a constantly `true` expression). You can also directly invoke the `getResult` to execute the query and obtain the result. In that case, the `where`-clause will be considered `true`. The `where` method is found in a number of variants: • A version that takes an `MiExpression` as argument. This expression will then be considered the `where`-clause. • A version that takes an `MiValueInspector`. In this case, the where-clause will be considered an expression comprising a number of "field-equals-value" sub-expressions. For example, if the provided `MiValueInspector` comprises three fields `a`, `b` and `c` with values v_a, v_b and v_c respectively, then the where-clause will be considered to be $$\mathtt{a} = v_a \wedge \mathtt{b} = v_b \wedge \mathtt{c} = v_c$$ • A version that takes a (boolean) field name as an argument, either as an `MiKey` or as a `String`. This will be considered a `where`-clause stating that the value of the specified field is `true`. • A zero-argument version that gives access to building a simple expression (see below.) After the `where` method is invoked, you can either refine the `where`-clause further by invoking the method `and`, or you can execute the query and obtain the result by invoking the `getResult` method. You also have the option to specify an order-by clause or to restrict the range of records returned (see below).
`and`	This method is similar to `where` except that it is used to further refine a `where`-clause by adding additional constraints (that are `and`'ed together). You can invoke the `and` method as many times as you like. Listing 6.14 shows an example of the use of the `and` method in line 17.

Method	Remarks
`orderBy`	This method can be optionally invoked after the `where`/`and` methods. You can invoke the `orderBy` method as many times as you want. The methods comes in a number of variants: • A variant that takes the field name on which sorting should be performed as an argument, either as an `MiKey` or a `String` argument. With this variant, you must immediately after invoke one of the methods `asc` or `desc` to indicate whether you want sorting to be ascending or descending. • A variant that takes the field name on which sorting should be performed and an argument that specified the sort-order as an argument of type `McSortOrder`. `MeSortType`. • A variant that takes one or more arguments of type `McSortOrder`.
`allRows`	This method specifies that you want all rows matching the query's where-clause. Since this is also the default, you don't need to specify this. Sometimes, however, it can be nice for code clarity.
`firstRow`	This method specifies that you want at most one row back: the first. If no rows match the query, no rows will be returned. This is equivalent to specifying `rows(0,1)`.
`row`	This method specifies that you want at most one row back: the one having the specified index (0 denotes the first row.) If the number of rows matching the query is less than or equal to the specified index, no rows will be returned. Invoking `row(index)` is equivalent to specifying `rows(index, 1)`.
`rowsTo`	This method specifies that you want the first `index`−1 rows that matches the query. Invoking `rowsTo(index)` is equivalent to specifying `rows(0, index - 1)`.
`rowsFrom`	This method specifies that you want the rows from (and including) the row with a specified `index`, hence skipping the first `index` − 1 rows. Invoking `rowsTo(index)` is equivalent to specifying `rows(index, 0)`.
`rows`	This method specifies that you want a number of specified rows starting at a specified index. If the number of specified rows is 0, it is interpreted as "all rows." the rows from (and including) the row with a specified `index`, hence skipping the first `index` − 1 rows. For example, if a given query is matched by 32 rows then `rows(30, 10)` will return the two rows with index 30 and 31.

Building `where`-Clause Expressions

A very important part of queries is obviously the `where`-clause. For this reason, the methods `where` and `and` mentioned above have variants that makes it easy to build

simple expressions which are then and'ed together. The no-argument versions of `where` and `and` returns objects that are used to construct such expressions. Once the expression has been specified, the return type is again the main query-builder type, allowing the programmer to continue building the query. The expression methods are:

Method	Remarks
`not`	This method is found in a number of variants • A variant that takes an `MiExpression` as argument. The resulting expression is the negation of that expression. • A variant that takes a (boolean) field name as an argument. The resulting expression corresponds to an expression stating that the value of the field is `false`.

Method	Remarks

`eq`	This method is used to specify a simple equality expression like
	`fieldName` = *value*
	The `eq` methods takes the `fieldName` as an argument (either as an `MiKey` or as a `String`). The returned value gives the possibility to specify the value using one of the following methods:
	• `val` specifies the value as a generic `McDataValue`
	• `str` specifies a string value
	• `integer` specifies an integer value
	• `real` specifies a real value
	• `amount` specifies an amount value
	• `date` specifies a date value, including a variant that specifies the year, month and date as integer values.
	• `time` specifies a time value, including a variant that specifies the hour, minute and second as integer values.
	• `bool` specifies a boolean value
	• `popup` specifies a popup value, including a variant that specifies the type name, literal name and ordinal value as separate arguments
	• `nullDate` specifies the empty date (`nullDate`.)
	• `nullTime` specifies the empty time (`nullTime`.)
	• `nil` specifies the empty popup value (`nil`) of a popup type with a given type name
	• `field` specifies the name of another field that must equal the first field, i.e., used for expressions of the form
	$\texttt{fieldName}_1 = \texttt{fieldName}_2$
	An example of using the `eq` method is:
	`// ProjectManagerNumber = '1205'`
	`eq("ProjectManagerNumber").str("1205")`

Method	Remarks

`neq`	This method is similar to `eq` above except that it is used to specify a simple non-equality expression like
	`fieldName` $\neq$ *value*
	An example of using the `neq` method is:
	`// CostPrice <> 0.00`
	`neq("CostPrice").amount(BigDecimal.ZERO)`

`ge`	This method is similar to `eq` above except that it is used to specify a simple inequality expression ("greater than or equals to") like
	`fieldName` $\geq$ *value*
	An example of using the `ge` method is:
	`// ProjectManagerNumber >= today`
	`ge("EmploymentDate").dage(McDate.today())`

`le`	This method is similar to `eq` above except that it is used to specify a simple inequality expression ("less than or equals to") like
	`fieldName` $\leq$ *value*
	An example of using the `le` method is:
	`// PeriodStart <= today`
	`eq("PeriodStart").date(McDate.today())`

`gt`	This method is similar to `eq` above except that it is used to specify a simple inequality expression ("greater than") like
	`fieldName` $>$ *value*
	An example of using the `gt` method is:
	`// Difference > trivialityLimit`
	`gt("Difference").real(trivialityLimit)`

Method	Remarks

`lt`	This method is similar to `eq` above except that it is used to specify a simple inequality expression ("less than") like
	
	`fieldName` $<$ *value*
	
	An example of using the `gt` method is:
	`// RevisionNumber < lastesApprovedRevNo`
	`lt("RevisionNumber").integer(latestApprovedRevNo)`

`rel`	This method is similar to `eq` above except that you must specify which relational operator to use. The relational operator can be specified using either en `enum` value or a `String`. The recognized `String`s are as follows:
	• `=` use for equality
	• `!=` or `<>` used for non-equality
	• `>=` used for greater-than-or-equals
	• `<=` used for less-than-or-equals
	• `>` used for greater-than
	• `<` used for less-than
	Examples are:
	`// ProjectManagerNumber = '1205'`
	`rel("ProjectManagerNumber", "=").str("1205")`
	`// PeriodStart <= today`
	`rel("PeriodStart", "<=").date(McDate.today())`

Method	Remarks
`inrange`	This method is used to specify that the value of a field is within a certain range. The range may be unbounded. The argument to the `inrange` method is a field name. In addition, lower and upper bound values must be specified by invoking a method on the result object. If the `inrange` expression is associated with the field `fieldName` and two values v_l (lower bound) and v_u (upper bound.) The corresponding expression is then: $$\begin{cases} \texttt{fieldName} \leq v_u & \text{if } v_l = empty \wedge v_u \neq empty \\ \texttt{fieldName} \geq v_l & \text{if } v_l \neq empty \wedge v_u = empty \\ v_l \leq \texttt{fieldName} \leq v_u & \text{if } v_l \neq empty \wedge v_u \neq empty \\ \texttt{true} & \text{if } v_l = empty \wedge v_u = empty \end{cases}$$ The "empty" value is different from type to type. For integer, reals and amounts, the "empty" value is defined as 0. For booleans, an empty value does not exit. A number of methods are available for specifying the lower and upper bound range: • `val` specifies the lower and upper bounds as generic `McDataValues` • `str` specifies the lower and upper bounds as strings • `integer` specifies the lower and upper bounds as integers • `real` specifies the lower and upper bounds as reals • `amount` specifies the lower and upper bounds as amounts • `date` specifies the lower and upper bounds as dates • `time` specifies the lower and upper bounds as time values Using the `inrange` method on popup values may give somewhat surprising results since the *ordinal values* are used when determining the bounds.

In addition to making simple expressions that are and'ed together, sometimes you might want to build more complex expressions, for example expressions containing disjunctions (or-expressions.) This is not directly possible using the simple query builder methods. The Extension Framework does, however, provide support for doing this. All data-models and action-handler abstract classes have a super-type implementation called `McAbstractSimpleExpressionBuilder`. This type provides methods for building simple expressions (as shown above) as well as simple `or`-expressions. By combining the available methods, you can build complex expressions in a reasonably easy-to-read and

understand syntax. For example, if you want a complex expression like

$$\neg a \vee b = v_1 \wedge (d \leq v_2 \vee e > v_3)$$

you can easily obtain this by the following piece of code:

```
or(not("a"),
    and(eq("b").val(v1),
        or(le("d").val(v2),
            gt("e").val(v3))))
```

Notice, that such expression builder-methods can be used for *any* need for expressions; it is not restricted to database queries. Sometimes, you may want to implement utility classes that are not necessarily defined in a data-model or action handler class. In this case, obtaining access to these expression builder utilities can be achieved in two ways:

1. Let your class extend `McAbstractSimpleExpressionBuilder`. Doing so, will enable a wide range of methods that are capable of creating simple boolean expressions. By combining these, arbitrarily complex boolean expression can be build.
2. If your class cannot extend that class, you may locally obtain an object of type `McSimpleExpressionBuilder`. You can obtain such an object by invoking the factory method `expr` on `McSimpleExpressionBuilder`. The returned object gives access to all the expression builder-methods mentioned above.

Examining a Query Result

As indicated above, a query is executed, and the result obtained by invoking the method `getResult`. The result of doing that for `mselect`/`select` methods is an object of type `MiQueryInspector`. This type implements the interface allowing you to use it with Java's `for-each` construct: `Iterable<MiValueInspector>`. This makes it very convenient to loop loop through all rows in the result, providing access to each row as a `MiValueInspector`, obviously skipping the loop if there are no rows in the result. An example is given in Listing 6.14 line 29. Apart from this, the return type provides the following methods

Method	Remarks
`getRowCount`	This method returns the number of rows returned by the query.
`containsRow`	This method returns a boolean indicating whether a row with a given index is comprised by the result.
`isEmpty`	This method returns `true` if there are no records in the result.

Method	Remarks
`getRecordOpt`	This method returns an optional record. If no row with the specified index exists, a `none`-value is returned. Otherwise the record value is returned.
`getRecord`	This method is like `getRecordOpt` except that it is asserted that the record is defined. If the records is not defined, an exception is thrown. The result records are always treated as being 0-indexed. So, even if the result is specified as rows from index i, the first record in the *result* will have index 0.
`allRecords`	This method returns a `Collection` of all records represented by the result.
`getProjector`	This returns a an object of type `MiQueryInspector.MiProjector` which is tied to this query inspector. It can be used to make projections of the current query inspector, i.e., considering only one or more fields in the return result. By doing so, it is possible to extract, for example, the unique values occuring for a specified field in the result.
`asPaneValue`	This method represents the result data as a `MiPaneValue`, similar to the older versions of the `mselect`/`select` methods. Usually, it should not be necessary or desirable to obtain the value as this type. Also notice, that the value returned by this mehtod may have a record-index offset which is not 0. Since a query inspector is essentially communicating a collection of records, this method (which reveals a lot of irrelevant details) is now deprecated. Instead use the method on the query inspector to examine the data.

The `MiQueryInspector.MiProjector` type that is returned by the `getProjector` method has the following methods:

Method	Remarks
`getUniqueVals`	This method returns a collection of unique values occurring in the query result for some specified field. For example, if you select a number of time sheet lines, you can obtain the unique set of job numbers referenced by those time sheet lines.
`getVals`	This method returns a collection of values occurring in the query result for some specified field. In case a specific value occurs multiple times, the returned collection will include that value several times.

Method	Remarks
`getUniqueValues`	This method returns a collection of unique combinations of values (i.e., of multiple fields) occurring in the query result for some specified fields. For example, if you select a number of time sheet lines, you can obtain the unique set of combinations of job number and task name referenced by those time sheet lines.
`getValues`	This method returns a collection of combinations of values (i.e., of multiple fields) occurring in the query result for some specified fields. In case a specific combination of values occurs multiple times, the returned collection will include that value combination several times.

Using the Old-Fashioned Query Interface

In previous version of the Extension Framework, doing database queries were done in a slightly different way. This way is still available, and in some specific cases, it may still be preferable. Likely, if you build very generic code for which the behavior is based on some kind of configuration or specification, you may benefit from using this interface. In most sitautions, however, the code is likely going to be much easier to read and write using the builder-style methods above.

Listing 6.15 shows an example where data is read from the database using MQL [MQL] using the `mselect` method using the non-builder-style interface. In order to do that, we need three things:

1. The name of the table/universe to read from.
2. A collection of fields to select (e.g., a list of field names.)
3. An `McQuery` which is basically an expression, and may additionally contain sort order and row range (paging information.)

In the example, an action handler implements some functionality that shows a message containing the project manager's e-mail address as well as his phone number. This information needs to be looked up from the `Employee` database table. First, in line 14, we make an expression[3] that is going to be used as the where clause, and then wraps this expression into a `McQuery`. In line 18 we invoke `mselect` on the database API which is obtained from the method `getDatabaseApi` made available in the data model classes

[3]In the example we use a `McExpressionParser` which parses an expression from a `String`. The syntax of such a `String` is identical to that of expressions in MDML and MWSL. See [EL]. There are several other ways to build expressions, e.g., from sub-sets of record values.

as well as from the `containerRunner` of the event method. The result of this (and most other database operations) is a "pane value", i.e., a structure similar to that used to present data to panes in the client. Such structures can be queried for the number of records, and it's possible to iterate over all records or to obtain the value of specific records etc. In lines 20–28 we investigate the number of rows; there will be either 0 or 1 in this case. If there is one row, we obtain that row in line 21 and use it to obtain the desired information.

Listing 6.15: Accessing the Database Using MQL.

```
2   public DatabaseExampleDataModel(final MiResources resources) {
3     super(resources);
4   }
5
6   @Action("ShowContactInfo")
7   private final class ShowContactInfoHandler extends
       McAbstractDataModelRootAction {
8     @Override
9     public void onAction(final MiActionPost containerRunner,
10                         final MiAction eventData) throws
                               Exception {
11      final MiValueInspector originalData = eventData.
           getOriginalData();
12
13      final MiExpression<McBooleanDataValue> pmEmployeeExpr =
14        McExpressionParser.parser("EmployeeNumber = '" + escapeStr
             (originalData.getStr("ProjectManagerNumber")) + "'").
             parse();
15      final MiList<MiKey> selectFields =
16        McTypeSafe.createArrayList(key("ElectronicMailAddress"),
17                                   key("Telephone"));
18      final MiQueryInspector mselectResult = getDatabaseApi().
           mselect(selectFields).from(key("Employee")).where(
           pmEmployeeExpr).getResult();
19
20      if (mselectResult.getRowCount() == 1) {
21        final MiValueInspector pm = mselectResult.getRecord(0);
22        final String message =
23          String.format("You can reach the PM at %s or phone: %s",
24                        pm.getStr("ElectronicMailAddress"),
25                        pm.getStr("Telephone"));
26
27        containerRunner.call().notification(message);
28      }
29    }
30  }
31
32  private String escapeStr(final String s) {
33    return s.replaceAll("'", "\\'");
```

Listing 6.17 shows an example where data is obtained from the database by using SQL directly. In this example, a Variable is added to the Listing 6.17 shows an example where data is obtained from the database by using SQL directly. In this example, a variable is added to the Just to compare, Listing 6.16 shows the exact same code, this time using the builder-style methods. Which version do you prefer?

Listing 6.16: Accessing the Database Using SQL (Builder-Style).

```
2   @Action("ShowContactInfo")
3   private final class ShowContactInfoHandler extends
        McAbstractDataModelRootAction {
4     @Override
5     public void onAction(final MiActionPost containerRunner,
6                          final MiAction eventData) throws
                               Exception {
7       final MiValueInspector originalData = eventData.
            getOriginalData();
8
9       final String pmNumber =
10        originalData.getStr("ProjectManagerNumber");
11      final MiQueryInspector mselectResult =
12        getDatabaseApi()
13          .mselect("ElectronicMailAddress", "Telephone")
14          .from("Employee")
15          .where().eq("EmployeeNumber").str(pmNumber)
16          .getResult();
17
18      if (mselectResult.containsRow(0)) {
19        final MiValueInspector pm = mselectResult.getRecord(0);
20        final String message =
21          String.format("You can reach the PM at %s or phone: %s",
22                        pm.getStr("ElectronicMailAddress"),
23                        pm.getStr("Telephone"));
24
25        containerRunner.call().notification(message);
26      }
27    }
```

Listing 6.17 shows an example where data is obtained from the database by using SQL directly. In this example, a variable is added to the `TimeSheets` container. This variable shows the number of hours registered for this employee from the beginning of the year until (but not including) the current time sheet. This is done by selecting the sum of the fields `ExternalTimeWeekTotal` and `InternalTimeWeekTotal` on time sheets of this employee, having the same year as this time sheet and a week number which is lower than that of this time sheet. When using SQL directly, we need to build an SQL query. This is done using the `McSqlBuilder` class. The best way to obtain such an object is by invoking the `sqlBuilder` on the `MiDatabaseApi`. The builder is generally a textual interface to writing SQL statements. However, because it is generally difficult

not to have your code being vulnerable to SQL injection, this textual interface insists on having control over variable content and String-values included in the SQL. Instead of letting the programmer have to remember to quote and escape String-values, format dates etc., this SQL builder api will do it for you. In order to reference a value, you put a *placeholder* in your SQL-string. By declaring the value of that placeholder (also on the builder object), once you invoke the `build` method, the framework will substitute the placeholders by the declared values, and it will ensure that each type is adequately formatted and escaped. This makes it very difficult to make your code vulnerable to SQL injection. It is, however, important to stress that when doing textual SQL, you must always take care. Deltek cannot be held liable for any vulnerability in that occurs from your code.

In the example, building an SQL query is done in lines 22–29. Line 28 is where values are assigned to placeholders. You can do that either by declaring the placeholder values individually, or you can do it (as in this example) by invoking the `setAll` method which assigns all the values in the provided record structure to placeholders of that name. Notice that you don't need to actually *use* all placeholders in your query. Adding multiple placeholder values in this way, simply mean that you are able to refer to all these names as placeholders in your SQL String. Placeholders have the form

$$\text{\^{}\{}\textit{Placeholder\_Name}\text{\}}$$

When the builder object is built using the `build` method, the result is an object of type `McSql`. This SQL content is now locked in this object, and cannot be changed. By invoking the `getResult` method, the SQL is executed and the result is brought back as a `MiQueryInspector`. In the example, the SQL statement is run in line 30. Optionally, you may provide a range (page indication) as an argument to the `getResult` method. In lines 32–37 we check the number of records in the output (there will be 0 or 1.) In case the result is non-empty (i.e., there is one), the resulting data is fetched from the row with index 0 in line 33. Notice that when using this API, properly guarding against SQL-injection is handled for you, just as formatting dates and other types is handled for you.

Listing 6.17: Accessing the Database Using SQL.

```
2   private static final MiKey NS = key("Trifolium");
3   private static final MiKey REG_YEAR_TO_WEEK_VAR =
4       NS.concat(":RegYearToWeekVar");
5
6   @Override
7   public MiKey defineNamespace() { return NS; }
8
9   @Override
10  public MiExtended defineDomesticSpec(final MiDefine
        containerRunner) {
11    return McPaneSpec.McExtended.pane()
```

```
12        .addRealVariable(REG_YEAR_TO_WEEK_VAR, "Reg. Year-to-Now").
              then()
13        .end();
14    }
15
16    @Override
17    public void refreshVariables(final MiContainerRunner.MiDataPost
          containerRunner,
18                                 final MiResult eventData) throws
                                       Exception {
19      final MiDataValues resultData = eventData.getResultData();
20      final MiDatabaseApi db = containerRunner.getDatabaseApi();
21      final McSql query = db
22        .sqlBuilder("select sum(ExternalTimeWeekTotal) as ExtTime,")
23        .append(    "sum(InternalTimeWeekTotal) as IntTime").nl()
24        .append(    "from TimeSheetHeader").nl()
25        .append(    "where EmployeeNumber = ^{EmployeeNumber}").nl()
26        .append(    "      and WeekNumber < ^{WeekNumber}").nl()
27        .append(    "      and TheYear = ^{TheYear}")
28        .setAll(resultData)
29        .build();
30      final MiQueryInspector sqlResult = query.getResult();
31
32      if (!sqlResult.isEmpty()) {
33        final MiRecordInspector addedTimeSheets = sqlResult.
              getRecord(0);
34        resultData.setReal(REG_YEAR_TO_WEEK_VAR,
35                           addedTimeSheets.getReal("ExtTime")
36                           .add(addedTimeSheets.getReal("IntTime")))
                                 ;
37      } else {
38        resultData.setReal(REG_YEAR_TO_WEEK_VAR, BigDecimal.ZERO);
39      }
```

The SQL builder used in the example, contains a number of methods that eases the task of wring SQL, generated programmatically from some data structures, or as more or less explicit SQL. These include:

Method	Remarks
`append`	This method appends a String of "raw SQL" to the content already contained by the builder. The SQL builder will ensure that there's always space between the content of subsequent build methods. Hence, you can only append new content at places where a space is required or does not matter. Therefore, you cannot split keywords such as `append("sel").append("ect")`. It also means, that you don't have to worry about silly mistakes pertaining from missing white-space separation. The content in the "raw SQL" may contain placeholders of the form `^{`*PlaceholderName*`}`. When the `build` method is invoked, the placeholders will be replaced by content obtained from the declared placeholder values. Placeholders of different kinds can be declared: values, identifiers, patterns and expressions. See below for more information. It is *not allowed* to specify the following characters in the raw SQL: single quote (`'`), semicolon (`;`), double-dash (comment) (`--`), hat (`^`) and curly braces (`{` and `}`). In case you need a literal hat or curly brace, you can obtain that by writing it twice. Single-quotes *must* be left to the framework.
`nl`	This method appends a newline (platform dependent) to the SQL content. It has not semantic meaning to the generated SQL, but it may be used to conveniently produce readable logging or debugging content.
`setVal`	This method declares the value of a placeholder. You specify the placeholder name as well as its value. Such a placeholder is a value placeholder, and the corresponding placeholder will be substituted by the formatted and escaped value. For example, if the value is a date, the date will be properly formatted. A boolean will be formatted to match the encoding of booleans in the database, and Strings are quoted and quotes inside the string contents is escaped.
`setBool`	This method is like `setVal` except that the argument type must be a boolean.
`setStr`	This method is like `setVal` except that the argument type must be a String. The length of the string may be truncated.
`setString`	This method is like `setVal` except that the argument type must be a String. The string contents will not be truncated.
`setPopup`	This method is like `setVal` except that the argument type must be a popup.
`setNil`	This method is like `setPopup` except that assigns a popup `nil`-value to the specified placeholder.

Method	Remarks
`setInt`	This method is like `setVal` except that the argument type must be an integer.
`setDate`	This method is like `setVal` except that the argument type must be a date.
`setNullDate`	This method is like `setDate` except that assigns the `nullDate` value to the specified placeholder.
`setTime`	This method is like `setVal` except that the argument type must be a time value.
`setNullTime`	This method is like `setTime` except that assigns the `nullTime` value to the specified placeholder.
`setReal`	This method is like `setVal` except that the argument type must be a real.
`setAmount`	This method is like `setVal` except that the argument type must be an amount.
`setAll`	This method takes a record value/value inspector as argument. The effect is that all fields of that record will be declared as placeholders with that name and the corresponding value will be the one found in the record. An additional variant of this method exists, that also lets you declare a *context*. Doing so will imply that all placeholder names are prefixed by the context name and a "dot". E.g., if the the record contains two fields `FieldA` and `FieldB` and the context is `empl` then the placeholders are: `empl.FieldA` and `empl.FieldB`. This is useful in cases where you want to add placeholders from different records with similar names (e.g., a customer and an employee both contain the fields `Name1` and `CompanyNumber`.) By using the context, you can easily separate the values from the two contexts. The framework will issue an error if a placeholder with a given name is declared twice.

Method	Remarks
`setPattern`	This method declares a kind of placeholder useful for patterns (in like expressions.) In addition to the usual escaping mechanisms, content that contains the characters percent (`%`) and underscore (`_`) might need to be escaped. For example, if you want to match the string "30%" the percent should not be treated as a wild-card character. The `setPattern` methods are capable of handling this and, optionally, adding a wild-card character at the beginning and/or at the end of the escaped pattern. You may also choose to leave the pattern content as is, and you may optionally declare characters in the patterns that should be converted into the standard SQL wild-cards. Please refer to the JavaDoc documentation for more information.
`like`	This method is similar to the `setPattern` in that it is used to handle patterns for like expressions. Only, this method doesn't explicitly declare a value for a placeholder. Instead, it inserts a complete `like` expression in your SQL content. Please refer to the JavaDoc documentation for more information.
`placeholder`	This method inserts a reference to a placeholder with the specified name. This is particularly useful in cases where your SQL query isn't completely known at compile time.
`idList`	This method can be used to insert a number of ids (such as a list of field names) into your SQL content. You can specify an any sequence that should be used as separator between the elements, as well as any content written before and after. This method is found in a number of flavours. The most simple basically just takes a list of ids, and formats those as a comma-separated list. This may be useful for building a selection-field list where the exact fileds are not known at compile time.

Method	Remarks
`valList`	This method can be used to list a number of values. You can choose a character sequence to be used as separator and you can specify a sequence to be inserted before and after the listed elements. Each value will be formatted and escaped according the the value type. The method exists in various flavours, the most simple basically just takes a list of values, and formats those in a way that is particular useful in combination with `in`-expressions. Hence, if you have a liste of employee numbers in a variable, `employees`, you can easily generate SQL that matches any of these: `append("where EmployeeNumber in").valList(employees)`.
`val`	This method appends a value, formatted and escaped according to it's type. This is similar to referencing a placeholder and declaring it's value at once. Hence: `.append("where JobNumber =").val(jobNumber)` in context where `jobNumber` has the value `10250001`, this will give the SQL: `where JobNumber = '10250001'`.
`str`	This method is similar to the `val` method, except that the argument value is a String.
`amount`	This method is similar to the `val` method, except that the argument value is an amount.
`real`	This method is similar to the `val` method, except that the argument value is a real.
`integer`	This method is similar to the `val` method, except that the argument value is an integer.
`date`	This method is similar to the `val` method, except that the argument value is a Date. There are several flavours of this method that lets you declare the date in various ways.
`nullDate`	This method is similar to the `val` method, except that the value is the `nullDate`.
`time`	This method is similar to the `val` method, except that the argument value is a time value. There are several flavours of this method that lets you declare the time value in various ways.
`nullTime`	This method is similar to the `val` method, except that the value is the `nullTime`.
`bool`	This method is similar to the `val` method, except that the argument value is a boolean.
`popup`	This method is similar to the `val` method, except that the argument value is a popup value.

Method	Remarks
`nil`	This method is similar to the `val` method, except that the value is the `nil` value.
`setExpr`	This method sets the value of a placeholder of kind *expression*. When the build method is invoked, SQL corresponding to the expression will be inserted. Note that only simplistic expressions can be converted to SQL. You cannot use most of the expression functions. If you do, an exception will be thrown at run time. This method is particularly useful in case you wish to construct, e.g., a where-clause that corresponds to the constraints of a given record. Then this method can declare a placeholder value that will be substituted accordingly. For example, if you have an expression of the form: `CostPrice = 100.00 and CompanyNumber = '1'` then using this method, that expression is being translated into the corresponding SQL format.
`expr`	This method is like the `setExpr` method, except that it appends the SQL-variant of the expression at the specified position, i.e., without specifying a placeholder.
`build`	This method "builds" an object of type `McSql` from the content found in the builder. If it is not possible to generate SQL at this point, a run-time error will occur. Once you have the built `McSql` this will never change again, even if you make additional changes to the builder. The `McSql` object can be used to pass to the method `sql` on a `MiDatabaseApi` or you can directly invoke the SQL by using the `getResult` method.

6.2.3 Modifying Data in the Database

Occasionally you may need to manually update certain entries in the database. As stated before, when you add fields to a pane *you should not manually* ***`insert`****,* ***`update`*** *or* ***`delete`*** *the corresponding records in the database! The Extension Framework will manage this for you.* Still, suppose you want to maintain some aggregated table. For example, you might want to aggregate all time sheet registration for each project. Then every time a time sheet line is updated or changed, you should update a record in this table, possibly creating it if it doesn't already exist.

You can do this sort of thing by invoking the `insert`, `update` (and in some cases the `delete`) methods from the `MiDatabaseApi`. Like for the query methods (`select`,

`count` and `exists`), the modification methods are also available in a builder-style as well as the old "everything-in-one-go" style. For the old-style methods, you are referred to the Java-doc. Here, we shall only consider the builder-style methods. Generally, the `update` and `delete` methods may have a `where`-clause. The way to build the where-clause is identical to how it is done for queries (see above.)

Listing 6.18 shows an example where an aggregate table is updated every time a time sheet line is created, updated or deleted. For each job, an aggregated total-number-of-hours value is maintained. The interesting part—accessing the database—takes place in the method `updateJobBy` in lines 40–62. In line 49 we determine if the record should be created ("`inserted`") or updated in the database. This is done by checking whether a corresponding record already exists. The insert-operation takes place in line 50. In case the record already exists, we use the value already looked up, obtain the current value and add the amount to increase it. Then the record is updated in line 56. Notice that in both cases, nothing happens until the `execute` method is invoked!

Listing 6.18: Modifying Database Records.

```
2   @Override
3   public void onChangePost(final MiChangePost containerRunner,
4                            final MiUserChange eventData) throws
                                Exception {
5     final MiDataValues resultData = eventData.getResultData();
6     final MiUserData userData = eventData.getUserData();
7     final MiValueInspector originalData = eventData.
          getOriginalData();
8
9     final String jobNumberCurrent = resultData.getStr("JobNumber")
          ;
10    final String jobNumberOriginal = originalData.getStr("
          JobNumber");
11    BigDecimal updateCurrent = BigDecimal.ZERO;
12    BigDecimal updateOriginal = BigDecimal.ZERO;
13
14    if (userData.changed("JobNumber")) {
15      updateOriginal = originalData.getReal("WeekTotal").negate();
16      updateCurrent = resultData.getReal("WeekTotal");
17    } else {
18      updateCurrent = resultData.getReal("WeekTotal").subtract(
            originalData.getReal("WeekTotal"));
19    }
20
21    if (updateOriginal.compareTo(BigDecimal.ZERO) != 0) {
22      updateJobBy(jobNumberOriginal, updateOriginal);
23    }
24    if (updateCurrent.compareTo(BigDecimal.ZERO) != 0) {
25      updateJobBy(jobNumberCurrent, updateCurrent);
26    }
27  }
```



```
28
29    @Override
30    public void onDeletePost(final MiDeletePost containerRunner,
31                             final MiDelete eventData) throws
                                    Exception {
32      final MiValueInspector originalData = eventData.
           getOriginalData();
33      final BigDecimal weekTotal = originalData.getReal("WeekTotal")
           ;
34      if (weekTotal.compareTo(BigDecimal.ZERO) != 0) {
35        updateJobBy(originalData.getStr("JobNumber"),
36                    weekTotal.negate());
37      }
38    }
39
40    private void updateJobBy(final String jobNumber,
41                             final BigDecimal updateNumber) throws
                                    Exception {
42      if (!jobNumber.isEmpty() ) {
43        final MiKey aggregateTable = key("TRI_JobTSRegistrations");
44        final MiDatabaseApi databaseApi = getDatabaseApi();
45        final MiQueryInspector aggregateValues =
46          databaseApi.mselect("TotalRegs").from(aggregateTable)
47                     .where().eq("JobNumber").str(jobNumber)
48                     .getResult();
49        if (aggregateValues.isEmpty()) {
50          databaseApi.insert(aggregateTable)
51                     .setReal("TotalRegs", updateNumber)
52                     .setStr("JobNumber", jobNumber)
53                     .execute();
54        } else {
55          final BigDecimal currentTotalRegs = aggregateValues.
               getRecord(0).getReal("TotalRegs");
56          databaseApi.update(aggregateTable)
57                     .setReal("TotalRegs", currentTotalRegs.add(
                          updateNumber))
58                     .where().eq("JobNumber").str(jobNumber)
59                     .execute();
60        }
61      }
62    }
```

The above example showed how to do this by using the database API directly. There is another way of doing this. The differences between the two are subtle, and mainly a matter of taste: you may use a `MiPersistenceStrategy`. However, using a persistence strategy, you can potentially update sources not directly related. Using the exact same code, except how the persistence strategy is obtained of course. In this way, persistence strategies are slightly more flexible. If your persistence strategy is targeted

directly towards the Maconomy database, the difference is minor. In Listing 6.19 an implementation corresponding to the method `updateJobBy` is shown using persistence strategies. Since persistence strategies do not offer a builder-style pattern, it may be somewhat more cumbersome to use persistence strategies.

Listing 6.19: Modifying Database Records using a Persistence Strategy.

```
2   private void updateJobByPS(final String jobNumber,
3                              final BigDecimal updateNumber) throws
                                   Exception {
4     if (!jobNumber.isEmpty()) {
5       final MiPersistenceStrategy aggregateTable =
            McMolPersistenceStrategy.create(key("
            TRI_JobTSRegistrations"), getApiProvider());
6       final MiRecordValue jobNumberRestriction = dataValues().
            setStr("JobNumber", jobNumber);
7       if (aggregateTable.exists(jobNumberRestriction)) {
8         final MiList<MiKey> fieldList = McTypeSafe.createArrayList
              (key("TotalRegs"));
9         final MiQueryInspector currentContent = aggregateTable.
              select(fieldList, jobNumberRestriction);
10        final BigDecimal currentTotalRegs = currentContent.
              getRecord(0).getReal("TotalRegs");
11        aggregateTable.update(dataValues().setReal("TotalRegs",
              currentTotalRegs.add(updateNumber)),
              jobNumberRestriction);
12      } else {
13        aggregateTable.insert(dataValues().setReal("TotalRegs",
              updateNumber));
14      }
15    }
```

6.2.4 Database Access with Name-Spaced Fields

Since fields in panes can have name-spaces, and since such fields often have a direct link to a database field, the Maconomy database API has been made "transparent" to using of name-spaces on fields.

For example, suppose that you have a data-model where you have defined constants representing named fields. And that these constants contain name-spaces. Also suppose that you for some reason need to make a query for these fields directly in the database. It is highly convenient, if you can just ask for whatever names you have defined, even if the field names in the database does not contain the name spaces. Also, it is highly practical if the database result contains the name-spaced names rather than the internal database-names.

As an example, suppose we have a table defined in the database in the following way:


```
CREATE TABLE TRI_EMPLOYEE(
  EMPLOYEENUMBER                  VARCHAR(255) NOT NULL,
  PREFERREDCUST                   VARCHAR(255) NOT NULL
);
```

Also, suppose that you have added the field `PreferredCust` to a pane in an extension data-model. And suppose that you have specified the name-space `Trifolium` in that data-model. Then, the field that is added to the pane will be called `Trifolium:PreferredCust`. Suppose that whenever we do an update, we want to check whether some other employee has the same preferred customer. We can do so by looking up data from the table `TRI_EMPLOYEE`. Now, suppose you have the following piece of code:

```
1  private static final MiKey PREF_CUST =
2    key("Trifolium:PreferredCust");
3  private static final MiKey TABLE = key("TRI_EMPLOYEE");
4
5    public void onChangePre(final MiChangePre containerRunner,
6                            final MiUserChange eventData) throws
                                Exception {
7      /* ... some code goes here ... */
8      MiQueryInspector result =
9        getDatabaseApi().mselect(PREF_CUST).from(TABLE)
10         .where().eq("EmployeeNumber").str(emplNo)
11         .getResult();
12
13     if (!result.isEmpty()) {
14       String prefCust = result.getRow(0).getStr(PREF_CUST);
15       /* handle preferred customer... */
16     }
17     /* ... more code goes here */
18   }
```

This will indeed work as expected: the field `PreferredCust` will be selected from the underlying database table (i.e., the name-space is ignored). But the result *will* contain the name-spaced field names, which is why it works to look up the value of that field using the name `Trifolium:PreferredCust`. The API can handle only queries with one name-space for a given record. If two different name-spaces are explicitly used in the same record, a run-time error will occur.

6.2.5 Obtaining Popup Values from the Maconomy Database

In the Maconomy type system, there is a concept of "popup types." A popup type is really just an enum. Each entry in the enum has an associated *literal value* which is case insensitive, an *ordinal value* which is an integer used as a "key" to distinguish this value from others, a *title value* which is a text eventually shown to the end-user and which

may be localized to various languages and finally a *popup type*. For a given popup type, all values have different ordinal and literal values.

Sometimes you may wish to set the value of some field to a certain popup value. In this case, you need to know the ordinal value and the literal value. A way to obtain the correct values, you can choose to access the database API. The method `getPopupValues` returns a list of `McPopupDataValue`s that currently exist for a specific popup type. This may be especially relevant for dynamic popup types, i.e., popup types where the entries are user-defined. Notice that using this method, you can only obtain information about popup types that are defined in the core Maconomy application—custom built popup types cannot be obtained in this way.

Listing 6.20 shows an example where this mechanism is used to fetch all popups of type `CountryType`. A list of defined values of this type is obtained in line 2. Then we iterate over the elements in this list until we find one with the literal name "France."

Listing 6.20: Obtaining Maconomy Popup Values.

```
2   final MiList<McPopupDataValue> countries = getDatabaseApi().
        getPopupValues(key("CountryType"));
3
4   // Find the popup value for "France"
5   MiOpt<McPopupDataValue> france = McOpt.none();
6   for (final McPopupDataValue country : countries) {
7     if (country.getLiteralValue().isLike("France")) {
8       france = opt(country);
9       break;
10    }
11  }
12  if (france.isDefined()) {
13    // a match for France was found...
14  } else {
15    // France was not found...
16  }
```

6.2.6 Using a Caching Buffer

As briefly explained in Section 4.12.2 the Extension Framework comes with a handy utility class called `McCachedDataHost`. This class is frequently used when looking up values presented in variables, but it may be generally used to fetch data.

The purpose of this class is to offer a short-lived caching buffer between the code and the database. It is often the case that:

- Several fields from the same database record is needed in different places in the code.
- The same record may be needed several times, especially in loops.

- You need to be prepared for a situation where the record you may need does not exist, in which case you want to use some default value instead.

The `McCachedDataHost` addresses these issues. In fact the `McCachedData`*`Host`* is just a common place to host a number of "caching buffers." The class offers the following methods:

Method	Remarks
`installCache`	This method "installs" a caching buffer in the host. The caching buffer is identified by a table name. This table name is the one from where data is obtained. Further, you must specify the name of the key field of this table. The current implementation supports only *one* key field. Therefore, this mechanism can be used only for tables having exactly one key field[4]. The next argument you must specify is a optional key-value that can be ignored. If no key values should be ignored, you must provide a `McOpt.none` value. For example, it is known that no Job exists with `JobNumber = ""` (e.g., blank.) By specifying this value as an ignored key value means that if you try to access information for this key value, the system *knows* that it doesn't have to contact the database: in such cases, a default value must be returned. Following this, you can specify any number of `String` or `MiKey` arguments denoting fields from this table that you might be interested in. A slightly different flavor of this method exists: it also requires a `MiPersistenceStrategy`. If install a cache in this way, data will be looked up through the persistence strategy, rather than directly from the Maconomy database. Generally, you will declare your `McCachedDataHost` as a member variable of your data-model class, and install the desired caches in the constructor.
`getCache`	This method returns a caching buffer that must have been "installed" using the `installCache` method above.

When you wish to make use of one of the installed caching buffers, you should obtain it using the `getCache` method. This returns an object of type `McCachedData`. This type offers a number of useful methods:

[4]Examples include: `JobHeader`, `EmployeeInformation` and `CompanyInformation`. Examples of tables that cannot be used with this utility include `TimeSheetHeader` and `CompanyCustomer`.

Method	Remarks
`exists`	This method takes a `McDataValue` and checks if a record exists with this key value. If the key value you're testing is identical to the one to be "ignored" when the cache was installed, this method returns `true`!
`existsUnconditionally`	This method is similar to `exists` above, except that it does not automatically treat an ignored value as existing.
`getValue`	This method can be used to obtain a value corresponding to one of the fields specified when the cache was installed. In addition to a field name, you provide a key value and a default value. If no record is found having the specified key, the default value is returned. This obviously will also happen, if the key value matches the ignored key value specified when the cache was installed.
`populate`	This method will populate the cache with a set of key values. Generally doing this will be more performance effective compared to populate or look up one key value at a time. Especially in the `refreshVariablesPrepare` data-model method, you should strongly consider using this mechanism if possible.
`clear`	This method clears the cache. Usually the cache should only be used "short-termed." If it is for some reason needed to clear the cache, you can do it using this method. If you don't two consecutive look ups for a field based on the same key value will always return the same answer, even if the database has been updated in the meantime.

Section 4.12.2 contains examples of using the `McCachedDataHost` class.

6.3 Creating Asynchronous Background Tasks

In Maconomy version 19 (2.3) the concept of *background tasks* was introduced. A background task represents some container event that will be carried out in a specified container and a specified record. Hence, a background task is in many ways similar to some event made with a container executor, only it is scheduled to run *asynchronously* and possibly at a later date and time.

A background task is represented as one or more records in the database. Creating a background task programmatically therefore leads to a number of records being created in the database. An important property of this is, that if an event that creates background

tasks fails (e.g., showing some error message to the user,) the existence of the created background tasks will be rolled back, since they are made in the same transaction as any other synchronously data modifications carried out by the current event handling. It also means that background tasks created during event handling will not be executed until the current event has succeeded.

So why would we want to schedule some operation to be carried out asynchronously? There may be several reasons for that, including:

- The scheduled background task needs to communicate with an external system—but only if the current event succeeds.
- A number of events need to be carried out, each in their own transaction.
- Performance reasons: the current event leads to a number of operations, but the user doesn't need to await the completion of all of these.
- The thing to be carried out should be done at a later data and time, not now.

Background tasks are created by accessing the `batch` method on a `containerRunner`. The result of this method is an object of type `MiScheduler`. There are two types of methods on this interface:

Method	Remarks
`task`	This method initiates creation of a background (or batch) task. Such a task represents an event in a pane in a container as well as an indication of the record[5] that needs to be addressed. This method is found in a couple of flavours: • A version with no parameters. This means that the background task relates to the same container and pane as the current container event. • A version tasking the name of the container as an argument.
`group`	This will initiate a *group* of background tasks. Actual tasks can then be added to this group. See page 268 for more details on task groups.

Listing 6.21 shows an example where a simple background task is created: Here the `onCreatePost` is implemented for a contribution extending the `maconomy:Jobs` container. The extension makes sure that when a job is created, that job is being "sent to People-Planner". This functionality is available through the action called `SendToPP`. However, this action doesn't work unless the job number is *externally visible*. As long as the `Create` event is still being handled, the newly created job is *not* externally visible. This problem

[5] In principle a background task can relate to several records, in which case the specified event will be carried out for each of these, all of them in one transaction.

is easily handled by making use of background tasks: the extension logic simply creates a background task that:

- Concerns the current container and pane, i.e., `maconomy:Jobs` container and `CARD` pane. This happens in line 4.
- Will execute the action `SendJobToPP`, represented by the constant `SEND_JOB_TO_PP`. This happens in line 5.
- Does so for the job just created, by invoking the `restrictBy` method in line 6.
- Annotates the background task with an informational description (line 7.) This has no semantic effect—it is just there to provide a better overview to whoever monitors the background tasks.

Finally, the `end` method in line 8 signals to the Extension Framework that there are no further attributes for this background task, and the background task will be created. By default such as task will be considered due "now."

Listing 6.21: Creating a Simple Background Task.

```
1  public void onCreatePost(final MiCreatePost containerRunner,
2                           final MiCreate eventData) throws
                                Exception {
3    final MiDataValues resultData = eventData.getResultData();
4    containerRunner.batch().task()
5      .action(SEND_JOB_TO_PP)
6      .restrictBy(resultData.asKeyValuesCopy(JOB_NUMBER))
7      .description("New Job to PeoplePlanner")
8      .end();
9  }
```

6.3.1 Adding Attributes to a Background Task

Background tasks are made using a "builder" as shown in Listing 6.21. The builder is started by the `task` method. From there, various attributes of the background task can be added. Some of which are mandatory, others are optional. Some are conditionally mandatory, and others only make sense if other attributes have certain values.

In order to handle this in a smooth way, the background task API makes heavily use of the compilers type system to offer a path through the various attributes, ensuring that compilation succeeds only if the background task has the required attributes in a sensible combination. This may sound more complex that it is. In fact, by making use of the built-in "content assistance" that many IDEs offer (including the Maconomy Extender,) the programmer is neatly guided through the various possibilities.

Figure 6.1 shows how the background task API is used to build a background task. A task begins with the `task` method. If a container name is specified, the pane can optionally

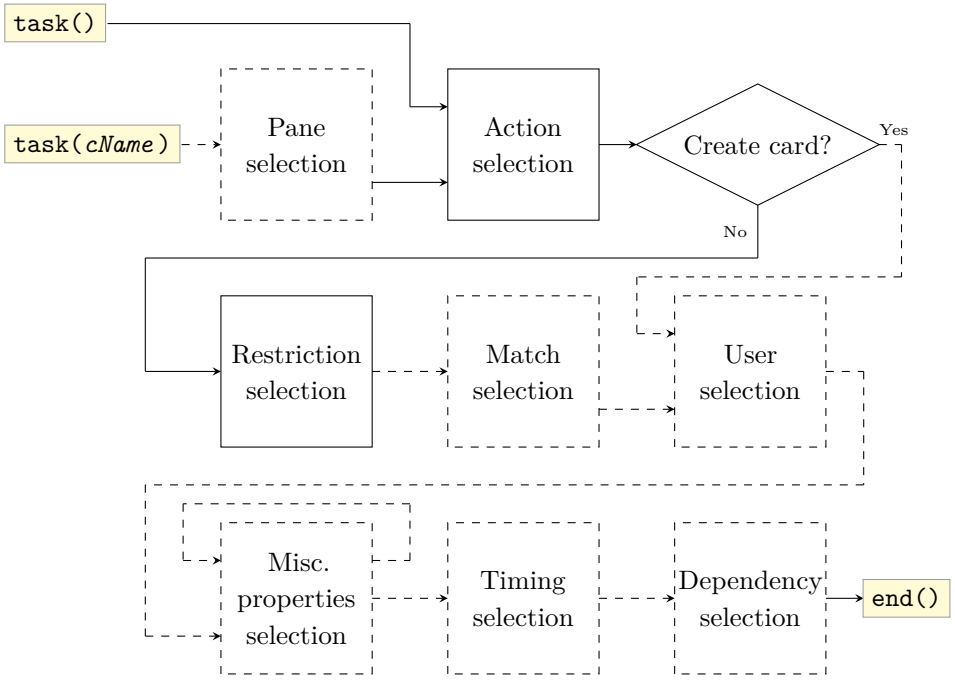


Figure 6.1: Flow of Background Task API Property Selection. The dashed arrows and boxes indicate optional sections. The solid arrows and boxes indicate mandatory sections.

be specified (the default is `Card`). If the no-argument version is used, the container and pane is considered the same as for the current event, in which case it makes no sense to specify the pane. Next, the action must be specified. If the pane is known to be a card pane, and the action is the `Create` action, it makes no sense to specify the container restriction and the matching properties of the action; otherwise it is mandatory to specify the container restriction, and optional to provide matching properties. From here, a number of properties of various categories can be optionally specified. The invocation of the `end` method indicates that the task is done and it will be constructed by the Extension Framework. Let us examine each of the property groups in more detail.

Pane Selection Properties

The pane selection properties are used to specify the pane in which the action of the background task is found. This property group is optional. If the pane is not specified, the `Card` pane will be assumed, since this is the most common use case. The available methods in this group are:

Method	Remarks
`card`	This method is used to indicate that the background task applies to the `Card` pane. A number of variants of this method exits, letting you specify the textual name of the card pane, in case it doesn't have the default name for panes of type `Card`. Usually the default name is used.
`table`	This method is similar to the `card` method above, except that it is used to indicate that the task applies to the pane of type `Table`.
`filter`	This method is similar to the `card` method above, except that it is used to indicate that the task applies to the pane of type `Filter`.
`pane`	This method takes the type or name of the pane as an argument. It is used to indicate that the background task applies to the pane specified as the argument, making it useful for highly generic use cases. In addition, a variant of this method takes a parameter set that will be applied as parameters for when reading the container when the background task is eventually executed. As this is considered a rare use case, this possibility is not supported the above methods, and you will have to use this method instead.

Action Selection Properties

The action selection properties are used to specify which action/operation should be performed by the background task in the specified container and pane. The available methods in this group are:

Method	Remarks
`create`	Invoking this method means that the background task will execute a `Create` operation in the specified container and pane. For table panes, the record will be added (i.e., appended to the end of the table) except if a `matchBy` condition is specified: in that case, a new line will be inserted above the first line matching that condition (if no line matches the condition, no lines will be created.) A number of flavours of this method exist, allowing specification of which user-input values and parameters should be applied upon creation. Notice that for background tasks, user-input values may be supplied as expressions that will be evaluated at run-time. In this way, for example, it is possible to specify a date that is relative to the date when the background task is executed. This is done by providing an argument of type `MiExprDataValues`. This interface is explained below.
`update`	This method indicates that the background task should perform an `Update` operation in the specified container and pane. For table panes, if a `matchBy` condition is specified, all lines matching that condition will be updated in this way. If the pane is a card pane, an update only occurs if the data in the card matches this condition. If no `matchBy` condition is specified, all records in the pane will be updated.
`delete`	This method indicates that the background task should perform a `Delete` operation in the specified container and pane. For table panes, if a `matchBy` method exits, only lines matching this condition will be deleted, and for card panes, the record will be deleted only if the data fulfills the `matchBy` criterion. If no such criterion is provided, all records in the pane will be deleted.
`print`	This method indicates that the background task should perform a `Print` operation in the specified container and pane. For table panes, if a `matchBy` method exists, the `Print` operation will be carried out for each matching line. For card panes, the print operation will only be carried out if the data in the pane fulfills the criterion. If no `matchBy` criterion is specified, the operation will be performed on all records in the pane. This method is found in a number of flavours, including one that accepts user-input as well as event parameter. If user-input is provided, it means that the background task will execute an `Update` event prior to the `Print` event.

Method	Remarks
`action`	This method indicates that the background task should perform a `Action` operation in the specified container and pane for the action with a specified name. For table panes, if a `matchBy` method exists, the `Action` operation will be carried out for each matching line. For card panes, the print operation will only be carried out if the data in the pane fulfills the criterion. If no `matchBy` criterion is specified, the operation will be performed on all records in the pane. This method is found in a number of flavours, including one that accepts user-input as well as event parameter. If user-input is provided, it means that the background task will execute an `Update` event prior to the `Action` event.

All of the above methods support user-input of type `MiExprDataValues`. Instances of this interface are constructed by invoking the `create` method on the `McExprDataValues` class. The methods of this class are like those for `MiRecordDataValues`, with the addition of the following methods:

Method	Remarks
`setAmountExpr`	This method is used to assign an expression of type Amount to a field name.
`setBoolExpr`	This method is used to assign an expression of type Boolean to a field name.
`setDateExpr`	This method is used to assign an expression of type Date to a field name.
`setIntExpr`	This method is used to assign an expression of type Integer to a field name.
`setPopupExpr`	This method is used to assign an expression of type Popup to a field name.
`setRealExpr`	This method is used to assign an expression of type Real to a field name.
`setStrExpr`	This method is used to assign an expression of type String to a field name.
`setTimeExpr`	This method is used to assign an expression of type Time to a field name.
`setExpr`	This method is used to assign an expression to a field name.
`getExpr`	Returns the specified field value as an expression. If the specified field does not exist, a run-time error occurs.

Method	Remarks
`getExprOpt`	Returns the specified field value as an optional expression. If the field does not exist, a `none` value is returned.
`getExprInfo`	Returns an object that is used to query information related to the expression of a field with a specified name. This value allows the caller to obtain the expression as a String (preferably preserving the original String-based expression that might have been assigned to a field.) In addition, the concrete type of the expression (if known) can be obtained.
`getAmount`	In addition to the usual variant, an additional variant of this method exists, where an evaluation context is provided. In case the field value is stored as an expression, the return value is obtained by evaluating the expression in this context.
`getBool`	This method is similar to the `getAmount` method above.
`getDate`	This method is similar to the `getAmount` method above.
`getInt`	This method is similar to the `getAmount` method above.
`getPopup`	This method is similar to the `getAmount` method above.
`getReal`	This method is similar to the `getAmount` method above.
`getStr`	This method is similar to the `getAmount` method above.
`getTime`	This method is similar to the `getAmount` method above.
`simpleValueFieldNames`	This method returns a collection of the field names that are represented by a simple value (i.e., not an expression that needs to be evaluated.)
`complexValueFieldNames`	This method returns a collection of the field names that are not simple, i.e., those that are assigned a value which is an expression that needs to be evaluated.
`isSimpleValueField`	Returns whether a given field name is a simple value field.
`isComplexValueField`	Returns whether a given field name is a complex value field, i.e., an expression that needs to be evaluated.

Listing 6.22 shows an example that makes a background task for creating a new Sales Order, as specified in line 8. Since no pane is explicitly stated, the `Card` pane is assumed. The user input is specified as an argument to the `create` action which is used to indicate that the background task should perform a create operation. This happens in line 9. Notice that two of the fields provided as user input are not open for editing during creation. The background task execution engine will automatically detect this and will perform an additional `Update` operation ensuring that the missing field values are obtained. Hence, you do not have to create two background tasks for this!

Listing 6.22: Setting User Input Fields of Background Task for Createion.

```
1  final MiRecordValue newSalesOrderData = McRecordValue.create()
2    .setStr("CustomerNumber", "33117722")
3    //NB! The following fields are closed in create
4    //     but open during update
5    .setStr("Remark", "Created by Background Task")
6    .setStr("Receiver", "Automatic Order System");
7  containerRunner.batch()
8    .task("SalesOrders")
9    .create(newSalesOrderData)
10   .description("Sales Order from Order System")
11   .end();
```

Listing 6.23 demonstrates how user input can make use of expressions. In lines 4–10 we assign values to the fields `NumberDay1`–`NumberDay7` of a time sheet line. The value for `NumberDay1` is declared as: $\max(0, -$`card.OvertimeNumberDay1Var`$)$. Using the `setRealExpr` method, we can declare that the value of the field is of type Real, but provided as an expression that will be evaluated at execution time. In this example, we are creating a line in the table pane of the `maconomy:TimeSheets` container, and therefore we can make references to values in the existing card pane, using the '`card.`' prefix. By using the max function, we express that the number of hours put in will never be negative. Line 12 is where we declare that the background task is targeted as the table pane in the `maconomy:TimeSheets` container. In line 13, it is specified that the background task should perform a create action (the new line will be appended to the end of the time sheet lines because no `matchBy`-constraint is given). Line 14 ties he operation to a specific time sheet. We shall investigate this further on page 252.

Listing 6.23: Setting User Input Fields of Background Task for Createion.

```
1  final MiExprDataValues missingTimeData = McExprDataValues.create()
2    .setStr("JobNumber", "STD1000200")
3    .setStr("TaskName", "MISS")
4    .setRealExpr("NumberDay1", "max(0, -card.OvertimeNumberDay1Var)"
       )
5    .setRealExpr("NumberDay2", "max(0, -card.OvertimeNumberDay2Var)"
       )
6    .setRealExpr("NumberDay3", "max(0, -card.OvertimeNumberDay3Var)"
       )
```



```
7     .setRealExpr("NumberDay4", "max(0, -card.OvertimeNumberDay4Var)"
          )
8     .setRealExpr("NumberDay5", "max(0, -card.OvertimeNumberDay5Var)"
          )
9     .setRealExpr("NumberDay6", "max(0, -card.OvertimeNumberDay6Var)"
          )
10    .setRealExpr("NumberDay7", "max(0, -card.OvertimeNumberDay7Var)"
          );
11  containerRunner.batch()
12    .task("TimeSheets").table()
13    .create(missingTimeData)
14    .restrictBy(timeSheetKey)
15    .description("Add missing time to time sheet")
16    .end();
```

For background tasks, you can specify that an `Update` event needs to take place prior to the actual event. This may be useful to ensure that certain fields (or variables) are properly assigned. Corresponding to what happens when a wizard is sometimes shown in the Workspace Client when an action is invoked. At other times, it is necessary to fill in certain selection criteria prior to running an action. Listing 6.24 shows an example of this. The background task is tied to the card pane of the `JobBudgets` container, and the action we want to be run is the `RemoveZeroLines`, as specified in line 6 which is the first argument to the `action` method. In this case, we invoke a variant of that method that takes three arguments: first the name of the action, second the user input and third the event parameters. The user input values are provided in line 7. Line 8 merely passes an empty parameter set to the event. Because the background task is instantiated with user values, the background-task execution engine will perform the corresponding update (in this case ensuring that the value of the field `ShowBudgetTypeVar` is set to the 'Working Budget' type.)

Listing 6.24: Setting User Input Fields of Background Task for Createion.

```
1   final MiRecordValue switchToWorkingBudget = McRecordValue.create()
2     .setPopup("ShowBudgetTypeVar", WORKING_BUDGET);
3
4   containerRunner.batch()
5     .task("JobBudgets")
6     .action("RemoveZeroLines",
7             switchToWorkingBudget,
8             McParameters.create())
9     .restrictBy(jobKey)
10    .description("Remove zero lines of Working Budget")
11    .end();
```

Restriction Selection Properties

The restriction selection properties are used to specify the container key of the container being operated on. The available methods in this group are:

Method	Remarks
`restrictBy`	This method is used to specify the container key values of the container being addressed. A number of flavours of this method exists, letting you provide up to four key fields and corresponding values. It should be noted that for background tasks, it is not a requirement that the formal key fields are used, nor that exactly one container key is addressed. If the formal key fields are not fully specified, the background-task execution engine will automatically perform a search for all keys where the provided values are as specified, and the described operation will take place for *each* of these, but in *one common transaction*. This means that either all of the invocations succeed, or they all fail. It is recommended that you consider creating a number of specific individual background tasks (that can be carried out in parallel, and which will run in their own transaction) rather than specifying a background task that will loop over a number of records.
`singleton`	This method needs to be used for singleton containers (i.e., containers that cannot be addressed using a specific container key.) If you attempt to invoke this action for a container that is *not* a singleton container, a run-time error will occur! Examples of singleton containers are: all of the `Import`-containers, all of the `Print`-containers, `maconomy:CentralTimeSheetTransfer`, `maconomy:ChangePaymentSelection` and `maconomy:TimeRegistration`.
`forAll`	This method can be used to create a background task that represents an operation that should be carried out for *all* available container keys. Even though the actual representation of such a task is the same as the representation used by the `singleton` method, the programmatic API has a dedicated method for this. The reason is that addressing *all* possible container keys is something out of the ordinary, and should only be done with great care! For this reason, if the container is a singleton container, a run-time error will occur, forcing you to use the `singleton` method instead.

Let us examine these method in more detail. Listing 6.25 shows an example where a specific job is to be converted to order. In line 4 the container key (in this case the job number) is specified.

Listing 6.25: Restricting a Background Task to a Specific Key.

```
1 containerRunner.batch()
2   .task("Jobs")
3   .action("ConvertToOrder")
4   .restrictBy("JobNumber", McStr.val("10250001"))
5   .description("Convert Job to Order")
6   .end();
```

Listing 6.26 shows a similar example, but this time, the restriction is not based on the formal container key fields. Instead, it indicates that only any job where the status is 'Quote' and which are not closed, should be converted to order. Since this is all specified by a single background task, it is handled in *one* transaction. Which means that either all of the indirectly referenced jobs succeed, or they all fail (the entire transaction will be rolled back.) The central part of this example is in lines 4–5. Here the `restrictBy` method is applied for fields that are *not* the formal key fields. Thereby potentially having a number of jobs processed by the same background task.

Listing 6.26: Restricting a Background Task to a Set of Keys.

```
1 containerRunner.batch()
2 .task("Jobs")
3 .action("ConvertToOrder")
4 .restrictBy("Closed", McBool.FALSE,
5             "Status", QUOTE)
6 .description("Convert Open Quote Jobs to Order")
7 .end();
```

Letting a single background task handle several entities (like jobs) in one transaction may be convenient, even desirable. There are, however, often many reasons why you should rather create a number of specific background tasks, each targeting a single job, instead:

- If just one of the matching jobs cannot be converted, no jobs will be converted, since the entire transaction is rolled back. If you instead have many individual background tasks, each individual task will either succeed or fail. If just one out of many fails, then the others will still be converted.
- Database locks may be held for a longer period of time, potentially slowing the overall system performance.
- Database roll back space grows, potentially slowing the overall performance.

- By having a number of individual tasks, the background-task execution engine may run several of these concurrently, increasing the processing wall-clock time. If many jobs are covered by one task, each job will be handled sequentially.

Listing 6.27 shows how to generate a number of individual background tasks. The behavior will be equivalent to that of Listing 6.26, but each job is converted independently of the others, and the processing of the background tasks may be handled concurrently. A container executor is obtained in order to extract the relevant job numbers. The code then iterates over all of the matching jobs, creating a background task for each of them. Each background task is targeted at that specific job number (line 9.)

Listing 6.27: Generating Individual Background Tasks in a Loop.

```
1  final MiContainerExecutor jobsFilter = containerRunner.executor("
       Jobs").construct(MePaneType.FILTER);
2  jobsFilter.control().select("JobNumber")
3                      .restrictBy(and(not("Closed"),
4                                      eq("Status").popup(QUOTE)));
5  for (final MiValueInspector job : jobsFilter.read().matchAll()) {
6    containerRunner.batch()
7    .task("Jobs")
8    .action("ConvertToOrder")
9    .restrictBy(job.asKeyValuesCopy("JobNumber"))
10   .description("Convert Open Quote Job to Order")
11   .end();
12 }
```

The `forAll` method may be used for cases where a background task must iterate over *all* keys in a container. You should be *very careful* about using this, since it is far from clear exactly which keys will be targeted: *any* will be targeted, with no exception! Considering that you know, this is what you want, Listing 6.28 shows how to do this.

Listing 6.28: Generating Individual Background Tasks for All Keys.

```
containerRunner.batch()
  .task("Jobs")
  .action("ConvertToOrder")
  .forAll() //! Beware - ALL keys will be targeted
  .description("Convert ALL Jobs to Order")
  .end();
```

The `singleton` method must be used to address singleton containers. A singleton container is a container where only one instance exists (at least seen from a given user/role) and which cannot be addressed by a key value. Listing 6.29 shows an example where the `SubmitTimeSheet` action is invoked on the singleton container `maconomy:TimeRegistration`. Line 9 indicates that the container is a singleton. Notice that a field is updated prior to the action, due to the additional argument in line 7. In this case to address the desired time sheet date.

Listing 6.29: Addressing Singleton Containers.

```
1   final MiValueInspector timeSheetDate =
2     McRecordValue.create().setDate("TheDateVar",
3                                    McDate.val(2016, 6, 6));
4   containerRunner.batch()
5     .task("TimeRegistration")
6     .action("SubmitTimeSheet",
7             timeSheetDate,
8             McParameters.create())
9     .singleton()
10    .description("Submit Time Sheet")
11    .end();
```

Match Selection Properties

The match selection properties are used to specify preconditions or record-level criteria. If a `matchBy`-condition is provided, the background-task execution engine will ignore any records that do *not* fulfil the criterion. For card panes, it means that either the batch execution will perform the specified operation, or it will skip the operation for the current container key. For table panes, the specified operation will be performed on *any* line where the condition is true. Except for the `Create` event: here a new record will be *inserted above* the *first* line where the condition is true. If no such line exist, no line will be created.

In this part, the following method exist:

Method	Remarks
`matchBy`	This method exists in a number of flavours: • One variant takes an expression object. If the expression—when evaluated in the context of the record in question—is `true`, the operation will be carried out for that record, otherwise not. • One variant takes a `String` argument as a syntactic representation of an expression. • Methods similar to the above, but with an additional argument, let you specify whether records for which the specified operation is disabled should be considered as “not matching” (which is the default.)

As an example where this might be useful, consider a case where a background task needs to convert a job to status ‘Order.’ If the job already has status ‘Order,’ the `ConvertToOrder` action is disabled. Attempting to execute the action anyway, will lead

to an error, and the background task would render as having "failed." By default, actions that are rendered as disabled will be treated as if the `matchBy`-condition has failed. But actions are not always disabled, even if it is not possible to run them. Sometimes this is done in order to give explanatory messages to an end user. However, the background-task execution engine is not a real end user, and will not interpret error messages and do other operations. In order to keep the number of tasks reported as having "failed" to a minimum, the `matchBy` method may come in handy. Listing 6.30 shows an example of this. Here we create a background task that will attempt to `Post` a general journal. However, this action will fail unless the `Balance` has a value of 0. The `Post` action is, however, still enabled in this case. As an additional guard, this background task will only attempt to post the journal if the field `ToBePosted` is set to `true`. This condition is specified in line 9.

Listing 6.30: Guarding Card Action using Match-By.

```
1 containerRunner.batch()
2   .task("GeneralJournal")
3   .action("Post")
4   .restrictBy("JournalNumber", McInt.val(journalNumber))
5   .matchBy("ToBePosted and Balance = 0")
6   .description("Post G/L Journal")
7   .end();
```

The matchBy can also be used to pinpoint one or more specific lines in a table pane. For example in Listing 6.31, we want to create a background task that creates a favorite from a specific time sheet line. Now, the `restrictBy` method is used to point out the *time sheet* in question; not the specific *line* in the time sheet. For this purpose, we can make use of the `matchBy` method, as done in line 6. This criterion will match the particular time sheet line, and only that. No matter if it is moved to another position. Obviously the time sheet line can be deleted, in which case no lines match.

Listing 6.31: Addressing a Specific Line in a Table Pane.

```
1 containerRunner.batch()
2   .task("TimeSheets")
3   .table()
4   .action("CreateJobFavorite")
5   .restrictBy(timeSheetKey)
6   .matchBy(currentTimeSheetLine.copyValues("InstanceKey").
      asExpression())
7   .description("Create Job Favorite from Time Sheet Line")
8   .end();
```

Obviously, the `matchBy`-mechanism can also be used to point out several lines in a table pane. Listing 6.32 shows a silly example that deletes all even lines of a given time sheet, simply be creating a matching expression in line 6.

Listing 6.32: Matching Several Lines in a Table Pane.

```
1 containerRunner.batch()
```



```
2     .task("TimeSheets")
3     .table()
4     .delete()
5     .restrictBy(timeSheetKey)
6     .matchBy("LineNumber % 2 = 0")
7     .description("Delete Even Time Sheet Lines")
8     .end();
```

User Selection Properties

The user selection properties are used to indicate that the background task should be run as a specific user or user role. *This is only a hint, and may be ignored if the current user does not have sufficient privileges!*

A non-administrator user is *only* allowed to create background tasks that run as that particular user. If a different user-name is attempted, it will be ignored. In this way, there is no risk that a non-administrator user can get access to actions or information that he or she does not already have access to. An administrator user is allowed to create background tasks that impersonate some other user. 2 For users having multiple roles[6], it is possible to create a background task that runs as a different role of the current user. This is possible for all users, also non-administrator users. Notice that a given user does *still* not get access to anything he/she would not be able to access manually.

If nothing is specified, the current user-role will be assumed for non-administrator users, for administrator users, the background-task user will be assumed.

In this part, the following method exist:

Method	Remarks
`runAsRoleInstance`	This method takes as argument an instance key of the desired user-role. If the current user is not an administrator user, the appointed role must be one of this users' roles.
`runAsUser`	This method takes as argument the desired user name. Notice that this will be ignored if the current user is not an administrator user. If the current user *is* an administrator user, and the specified user has several user-roles, the user-role marked as "Default for Background Tasks" will be assigned.

[6]The concept of multiple user roles was introduced in Maconomy 19 (2.3)

Method	Remarks
`runAsEmployee`	This method takes as argument an employee number. If this employee is associated with a user, that user will be used (although, as explained above, the user can only be specified if the current user is an administrator user.) In case several users are associated with this employee number, an arbitrary one of these will be chosen, although—if possible—a non-administrator user will be preferred. Once a user-name is in this way settled, the user-role marked as "Default for Background Tasks" will be assigned.

To demonstrate how this works, consider Listing 6.33. The purpose of this snippet is to generate a number of background tasks: one for each user-role belonging to a given company. If the user has several roles for the same company, only one of these are used to create a background task. Each created background task will instantiate the `Transfer` action in the `maconomy:CentralTimeSheetTransfer` container, setting the range of companies to be that of the user-role in question. All this is done by first making a database query that returns the user-role instance keys of the current user (lines 2–6.) Then we loop over each of these in line 8. Lines 19–21 declares that the `Transfer` action is to be run by the background task, and the the company range should equal that of the user-role being considered. Finally, in line 23, it is specified that the background task should be run as the user-role of the current iteration.

Listing 6.33: Run Background Task on Behalf of a User Role.

```
1  final String currentUserName = containerRunner.getEnvironmentInfo
       ().getUserName();
2  final MiQueryInspector currentUserRoles =
3    containerRunner.getDatabaseApi()
4      .mselect(COMPANY_NUMBER, INSTANCE_KEY).from("
         UserRoleInformation")
5      .where().eq("NameOfUser").str(currentUserName)
6      .getResult();
7  final MiSet<McStringDataValue> processedCompanies = McTypeSafe.
       hashSet();
8  for (final MiValueInspector currentUserRole : currentUserRoles) {
9    final McStringDataValue userRoleCompany = currentUserRole.
       getStrVal(COMPANY_NUMBER);
10   if (!processedCompanies.containsTS(userRoleCompany)) {
11     processedCompanies.add(userRoleCompany);
12     final McStringDataValue roleInstanceKey = currentUserRole.
         getStrVal(INSTANCE_KEY);
13     final MiValueInspector companySelection = McRecordValue.create
         ()
14       .setStr("FromCompanyNumber", userRoleCompany)
```



```
15        .setStr("ToCompanyNumber", userRoleCompany);
16
17      containerRunner.batch()
18        .task("CentralTimeSheetTransfer")
19        .action("Transfer",
20                companySelection,
21                McParameters.create())
22        .singleton()
23        .runAsRoleInstance(roleInstanceKey)
24        .description("Central Time Sheet Transfer, Company " +
             userRoleCompany)
25        .end();
26    }
27  }
```

Listing 6.34 shows another user case. In this snippet, we (somehow—it is not important for the example) obtain a list of jobs that should be converted to order. For each such job, a background task is created specifying the action `ConvertToOrder`. In line 14 it is specified that the action should be run on behalf of whatever user is associated with the employee stated in the field `ProjectManagerNumber`. In case no such employee is specified, the task will be run as whatever user has been configured as the background task administrator.

Listing 6.34: Run Background Task on Behalf of an Employee.

```
1  final String currentUserName = containerRunner.getEnvironmentInfo
       ().getUserName();
2  final boolean isAdminUser = containerRunner.getDatabaseApi()
3    .mexists("UserInformation")
4    .where("IsAdministrator")
5    .and().eq("NameOfUser").str(currentUserName)
6    .getResult();
7  containerRunner.check(isAdminUser)
8                  .error("Only an administrator may do this");
9  for (final MiValueInspector job : convertToOrderJobs(
       containerRunner)) {
10   containerRunner.batch()
11     .task("Jobs")
12     .action("ConvertToOrder")
13     .restrictBy(job.asKeyValuesCopy("JobNumber"))
14     .runAsEmployee(job.getStr("ProjectManagerNumber"))
15     .description("Convert Job to Order")
16     .end();
17 }
```

Miscellaneous Selection Properties

The miscellaneous selection properties are used for various "ad hoc" properties. Unlike the other selection groups, you can declare several of these properties for the same background task.

In this part, the following method exist:

Method	Remarks
`executionRequired`	This method sets whether it should be considered an error, if the background task does not *really* execute the specified action on any record. For example, consider an background task that is supposed to post a journal, but will only do so if the balance is 0. This can be expressed using the `matchBy` method. In this case, the operation may be skipped in case the balance of the journal is not 0. By default this will mark the background task as having "succeeded" (it has done all it was asked to do.) If you need to consider the case where *no* journal was posted as an error, you can specify that execution is required. In that case, the background task will be marked as having "failed" if the balance is 0.
`autoDelete`	This method can be used to mark whether a background task should automatically be deleted from the database as soon as it finishes (successfully.) By default finalized background tasks are kept in the database until explicitly deleted (individually or in batch.) It is recommended that auto-deletion is only used for internal background tasks that only exist as an implementation detail. Notice that if a task marked as auto-delete fails, or if logging (e.g., information of where output documents are stored) exists, the task will *not* be automatically deleted.
`description`	This method is used to put a descriptive label on a background task. Such descriptive labels makes it easier for a user who is monitoring the background task system (or views past background tasks) to understand what various tasks are all about. There is no semantic information in this concept, but it is considered good practice to use it. In the past examples, this method has been used in every case.

`remark1`
`remark2`
`remark3`
	These methods can be used to add up to three remarks to a background task. There is no semantic information in this.

Method	Remarks
`remarks`	This method takes a value inspector as argument and copies the values of the fields `Remark1`, `Remark2` and `Remark3` to the corresponding remark fields of the background task. These field names need not exist, in which case the relevant remark field will be left unchanged. The value inspector may container other fields.
`accessLevel`	This method takes an access level name as an argument as associates the background task with it. Only users having access to that data access level will be able to see it.
`maxDuration`	This method takes an integer argument specifying the maximum expected duration of the background task. If this method is not invoked, the system-wide default maximum duration will be assumed. The time is measured in minutes. If a background task runs longer than this amount of type, the background-task execution engine may automatically abort the task (leaving it as failed with status "Aborted"), assuming that something out-of-the-ordinary has occurred (such as a broken connection or a server having been forcibly closed while the background task was executed.)
`callbackHandling`	This method is used to declare how document input/output callbacks should be handled. It is possible to specify that output document should be stored on a file server, or e-mailed to a specified recipient. Likewise, it is possible to point out a file location where input files (for the `Load` callback) are found. This is all declared by providing an argument of type `McTaskCallbackHandling`. See below for information on how to obtain such an object.

Whereas most of the above methods are more or less self explanatory and straight forward. This is not quite so for the `callbackHandling` method: this method takes an argument of type: `McTaskCallbackHandling`. Such object are constructed by instantiating a *builder* by invoking: `McTaskCallbackHandling`.`builder`. Doing so will yield a builder type having the following methods:

Method	Remarks
`outputFileHandler1` `outputFileHandler2`	These method can be use to instantiate two different output file handlers. An output file handler can be instrumented to match only files of a certain type. In this way, it is possible to handle, e.g., PDF files in one way, and JPEG files in another. An output handler can use wild-cards to match. As a special case, all files can be matched by a handler. If no matching pattern is specified, no files match. It is possible to use the characters `?` and `*` indicating exactly one character and zero-or-more characters respectively. Apart from the matcher, an output file handler must be given a path to a file-system folder, e.g., to somewhere on a file server. Any output document occurring from a background task with this handler, and which matches the file pattern, will be stored on this location. The file path may be given by a *named reference* to a folder with an optional additional relative path, or by specifying an absolute path. By making use fo the named references, it easily possible to configure slightly different behaviors of, e.g., production systems and test systems. If both output file handlers match the same document, the document will be stored in both locations. Output file handlers will be used for `Show` and `Save` document callbacks.
`outputEmailHandler1` `outputEmailHandler2`	These method are similar to the `outputFileHandler1` method except that they are used to specify a mail recipient. For a document that matches an e-mail handler, the document will be sent as a mail to that e-mail recipient. In addition to specifying a file-matching pattern, the recipient e-mail address must be specified, as well as the sender e-mail address (e.g., `no-reply@trifolium.com`.) Files matching several output e-mail handlers will be handled by both. Also, if an output document matches both a file- and an e-mail handler, it will be handled by both. In this way it is possible to have output documents stored on a file-server as well as having them sent by e-mail.

Method	Remarks
`inputFileHandler1`	This method is used to specify where input documents should be picked up from. This situation may occur if the targeted background action requests file using the `Load` callback. An input handler is given a document pattern match and an input file-path. The input file path can be specified by a *named reference* and an optional additional relative path, or by specifying an absolute file path. It is recommended to make use of the named references since this will make it easy to configure slightly different behaviors for, e.g., production systems and test systems. If a `Load` callback occurs during the background task execution, the specified file-path directory is scanned for files matching the specified pattern, and any such file will be provided as the result of the `Load` callback. In addition, it is possible to specify an associated *output* file pattern. Doing so means that if an output file occurs *after* a `Load` callback and the document name matches this output file pattern, that output file will be placed in the same directory as the input file was found in. This may be used with Imports where a result log is typically shown after an import file has been uploaded.
`build`	This method ends the builder and constructs a `McTaskCallbackHandling` object that can be used as argument to the `callbackHandling` method of the background task builder.

Let us examine the file handling in more detail. Suppose we want to create a background task that posts whatever journals are marked as "To be posted" in the `maconomy:Posting` container. Since no end-user will sit around and wait for the posting journals to appear, instead we want to send the output to `accounting@trifolium.com`. Listing 6.35 shows how to do this. First, a background-task callback handler is created. It specifies that output documents matching the pattern "*.pdf" (line 2) should be sent to the e-mail address `accounting@trifolium.com` (line 3) and that the mail should appear as having sender `no-reply-maconomy@trifolium.com` (line 4.) The building of the callback handler object occurs in line 5. In the background task definition, the callback handler is associated in line 10.

Listing 6.35: Specifying E-mail Recipients for Output Documents.

```
1 final McTaskCallbackHandling emailHandler = McTaskCallbackHandling
    .builder()
2   .outputEmailHandler1("*.pdf",
```

```
3                          "accounting@trifolium.com",
4                          "no-reply-maconomy@trifolium.com")
5    .build();
6  containerRunner.batch()
7    .task("Posting")
8    .action("Post")
9    .singleton()
10   .callbackHandling(emailHandler)
11   .description("Post Journals Ready for Posting")
12   .end();
```

In a similar way, you can associate an output file handler to a background task. Listing 6.36 shows an example where the `PrintInvoice` action is run for a specific job. The corresponding output document (the invoice) is stored on a file server (referenced through the named path "`invoice_storage`",) in a job-specific directory, as specified in line 3–4. If the specified directory does not exist, it will be created. Obviously, this requires that the coupling service process has adequate rights to do that. In addition, the invoice will be mailed to: `invoicing@trifolium.com`, which is ensured by also assigning an e-mail handler in line 5. The callback handler is associated with the background task in line 13.

Listing 6.36: Specifying File Destination for Output Documents.

```
1  final McTaskCallbackHandling outputFileHandler =
       McTaskCallbackHandling.builder()
2    .outputFileHandler1("*",
3                        key("invoice_storage"),
4                        jobNumber)
5    .outputEmailHandler1("*",
6                         "invoicing@trifolium.com",
7                         "no-reply-maconomy@trifolium.com")
8    .build();
9  containerRunner.batch()
10   .task("InvoiceSelection")
11   .action("PrintInvoice")
12   .restrictBy("JobNumber", McStr.val(jobNumber))
13   .callbackHandling(outputFileHandler)
14   .description("Print job invoice")
15   .end();
```

Input file handlers are declared similarly. Listing 6.37 shows how. First an input file handler is declared. Line 2 specifies that the import file must have the pattern: `vi_imp*.txt`. In line 3 it is declared where such import files should be found: this is done by referencing a named file path ("`vendor_invoice_integration`".) Line 4 says that any output file that occurs after the import, should be stored at that location. The declaration of the background task, tying it to the `maconomy:ImportVendorInvoices` container and the action `Import` is declared as usual. The input file handler is associated in line 12.

Listing 6.37: Specifying an Input File Handler.

```
1  final McTaskCallbackHandling inputFileHandler =
       McTaskCallbackHandling.builder()
2    .inputFileHandler1("vi_imp*.txt",
3                       key("vendor_invoice_integration"),
4                       "*")
5    .build();
6  containerRunner.batch()
7    .task("ImportVendorInvoices")
8    .action("Import",
9            dataValues().setBool("UseInternalNamesVar", true),
10           McParameters.create())
11   .singleton()
12   .callbackHandling(inputFileHandler)
13   .description("Import Vendor Invocies")
14   .end();
```

Timing Selection Properties

The timing selection properties are used to schedule the background task for a certain date and time. A background task cannot be picked up for execution until *after* the date and time specified for it. Notice that there are no guarantees about *exactly when* the background task will be executed: it just does not happen before the specified date and time.

In this part, the following method exist:

Method	Remarks
due	This method sets the due date and time for the background task. If this method is not invoked, the due date/time will be "now." The method exists in a number of flavours, one specifying the time relative to now, and a couple of variants that specify the precise date and time.

As an example, suppose you want to make a background task that will post all journals due for posting. Only, you want to do that at 10pm, not *now*. Listing 6.38 shows how this may be done. The scheduling occurs in line 7 indicating that the due date and time is today at 10pm (22:00).

Listing 6.38: Absolute Time Schedule of a Background Task.

```
1  containerRunner.batch()
2    .task("Posting")
```

```
3     .action("Post")
4     .singleton()
5     .callbackHandling(emailHandler)
6     .description("Post Journals Ready for Posting")
7     .due(McDate.today(), McTime.val(22, 0, 0))
8     .end();
```

Sometimes you may want to schedule a task and have it run at a time relative to the current time. Listing 6.39 shows how to make a background task that becomes due in 10 minutes. The action being run synchronizes an employee in scope with information in the Deltek Talent Management System. The relative time schedule happens in line 6.

Listing 6.39: Relative Time Schedule of a Background Task.

```
1 containerRunner.batch()
2     .task("Employees")
3     .action("SyncWithHRSmart")
4     .restrictBy(originalData.asKeyValuesCopy("EmployeeNumber"))
5     .description("Synchronize with Deltek Talent Management")
6     .due(10) //minutes
7     .end();
```

Dependency Selection Properties

Dependency selection properties are used to declare dependencies between background tasks. Every task is considered part of a "task group" indicated by some group id. A task group may comprise zero or more tasks.

A background task can be declared as being dependent of a given task group. That means that as long as there are unfinished tasks in that group, *this* task cannot be picked up for execution—no matter its due time. A task is considered "finished" when it has either succeeded or failed! Hence, as long as it is "pending" or "running," it is unfinished.

In this part, the following method exist:

Method	Remarks
`awaiting`	This method is used to declare which task or task group the current task is dependent of. This is also a version that takes an optional `MiBatchCommon` instance (the common super type of tasks and task groups.) This variant can be used to conditionally declare a dependency to a batch task group/batch task or not.

Listing 6.40 shows how to use this feature. The code generates three background tasks.

The second is depending on the finalization of the first, the third depends on the second. To do this, the first background task is assigned to a variable, representing the background task. This happens in line 1. Upon creating the second background task, the dependency of the first is declared in line 15. And the second background task is assigned to a variable in line 10, which is in turn used as a dependency for the third task in line 23.

Listing 6.40: Declaring Dependencies of Background Tasks.

```
1  final MiBatchTask approveBudgetTask = containerRunner.batch()
2    .task("API_JobBudgetsCardByType")
3    .action("ApproveBudget")
4    .restrictBy(JOB_NUMBER, jobNumber,
5                BUDGET_TYPE, fixedPriceBudgetType)
6    .matchBy("Submitted and not Approved")
7    .description("Approve Fixed Price Budget")
8    .end();
9
10 final MiBatchTask calculateRevenue = containerRunner.batch()
11   .task("JobRevenueRecognitionDetails")
12   .action("CalculateRevenueRecognition")
13   .restrictBy(JOB_NUMBER, jobNumber)
14   .description("Calculate Revenue Recognition")
15   .awaiting(approveBudgetTask)
16   .end();
17
18 containerRunner.batch()
19   .task("JobRevenueRecognitionDetails")
20   .action("ApproveRevenueRecognition")
21   .restrictBy(JOB_NUMBER, jobNumber)
22   .description("Approve Revenue Recognition")
23   .awaiting(calculateRevenue)
24   .end();
```

Sometimes you may or may not be dependent of some other background task. Listing 6.41 shows a use case of that: a number of background tasks are made, each creating a new job. In principle, these tasks need to be dependent of each other. However, in order to get most out of the background task execution engine, in this case, we generate a long sequence of tasks, each one depending on the previous one. This means that the background tasks will never be run concurrently: doing so would only mean that each of the current tasks will await for each other because of database locks obtained upon drawing a system number from the job number series. Instead of ending up in that situation, we might as well declare the tasks as a long sequence of tasks, thereby allowing concurrent processes to do things that *can* be done concurrently with creating jobs. The code works by having a notion of the "previous task" (in line 5.) This is an optional `MiBatchTask` which is initially undefined. Then we loop over all of the data input that each will be used for job creation data. Each time we reassign the previous task, and declare a dependency to the previous task (line 11). And thereby a chain of tasks is obtained. Since the previous task is an optional, and is initially undefined, it means that

the for the *first* background task, there is no dependency, for the second one, the first task is declared as the dependency task etc.

Listing 6.41: Conditionally Declaring Task Dependencies.

```
1  // Generate a sequence of tasks each awaiting the previous
2  // in order to avoid unnecessary database locking
3  // of concurrent tasks:
4  // Create Job 1 <-- Create Job 2 <-- Create Job 3 <-- etc.
5  MiOpt<MiBatchTask> previousTask = McOpt.none();
6  for (final MiValueInspector newJobData : newJobsDataSet) {
7    previousTask = opt(containerRunner.batch()
8      .task("Jobs")
9      .create(newJobData)
10     .description("Create Job")
11     .awaiting(previousTask)
12     .end());
13 }
```

Defining Groups of Multiple BackgroundTasks

Sometimes you need to declare that a task should await the completion of *several* other tasks. For this purpose, you can make use of the task group concept. A group is defined by invoking the `group` method on the `MiScheduler` interface.

As an example, suppose you have made a custom action in the `maconomy:Employees` container called `trifolium:NotifyPMJobs` which sends an e-mail to the employee in question with the list of open jobs for which that employee is the project manager.

Now suppose that we implement some functionality which is capable of replacing the project manager on all open jobs, and suppose we need to do that as background tasks. Once the replacement is done for all of those jobs, we want the new project manager to be notified that he/she is now managing a number of new jobs.

We could do that by chaining the background tasks so that the replacement is first done for one job, then the next and so on until all jobs have been handled, and finally invoking the custom action `trifolium:NotifyPMJobs`. If we do it like this, however, the replacement of the project manager cannot be executed in parallel. By making use of the `group` functionality, we can declare that all the background tasks about replacing the project manager belong to the same group, and the `trifolium:NotifyPMJobs` can then await the completion of the entire group. Meaning that as long as any of the background tasks in that group are still unfinished, the `trifolium:NotifyPMJobs` cannot be invoked.

Listing 6.42 shows how this could be implemented. First, the original project manager and the replacement project manager is derived (it is not important how.) Next, the open jobs of the original project manager are picked up. In line 12 we prepare a background task group initiator. On such an initiator, concrete tasks can be assigned. Then the

code loops over each of the jobs for which a replacement is going to happen. For each of those, a new task (specifying an update with the required change) is declared, simply by invoking the `task` method on the initiator (line 16.) From there, the task properties of a background task can be declared as we have seen above. After the loop, if any tasks were put into the group, we create the task invoking the `trifolium:NotifyPMJobs` action of the new project manager. This task is declared in line 24–30. Of special interest is the dependency declaration in line 29: here the task group is finally built, and the this task is declared as being depending on that group. Meaning that it cannot be executed until every background task of that group has been executed. The tasks in the group may be executed concurrently.

Listing 6.42: Declaring Task Groups and Dependencies.

```
1  final McStringDataValue originalProjectManager = getOriginalPM();
2  final McStringDataValue futureProjectManager   = getFuturePM();
3  final MiQueryInspector replacePMJobs = getDatabaseApi()
4    .mselect(JOB_NUMBER).from(JOB_HEADER)
5    .where().eq(PROJECT_MANAGER_NUMBER).str(originalProjectManager)
6    .and().not(CLOSED)
7    .getResult();
8  final MiValueInspector changePM =
9    dataValues().setStr(PROJECT_MANAGER_NUMBER,
10                       futureProjectManager);
11
12 MiTaskGroupInitiator groupBuilder = containerRunner.batch().group
       ();
13 MiOpt<MiTaskGroupMisc> group = McOpt.none();
14 for (final MiValueInspector job : replacePMJobs) {
15   group = opt(groupBuilder
16     .task("Jobs")
17     .update(changePM)
18     .restrictBy(job.asKeyValuesCopy(JOB_NUMBER))
19     .description("Replace Project Manager"));
20   groupBuilder = group.get();
21 }
22
23 if (group.isDefined()) {
24   containerRunner.batch()
25     .task("Employees")
26     .action("trifolium:NotifyPMJobs")
27     .restrictBy(EMPLOYEE_NUMBER, futureProjectManager)
28     .description("Notify Project Manager")
29     .awaiting(group.get().end())
30     .end();
31 }
```


Chapter 7

Advanced Topics

In this chapter, we shall have a look at some of the more advanced features of the Extension Framework. The main functionality behind the topics in this chapter has already been introduced. The topics in this chapter has been left out from the previous chapters in order to increase the clarity.

7.1 Determining the Order of Container Contributions

By now, it should be clear that a container may contain several container contributions. As mentioned in Section 2.3, the Extension Framework will order these contributions in a specific way. Usually, the programmer writing a container contribution should not care about the order of these contributions, except that the root contribution is always at the bottom. Implicating that every contribution can rely on the behavior of the root.

Why do various extension contributions not need to know about each other? The truth is that sometimes they do need this. Often they don't! You should strive to make your extensions indifferent of the ordering. This makes good sense. Suppose that someone makes an extension that adds some checks to the submission of time sheets. At a different time, you are asked to make an extension to the time sheets container as well: this time adding a variable that calculates the utilization %. This is obviously made in a different extension: the two pieces of functionality have *nothing* to do with each other. And they are best left this way in order to increase the maintainability of the code, and to lower the complexity. In this case, it makes absolutely no difference which extension comes first.

There are, however, situations where the order *does matter*. For example, suppose that someone has made an extension that adds an action to a container. At a later point you need to extend the functionality of that action. But since it's not your code, you shouldn't change the code. Maybe you don't even have access to it! Remember what

happens when a new (root) action is introduced in an extension (see Figure 4.12.) In this case, *no* container contributions between the place where the action is introduced and the root contribution are even notified about that the action occurs. Consequently, if you want to *extend* the behavior of that action, your contribution *must* come before the one introducing the action. In such a case, you can specify the required ordering in the `plugin.xml` file that defines the presence of your extension.

Remember that you must specify an `id` for your contributions in the `plugin.xml` file. These id's must be globally unique. For this reason, we encourage to use a naming convention like the one usually used for Java packages: first, you revert the domain name of your customer or your organization, e.g., `com.trifolium`. Next you give an optional string that indicates the type of container you extend and something more specific relating to the functionality in question. For example, `com.trifolium.TimeSheets.UtilizationPct`. This name is *public API*, and you should think twice before changing it! Other programmers around the world might depend on it!

Listing 7.1 shows an example of a `plugin.xml` file which contributes two extensions to the `Jobs` container. Notice that these two contributions will often be declared in separate bundles, and consequently, in separate files. In the example one contribution adds an action to the container. That contribution has its `id` specified in line 7. The other contribution extends the functionality of that action. In line 18 the other contribution declares that the contribution adding the action must be invoked *before* this contribution. Hence, that the contribution extending the added action will be *above* the one adding the action in the first place. Think of it in this way: the root is positioned at the very bottom. So "above" means that *pre*-events are invoked before the root contribution is invoked.

Listing 7.1: Declaration of Inter-Dependant Contributions.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <?eclipse version="3.2"?>
3  <plugin>
4     <extension name="Extending standard containers"
5                point="com.maconomy.api.container">
6        <extend container="Maconomy:Jobs"
7                id="com.trifolium.Jobs.AddAction">
8           <factory
9              class="com.trifolium.examples.containers.AddAction$Factory">
10          </factory>
11       </extend>
12
13       <extend container="Maconomy:Jobs"
14               id="com.trifolium.Jobs.ExtendAddedAction">
15          <factory
16             class="com.trifolium.examples.containers.
                  ExtendAddedAction$Factory">
17          </factory>
18          <above id="com.trifolium.Jobs.AddAction" />
19       </extend>
20    </extension>
```



```
21 </plugin>
```

In a similar way, it is possible to specify that a given contribution must be positioned *below* some other contribution (i.e., that it must be positioned closer to the root than some other specified contribution.) This is done in a similar way, by using a `<below>`-tag.

Obviously, it is possible to declare cycles in the dependencies. E.g., contribution `X` must come before `Y`, `Y` must come before `A`, and `A` must come before `X`. If such a cycle is detected at run-time, an error will be given, and the container will not be able to start!

7.1.1 Grouping of Container Contributions

In addition to declaring specific ordering dependencies between contributions, it is possible to declare the "grouping" of contributions. There are three different groups:

`root` The root implicitly is part of this group, and the root will always be positioned at the very bottom of a chain of container contributions. In addition, extension contributions can be put into the `root` group. Such extension contributions will be positioned closer to the root than other extension contributions. If several extension contributions of type `root` exist, these will be ordered in some manner. It is possible to declare specific ordering constraints for these using the `<above/>` and `<below/>` declarations. References can only be declared for contributions belonging to the same group.

`standard` By default, an extension contribution will be considered belonging to the `standard` group. I.e., it is optional to specify this group membership. Extension contributions of this group will appear above all of the contributions belonging to the `root` group. Dependencies (`<above/>` and `<below/>`) can be declared against other contributions of this group. Normally, extension contributions should go into this group.

`top` The `top` group contains extensions that appear before any extensions in the groups `root` and `standard`. Dependencies (`<above/>` and `<below/>`) can be declared against other contributions of this group. This group is slightly different compared to the `root` and `standard` groups, since there is only *one* `top` group for a container, also if it is cloned. See Figure 7.1 and Figure 7.4 below for more details.

Figure 7.1 shows how container contributions are organized. The actual root contribution is always at the bottom. Above that, the extension contributions declared as having the group `root` are found. Above those, the extension contributions of the group `standard` are found, and at the very top, the extension contributions of the group `top` are found. As mentioned earlier, it is possible to declare specific ordering dependencies between extension contributions. But you may only declare such dependencies between contributions of the same group. Hence, in Figure 7.1, it is not possible to declare dependencies between, say, Ext_{R_1} and Ext_{S_1} since these two extension contributions belong to different groups. It *is*

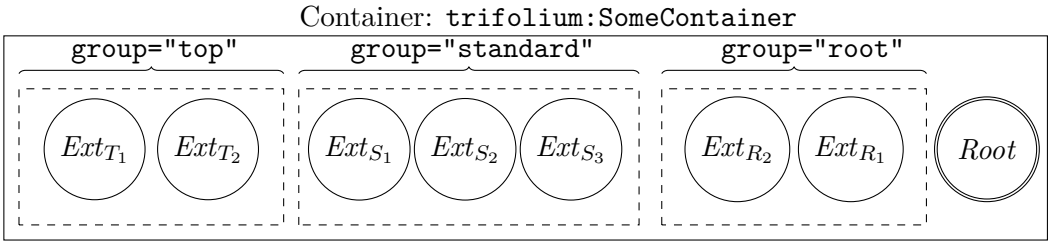


Figure 7.1: Extensions for a container are organized within the declared group. Extensions in the group `root` are closest to the root of the container, extensions in the group `top` are at the opposite end.

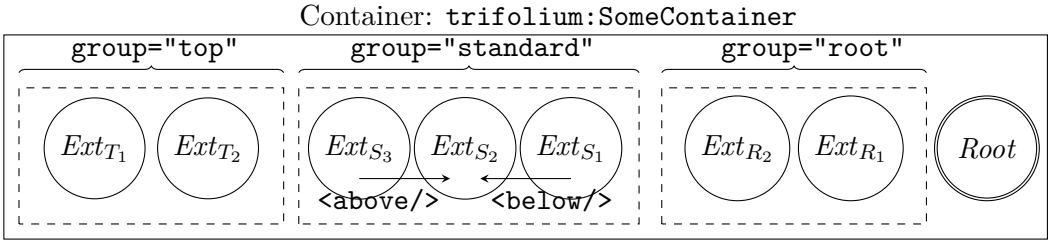


Figure 7.2: Contributions found in the same group can declare ordering dependencies to other contributions in that group. In this case, Ext_{S_1} has declared that it must be *below* the extension contribution Ext_{S_2}. Similarly, extension Ext_{S_3} has declared that it must be *above* the extension contribution Ext_{S_2}.

possible to declare ordering dependencies between Ext_{S_1} and Ext_{S_2}, since both belong to the same group. This is shown in Figure 7.2. Inside each of the groups, the various extension contributions are, by default, ordered in the following way:

- Closest to the "root end", extension contributions that are made for *any* container will be found. This can be specified by `<extend any="true">` rather than specifying a name or a name space.
- Above those, the extension contributions for a specific *namespace* are found. This can be specified by `<extend namespace="trifolium">`.
- Above those, the extension contributions for the specific *container name* are found. This can be specified by `<extend container="trifolium:SomeContainer">`.

It is possible to make ordering dependencies between these. But it is only possible to refer to other extensions at the same extension declaration level (any, name space or container name) and by referencing contributions that originate from a declaration level that, by default, is found closer to the root. This is shown in Figure 7.3: two extensions that was declared for `any` container, may have ordering dependencies declared between them. But an extension specified for `any` container cannot declare that it must come

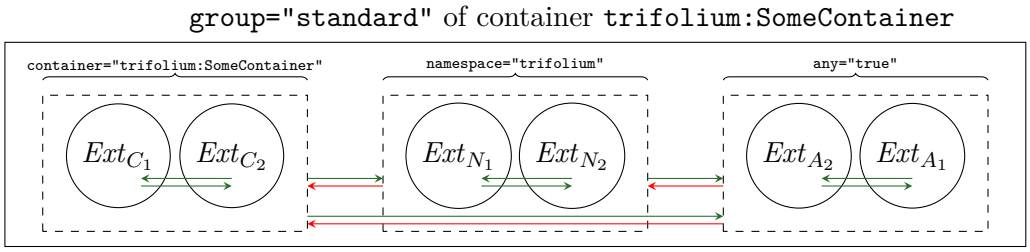


Figure 7.3: Referencing between extension contributions originating from different matching levels. Contributions Ext_{C_1} and Ext_{C_2} are declared as extensions for a specific container name. Contributions Ext_{N_1} and Ext_{N_2} are declared as extensions for all containers in the name space `trifolium`, and the contributions Ext_{A_1} and Ext_{A_2} are declared as extensions for *any* container. The green arrows indicate where it makes sense/is allowed to declare ordering dependencies. The red arrows indicate where this is not so. For example, it makes no sense that a contribution for *any* container explicitly states that it must come above or below an extension for a specific container (since that other extension will only occur for *one* container.)

above or below some contribution that is not in scope for any container. An extension specific to a named `container`, or a specific `namespace`, can, however, declare that it must be ordered before or below a contribution that is in scope for `any` container.

7.1.2 Cloned containers and Ordering of Extension Contributions

The rules explained above always apply. However, when a new container is created by cloning, the ordering of extension contributions are made relative to the cloning level.

In short, you can create a container that is "a clone" of another container, where the entire content of the container being cloned is seen as the "root" for the name provided for the new container.

```
1 <create container="Trifolium:SpecialJobs">
2   <clone container="Maconomy:Jobs" />
3 </create>
```

In the snippet above, we create a new container called `Trifolium:SpecialJobs`. It is based on a *clone* of the `Maconomy:Jobs` container. Meaning that the root *as well as any current and future extensions to* `Maconomy:Jobs` will be seen as the "root behavior" of the container `Trifolium:SpecialJobs`. By doing so, it is possible to extend the container `Trifolium:SpecialJobs` *without* modifying the behavior of `Maconomy:Jobs` any further. Containers can be cloned as many times as you want, so the `Trifolium:SpecialJobs` could again be cloned and exposed through a new name: `Trifolium:OddSpecialJobs`. Hence, the contents of `Trifolium:OddSpecialJobs` will comprise of everything comprised

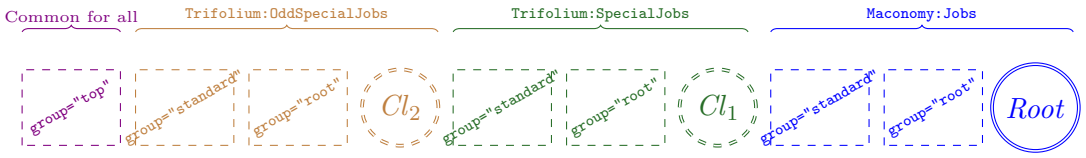


Figure 7.4: Grouping of extensions when cloned containers are involved. In this example, the container in scope is `Trifolium:OddSpecialJobs`. That container is made from a clone of `Trifolium:SpecialJobs`, which in turn is made from a clone of `Maconomy:Jobs`. Contributions relating to each of these names are grouped together in the same way as explained above. The top group, however, is shared between all of the involved container names. Hence, there is only one "top" group for a given container. The two nodes Cl_1 and Cl_2 represent the system-provided "root" nodes for a container created by cloning another container.

by `Trifolium:SpecialJobs` with the extensions specific to `Trifolium:OddSpecialJobs` on top of that. Since `Trifolium:SpecialJobs` itself is a clone, it comprises everything comprised by `Maconomy:Jobs` with any extension applicable for `Trifolium:SpecialJobs` on top. Hence, the contents of `Trifolium:OddSpecialJobs` contains:

- The root of `Maconomy:Jobs`
- Any extension that matches `Maconomy:Jobs`
- A system-provided "root" of the `Trifolium:SpecialJobs` that takes care of the name transitioning.
- Any extension that matches `Trifolium:SpecialJobs`
- A system-provided "root" of the `Trifolium:OddSpecialJobs` that takes care of the name transitioning.
- Any extension that matches `Trifolium:OddSpecialJobs`

For each of these container names that are in scope in this chain of container names, extensions with `group="root"` will be organized close to the "root" node of the name in scope. Extensions with `group="standard"` will be placed above those ones. As a special thing, extensions with `group="top"` will be put in *one common group* at the very top. Figure 7.4 illustrates this: there is just *one* `top` group, and for each involved container name (cloned or not) the applicable extension contributions of the groups `root` and `standard` are organized as usual. In case some extension (not belonging to the `top` group) can apply to several of the named scopes, they will go to the earliest place (e.g., towards the root.)

7.2 Building Generally Applicable Extensions

Until now, we have considered situations where we either create a specific container with a specific name, or where we extend a container with a specific name.

Sometimes, you can benefit from making an extension that is *generally applicable*. You can do this with the extension framework. This feature has been used in several instances, including:

`ExportDataSet` ("Export to Excel.") This action has been implemented as a generic extension. Consequently, all panes in all containers, including *future containers*, will have this action available, although it is not visible in card panes by default.

`EmailOnAction` is an action that makes it possible to execute an action specified in a parameter, and send an e-mail to some e-mail address which is also specified in a parameter. The content of the e-mail may or may not contain attached documents resulting from running the action. This action is available for *all* panes in any container, including *future containers*, since the action has been specified as being generic.

`RunReport` is an action that makes it possible to generate a Business Objects report based on settings specified in parameters. Since this action has been specified as being generic, this action is available for all panes in any container, including *future containers*.

`ActionSequence` is an action that makes it possible to run a number of other actions in a sequence. The actions to be run in sequence are provided using paramters to the action. Since this action has been specified as generic, it is available for all panes in any container, including *future containers*.

Such generic extensions may save a lot of coding because the contribution will be present anywhere.

Listing 7.2 shows an example of how to declare an extension as generic. In line 5 the declaration says `any="true"`. This means that this extension is applicable for *any* container. In addition, it says `group="top"`. This means that the extension will be placed in a "top" group, thereby making it something that generically applies on top of all containers.

Listing 7.2: Declaration of a Generic Extension.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <?eclipse version="3.4"?>
3 <plugin>
4    <extension point="com.maconomy.api.container">
5       <extend any="true" group="top" id="com.trifolium.extension.generic.
            my">
6         <factory class="com.trifolium.my.generic.extension.
              MyGenericExtension$Factory" />
```

```
7         </extend>
8       </extension>
9   </plugin>
```

In addition to making an extension for any present or future container, it is possible to make a generic implementation for an entire *class* of containers. This can be done by making a root contribution which is not attributed to any specific container, but is instead attributed to a *namespace*. Suppose that you need to build a series of containers which all communicates with some specific back-end, e.g., a salary system. And suppose the the way to communicate with the salary system back-end is almost identical; the only difference being the name of the part of the system which is addressed, represented as a container. Using this feature, you can implement one generic root container and associate that implementation with a specific name space. This will allow the Extension Framework to recognize all container names under a given name space, and to invoke the common implementation for all. This is how the implementation for the name space `maconomy` has been made. The only difference compared to creating a specific root container is that instead of specifying the `name` attribute, you specify the `namespace` attribute instead!

Usually, an error will be given at run-time, if two different contributions claim that they are both the root contribution for a given container. This is not the case, however, if one contribution is a `namespace`-type and the other is a `name`-type (i.e., an explicitly named container.) In this case, the `namespace`-contribution is disregarded, and the `name`-type will take precedence as the sole root contribution!

7.3 Dynamically Changing an Event Flow

In Chapter 4 we have seen how the life-cycle of data-carrying events invokes "Pre" and "Post" scripts in every contribution of a container, as illustrated in Figure 4.2.

Sometimes it is needed to act slightly differently, by *skipping* all remaining contributions. There could be several reasons for that:

1. You have introduced an event which is not supported in following contributions. This is for example the case when you introduce a root action in an extension contribution.
2. You have changed the data-model so significantly, that a given event may not make sense in following contributions. This could be the case, for example, if you "emulate long text" by joining several lines together as one logical line. An "update" of such a line makes no sense in following contributions, but will have to be "re-mapped" into other events.
3. If your logic programmatically executes events on the current record in the "Pre" script, the event will not work for following contributions because it will be believed

that the data has been changed without your knowing. In such cases, you must short-cut the event-flow.

The `containerRunner` parameter has a method called `skip`. This method will ensure that the following contributions are short-circuited. Instead of continuing the flow by invoking the "Pre" script of the next contribution, the "Post" script will be invoked on your contribution right after the "Pre" script terminates. It is not possible to invoke the `skip` method in "Post" events, as this makes no sense.

When a root (new) action is introduced in some extension, the framework *knows* that it is new and therefore is completely unknown by the following extensions. And the framework will automatically invoke `skip`.

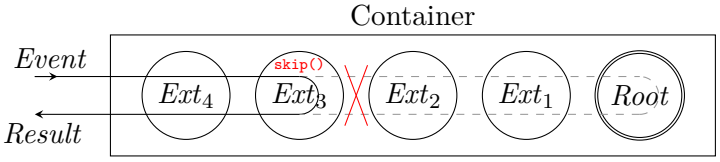


Figure 7.5: Invoking the `skip` method in contribution Ext_3 will cause the following contributions to be disregarded for the current event.

Figure 7.5 shows the event flow of some event in a case where the `skip` method is invoked in Ext_3: this causes the following contributions: Ext_2, Ext_1 and *Root* to be "skipped" in this particular event flow. The result is that the "Post" script of Ext_3 is invoked immediately after the "Pre" script terminates. The `skip`ping is *only* taking place for this particular event. Any other event being run, programmatically or not, will work as usually.

7.4 Enforcing Full Data Refresh

Sometimes an operation makes changes not only to the current record, but to other records of the same container. For example, an action in a card pane could "delete all lines" or "recalculate all lines." In such cases, you need to enforce that the response eventually going to the client contains *all* data of the container, not just, e.g., the "current record." Or maybe your extension makes changes to this record and this container in the "Post"-script by programmatically running a number of additional operations. For example, during a `Create` process, your extension might choose to `Update` one or more fields in this container in the "Post"-script. In this case, the data resulting from the `Create` operation is no longer the most recent data. If you don't do something about this, a "data has been changed" error will likely follow!

The solution to this problem is to enforce a full data refresh. The Extension Framework gives the possibility of invoking the method `fullRefresh`. This method basically raises

a flag to notify the framework that a re-read should be made immediately after this event is done. It makes no difference whether this method is invoked one or 100 times—only *one* re-read will be made. And the re-read will happen *after* this event has been fully completed in all contributions.

Listing 7.3 shows an example where a specific time sheet line is created whenever a time sheet is created (a line automatically filling out required information for a weekly department meeting which takes place every Tuesday.) When doing so, it is necessary to do a full refresh because the data in another pane has been altered. This takes place in line 18 where the `fullRefresh` method is invoked, thereby indicating that the Extension Framework must perform a re-read of the container immediately after the `Create` operation is done.

Listing 7.3: Enforcing a Full Data Refresh

```
2   private static final MiDataValues WEEKLY_DPT_MEETING =
3     dataValues().setStr("JobNumber", "210770")
4                 .setStr("Task", "200")
5                 .setReal("NumberDay2", BigDecimal.ONE);
6   @Override
7   public void onCreatePost(final MiCreatePost containerRunner,
        final MiCreate eventData) throws Exception {
8     // Upon creating a time sheet line, we also
9     // create a specific time sheet lines
10    // covering Weekly Department Meeting
11    final MiKeyValues tsKey = eventData.getResultData().
          asKeyValuesCopy("EmployeeNumber", "PeriodStart");
12
13    final MiContainerExecutor timeSheetTable = containerRunner.
          executor().construct(MePaneType.TABLE);
14    timeSheetTable.control().restrictBy(tsKey);
15
16    timeSheetTable.add().create(WEEKLY_DPT_MEETING);
17
18    containerRunner.fullRefresh();
```

7.5 Accessing Configuration Settings and Integrating with 3rd-Party Systems

Sometimes you may want to integrate to 3rd-party systems. The communication with such systems could happen in several ways, including:

- By using the file system
- By accessing a URL
- By means of a mail server

However, you likely don't want to mix the behavior of a production system with that of a test system. Imagine that you make an extension that picks up vendor invoice imports from a location where an external invoice scanning system places such files. If that file-path is hard-coded you are bound to having problems: the production system and the test system may compete about importing scanned vendor invoices. With the result that some go to the production system while others go to the test system.

Similarly, if you need to integrate, for example, a salary system through a URL, you don't want to let the production salary system be updated from the test system. Instead, you would like a test-installation of the salary system to be updated by the test system, while the production salary system is the one being updated from the production system.

This can be achieved by making use of the configurations in the Coupling Service's `server.ini` file which resides in the `configuration` folder in the Coupling Service's installation directory.

The interface `MiConfigurationInfo` is used to represent part of the information found in this file. This interface is obtained by invoking the factory method `create` in the class `McConfigurationInfo`. The resulting interface has the following methods:

Method	Remarks
`getSystemNature`	Returns the system nature defined for the current short name. If no system nature is defined, an empty (undefined) value is returned.
`getEmailConfiguration`	This method returns information about the email configurations for this short name through a type called `MiEmailConfiguration`. This type contains information about which mail server is configured for this short name (and the current user), a possible default sender and a possible fixed recipient. In several places (e.g., for background tasks and the `EmailOnAction` extension, this recipient will be used if defined, thereby overshadowing any other recipient.)
`getFilePath`	This method returns a `File` object representing the path currently configured for a specified reference name for the current short name. The method takes an optional additional relative path as an argument. The resulting `File` object represents the folder found by joining the two paths.
`getFilePathOpt`	This method is similar to `getFilePath` except that it returns an optional `File`. In case the file reference is not declared, a `none` value will be returned.

Method	Remarks
`getUrl`	This method returns a `String` representing a URL associated with a given reference name. The value is the one configured for this short name. If the name is not defined, an empty `String` is returned.
`getUrlOpt`	This method is similar to `getUrl` except that it returns an optional value. If the named URL is not defined, a `none` value is returned.
`getSystemNatures`	This method returns a collection of all defined system natures.
`getFilePathReferences`	This method returns a collection of all defined file path reference names.
`getUrlReferences`	This method returns a collection of all defined URL reference names.

As an example, suppose we have three short names: `macoprod` (used as the production system) and `macouat1` (used as user acceptance test system 1) and `macouat2` (used as a second user acceptance test system.) Also, suppose the `server.ini` contains the following information:

```
...
# define macoprod as being of nature "production"
# and define macouat1 and macouat2 to be of nature "test".
system.nature.macoprod=production
system.nature.macouat1=test
system.nature.macouat2=test

# standard mail server
mailserver.address=mailserv.trifolium.com:25
# secondary mail server to be used for test systems
mailserver.address.nature.test=mailtest.trifolium.com:25

# Overshadow the email recipients for test systems
email.to.nature.test=testgroup@trifolium.com

# set the default sender
email.from.default=no-reply@trifolium.com

# declare file path reference names
# standard location for production
filepath.vendor_invoice_imports=//fileserver/scans/vendorinvoices/
# location for test systems
filepath.vendor_invoice_imports.nature.test=//tstserver/vendorinv/
#
# standard location for project folders
```



```
26 filepath.job_reports=//projectshare/reports/
27 # location for test systems
28 filepath.job_reports.nature.test=//tstserver/jobreports/
29
30 # declare URL reference names
31 # URL for salary system
32 url.salary=https://www.supersalarix/trifolium/
33 # URL for salary system used by test systems
34 url.salary.nature.test=https://test.supersalarix/trifolium
```

The above configuration declares a named reference to a file path, `vendor_invoice_imports`. The source code can obtain the file location configured for this name: when run on the production system, the resulting file path location will be `//fileserver/scans/vendorinvoices/`, but when run on one of the test systems, the file path location `//tstserver/vendorinv/` will be used instead. In this way, an extension referencing the named file path location `vendor_invoice_imports`, can obtain the applicable path depending on whether it is run on a test system or on the production system.

Listing 7.4 shows an example of this: an action which is intended to import data from somewhere in the file system—supposedly data generated by some vendor invoice scanning software—picks up the appropriate file system path and then invokes a method that does the actual import (not shown.) The interesting part goes on in line 8: there we ask the configuration to resolve the file path represented by the name `vendor_invoice_imports`. With the configuration settings shown above, this path will resolve to `//tstserver/vendorinv/` when run on one of the test systems (database short names `macouat1` or `macouat2`) whereas on other systems (in this case the `macoprod` short name) then file path will resolve to `//fileserver/scans/vendorinvoices/`. Obviously, if the name `vendor_invoice_imports` has not been declared as a file path in the configuration, the code won't work. For this reason, it is checked whether the file reference is defined or not (line 10.) If this is not the case, an error is shown to the end user, otherwise, we proceed with whatever file path was resolved.

Listing 7.4: Obtaining File Path References from the System Configuration

```
1  private static MiKey VINV_IMPORTS = key("vendor_invoice_imports");
2  @Override
3  public void onAction(final MiActionPost containerRunner,
4                       final MiAction eventData) throws Exception {
5    final MiConfigurationInfo configInfo = McConfigurationInfo.
       create(containerRunner);
6
7    final MiOpt<File> importPath =
8        configInfo.getFilePathOpt(VINV_IMPORTS);
9
10   containerRunner.check(importPath.isDefined())
11                  .error("No location configured for vendor invoice
                           imports - contact your system administrator")
                    ;
```

```
12    importVendorInvoices(importPath.get());
13  }
```

It is also possible to obtain a partly variable file-path reference. For example, suppose we want to implement an action that generates some report for a job, and then stores the report output as a file in the file system. Only, reports for different jobs should be placed in a job-specific sub-folder to a common file path. Listing 7.5 shows an example of this: we extract the relevant job number from the `eventData`, and in line 11 obtain the file path reference of `job_reports`. And as an additional argument to `getFilePathOpt`, we provide the job number. In this way, the resulting `File` represents the sub folder having the same name as the job number under the resolved file path obtained using the reference name. Hence, running this code (with the above configuration) on the production system would result in that the job reports are stored in some location. Running it from one of the test systems results in a different file location; in both locations, however, the job reports are stored in sub folders corresponding to the job number in context.

Listing 7.5: Obtaining Partially Variable File Path References

```
1  private static MiKey JOB_REPORTS = key("job_reports");
2  private static MiKey JOB_NUMBER = key("JobNumber");
3  @Override
4  public void onAction(final MiActionPost containerRunner,
5                       final MiAction eventData) throws Exception {
6    final MiConfigurationInfo configInfo = McConfigurationInfo.
       create(containerRunner);
7
8    final MiValueInspector originalData = eventData.getOriginalData
       ();
9    final String jobNumber = originalData.getStr(JOB_NUMBER);
10   final MiOpt<File> jobReportPath =
11       configInfo.getFilePathOpt(JOB_REPORTS, jobNumber);
12
13   containerRunner.check(jobReportPath.isDefined())
14                  .error("No location configured for job reports -
                          contact your system administrator");
15   storeJobReports(originalData, jobReportPath.get());
16 }
```

Similarly, it is possible to obtain URLs declared by the system configuration. Listing 7.6 shows an example of this: in line 8 we obtain the base URL to some salary system from the system configuration; with the settings above, this would result in different URLs when the code is run from the production system, compared to when running from one of the test systems.

Listing 7.6: Obtaining a URL from the System Configuration

```
1  private static MiKey SALARY_SYSTEM = key("salary");
2  @Override
3  public void onAction(final MiActionPost containerRunner,
```



```
4                       final MiAction eventData) throws Exception {
5   final MiConfigurationInfo configInfo = McConfigurationInfo.
        create(containerRunner);
6
7   final MiOpt<String> salaryBaseUrl =
8       configInfo.getUrlOpt(SALARY_SYSTEM);
9
10  containerRunner.check(salaryBaseUrl.isDefined())
11                  .error("The Salary System is not defined -
                        contact your system administrator");
12  synchronizeSalary(salaryBaseUrl.get());
13 }
```

In cases your extension logic needs to construct emails, it is recommended that you hook into the mail-configuration settings provided for the system in question. Again: such configurations can differ for example between production systems and test systems. There are several things that can be configured:

- The mail server address and port
- A recipient (that is intended to overshadow any "normal" e-mail recipient.) This may be used to ensure that—no matter what—e-mails sent by a test system always go to some internal mail account; never to a customer or vendor
- A default sender, such as `no-reply@trifolium.com`

Listing 7.7 shows how to obtain this kind of information, resolved for whatever system is currently running. In line 8 the overall email-configuration object is obtained from the system configuration. This object comprises several pieces of information. The mail server (address and port) is obtained in line 11. In case no mail server has been configured, the resulting value is an undefined (`none`) value. If it is, we can query the resulting object to obtain the address and port separately. In the code snippet, this is done to log the mail-server information in lines 25–26. The "overshadow" e-mail recipient is obtained in line 18: if it is defined, it is used as the recipient, otherwise, we identify the recipient "as usual." Exactly what "as usual" means depends on the actual business logic. Finally, line 21 looks up which (if any) default sender has been defined in the system configuration.

Listing 7.7: Obtaining Email Configuration Properties from the System Configuration

```
1  private static final Logger logger = LoggerFactory.getLogger(
       SendAnEmailAction.class);
2  @Override
3  public void onAction(final MiActionPost containerRunner,
4                       final MiAction eventData) throws Exception {
5    final MiValueInspector originalData = eventData.getOriginalData
       ();
6
```

```
7     final MiConfigurationInfo configInfo = McConfigurationInfo.
          create(containerRunner);
8     final MiEmailConfiguration emailConfiguration = configInfo.
          getEmailConfiguration();
9
10    final MiOpt<MiMailServer> mailServer =
11        emailConfiguration.getMailServer();
12
13    containerRunner.check(mailServer.isDefined())
14                   .error("No mail server has been defined - contact
                              your system administrator");
15
16    // take into account that the recipient may have been
          overshadowed!
17    final String recipient =
18        emailConfiguration.getEmailTo().getElse(getRecipient(
              originalData));
19
20    final MiOpt<String> optSender =
21        emailConfiguration.getEmailFrom();
22
23    if (logger.isTraceEnabled()) {
24      logger.trace("Using mailserver address={}, port={}",
25                   mailServer.get().getAddress(),
26                   mailServer.get().getPort());
27    }
28    handleDataAndSendMail(originalData,
29                          mailServer.get(),
30                          recipient,
31                          optSender);
32
33  }
```

7.6 Defining Custom Popup Types

As explained in Section 6.2.5, the Maconomy system has a number of so-called popup-types defined. Each type represent a number of values that can be assigned to fields of a specific type. In the workspace client, a field of type popup is shown as a drop-down box. The user is enforced to select a value from the box and cannot start typing a value. Also, searching is not supported. For this reason, use of popups should probably be avoided, especially in cases where there are many possible values. In such cases, using standard foreign-key searches is much better, see Section 4.14.2.

It is, however, perfectly possible to add a field of some defined popup type. But what if you really want to use a popup, but none of the existing Maconomy types suffice? In this case, you can add your own popup types to the system.

Technically, a popup just attempting to read a certain container which happens to have a result of a specific kind. So in prinicple, you can add your own popup-types by implementing such a root container from scratch. While this is perfectly doable, it is a tedious and very bound task. For this reason, the Extension Framework comes with a way to easily add your own custom popup types: all you need to do is to let your container factory extend the abstract class `McAbstractFixedValuesPopupContainerFactory`. Doing so will enforce you to implement one method: `definePopupTypes`. This method takes an parameter of type `McPopupTypeDefiner`. And this type in turn, has a method called `definePopupType`. This method takes as argument the type being defined, and returns a `McPopupValueDefiner`. That class offers a couple of methods:

Method	Remarks
`defineValue`	Defines a single popup value. You must provide the *literal value*, the *ordinal value* and a *title value*.
`defineValues`	This method takes a list of `McPopupDataValues` and associates each of these with the popup type for which values are being defined.

In order to avoid conflict with present and future popup values in the Maconomy core system, *you should always provide a name-space for your popup type!* All Maconomy-defined popups are in the name-space `maconomy` (which is implicit.) You should use a different name-space, such as, `trifolium` (a customer name space) or `salary` (a name space for a specific salary system.) Once a new popup type has been defined, you can refer to it when adding fields and variables.

For example, suppose we add the following popup-type to the system: `trifolium:PhoneType`. Here `trifolium` is the name-space, and the `PhoneType` is the name of the type in that name-space. Then if we wish to add a field of this type, we can write:

```
MiKey fieldPhoneType = key("trifolium:PhoneType"); // field
MiKey phonePopupType = key("trifolium:PhoneType"); // type
return McPaneSpec.McExtended.pane()
  .addPopupField(fieldPhoneType, "Phone Type", phonePopupType).
     open();
```

Notice that the type of the popup field is specified as `trifolium:PhoneType`, i.e., using the full name-space version!

Listing 7.8 shows how to implement a container that defines a custom popup type. Notice that you *only* need to implement the container *factory*. The class implements one method: `definePopupTypes`. This value contributes the type `trifolium:PhoneType` and specifies the values that exist. In addition to the specified value a `nil` value (ordinal value -1) will always be defined.

Listing 7.8: Defining a Custom Popup Type

```
2  public class PhonePopupType extends
       McAbstractFixedValuesPopupContainerFactory {
3    private static final String PHONE_TYPE_NAME_STR = "trifolium:
       PhoneType";
4    private static final MiKey PHONE_TYPE_NAME = key(
       PHONE_TYPE_NAME_STR);
5
6    public static final MiList<McPopupDataValue> PHONE_TYPE_VALUES =
7        McTypeSafe.createArrayList(
8          McPopup.nil(PHONE_TYPE_NAME),
9          McPopup.val(PHONE_TYPE_NAME_STR, "ios", 0, "iOS"),
10         McPopup.val(PHONE_TYPE_NAME_STR, "android", 1, "Andriod"),
11         McPopup.val(PHONE_TYPE_NAME_STR, "windows", 2, "Windows
             Phone"),
12         McPopup.val(PHONE_TYPE_NAME_STR, "blackberry", 3, "
             Blackberry"),
13         McPopup.val(PHONE_TYPE_NAME_STR, "other", 4, "Other")).
             asUnmodifiableList();
14
15   @Override
16   protected void definePopupTypes(final McPopupTypeDefiner definer
       ) {
17     definer.definePopupType(PHONE_TYPE_NAME)
18       .withValues(PHONE_TYPE_VALUES);
19   }
```

Listing 7.9 shows how the contribution would be in `plugin.xml` file.

Listing 7.9: Defining a Custom Popup Type

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <?eclipse version="3.2"?>
3  <plugin>
4     <extension point="com.maconomy.api.container" name="Custom Popup Type"
         >
5
6        <create container="trifolium:PhoneType" id="com.trifolium.popup.
           phonetype" >
7          <factory class="com.trifolium.examples.containers.PhonePopupType
             " />
8        </create>
9     </extension>
10 </plugin>
```

There is one caveat with custom popups: since the Maconomy server knows nothing about these types, they cannot be directly persisted in the database as is. You will have to store them as some other type, e.g., the an integer representing the ordinal value. Likewise, in MOL-definitions, you cannot use custom popup types. Please refer to Section 7.8 for more information on how to handle this situation.

7.7 Implementing You Own Persistence Strategy

In Chapter 4 it is explained how you can inform the framework how data is fetched and persisted. The Extension Framework comes with a few implementations of the interface `MiPersistenceStrategey`. A couple of these relate to using the Maconomy database (`McMolPersistenceStrategy` and `McMolAutoPositionablePersistenceStrategy`.) Others are meant to get something running quickly, but are *completely* unsuited for production (`McTestPersistenceStrategy` and `McAutoPositionableTestPersistenceStrategey`.)

Sometimes you might wish for another persistence strategy. In such cases, you should implement your own persistence strategy (unless your needs can be solved by applying a "codec," see Section 7.8.)

For example, you might want to implement a persistence strategy with one of the following capabilities:

- Persist data using a container. This is necessary, for example, if you must for some reason persist data using standard Maconomy data.
- Persist data that partly uses read-only standard Maconomy data and partly uses custom (read/write) MOL-tables.
- Persist data using a combination of existing persistence strategies.
- Persist data using some web-service.
- A read-only persistence strategy (in reality a fetch strategy.)

There could be several other reasons why you might consider implementing the `MiPersistenceStrategy` interface yourself. In any case, this interface is expected to be implemented by 3rd-parties.

Sometimes you might wish to implement a persistence strategy which is not expected to be used for general searching. This might be the case, if you know your persistence strategy is to be used by card and table panes. In this case, the Extension Framework will look up data using relatively simple queries—key values of the corresponding container. This might be easier to implement than general queries that might impose arbitrary restrictions, search ordering and paging.

In some cases, you may wish to build a persistence strategy, that reads and persists through some container (rather than by using the database directly). In such cases, you may want to make use of the utility class `McValueMapExpressionConverter`. This class provides methods that can convert an expression into a (set of) field/value expressions. The idea is that you create a `McValueMapExpressionConverter` based on an expression that is provided to some method in the persistence strategy. In some cases, you wish to ensure that this corresponds to one or more specific key/value pairs that correspond to a key value of a container or similar. I.e., not abitrary expressions, but expression of the form:

$$
\begin{array}{ll}
 & \texttt{keyField}_1 = v_{1_1} \wedge \texttt{keyField}_2 = v_{2_1} \wedge \ldots \texttt{keyField}_n = v_{n_1} \\
\vee & \texttt{keyField}_1 = v_{1_2} \wedge \texttt{keyField}_2 = v_{2_2} \wedge \ldots \texttt{keyField}_n = v_{n_2} \\
\vdots & \\
\vee & \texttt{keyField}_1 = v_{1_m} \wedge \texttt{keyField}_2 = v_{2_m} \wedge \ldots \texttt{keyField}_n = v_{n_m}
\end{array}
$$

That is that the provided expression has a form that can be seen as a disjunction of m keys. In the example above, the expression says that we want to consider m different keys, where key is specified by the n different key fields, $\texttt{keyField}_1 \ldots \texttt{keyField}_n$. And where each of the n key fields in each of the m keys have a certain value. Thus, the first key value is $\texttt{keyField}_1 = v_{1_1}$, and $\texttt{keyField}_2 = v_{2_1}$, and $\texttt{keyField}_n = v_{n_1}$.

In simple cases, there could be just one key ($m = 1$). Of course, as an extension programmer, you could write this code. However, traversing the structures of expressions is not the most straight-forward thing to do. Therefore, the `McValueMapExpressionConverter` has been introduced to help with this task: it can convert expressions of the above form into a series of `MiValueInspectors` that can easily been used to address containers using container executors. If the expression is of a more complex form, the utility class will throw an `McError`. In cases where you want to use, e.g., other containers to host the values of certain fields, it makes little sense to support expressions of other forms. If it does, it is likely not very generic in nature, and you will have to implement the expression converter yourself.

Please refer to the in-line IDE JavaDoc help for more information.

7.7.1 Persistence Strategies for Storing Long Texts

As the Maconomy server does not natively support text strings longer than 255 bytes, it is always an issue how to address longer texts. By implementing a persistence strategy that can add long texts by storing the data as smaller chunks of text is a possibility. And that would require you to implement a persistence strategy, likely using the `McValueMapExpressionConverter` mentioned above.

However, as this is considered a thing that many programmers would eventually like to do, the framework comes with an implementation of a persistence strategy that can do this: `McTextsPersistenceStrategy`: this persistence strategy can store (and fetch) arbitrarily long texts by peristings them as several records. This is done using the underlying `maconomy:Texts` container that was introduced in version 16sp2 (2.1.1).

In order to make it even easier to add long-sized strings to containers, the Extension Framework even comes with an abstract data-model implementation that will automatically use this persistence strategy: `McAbstractLongTextFieldExtendedDataModel`. Using

this data model, an extension programmer only has to implement *one method* to add one or more "emulated" long-text fields[1]. This method, `defineTextContentMapping`, requires the programmer to specify:

- How many long-text fields should be added
- For each field: which (already existing) text field (of "normal size") is used to store a reference to the record set that contains the long text.
- The title of each long-text field.
- A possible length limit (this is optional and may be left unspecified which means "no limit").
- The "openness" of the long text field (i.e., whether it is open in the create and/or exists state). This state *must* follow the settings for the underlying reference field, or more restrictive. For example, if the underlying reference field is not open in "create", then the long text field must also not be open in "create".

All of these properties are specified using a type called `McTextContentMapper`. Such objects are created using a *builder* object: `McTextContentMapperBuilder`. You obtain such an object by invoking the static `create` factory method on the `McTextContentMapperBuilder` class. The builder class has the following methods:

Method	Remarks
`create`	The factory method: this method must be given a list of the key fields of the record/pane that you are *extending*. For example, if you extend the card of the `maconomy:Employees` container, you must specify the key field for records in that pane, i.e., `EmployeeNumber`.

[1] We refer to such fields as "emulated" because the field is not natively stored as one field in one record; this may lead to less flexibility than you are used to with ordinary fields!

Method	Remarks
`textValueField-FromTextKeyField`	This method specifies a new long-text field. The method is found in a number of flavours, but it basically specifies: • The name of the (new) emulated long-text field. • The name of the (existing) normal-sized text field that will be used to contain a reference to the data holding the long text. • The title of the (new) emulated long-text field. • The openness of the (new) emulated long-text field. This can be left out in some variants of the method, which will be interpreted as open in all states. • The length-limit of the long text field. This value can be left out in some variants, which will be interpreted as "unlimited."

For each emulated long-text field that you wish to add, you declare a text-content mapper and call the method `textValueFieldFromTextKeyField` for each of them.

Using this abstract data-model class will allow you to easily add emulated long-text fields. But only those. If you wish to add other fields as well, you should add those in an additional data model (otherwise you will have to implement your own persistence strategy and/or data model). And there is really no need for that.

As an example, suppose we would like to add two emulated long-text fields to the `maconomy:Jobs` container: `LongDescription` and `LongRemarks`. Furthermore, we wish to use the fields `Text7` and `Text8` to use as references to the data areas where the two long text fields are stored.

Listing 7.10: Adding Two (Emulated) Long Text Field to Jobs

```
2 public class AddLongFieldsToJobsDataModel extends
      McAbstractLongTextFieldExtendedDataModel {
3   private static final MiKey JOB_NUMBER = key("JobNumber");
4   private static final MiKey LONG_DESCRIPTION = key("
      LongDescription");
5   private static final MiKey LONG_REMARKS = key("LongRemarks");
6   private static final MiKey TEXT7 = key("Text7");
7   private static final MiKey TEXT8 = key("Text8");
8
9   public AddLongFieldsToJobsDataModel(final MiResources resources)
        {
```



```
10      super(resources);
11    }
12
13    private static final McTextContentMapper TEXT_CONTENT_MAPPER =
14        McTextContentMapperBuilder.create(JOB_NUMBER)
15          .textValueFieldFromTextKeyField(LONG_DESCRIPTION,
16                                          TEXT7,
17                                          Terms.longDescription())
18          .textValueFieldFromTextKeyField(LONG_REMARKS,
19                                          TEXT8,
20                                          Terms.longRemarks())
21          .build();
22
23    /** {@inheritDoc} */
24    @Override
25    protected McTextContentMapper defineTextContentMapping(final
          MiDefine containerRunner) {
26      return TEXT_CONTENT_MAPPER;
27    }
```

Listing 7.10 shows an implementation of a data model that will extend the card pane of the `maconomy:Jobs` container with two fields. Each of the added fields are represented to the end-user as long-text fields. The long-text content is "emulated," and the data is persisted by using the `maconomy:Texts` container. The price is that you need to give up an existing (limited) `String` field for each of the added long-text fields. In addition to this, there's the increased performance overhead in reading/storing the long text content.

The data-model class extends `McAbstractLongTextFieldExtendedDataModel`. This requires us to implement one method: `defineTextContentMapping`. This implementation can be seen in line 25. As it can be seen, this method merely returns a statically defined constant. This constant is defined in lines 13–21. For each of the added long-text fields, a call to the method `textValueFieldFromTextKeyField` is made. The first call declares the long text field `LongDescription`, and it specifies that the existing field `Text7` is used to hold the reference to the data. The title of the long text field is obtained as a dynamically localized term (see Section 7.10 for an explanation of this.) Similarly, the other long-text field: `LongRemarks` is declared allocating the field `Text8` to hold the reference to the field.

Notice: In order for this implementation to work successfully, the end-users must have read/write access to the `maconomy:Texts` container.

You can also add long-text fields to table parts. You should, however, be particularly sensitive about performance in this case: if the table contains many lines, the overhead may be annoying, since it is necessary to fetch the long text fields (which each correspond to reading a table in the `maconomy:Texts` container) for each line. If you even add more than one long-text field for each line, this issue becomes event more important! *So always*

carefully consider whether it will benefit your customer to add long text fields to a table part. The work-around could be extending a card-pane container (showing the contents of a given line in the card part) with the long-text fields. And then showing that card pane as an assistant to the table in question. In this case, the long-text is only needed when the assistant is visible, and only for the *current line* of the table.

Anyway, let us assume that you *do* want to add a long-remarks field to the purchase orders table. I.e., you want every line in the table to contain a long-text remark field. The first thing to notice is the formal key fields of a purchase order line. Those are: `PurchaseOrderNumber` and `LineNumber`. Now, since the text-content-mapper needs information about the formal key fields of the underlying table, it would be an easy mistake to specify these two fields. *This will not do!* The problem obviously is that the field `LineNumber` does not form a key that is invariant; as the end-user adds or deletes lines, the key fields of the various lines will change! A very nasty property for key fields! In this case, you should instead use some other set of fields which is guaranteed to be invariant and unique. Fortunately, all Maconomy database tables contain such a field: the `InstanceKey`. Obviously, you need to ensure that instance keys are enabled for this particular table. This is managed from the container `maconomy:DatabaseRelations`.

In the example, we are going to present, we shall use the `InstanceKey` field to uniquely identify key fields. Furthermore, we need to point out an existing field that is used to contain the reference to the contents of the added long-text field. In the example, we shall use the field `SupplementaryText10`. We notice that this field is closed in the "create" state, i.e., when creating new lines. Therefore, the added long-text field also needs to be closed in this state.

Listing 7.11: Adding an Emulated Long-Text Field to Purchase Orders Table

```
2  public class AddLongFieldsToPOLinesDataModel extends
       McAbstractLongTextFieldExtendedDataModel {
3    private static final MiKey INSTANCE_KEY = key("InstanceKey");
4    private static final MiKey LONG_REMARKS = key("LongRemarks");
5    private static final MiKey SUPP_TEXT10 = key("
       SupplementaryText10");
6
7    public AddLongFieldsToPOLinesDataModel(final MiResources
       resources) {
8      super(resources);
9    }
10
11   private static final McTextContentMapper TEXT_CONTENT_MAPPER =
12       McTextContentMapperBuilder.create(INSTANCE_KEY)
13         .textValueFieldFromTextKeyField(LONG_REMARKS,
14                                         SUPP_TEXT10,
15                                         Terms.longRemarks(),
16                                         MeOpenness.OPEN_UDPATE)
17         .build();
```



```
18
19    /** {@inheritDoc} */
20    @Override
21    protected McTextContentMapper defineTextContentMapping(final
          MiDefine containerRunner) {
22      return TEXT_CONTENT_MAPPER;
23    }
```

Listing 7.11 shows the implementation of a data-model that adds a single long-text field to the table pane of the `maconomy:PurchaseOrders` container. Since the class extends `McAbstractLongTextFieldExtendedDataModel`, only one method needs to be implemented: `defineTextContentMapping`. This is done in line 21. This method merely returns a statically declared constants which is defined in lines 11–17. Notice that the specified "key fields" are not the actual formal key fields (since these are not invariant,) but instead the `InstanceKey` field. In line 16, it is specified that the added long-text field is only open in update (i.e., *not* in create.)

7.8 Applying a Codec to a Persistence Strategy

When introducing custom popup types—and potentially in other cases—it is necessary to store some field values using a representation which is different from the value represented internally in the container. In this case you may apply a *codec*. A codec is a mechanism that is enable to encode and decode data.

Using the class `McPersistenceStrategyUtil`, it is easy to transform an arbitrary existing persistence strategy into a similar persistence strategy with a codec. This utility class offers the following two factory methods:

Method	Remarks
`applyCodec`	This method takes as arguments a persistence strategy and a codec. It returns a new persistence strategy which is in principle identical to the one given as argument, with the exception that all data being stored is encoded before it is stored, and decoded when fetched. In this way, it is transparent for the user of that persistence strategy whether or not encoding/decoding takes place.
`applyCodecAutoPositionable`	This method is similar to `applyCodec` except that it takes and produces a `MiAutoPositionPersistenceStrategy`.

The codec must be an instance of `MiPersistenceStrategy`.`MiCodec`. When implementing this interface, there are two methods to consider:

Method	Remarks
`encode`	The purpose of this method is to generate a `MiDataValues` object from a `MiValueInspector`. Usually, you will do that by copying the contents of the value inspector as a `MiDataValues` structure, using the `copyValues` method. The values which must are represented differently when persisted must be changed into that value by this method. For example, encoding a custom popup value may encode a `McPopupDataValue` as an `McIntegerDataValue` by using the ordinal value.
`decode`	This method is to opposite of the `encode` method above. Given a `MiValueInspector` you must produce a corresponding `MiDataValues` structure which represents the similar, but decoded, values. Hence the returned object must contain the values as represented by the container. For example, decoding a custom popup value which is encoded by using its ordinal value as a `McIntegerDataValue`, will be decoded by converting that ordinal value into the corresponding `McPopupDataValue`.

Listing 7.12 shows an example of using a codec with a persistence strategy. In the example, which shows a complete data model adding two fields, one of the fields, `PhoneType` is being encoded/decoded using a codec. The type of the field is the custom popup shown in Listing 7.8 above. Since it is a custom popup, we must use a codec. The persistence strategy is created in the `definePersistenceStrategy` method. It is based on a normal MOL-based persistence strategy, and then a codec is applied in line 31. The codec is implemented by the private class in lines 39–56. The `encode` method takes a copy of whatever must be encoded, and overwrites the value of the field `PhoneType` in lines 44–45. Similarly, the value is decoded by copying the input values and overwriting the value of the field `PhoneType` with the custom popup value corresponding to the ordinal. This is done in lines 52–53. The decoding uses an unmodifiable list defined in the factory class declaring the existence of the type `trifolium:PhoneType`.

Listing 7.12: Using a Codec to Store Custom Popup Values

```
2  public class AddPhoneDataModel extends
       McAbstractPersistingExtendedDataModel {
3
4    private static final MiKey NS = key("Trifolium");
5    private static final MiKey FIELD_PHONE_TYPE =
6        NS.concat(":PhoneType");
7    private static final MiKey FIELD_PHONE_VENDOR =
8        NS.concat(":PhoneVendor");
9    public AddPhoneDataModel(final MiResources resources) {
10     super(resources);
```



```
11    }
12
13    /** {@inheritDoc} */
14    @Override
15    public MiKey defineNamespace() {
16      return NS;
17    }
18
19    @Override
20    public MiExtended defineDomesticSpec(final MiDefine
          containerRunner) throws Exception {
21      return McPaneSpec.McExtended.pane()
22        .addPopupField(FIELD_PHONE_TYPE, "Phone Type", key("
              trifolium:PhoneType")).open().then()
23        .addStringField(FIELD_PHONE_VENDOR, "Phone Vendor").open()
24        .end();
25    }
26
27    @Override
28    public MiPersistenceStrategy definePersistenceStrategy(final
          MiContainerRunner.MiDefine containerRunner) {
29      final MiPersistenceStrategy basePersistenceStrategy =
            McMolPersistenceStrategy.create(key("TRI_PhoneData"),
            getApiProvider());
30      return McPersistenceStrategyUtil
31              .applyCodec(new PopupCodec(),
32                          basePersistenceStrategy);
33    }
34
35    /**
36     * Codec class to encode/decode the field PhoneType which is of
37     * type trifolium:PhoneType
38     */
39    private static final class PopupCodec implements
          MiPersistenceStrategy.MiCodec {
40
41      @Override
42      public MiDataValues encode(final MiValueInspector
            nonEncodedValues) {
43        return nonEncodedValues.copyValues()
44          .setInt(FIELD_PHONE_TYPE,
45                  nonEncodedValues.getPopupOrdinal(FIELD_PHONE_TYPE
                        ));
46      }
47
48      @Override
49      public MiDataValues decode(final MiValueInspector
            encodedValues) {
50        final MiDataValues decodedValues = encodedValues.copyValues
```

```
            ();
51          final int ordinal = encodedValues.getInt(FIELD_PHONE_TYPE);
52          decodedValues.setPopup(FIELD_PHONE_TYPE,
53                                 PhonePopupType.PHONE_TYPE_VALUES.get(
                                       ordinal + 1));
54          return decodedValues;
55        }
56    }
```

7.9 The Transformation “Event”

Sometimes, you may wish to significantly change the perception of data in a given pane. To support this, the Extension Framework has a notion of a *transformation “event.”* This takes place for all data-carrying events, just before the container-contribution is about to return the container value. This event has been used to emulate long text editing in containers such as `maconomyLongText:InvoiceEditing` and `maconomyLongText:QuoteEditing`. These containers present the records of the containers `maconomy:InvoiceEditing` and `maconomy:QuoteEditing` so that “text rows” are shown as though the text was part of the “logical line” to which the text belongs. For example, suppose that, in `maconomy:InvoiceEditing`, the table part comprise the following lines:

Line No.	Text	Billing Price
1	This is a line	2000.00
2	having text that	
3	spans multiple	
4	lines.	
5	This is another	1000.00
6	line.	

Hence, the invoice comprises six records. The records 1–4 belong together as one “logical line,” as does the lines 5–6. By implementing the transformation event, `onTransformPane`, it is possible to completely change the contents of the pane, as visualized against the client-side/end-user. In the example above, what we wish to present to the end user is a data-set comprising of *two* records:

Line No.	Text	Billing Price
1	This is a line having text that spans multiple lines.	2000.00
2	This is another line.	1000.00

Hence, two lines where the "Text" is long. The end-user should be able to edit the long text. The internal representation of eventually splitting this text into a number of records in the database is an underlying implementation detail that shouldn't bother the end-user.

If you do implement the `onTransformPane` event, you have access to a result object, which you access through the `getResult` of the `eventData`. This method gives you access to an object of type `MiPaneTransformation`. Such a pane transformation can either be an "no transformation" value. This is the default, and it means that the pane content is not being transformed. Or, it can comprise a number of records; those records must equal the records to be found in the transformed pane.

It must be noted, that transforming a pane is a very low-level and highly advanced thing to do. It should only be performed by programmers having a through understanding of the pane-content structures and how the container mechanisms work. For example, if you change the number of rows in a pane (as in the `maconomyLongText:InvoiceEditing` example), the usual data-carrying events (except for `Read`, and possibly `Initialize`) can no longer be allowed to progress past your contribution! There are two reasons for this:

1. The row-index will likely be wrong

2. The data seen by the client-side is different from the data found in the database, so a "data changed by another user" will likely occur!

Therefor, if you implement the `onTransformPane` for panes that are not read-only, you must `skip` all events in the "pre"-phase, and do something else using a container executor.

If you are considering implementing the `onTransformPane` method, it is strongly advised that you have a look at the abstract class `McAbstractEmulatedLongTextTableDataModel` which is an abstract implementation of a data-model that can be used to collate "text records" with "logical records."

For read-only panes, it is easier to implement the transformation event, because you don't need `skip` and figure out how to transform events into something that can be applied to the underlying "pure" implementation.

Other use-cases of the transformation event could include applying some kind of filtering of the data shown in a given pane. Due to the increased complexity and decreased performance that will follow from utilizing the transformation event, you should only use it reluctantly, and only when you are very confident about what you are doing.

7.10 Supporting Multiple Languages

Maconomy supports multiple languages. Your extension should also be made so that multiple languages are supported. This may be needed for installations targeting users with multiple language preferences. Also, if your extension is "generic" in the sense that it could be used for several installations, your extension *must* support multiple languages.

First, let us have a look at the type of things your extension can do involving multiple languages

- Adding new fields or variables: the *title* needs to be adjusted to the language in question. This also applies when changing the title using `changeField` or `changeVariable`.
- Adding new actions: the *title* needs to be adjusted to the language in question. This also applies when changing the title using `changeAction`.
- Adding new foreign keys or search keys: the *title* needs to be adjusted to the language in question. This also applies when changing the title using `changeForeignKey` or `changeSearchKey`.
- When an *error message*, *warning message* or *notification message* is given to the user, the message needs to be adjusted to the language in question.
- When a *progress bar* is started or updated, the descriptive message needs to be adjusted to the language in question.

It is important to notice, that the extension programmer should never care *which* language or dialect is used by the end-user. For this reason, all terms that need to be localized[2] should always be provided in a *common source language*. By convention, this common source language is (US) English. In the dictionaries supplied for a Maconomy installation, the source language is that found in the left-hand column.

7.10.1 Specifying a Localized Term

Whenever you need to provide a term that needs to be localized, you must specify *a reference* to the term, rather than the actual term/text. The term-references are identified by a key. A special file, called a *.properties*-file must specify the phrasing of the term in the source language. On run-time, the extension framework will ensure that the terms are automatically localized into the language chosen by the user by consulting the installed dictionaries. Obviously, if a term is missing in a given dictionary, the term cannot be localized, and will be left untouched. An example .properties-file could contain the following:

[2] I.e., translated into the language chosen by the end-user

Listing 7.13: Declaring Terms in a `.properties` File.

```
2 # This file contains term definitions for use with the
3 # price-adjustment extension
4 #
5 AdjustPricesAction = Adjust Prices
6 AjdustmentPercentageField = Adjustment %
7 CostPriceCannotBeNegative = The cost price cannot be negative
```

The file in Listing 7.13 declares three different terms:

`AdjustPricesAction` defined in line 5 is the key to use to reference the term which—in English–would read "Adjust Prices." There is no particular requirement to the format of the key, but it is usually some sort of readable but abbreviated form of the actual term. In this case, it also explains to the programmer that it is indented as an action title.

`AdjustmentPercentageField` is the key to use to reference the term which in English would read "Adjustment %"

`CostPriceCannotBeNegative` is the key to use to reference the term which in English would read "The cost price cannot be negative"

Notice that the keys are readable (and typically "short") but *not* suited to be shown to an end-user. For example, they don't contain spaces. Whereas the *actual term* is written in natural language (suitable for it's intended scope) and may contain spaces, special characters, common abbreviations etc. Also notice that you may specify comments by starting the line with a hash mark character (`#`)[3].

7.10.2 Referencing a Term

Defining the terms is not enough. Obviously, as an extension programmer, you need to specify *when* a given term is used. You can do that in a number of ways:

Invoking a text factory By invoking the `term` method of text *factory* implementing the `MiTextFactory`.`MiLocalize` interface. You can construct such an object my invoking the `bundle`-factory method in the class `McTextFactory`. By doing that, you instruct the framework about which OSGi-bundle is hosting the properties file. In that bundle, the framework will look for a `.properties`-file called `Messages.properties` in the package called `messages`. Notice, that this package, hence, must be a top-level package. I.e, *not* `com.trifolium.messages` or `my.package.prefix.messages`! The `messages` package should *not* be exported by the OSGi-bundle in question! An example showing how to use the text-factory is:

```
MiTextFactory.MiLocalize tf =
  McTextFactory.bundle("com.myorg.adjustpr");
```

[3]The hash-mark character must be the first character of the line

```
MiText localizedTerm = tf.term("CostPriceCannotBeNegative");
```

As we shall see in Section 7.10.3, there is an even more elegant way to obtain a text factory. Using a text factory is the recommended way of obtaining localized terms.

Specifying a bundle and key using `McText` The `McText` class is normally used to provide *non-localized and direct* texts using the `text`-method. However, by invoking the `term` method you can reference a term instead. Doing so, you *must* specify the id of the bundle hosing the corresponding `Messages.properties` file (in the `messages` package.) You do that by providing a String that contains the bundle-id, followed by a colon ":" followed by the term-key. For example

```
MiText localizedTerm =
McText.term("com.myorg.adjustpr:CostPriceCannotBeNegative");
```

The downside of this is that it is somewhat cumbersome to read, write and maintain.

Inlining in term Rather than providing the term directly you can specify the actual term-phrase (in English). This is done by invoking the method `termInline` of the `McText` class[4].

```
MiText localizedTerm =
  McText.termInline("The cost price cannot be negative");
```

The down-side of this way of obtaining a localized term is that there is no easy way to identify all the terms that must be added to the dictionary (in case they are not already present.)

7.10.3 Using Text Factories

As mentioned above, using a text-factory is the recommended way of referencing dynamically localized terms. This is mainly due to the following reasons:

1. Term-references are easy to read and write
2. All terms that are needed in the dictionaries are found in *one* common and standardized place: this standardized place is the `Messages.properties` file, and the terms that needs to be present in the dictionaries are found as the right-hand side of `=` in the term definitions.

At this stage, you may feel that using the text-factory seems more cumbersome because you have to declare it, and you have to deal with the bundle id. Fortunately, these reasons are very weak. First, the text factory can be declared as a `final static` constant in your class. Second, the text-factory can be declared with an *indirect* bundle-reference. By providing a class implementing the interface `MiBundleText` (which is just a marker

[4]This will be supported from version 15.0sp8

interface[5]), the Extension Framework will assume the bundle in which that provided class is declared. Hence, you can do the following:

- Declare a class inside your bundle that implements `MiBundleText`. For example, you can define a singleton `enum` class, thereby giving you easy access to an object instance.
- In your data-model class declare the a `final static` constant text factory by providing the singleton enum as input to the `bundle` method of the `McTextFactory` class.
- Everywhere you need a term, obtain it through that text-factory constant.

Let us have a look at an example. First we need to declare an `enum` class with one element, letting it implement `MiBundleText`.

```
1 public enum Terms implements MiBundleText {
2   /** Singleton class */
3   INSTANCE;
4 }
```

Next the data-model class (or whatever class needs to reference terms) declares a constant:

Listing 7.14: Referencing Terms from Java Code.

```
2   private static final MiKey NS = key("Trifolium");
3   private static final MiKey ACTION_ADJUST_PRICES =
4       NS.concat(":AdjustPrices");
5   private static final MiKey FIELD_ADJUSTMENT_PCT =
6       NS.concat(":AdjustmentPercentage");
7   private static final MiKey COST_PRICE = key("CostPrice");
8
9   private static final MiTextFactory.MiLocalize tf =
10      McTextFactory.bundle(Terms.INSTANCE);
11
12  /** {@inheritDoc} */
13  @Override
14  public MiPaneSpec.MiExtended defineDomesticSpec(final MiDefine
        containerRunner) {
15    return McPaneSpec.McExtended.pane()
16      .addAction(ACTION_ADJUST_PRICES,
17                 tf.term("AdjustPricesAction"))
18                .then()
19      .addRealField(FIELD_ADJUSTMENT_PCT,
20                    tf.term("AjdustmentPercentageField"))
21                   .open()
22                   .then()
23      .end();
```

[5]I.e., an interface with no requirements to implement any specific methods

```
24    }
25
26    /** {@inheritDoc} */
27    @Override
28    public void onChangePre(
29        final MiContainerRunner.MiChangePre containerRunner,
30        final MiEventData.MiUserChange eventData) throws Exception {
31      final MiUserData userData = eventData.getUserData();
32      containerRunner.check(userData.unchanged(COST_PRICE)
33                            || userData.getAmount(COST_PRICE).signum
                                 () >= 0)
34                    .error(tf.term("CostPriceCannotBeNegative"));
35    }
```

Listing 7.14 shows how terms are referenced from Java code. The example code shows part of a data-model implementation. In lines 9–10 a `private static` text-factory constant is declared. This text-factory is then used to reference the terms in multiple locations in the code. In line 17 the term `AdjustPricesAction` is referenced to give a title for an added action. In line 20 the term `AjdustmentPercentageField` is referenced to produce a title for an added field. Finally, in line 34 the term `CostPriceCannotBeNegative` is referenced to produce an error message. Depending on the selected language, the end-user would experience the following[6]:

Language	*Action Title*	*Field Title*	*Error message*
English	Adjust Prices	Adjustment %	The cost price cannot be negative
French	Ajuster les prix	% d'ajustement	Le coût de revient ne peut pas être negatif
Danish	Justér priser	Justrings%	Kostprisen må ikke være negativ
Swedish	Justera priser	Justering %	Kostpriset kan inte vara negativ
Dutch	Prijzen aanpassen	Aanpassings-%	De kostprijs kan niet negatief zeijn

Hence, the language presented to the end-user is automatically adjusted without the extension programmer knowing or caring about it. The only prerequisite is that the extension programmer informs the framework that a named term is intended, rather than specifying a static text (using the `McText.text()`-method.)

The observant reader will notice that there is still a downside: the reference to the terms may easily contain typos, leading to unresolved terms at run-time. For this reason, it is recommended that you implement a class that has a static method for each term. This class is then referenced everywhere you need that specific term. In this way, you will not only get content-assistence from the IDE while coding, the compiler will complain if you reference a non-existing term!

[6] Assuming that corresponding translations are defined in the installed dictionaries

A good candidate class for containing such `static` term-methods is the singleton `enum` class introduced above. Also, this class could be placed in the `messages` package that also contains the `Messages.properties` file. If the terms should be made available to several bundles, this class should be placed in a package that is exported by the OSGi-bundle. And the `messages` package should *not* be exported!

Let us have a look at how the `enum` class could be implemented provided that it contains methods to obtain the localized terms.

Listing 7.15: Implementing Terms as Utility-Methods.

```
2  public enum Terms implements MiBundleText {
3    /** Singleton instance */
4    INSTANCE;
5
6    private static final MiTextFactory.MiLocalize tf = McTextFactory
         .bundle(INSTANCE);
7
8    public static MiText adjustPricesAction() {
9      return tf.term("AdjustPricesAction");
10   }
11
12   public static MiText ajdustmentPercentageField() {
13     return tf.term("AjdustmentPercentageField");
14   }
15
16   public static MiText costPriceCannotBeNegative() {
17     return tf.term("CostPriceCannotBeNegative");
18   }
```

Listing 7.15 shows a number of `static` methods defined in the `Terms` class. In line 6 the text-factory is declared locally as a private constant. This text-factory is then used by each term-defining method to obtain a specific term (lines 9, 13 and 17.) By convention, the method name resembles the internal name of the term-id, except that it begins with a lower-case letter in order to adhere to standard Java naming conventions.

In this way, there is only *one* place to reference the term key specified in the `Messages.properties` file. Using these term-methods, the data-model from above would now look like:

Listing 7.16: Referencing Terms from Java Code Using `static` Methods.

```
2    private static final MiKey NS = key("Trifolium");
3    private static final MiKey ACTION_ADJUST_PRICES =
4        NS.concat(":AdjustPrices");
5    private static final MiKey FIELD_ADJUSTMENT_PCT =
6        NS.concat(":AdjustmentPercentage");
7    private static final MiKey COST_PRICE = key("CostPrice");
8
9    /** {@inheritDoc} */
```

```
10    @Override
11    public MiPaneSpec.MiExtended defineDomesticSpec(final MiDefine
          containerRunner) {
12      return McPaneSpec.McExtended.pane()
13        .addAction(ACTION_ADJUST_PRICES,
14                   Terms.adjustPricesAction())
15                  .then()
16        .addRealField(FIELD_ADJUSTMENT_PCT,
17                      Terms.ajdustmentPercentageField())
18                     .open()
19                     .then()
20        .end();
21    }
22
23    /** {@inheritDoc} */
24    @Override
25    public void onChangePre(
26        final MiContainerRunner.MiChangePre containerRunner,
27        final MiEventData.MiUserChange eventData) throws Exception {
28      final MiUserData userData = eventData.getUserData();
29      containerRunner.check(userData.unchanged(COST_PRICE)
30                            || userData.getAmount(COST_PRICE).signum
                                 () >= 0)
31                     .error(Terms.costPriceCannotBeNegative());
32    }
```

The code in Listing 7.16 is functionally equivalent to that from Listing 7.14. The only difference is that the terms are obtained using the `static` term-defining methods, as it can be seen in lines 14, 17 and 31.

7.10.4 Handling Terms with Variable Content

Sometimes, the messages you want to display to an end-user has content that may vary depending on the given situation. For example, suppose you have an extension giving an error if the user enters an amount in a field, and the specified amount exceeds another amount specified on the customer information card. In such a case, you could give a generic message such as "The entered amount exceeds the maximum specified for the customer." However, the usability would probably be higher if the user had a little more context such as: "The entered amount of 10.000 exceeds the maximum of 8.000 specified for customer 665-8827." We obviously don't want to provide static terms for all possible combinations. Instead, we can use *placeholders* in the term definition. Placeholders are specified with the following syntax: ^*digit*.

Listing 7.17: Declaring Terms with Placeholders

```
2  # This file contains term definitions for use with the
3  # price-adjustment extension
```



```
4 #
5 AdjustPricesAction = Adjust Prices
6 AjdustmentPercentageField = Adjustment %
7 CostPriceCannotBeNegative = The cost price cannot be negative
8
9 AmountExceeded = The entered amount of ^1 exceeds the maximum of
    ^2 for customer ^3
```

Listing 7.17 shows an example of a `Messages.properties` file defining a term with placeholders. This is done in line 9. Since the placeholders are enumerated, it is possible to define a translation that puts the placeholders in a different position. Switching the placeholders may be required by the grammar of certain languages. Luckily, this is of no concern to the extension programmer.

When a term is defined with placeholders, it means that representatives for each placeholder must be provided when the term is instantiated. This means that the term `AmountExceeded` expects three arguments, since there are three placeholders. The actual value of a specific invocation of the term will determine the exact content that is eventually presented to the user. In order to provide arguments for a term, you can use the overloaded version of the `term` method that takes an arbitrary number of additional `String` arguments.

Listing 7.18: Referencing Terms With Placeholders.

```
2     final String customerNumber =
3         originalData.getStr(CUSTOMER_NUMBER);
4     final BigDecimal enteredAmount =
5         userData.getAmount(AMOUNT_FIELD);
6     final BigDecimal customerLimit =
7         getCustomerLimit(customerNumber);
8     containerRunner.check(
9         enteredAmount.compareTo(customerLimit) > 0)
10      .error(tf.term("AmountExceeded",
11                     enteredAmount.toPlainString(),
12                     customerLimit.toPlainString(),
13                     customerNumber));
```

Listing 7.18 shows how a term with placeholders is obtained. First some values are retrieved, and then used in lines 10–13. Notice that all placeholder-values must be converted into `String` objects.

In this case, introducing methods that generate the actual localized terms (as suggested in Listing 7.15) becomes even more desirable. In Listing 7.18 above, there's nothing preventing the programmer from providing arbitrary `String`-typed arguments, even though, the intention is clearly that two of the parameters are amount-typed values. Arguments of the correct type can be ensured by introducing a `static` method to obtain the localized term. We can add the following to the `Term` class outlined in Listing 7.15.

Listing 7.19: Implementing Placeholder Terms with Type-Safe Arguments.

```
2   /**
3    * Returns a localized message indicating that an entered
4    * amount exceeds the limit specified for a specific customer.
5    * @param enteredAmount the amount entered by the user.
6    * @param customerLimit the limit applicable for the customer
7    * @param customerNumber the customer number
8    * @return a localized text with the specific amount and
9    * customer references inserted.
10   */
11  public static MiText amountExceeded(
12      final McAmountDataValue enteredAmount,
13      final McAmountDataValue customerLimit,
14      final McStringDataValue customerNumber) {
15    return tf.term("AmountExceeded",
16                   McAmount.of(enteredAmount).toPlainString(),
17                   McAmount.of(customerLimit).toPlainString(),
18                   McStr.of(customerNumber));
19  }
20 }
```

Using the `static` method defined in Listing 7.19, you are *enforced* to provide arguments of the correct type when instantiating the term, and the term-method is now in control of how, e.g., amount values are converted into a `String`s. The extension code obtaining the term would look like:

Listing 7.20: Referencing Terms from With Placeholders.

```
2     final McStringDataValue customerNumber =
3         originalData.getStrVal(CUSTOMER_NUMBER);
4     final McAmountDataValue enteredAmount =
5         userData.getAmountVal(AMOUNT_FIELD);
6     final McAmountDataValue customerLimit =
7         getCustomerLimit(customerNumber);
8     containerRunner.check(
9         enteredAmount.compareTo(customerLimit) > 0)
10      .error(Terms.amountExceeded(enteredAmount,
11                                  customerLimit,
12                                  customerNumber));
```

Listing 7.20 shows how a method defining a term with typed placeholders is invoked from extension code. Notice that not only will you get assistance by the IDE (ensuring the correct number of arguments, and showing the documentation for each placeholder,) you would also get a compile-error if you provide arguments of incorrect types, e.g.

```
//...
containerRunner.check(enteredAmount.compareTo(customerLimit) > 0))
   //! Gives a compile-time error!
  .error(Terms.amountExceeded(customerNumber,
```



```
                                    enteredAmount,
                                    customerLimit);
```

7.10.5 Annotating Terms with Comments

Sometimes you might want to provide some kind of context to the terms you define. Basically, this is needed to help whoever makes dictionary translations into knowing the context in order to provide a suitable translation. As an example, suppose you introduce an action used to close a financial year in some context. If you give this action the title "Close," the person responsible for translating this into a given language gets no more information than the word "Close." If you're lucky, the translator will make the correct translation for the context. If you're unlucky, the translation will be made from 'Close' as in 'Nearby.' Or you maybe another term 'Close' was previously introduced and translated different from what you expect.

In order to get around such unfortunate situations, Maconomy supports the concept of *comments* inside your terms. A comment has the following syntax:

{ *coment text* }

Hence, text inside curly braces. At the moment the curly braces cannot be escaped. Therefor, it is impossible to present a message containing curly braces.

Listing 7.21: Annotating Terms with Comments

```
2  # This file contains term definitions for use with the
3  # price-adjustment extension
4  #
5  AdjustPricesAction = Adjust Prices
6  AjdustmentPercentageField = Adjustment %
7  CostPriceCannotBeNegative = The cost price cannot be negative
8
9  AmountExceeded = The entered amount of ^1 exceeds the maximum of
       ^2 for customer ^3
10
11 CloseAction = Close{Financial Year}
```

Listing 7.21 shows an example `Messages.properties` file that contains a term which is annotated by a comment in line 11. The comment will *not* be shown to the end-user, but it *will* be shown to whoever needs to make a translation. Also, when the term is looked up in a given dictionary, the comment is part of the entry being looked up. If another term "Close" (without comments or with some other comment) is also defined in the dictionary, that term will *not* match. Unless it is absolutely clear what is meant, comments should be used in order to ensure good-quality translations.

You can have as many comments as you want, and they can be placed anywhere. For example

```
The job is closed{No entries allowed} for ^1{User Name}
```

Here a term is defined and two comments are defined. The first comment gives context to what it means that the "job is closed", and the second gives context to the placeholder (indicating that it is expected to be a reference to a user name.)

7.10.6 Locale Annotations

It is possible to specify terms that are locked to a specific language. By having a specific kind of comment *at the very end* of the text, specifying a specific locale in square brackets, means that the term is defined in the specified language. At the moment, there is no way of translating terms from arbitrary languages, so doing that will imply that localization will not take place for the specified term. Hence specifying:

```
MyDanishTerm = Check CVR Register{[da_DK]}
```

means that the term `MyDanishTerm` is defined in Danish and will never be translated to another language. Hence, even if the end-user has specified French as a preferred language, the term above will be shown in Danish. You can specify the default locale at the top of a `Messages.properties` file, by specifying a value for `Locale@locale` at the top of the properties-file[7]. This will make the framework interpret all terms as defined in that language, except if something else is explicitly specified. For example

```
Locale@locale={[da_DK]}
MyDanishTerm = Check CVR Register

MySwedishTerm = Kolla SJ tidtabell{[sv_SE]}

MyTranslatableTerm = This term can be translated{[]}

MyEnglishTerm = This term is always in American-English{[en_US]}
```

In this example, it is stated that all terms by default are specified with Danish locale. Hence, the term `MyDanishTerm` does not explicitly need to specify a locale. Since it's locale is fixed, it will never be translated. The term `MySwedishTerm` specifies that the corresponding term is in Swedish. Since it has a fixed locale, it will never be translated. The term `MyTranslatableTerm` is explicitly annotated with an empty locale. An empty locale corresponds to the common source language. This is interpreted by the Extension Framework such that it *will* be looked up in the dictionaries. It is consequently expected that the term "This term can be translated" is found in the dictionaries on run-time. The term `MyEnglishTerm` has a fixed locale of American-English, and will therefore not be localized.

Except if you have very explicit needs, we recommend that you define terms without locale, thus letting your terms be translated to any language.

[7] If no such locale annotation is specified, the default locale will be the common source language

If you specify a term in a specific language, it is highly recommended that you also specify the locale. If you don't, unexpected results may occur! For example, suppose you are defining the term "Save." But since you (think you) know that the client you're working for only needs to have the terms presented in Danish, you opt to *not* specify the term in English and expanding the dictionary. So, your term is just specified as:

```
MyTerm = Gem
```

In this case, no locale is specified, altough the term was intended to be specified in Danish[8]. This means that the term *will* be attempted localized, and terms *may* be extracted as input to the dictionaries. Hence, the Danish-speaking user would suddenly not see the expected term "Gem" whenever the save-context should appear. Instead, he'd see "Ædelsten" (i.e., "precious stone,") since that is the correct Danish translation of the English word "Gem." Highly unexpected, and highly confusing. This would *not* have happened, had the term been defined with the proper locale:

```
MyTerm = Gem{[da_DK]}
```

Of course, if the term had been specified *without* locale as `Save`, the term would be translated into Danish (or whatever other language one of the end-users one day opts to use—provided that the dictionaries are properly defined!)

7.11 Enabling Logging

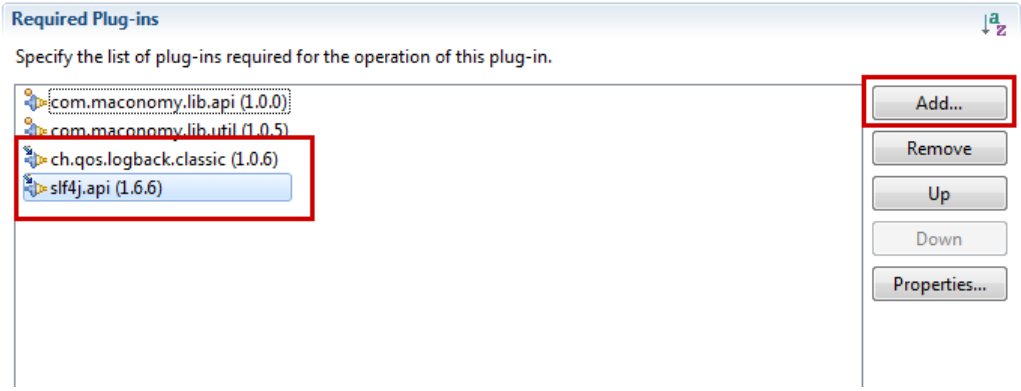
Often when developing software, it is practical to have the ability to log some output. This can happen for a number of reasons, typically for debugging/diagnostic purposes on running installations.

The coupling service is pre-configured with the ability to do logging. The underlying logging frameworks that are used are: "Logback" [Log] and "SLF4J" [Slf].

By using these logging frameworks, logging output from your extension code can be managed in the same way as logging output from other parts of the coupling service. What to log, how to log and which log-levels are wanted can be configured by editing the file `logback.xml` which resides in the `configurations` folder inside the folder in which the coupling service is installed.

In order to be able to access the logging frameworks, you need to instruct the OSGi engine that you depend on this. This is done by opening the `MANIFEST.MF` file of your project, selecting the "Dependencies" tab, and the declare a dependency to the slf4j and the logback bundles:

[8] "Gem" in Danish means "Save"



Once this is done, you can start adding a logger to your classes. The recommended way of doing this is to declare a `static` constant, by convention called `logger`, which is named in the same way as your class (including package prefix!)

In order to do this, you need to import the following in your class (or let the Extender IDE help you)

- `org.slf4j.Logger`
- `org.slf4j.LoggerFactory`

Listing 7.22: Enabling Logging Output.

```
2  public class LoggingExampleDataModel extends
       McAbstractExtendedDataModel {
3    // Declare a logger specific to this class
4    private static final Logger logger = LoggerFactory.getLogger(
       LoggingExampleDataModel.class);
5
6    private static final MiKey BLOCKED_FOR_INVOICING = key("
       BlockedForInvoicing");
7
8    public LoggingExampleDataModel(final MiResources resources) {
9      super(resources);
10   }
11
12
13   /** {@inheritDoc} */
14   @Override
15   public void onUpdatePre(final MiUpdatePre containerRunner,
16                           final MiUpdate eventData) throws
                                Exception {
17     final MiUserData userData = eventData.getUserData();
18     final String currentUser = containerRunner.getEnvironmentInfo
         ().getUserName();
19     if (logger.isDebugEnabled()) logger.debug("Log a debug message
         ");
```



```
20
21      if (userData.changed(BLOCKED_FOR_INVOICING)) {
22        if (logger.isDebugEnabled()) {
23          logger.debug(
24            "Blocked inv. status for job: {} set by user {}",
25            userData.getBool(BLOCKED_FOR_INVOICING),
26            currentUser);
27        }
28        // do your own logic
29      }
30      // If you need to output more than 2 pieces of information
31      // you must make an Object-array containing the
32      // elements
33      if (logger.isDebugEnabled()) {
34        final Object[] info =
35            new Object[] {currentUser,
36                          Calendar.getInstance().getTime(),
37                          userData.getStr("JobNumber"),
38                          userData.getUserChange()};
39        logger.debug("Fields changed by user {} on {} for job {}: {}
              ", info);
40      }
41    }
```

Listing 7.22 shows how to enable logging output. The first thing you should note is the declaration of the static `logger` in line 4. This logger can now be used throughout the code. If logging has been enabled for that specific logger for an adequate log-level, logging will occur once one of the methods `info`, `warn`, `error`, `debug` and `trace` is invoked. If the corresponding logger and log-level is not activated (through the logging configuration), then nothing is done. You should be careful to guard your logging statements with the corresponding log-level-enabled methods, like `isDebugEnabled`. The reason is that if any of the arguments to the logger are non-trivial object-references, then the code evaluating the logging content will be executed *even if logging is not enabled*. For this reason, it is good practice to always guard the logging statements. In the example, a simple static text is logged in line 19 (but again only if debug log-level is enabled for the current logger at run-time.) In lines 24–26 a logging statement is potentially made with *variable logging content*. Each `{}` will denote a placeholder. The first argument object will be inserted at that place if logging does occur. Hence, you *should not* construct your logging messages using the `+` operator for `String`s. If you need more than one or two variables inserted, you must provide the information objects as an `Object`-array. This is shown in lines 35–38, and the logging statement is done in line 39. The exact format of the logging contents will depend on the contents of the `logback.xml` file at run-time. It is possible to edit the configuration file while running, and the log output will adapt almost immediately.

For more details on how to manage and configure logging, you are left to the documentation

[Log, Slf].

7.12 Miscellaneous Utilities

This section contains references to miscellaneous utilities that are not mentioned elsewhere in this book.

7.12.1 Long-Text Splitting

In order to work with strings that exceed the 255 byte limit offered by the Maconomy database, you may sometimes need to split a string into smaller chunks. If you eventually want to print these strings using MPL, you may want to cut a string into chunks that are restricted not only by the number of bytes, but also by the width the text will have when printed. Of course, this width will depend on the font properties being used.

The framework contains a utility class that can be used for this: `McSplitTextUtil`. This class contains a single method, `splitText` that—given a `String` and a specification of font-properties and the desired maximum width, will split the text into chucks that:

1. Will not exceed the specified width, if printed using the specified font characteristics
2. Will not exceed 255 bytes
3. Will attempt to break the string at a space or newline
4. Will contain a marker, if the string was split in mid-word. For this the constant `McSplitTextUtil.APPEND_USING_NO_SEPARATOR`[9] is used.
5. Will contain a marker, if the text was broken at a space (rather than at a newline). For this the the constant `McSplitTextUtil.APPEND_USING_WORD_SEPARATOR`[10] is used.

You are referred to the JavaDoc visible inside the Maconomy Extender IDE for more information.

[9] Currently represented by the invisible unicdoe character `\uFEFF`
[10] Currently represented by the invisible unicode character `\u200B`

Chapter 8

Tips and Tricks

This chapter gives various tips and tricks about performing certain tasks in ways that may not be obvious.

8.1 Filter Containers Based on Custom Universes

A thing that is frequently needed, is building a container that provides access to a given universe through a filter pane.

Prior to version 17 (2.2), the way to do this was to do a number of steps:

- Create a universe
- Create a new root container contribution that implements the `definePersistenceStrategy` method by returning an instance of `McMolPersistenceStrategy` associating the universe name.
- Let the `defineDomesticSpec` of the filter data-model replicate all the field definitions of the universe

While this is simple to explain, it is a tedious, error-prone and impractical task to do. This is so, because:

- You have to replicate all fields from the universe (the ones you wish to expose), their type and their titles.
- If you extend the universe, you will have to update your data-model with these fields, recompile and deploy.
- All though it's not a lot, you need to actually write the boiler-plate Java code that declares the container and the data model.

In order to ease this kind of task, Maconomy 17 comes with a feature that can almost automatically provide a container having a filter that is targeted at a specific universe. You can get this without writing a single line of Java code. The steps you must do are the following:

1. Create the universe, including an interface that specifies the fields you wish to expose. The interface fields must be "flat," i.e., it is not supported to expose "dotted" field names (such as `ProjectManager.EmployeeNumber`.)
2. Create a *mapping* MOL file that maps to the universe, identifies the key fields and specifies the container name-space

Having done this, you can now operate a container, which has a filter pane that is targeted at the universe.

Example: assume you have a universe called `Trifolium::CustomerU` that exposes information about customers with additional information from a custom MOL file (`Tri_CustomerAddendum`). Among other things, this MOL table contains information about whether or not a given customer is considered a "premium" customer.

Now, we add a MOL-file, called `tri_CustomerU`[1]. This MOL file declares the connection to the `Trifolium::CustomerU` universe, declares that there is one key field (the field `CustomerNumber`) and that the name space for the filter-pane container is to be `Trifolium`. Now, you can invoke the container called `Trifolium:Find_tri_CustomerU` from workspaces, as well as programmatically!

In this example, the MOL-file `tri_CustomerAddendum` is defined like this:

Listing 8.1: Regular MOL file containing additional customer fields.

```
<MOL 1.2>
<Entity name=tri_CustomerAddendum "Customer Addendum"
        ContainerNameSpace=Trifolium>

  .CustomerNumber     :String  :"Cust. No." :Key+
  .PremiumCustomer    :Boolean :"Premium Cust."
  .AgreementId        :String  :"Agreement Id"
  .CreditBalanceLimit :Amount  :"Credit Bal. Limit"
<End Entity>
```

The universe definition could be something like this:

Listing 8.2: Universe that we want to expose through new container.

```
<MUL 1.4>

<universe Trifolium::CustomerU "Customer">

  <object Customer>
  <object CustomerBalance>
```

[1]The name can be freely chosen, although it must contain a MOL name-space, in this case `tri_`.


```
7    <object Tri_CustomerAddendum>
8
9    <join (Customer, CustomerBalance) accessControl = skipKeep>
10     .CustomerNumber
11   <end join>
12
13   <join (Customer, Tri_CustomerAddendum)>
14     .CustomerNumber
15   <end join>
16
17   <field LimitExceeded "Limit Exceeded" >
18     CustomerBalance.CreditBalanceBase > Tri_CustomerAddendum.
           CreditBalanceLimit
19   <end field>
20
21   <field CreditLimitGap "Credit Limit Gap" >
22     Tri_CustomerAddendum.CreditBalanceLimit - CustomerBalance.
           CreditBalanceBase
23   <end field>
24
25   <interface basis=Customer>
26     .DebitBalanceBase       :CustomerBalance.DebitBalanceBase
27     .CreditBalanceBase      :CustomerBalance.CreditBalanceBase
28     .DebitBalanceStandard   :CustomerBalance.DebitBalanceStandard
29     .CreditBalanceStandard  :CustomerBalance.CreditBalanceStandard
30
31     .PremiumCustomer        :Tri_CustomerAddendum.PremiumCustomer
32     .AgreementId            :Tri_CustomerAddendum.AgreementId
33     .CreditBalanceLimit     :Tri_CustomerAddendum.CreditBalanceLimit
34     .LimitExceeded          :.LimitExceeded
35     .CreditLimitGap         :.CreditLimitGap
36   <end interface>
37 <end universe>
```

8.1.1 Mapping MOL Specifications

So, we want a root container contribution that connects to the universe `Trifolium::CustomerU` defined in Listing 8.2. Instead of writing that by hand, we can let the Maconomy server know about (and handle) that container automatically. All we need to do is to specify a *mapping MOL file* that connects the universe to a container. Notice that the universes does not contain any information about key fields, so an important part of this mapping MOL file is to declare how to identify the key (there could be several possibilities for a given universe.) Also, we must specify the name space of the container. This name space *must* be defined in order to avoid name clashing with standard Maconomy containers.

Listing 8.3: Mapping MOL file establishing a link between a universe and a container.

```
1 <MOL 1.2>
2 <Entity name=tri_CustomerU  "Customer"
3             Universe=Trifolium::CustomerU
4             interface=SearchInterface
5             ContainerNamespace=Trifolium>
6
7   .CustomerNumber :Key+
8 <End Entity>
```

Listing 8.3 shows such a mapping MOL file. Notice the MOL version number that must be at least `1.2`. Also notice that in MOL version 1.2, the tag formerly called ~~`<Object>`~~ must now be called `<Entity>`. This subtle change in terminology reflects that the MOL file declares an abstract data *entity* which gives you, more possibilities, such as a filter container.

In line 3 the name of the universe which is being mapped by this MOL file is declared. Furthermore, it must be specified which *interface* of that universe is being exposed by the container. This is done in line 4. At the time of writing, the interfaces referenced in mapping MOL files *must* be flat. That is, the field names must not be "structured" inside sub-names, such as `.CustomerBalance.CreditBalanceBase`. Instead these must be "flattened out" as it has been done in Listing 8.2, lines 26–35. In line 5 the name space of the new container is defined, in this case as `Trifolium`. Finally, the mapping MOL file declares which fields act as key fields of this container. In line 7 it is declared that the container has one key field; the field `CustomerNumber`.

When this MOL file is installed, the Maconomy system will now automatically comprise a container called `Trifolium:Find_tri_CustomerU`. Hence the name of the container is built in the following way:

⟨*container name-space*⟩`:Find_`⟨*name of mapping MOL (incl. MOL name-space)*⟩

The resulting container will, thus, include all fields comprised by the specified interface. In addition, all foreign keys that can be deduced from the definitions of the underlying Maconomy tables will be included, provided that all necessary fields are part of the interface. These foreign keys can be used to bind other panes together with the generated container in workspaces.

8.1.2 Using the Mapping MOL Container

The container resulting from the mapping MOL can now be address in exactly the same way as any other containers. For example, in workspaces.

```
1 <Filter source="Trifolium:Find_tri_CustomerU">
2   <Bind foreignKey="primary">
3     <Card source="CustomerCard">
4   </Bind>
5   <Bind foreignKey="StandardBillingPriceList_JobPriceListInformation">
```



```
6       <Card source="JobPriceLists" title="Std. Price List">
7     </Bind>
8   </Filter>
```

In this workspace specification, the container `Trifolium:Find_tri_CustomerU` is referenced in line 1. In line 2, a binding is made (using the key fields, e.g., the `CustomerNumber`) to the `maconomy:CustomerCard` container. Also, in line 5, a binding is made using one of the inherited foreign keys to the `maconomy:JobPriceLists` container.

Just like you can reference the container from workspaces, you can reference the container programmatically. For example, suppose that you would like to change the search container used when the user searches for customers from the `maconomy:Jobs` container. In order to achieve that, we can change the foreign key used for this purpose (`CustomerNumber_Customer`). All we need to do is to change the search container into `Trifolium:Find_tri_CustomerU`.

Listing 8.4 shows an implementation that does the following:

- Changes the search container used to search for customers. This is done in line 14.
- Restricts the search into only showing "premium customers" in case the responsible department (represented by the Location) is "SPOP" and showing only ordinary (i.e., non-premium) customers for jobs belonging to other departments. The premium field is put into the custom (ordinary) MOL table (Listing 8.1), and exposed through the universe (Listing 8.2.) This takes place in the `onRestrict` method in lines 31 and 33.
- Whenever the customer is changed, it is checked that for SPOP jobs, the new customer is a premium customer, and that the customer is an ordinary customer for non-SPOP jobs. This is done by programmatically accessing the container `Trifolium:Find_tri_CustomerU`. This goes on in the method `onChangePost`. In line 53 a container executor referencing the universe-container is declared. In line 64 the container is read, after having been adequately restricted.

Listing 8.4: Programmatic References to Universe Container

```
2   private static final MiContainerName FIND_TRI_CUST_U =
3       McContainerName.create("Trifolium:Find_Tri_CustomerU");
4   private static final McSimpleExpressionBuilder expr =
       McSimpleExpressionBuilder.expr();
5   private static MiExpression<McBooleanDataValue> PREMIUM =
6       expr.eq("PremiumCustomer").bool(true);
7   private static MiExpression<McBooleanDataValue> ORDINARY =
8       expr.not("PremiumCustomer");
9
10  @Override
11  public MiExtended defineDomesticSpec(final MiDefine
       containerRunner) {
12    return McPaneSpec.McExtended.pane()
```

```
13        .changeForeignKey("CustomerNumber_Customer")
14          .searchContainer(FIND_TRI_CUST_U)
15        .end();
16    }
17
18    @Override
19    public void onRestrict(final MiRestrict containerRunner,
20                           final MiEventData.MiRestrict eventData)
                                {
21      final MiKey fkName = eventData.getForeignKeyName();
22      final MiValueInspector restrictionValues = eventData.
          getRestrictionValues();
23      final MiQueryExpressionAdmission queryExpr =
24          eventData.getQueryExpressionAdmission();
25
26      if (fkName.isLike("CustomerNumber_Customer")) {
27        // Jobs having location SPOP must be linked to
28        // premium customers.
29        // Other jobs must not be linked to premium
30        if (isSpopJob(restrictionValues)) {
31          queryExpr.and(PREMIUM);
32        } else {
33          queryExpr.and(ORDINARY);
34        }
35      }
36    }
37
38    private final boolean isSpopJob(final MiValueInspector job) {
39      final String location = job.getStr("LocationName");
40      return location.equals("SPOP"); // value for Special-Priority
          Operations department
41    }
42
43    @Override
44    public void onChangePost(final MiChangePost containerRunner,
45                             final MiUserChange eventData) throws
                                  Exception {
46      final MiDataValues resultData = eventData.getResultData();
47      final MiUserData userData = eventData.getUserData();
48      if (userData.changed("CustomerNumber")) {
49        // Check if the search container contains a customer
50        // with the specified customer number
51        // and the required premium status
52        final MiContainerExecutor filter =
53          containerRunner.executor(FIND_TRI_CUST_U)
54                         .construct(MePaneType.FILTER);
55        MiExpression<McBooleanDataValue> condition =
56          resultData.copyValues("CustomerNumber").asExpression();
57        if (isSpopJob(resultData)) {
```



```
58          condition = and(condition, PREMIUM);
59        } else {
60          condition = and(condition, ORDINARY);
61        }
62        filter.control().select("CustomerNumber")
63                        .restrictBy(condition)
64                        .read();
65        containerRunner.check(filter.getRowCount() > 0)
66                       .error("No such customer (or customer premium
                              status is not as required)");
67      }
68    }
```

8.1.3 Filter Containers Based on Plain MOL Tables

Sometimes you just want a filter container based on a plain MOL table. Thus, a table that isn't used to map a *universe*, but one that represents a real database table.

It is very easy to get such a container. Basically, all you need to do is to specify the `ContainerNameSpace` attribute in the MOL header. And this will give you a filter container reflecting all fields of the MOL table.

For example, consider the custom MOL-file in Listing 8.1. In that file, a `ContainerNameSpace` attribute is declared (in fact, in MOL version 1.2, it is mandatory to specify this attribute.) Hence, for this plain MOL file, a filter container is automatically provided. It's name is `Trifolium:Find_tri_CustomerAddendum`.

8.2 Obtaining Universe Definitions

A frequently asked question goes something like:

> "In M-Script, we used to be able to get programmatic access to universe definitions. Can we do this in the Java Extension Framework?"

With the possibilities mentioned in Section 8.1, this is now possible by making use of the universe-based filter containers.

For example, consider Listing 8.2. Suppose we want programmatic access to this universe. Or rather, to a specific *interface* of that universe. We can obtain that through the container provided by the mapping MOL-file, defined in Listing 8.3.

The trick is to obtain a container executor for the filter pane of the associated container, and then examine the result of the `specify` method. The `specify` method returns an object of type `MiContainerSpec`. From this, you can obtain an inspector of the container specification by using the method `container`, which returns an object of type

`MiContainerSpecInspector`. Using this class, you can query all panes of a container about its capabilities such as fields (including their types, titles, mandatoryness etc.), foreign keys and search keys (including their specification, titles, corresponding search containers etc.) and actions (including their title, availability and default icon specifications.) Using the `MiContainerSpec` you can in principle also access this information by addressing the Java-object representation of the MDSL XML DOM. These classes are constructed by the JAXB library [OM03], and therefore the interface to these are slightly different from the usual framework classes. For instance, some of the methods may return `null`. *These JAXB classes are notoriously difficult to work with, and doing so should be left to the framework!* The API of these classes cannot be guaranteed to be completely stable. Use the inspector interfaces instead!

Listing 8.5: Obtaining Universe Specs Programmatically

```
2     final MiMap<MiDataType.MeType, MiList<String>> fieldsByType =
          McTypeSafe.createHashMap();
3
4     final MiContainerExecutor.MiProvider universeContainer =
5         containerRunner.executor("Trifolium:Find_tri_CustomerU").
              initiate();
6
7     final MiContainerSpecInspector specification =
8         universeContainer.specify().container();
9     final MiPaneSpecInspector filter =
10        specification.panes().get(MePaneType.FILTER);
11
12    for (final MiFieldSpecInspector field : filter.fields()) {
13      final MiDataType.MeType fieldType =
14          field.getType().getType().getType();
15
16      if (!fieldsByType.containsKeyTS(fieldType)) {
17        fieldsByType.putTS(fieldType,
18                           McTypeSafe.<String>createArrayList());
19      }
20      final String fieldDescriptor;
21      if (field.getTitle().isDefined()) {
22        fieldDescriptor = field.getTitle().asString();
23      } else {
24        fieldDescriptor = "<Untitled: " + field.getName().asString
              () + ">";
25      }
26      fieldsByType.getTS(fieldType).add(fieldDescriptor);
27    }
28    // now list all the fields, grouped by type
29    for (final Map.Entry<MiDataType.MeType, MiList<String>>
          entries : fieldsByType.entrySetTS()) {
30      System.out.println("Fields of type " + entries.getKey());
31      for (final String fieldDescriptor : entries.getValue()) {
32        System.out.println("  " + fieldDescriptor);
```



```
33          }
34      }
```

Listing 8.5 shows an example, where the specification is obtained for the container `Trifolium:Find_tri_CustomerU`, and therefore, in effect, the specification of the universe `Trifolium::CustomerU`. The example code traverses all the available fields, group them by type, and then print out the types followed by the fields of that type. In line 8 the `specify` method is invoked to obtain the (MDSL) specification of the entire container, and the `container` method is used to obtain an inspector. In line 10 we obtain the information related to the filter pane (assuming that the filter pane is indeed present!) In line 12 we loop over all fields of that pane. For each field, we first check its most basic type information (line 14) and add a description of the field to a list mapped by that type.

In order to obtain the specification of a plain MOL file, simply access the specification of the corresponding `Find_`-container. For example, in order to programmatically inspect the fields available in the custom table `tri_CustomerAddendums` (see Listing 8.1) you can do that by examining the specification of the container `Trifolium:Find_tri_CustomerAddendums`.

Similarly, you can inspect the specification of any "ordinary" container, such as `maconomy:TimeSheets` or `maconomy:GeneralJournal`.

8.2.1 Inspecting Specifications

Obviously, you can use such inspectors to inspect the properties of any pane (not only the filter pane, as shown in the example above.) In the following, we shall list the relevant methods used for this purpose.

Everything starts with the `MiContainerSpec` which is the direct result of the `specify` method. This type has a number of methods:

Method	Remarks
`container`	This method returns a `MiContainerSpecInspector`. This is the recommended way of inspecting container specifications!
`getResource`	This method returns the entire specification as an MDSL file resource. This is rarely used outside of the framework, but may be useful for debugging.
~~`getXContainer`~~	This method returns a JAXB representation of the underlying XML. *You are discouraged from using this information, and the API is not guaranteed to be stable!* Use the inspectors instead!

So, the inspector top-enty level is the `MiContainerSpecInspector`. This type has a number of methods:

Method	Remarks
`getName`	Returns the name of the container.
`getKeyFieldNames`	Returns a collection of names, representing the key-fields used when addressing this container. For example, the `maconomy:Jobs` container will return one key field: `JobNumber`, whereas the `maconomy:TimeSheets` container will return `EmployeeNumber` and `PeriodStart`.
`panes`	This method returns an object representing the collection of panes. Among other things, this type implements an `Iterable` that allows you loop through all the panes using Javas `for-each` construct.

Pane Specifications

The `panes` method returns an object of type `MiPaneSpecsInspector`. This represents the overall collection of panes within the container.

Since it implements `Iterable<MiPaneSpecInspector>` you can easily iterate through all pane inspectors of panes in this container. In addition, this type has a number of interesting methods:

Method	Remarks
`getNames`	Returns a collection of all known pane names.
`get`	This method takes either a pane type or a pane name as argument, and returns an inspector giving access to information of the specified pane. If the specified pane does not exist, an exception is thrown.
`getOpt`	This method is similar to `get` except that it returns an optional (i.e., potentially undefined) inspector. If the specified pane does not exist, and undefined value is returned.

Each pane specification inspector is represented by `MiPaneSpecInspector`. It has a number of interesting methods:

Method	Remarks
`getName`	Returns the name of this pane.
`getType`	Returns the pane type.
`getTitle`	Returns the title of the pane.
`getEntityName`	Returns the name of the entity (i.e., the actual or virtual database table) containing the data in this pane. For example `TaskListLine` or `JobHeader`.
`getEntityTitle`	Returns the title of the entity of this pane. The title is always descriptive, such as `Task` or `Job`.
`fields`	This method returns an object of type `MiFieldSpecsInspector`, representing the collection of fields (and variables) in this pane.
`actions`	This method returns an object of type `MiActionSpecsInspector`, representing the collection of actions in this pane.
`foreignKeys`	This method returns an object of type `MiForeignKeySpecsInspector`, representing the collection of foreign keys (and search keys) in this pane.
`getKeyFieldNames`	Returns the key field names of the data in this pane.

Field Specifications

The `MiFieldSpecsInspector` class implements `Iterable<MiFieldSpecInspector>`. This means that you can easily iterate over all fields in a pane using Javas `for-each` construct. In addition this class has the following methods of interest:

Method	Remarks
`getNames`	Returns a collection of all field names defined in this pane.
`get`	This method takes a field name as argument, and returns an inspector giving access to information about that field. If the specified field does not exist, an exception is thrown.
`getOpt`	This method is similar to `get` except that it returns an optional (i.e., potentially undefined) inspector. If the specified field does not exist, and undefined value is returned.

Hence, a field specification is represented by an object of type `MiFieldSpecInspector`. This type contains a number of interesting methods:

Method	Remarks
`getName`	Returns the name of this field.
`getType`	Returns type information about the field, represented by the type `MiFieldTypeSpecInspector`. A number of type-related information is comprised by that type.
`getTitle`	Returns the title of the field.
`isMandatory`	Returns information about the mandatoryness of the field.
`getOpenness`	Returns information about whether this field is open for editing or not, and if it is, in what states the field is open.
`isAutoSubmit`	Returns information about the default auto-submit behavior of the field.
`isHidden`	Returns information about whether or not the field is considered "hidden" (i.e., cannot be part of a layout.)
`isSecret`	Returns information about whether or not this field is considered "secret" (i.e., it's content is shown using bullets or similar rather than clear text.)
`isKey`	Returns information about whether this field is a key field.
`isField`	Returns `true` if the field is a "field" (i.e., persisted.)
`isVariable`	Returns `true` if the field is a "variable" (i.e., not persisted.)
`getSearchSuggestions`	Returns the search-suggestions behavior used for this field by default.
`getForeignKeyOrdering`	Returns an ordered list of names representing the foreign keys to which this field is related. The items are "ordered" in the sense that the first (enabled) searchable foreign key is used when/if searching is applied.

The type information, represented by `MiFieldTypeSpecInspector` contains detailed information about the type of the field. It has the following relevant methods:

Method	Remarks
`getType`	Returns the actual type information about this field, for example whether this is a String, an Amount, a Boolean etc. Also some of this information is further annotated. For example, it is indicated what the maximum length of Strings are, and what the actual type of a "popup" types is. You can obtain the very pure type information (i.e., String, Amount, Boolean etc.) by further invoking the `getType` method on the resulting object.

Method	Remarks
`getSubType`	This method returns sub-type information, if defined. For popup types, it will represent the actual popup type. Reals can have a sub type of "TimeDuration" meaning that the real-value should be shown as *hh:mm* by default.
`getMaxLength`	If max-length information is specified (for String types), the information can be obtained using this method. An undefined value means that there is no maximum length.
`isMultiLine`	Returns information about whether this field may contain newline characters.

Action Specifications

Action specifications are represented by the `MiActionSpecsInspector` type which implements `Iterable<MiActionSpecInspector>`. This means that you can easily iterate over all actions of a pane using Javas `for-each` construct. In addition this class has the following methods of interest:

Method	Remarks
`getNames`	Returns a collection of all action names defined in this pane.
`get`	This method takes an action name as argument, and returns an inspector giving access to information about that action. If the specified field does not exist, an exception is thrown.
`getOpt`	This method is similar to `get` except that it returns an optional (i.e., potentially undefined) inspector. If the specified action does not exist, and undefined value is returned.
`standardActions`	This method returns a collection of all actions which represent *standard actions*. A standard action is one of the generically available actions representing an event on a container: `Add`, `Insert`, `Create`, `Read`, `Update`, `Delete`, `PrintThis`, `Move_Up`, `Move_Down`, `Indent` or `Outdent`. Notice the absence of the ~~`Print`~~ method: it is not considered a "standard action", but a "print action" (see below.)
`specificActions`	This method returns a collection of all *specific actions* of this pane. A specific action is one of the named actions that can be defined "ad hoc." on a container pane, such as `SubmitTimeSheet`, `Post` and `CloseJob`. Hence, these are the actions representing events on the container pane which are *not* considered standard acttions.

Method	Remarks
`getPrintAction`	This method returns an optional inspector giving information about the "Print" (or Print...) action. Notice that this is *not* the "Print This" action! The Print "action" is not really an action, certainly not one invoking an event in this container. Rather, it represents a hint to the client side, that the card pane of some *auxiliary* container can be launched.

The `MiActionSpecInspector` type, thus, contains information about the actions of this pane. I.e., actions that can lead to an event in the container in question. This type has the following methods of interest:

Method	Remarks
`getName`	Returns the name of this action.
`getType`	Returns type information about the field, represented by the type `MiFieldTypeSpecInspector`. A number of type-related information is comprised by that type.
`getTitle`	Returns the title of the action.
`getIcon`	Returns the default icon name associated with this action. This value may be undefined, in which case the system default will be assumed.
`getAvailability`	Returns an enum value representing the availability of the action, i.e., whether it must be explicitly stated in the layout in order to appear.
`isStandardAction`	Returns `true` if the action is one of the "standard actions", i.e., `Add`, `Insert`, `Create`, `Read`, `Update`, `Delete`, `PrintThis`, `Move_Up`, `Move_Down`, `Indent` or `Outdent`.
`isSpecificAction`	Returns `true` if the action is a "specific action", such as `SubmitTimeSheet`, `Post` and `CloseJob`. Hence, an action that is not a standard action and is not a the Print... action.
`getType`	Returns the action type. The resulting enum has one of the following values: • `STANDARD` which indicates that this is a standard action. • `SPECIFIC` which indicates that this is a specific action. • `PRINT` which indicates that this is the Print action.

The Print-"action" is represented by the `MiPrintActionSpecInspector` type. In addition to the methods listed above for the `MiActionSpecInspector`, it has the following methods:

Method	Remarks
`getContainerPaneName`	Returns the name of the container/pane which can be launched from the client side.
`getLayoutName`	Returns the layout name that should be used to display the launched pane.

Foreign Key and Search Key Specifications

The specification of foreign keys and search keys are described using the same types. The `MiForeignKeySpecsInspector` type represents the collection of all foreign- and search keys. This type implements `Iterable<MiForeignKeySpecInspector>` and can therefore be used with Javas `for-each` construct. Apart from this, it contains the following methods of interest:

Method	Remarks
`getNames`	Returns a collection of all foreign key/search key names defined in this pane.
`get`	This method takes a foreign key/search key name as argument, and returns an inspector giving access to information about that foreign key. If it does not exist, an exception is thrown.
`getOpt`	This method is similar to `get` except that it returns an optional (i.e., potentially undefined) inspector. If the specified foreign key/search key does not exist, and undefined value is returned.

The type `MiForeignKeySpecInspector` is used to describe each foreign key/search key, and contains the following methods:

Method	Remarks
`getName`	Returns the name of this foreign key/search key.
`getTitle`	Returns the title of this foreign key/search key.
`isSearchable`	Returns `true` if this foreign key can be used for searching. Notice that all search keys are searchable, whereas a foreign key need not be.
`isSearchKey`	Returns `true` if this specification represents a search key (not a foreign key.)
`isForeignKey`	Returns `true` if this specification represents a foreign key (not a search key.)

Method	Remarks
`isParentReference`	Returns `true` if this foreign key is used to designate the parent reference in a tree-table structure.
`getSearchContainerPane`	Returns the name of the search container/pane used when searching takes place. In case the foreign key is not searchable, an undefined value is returned.
`getLinks`	Returns a collection of all links of this foreign key/search key. Each link is represented by the type `MiLinkSpecInspector`.
`getSupplementLinks`	Returns a collection of all supplement links of this foreign key/search key. Each supplement link is represented by the type `MiLinkSpecInspector`.
`getEnablementOpt`	This method returns an optional enablement specification. If this is undefined, it means that there is no specific enablement condition and, hence, that this foreign key/search key is always considered enabled. If the returned value is defined, the enablement condition is specified by the type `MiEnablementSpecInspector`.

The `MiLinkSpecInspector` has the following methods of interest:

Method	Remarks
`getFieldName`	Returns the name of a field in *this* container which is related to a field in the foreign container.
`getForeignFieldName`	Returns the name of a field in the foreign container which is linked to the field returned by `getFieldName`.

The `MiEnablementSpecInspector` is used to describe a conditional foreign key/search key. A condition has one of the forms:

- `field` = ⟨*popup value*⟩
- "default" which means that this is valid for any value of the `field` that is not used by any other foreign key.

and has the following methods of interest:

Method	Remarks
`getFieldName`	Returns the name of a field which is part of the enablement condition.

Method	Remarks
`getValue`	Returns the specific value forming the enablement condition. If this enablement condition does not have a specific value, an exception is thrown.
`getValueOpt`	Returns the specific value forming the enablement condition, or an undefined value, if the enablement is a *default* enablement.

Bibliography

[CMd] Deltek Inc. *Deltek Maconomy—MDML Language Reference Guide.*

[Del13] Deltek Inc. *Deltek Maconomy Extender—Handbook for Extending the Workspace Client*, March 2013.

[EL] Deltek Inc. *Deltek Maconomy—MDML/Expression Language Standard Function.*

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java^{TM} Language Specification.* Addison-Wesley, 3rd edition, 1996–2005.

[iTe08] iText Software Corp. itext 2.1.3. http://itextpdf.com/history/?branch=21&node=213, July 2008.

[jav] What's new in jdk 8.

[Log] Logback documentation. http://logback.qos.ch/manual/.

[Low10] Bruno Lowagie. *iText in Action.* Manning, 2nd edition, November 2010.

[MQL] Deletek Inc. *MQL Language Reference.*

[MSca] Deltek Inc. *M-Script Language Reference.*

[Mscb] Deltek Inc. *M-Script Maconomy API Reference.*

[MVA10] Jeff McAffer, Paul Vanderlei, and Simon Archer. *OSGi and Equinox, Creating Highly Modular Java^{TM} Systems.* Addison-Wesley Professional, 2010.

[OM03] Ed Ort and Bhakti Mehta. Java architecture for xml binding (jaxb), March 2003.

[OSG08] OSGi Alliance. http://www.osgi.org/Specifications/HomePage/, June 2008.

[Pdm] Deltek Inc. *Portal Development Method.*

[QMc] Deltek Inc. *Deltek Maconomy—MCSL Quick Reference Guide.*

[QMd] Deltek Inc. *Deltek Maconomy—MDML Quick Reference Guide.*

[QMm] Deltek Inc. *Deltek Maconomy—MMSL Quick Reference Guide.*

[QMn] Deltek Inc. *Deltek Maconomy—MNSL Quick Reference Guide.*

[QMw] Deltek Inc. *Deltek Maconomy—MWSL Quick Reference Guide.*

[Slf] Slf4j documentation. http://www.slf4j.org/docs.html.

Migration Guidelines

This appendix describes migration issues that may occur when migrating from version 16 to a later version, and how to address them.

Methods that have been Removed

Migrating from 16 to 17

In version 16, a number of methods were marked as deprecated. Following the deprecation cycle, these have now been removed in version 17.

Type/Area	Method	Remarks
`MiExtendedDataModel` (extended data models)	~~`setKeyField`~~	This previously deprecated method was clearly marked with a "Do not use" warning, since it was meant for framework use only. There is no replacement for this method.

`McKey`	`key(`~~`MiKey`~~`)`	This method (creating an `MiKey` based on an `MiKey`) existed by accident. It is not at all needed, and has been removed. If this method has been used, it is equivalent to just reference the argument. Hence:
		`MiKey key1 = ...;`
		`MiKey key2 = McKey.key(key1)`
		is equivalent to:
		`MiKey key1 = ...;`
		`MiKey key2 = key1`
		
		Notice that the `key` method taking a `String` argument still exists!

A single method has been removed (without having been previously marked as depre-

cated.)

Method	Remarks
`containerExecutor.specify()`	This method has been removed from the container executor interface. Instead, the method `inspect` has been introduced. This method is, however, *not* a 1:1 replacement of the `specify` method. Instead, the `specify` method is now available on the object returned by the `executor` method.

Version 17 to 19

In version 17, a number of methods were marked as deprecated. Following the deprecation cycle, these have now been removed in version 19.

Type/Area	Method	Remarks
`MiDatabaseApi` (SQL)	`sql(String, McRowRange)`	This method was deprecated in 17 service pack 1 (2.2.1). Due to the potential security issues that may arise, unless the programmer takes great care, this method has now been completely removed. While it is still possible to make SQL queries based on a textual content, the API has been changed such that it can enforce proper guarding against SQL-injection, without requiring special attention from the programmer using the method. Therefore, the method `sqlBuilder` found on the `MiDatabaseApi` interface, should be used to constuct a `McSql` object. From such an object, you can either directly run the query, or you may pass it on to the method `sql(McSql)` instead. Please refer to Section 6.2.2 for more information on using this.
`MiDatabaseApi` (Commit etc.)	`commit()`	This method was always clearly marked as having been deprecated, and that it must not be used. It has now been removed from the database interface, and there is no replacement for this method.

Type/Area	Method	Remarks
	~~`rollback()`~~	This method was always clearly marked as having been deprecated, and that it must not be used. It has now been removed from the database interface, and there is no replacement for this method.
	~~`setAutoCommit()`~~	This method was always clearly marked as having been deprecated, and that it must not be used. It has now been removed from the database interface, and there is no replacement for this method.
	~~`getAutoCommit()`~~	This method was always clearly marked as having been deprecated, and that it must not be used. It has now been removed from the database interface, and there is no replacement for this method.
`MiQueryInspector`	~~`asPaneValue()`~~	This method is now deprecated, since it clearly returns information that cannot be relevant for a query-result inspection. Code that makes use of this method should therefor be rewritten to query the data by directly invoking the query inspector methods.

A single method has been removed (without having been previously marked as deprecated.)

Method	Remarks
`containerExecutor.`~~`specify()`~~	This method has been removed from the container executor interface. Instead, the method `inspect` has been introduced. This method is, however, *not* a 1:1 replacement of the `specify` method. Instead, the `specify` method is now available on the object returned by the `executor` method. The `inspect` method relates to the pane targeted by the current container executor, the `specify` method relates to the entire container.

In addition to the above methods, the methods that were deprecated between 16 and 17 have been removed in version 19. See below for details.

Methods With Changed API

Version 17 to 19

Until and including version 17, `MiFetchStrategy`, `MiPersistenceStartegy` and extensions of those interfaces have exposed a return value of type `MiPaneInspector` or `MiPaneAdmission`.

Apart from the fact, that it's clearly a mistake that a `MiPaneAdmission` was returned, (and even the fact that *pane*-related information is returned), such values are very difficult to provide. A much better choice would have been to return `MiQueryInspector`s.

For this reason, we have decided to break backwards compatibility by letting the `select` methods return a `MiQueryInspector`. And we have enhanced the way `MiQueryInspector`s can be constructed: this now happened using easy-to-apply factory methods `create` on the `McQueryInspector` class. To summarize

Method	Remarks
`select`	These methods on fetch- and persistence strategies, thus `MiFetchStrategy` and `MiPersistenceStrategy` now return a `MiQueryInspector` rather than a `MiPaneInspector` or `MiPaneAdmission`. The migration is expected to be easily done, since essentially the intentionn was to return a set of records: something that is nicely done by a `MiQueryInspector`. We would advice *not* to be tempted to make the code conversion by applying the `asPaneValue()`-method on the returned `MiQueryInspector`. Instead make use of the methods directly offered on that type to query the data. In case you have implemented your own `MiPersistenceStrategy`, simply let the `select` methods return a `MiQueryInspector`. Either construct such values directly from the pane value you already have, or simplify your code by creating a query inspector from a set of records or from a `MiRecordCollection`.

Methods that Have Become Deprecated

Version 16 to 17

In version 17, a number of methods are now marked as deprecated for various reasons. These will be removed in a future version of Maconomy, most likely version 18. Therefore, code making use of deprecated code should be changed as soon as possible.

Root Data Models

The following methods may have been implemented by root data models. Please refer to the guide line in the table below:

Method	Remarks
~~defineDomesticSpec~~ ~~definePersistenceStrategy~~ ~~defineAdditionalReadCondition~~ ~~defineReadCondition~~ ~~defineDefaultSortOrder~~ ~~defineSomeKeyValue~~ ~~defineAutoChildrenDeletion~~	All of these methods are superseded by methods with a similar name, but having an additional argument: a `containerRunner` of type `MiContainerRunner.MiDefine`. Simply change your data model into implementing the method with the additional `containerRunner` argument.
~~getPaneName~~	This method has been deprecated. Instead, the pane name can be obtained from the `containerRunner`. Methods that did not previously have a `containerRunner` argument now do.
~~searchSuggestions~~	Used inside the `defineDomesticSpec` method. This method has been deprecated. Instead, similar functionality can be obtained by using the method `searchBehavior`. This method takes a new (but similar) enumeration type as argument. The values of the enumeration type arguments have the same names as previously. Thus, simply change the argument type from `XeSuggestionsType` to `MeSearchBehavior` and change the method name into `searchBehavior`. In the unlikely case that the method argument has previously been obtained from somewhere else, the `MeSearchBehavior` type contains a method that can convert from the old type to the new.

Extended Data Models

The following methods may have been implemented by extended data models. Please refer to the guide line in the table below:

Method	Remarks
~~defineDomesticSpec~~ ~~definePersistenceStrategy~~ ~~defineSomeKeyValue~~ ~~defineDomesticAutoDeletion~~	All of these methods are superseded by methods with a similar name, but having an additional argument: a `containerRunner` of type `MiContainerRunner.MiDefine`. Simply change your data model into implementing the method with the additional `containerRunner` argument.
~~defineDualField~~	Used inside the `defineDomesticSpec` method. The concept of "dual fields" is deprecated. Instead of "automatically" storing duplicate information in database tables representing additional fields, make use of universes or database joins to efficiently retrieve information from the original source record.
~~searchSuggestions~~	Used inside the `defineDomesticSpec` method. This method has been deprecated. Instead, similar functionality can be obtained by using the method `searchBehavior`. This method takes a new (but similar) enumeration type as argument. The values of the enumeration type arguments have the same names as previously. Thus, simply change the argument type from `XeSuggestionsType` to `MeSearchBehavior` and change the method name into `searchBehavior`. In the unlikely case that the method argument has previously been obtained from somewhere else, the `MeSearchBehavior` type contains a method that can convert from the old type to the new.
~~getPaneName~~	This method has been deprecated. Instead, the pane name can be obtained from the `containerRunner`. Methods that did not previously have a `containerRunner` argument now do.

Extended Data Models with Long-Text Fields

The class `McAbstractLongTextFieldDataModel` has been superseded by the class `McAbstractLongTextFieldExtendedDataModel`. Therefore, if you have made data models extending `McAbstractLongTextFielDataModel`, change them into extending `McAbstractLongTextFieldExtendedDataModel` instead.

The following methods may have been implemented by extended data models based on `McAbstractLongTextFieldExtendedDataModel`

Method	Remarks
~~`defineTextContentMapping`~~	This method is superseded by a method with a similar name, but having an additional argument: a `containerRunner` of type `MiContainerRunner.MiDefine`. Simply change your data model into implementing the method with the additional `containerRunner` argument.
~~`defineSplitTextSpec`~~	This method has changed name, and in addition takes an additional `containerRunner` argument. The name of the method is now `defineSplitTextFormat`. Simply implement this method instead, using the same code body as the one used for `defineSplitTextSpec`.
~~`thisFieldCannotBeChangedMessage`~~	This method has changed name, and in addition takes an additional `containerRunner` argument. The name of the method is now `defineMessageFieldCannotBeChanged`. Simply implement this method instead, using the same code body as the one used for `thisFieldCannotBeChangedMessage`.
~~`defineNoTextsAccessErrorMessage`~~	This method has changed name, and in addition takes an additional `containerRunner` argument. The name of the method is now `defineMessageNoTextsAccess`. Simply implement this method instead, using the same code body as the one used for `defineNoTextsAccessErrorMessage`.

Container Executors

In order to keep the API of the framework in good shape, it has been decided to change how container executors are obtained. The reasons for this are:

- The number of methods available for obtaining a container executor is quite large. This is due to the fact that a number of different parameters were needed *and* that each of these parameters should be supported using different concrete types. This has led to an API that is harder to manage both for programmers and framework providers.
- Even though a large number of methods were available, not all relevant combinations were supported. After the change, all (including the missing) combinations exist.

- The API has now been changed to a style that offers more flexibility, a smaller number of methods, and a style that is very much in line with other areas in the framework.

As a result, the following methods have been deprecated:

Method	Remarks
~~`openExecutor`~~	A container executor is now obtained by first invoking the `executor` method and then the `construct` method. The argument to the `executor` method corresponds to the container name (it can be left out similar to cases where there were no container name provided for the `openExecutor` method). The argument to the `construct` method corresponds to the pane name/pane type. Leaving out the pane name/type corresponds to the situation where `openExecutor` was invoked without arguments. The following entries in this table points out how to migrate invocations to `openExcecutor`.
~~`openExecutor()`~~	Replace by: `executor().construct()`
~~`openExecutor(pType)`~~	Replace by: `executor().construct(pType)`
~~`openExecutor(pName)`~~	Replace by: `executor().construct(pName)`
~~`openExecutor(cName, pType)`~~	Replace by: `executor(cName).construct(pType)`
~~`openExecutor(cName, pName)`~~	Replace by: `executor(cName).construct(pName)`
~~`createContainer`~~	A container executor provider is now obtained by first invoking the `executor` method and then the `initiate` method. The argument to the `executor` method corresponds to the container name. The following entries in this table points out how to migrate invocations to `createContainer`.
~~`createContainer(cName)`~~	Replace by: `executor(cName).initiate()`
`ce.`~~`deriveExecutor`~~	To derive a container executor from a "master" container executor, `ce`, the method name is now simply `derive`. Hence, simply replace `deriveExecutor(pName)` by `derive(pName)`. The new method is now additionally found in a variant that accepts the pane *type*, in addition to accepting the pane name.

Version 17 to 19

Pane Values

A number of interfaces reflecting pane values (read-only as well as read-writable). This applies to interfaces `MiPaneInpspector`, `MiPaneAdmission` and `MiPaneValue`.

Here the method previously called ~~`getKeys`~~ has been deprecated. The same functionality is now offered through the method called `getKeyValues`.

Database Access

In the `MiDatabaseApi` the following methods have been deprecated.

Method	Remarks
`mselect(MiKey, MiQuery)`	This variant of the method has been deprecated. Use the builder-style variant: `mselect(...).from(...).where(...)` instead.
`select(MiKey, MiQuery)`	This variant of the method has been deprecated. Use the builder-style variant: `select(...).from(...).where(...)` instead.
`mcount(MiKey, MiQuery)`	This variant of the method has been deprecated. Use the builder-style variant: `mcount(...).where(...)` instead.
`count(MiKey, MiQuery)`	This variant of the method has been deprecated. Use the builder-style variant: `mcount(...).where(...)` instead.
`insert(MiKey, MiValueInspector)`	This variant of the method has been deprecated. Use the builder-style variant: `insert(...).setAll(...)` instead.
`update(MiKey, MiValueInspector, MiQuery)`	This variant of the method has been deprecated. Use the builder-style variant: `update(...).setAll(...).where(...)` instead.
`delete(MiKey, MiQuery)`	This variant of the method has been deprecated. Use the builder-style variant: `delete(...).where(...)` instead.

Record-like classes

Method	Remarks
~~`retainAllCopy`~~	This method has been deprecated and replaced with a method named `absorbAllCopy`. The two methods are identical in behavior: only the name has changed. This has happened because the name of the old method has proved hard to understand.
~~`retainAll`~~	This method has been deprecated and replaced with a method named `absorbAll`. The two methods are identical in behavior: only the name has changed. This has happened because the name of the old method has proved hard to understand.

Controlling Container Contribution Dependencies

The ordering of container contributions can be impacted by certain declarations in the `plugin.xml` files.

In order to make the terminology more clear as well as to introduce new functionality, some of those declarations have changed.

Method	Remarks
~~`<predecessor id="...">`~~	This declaration was used to declare that *this* contribution "comes after/is below" (i.e., is closer to the root contribution) than the contribution indicated by the `id`. This declaration has been deprecated, and changes name to `<below id="...">`. This name change has been made to make the statement more readable: If *a* is "below" *b*, *a* is closer to the root (the root is at the "bottom".)
~~`<successor id="...">`~~	This declaration was used to declare that *this* contribution "comes before/is above" (i.e., is further away from the root) than the contribution indicated by the `id`. This declaration has been deprecated, and changes name to `<above id="...">`. This name change has been made to make the statement more readable: it is the opposite of `<below>` (see above).
~~`<drop id="...">`~~	This declaration was used to declare that some other contribution should be entirely disregarded. This notion is unsound, and has been deprecated. It is not replaced by anything: you should change your code so that ~~`<drop>`~~ is *not* needed!

`<extend ...type="generic">`	This declaration was used to indicate that an extension contribution was "generic" and thereby (even if it matches all names in a sequence of cloned containers) should only occur once and at the very top (i.e., opposite the root contribution). This concept was easily confused with the `any="true"` declaration which means something different, although the two was in practice used together. The `type` attribute has been deprecated and is now replaced by the attribute `group`. A contribution that was before marked as `type="generic"` must be converted into: `group="top"`.

`Lock` and `Unlock` events

As of Maconomy 2.3, `Lock` and `Unlock` events will not occur. These events have always been considered "convenience locks," with no real guarantee that locking really does occur. Since these events have been removed, a number of methods are now deprecated, with no replacements.

Method	Remarks

`onLock`
`onLockPre`
`onLockPost`
`onUnlock`
`onUnlockPre`
`onUnlockPost`
	Since lock/unlock events will no longer occur, implementing these methods in data models has been deprecated. Simply remove implementations of these methods.

`supportLocking`	Upon defining a domestic spec, it is now desupported to specify that locking is supported (since locking will never occur.) Simply remove invoking this method.
`containerExecutor.lock` `containerExecutor.unlock`	Since lock/unlock events never occur, attempts to invoke these programmatically will not do anything. For this reason, these methods have been deprecated. Simply do not invoke these methods. In case "real locks" must be taken to ensure code correctness, you should make use of database locks (as previously.)

New and Noteworthy

This appendix highlights new & noteworthy features in the Extension Framework. It was first introduced from Maconomy version 19 (2.3) and therefore does not include information prior to that version.

For more details about the news, you are referred to the applicable sections elsewhere in this guide.

Version 19 (2.3)

Background Tasks

The Maconomy server/application now has a built-in framework for defining and executing tasks in the background, at a scheduled date and time. This framework is expected to conveniently compliment extensions. While it does not require any coding to set-up such scheduled background tasks, any extension logic will be taken into account by this framework, including new actions and new containers.

Furthermore, it is possible to programmatically add tasks from the Extension Framework.

Container Executors and Record Executors

New API has been introduced to easily iterate over records in a container executor, giving access to invoke events on each of the iterated records. The concept of a "record executor" has been introduced to supplement this (see Section 6.1.7.) As an example, deleting all rows where the `ActivityNumber` is empty is now significantly easier. In *previous versions* you had to do something like:

```
// Versions < 19
MiContainerExecutor ce = ...;
final int lastRowIndex = ce.getRowCount() - 1;
boolean hasMoreRows = ce.setRowIndex(lastRowIndex);
while (hasMoreRows) {
```

```
  MiRecordInspector record = ce.getRecord();
  if (record.getStr(ACT_NO).isEmpty()) {
    ce.delete();
  }
  hasMoreRows = ce.decRowIndex();
}
```

In *version 19* this can be expressed like:

```
// Version 19
MiContainerExecutor ce = ...;
MiExpression<McBooleanDataValue> cond = eq(ACT_NO).str("");
for (MiRecordExecutor record : ce.matchBy(cond).reverse()) {
  record.delete();
}
```

The events `create` and `update` have had the type of the `changeValues` argument loosened to accept a `MiValueInspector` rather than a `MiDataValues` type. This change does not break any current use of container executors, but make the methods easier to use in addition to making it clear, that the argument is not (cannot be) modified.

A new method on container executors, `getKeyValues`, can return the formal key value of the record having focus.

Miscellaneous

- The Maconomy server 2.x (previously known as the coupling service) now supports Java 8. This means that extension code can make use of Java 8 features, including lambda-expressions and `Streams`. Please refer to the Java 8 release notes [jav] for more information.
- A few handy utility methods have been added to record-like classes. These include: `keepAllCopy`, `keepAll`, `getPopupLiteral`, `entryIterator`
- It is now possible to declare that a container contribution should be grouped "close to the root." For example, if you clone an existing container and expose it by a new name, and then make an extension to the new name that you would like to be effectively seen as a part of the "root" of that container, you can do that by declaring `group="root"` in the declaration of the extension contribution. Any other standard extension contributions will be placed "above" any contribution in the `root` group. It is only possible to declare explicit dependencies of extensions that are part of the same group.
- An API has been introduced for picking up certain configurable system settings. This includes mail-server information, references to file paths and references to URLs. Using this API, it is possible to make, e.g., integrations to 3rd-party systems

while ensuring that conflicts do not accidentally occur between, e.g., a production system and various test systems.

Document History

Version	Remarks
$1.0.0_{16}$	First final version.
$1.1.0_{17}$	Updated with new features for version 17 (2.2). Section 4.2.4 is completely new and describes how name spaces can be used for fields, variables, actions, foreign keys and search keys. Significant changes to sections 6.2.2, 6.2.3 and 6.2.4. These sections have been updated to cover builder-style queries. Added warning against SQL injection when using `String`-based SQL API. Updated virtually all examples to use name space. Section 6.1 has been updated to cover new way of obtaining container executors. All examples have been updated accordingly. Added Chapter 8, explaining how filter-pane containers based on custom universes may be automatically obtained from the Maconomy server, and how to obtain a universe or MOL specification programmatically. Appendix on Migration Guidelines has been added. Section 1.3 has been added to make it easier to understand code listings.
$1.1.1_{17}$	Updated documentation to cover revised textual SQL api.
$1.2.0_{19}$	Updated documentation to cover Maconomy version 2.3 (19). Section 6.1 has been updated to cover the use of record executors. Record executors are especially documented in Section 6.1.7. Section 6.3 includes information on background tasks and the background task API. Section 7.5 documents how to access the system configuration, including using named references for paths and URLs. Lock/Unlock event no longer occur and have become deprecated.

Index

D

S

T

