Maconomy RESTful Web Services

Programmer's Guide
2015

Edited by

Rune Glerup

Note: This document is not updated for the 2.3 GA release.

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 April 2015.

Contents

Chapter 1

Introduction

The Maconomy RESTful Web Service Interface is a programmatic interface that provides access to data and business functionality in the Deltek Maconomy ERP product.

1.1 The Container Abstraction

The web service exposes data and functionality through so-called containers. Containers are an abstraction that gives a uniform interface to all functionality within the Maconomy system. The same generic structure, conventions, and organization are used for data retrieval and other interactions throughout the system.

A container is made up of a number of panes. Three types of panes exist:

- Card panes
- Table panes
- Filter panes

1.1.1 Card panes

Card panes contain a single record. Examples in Maconomy include the `Jobs` container and the time sheet header part of the `TimeSheets` container.

1.1.2 Table panes

Table panes contain zero or more records. Examples in Maconomy include the job budget lines part of the `JobBudgets` container and the time sheet lines part of the `TimeSheets` container.

1.1.3 Filter panes

Filter panes, like tables, contain zero or more records. Filters allow you to select subsets of the potential content by applying certain restrictions. This can be used to provide search functionality and filter options.

Filters also support functionality like paging (for example, showing records 1 to 25 of 3200), sorting, and limiting which fields are included in the data.

In the Maconomy Workspace Client, panes are composed into workspaces. In a workspace panes are tied together by key bindings that govern how data is distributed in the workspace.

In the web service interface, you interact directly with containers and programmatically navigate the key bindings, similar to how the automatic data distribution occurs in the workspace engine.

The following example shows how a user workflow in the Workspace Client is similar to the workflow of a client program that interacts with the web service:

1. The user opens the Expense Sheets workspace.
2. The user uses the topmost panel List of Expense Sheets and locates needed expense sheets.
3. The user clicks on an expense sheet in the List of Expense Sheets and the corresponding card and table pane data is shown in Expense Sheet and Expense Sheet Lines respectively.

This interaction is closely mirrored by the interactions that a client program connecting to the web service performs:

1. The client program gets the `ExpenseSheets` container resource.
2. The client program follows a hyperlink to the filter pane of the `ExpenseSheets` container, and interacts with the filter to find the specific expense sheets for the task at hand.
3. The client program follows a hyperlink to view the card and table pane data for a specific expense sheet in the `ExpenseSheets` container.

1.2 REST

The preceding example touched upon some of the central concepts in REST such as *resources* and *hyperlinks*. It is useful to know a little about what REST is, and the concepts and terminology associated with it.

REST stands for Representational State Transfer and is a style of web services that conform to a set of principles and conventions. A web service that is built on REST principles is said to be RESTful.

1.2.1 Resources

The central concept in REST is a *resource*. A resource is a domain object that is uniquely identified by a URL. For example, each expense sheet in a Maconomy system has its own URL.

When you access the URL for a resource, you get a *representation* of the current state of the resource. For an expense sheet, this representation contains all of the data in the card and table pane of the expense sheet. The same resource may have multiple representations, for example XML or JSON. When interacting with a resource, a client program can choose the representation it prefers. The payload of a request or response in HTTP is called an *entity*.

Resources are manipulated (read, updated, deleted, and so on) by a fixed set of HTTP verbs. The verbs used in the Maconomy RESTful web service interface are `GET`, `POST`, and `DELETE` [5].

1.2.2 Hyperlinks

Hyperlinks is a well-known concept from the web, and they are pervasive in RESTful web services. Hyperlinks work just like links on a web page and point to related resources. For example, the expense sheet filter has hyperlinks that point to specific expense sheets.

Hyperlinks are also used to represent available *state transitions*. For example, to update an expense sheet line, the client program need to follow a specific hyperlink. Resources have hyperlinks for all available state transitions.

Each link has an associated *link relation*, which is a value that defines what the link can be used for (for example, accessing a related resource, updating, submitting, transferring, and so on).

1.2.3 Other Styles of Web Services

REST is often contrasted with another style of web services exemplified by the SOAP protocol.

Rather than interacting with stateful resources via a standard set of verbs and following the standard HTTP application protocol used consistently across many web services from different sources, a typical SOAP web service offers a list of custom procedures that may be invoked over the network.

Instead of assigning each domain object a URL that can be used to retrieve and manipulate the object, a SOAP web service uses IDs to refer to domain objects. The IDs must then be supplied to appropriate procedure calls to operate on the objects. HTTP is only

incidentally used to transmit messages, but none of the useful features and properties of the web architecture are leveraged.

Rather than being discoverable by representing the possible interactions as hyperlinks, a typical SOAP web service relies on out-of-band means (such as detailed manuals and specifications) to communicate the interaction protocol for the web service.

1.2.4 Further Reading

It is recommended for developers working with producing or consuming RESTful web services to read the book "REST in Practice: Hypermedia and Systems Architecture" [11].

1.3 Example

This section shows how these concepts work in a practical example. You can use the `curl` command-line tool to get a representation of the current state of the service endpoint resource for the shortname `w16p2`:

```
$ curl -i
       'http://server/containers/v1/w16p2'
HTTP/1.1 200 OK
Date: Tue, 11 Nov 2014 12:55:40 GMT
Server: Jetty(8.1.14.v20131031)
Content-Language: en-US-x-lvariant-W
Vary: Content-Language,Accept-Encoding,User-Agent
Cache-Control: no-cache, no-transform
Content-Type: application/json; charset=utf-8
Transfer-Encoding: chunked

{
  "shortname": "w16p2",
  "authenticated": false,
  "languages": [
    {
      "locale": "da_DK",
      "tag": "da-DK",
      "title": "Dansk (Danmark)"
    },
    {
      "locale": "en_US",
      "tag": "en-US",
      "title": "English (United States)"
    }
  ],
  "versions": {
```

```
    "apu": {
      "major": "17",
      "minor": "0",
      "patch": "0",
      "hotfix": ""
    },
    "tpu": {
      "major": "17",
      "minor": "0",
      "sp": "100",
      "fix": "0",
      "beta": ""
    }
  }
}
```

The first lines are the HTTP status code and response headers (`curl` outputs this information when the `-i` option is used). The `200 OK` status code indicates a successful request.

The service endpoint provides a list of supported languages, basic version information about the system, and a flag that indicates if you authenticated the request. The service endpoint resource does not require authentication, but interacting with most other resources does require authentication. For example, this is evident when you access the `ExpenseSheets` container:

```
$ curl -i
       'http://server/containers/v1/w16p2/ExpenseSheets'
HTTP/1.1 401 Unauthorized
Date: Fri, 28 Nov 2014 14:42:26 GMT
Server: Jetty(8.1.14.v20131031)
Content-Type: application/json; charset=utf-8
WWW-Authenticate: Basic realm="Maconomy"
Vary: Accept-Charset,Accept
Transfer-Encoding: chunked

{
  "errorFamily": "service",
  "errorMessage": "The request requires user authentication",
  "errorSeverity": "error"
}
```

The HTTP status code `401 Unauthorized` indicates a particular error condition, and the request entity contains an error message as well as additional information about the kind of error. Client programs normally use the status code to dispatch to the appropriate error handling code.

To fix this error, you can authenticate by using the `-u` option in `curl`:

```
$ curl -i
       -u 'Administrator:123456'
       'http://server/containers/v1/w16p2/ExpenseSheets'
HTTP/1.1 200 OK
Date: Fri, 28 Nov 2014 14:42:53 GMT
Server: Jetty(8.1.14.v20131031)
Content-Language: en-US-x-lvariant-W
Vary: Content-Language,Accept-Encoding,User-Agent
Cache-Control: no-transform, max-age=86400
Content-Type: application/json; charset=utf-8
Transfer-Encoding: chunked

{
  "containerName": "expensesheets",
  "links": {
    "action:create": {
      "href": "http://server/containers/v1/w16p2/expensesheets/data/card",
      "rel": "action:create"
    },
    "action:insert": {
      "href": "http://server/containers/v1/w16p2/expensesheets/data/card/ ↩
    init",
      "rel": "action:insert"
    },
    "data:any-key": {
      "href": "http://server/containers/v1/w16p2/expensesheets/data;any",
      "rel": "data:any-key"
    },
    "data:filter": {
      "href": "http://server/containers/v1/w16p2/expensesheets/filter",
      "rel": "data:filter"
    },
    "specification": {
      "href": "http://server/containers/v1/w16p2/expensesheets/specification ↩
    ",
      "rel": "specification"
    }
  }
}
```

When you access the container resource (using the HTTP `GET` verb), you get a representation that serves as an entry point for interacting with the container. It contains hyperlinks to the specification, filter, and data subresources of the container. The `rel` property contains the link relation used by client programs to distinguish the purpose of each of the links. The link relation is also used as the key for each link in the `links` object.

Use the URL pattern `http://{host}/containers/v1/{shortname}/{container}` to access the entry point for a container. This is the only URL a client should construct itself. All further interactions should happen by navigating hyperlinks. This allows clients to keep working with future versions of the web service that use the existing link relations, but with new URL patterns.

The default representation of the resource is in JSON format. The HTTP response header `Content-Type` contains the media type for the representation. If you prefer to work with the resource in an XML representation, ask specifically for XML by including an `Accept` HTTP header in the request:

```
$ curl -i
       -u 'Administrator:123456'
       -H 'Accept: application/xml'
       'http://server/containers/v1/w16p2/ExpenseSheets'
HTTP/1.1 200 OK
Date: Fri, 28 Nov 2014 14:43:24 GMT
Server: Jetty(8.1.14.v20131031)
Content-Language: en-US-x-lvariant-W
Vary: Content-Language,Accept-Encoding,User-Agent
Cache-Control: no-transform, max-age=86400
Content-Type: application/xml; charset=utf-8
Transfer-Encoding: chunked

<?xml version="1.0" encoding="UTF-8"?>
<Overview xmlns="http://www.deltek.com/ns/webservices/container"
          containerName="expensesheets">
  <Links>
    <Link href="http://server/containers/v1/w16p2/expensesheets/ ↩
    specification"
          rel="specification"/>
    <Link href="http://server/containers/v1/w16p2/expensesheets/filter"
          rel="data:filter"/>
    <Link href="http://server/containers/v1/w16p2/expensesheets/data;any"
          rel="data:any-key"/>
    <Link href="http://server/containers/v1/w16p2/expensesheets/data/card/ ↩
    init"
          rel="action:insert"/>
    <Link href="http://server/containers/v1/w16p2/expensesheets/data/card"
          rel="action:create"/>
  </Links>
</Overview>
```

1.4 curl

This document uses the free `curl` tool for all examples. You can download the tool from: http://curl.haxx.se/.

On Windows, the built-in Command Prompt has poor support for quoting and escaping URLs and other parameters to `curl`. To use the `curl` examples in this document, you must install and use a shell that supports Bash-style quoting and escaping. An easy way to do this is to install Git for Windows, which comes with the Git Bash shell emulator and the `curl` tool: http://msysgit.github.io/

`curl` allows a programmer to make HTTP requests from the command line, and is a very valuable tool when developing client code that interacts with a web service. In this document, `curl` is used to provide working examples for the functionality that document on how to correctly interact with the service.

The full documentation is available from the `curl` website, but the following table lists the options used in this document.

Option	Description
`-i`	Include the HTTP response headers in the output.
`-u username:passwd`	Use the specified username and password as HTTP Basic Authentication credentials.
`-H 'Header: Value'`	Include the specified HTTP request header in the request.
`-d @file`	Make an HTTP `POST` request with the contents of `file` as the request entity.
`-X POST`	Make an HTTP `POST` request. If the `-d` option is not used, the request will have an empty request entity.
`-X DELETE`	Make an HTTP `DELETE` request.

Chapter 2

Basics

2.1 JSON and XML

Every container resource in the Maconomy RESTful web service interface can be represented in a JSON format [1, 2] or an XML format [7].

JSON is a lightweight data interchange format derived from JavaScript. It is widely used in RESTful web services and is prominent in dynamically typed languages such as JavaScript, Ruby, and Python. Mature tooling and library support is also available for Java and .NET languages.

XML is a well-known standardized data interchange format that is heavily used in enterprise software and has mature tooling and library support in Java and .NET languages. It is less widely used in browser applications and dynamically typed languages.

When implementing a client program, choose the appropriate format for the environment where the client program is deployed. Factors that may contribute to the decision are technology stack, corporate or programmer preference, and interoperability with existing code. The formats are functionally equivalent in the sense that they both contain the same data and support the same business functionality.

If no format is specified, the default is JSON.

To request another format, include the `Accept` HTTP request header to explicitly request a particular media type [see 5, section 14.1]. The media types supported in the web service interface are:

Media type	Description
`application/json`	A JSON representation of the resources in the Maconomy RESTful web service interface.

Media type	Description
`application/xml`	An XML representation of the resources in the Maconomy RESTful web service interface.

To update the state of a resource, a client program must send a request entity with the new state of the resource. Like the response entity, the request entity can be in either JSON or XML format. When a client program sends a request entity, it must always specify the media type by including the `Content-Type` HTTP request header.

2.2 Language

You can specify the preferred language by including the `Accept-Language` HTTP request header. As seen earlier, the state of the service endpoint includes a list of supported languages:

```
$ curl -i
       'http://server/containers/v1/w16p2'
HTTP/1.1 200 OK
Date: Fri, 28 Nov 2014 14:51:07 GMT
Server: Jetty(8.1.14.v20131031)
Cache-Control: no-cache, no-transform
Content-Type: application/json; charset=utf-8
Vary: Accept-Encoding,User-Agent
Transfer-Encoding: chunked

...
"languages": [
  {
    "locale": "da_DK",
    "tag": "da-DK",
    "title": "Dansk (Danmark)"
  },
  {
    "locale": "en_US",
    "tag": "en-US",
    "title": "English (United States)"
  }
],
...
```

This system is configured to support two languages. To specify the preferred language for a resource, include the `Accept-Language` HTTP request header with the language tag as the header field value. To get the resource state in US English:

```
$ curl -i
       -u 'Administrator:123456'
       -H 'Accept-Language: en-US'
       'http://server/containers/v1/w16p2/expensesheets/specification'
HTTP/1.1 200 OK
Date: Fri, 28 Nov 2014 14:52:03 GMT
Server: Jetty(8.1.14.v20131031)
Content-Language: en-US
Vary: Content-Language,Accept-Encoding,User-Agent
Cache-Control: no-transform, max-age=86400
Content-Type: application/json; charset=utf-8
Transfer-Encoding: chunked

...
"title": "Expense Sheets"
...
```

To get the resource state in Danish:

```
$ curl -i
       -u 'Administrator:123456'
       -H 'Accept-Language: da-DK'
       'http://server/containers/v1/w16p2/expensesheets/specification'
HTTP/1.1 200 OK
Date: Fri, 28 Nov 2014 14:55:39 GMT
Server: Jetty(8.1.14.v20131031)
Content-Language: da-DK
Vary: Content-Language,Accept-Encoding,User-Agent
Cache-Control: no-transform, max-age=86400
Content-Type: application/json; charset=utf-8
Transfer-Encoding: chunked

...
"title": "Udgiftsedler"
...
```

To unambiguously apply the language preference, it is recommended that client programs always include the `Accept-Language` HTTP request header with all requests. The header field value should be the exact language tag value of one of the supported languages (obtained from the service endpoint resource).

2.3 Specifications

Every container in the Maconomy RESTful web service interface has a *specification* subresource.

The specification is used to programmatically determine the following:

1. The names and titles of the panes in the container
2. The names and titles of the actions supported by each pane
3. The names, titles, and data types of the fields present in records in each pane

To correctly interpret and manipulate records in the panes of a container, a client program must read the specification resource to obtain the field names and data types.

In a previous example we accessed the `ExpenseSheets` container and obtained a link to its specification subresource:

```
"specification": {
  "href": "http://server/containers/v1/w16p2/expensesheets/specification",
  "rel": "specification"
}
```

By looking a the `rel` property and discovering that the relation `specification` is present, a client program can determine that this particular hyperlink points us to the specification resource. The link relation is simply an identifier that tells client programs about the meaning of a particular hyperlink. When writing client programs you should *only* rely on the link relations and consider the links as opaque. You should *not* attempt to guess the pattern for particular kinds of resources because only the link relation is guaranteed to be stable.

If you follow the link, you acquire the specification for the `ExpenseSheets` container:

```
$ curl -i
       -u 'Administrator:123456'
       -H 'Accept-Language: en-US'
       'http://server/containers/v1/w16p2/expensesheets/specification'
HTTP/1.1 200 OK
Date: Fri, 28 Nov 2014 15:00:02 GMT
Server: Jetty(8.1.14.v20131031)
Content-Language: en-US
Vary: Content-Language,Accept-Encoding,User-Agent
Cache-Control: no-transform, max-age=86400
Content-Type: application/json; charset=utf-8
Transfer-Encoding: chunked

{
  "containerName": "expensesheets",
  "panes": {
    "filter": {
      "paneName": "filter",
      "title": "List of Expense Sheets",
      "actions": { ... },
      "fields": { ... }
    },
    "card": {
      "paneName": "card",
```

```
      "title": "Expense Sheets",
      "actions": { ... },
      "fields": { ... }
    },
    "table": {
      "paneName": "table",
      "title": "Expense Sheet Lines",
      "actions": { ... },
      "fields": { ... }
    },
  },
  "relatedContainers": { ... }
}
```

The preceding example shows the high-level structure of the specification resource. If you try this in `curl`, you will notice that the full response is substantially larger. The omitted parts are discussed in the succeeding sections.

This JSON object tells that the `ExpenseSheets` container has 3 panes identified by the names `filter`, `card`, and `table`. It also contains the title of each pane, which is a text appropriate to display in a user interface.

2.3.1 Actions

This is an example of the contents of the `actions` property omitted in the previous example:

```
"table": {
  "paneName": "table",
  "title": "Expense Sheet Lines",
  "actions": {
    "action:create": {
      "rel": "action:create",
      "title": "Save Expense Sheet Line"
    },
    "action:delete": {
      "rel": "action:delete",
      "title": "Delete Expense Sheet Line"
    },
    "action:add": {
      "rel": "action:add",
      "title": "Add Expense Sheet Line"
    },
    ...
```

Each action is represented by an object that contains a `rel` property. The value of this property uniquely identify the action within the container. The object also contains a

`title` property appropriate for display in a user interface. Notice that the `rel` property is used as the key in the `actions` object.

The actions listed in the specification are a gross list. When interacting with a particular expense sheet, the client program determines if an action can be invoked in the current state of the resource by examining whether a hyperlink with the link relation corresponding to the action's `rel` property is present.

2.3.2 Fields

The following is an example of the contents of the `fields` property omitted in the previous example:

```
"table": {
  "paneName": "table",
  "title": "Expense Sheet Lines",
  "actions": { ... },
  "fields": {
    "activitynumber": {
      "autoSubmit": false,
      "create": true,
      "hidden": false,
      "mandatory": false,
      "maxLength": 255,
      "multiLine": false,
      "name": "activitynumber",
      "secret": false,
      "suggestions": "onDemand",
      "title": "Activity No.",
      "type": "string",
      "unfilterable": false,
      "update": true
    },
    ...
    "currency": {
      "autoSubmit": false,
      "create": true,
      "enumType": "CurrencyType",
      "hidden": false,
      "mandatory": true,
      "multiLine": false,
      "name": "currency",
      "secret": false,
      "subtypeContainer": "popup_currencytype",
      "suggestions": "none",
      "title": "Currency",
      "type": "enum",
      "unfilterable": false,
```

```
    "update": true
  },
  ...
```

These are examples of field objects that contain metadata for the `activitynumber` and `currency` fields. The field objects describe the fields that will be present in records in the `table` pane in `ExpenseSheets` container. A very important property of a field object is `type`. The value of `type` determines how client programs must interpret and represent values for that field when interacting with records in the `table` pane of the container. The specifics of each format are detailed in the Data Types section.

The following table provides a description of each of the properties of a field object in a specification:

Property	Description
`name`	The identifier used to refer to the field in representations. This is intended for use by the software, and is normally not visible in a user interface.
`title`	The human-readable name for the field. The title is an appropriate label for the field in a user interface.
`type`	The data type of the field. The data type is one of: `integer`, `real`, `amount`, `boolean`, `string`, `date`, `time`, `enum`, `timeduration`, or `autotimestamp`. See the section on Data Types for a description of each data type.
`subtypeContainer`	This property is defined for fields that have the `enum` data type, and it contains the name of the container that supplies the possible values for this particular `enum` type. For example, the field `currency` in the preceding example has the container `popup_currencytype` as the source of its enum values. To find an appropriate link to this container, client programs should go through the `relatedContainers` (see the following).
`enumType`	This property is defined for fields that have the `enum` data type and it contains the name of the enumeration type. This value may be used when client programs need to construct expressions used in filter restrictions, for example.
`create`	This property defines whether the field is editable when a record is created. If this value is false, a client program is not permitted to change the value of this field in the template record obtained from the `init` operation.
`update`	This property defines whether the field can be updated after the record is created. Some fields are immutable after the record is created in the system.

Property	Description
`hidden`	This property indicates that a field is part of the protocol between the client and server, but it should not be visible in a user interface.
`secret`	This property indicates that the contents of the field must not be displayed unmasked in a user interface (for example, a password).
`unfilterable`	This property indicates that the field must not be used as part of a filter restriction.
`autoSubmit`	This property indicates to the client that it should automatically update the resource when a user finishes editing this field.
`suggestions`	This property indicates how the client should present inline searches from this field in a user interface. The value can be one of: `onDemand`, `automatic`, `none`, or `standard`. `onDemand` indicates inline search on demand. `automatic` indicates a search-as-you-type style inline search. `none` indicates no inline search. `standard` indicates that the client program should apply its own preferred default, and use the behavior of either `onDemand`, `automatic`, or `none`.

2.3.3 Related Containers

The `relatedContainers` property of a container consists of references to other related containers that are related to this container. Currently, only containers that supply enum values for enum types used in the container that owns the `relatedContainers` property are referenced.

In the previous example, the `currency` field in the `table` pane has `popup_currencytype` as its subtype container. The `relatedContainers` contains a reference that the client program can use to obtain the possible values for this field:

```
"relatedContainers": {
  ...
  "popup_currencytype": {
    "containerName": "popup_currencytype",
    "links": {
      "data:enumvalues": {
        "href": "http://server/containers/v1/w16p2/popup_currencytype/filter ↩
    ",
        "rel": "data:enumvalues"
      },
```

```
      "specification": {
        "href": "http://server/containers/v1/w16p2/popup_currencytype/ ↩
    specification",
        "rel": "specification"
      }
    }
  },
  ...
}
```

2.4 Data Types

Maconomy uses eight primitive data types. In the container resources in the web service interface these data types are embedded in XML and JSON documents and are encoded in a locale independent way.

The XML and JSON representations of a resource differ slightly from each other. All attributes in XML are quoted strings, while JSON permits numeric types that are not quoted.

Several Maconomy data types use the `number` grammar rule of the JSON data interchange format [2]. For reference the `number` grammar rule is defined as [10]:

```
number = [ minus ] int [ frac ] [ exp ]
decimal-point = %x2E          ; .
digit1-9 = %x31-39            ; 1-9
e = %x65 / %x45               ; e E
exp = e [ minus / plus ] 1*DIGIT
frac = decimal-point 1*DIGIT
int = zero / ( digit1-9 *DIGIT )
minus = %x2D                  ; -
plus = %x2B                   ; +
zero = %x30                   ; 0
```

2.4.1 Integer

The integer data type consists of negative and non-negative integer values: `{..., -1, 0, 1, ...}`.

JSON Integer values are represented as a JSON number that must conform to the `number` grammar rule [2] with the additional restriction that the number must be an integer. Integers *should not* include a fraction or exponent part. Numbers *may* be accepted if they include a fraction and/or exponent part as long as they are integers. Examples of acceptable values are `1000` and `-549`.

XML Integer-valued attributes contain numbers that may start with an optional sign (`-` or `+`) and must otherwise consist of one or more decimal digits. Examples of acceptable values are `"1000"`, `"-549"`.

2.4.2 Real

The real data type is a floating point data type.

JSON Real values are encoded as JSON numbers. Values must conform to the `number` grammar rule [2]. Examples of acceptable values are `100`, `.892`, `2e10`, and `314159e-5`.

XML Real-valued attributes contain numbers encoded similar to JSON numbers. The contents of the attribute value must conform to the `number` grammar rule in the JSON syntax [see 2]. Examples of acceptable values are `"100"`, `".892"`, `"2e10"`, and `"314159e-5"`.

2.4.3 Amount

The amount data type is used to represent monetary values as a number of hundredths (cents).

JSON Amount values are encoded as integers that represent the number of hundredths in the amount value. The restrictions and recommendations for encoding integers in JSON also apply to amounts. Examples of acceptable values are `0`, `1000`, `-5795`.

XML Amount-valued attributes contain numbers that start with an optional sign (`-` or `+`), and must otherwise consist of zero or more decimal digits followed by a decimal point and two digits. Examples of acceptable values are `"0.00"`, `"10.00"`, `"-57.95"`.

2.4.4 Boolean

The boolean data type consists of the values `true` and `false`.

JSON Booleans are represented as the JSON values `true` and `false`.

XML Boolean-valued attributes have `"true"` and `"false"` as acceptable values.

2.4.5 String

The string data type is used to represent text. The character set used is determined by the enclosing JSON or XML document, and may be indicated in the `Content-Type` header.

UTF-8 is the default and *should* be used for both JSON and XML. Note that Unicode characters may be escaped using the `\uXXXX` where `X` is a hexadecimal digit.

JSON String values are represented as JSON string values and must conform to the `string` grammar rule [2]. Examples of acceptable values are `""` and `"Hello world"`.

XML String-valued attributes are a standard XML quoted attribute value. Examples of acceptable values are `""` and `"Hello world"`.

2.4.6 Date

The date data type is used to represent a date that is composed of the year, month, and day.

Both the JSON and XML representations use the following date format: `YYYY-MM-DD`. `YYYY` is the year (for example, `2014`). `MM` is the month (`01` is january, `02` is february, ..., `12` is december). `DD` is the day of the month (`01`, `02`, ..., `31`). In addition to conforming to the format, a date value must be a valid date in the Gregorian calendar.

The date data type also has a special *null* data value that is represented as an empty string.

JSON Date values are represented as a JSON string [2] whose contents conform to the date format described in the previous section. Examples of acceptable values are: `""`, `"1950-04-05"`, `"1945-04-25"`, `"1946-12-16"`, and `"1945-11-15"`.

XML Date-valued attributes contain values that conform to the date format described in the previous section. Examples of acceptable values are: `""`, `"1950-04-05"`, `"1945-04-25"`, `"1946-12-16"`, and `"1945-11-15"`.

2.4.7 Time

The time data type is used to represent a time that is composed of hour, minutes, and seconds.

Both the JSON and XML representations use the following time format: `hh:mm:ss` where `hh` is the hour (`00`, `01`, ..., `23`), `mm` are the minutes (`00`, `01`, ..., `59`) and `ss` are the seconds (`00`, `01`, ..., `59`).

The time data type also has a special *null* data value that is represented as an empty string.

JSON Time values are represented as a JSON string [2] whose contents conform to the time format described in the previous section. Examples of acceptable values are: `""`, `"10:59:23"`, and `"19:21:49"`.

XML Time-valued attributes contains values that conform to the time format described in the previous section. Examples of acceptable values are: `""`, `"10:59:23"`, and `"19:21:49"`.

2.4.8 Enum

The enum data type (also called *popup types* in Maconomy) is a class of types. Each particular enum type has a list of possible values. One example of an enum type is `CountryType`, where the possible values are the countries available in the system. In other contexts (for example, expressions), enum values are usually written using the notation `PopupType'PopupLiteral`. In the JSON and XML representations, only the *enum literal* value is used to avoid the need to parse the enum notation client-side. For example, the value `CountryType'Norway` is encoded as the literal string `"norway"`.

All enum types have a special *nil* enum value which is represented as the string `"nil"`.

JSON Enum values are represented as a JSON string [2] that contain the enum literal value of a valid value of the enum type of the field.

XML Enum-valued attributes contain an enum literal value of a valid value of the enum type of the field.

2.4.9 Time Duration

The time duration data type is a special-purpose variant of the real data type. It has the same JSON and XML representation as the real data type, but it specifically represents a time duration and should be formatted accordingly by client programs if the value is to be presented in a user interface, print, or similar context.

2.4.10 Auto Timestamp

The auto timestamp datatype is a special-purpose variant of the string data type. It has the same JSON and XML representation as the string data type.

2.5 Structures

Client programs must interact with two basic structures used to encode resource states:

1. A container resource state
2. A record resource state

2.5.1 Container Resource State

A container resource state encodes the state of a resource in a container. The resource state includes the state of all panes and the data records within each pane.

For example, if you have at least one expense sheet in our system, you can use the `data:any-key` link from the `ExpenseSheets` container that links to an unspecified expense sheet:

```
"data:any-key": {
  "href": "http://server/containers/v1/w16p2/expensesheets/data;any",
  "rel": "data:any-key"
},
```

This kind of link is normally not very useful. A client program typically wants to interact with a specific expense sheet, rather than any expense sheet in the system. To obtain a link to a particular expense sheet, use the filter resource to search for the expense sheet. However, to examine the general structure of a container resource state, you can use any expense sheet:

```
$ curl -u 'Administrator:123456'
       -H 'Accept-Language: en-US'
       'http://server/containers/v1/w16p2/expensesheets/data;any'
{
  "meta": {
    "containerName": "expensesheets"
  },
  "links": {
    "self": {
      "href": "http://server/containers/v1/w16p2/expensesheets/data; ↩
    expensesheetnumber=10760016",
      "rel": "self"
    }
  },
  "panes": {
    "card": {
      "meta": {
        "concurrencyControl": "",
        "paneName": "card",
        "rowCount": 1,
        "rowOffset": 0
      },
      "links": { ... },
      "records": [ ... ]
    },
    "table": {
      "meta": {
        "concurrencyControl": "\"card\"=\"2 ↩
    e9af88884dcd0a1de7b0f6ad2c41afee1891c7b\"",
```

```
        "paneName": "table",
        "rowCount": 1,
        "rowOffset": 0
      },
      "links": { ... },
      "records": [ ... ]
    }
  }
}
```

The preceding example gives you an idea of the overall structure of a resource state, which is composed of three levels:

1. The first level is the container resource state
2. The second level consists of a number of panes in the container resource state
3. The third level consists of a number of records in each pane

The objects at each of these three levels have the `meta` and `links` properties.

The `meta` property contains metadata about the object. For example, the `meta` object for a container resource state includes the `containerName`, while the `meta` object for a pane object contain the `paneName` as well as other metadata. By convention, the `paneName` for a filter pane is `filter`, `card` for a card pane, and `table` for a table pane.

The `links` property contains hyperlinks used to manipulate that particular object.

In the preceding example, the container resource state contains the very important `self` link that uniquely identifies *this* resource (in this case, a particular expense sheet). Recall that an *any key* link was used to obtain an unspecified expense sheet. The `self` link provides a stable link to this particular expense sheet so that you can find *this* expense sheet even if the *any key* link should point to another expense sheet in the future. A client program should use the self link as its local identifier for a resource state, and whenever it receives a new state for a particular resource, it should replace any existing local copy with the new one.

In the same way, the `links` property in a pane object contains links that can operate on the pane object. For an example of links for the `table` pane, refer to the following:

```
"links": {
  "action:create": {
    "href": "http://server/containers/v1/w16p2/expensesheets/data; ↩
    expensesheetnumber=10760016/table",
    "rel": "action:create"
  },
  "action:add": {
    "href": "http://server/containers/v1/w16p2/expensesheets/data; ↩
    expensesheetnumber=10760016/table/init",
    "rel": "action:add"
  }
```

```
},
```

The `table` pane object contains links that allow you to create rows in the table.

2.5.2 Records

Each pane contains zero or more records in the `records` list. Look at the record in the preceding `card` pane:

```
{
  "meta": {
    "concurrencyControl": "\"card\"=\"2 ↩
    e9af88884dcd0a1de7b0f6ad2c41afee1891c7b\"",
    "rowNumber": 0
  }
  "links": {
    ...
  },
  "data": {
    "accountnumbervar": "",
    "amountbase": 0,
    "amountenterprise": 0,
    ...
  },
}
```

The record object also has `meta` and `links` properties. The `meta` object contains the `rowNumber` of the record. The `links` property contains various hyperlinks that represent state transitions available for the particular record, for example, update fields, delete the record, or run an application action.

The `data` property contains an object with each of the field values in the record.

State transitions such as updating, deleting, and application actions occur in the context of a particular record. When you update data, you interact with the record subresource. You must use the record structure as the request entity when, for example, updating data. When sending a record structure as a request entity, a client program may omit the `meta` and `links` properties as well as any untouched fields in the `data` object. The following is a valid record structure for updating the value of the `description` field:

```
{
  "data": {
    "description": "New description"
  }
}
```

Note that even though updates, actions, and so on, happen on a single record, the response entity is always a copy of the full container resource state.

2.6 Hyperlinks

Hyperlinks are used for two purposes in a web service:

1. Referencing related resources

   On websites such as Wikipedia one document, such as an article, may contain references to other related documents. A link can have one or more of various relations to the context resource: it may link to another article, a web page used as source, or something else. Similarly, hyperlinks in a web service contain links to other resources. An Expense Sheet can, for example, contain a hyperlink to the employee who owns the expensesheet.

2. State transitions

   On some websites, using hyperlinks can change the state of the page. On an online store such as Amazon, hyperlinks are also used to change the state of a resource, the shopping basket. When a user clicks the Add to Basket hyperlink, it performs a state transition on the shopping basket resource (adding an item). If the user later clicks the Delete link, another state transition occurs (removing an item). The available links represent the available state transitions. If the shopping basket is empty, there will not be a Delete link. The same principles are used in web services. An expense sheet may, for example, have links to submit the expense sheet for approval. If a client program interacts with a submitted expense sheet, it may have a link that can be used to reopen the expense sheet for further entries.

2.6.1 Link Relations

On a web page the title of the link communicates its purpose. In a web service the purpose of a link is communicated to client programs via the link relation. The link relation [9] defines the relationship between the context resource (the resource that contains the link) and the target resource. In other words, link relations are simple keywords that identify the purpose of a hyperlink.

`self` The `self` link relation indicates a hyperlink to the context resource. This is useful when a client program interacts with one resource and the web service responds with the state of another resource.

`specification` Indicates a reference to the specification resource for the context resource (a container).

`file` Indicates a reference to a file that is produced as part of handling the request.

`data:filter` Indicates a reference to a container filter resource that can be used to search for specific resources within the container. If, for example, a client program

wants to display and interact with a particular expense sheet, it uses the filter to find the link to the resource state of that particular expense sheet.

`data:enumvalues` Indicates a reference to a container resource state that provides the possible values of an enumeration type.

`data:any-key` Indicates a reference to a container resource state that is identified by "any" key. This is often not very useful, since a client program often needs to interact with a specific resource rather than just any resource. To find a reference to the specific resource, a client program must use the filter resource to search for the required resource. However, in some situations, notably in singleton containers that conceptually contain exactly one record for each user, this is the only way to access the resource state.

`data:same-key` Indicates a reference to a container resource state that is identified by the same key as the context resource. This kind of link occurs when a record in a filter contains a link to the full resource state for that particular resource. A record in the filter pane of the `ExpenseSheets` container links to the full resource state of that particular expense sheet.

`action:insert` Indicates a link that is used to perform the *initialize* state transition in the *insert* variant. This resource computes a template to be used when creating a record. The template record is pre-filled with the default value for each field in the record. The insert variant is significant in a table pane, where the new record will be inserted at the position of the record that contains the hyperlink. Client programs must use the `POST` method with no request entity to perform this state transition.

`action:add` Indicates a link that is used to perform the *initialize* state transition in the *add* variant. This works like the insert variant described previously, but in a table pane the new record is added at the end of the table. Client programs must use the `POST` method with no request entity to perform this state transition.

`action:create` Indicates a link that is used to perform the *create* state transition. This creates a record in a pane. In a card pane, for example, in the `ExpenseSheets` container, this creates an expense sheet. In a table pane it creates a row in the table, for example another line on the expense sheet. Client programs must use the `POST` method with a record structure as the request entity.

`action:read` Indicates a link that is used to perform the *read* state transition. This obtains a fresh copy of the current resource state. This maps naturally to the HTTP `GET` method.

`action:update` Indicates a link that is used to perform the *update* state transition. This state transition changes the values of one or more fields in a record. Client programs must use the `POST` method with a record structure as the request entity.

`action:delete` Indicates a link that is used to perform the *delete* state transition. This

state transition deletes a record. In a card pane, for example, in the `ExpenseSheets` container, this deletes the expense sheet including any expense sheet lines. In a table pane this deletes a row in the table, such as an expense sheet line. Client programs must use the HTTP `DELETE` method.

`action:print` Indicates a link that is used to perform the *print* state transition. This state transition produces a print from the resource state. Client programs must use the `POST` method with no request entity. A link to the resulting print is included as an HTTP response header field as described in Running an Action and Downloading a Resulting File.

`action:...` Indicates a link that is used to perform an *action* state transition. Actions other than the previously described must be invoked using the HTTP `POST` method with no request entity. The actual set of supported action state transitions must be obtained by client programs from the specification. An expense sheet may, for example, support submitting the expense sheet for approval. The link relation for that particular action is: `action:submitexpensesheet`

2.7 Authentication

Each request must be authenticated by using HTTP Basic Authentication [6].

This method of authentication entails transmitting the username and password on each request, and in itself offers very weak protection of the user credentials. To be secure, the Maconomy web services *must* be deployed behind an SSL/TLS termination proxy that encrypts the traffic between the server and the client. If an SSL/TLS termination proxy is not deployed, the user credentials sent to the Maconomy web services are vulnerable to stealing by an attacker.

Note that Franks et al. [6] implicitly requires the credentials to be encoded as `ISO-8859-1` by using the `TEXT` grammar rule defined in Fielding et al. [5]. However, most (but not all) modern browsers encode the credentials as `UTF-8`. The Maconomy RESTful web service interface follows the modern convention and requires user credentials to be `UTF-8` encoded. This allows a wider range of special characters to appear in usernames and passwords.

2.7.1 HTTP Basic Authentication

HTTP client library code normally has the ability to send HTTP Basic Authentication credentials to the server. For completeness, the following describes the simple, underlying mechanism.

When a client program tries to interact with a resource that requires authentication, the server responds with a status of `401 Unauthorized`:

```
$ curl -i
       -H 'Accept-Language: en-US'
       'http://server/containers/v1/w16p2/ExpenseSheets'
HTTP/1.1 401 Unauthorized
Date: Fri, 28 Nov 2014 15:05:38 GMT
Server: Jetty(8.1.14.v20131031)
Content-Type: application/json; charset=utf-8
WWW-Authenticate: Basic realm="Maconomy"
Vary: Accept-Charset,Accept
Transfer-Encoding: chunked

{
  "errorFamily": "service",
  "errorMessage": "The request requires user authentication",
  "errorSeverity": "error"
}
```

To resolve this a client program must look at the `WWW-Authenticate` HTTP response header to see the method of authentication that the server requires. The token `Basic` in the header value in the preceding example indicates that the server requires the client to use HTTP Basic Authentication. The client must construct HTTP Basic Authentication credentials and retry the request.

The following is an example of a simple Python program that illustrates how to compute the credentials:

```
username = u"Administrator"
password = u"123456"

# 1. Combine the username and password separated by colon
combined = username + ":" + password

# 2. Encode the string into UTF-8 yielding sequence of bytes
utf8_bytes = combined.encode("utf-8")

# 3. Encode the byte sequence into Base64
base64_chars = base64.b64encode(utf8_bytes)

# 4. Prepend the result with the string "Basic " to indicate the ↩
    authentication method
authorization = "Basic " + base64_chars
```

In this example the `authorization` string has the value `Basic QWRtaW5pc3RyYXRvcjoxMjM0NTY=`. The client program must retry the request and supply this value in the `Authorization` HTTP request header:

```
$ curl -i
       -H 'Authorization: Basic QWRtaW5pc3RyYXRvcjoxMjM0NTY='
       -H 'Accept-Language: en-US'
```

```
       'http://server/containers/v1/w16p2/ExpenseSheets'
HTTP/1.1 200 OK
Date: Fri, 28 Nov 2014 15:06:06 GMT
Server: Jetty(8.1.14.v20131031)
Content-Language: en-US
Vary: Content-Language,Accept-Encoding,User-Agent
Cache-Control: no-transform, max-age=86400
Content-Type: application/json; charset=utf-8
Transfer-Encoding: chunked

...
```

Note that while encoding the string as Base64 masks the password, it is trivially reversible and completely insecure in itself. That is why the web service must be deployed behind an SSL termination proxy to be secure.

2.7.2 Suppressing the Browser's Login Prompt

Client programs that run in a web browser by default get the browser's native login prompt when the web service requires authentication. The reason is that the web service signals that it needs authentication via the `Basic` authentication scheme by sending the `WWW-Authenticate` HTTP response header:

```
$ curl -i
       -H 'Accept-Language: en-US'
       'http://server/containers/v1/w16p2/ExpenseSheets'
HTTP/1.1 401 Unauthorized
Date: Fri, 28 Nov 2014 15:08:05 GMT
Server: Jetty(8.1.14.v20131031)
Content-Type: application/json; charset=utf-8
WWW-Authenticate: Basic realm="Maconomy"
...
```

When the web browser detects this response header, it automatically intercepts the response and shows its native login prompt.

If a browser-based client program prefers to handle logins itself using a web UI instead of the native login prompt, it must include the custom HTTP request header `Maconomy-Authentication` and set its value to `X-Basic`. This causes the server to modify its subsequent `WWW-Authenticate` challenge to advertise the `X-Basic` authentication scheme rather than the `Basic` authentication scheme:

```
$ curl -i
       -H 'Maconomy-Authentication: X-Basic'
       -H 'Accept-Language: en-US'
       'http://server/containers/v1/w16p2/ExpenseSheets'
HTTP/1.1 401 Unauthorized
```

```
Date: Fri, 28 Nov 2014 15:07:28 GMT
Server: Jetty(8.1.14.v20131031)
Content-Type: application/json; charset=utf-8
WWW-Authenticate: X-Basic realm="Maconomy"
...
```

Note that the client program must use the `Basic` authentication scheme, rather than `X-Basic`, when it supplies the username and password via the `Authorization` HTTP request header.

2.7.3 Expired User Passwords

If the user's password has expired a request fails with a `401 Unauthorized` status:

```
$ curl -i
       -u 'Anders Hansen:123456'
       -H 'Accept-Language: en-US'
       'http://server/containers/v1/w16p2/ExpenseSheets'
HTTP/1.1 401 Unauthorized
Date: Fri, 28 Nov 2014 15:11:50 GMT
Server: Jetty(8.1.14.v20131031)
Content-Language: en-US
Vary: Content-Language,Accept-Charset,Accept
Content-Type: application/json; charset=utf-8
WWW-Authenticate: X-ChangePassword realm="Maconomy"
Transfer-Encoding: chunked

{
  "errorFamily": "service",
  "errorMessage": "The password has expired. Please enter a new password.",
  "errorSeverity": "error"
}
```

In this situation the server offers a custom authentication method `X-ChangePassword`. This method authenticates the request and change the user's password. The credentials are computed in a similar way to the standard HTTP Basic Authentication described previously.

```
username     = u"Anders Hansen"
old_password = u"123456"
new_password = u"MyNewPassword"

# 1. Combine the username, old password and new password with the required ↩
    separators
combined = username + ":" + old_password + "\n" + new_password

# 2. Encode the string into UTF-8 yielding sequence of bytes
utf8_bytes = combined.encode("utf-8")
```

```
# 3. Encode the byte sequence into Base64
base64_chars = base64.b64encode(utf8_bytes)

# 4. Prepend the result with the string "X-ChangePassword " to indicate the ↩
    authentication method
authorization = "X-ChangePassword " + base64_chars
```

The difference here is that the combined credentials are appended with a single line feed character followed by the new password. The line feed character is usually written as `\n` in string literals in programming languages. The token that indicates the authentication method is `X-ChangePassword`, rather than `Basic`.

With this in place the client program can resolve this situation by letting the user change the password by retrying the request with the resulting credentials:

```
$ curl -i
       -H 'Authorization: X-ChangePassword ↩
    QW5kZXJzIEhhbnNlbjoxMjM0NTYKTXlOZXdQYXNzd29yZA=='
       -H 'Accept-Language: en-US'
       'http://server/containers/v1/w16p2/ExpenseSheets'
HTTP/1.1 200 OK
Date: Fri, 28 Nov 2014 15:13:52 GMT
Server: Jetty(8.1.14.v20131031)
Content-Language: en-US
Vary: Content-Language,Accept-Encoding,User-Agent
Cache-Control: no-transform, max-age=86400
Content-Type: application/json; charset=utf-8
Transfer-Encoding: chunked
...
```

The user's password is now changed to `MyNewPassword`, and the client program should use regular HTTP Basic Authentication for the following requests.

Note that the `X-ChangePassword` authentication method may be used at any time to allow a user to change the password.

2.8 Status Codes and Errors

Each response from the web service has an HTTP status code. The status code tells whether the request was successful. If the request was unsuccessful the status code indicates what kind of failure occurred which is used by client programs to decide how to proceed.

Most people have encountered the `404 Not Found` status code when browsing the web. The three-digit integer status code (`404`) is the significant part that client programs use to categorize the error. The text (`Not Found`) is called the *reason phrase* and is intended

for humans to help explain the error. The numeric status code is standardized and has a particular meaning, while the reason phrase may differ in various ways (it may differ between web server software, it may be localized, and so on).

Status codes are categorized into *families*, where the family is indicated by the first digit of the status code. The important ones for the Maconomy RESTful web service interface are:

Status Codes	Family	Explanation
`1xx`	Informational	Request received, continuing process. This family is not used in the Maconomy RESTful web service interface.
`2xx`	Success	The action was successfully received, understood, and accepted.
`3xx`	Redirect	Further action must be taken to complete the request.
`4xx`	Client Error	The request contains bad syntax or cannot be fulfilled.
`5xx`	Server Error	The server failed to fulfill an apparently valid request.

The following is a list of the status codes that are used in the Maconomy RESTful web service interface along with their meanings:

Status Code	Reason phrase	Explanation
`200`	`OK`	The request has succeeded. If the request is a `GET` request the response is a representation of the requested resource. If the request is a `POST` or `DELETE` request the response may be the representation of the resource that was affected by the request.
`400`	`Bad Request`	The request entity or request headers contained malformed or incomplete information. This usually indicates a programming error in the client program.
`401`	`Unauthorized`	The request requires user authentication. The client program must retry the request with valid HTTP Basic Authentication credentials.
`403`	`Forbidden`	The requested resource or action is not permitted with the supplied credentials.

Status Code	Reason phrase	Explanation
404	`Not Found`	The requested resource was not found. It may or may not have existed at an earlier point in time and was subsequently deleted by another user.
405	`Method Not Allowed`	The HTTP method is not allowed for the resource. For example, a resource may not support `GET`, `POST`, or `DELETE`.
406	`Not Acceptable`	The resource cannot be represented in the media type specified in the `Accept` request header.
408	`Request Timeout`	The client did not produce a request within the time that the server was prepared to wait. The client may retry the request.
409	`Conflict`	The request could not be completed because of a conflict with the current state of the resource. This indicates that the resource was updated by another user. The client may refresh its current state of the resource and retry the request.
413	`Request Entity Too Large`	The request entity was larger than the maximum size supported by the server.
414	`Request-URI Too Long`	The request URI/URL was larger than the maximum length supported by the server.
415	`Unsupported Media Type`	The server does not support the media type specified in the `Content-Type` request header.
422	`Application Error`	The request could not be completed because it violated application business logic.
500	`Internal Server Error`	A catch-all status code for unexpected errors.

Fielding et al. [5] contains a detailed specification of the semantics of each of the status codes, except for `422 Application Error`, which is adopted from Dusseault [4].

Note that when the Maconomy RESTful web services are deployed behind an HTTP reverse proxy, the proxy server may use additional status codes. The status code `503 Service Unavailable` may, for example, be used to indicate that the Maconomy system is unreachable.

2.8.1 Error Response Entities

When an error occurs, the HTTP status code is typically used by client programs to dispatch to error handling code that is appropriate for that particular type of error. What is appropriate depends on the nature of the client program, but in many cases it makes sense to log or display an error message. The response entity for an unsuccessful request has an error message and metadata that can be used to signal the error.

The following example illustrates what happens if you try to register 30 hours on a Monday on a time sheet:

```
$ curl -i
       -u 'Administrator:123456'
       -H 'Accept-Language: en-US'
       -H 'Maconomy-Concurrency-Control: "card"=" ↩
    b666368b8fb209c3e1e98dde6f7d39b3f5797281", "table"=" ↩
    a66cc1b06fdcc3275e4cccdd796c4d39927b2ff6"'
       -H 'Content-Type: application/json'
       -d '{ "data" : { "numberday1" : 30.0 } }'
       'http://server/containers/v1/w16p2/timesheets/data;employeenumber=11; ↩
    periodstart=2012-05-28/table/0'
HTTP/1.1 422 Unprocessable Entity
Date: Fri, 28 Nov 2014 15:17:32 GMT
Server: Jetty(8.1.14.v20131031)
Content-Language: en-US
Vary: Content-Language,Accept-Charset,Accept
Cache-Control: no-cache, no-transform
Content-Type: application/json; charset=utf-8
Transfer-Encoding: chunked

{
  "errorFamily": "application",
  "errorMessage": "The employee's maximum hours are exceeded Monday",
  "errorSeverity": "error",
  "focus": {
    "fieldName": "numberday1",
    "paneName": "table",
    "rowNumber": 0
  }
}
```

The HTTP status code `422 Application Error` indicates that the request was unsuccessful because of an application error.

The `errorMessage` property contains an error message that is appropriate to display to a user or report in the way that is appropriate for the client program.

The `errorFamily` property describes what kind of error occurred. The possible values are:

Family	Explanation
`application`	An application error indicates that the request was unsuccessful because it violated business logic in the Maconomy system.
`service`	A service error indicates a technical problem or other condition that was not caused by the business logic. One example is trying to interact with a record that was changed or deleted by another user.
`internal`	An internal error is an unexpected error that can indicate a problem in the system setup or a bug in the web service. The server log files usually contain a message indicating the underlying cause. An example could be that the database is not running.

The `errorSeverity` indicates the severity of the error. The possible values are:

Family	Explanation
`fatal`	The `fatal` severity indicates an unexpected error where an invariant was violated.
`error`	The `error` severity indicates a regular error condition, for example, a business constraint was violated.
`warning`	The `warning` severity indicates a warning to the user about a potential problem.

The `focus` property may be present if the error relates to a particular field. It indicates to client programs to put the focus in the field to help the user identify the cause of the error. The property `paneName` indicates in which pane the field is located, the `rowNumber` indicates which row the offending value is found, and the `fieldName` indicates which field is related to the error.

2.8.2 Warnings and Notifications

Maconomy may raise warnings and notification messages. These are included in the HTTP response header fields `Maconomy-Warning` and `Maconomy-Notification`. These HTTP response header fields can appear multiple times.

In Maconomy a warning reports a message to the client and allows the user to continue or abort the operation. In traditional Maconomy clients this is implemented by a synchronous callback where the server waits for the human user or client program to continue or abort. This protocol is not naturally encoded in an HTTP-based interface and the containers in the Maconomy RESTful web service interface automatically accept

any warnings and include the messages of the accepted warnings in the HTTP response header.

2.9 Compression

The web services support `gzip` compression via the standard HTTP mechanism [see 5, section 14.4]. The client program includes the `Accept-Encoding` HTTP request header to indicate to the server which kinds of compression it supports. If the client includes the `gzip` encoding, the server compresses the entity in the response. HTTP client library code normally supports this transparently. In `curl` this is enabled by using the `--compress` option.

Chapter 3

Filtering

A previous example obtained an expense sheet by using the *any key* hyperlink. This provided the resource state of some unspecified expense sheet in the system. A client program typically wants to interact with a specific expense sheet rather than any expense sheet in the system. The way to obtain a link to a particular expense sheet is to use the filter resource to search for the expense sheet.

The container resource obtained earlier contains a link to a filter resource:

```
"data:filter": {
  "href": "http://server/containers/v1/w16p2/expensesheets/filter",
  "rel": "data:filter"
},
```

You can use that to search for expense sheets in the system:

```
$ curl -u 'Administrator:123456'
       -H 'Accept-Language: en-US'
       'http://server/containers/v1/w16p2/expensesheets/filter'
{
  "meta": {
    "containerName": "expensesheets"
  },
  "links": {
    "self": {
      "href": "http://server/containers/v1/w16p2/expensesheets/filter",
      "rel": "self"
    }
  },
  "panes": {
    "filter": {
      "meta": {
        "paneName": "filter",
        "rowCount": 25,
```

```
        "rowOffset": 0
      },
      "links": {},
      "records": [ ... ]
    }
  }
}
```

The filter resource has the same structure as any other resource state, but it supports additional features that are relevant for searching for resources:

- Paging
- Sorting
- Selecting fields
- Restrictions

Look at the first record in the filter:

```
{
  "meta": {
    "rowNumber": 0
  },
  "links": {
    "data:same-key": {
      "href": "http://server/containers/v1/w16p2/expensesheets/data; ↩
    expensesheetnumber=10760001",
      "rel": "data:same-key"
    }
  },
  "data": {
    "amountbase": 8200,
    "amountenterprise": 1100,
    "approvaldate": "",
    ...
    "expensesheetnumber": 10760001,
    ...
  }
}
```

This record contains fields that are appropriate to display in a user interface where the user is searching for a particular expense sheet. The `links` object for this record contains a link that points to this particular expense sheet resource, indicated by the `data:same-key` link relation. A client program must follow this link to interact with the expense sheet.

3.1 Paging

You may notice that you get exactly 25 records in the filter even though the system contains more than 25 expense sheets. This is because the filter resource splits the results into pages. Two query parameters control the paging.

Query Parameter	Function
`limit`	Defines the maximum number of records the filter will contain. The default value is `25`.
`offset`	Skips the first `offset` results. The default value is `0`.

The following provide examples:

`.../expensesheets/filter` Using the default values for `limit` (`25`) and `offset` (`0`): Find up to 25 records, starting from zero.

`.../expensesheets/filter?limit=25&offset=0` With explicit values for `limit` and `offset`: Find up to 25 records, starting from zero.

`.../expensesheets/filter?limit=25&offset=25` Find up to 25 records, starting from record number 25.

`.../expensesheets/filter?limit=11&offset=8` Find up to 11 records, starting from record number 8.

3.2 Sorting

The filter resource supports changing the sort order of the results in the filter. This is controlled by the `orderby` query parameter.

The `orderby` parameter takes a comma-separated list of field names in the filter. For example:

`.../expensesheets/filter?orderby=DateSubmitted` Sort using the field `DateSubmitted` in ascending order.

`.../expensesheets/filter?orderby=DateSubmitted,EmployeeName` Sort using the field `DateSubmitted` in ascending order. Records having the same `DateSubmitted` are then sorted by `EmployeeName` in ascending order.

If you provide only a a field name, it sorts in ascending order. You can also prefix the field name with `+` to indicate ascending order, or `-` to indicate descending order:

`.../expensesheets/filter?orderby=+DateSubmitted` Sort using the field `DateSubmitted` in ascending order.

`.../expensesheets/filter?orderby=-DateSubmitted` Sort using the field `DateSubmitted` in descending order.

`.../expensesheets/filter?orderby=-DateSubmitted,EmployeeName` Sort using the field `DateSubmitted` in descending order. Records having the same `DateSubmitted` are then sorted by `EmployeeName` in ascending order.

`.../expensesheets/filter?orderby=-DateSubmitted,+EmployeeName` This is the same as the preceding example above: Sort using the field `DateSubmitted` in descending order. Records having the same `DateSubmitted` are then sorted by `EmployeeName` in ascending order.

3.3 Selecting Fields

The filter resource supports fetching a subset of the available fields. This is controlled by the `fields` query parameter that takes a comma-separated list of fields. If the `fields` query parameter is not used, all available fields are included in the response. Note that key fields are always included in the response.

For example, if you get `.../expensesheets/filter?fields=EmployeeName,Description`, it returns these records:

```
...
{
  "data": {
    "description": "Meals, working weekend",
    "employeename": "J\u00f8rgen Jansen",
    "expensesheetnumber": 10760001
  },
  "links": { ... },
  "meta": {
    "rowNumber": 0
  }
},
{
  "data": {
    "description": "Conference",
    "employeename": "J\u00f8rgen Jansen",
    "expensesheetnumber": 10760012
  },
  "links": { ... },
  "meta": {
    "rowNumber": 1
  }
```

```
},
{
  "data": {
    "description": "Customer Visit",
    "employeename": "J\u00f8rgen Jansen",
    "expensesheetnumber": 10760014
  },
  "links": { ... },
  "meta": {
    "rowNumber": 2
  }
},
...
```

This explicitly selects the fields `description` and `employeename`. The field `expensesheetnumber` is a key field and is therefore automatically included in the result.

If you select only fields that are actually used by the client program, performance when searching improves.

3.4 Restrictions

You can select only a subset of the records that satisfies an expression. This works similar to a `WHERE` clause in SQL. This is controlled by the `restriction` query parameter. The syntax used is the Expression Language which is also used in MDML and other XML specification languages in Maconomy. See the MDML Language Reference [3] for a full description of the Expression Language.

Note that this and all other query parameters must be URL-escaped (sometimes called *percent encoded*). This is normally done automatically by the HTTP library code, but when you use the command-line, you might need to perform the conversion first:

`.../expensesheets/filter?restriction=CreatedDate%20>%20date(2014,7,1)` Here the expression `CreateDate > date(2014,7,1)` returns the expense sheets that were created after July 1, 2014.

`.../expensesheets/filter?restriction=Submitted` Here the expression `Submitted` returns the expense sheets that have been submitted for approval.

`.../expensesheets/filter?restriction=Submitted%20and%20EmployeeName%20like%20%22Bob*%22` Here the expression `Submitted and EmployeeName like "Bob*"` returns the expense sheets that have been submitted for approval where the employee's name starts with "Bob".

Chapter 4

Updating Data

The containers in the Maconomy RESTful web service interface support various state transitions for the resources:

- Create a record
- Update a record
- Delete a record
- Run an action (in the context of a record)

4.1 Using the POST Method

All of the preceding examples have just used a `curl` with a URL as a parameter. When you do that you use the `GET` HTTP method. This method is used to get a copy of the state of a resource, but it should not have any effect on the system.

When you need to change the state of a resource or create a resource, you use the `POST` HTTP method. You can use the `POST` method with or without a request entity, depending on what you want to happen. The following are some examples:

- Before creating a record the client program use the *initialize* transition to get a template record structure with default values. The client program must use the `POST` HTTP method without a request entity.
- The client program wants to use the *update* transition to change some of the values in a record. The client program must use the `POST` method with a record structure with the updated data fields as the response entity.
- The client program wants to run an application action. It must use the `POST` method without a request entity.

When a client program uses the `POST` method with a request entity, it must always specify the media type of the request entity by including the `Content-Type` HTTP request header. Recall that the media type is either `application/json` when using JSON format, or `application/xml` when using XML format.

In `curl` the `-X` parameter allows you to set the HTTP method to something other than the default `GET` method.

For example, try the *initialize* transition. Recall that you obtained this link from the `ExpenseSheets` container earlier:

```
"action:insert": {
  "href": "http://server/containers/v1/w16p2/expensesheets/data/card/init",
  "rel": "action:insert"
},
```

When you look at the description of the link relations in this document, you see that you must use the `POST` method with no response entity:

```
$ curl -u 'Administrator:123456'
       -H 'Accept-Language: en-US'
       -X POST
       'http://server/containers/v1/w16p2/expensesheets/data/card/init'
{
  "meta": {
    "concurrencyControl": "",
    "rowNumber": 0
  },
  "links": {
    "action:create": {
      "href": "http://server/containers/v1/w16p2/expensesheets/data/card",
      "rel": "action:create"
    }
  },
  "data": {
    "accountnumbervar": "",
    "amountbase": 0,
    "amountenterprise": 0,
    ...
  }
}
```

The response entity is a template record structure. The template record contains a link that you can use to create the record.

4.2 Concurrency Control

In a system like Maconomy, with support for multiple concurrent users, sometimes more than one user tries to work on the same data concurrently. This can cause a problem known as a lost update. Consider this series of events:

1. Alice fetches the state of a particular job resource and starts editing her local copy
2. Bob fetches the state of the same job resource and starts editing his local copy
3. Alice finishes and sends her updates to the server
4. Sometime later, Bob finishes and sends his updates to the server

In a system without concurrency control, Bob's updates can overwrite Alice's updates with neither Bob nor Alice realizing it.

In Maconomy, Bob's update is rejected and he is informed that the record was changed by another user since he last refreshed his data. The client program must then refresh the data and possibly try the update operation again.

In the containers in the Maconomy RESTful web service interface this is implemented using a concurrency control tag, which can be thought of as a fingerprint of the current resource state. When using the `POST` method, a client program must include the `Maconomy-Concurrency-Control` HTTP request header with the concurrency control tag. To find the correct tag for a particular link, the client program must include the value of the `concurrencyControl` property in the `meta` object on the same level as the `links` object that contains the link. Consider this example from a record in an expense sheet resource state:

```
{
  "meta": {
    "concurrencyControl": "\"card\"=\"13 ↩
    d35dc81751aa5dc76c7cc8726e4600f69d2265\"",
    "rowNumber": 0
  },
  "links": {
    ...
    "action:update": {
      "href": "http://server/containers/v1/w16p2/expensesheets/data; ↩
    expensesheetnumber=10760016/card/0",
      "rel": "action:update"
    },
    ...
  },
  "data": { ... },
}
```

Note that the value of the `concurrencyControl` property is string-escaped in both JSON or XML format. Normally, unescaping is handled automatically by the JSON or XML

library code, but when you experiment on the command line you must unescape the string value manually. For example `"\"card\"=\"13d35dc81751aa5dc76c7cc8726e4600f69d2265\""` in the above example must be unescaped to `"card"="13d35dc81751aa5dc76c7cc8726e4600f69d2265"`.

4.3 Updating a Record

Now you are ready to try to update a record. This continues the preceding example:

```
$ curl -u 'Administrator:123456'
       -H 'Accept-Language: en-US'
       -H 'Maconomy-Concurrency-Control: "card"="13 ↩
    d35dc81751aa5dc76c7cc8726e4600f69d2265"'
       -H 'Content-Type: application/json'
       -d '{ "data" : { "description": "New Description" } }'
       'http://server/containers/v1/w16p2/expensesheets/data; ↩
    expensesheetnumber=10760016/card/0'
...
```

When you perform this update, the web service responds with a full copy of the updated expense sheet resource state (`http://server/containers/v1/w16p2/expensesheets/data;expenseshee`
Notice that the state transitions in Maconomy occur on record subresources, but the result is always the full container resource state.

This example includes the concurrency control tag in the `Maconomy-Concurrency-Control` HTTP request header. This example also includes the media type of the request entity in the `Content-Type` HTTP request header. This example uses the `-d` parameter to specify the request entity directly on the command line. Notice that when you use the `-d` parameter `curl` automatically use the `POST` method.

The following is the result if you forget to include the concurrency control tag:

```
$ curl -i
       -u 'Administrator:123456'
       -H 'Accept-Language: en-US'
       -H 'Content-Type: application/json'
       -d '{ "data" : { "description": "New Description" } }'
       'http://server/containers/v1/w16p2/expensesheets/data; ↩
    expensesheetnumber=10760016/card/0'
HTTP/1.1 400 Bad Request
Date: Wed, 03 Dec 2014 14:23:57 GMT
Server: Jetty(8.1.14.v20131031)
Content-Language: en-US
Vary: Content-Language,Accept-Charset,Accept
Cache-Control: no-cache, no-transform
Content-Type: application/json; charset=utf-8
Connection: close
Transfer-Encoding: chunked
```

```
{
  "errorFamily": "service",
  "errorMessage": "Missing concurrency digest for pane 'card'. Please check ↩
    the value of the 'Maconomy-Concurrency-Control' HTTP header.",
  "errorSeverity": "error"
}
```

This returns a response with the status `400 Bad Request` and a message advising you to check the `Maconomy-Concurrency-Control` HTTP request header.

The following is an example of what happens if another user changed the record since you last refreshed the resource state:

```
$ curl -i
       -u 'Administrator:123456'
       -H 'Accept-Language: en-US'
       -H 'Maconomy-Concurrency-Control: "card"="13 ↩
    d35dc81751aa5dc76c7cc8726e4600f69d2265"'
       -H 'Content-Type: application/json'
       -d '{ "data" : { "description": "New Description" } }'
       'http://server/containers/v1/w16p2/expensesheets/data; ↩
    expensesheetnumber=10760016/card/0'
HTTP/1.1 409 Conflict
Date: Wed, 03 Dec 2014 14:25:56 GMT
Server: Jetty(8.1.14.v20131031)
Content-Language: en-US
Vary: Content-Language,Accept-Charset,Accept
Cache-Control: no-cache, no-transform
Content-Type: application/json; charset=utf-8
Transfer-Encoding: chunked

{
  "errorFamily": "service",
  "errorMessage": "The operation cannot be performed because the data was ↩
    changed by another user. Please refresh and retry the request.",
  "errorSeverity": "error"
}
```

The web service responds with the status `409 Conflict`. This indicates to the client program that the request could not be performed because the same resource was updated by another user. In this situation the client program must refresh its resource state, and possibly retry the operation.

4.4 Creating a Record

When creating a record, for example another line in an expense sheet, the client program should use the *initialize* state transition to obtain a template record:

```
$ curl -u 'Administrator:123456'
       -H 'Accept-Language: en-US'
       -H 'Maconomy-Concurrency-Control: "card"=" ↩
    f39c251d85a4482c40ab3880fb401aea4b61da18"'
       -X POST
       'http://server/containers/v1/w16p2/expensesheets/data; ↩
    expensesheetnumber=10760016/table/init?row=0'
{
  "meta": {
    "concurrencyControl": "\"card\"=\" ↩
    f39c251d85a4482c40ab3880fb401aea4b61da18\"",
    "rowNumber": 0
  },
  "links": {
    "action:create": {
      "href": "http://server/containers/v1/w16p2/expensesheets/data; ↩
    expensesheetnumber=10760016/table?row=0",
      "rel": "action:create"
    }
  },
  "data": {
    "activitynumber": "",
    "activitytextvar": "",
    "amountbase": 0,
    ...
  }
}
```

The client program can then edit the template record and use the *create* link in the template record to actually create the record in the database. Notice that the template record also contains the necessary `concurrencyControl` tag. In the following example the template record was saved to the file `new-record.json` and subsequently edited. This request completes the creation of the record on the server:

```
$ curl -u 'Administrator:123456'
       -H 'Accept-Language: en-US'
       -H 'Maconomy-Concurrency-Control: "card"=" ↩
    f39c251d85a4482c40ab3880fb401aea4b61da18"'
       -H 'Content-Type: application/json'
       -d '@new-record.json'
       'http://server/containers/v1/w16p2/expensesheets/data; ↩
    expensesheetnumber=10760016/table/init?row=0'
...
```

Again, this creates the record and returns a full copy of the container resource state.

4.5 Deleting a Record

To delete a record, a client program must use the `DELETE` method on a link with the link relation `action:delete`.

For example, the following deletes a line in the expense sheet:

```
$ curl -u 'Administrator:123456'
       -H 'Accept-Language: en-US'
       -H 'Maconomy-Concurrency-Control: "card"=" ↩
    f39c251d85a4482c40ab3880fb401aea4b61da18", "table"="2 ↩
    d6cf7f5263d2272eaf794d41303b1d1b0f2867f"'
       -X DELETE
       'http://server/containers/v1/w16p2/expensesheets/data; ↩
    expensesheetnumber=10760016/table/0'
```

Again, this deletes the record and returns a full copy of the container resource state.

But what happens if you delete the `card` record? Consider the following example:

```
$ curl -i
       -u 'Administrator:123456'
       -H 'Accept-Language: en-US'
       -H 'Maconomy-Concurrency-Control: "card"="9 ↩
    f6f6c07dc4edf1457b03fd52c221b7230e0c09d"'
       -X DELETE
       'http://server/containers/v1/w16p2/expensesheets/data; ↩
    expensesheetnumber=10760016/card/0'
HTTP/1.1 204 No Content
Date: Wed, 03 Dec 2014 14:31:59 GMT
Server: Jetty(8.1.14.v20131031)
Content-Language: en-US
Vary: Content-Language
Cache-Control: no-cache, no-transform
Content-Length: 0
```

This is a special case that the client program must handle correctly. When a client program deletes the card record (the expense sheet) the expense resource itself is gone. The web service responds with the status `204 No Content` to indicate that the request was successful, but because there is no longer a resource state, there is no response entity.

4.6 Running Actions

In Maconomy each container can define state transitions specific to that container, known as *application actions*. If the record state contains a hyperlink for an application action, the client program can use the link to perform the state transition. The following example uses this link from an expense sheet card record:

```
"action:submitexpensesheet": {
  "href": "http://server/containers/v1/w16p2/expensesheets/data; ↩
    expensesheetnumber=10760015/card/0/action;name=submitexpensesheet",
  "rel": "action:submitexpensesheet"
}
```

When a client program invokes an application action, it must use the `POST` HTTP method without a request entity. For example:

```
$ curl -u 'Administrator:123456'
       -H 'Accept-Language: en-US'
       -H 'Maconomy-Concurrency-Control: "card"="33731 ↩
    bc47b290740e31f56838a5286361de7cd76"'
       -X POST
       'http://server/containers/v1/w16p2/expensesheets/data; ↩
    expensesheetnumber=10760015/card/0/action;name=submitexpensesheet'
```

Again, this runs the `SubmitExpenseSheet` action and returns a full copy of the resulting container resource state.

Chapter 5

Advanced Topics

Some containers have special functionality that requires client programs to use more complex interaction patterns:

Singleton Containers Some containers conceptually expose a single resource rather than a collection of resources.

Mutable Variable State Some containers have mutable variables that serve as parameters to actions.

Consuming and Producing Files Some containers have actions that consume or produce files.

This chapter covers these more advanced topics.

5.1 Singleton Containers

A container generally holds a collection of resources. For example the `ExpenseSheets` container can hold a number of separate expense sheets. But some containers—singleton containers—hold only a single resource. A container can be a per-user singleton or a global singleton.

The special thing about a singleton container is how the resource is addressed. To access a particular resource in a non-singleton container, a client program must access the filter resource and navigate the link with the `data:same-key` link relation. To access the resource in a singleton container the link with the `data:any-key` link relation must be used.

The reason for this is that behind the scenes a particular resource in a non-singleton container is identified by key field values that are included in the link target. A singleton container must not be addressed using the key fields even if it defines a key in its

specification. Instead it must always be addressed by "any" key to indicate that you access the single resource that is available in the container.

One example of a singleton container is `TimeRegistration`. This container is a per-user singleton container that provides access to the current user's time sheets. A mutable variable `DateVar` in the `card` pane of the container is used to control which time sheet lines to show and interact with in the table part.

Singleton containers use their card parts to represent selection criteria such as the `DateVar` variable in `TimeRegistration`. The selection criteria can be used to parameterize an action or a select particular set of lines in the table part.

The following example accesses the container entry point for `TimeRegistration`:

```
$ curl -i
       -u 'Administrator:123456'
       -H 'Accept-Language: en-US'
       'http://server/containers/v1/w16p2/TimeRegistration'
HTTP/1.1 200 OK
Date: Wed, 03 Dec 2014 14:33:06 GMT
Server: Jetty(8.1.14.v20131031)
Content-Language: en-US
Vary: Content-Language,Accept-Encoding,User-Agent
Cache-Control: no-transform, max-age=86400
Content-Type: application/json; charset=utf-8
Transfer-Encoding: chunked

{
  "containerName": "timeregistration",
  "links": {
    "data:any-key": {
      "href": "http://server/containers/v1/w16p2/timeregistration/data;any",
      "rel": "data:any-key"
    },
    "specification": {
      "href": "http://server/containers/v1/w16p2/timeregistration/ ↩
    specification",
      "rel": "specification"
    }
  }
}
```

If you contrast that with the entry point for `ExpenseSheets` you see that `TimeRegistration` has no filter resource and no links to create a resource. These are not useful in a singleton container, because the container only holds a single resource.

The only choices that you have are accessing the specification and accessing the single resource via the `data:any-key` link.

5.2 Maintaining Mutable Variable State

A record in a container resource is generally composed of fields that may be:

- Fields that are persisted in the database and make up the resource state. Such fields may be writable or read-only.
- Fields that are derived or calculated from the resource state, sometimes called variables. Such fields are read-only.
- Mutable variables. Mutable variables are not persisted in the database, but may be set by the client program anyway. Mutable variables are often used to transmit values that serve as parameters to an action.

Maconomy client programs have traditionally transmitted the entire state of all mutable variables on each request, which requires complex program logic in the client programs. Since one of the main goals of the Maconomy RESTful web service interface is to make it easy to write client programs, the containers in the web service interface offer a more limited mechanism for maintaining mutable variables on an as-needed basis by providing the variable values as query parameters.

Consider the following examples. The container `JobTasks` has an action in its `card` pane that copies the job tasks from another job:

```
$ curl -i
       -u 'Administrator:123456'
       -H 'Accept-Language: en-US'
       'http://server/containers/v1/w16p2/jobtasks/data;jobnumber=10250001'
HTTP/1.1 200 OK
Date: Wed, 03 Dec 2014 14:34:06 GMT
Server: Jetty(8.1.14.v20131031)
Content-Language: en-US
Vary: Content-Language,Accept-Encoding,User-Agent
Cache-Control: no-cache, no-transform
Content-Type: application/json; charset=utf-8
Transfer-Encoding: chunked

...
"action:copyjobtasks": {
  "href": "http://server/containers/v1/w16p2/jobtasks/data;jobnumber ↩
    =10250001/card/0/action;name=copyjobtasks",
  "rel": "action:copyjobtasks"
},
...
```

The following tries to run the action:

```
$ curl -i
       -u 'Administrator:123456'
       -H 'Accept-Language: en-US'
```

```
       -H 'Maconomy-Concurrency-Control: "card"="4 ↩
    f1d086b148ebf0f15b9126e3da239b6d00e680b"'
       -X POST
       'http://server/containers/v1/w16p2/jobtasks/data;jobnumber=10250001/ ↩
    card/0/action;name=copyjobtasks'
HTTP/1.1 422 Unprocessable Entity
Date: Wed, 03 Dec 2014 14:38:28 GMT
Server: Jetty(8.1.14.v20131031)
Content-Language: en-US
Vary: Content-Language,Accept-Charset,Accept
Cache-Control: no-cache, no-transform
Content-Type: application/json; charset=utf-8
Transfer-Encoding: chunked

{
  "errorFamily": "application",
  "errorMessage": "A job number must be entered",
  "errorSeverity": "error",
  "focus": {
    "fieldName": "copyfromjobvar",
    "paneName": "card",
    "rowNumber": 0
  }
}
```

This actions fails because the mutable variable `CopyFromJobVar` in the card pane is used to indicate the job number of the job to copy from. The variable acts as a parameter to the action. To run this action successfully the client program must supply the job number. This is done by adding a query parameter in the URL that gives the value for the `CopyFromJobVar` variable in the pane `card`. If the job to copy from has the job number `Job-0042` then the query parameter takes the form `card.copyfromjobvar=Job-0042`.

Values in query parameters are encoded as described in Data Types. Values must also be URL-escaped (sometimes called *percent encoded*). HTTP library code normally does this automatically.

If you retry the action with the query parameter added to the URL, the action runs successfully:

```
$ curl -i
       -u 'Administrator:123456'
       -H 'Accept-Language: en-US'
       -H 'Maconomy-Concurrency-Control: "card"="4 ↩
    f1d086b148ebf0f15b9126e3da239b6d00e680b"'
       -X POST
       'http://server/containers/v1/w16p2/jobtasks/data;jobnumber=10250001/ ↩
    card/0/action;name=copyjobtasks?card.copyfromjobvar=Job-0042'
HTTP/1.1 200 OK
```

```
Date: Wed, 03 Dec 2014 14:40:32 GMT
Server: Jetty(8.1.14.v20131031)
Content-Language: en-US
Vary: Content-Language
Cache-Control: no-cache, no-transform
Content-Type: application/json
Transfer-Encoding: chunked

...
```

In this example, the client program only needs to set the variable when it runs the particular action that uses the variable as a parameter. But in other situations it is important that the client program sends one or more mutable variables on every interaction. One example is in the `TimeRegistration` singleton container which provides access to the current user's time sheets. A mutable variable `DateVar` in the `card` pane controls which time sheet lines are included in the table pane.

If you simply access the link as-is:

```
$ curl -i
       -u 'Administrator:123456'
       -H 'Accept-Language: en-US'
       'http://server/containers/v1/w16p2/timeregistration/data;any'
HTTP/1.1 200 OK
Date: Wed, 03 Dec 2014 14:41:56 GMT
Server: Jetty(8.1.14.v20131031)
Content-Language: en-US
Vary: Content-Language,Accept-Encoding,User-Agent
Cache-Control: no-cache, no-transform
Content-Type: application/json; charset=utf-8
Transfer-Encoding: chunked

...
"datevar": "2014-12-03",
...
```

Then the table part gives access to show and interact with the time registration for the time sheet associated with the current date: the application business logic set the default variable value to today's date.

To work on time registrations that belongs to a particular time sheet, you include the variable value in the query parameter: `card.datevar=2014-12-24`.

```
$ curl -i
       -u 'Administrator:123456'
       -H 'Accept-Language: en-US'
       'http://server/containers/v1/w16p2/timeregistration/data;any?card. ↩
    datevar=2014-12-24'
HTTP/1.1 200 OK
```

```
Date: Wed, 03 Dec 2014 14:43:08 GMT
Server: Jetty(8.1.14.v20131031)
Content-Language: en-US
Vary: Content-Language,Accept-Encoding,User-Agent
Cache-Control: no-cache, no-transform
Content-Type: application/json; charset=utf-8
Transfer-Encoding: chunked

...
"datevar": "2014-12-24",
...
```

For example, if you want to update a time sheet line in the `table` pane, you must include the `DateVar` variable value. If you do not, then the update operation would potentially be applied to a completely different time sheet line and would fail due to the concurrency control mechanism. In fact, on every request in the `TimeRegistration` container the client program should set the variable.

In similar containers where mutable variables in the card part control which records are displayed in the table part, the mutable variable state should always be set on every request.

5.3 Working with Files

Maconomy supports actions that produce a file as output, for example, a print or data export; and actions that consume a file, for example, attaching a receipt to an expense sheet. In the Maconomy RESTful web service interface the `filedrop` endpoint enables this.

A file drop is a *temporary* file store where a user can upload a single file. A file drop has a very simple state space:

1. Unresolved: The file drop does not contain a file.
2. Resolved: The file drop contains a file.

5.3.1 Uploading a File and Using It in an Action

Consider the following example. In a client program you want to attach a receipt to an expense sheet. This is done by invoking the action `AttachDocumentToLine`:

```
$ curl -i
       -u 'Administrator:123456'
       -H 'Accept-Language: en-US'
       -H 'Maconomy-Concurrency-Control: "card"="919 ↩
    fa16d3df3a66516372fbb2e98d06ab0f43db5", "table"="4707 ↩
    fce4f4e756b3b5b2909024d78ebcbd703963"'
```

```
       -X POST
       'http://server/containers/v1/w16p2/expensesheets/data; ↩
    expensesheetnumber=10760040/table/0/action;name=attachdocumenttoline'
HTTP/1.1 400 Bad Request
Date: Wed, 03 Dec 2014 10:59:29 GMT
Server: Jetty(8.1.14.v20131031)
Content-Language: en-US
Vary: Content-Language,Accept-Charset,Accept
Cache-Control: no-cache, no-transform
Content-Type: application/json; charset=utf-8
Connection: close
Transfer-Encoding: chunked

{
  "errorFamily": "service",
  "errorMessage": "Missing file callback. Please check the value of the ' ↩
    Maconomy-File-Callback' HTTP header.",
  "errorSeverity": "error"
}
```

The request fails because the action consumes a file (the receipt), and you did not provide the file. To perform the action you need to create a file drop containing the file and pass the file drop URI in the `Maconomy-File-Callback` HTTP request header field when you invoke the action.

First you create a file drop:

```
$ curl -i
       -u 'Administrator:123456'
       -H 'Accept-Language: en-US'
       -X POST
       http://server/filedrop/v1/w16p2/new
HTTP/1.1 201 Created
Date: Wed, 03 Dec 2014 12:48:06 GMT
Server: Jetty(8.1.14.v20131031)
Content-Language: en-US
Vary: Content-Language
Location: http://server/filedrop/v1/w16p2/3404797840542625411/
Cache-Control: no-cache, no-transform
Content-Type: application/json; charset=utf-8
Transfer-Encoding: chunked

{
  "location": "http://server/filedrop/v1/w16p2/3404797840542625411/"
}
```

You use the `POST` HTTP method, and the URL pattern for the `new` resource is:

```
http://{host}/filedrop/v1/{shortname}/new
```

The location of the file drop is included both in the response entity and in the `Location` HTTP response header field. Try to get the state of the file drop at this point:

```
$ curl -i
       http://server/filedrop/v1/w16p2/3404797840542625411/
HTTP/1.1 404 Not Found
Date: Wed, 03 Dec 2014 12:48:41 GMT
Server: Jetty(8.1.14.v20131031)
Cache-Control: no-cache, no-transform
Content-Type: application/json; charset=utf-8
Vary: Accept-Charset,Accept
Transfer-Encoding: chunked

{
  "errorFamily": "service",
  "errorMessage": "The file drop exists, but no file has been uploaded.",
  "errorSeverity": "error"
}
```

This produces an error that indicates that the file drop has not been resolved yet. You can resolve it by uploading a file:

```
$ curl -i
       -u 'Administrator:123456'
       -H 'Accept-Language: en-US'
       -H 'Content-Type: application/octet-stream'
       -H 'Content-Disposition: attachment; filename="receipt.jpg"'
       --data-binary '@receipt.jpg'
       http://server/filedrop/v1/w16p2/3404797840542625411/
HTTP/1.1 100 Continue

HTTP/1.1 204 No Content
Date: Wed, 03 Dec 2014 12:50:47 GMT
Server: Jetty(8.1.14.v20131031)
Content-Language: en-US
Vary: Content-Language
Cache-Control: no-cache, no-transform
Content-Length: 0
```

This example makes an HTTP `POST` request with the contents of the file `receipt.jpg` as the payload. You use the media type `application/octet-stream` as `Content-Type` to indicate that the request entity is the raw binary content of the file. The `Content-Disposition` header field is used to suggest a filename that the server may use to store the file. The response has a status of `204 No Content`, which means that the request was successful.

If you try to get the state of the file drop you get back the file contents:

```
$ curl -I http://server/filedrop/v1/w16p2/3404797840542625411/
HTTP/1.1 200 OK
```

```
Date: Wed, 03 Dec 2014 13:12:55 GMT
Server: Jetty(8.1.14.v20131031)
Content-Type: image/jpeg
Cache-Control: no-cache, no-transform
Content-Length: 33888
```

If you try to upload a file to the same file drop you get an error:

```
$ curl -i
       -u 'Administrator:123456'
       -H 'Accept-Language: en-US'
       -H 'Content-Type: application/octet-stream'
       -H 'Content-Disposition: attachment; filename="receipt.jpg"'
       --data-binary '@receipt.jpg'
       http://server/filedrop/v1/w16p2/3404797840542625411/
HTTP/1.1 100 Continue

HTTP/1.1 409 Conflict
Date: Wed, 03 Dec 2014 13:15:19 GMT
Server: Jetty(8.1.14.v20131031)
Content-Language: en-US
Vary: Content-Language,Accept-Charset,Accept
Cache-Control: no-cache, no-transform
Content-Type: application/json; charset=utf-8
Transfer-Encoding: chunked

{
  "errorFamily": "service",
  "errorMessage": "Cannot upload file. A file has already been uploaded to ↩
    this file drop.",
  "errorSeverity": "error"
}
```

Since the file drop has already been resolved, the request fails with a status of `409 Conflict`.

Now you can retry the `AttachDocumentToLine` action, supplying the uploaded file through the `Maconomy-File-Callback` HTTP request header field:

```
$ curl -i
       -u 'Administrator:123456'
       -H 'Accept-Language: en-US'
       -H 'Maconomy-Concurrency-Control: "card"="919 ↩
    fa16d3df3a66516372fbb2e98d06ab0f43db5", "table"="4707 ↩
    fce4f4e756b3b5b2909024d78ebcbd703963"'
       -H 'Maconomy-File-Callback: <http://server/filedrop/v1/w16p2 ↩
    /3404797840542625411/>'
       -X POST
       'http://server/containers/v1/w16p2/expensesheets/data; ↩
    expensesheetnumber=10760040/table/0/action;name=attachdocumenttoline'
```

```
HTTP/1.1 200 OK
Date: Wed, 03 Dec 2014 13:24:12 GMT
Server: Jetty(8.1.14.v20131031)
Content-Language: en-US
Vary: Content-Language
Cache-Control: no-cache, no-transform
Content-Type: application/json
Transfer-Encoding: chunked

...
```

This time the action succeeded, and the file was supplied from its temporary location in the file drop. The `AttachDocumentToLine` has now saved the file into a document archive in Maconomy where it can be retrieved later.

The preceding examples shows the basic workflow:

1. Create a file drop.
2. Upload a file to the file drop.
3. Run an application action that consumes a file and pass the file drop URI in the `Maconomy-File-Callback` HTTP request header field.

5.3.2 Maconomy-File-Callback

The `Maconomy-File-Callback` HTTP request header field is used for requests that need one or more files from the client program. The format for the header field is:

```
Maconomy-File-Callback = "Maconomy-File-Callback" ":" 1#file-uri-value
file-uri-value = "<" URI-Reference ">"
URI-Reference = <URI-reference, see [RFC3986], Section 4.1>
```

Each file drop URI must be enclosed in angle brackets. To supply multiple file drop URIs for the same request, client programs can either include the `Maconomy-File-Callback` header field multiple times, or supply the values comma-separated in the same header field value [see 5, section 4.2].

5.3.3 Uploading Binary Data

The simplest way for the client program to upload a file to a file drop is to `POST` the binary data as the previous example showed.

In this case you include the following HTTP headers:

HTTP request header	Description
`Content-Type`	Value must be `application/octet-stream` to indicate that the request entity of the `POST` request is unstructured binary data.
`Content-Disposition`	Value is `attachment; filename="myfile.txt"` where 'myfile.txt is the client program's suggestion for the filename the server should use when storing the file.

The request entity of the `POST` request is the binary file contents.

5.3.4 Uploading multipart/form-data

The file drop also supports using the `multipart/form-data` media type. This enables client programs to upload a file through a classic HTML form [see 8, for technical details]. The name of the file part must be `file`. The following is an example of an HTML form that uploads a file to a file drop:

```
<form action="http://server/filedrop/v1/w16p2/340479784054262541l/"
      method="post"
      enctype="multipart/form-data">
  <input type="file" name="file"><!-- name must be "file" -->
  <input type="submit" value="Upload file">
</form>
```

5.3.5 Running an Action and Downloading a Resulting File

Actions in containers may also produce one or more files. When an action produces files, links will be included in the `Link` HTTP response header field.

The following example retrieves the receipt that the previous example attached to the expense sheet by running the `ShowDocument` action:

```
$ curl -i
       -u 'Administrator:123456'
       -H 'Accept-Language: en-US'
       -H 'Maconomy-Concurrency-Control: "card"="919 ↩
    fa16d3df3a66516372fbb2e98d06ab0f43db5", "table"=" ↩
    f648c00d3ec40456b08f59f495a60c362653c203"'
       -X POST
       'http://server/containers/v1/w16p2/expensesheets/data; ↩
    expensesheetnumber=10760040/table/0/action;name=showdocument'
HTTP/1.1 200 OK
```

```
Date: Wed, 03 Dec 2014 14:04:08 GMT
Server: Jetty(8.1.14.v20131031)
Content-Language: en-US
Vary: Content-Language
Cache-Control: no-cache, no-transform
Link: <http://server/filedrop/v1/w16p2/2274162197919216380/>;rel=file;type= ↩
    image/jpeg
Content-Type: application/json
Transfer-Encoding: chunked

...
```

The response includes a `Link` HTTP response header field value. The `file` link relation indicates that this is a file produced by the request, while the `image/jpeg` indicates the media type of the linked resource.

See Nottingham [9] for the details of the `Link` header field format. A library function for parsing the `Link` header field value is often available on client platforms.

Bibliography

[1] JSON. URL http://www.json.org.

[2] ECMA-404: The json data interchange format, October 2013. URL http://www.ecma-international.org/publications/standards/Ecma-404.htm.

[3] CMdml. *Deltek Maconomy—MDML Language Reference Guide*. Deltek Inc.

[4] L. Dusseault. HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV). RFC 4918 (Proposed Standard), June 2007. URL http://www.ietf.org/rfc/rfc4918.txt.

[5] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. URL http://www.ietf.org/rfc/rfc2616.txt.

[6] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. HTTP Authentication: Basic and Digest Access Authentication. RFC 2617 (Draft Standard), June 1999. URL http://www.ietf.org/rfc/rfc2617.txt.

[7] Eve Maler, Jean Paoli, Tim Bray, François Yergeau, and Michael Sperberg-McQueen. Extensible markup language (XML) 1.0 (fifth edition). W3C recommendation, November 2008. URL http://www.w3.org/TR/2008/REC-xml-20081126/.

[8] L. Masinter. Returning Values from Forms: multipart/form-data. RFC 2388 (Proposed Standard), August 1998. URL http://www.rfc-editor.org/rfc/rfc2388.txt.

[9] M. Nottingham. Web Linking. RFC 5988 (Proposed Standard), June 2010. URL http://www.ietf.org/rfc/rfc5988.txt.

[10] Ed. T. Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 7159 (Proposed Standard), March 2014. URL http://www.ietf.org/rfc/rfc7159.txt.

[11] Jim Webber, Savas Parastatidis, and Ian Robinson. *REST in Practice: Hypermedia and Systems Architecture*. O'Reilly Media, 2010.