The Maconomy RESTful Web Services

Programmer's Guide
2022

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 December 2022.

Revision	Date	Description
1.0	2015	First version of the programmer's guide describing the now deprecated version of the Maconomy RESTful web services.
2.0	2021	Updated to cover Containers Web Service version 2.0 and additional new services.
2.1	2021	Configuration added to describe `server.ini` settings relevant for the Maconomy RESTful web services covered.
2.2	2021	Updated to cover Containers Web Service version 3.0: • Tree table panes are now supported and hence dot indices have been introduced for pointing out Record Positions. • Partial Data Responses extended to cover the new move record patch operation. Media Types covers the `version` parameter now part of the custom media types. Web Access Configuration extended with section about diagnostic logging. The different versions of the Maconomy RESTful web services are now shortly described in Versions.
2.3	2021	Updated to cover Containers Web Service version 4.0: • Filtering parameters may be supplied in the request body instead of as query parameters. • When no `fields` parameter is supplied, only key fields are included in the filter response. • For table panes with moveable records, the container Specification exposes a `moveMode` property and targeted title properties `upTitle`, `downTitle`, `indentTitle`, and `outdentTitle`. Updated to cover Root Web Service version 2.0.

2.4	2022	Updated to cover Containers Web Service version 5.0: • An `access` hyperlink is included in container resources. • Unless otherwise specified during Data Fields Slicing, only key fields can be mentioned in update requests and only key fields are included in data responses. • Whenever applicable, container instance resources include a `data:some-key` hyperlink. Updated to cover Messages Web Service version 1.0. Updated to cover Diagnostics Web Service version 1.0. Updated to cover `errorIds` introduced in Error Responses.
2.5	2022	Updated to cover Containers Web Service version 6.0: • For popup containers: – A `hidden` field has been made available for popup containers, indicating whether an enum value should be selectable in a user interface or not. – The Filtering parameters `fields`, `restriction`, and `orderBy` are now also supported for popup containers. – If no `fields` parameter is supplied in a popup container filter request, only the `value` field will be included in the response. • Multi-column Table Sorting is now configurable for container instances. • Table Paging is now supported for data responses. • Query parameters have been eradicated and request body parameters are used instead. Consequently, the following stand: – Only the HTTP verb `POST` is applicable when following a `data:filter` or `data:enumvalues` hyperlink. – Instead of `DELETE`, the HTTP verb `POST` must be applied when following an `data:delete` hyperlink. – The HTTP verb `POST` may also be applied when following an `instance:data` hyperlink. Updated to cover the Maconomy User Resource introduced in Authentication Web Service version 1.3.

Contents

Chapter 1

Introduction

The Maconomy RESTful web services are a collection of programmatic interfaces providing access to data and business functionality within the Deltek Maconomy ERP product.

1.1 REST

Before going into details about the Maconomy RESTful web services, it is relevant to quickly go over what REST is and the concepts and terminology associated with it.

REST stands for *Representational State Transfer* and refers to a certain architectural style to be used when creating web services. A web service that is built on REST principles is said to be *RESTful*.

1.1.1 Resources

A central concept in REST is the *resource*. A resource is a domain object that is uniquely identified by a URL.

When accessing the URL of a resource, one gets a *representation* of the current state of that resource back. 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.

Resources are accessed and manipulated (read, updated, deleted, and so on) by a fixed set of HTTP verbs. The verbs used by the Maconomy RESTful web services are `GET`, `PUT`, `POST` and `DELETE` [7]. Throughout this document, the `GET` verb is the one applicable, if nothing else is mentioned.

1.1.2 Hyperlinks

Hyperlinks are a well-known concept from the web and are also pervasive in RESTful web services. Just like on a web page, hyperlinks there point to *related resources*.

Hyperlinks are also used to represent available *state transitions*. For example, to update the state of a resource, the client program needs to follow some specific hyperlink. Resources have hyperlinks for all available state transitions.

Each hyperlink has an associated *link relation* which is simply an identifier that tells client programs what the hyperlink can be used for (for example, accessing a related resource, updating, submitting, transferring). When writing client programs, one should only rely on link relations and consider all URLs opaque. One should *never* attempt to guess the URL pattern for any resource. Only the link relation of a hyperlink is guaranteed to be stable.

The REST principle of enforcing client programs to dynamically discover the service and its hyperlinks is referred to as *Hypermedia as the Engine of Application State*, or *HATEOAS* in short.

1.1.3 Other Styles of Web Services

REST is often contrasted with other styles 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.

1.1.4 Further Reading

It is recommended that developers working with producing or consuming RESTful web services read the book “REST in Practice: Hypermedia and Systems Architecture” [12].

1.2 cURL

This document uses the free cURL tool for all examples. If cURL is not already installed on your machine (on macOS and Linux it is likely already installed), you can download it from here: https://curl.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. Git for Windows can be found here: https://git-scm.com/

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, documenting 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	Explanatory text
`-i`	Include the HTTP response headers in the output.
`-u USERNAME:PASSWORD`	Use the specified username and password as HTTP Basic Authentication credentials.
`-H 'HEADER: VALUE'`	Include the specified HTTP header in the request.
`-d @FILE`	Make an HTTP `POST` request with the contents of given file in the request body.
`-X POST`	Make an HTTP `POST` request. If the `-d` option is not used, the request will have an empty request body, else, if the `-d` option *is* used, the `-X POST` can be left out.
`-X DELETE`	Make an HTTP `DELETE` request.

1.3 Version History

The first version of the Maconomy RESTful web services was released with Maconomy 2.1.3 and consisted of a suite of web services that allowed for a number of different interactions with a Maconomy installation. Central to the suite of web services is the Containers Web Service allowing for interaction with the so-called *Maconomy containers* which exposes all business functionality in Maconomy.

1.3.1 Changes in Maconomy 2.5.2

In the version of the Maconomy RESTful web services released with Maconomy 2.5.2, a major rewrite of the Containers Web Service has been carried out. The previous version of the Containers Web Service (CWS1) used a model that in certain situations incurred a non-negligible performance overhead. For each interaction, the server had to do a large amount of recalculations.

To address this performance issue, a different interaction model is used in the improved version of the Containers Web Service (CWS2). To interact with the data within a container using CWS2, it is necessary to create a so-called *container instance* holding important parts of the container's state. This eliminates the need for the recalculations that were necessary in CWS1 and has greatly improved performance. CWS2 now performs on par with other APIs used by Maconomy clients. iAccess for Maconomy 2.5.2 uses CWS2 and sees significant performance improvements.

In this document, we only describe how to work with CWS2. For a detailed description of CWS1, we refer to the previous version of this document.

The most important changes to the Maconomy RESTful web services in Maconomy 2.5.2 are summarized below:

- A new Root Web Service has been added. From here, all other active RESTful web services can be discovered in accordance with the HATEOAS principle.
- All web services now have their own custom media types. This allows for versioning of the services and for client programs to stay compatible by requesting particular versions. For backwards compatibility, some web services can still be addressed with `application/json` in `Accept` and `Content-Type` headers. However, this is discouraged and in the future, such requests may fail with `406 Not Acceptable` or `415 Unsupported Media Type`.
- As described above, the Containers Web Service has been reimplemented in a new version 2:
  1. All interactions with a container's data now go through a container instance which needs to be created first (see Container Instances). This has led to a significant performance improvement.
  2. Two features have been added to significantly reduce the amount of data transferred between the client program and the server:
     - You can limit the number of fields returned in data responses (see Data Fields Slicing). This is in particular relevant for containers where the client program is only interested in a small subset of the container's fields.
     - Partial Data Responses is a new feature that, if enabled, causes the server to reply with only changes to data instead of the full data response. This

means that only the changed field values are communicated and unaltered records are completely left out of the data response.

3. Printing where print layout selection is necessary now works for all containers. Previously, prints where the client program had to select a print layout did not work with the Containers Web Service.

4. Containers with variable state now work correctly. Previously, containers with significant variable state malfunctioned to various degrees. Time Registration is an example of such a container and so are containers where the client program has to select in the card pane which records will be displayed in the table pane.

1.3.2 Changes in Maconomy 2.5.3

In the version of the Maconomy RESTful web services released with Maconomy 2.5.3, the Containers Web Service now fully supports data containers that have tree table panes with hierarchically organized records. From a client program's perspective, the main differences are:

- JSON objects received as representations of table pane records now reflect any hierarchical structure.
- Record Positions are pointed out by so-called dot indices.
- A new move record patch now occurs in connection with Partial Data Responses.

With the File Drop Web Service included in Maconomy 2.5.3, a file drop can no longer be both resolved and retrieved by a client program. Instead a file drop can now *either* be resolved *or* retrieved.

1.3.3 Changes in Maconomy 2.5.4

In the version of the Maconomy RESTful web services released with Maconomy 2.5.4, Filtering and foreign key searching (see Foreign Keys) in the Containers Web Service have changed in the following ways:

- The filtering parameters `fields`, `restriction`, `orderBy`, `offset`, and `limit` may now be supplied as properties of a JSON object in the body of a filter or foreign key search `POST` request instead of as query parameters.
- When no `fields` parameter is supplied with a filter or foreign key search request, only key fields are included in the filter response.

Also, the Specification of a container's table pane now holds more details about any defined move action, allowing client programs to discover in advance if records may only be moved around inside their current context.

Finally, the version information exposed as part of a root resource representation in the Root Web Service has been streamlined and thus extended with an application build number.

1.3.4 Changes in Maconomy 2.6

In the version of the Maconomy RESTful web services released with Maconomy 2.6, the Diagnostics Web Service has been introduced. The purpose of this service is to facilitate a way of easily judging whether the system seems to be correctly configured.

Also, the Messages Web Service has been introduced, offering web-based clients a way of viewing and interacting with the current set of system messages in the Maconomy system.

In the Containers Web Service, container resources have been extended with an `access` hyperlink, allowing the client program to retrieve information about the authenticated user's CRUD access rights (see Access). Besides this, only key fields can be mentioned in update requests and only key fields are included in data responses, unless otherwise specified in the Data Fields Slicing JSON object submitted on container instance creation. Also, whenever applicable, container instance resources include a `data:some-key` hyperlink to be followed in cases where the client program wants to load the data entry corresponding to some already known key (see Loading a Data Entry).

In general, an `errorIds` property has been introduced in Error Responses.

1.3.5 Changes in Maconomy 2.6.1

In the version of the Maconomy RESTful web services released with Maconomy 2.6.1, the Filtering in the Containers Web Service has changed for popup containers. Aside from the `value`, `ordinal`, and `title` field previously available for popup containers, a `hidden` field has been made available, providing the client with information about whether an enum value should be selectable in a user interface or not. Also, like for other containers, the filtering parameters `fields`, `restriction`, and `orderBy` are now supported for popup containers, and if no `fields` parameter is supplied in a popup container filter request, only the `value` field will be included in the response.

Furthermore, for the Containers Web Service, Table Paging is now available for data responses produced during data container interactions, and in order to make such paging useful, the client program has also been enabled to control the order of table pane records by configuring multi-column Table Sorting for container instances.

As a consequence of query parameters having been eradicated from the Containers Web Service and request body parameters being used instead, the following stand:

- Only the HTTP verb `POST` is applicable when following a `data:filter` (or a `data:enumvalues`) hyperlink (see Filtering).

- Instead of `DELETE`, the HTTP verb `POST` must be applied when following a `data:delete` hyperlink (see Deleting a Record).
- The HTTP verb `POST` may also be applied when following an `instance:data` hyperlink (see Container Instances).

In the Authentication Web Service, an `auth:maconomy` hyperlink has been made available from the root resource, enabling the client program to select the role assigned to the authenticated user.

Chapter 2

General

This chapter covers some of the concepts that are relevant across all the Maconomy RESTful web services.

2.1 Proxy Requirements

To be secure, the Maconomy RESTful web services must be deployed behind an SSL/TLS termination proxy (a reverse proxy) encrypting the traffic between the server and the client. Direct access via `http` must be blocked, and the client's use of the `https` protocol must be communicated to the web service by having the proxy set the following request header:

```
X-Forwarded-Proto: https
```

In order to eliminate any header injection vulnerability, the reverse proxy must flush the dominating `X-Forwarded-Host` header and populate it with the host information expected to appear in the hyperlinks produced by the web services.

Also, if the path of the root resource of the Maconomy RESTful web services is configured to something other than `/`, the reverse proxy must pass this information along using a `Maconomy-Forwarded-Base-Path` request header. If, for example, the root resource is available at `/maconomy-api`, the following header must be set by the reverse proxy:

```
Maconomy-Forwarded-Base-Path: maconomy-api
```

2.2 JSON

Every resource of the the Maconomy RESTful web services can be requested in a JSON format [1, 4] only.[1]

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.

2.3 Media Types

Each Maconomy RESTful web service introduces one or more custom JSON media types covering the JSON representations within the web service. A main purpose of these custom media types is version handling as explained in this section.

In the chapters Root Web Service, Containers Web Service, Popup Types Web Service, File Drop Web Service, Logging Web Service, User Settings Web Service, Messages Web Service, and Diagnostics Web Service, the latest custom media type of the web service described is specified.

2.3.1 `Accept` Request Header

Taking the Containers Web Service as an example, a client program signals that it wants to interact with a version of the service compatible with version 6.0 by including the following custom media type (or one that it is compatible with) in an `Accept` header on the request:

```
application/vnd.deltek.maconomy.containers+json; charset=utf-8; version ↩
    =6.0
```

If the newest compatible version of the Containers Web Service available in the system is version 6.0, a request with such an `Accept` header will be served by this version and the following `Content-Type` header will be sent back on any successful payload carrying response:

```
Content-Type: application/vnd.deltek.maconomy.containers+json; charset=utf ↩
    -8; version=6.0
```

In general, when a client program supplies a custom media type with parameter `version=`$x.y$, the request must be served by version $x.y'$ of the web service for some $y' \geq y$. If such compatible version happens to be available in the system, $x.y'$ will be the

[1] There are a few exceptions to this rule. For example, when downloading a file using the File Drop Web Service, the format is determined by the actual file being downloaded. The media type of a PDF file, for example, is `application/pdf`.

value of the `version` parameter of the custom media type sent back in a `Content-Type` header on any successful payload carrying response. If no compatible service version exists, a `406 Not Acceptable` will be responded.

Note that leaving out a `version` parameter from the `Accept` header corresponds to supplying a `version` parameter pointing out the latest version of the service. The latest version will also be the one reached, if no `Accept` header is supplied at all.

In addition, note that since a request submitted towards version $x.y$ of a web service may in fact be served by version $x.y'$ for some $y' > y$, the client program must be able to cope with situations where a JSON object received in a response contains some extra properties compared to a similar version $x.y$ response.

Finally, note that there are resources within the Maconomy RESTful web services whose representations are not provided in a JSON format. For example, in order to retrieve a PDF document stored in some file drop on the server (see File Drop Web Service), the media type `application/pdf` must be among the types acceptable for the client program.

2.3.2 `Content-Type` Request Header

When a client program submits a request body towards some Maconomy RESTful web service, the media type of the body's content must always be provided in a `Content-Type` header on the request.

For JSON formatted payloads, the `Content-Type` header on the request must be populated with a custom media type that the one derived from the `Accept` header is compatible with. That is, if the client program submits an `Accept` header which leads to a custom media type with `version=`$x.y$ being served, then the custom media type sent in the `Content-Type` header on the same request must have `version=`$x.y'$ for some $0 \leq y' \leq y$.

For the Containers Web Service, for example, the following is a valid `Content-Type` header for the client program to provide when submitting uncommitted record data along with a foreign key search request (see Foreign Keys) being handled by a version 6.0 compatible version of the service:

```
Content-Type: application/vnd.deltek.maconomy.containers+json; charset=utf ↩
    -8; version=6.0
```

Leaving out the `version` parameter from a `Content-Type` header corresponds to supplying a `version` parameter holding the latest version of the service.

For requests carrying non-JSON formatted payloads, the client program must provide an appropriate standard media type in the `Content-Type` header. For example, when resolving a file drop (see File Drop Web Service), a valid `Content-Type` request header

contains one of the following two standard media types: `application/octet-stream` or `multipart/form-data`.

If the client program provides an invalid `Content-Type` header along with a payload carrying request, a `415 Unsupported Media Type` response is sent back. This is also the response when the client program does not include a payload and a `Content-Type` header when required.

2.4 Compression

The Maconomy RESTful web services support `gzip` compression via the standard HTTP mechanism [5, section 14.4]. If a client program includes `gzip` in an `Accept-Encoding` HTTP header on a request, the server will `gzip` compress the body of the response. HTTP client library code normally handle compressed responses transparently. In cURL it can be handled by using the `--compress` option.

2.5 Language

As described in the Root Web Service section, the state of the outermost root resource includes a list of languages supported by the Maconomy system, for example:

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

In this example, the system is configured to support two languages, Danish and US English.

To specify the preferred language for a resource, include an `Accept-Language` HTTP header in the request holding the relevant language tag. For example, to get the resource state in US English include `-H 'Accept-Language: en-US'`, or, to get the resource state in Danish, include `-H 'Accept-Language: da-DK'`.

To unambiguously apply the language preference, it is recommended that client programs include an `Accept-Language` HTTP header with all requests. The value of the header should be the exact language tag value of one of the supported languages.

2.6 Formats

Preferred formats can be indicated by inclusion of a `Maconomy-Format` HTTP header in a request. This is significant in the cases where the server will apply formatting to the data. For example, when printing an expense sheet using the Containers Web Service, the user's date format and decimal separator should be used in the printed document. Conversely, the formats do not apply to container data values that are independent of the user's locale and format preferences.

Consider the following example:

```
Maconomy-Format: date-format="dd-MM-yyyy", time-format="HH:mm", thousand- ↩
    separator=".", decimal-separator=",", number-of-decimals=2
```

This example shows all the possible format directives that the client program may specify. Not all possible date and time formats are supported by the Maconomy system.

Directive	Explanatory text
`date-format`	This directive indicates how the server should format date values.
`time-format`	This directive indicates how the server should format time values.
`thousand-separator`	This directive indicates the character used as a thousand separator.
`decimal-separator`	This directive indicates the character used as a decimal separator.
`number-of-decimals`	This directive indicates the number of decimals to include.

2.7 Data Types

Maconomy uses eight primitive data types. For container data resources (see Containers Web Service), these data types are embedded in JSON objects and are encoded in a locale-independent way.

Several Maconomy data types use the `number` grammar rule of the JSON data interchange format [4]. For reference the `number` grammar rule is defined as [11]:

```
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
```

2.7.1 Integer

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

Integer values are represented as a JSON number that must conform to the `number` grammar rule [4] 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`.

2.7.2 Real

The real data type is a floating point data type.

Real values are encoded as JSON numbers. Values must conform to the `number` grammar rule [4]. Examples of acceptable values are `100`, `.892`, `2e10`, and `314159e-5`.

2.7.3 Amount

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

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`.

2.7.4 Boolean

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

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

2.7.5 String

The string data type is used to represent text. The character set used is determined by the enclosing JSON document and may be indicated in the `Content-Type` header. UTF-8 is the default. Note that Unicode characters may be escaped using the `\uXXXX` where `X` is a hexadecimal digit.

String values are represented as JSON string values and must conform to the `string` grammar rule [4]. Examples of acceptable values are `""` and `"Hello world"`.

2.7.6 Date

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

Date values are represented as a JSON string [4] whose contents conform to the 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.

Examples of acceptable values are: `""`, `"1950-04-05"`, `"1945-04-25"`, `"1946-12-16"`, and `"1945-11-15"`.

2.7.7 Time

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

Time values are represented as a JSON string [4] whose contents conform to the 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.

Examples of acceptable values are: `""`, `"10:59:23"`, and `"19:21:49"`.

2.7.8 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 some contexts (for example, within expressions), enum values are written using the notation `PopupType'PopupLiteral`, but in order to avoid the need to parse this enum notation client-side, enum values are represented as a JSON string [4] that contains only the *enum literal* value. 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"`.

2.7.9 Time Duration

The time duration data type is a special purpose variant of the real data type. It has the same JSON 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.

2.7.10 Auto Timestamp

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

2.8 Authentication

Most requests towards the Maconomy RESTful web services require authentication and in this section the protocols currently supported are presented.

Authentication may entail transmission of credentials on a request and therefore, as described in Proxy Requirements, 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 RESTful web services are vulnerable to eavesdropping by an attacker.

2.8.1 HTTP Basic Authentication

Requests towards the Maconomy RESTful web services can be authenticated using HTTP Basic Authentication [8].

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

When a client program tries to interact unauthenticated with a resource requiring authentication, the server responds with the status `401 Unauthorized` and includes a `WWW-Authenticate` HTTP header indicating the method of authentication to be used to gain access to the resource. In the Maconomy case, this header looks something like this:

```
WWW-Authenticate: Basic realm="Maconomy"
```

The token `Basic` in the header indicates that the server requires the client to use HTTP Basic Authentication, and hence the client has to construct HTTP Basic Authentication credentials and retry the request.

The following is a simple Python program illustrating how to compute such 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 client program must retry the request, supplying the following header:

```
Authorization: Basic QWRtaW5pc3RyYXRvcjoxMjM0NTY=
```

Note that Franks et al. [8] implicitly requires the credentials to be encoded as `ISO-8859-1` by using the `TEXT` grammar rule defined in Fielding et al. [7]. However, most (but not all) modern browsers encode the credentials as `UTF-8`. The Maconomy RESTful web services follow the modern convention and require user credentials to be `UTF-8` encoded. This allows a wider range of special characters to appear in usernames and passwords.

Also 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 (see Proxy Requirements).

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 when the web browser detects the `Basic` authentication scheme in the `WWW-Authenticate` HTTP 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 following custom HTTP request header:

```
Maconomy-Authentication: 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:

```
WWW-Authenticate: X-Basic realm="Maconomy"
```

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

Expired User Passwords

If the user's password has expired, a request fails with a `401 Unauthorized` status and the `WWW-Authenticate` HTTP header included in the response will indicate the custom authentication method `X-ChangePassword` offered by the server:

```
WWW-Authenticate: X-ChangePassword realm="Maconomy"
```

The change password authentication method authenticates the request and changes the user's password. The credentials are computed in a way similar to the standard HTTP Basic Authentication described above:

```
username     = u"Anders Hansen"
old_password = u"123456"
new_password = u"654321"

# 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. Also, the token indicating the authentication method is `X-ChangePassword`, rather than `Basic`.

In the above example, the client program can resolve the situation by retrying the request with the following HTTP request header:

```
Authorization: X-ChangePassword QW5kZXJzIEhhbnNlbjoxMjM0NTYKNjU0MzIx
```

The user's password is then changed to `654321` and the client program can 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 his password.

2.8.2 Maconomy Reconnect Authentication

The Maconomy RESTful web services support a proprietary authentication mechanism known as the Maconomy Reconnect Authentication. This authentication option allows the client program to acquire a reconnect token on login and then use this token for authentication in subsequent requests. Using reconnect tokens when issuing a series of requests improves performance as the server then does not have to spend time on expensive hash calculations doing password verification.

A client can have a Maconomy reconnect token returned either in an HTTP session cookie or in a custom `Maconomy-Reconnect` HTTP header. If the client program is browser-based, it is strongly recommend to use the HTTP session cookie option as it offers protection against several kinds of session theft attacks while relieving the web service client code from having to manage the login session.

HTTP Session Cookie

This is the workflow of using Maconomy Reconnect Authentication via an HTTP session cookie:

1. The client program authenticates (for example, using HTTP Basic Authentication), including the following header on the request:

   ```
   Maconomy-Authentication: X-Cookie
   ```

2. The response received from the server includes a `Set-Cookie` header holding a Maconomy reconnect token session cookie and a `Maconomy-Cookie` header holding the name of that session cookie.

3. On subsequent requests, the client includes the following header, where `<cookie name>` is replaced by the session cookie name received from the server:

   ```
   Authorization: X-Cookie <cookie name>
   ```

4. Each response received from the server may include a `Set-Cookie` header which updates the stored session cookie.

5. On the last request, the client includes the following header to indicate that the server can log the user out and release any cached resources:

   ```
   Maconomy-Authentication: X-Log-Out
   ```

6. The response received from the server includes a final `Set-Cookie` header that will expire the session cookie, causing it to be deleted from the client's cookie store.

A Maconomy reconnect token HTTP session cookie is always marked with `httpOnly` to protect it from being accessed directly by client-side scripts. If the server detects that an authentication request is performed on a secure channel, it also marks the cookie with `secure` to prevent it from being used for unencrypted requests.

`Maconomy-Reconnect` HTTP header

This is the workflow of using Maconomy Reconnect Authentication via a `Maconomy-Reconnect` HTTP header:

1. The client program authenticates (for example, using HTTP Basic Authentication), including the following header on the request:

   ```
   Maconomy-Authentication: X-Reconnect
   ```

2. The response received from the server includes a `Maconomy-Reconnect` header holding a Maconomy reconnect token.

3. On subsequent requests, the client includes the following header, where `<reconnect token>` is replaced by the reconnect token received from the server:

   ```
   Authorization: X-Reconnect <reconnect token>
   ```

4. Each response received from the server may include a `Maconomy-Reconnect` header, and the client must always use the most recently received reconnect token.

5. On the last request, the client includes the following header to indicate that the server can log the user out and release any cached resources:

   ```
   Maconomy-Authentication: X-Log-Out
   ```

2.8.3 Kerberos

Kerberos Domain Credentials

If a Maconomy system is set up to use Kerberos authentication, any HTTP Basic Authentication credentials will, by default, be interpreted as Kerberos domain credentials.

As described in Installation, the state of an installation resource includes information about enabled authentication schemes, and by examining this, a client program should be able to figure out if and how to use Kerberos domain credentials. In a Kerberos authentication enabled system, the JSON object held in the `authentication` property of the installation resource state will look something like this:

```
{
  "useDomainCredentialsForBasicAuthentication": true,
  ...,
  "kerberos": {
    "kdc": "PSO-DC.PSO.COM",
    "realm": "PSO.COM",
    "realms": {
      "PSO.COM": {
        "kdc": "PSO-DC.PSO.COM",
        "name": "PSO.COM"
      }
    },
    "serviceName": "MACONOMYSSO/PSO.COM"
  },
  ...
}
```

The `useDomainCredentialsForBasicAuthentication` property holding the boolean value `true` indicates that Kerberos authentication is enabled. Information about the available Kerberos realms can be found in the `kerberos` property.

If a client program needs to use Maconomy credentials in a Kerberos authentication enabled system, this has to be indicated by inclusion of the following HTTP request header:

```
Maconomy-Authentication: X-Force-Maconomy-Credentials
```

Such a request header indicates to the server that any credentials sent with the request are Maconomy credentials.

Kerberos Single Sign-On

If a Maconomy system is set up to use Kerberos Single Sign-On (SSO), the web service offers authentication via the Negotiate mechanism [7]. The purpose of this mechanism is to allow SSO by letting the client program, for example, the user's web browser, obtain a Kerberos ticket for the web service without user interaction. The web service forwards these credentials to the Maconomy system for verification.

2.8.4 OpenID Authentication

If a Maconomy system is set up to use the OpenID Connect protocol [10] for authentication, the web service accepts authorization codes issued by the configured OpenID provider (for example, Microsoft Azure). This section assumes basic familiarity with the OpenID protocol and in particular with the Authorization Code Flow [10, Section 3.1].

As described in Installation, the state of an installation resource includes information about enabled authentication schemes, and in case of an openID Connect enabled system,

this reveals all relevant metadata required to initiate an Authorization Code Flow. For example:

```
{
  "useDomainCredentialsForBasicAuthentication": false,
  "schemes": {
    ...,
    "x-oidc-code": {
      "name": "x-oidc-code"
    },
    ...
  },
  "openIDProviders": [
    {
      "authorizationEndpoint": "https://login.microsoftonline.com/d2a26c48 ↩
    -d40f-4406-8a62-68073368e07c/oauth2/authorize",
      "redirectURI": "https://login.microsoftonline.com/common/oauth2/ ↩
    nativeclient",
      "clientID": "29074461-0743-4bc2-a7cc-1e983ac3f2e7"
    }
  ],
  ...
}
```

The properties `authorizationEndpoint`, `redirectURI`, and `clientID` tell the client program how to initiate an authentication request towards the given OpenID provider (in this case Microsoft Azure) using the Authorization Code Flow.

The `redirectURI` property holds a redirect URI which is guaranteed to be accepted by the OpenID provider and typically resolves to an empty web page. Such a redirect URI can be used by so-called native clients having full control over an embedded user agent and hence the ability to extract values returned via query or fragment parameters directly from the location of the user agent. The Workspace Client is an example of such a client, but in principle a smartphone app could operate in the same way. All non-native clients (such as pure web apps) have to use a redirect URI of their own that has been pre-registered with the OpenID provider.

Once the user has successfully authenticated with the identity provider and the client program has obtained an authorization code, this code can be used as one-time authentication credentials using the `X-OIDC-Code` authorization scheme. The string put into the `Authorization` request header must then follow the `OIDC-Credentials` production rule given by the following grammar:

```
OIDC-Credentials = "X-OIDC-Code" SP Authz-Cookie
Authz-Cookie     = <base64-encoded Authz-Grant (no newlines)>
Authz-Grant      = "<" Redirect-URI ">" ":" Authz-Code
Redirect-URI     = <URI-Reference, see [RFC3986], Section 4.1>
Authz-Code       = *TEXT
```

For example, if a client program has obtained authorization code `AABAQE1_2345` by use of redirect URI `https://example.com/oauth2/authorize`, then the header to include in the request will look like this:

```
Authorization: X-OIDC-Code ↩
    PGh0dHBzOi8vZXhhbXBsZS5jb20vb2F1dGggyL2F1dGhvcml6ZT46QUFCQVFFMV8yMzQ1
```

Here, the base64-encoded string following the authentication scheme token `X-OIDC-Code` encodes the following string:

```
<https://example.com/oauth2/authorize>:AABAQE1_2345
```

Note that since OpenID credentials can only be used once, it is wise to also include a `Maconomy-Authentication` header holding a reconnect directive in order to obtain reconnect credentials to be used with subsequent requests (see Maconomy Reconnect Authentication).

2.8.5 Two-Factor Authentication

If the Maconomy Two-Factor Authentication (2FA) system is enabled, a client program must provide a second authentication factor along with the standard `Authorization` HTTP header. Such second factor consists of a six-digit One-Time Password (OTP) generated by a TOTP-compatible program running on some device, usually in the form of a smartphone app (the first T in TOTP stands for time-based).

Since an OTP can be used only once, the user has to supply a new OTP for each and every request. This is, unless the client program provides a header with the Maconomy reconnect directive and uses reconnect tokens for authentication on subsequent requests (see Maconomy Reconnect Authentication). No OTPs are required when using Maconomy reconnect tokens.

The custom header field `Maconomy-OTP` is the one to be used by the client when sending OTPs to the server. This header field is also the one used by the server when details about authentication failures (OTP is found missing or invalid) need to be communicated back to the client.

The format of a `Maconomy-OTP` header when included in a *request* (always along with an `Authorization` header) can be described by the following ABNF:

```
Maconomy-OTP-Request  = "Maconomy-OTP" ":" otp-request-directive
otp-request-directive = "authenticate" ";" "otp" "=" 1*DIGIT
                      / "reset" [";" "method" "=" otp-reset-method]
                                [";" "token" "=" reset-token]
otp-reset-method      = quoted-string
reset-token           = quoted-string
```

The format of a `Maconomy-OTP` header when included in a *response* (always along with a `401 Unauthorized` status) can be described by the following ABNF:

```
Maconomy-OTP-Response  = "Maconomy-OTP" ":" otp-response-directive
otp-response-directive = "required" [";" "reset" "=" otp-reset-method]
                                    [";" "enroll" "=" "<" URI-Reference ">"]
otp-reset-method       = quoted-string
URI-Reference          = <URI-reference, see [RFC3986], Section 4.1>
```

The `authenticate` Directive

The `authenticate` directive with an `otp` parameter in a `Maconomy-OTP` request header is used by the client to provide an OTP as a second authentication factor. For example, for the OTP `012345`, the following custom header should be included in the request:

```
Maconomy-OTP: authenticate;otp=012345
```

The `reset` Directive

A `Maconomy-OTP` request header may also contain a `reset` directive, which is intended for the cases where a user wants OTP settings reset in order to allow for re-enrollment with a new 2FA device. As sending a reset token to the user via a trusted channel is part of such reset procedure, the trusted channel preferred by the user can be specified in the request header using a `method` parameter. For example, for email:

```
Maconomy-OTP: reset;method=email-token
```

As `email-token` is in fact the only reset method currently supported by Maconomy, this method is also the one being applied by default.

If a requested reset procedure is successfully initiated (the `Authorization` header successfully authenticates the user and an email address owned by the user is familiar to the system), the server sends back a response with the following `Maconomy-OTP` header:

```
Maconomy-OTP: required;reset=email-token
```

The body of this response contains a message instructing the user to find a reset token in an email sent to the user by the server. To finalize the reset procedure, the reset token received must to be included in a subsequent request using a `token` parameter of the `reset` directive (notice the double quotes around the token):

```
Maconomy-OTP: reset;token="<token>"
```

The server's response to such reset token request contains a handy enrollment header, see the `required` directive.

The `required` Directive

Whenever an OPT is required, but the client submits a request with either no OTP (this is the case for reset requests, for example) or with an invalid OTP, the server sends back a `401 Unauthorized` response. This response carries an appropriate description of the situation in its body and holds a `Maconomy-OTP` header with the `required` directive.

If an out-of-band reset process is currently in progress, the `required` directive is accompanied by a `reset` parameter specifying a reset method (see the `reset` directive), whereas an `enroll` parameter is present if the user has no security token device currently enrolled.

An `enroll` parameter in a `Maconomy-OTP` response header holds an enrollment URI pointing to a Maconomy TOTP key resource. For example:

```
Maconomy-OTP: required;enroll=<http://SERVER/maconomy-api/auth/macoprod/ ↩
    totp/keyURI?account=Administrator&secret=FY5lYCVs8xIbqx8YDENI7OnupSs%3 ↩
    D>
```

By following such an enrollment URI, the user is able to configure a preferred security token device which can then provide all the OTPs ever needed.

TOTP Key Resource

A TOTP key resource is a web service endpoint that serves a shared TOTP secret in different formats suitable for enrollment with a compatible device. URIs to key resources are discovered via a `Maconomy-OTP` response header (see the `required` directive).

A TOTP key resource only supports the HTTP verb `GET` and no authentication is required. Furthermore, an `Accept` request header holding one of the following values is required:

Value	Explanatory text
`image/*`, `image/png`	A PNG image of a QR code encoding the shared TOTP secret as a key URI is returned. The code is suitable for scanning with a range of compatible smartphone apps.
`application/json`	A JSON object containing the key URI in plain text is returned.

A JSON object returned from a TOTP key resource contains the following fields:

Field	Explanatory text
`totp-key-uri`	A key URI encoding the shared TOTP secret.

URIs to TOTP key resources encode sensitive information, and thus client programs *must* take measures to ensure that these URIs are not stored in browser histories or elsewhere where an adversary could get access to them.

2FA Example

To exemplify the concepts of 2FA described above, let us imagine an `ExpenseSheets` container filter request (see Filtering) towards the Containers Web Service in a 2FA enabled Maconomy system (shortname `macoprod`):

```
$ curl -i
       -u 'Administrator:123456'
       -H 'Accept-Language: en-US'
       -H 'Accept: application/vnd.deltek.maconomy.containers+json;  ↩
    charset=utf-8; version=6.0'
       'http://SERVER/maconomy-api/containers/macoprod/expensesheets/ ↩
    filter'

HTTP/1.1 401 Unauthorized
Maconomy-OTP: required;enroll=<http://SERVER/maconomy-api/auth/macoprod/ ↩
    totp/keyURI?account=Administrator&secret=FY5lYCVs8xIbqx8YDENI7OnupSs%3 ↩
    D>
WWW-Authenticate: Basic realm="Maconomy"
Content-Type: application/json; charset=utf-8

{
  "errorMessage": "Mandatory two-factor authentication must be configured ↩
    .\nPlease scan the QR code using a supported smartphone app.",
  "errorFamily": "service",
  "errorSeverity": "error"
}
```

Here, the `Maconomy-OTP` header on the response indicates that the user has not yet enrolled a security token device and that the following TOTP key URI should be followed in order to do so:

```
http://SERVER/maconomy-api/auth/macoprod/totp/keyURI?account=Administrator ↩
    &secret=FY5lYCVs8xIbqx8YDENI7OnupSs%3D
```

Using a compatible smartphone app, the user is supposed to scan the QR code in the PNG image available at this URI and finish off the configuration of a security token app.

When accomplished, the user is able to generate an OTP, say `980461`, and repeat the request with a `Maconomy-OTP` header carrying this OTP:

```
$ curl -i
       -u 'Administrator:123456'
       -H 'Maconomy-OTP: authenticate;otp=980461'
       -H 'Maconomy-Authentication: X-Reconnect'
       -H 'Accept-Language: en-US'
       -H 'Accept: application/vnd.deltek.maconomy.containers+json;  ↩
    charset=utf-8; version=6.0'
       'http://SERVER/maconomy-api/containers/macoprod/expensesheets/ ↩
    filter'

HTTP/1.1 200 OK
Maconomy-Reconnect: MDVkMmZkOT...JMTAJMTYwODAyMTk3Ng==
Content-Type: application/vnd.deltek.maconomy.containers+json; charset=utf ↩
    -8; version=6.0

{ ... }
```

Because a `Maconomy-Authentication` header with the reconnect directive `X-Reconnect` was also included on the request, the response includes a `Maconomy-Reconnect` header with a reconnect token. This reconnect token enables OTP free authentication on a subsequent request (see Maconomy Reconnect Authentication):

```
$ curl -i
       -H 'Authentication: X-Reconnect MDVkMmZkOT...JMTAJMTYwODAyMTk3Ng==='
       -H 'Accept-Language: en-US'
       -H 'Accept: application/vnd.deltek.maconomy.containers+json;  ↩
    charset=utf-8; version=6.0'
       'http://SERVER/maconomy-api/containers/macoprod/expensesheets/ ↩
    filter'

HTTP/1.1 200 OK
Maconomy-Reconnect: MDVkMmZkOT...JMTAJMTYwODAyMTk3Ng==
Content-Type: application/vnd.deltek.maconomy.containers+json; charset=utf ↩
    -8; version=6.0

{ ... }
```

With a security token device enrolled for the user, a TOTP key URI is no longer included as an `enroll` parameter to the `required` directive in the `Maconomy-OTP` header set on the response to a request carrying neither an OTP nor a reconnect token:

```
$ curl -i
       -u 'Administrator:123456'
       -H 'Accept-Language: en-US'
       -H 'Accept: application/vnd.deltek.maconomy.containers+json;  ↩
    charset=utf-8; version=6.0'
```

```
       'http://SERVER/maconomy-api/containers/macoprod/expensesheets/ ↩
    filter'

HTTP/1.1 401 Unauthorized
Maconomy-OTP: required
WWW-Authenticate: Basic realm="Maconomy"
Content-Type: application/json; charset=utf-8

{
  "errorMessage": "Two-factor authentication required.",
  "errorFamily": "service",
  "errorSeverity": "error"
}
```

Also, if the user at some point no longer wishes to (or is able to) use the enrolled security token device, a reset procedure can be initiated by submitting an authenticated request with a `Maconomy-OTP` header holding the `reset` directive:

```
$ curl -i
       -u 'Administrator:123456'
       -H 'Maconomy-OTP: reset'
       -H 'Accept-Language: en-US'
       -H 'Accept: application/vnd.deltek.maconomy.containers+json;  ↩
    charset=utf-8; version=6.0'
       'http://SERVER/maconomy-api/containers/macoprod/expensesheets/ ↩
    filter'

HTTP/1.1 401 Unauthorized
Maconomy-OTP: required
WWW-Authenticate: Basic realm="Maconomy"
Content-Type: application/json; charset=utf-8

{
  "errorMessage": "Enter Token [15d7e2d749f92a3340e61d336ea]",
  "errorFamily": "service",
  "errorSeverity": "error"
}
```

As part of such a reset procedure, a reset token would normally be sent to the user via email (the default trusted channel). However, in our example here, the reset procedure initiated is a simplified procedure for demonstration purposes and therefore a reset token has been included directly in the response body.

The user is now able to finalize the reset procedure by including the received reset token `15d7e2d749f92a3340e61d336ea` as a `token` parameter to the `reset` directive in a `Maconomy-OTP` header (note the double quotes around the token) on an authenticated request:

```
$ curl -i
       -u 'Administrator:123456'
       -H 'Maconomy-OTP: reset;token="15d7e2d749f92a3340e61d336ea"'
       -H 'Accept-Language: en-US'
       -H 'Accept: application/vnd.deltek.maconomy.containers+json;  ↩
    charset=utf-8; version=6.0'
       'http://SERVER/maconomy-api/containers/macoprod/expensesheets/ ↩
    filter'

HTTP/1.1 401 Unauthorized
Maconomy-OTP: required;enroll=<http://SERVER/maconomy-api/auth/macoprod/ ↩
    totp/keyURI?account=Administrator&secret=PhImtxfDQI9l7RTMJ50ONBLJFgY%3 ↩
    D>
WWW-Authenticate: Basic realm="Maconomy"
Content-Type: application/json; charset=utf-8

{
  "errorMessage": "Mandatory two-factor authentication must be configured ↩
    .\nPlease scan the QR code using a supported smartphone app.",
  "errorFamily": "service",
  "errorSeverity": "error"
}
```

The user is now back where this example began.

Note that in any real world example, the reset token should never be sent directly back to the user. By doing so, an attacker in possession of the user name and the password would be able to remove the OTP-generating token and enroll the attacker's own token. The security of using OTPs would be completely undermined.

2.9 Request Identification in APM Logs

All requests originating from the Maconomy RESTful web services are assigned a *request id*. This request id can be supplied by the client program via a `Maconomy-RequestId` header on the request:

```
Maconomy-RequestId: REQUEST_ID
```

If no such `Maconomy-RequestId` header is included, the server supplies a request id. In both cases, the assigned request id is communicated back in a `Maconomy-RequestId` header on the response.

Note that request ids supplied be the server are unique across all requests, and that any request id supplied by the client program should be one previously received from the server.

The purpose of request ids is to ease debugging. All log entries within the Maconomy

Application Performance Monitoring (APM) framework refer to a request id, and by giving the client program control over which id is assigned to a request, it becomes possible to have log entries relating to the same user interaction refer to the same request id. Request ids carefully assigned in this way allow the APM to make timing and statistics based on user interactions rather than single requests.

For example, to associate an instance creation with a subsequent data entry load request for the `ExpenseSheets` container using the Containers Web Service:

```
$ curl -i
       -u 'Administrator:123456'
       -H 'Maconomy-Authentication: X-Reconnect'
       -H 'Accept-Language: en-US'
       -H 'Accept: application/vnd.deltek.maconomy.containers+json;  ↩
    charset=utf-8; version=6.0'
       -X POST
       'http://SERVER/maconomy-api/containers/macoprod/expensesheets/ ↩
    instances'

HTTP/1.1 200 OK
Maconomy-Reconnect: Zjk1YjUzMT...JMTAJMTYwMjY4MDE4NQ==
Maconomy-Concurrency-Control: d2a39243-a63f-4bd5-8eab-676952009e93
Maconomy-RequestId: 58d646d7-f2e5-4837-a1de-c3ef6fab9fe5
Content-Type: application/vnd.deltek.maconomy.containers+json; charset=utf ↩
    -8; version=6.0

{
  "meta": {
    "containerName": "expensesheets",
    "containerInstanceId": "1701829c-7a34-464c-a7d4-e7f1d9a44537"
  },
  "links": {
    ...,
    "data:any-key": {
      "href": "http://SERVER/maconomy-api/containers/macoprod/ ↩
    expensesheets/instances/1701829c-7a34-464c-a7d4-e7f1d9a44537/data;any ↩
    ",
      "rel": "data:any-key"
    },
    ...
  }
}
```

Here, no `Maconomy-RequestId` request header was included by the client program and thus the server generated a new unique id for the request. This id appears in the `Maconomy-RequestId` header on the response and can now be included on a subsequent data entry load request:

```
$ curl -i
       -H 'Authorization: X-Reconnect Zjk1YjUzMT...JMTAJMTYwMjY4MDE4NQ=='
       -H 'Accept-Language: en-US'
       -H 'Accept: application/vnd.deltek.maconomy.containers+json;  ↩
    charset=utf-8; version=6.0'
       -H 'Maconomy-Concurrency-Control: d2a39243-a63f-4bd5-8eab-676952009 ↩
    e93'
       -H 'Maconomy-RequestId: 58d646d7-f2e5-4837-a1de-c3ef6fab9fe5'
       -X POST
       'http://SERVER/maconomy-api/containers/macoprod/expensesheets/ ↩
    instances/1701829c-7a34-464c-a7d4-e7f1d9a44537/data;any'

HTTP/1.1 200 OK
Maconomy-Reconnect: YTg1ZGY3Ym...wCTEwCTE2MDYyMzE3Mzg=
Maconomy-Concurrency-Control: 8db720b5-015d-4cfe-926a-3edf96f3e724
Maconomy-RequestId: 58d646d7-f2e5-4837-a1de-c3ef6fab9fe5
Content-Type: application/vnd.deltek.maconomy.containers+json; charset=utf ↩
    -8; version=6.0

{ ... }
```

As expected, the request id received in the `Maconomy-RequestId` header on the response matches the request id passed in the `Maconomy-RequestId` request header. In this case, the server did not assign a new id to the request, but simply used the one passed by the client program.

2.10 Client Identification in APM Logs

All requests originating from the Maconomy RESTful web services are assigned a *client name* whose value can be controlled by the client program via a `Maconomy-Client` header included on the request:

```
Maconomy-Client: CLIENT_NAME
```

The value of this header is included in the `Client` field on all log entries generated for the request within the Maconomy Application Performance Monitoring (APM) framework. If no `Maconomy-Client` header is included, the `Client` fields are populated with the value `Web`.

2.11 Status Codes and Errors

Each response from a Maconomy RESTful web service contains an HTTP status code telling whether the request was successful or not. If the request was unsuccessful, the status code indicates what kind of failure occurred and can be used by the client program to decide on how to proceed.

Most people have encountered the `404 Not Found` status while browsing the web. The three-digit integer status code `404` is the significant part used by the client program to categorize the error. The status text `Not Found` is called the *reason phrase* and is there to help humans understand the error. The numeric status code is standardized and has a particular meaning, while the reason phrase may differ between web server software, may be localized, and so on.

Status codes are categorized into *status families* by their first digit:

Status Codes	Status Family	Explanatory text
`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 by the Maconomy RESTful web services:

Status Code	Reason phrase	Explanatory text
`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.
`204`	`No Content`	The request has succeeded but the response contains no content.

Status Code	Reason phrase	Explanatory text
400	`Bad Request`	The request body or headers contained malformed or incomplete information. This usually indicates a programming error in the client program.
401	`Unauthorized`	The request requires user authentication and thus the client program must retry the request with valid credentials. See Authentication.
403	`Forbidden`	The requested resource or action is not permitted with the supplied credentials.
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 verb applied by the client program is not allowed for the resource. For example, a resource may not support either the `GET`, the `POST`, or the `DELETE` verb.
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 may indicate that the resource was updated by another user and the client may thus refresh its current state of the resource and retry the request.

Status Code	Reason phrase	Explanatory text
413	`Request Entity Too Large`	The request body 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	`Unprocessable Entity`	The request could not be completed because of violated application business logic.
500	`Internal Server Error`	A catch-all status code for unexpected errors.

Fielding et al. [7] contains a detailed specification of the semantics of each of the status codes, except for `422 Unprocessable Entity` which is adopted from Dusseault [6].

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.

2.11.1 Error Responses

When an error occurs, the HTTP status code is typically used by client programs to dispatch to the error handling 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 body of the response to an unsuccessful request contains a descriptive message along with other metadata that can be useful in signalling the error.

The standard properties found in a JSON object enclosed in an error response are these:

Property	Explanatory text
`errorFamily`	Name of the error family to which the error belongs. Find the possible values below.
`errorSeverity`	Indicates the severity of the error. Find the possible values below.

Property	Explanatory text
`errorMessage`	Error message appropriate for displaying in a user interface or for reporting in some other way.
`errorId`	Optional. Id uniquely identifying the error. It makes sense for the client program to recognize the error using such error id instead of using the unstable and possibly localized error message.

`errorIds`	List of overridden ids for any error id held in `errorId`. If the error has no id assigned, the `errorIds` list will be the empty one.
	The order of the ids in the `errorIds` list goes from most specific to more general, and any id held in `errorId` will be listed first.
	Imagining the following pair of properties for some error:
	`"errorId": "B"`
	`"errorIds": ["B", "A"]`
	Then, if a more specific error id, `C`, is introduced, the error response will start carrying the following properties instead:
	`"errorId": "C"`
	`"errorIds": ["C", "B", "A"]`
	In this way, the client program will still be able to recognize the error as a `B` error.

The possible values of the `errorFamily` property are:

Error family	Explanatory text
`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 some other condition not caused by the business logic. Interacting with a time sheet that has been changed or deleted by someone else would for example trigger a service error.
`internal`	An internal error is an unexpected error that may indicate a problem in the system setup or a bug in the web service. The server log files usually contain messages indicating the underlying cause. An example of this could be that the database is not running.

The possible values of the `errorSeverity` property are:

Error severity	Explanatory text
`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.

For example, if a client program tried to register 30 hours on a Monday in a time sheet using the Containers Web Service, it would get back a `422 Unprocessable Entity` response with something like this in the body:

```
{
  "errorFamily": "application",
  "errorSeverity": "error",
  "errorMessage": "You cannot enter more than 24 hours for a day",
  "errorId": "A-9e047846",
  "errorIds": [
    "A-9e047846",
    "S-cedeca11",
    "S-bad7ac0b"
  ],
  "focus": {
    "paneName": "table",
    "rowNumber": 0,
    "fieldName": "numberday1"
  }
}
```

The `focus` property is special for the Containers Web Service and is present because the error relates to a particular record field. The idea is to allow the client program to put focus in the problematic field and thereby help the user identify the cause of the error. The problematic field can be identified by the information held in `paneName`, `rowNumber`, and `fieldName`.

Chapter 3

Root Web Service

The Maconomy RESTful Root Web Service is where it all begins. Starting from its root resource available at path `/BASEPATH` (`BASEPATH` being the information passed by the reverse proxy in a `Maconomy-Forwarded-Base-Path` header, see Proxy Requirements), a client program should be able to discover all parts of the enabled Maconomy RESTful web services by following hyperlinks. As described further in below sections, the Root Web Service also exposes useful information like available languages and enabled authentication schemes.

Now, this is the custom media type covering the JSON representations within the encompassed version of the Root Web Service (see Media Types):

```
application/vnd.deltek.maconomy.root+json; charset-utf-8; version=2.1
```

Starting at the root resource, these are the possible properties of an acquired JSON representation:

Property	Explanatory text
`timeInfo`	Information about the current point in time according to the server.
`versions`	Version information regarding the installed system. This information can be disabled (see Version Information).
`languages`	Languages available in the system. See Language.
`installations`	Information about the databases installed in the system. The `shortname` of each installation can replace the `{shortname}` placeholder in the template URL of the `installation` hyperlink mentioned in the `links` property. The information regarding installations can be disabled (see Shortnames).

Property	Explanatory text
`links`	Hyperlinks available from the root resource. Find the list of link relations below.

The purposes of the hyperlinks available through the `links` property listed above are explained here:

Link relation	Explanatory text
`handshake1`	Reference to the handshake resource version 1. See Handshake.
`installation`	Reference to the root resource of some installation. See Installation.
`diagnostics`	Reference to the root resource of the Diagnostics Web Service.
`self`	Reference to the root resource itself.

For example, for the base path `/maconomy-api`:

```
$ curl -i
       -H 'Accept: application/vnd.deltek.maconomy.root+json; charset=utf ↩
    -8; version=2.1'
       'http://SERVER/maconomy-api'

HTTP/1.1 200 OK
Content-Type: application/vnd.deltek.maconomy.root+json; charset=utf-8;  ↩
    version=2.1

{
  "timeInfo": {
    "time": "2020-10-20T12:43:28.462Z[UTC]",
    "zone": {
      "id": "UTC",
      "offset": {
        "id": "Z",
        "totalSeconds": 0
      }
    }
  },
  "versions": {
    "apu": {
      "major": "21",
      "sp": "102",
      "hotfix": "0",
      "build": "99999999"
```

```
    },
    "tpu": {
      "major": "21",
      "sp": "102",
      "hotfix": "0",
      "build": "99999999"
    }
  },
  "languages": [
    {
      "title": "Dansk (Danmark)",
      "locale": "da_DK",
      "tag": "da-DK"
    },
    {
      "title": "English (United States)",
      "locale": "en_US",
      "tag": "en-US"
    }
  ],
  "installations": [
    {
      "shortname": "macoprod",
      "company": "Foo"
    }
  ],
  "links": {
    "handshake1": {
      "href": "http://SERVER/maconomy-api/handshake/1",
      "rel": "handshake1"
    },
    "installation": {
      "template": "http://SERVER/maconomy-api/installations/{shortname}",
      "rel": "installation"
    },
    "diagnostics": {
      "href": "http://SERVER/maconomy-api/diagnostics",
      "rel": "diagnostics"
    },
    "self": {
      "href": "http://SERVER/maconomy-api",
      "rel": "self"
    }
  }
}
```


In this case the root resource JSON reveals that only one database installation is available in the system, and only the languages Danish and US English are currently

supported.

The `handshake1` and the `installation` resource reachable from the root resource are described in the next two sections respectively.

3.1 Handshake

The Workspace Client is able to discover information about how to connect to the Maconomy server by following the hyperlink with link relation `handshake1`:

```
{
  "href": "http://SERVER/BASEPATH/handshake/1",
  "rel": "handshake1"
}
```

This is only for internal use by the Workspace Client and is not described in any further detail in this document. Please note that the information exposed by the handshake service is subject to change without further notice.

3.2 Installation

To discover the authentication schemes and RESTful web services available for some installation among the ones listed in the `installations` property of the root resource representation described above, one should follow the available hyperlink with link relation `installation`, substituting an appropriate installation `shortname` into the `{shortname}` placeholder:

```
{
  "template": "http://SERVER/BASEPATH/installations/{shortname}",
  "rel": "installation"
}
```

These are the possible properties of a JSON object representing an installation resource:

Property	Explanatory text
`authentication`	Information about enabled authentication schemes. See Authentication.
`links`	Hyperlinks available from the installation resource. Find the list of link relations below.

These are the purposes of the hyperlinks available from an installation resource:

Link relation	Explanatory text
`containers`	Reference to the root resource of the Containers Web Service.
`popups`	Reference to the root resource of the Popup Types Web Service.
`filedrop`	Reference to the root resource of the File Drop Web Service.
`logging`	Reference to the root resource of the Logging Web Service.
`configurations`	Reference to the root resource of the Configurations Web Service.
`environment`	Reference to the root resource of the Environment Web Service.
`usersettings`	Reference to the root resource of the User Settings Web Service.
`authentication`	Reference to the root resource of the Authentication Web Service.
`messages`	Reference to the root resource of the Messages Web Service.
`analyzer`	Reference to the root resource of the Analyzer Web Service.
`self`	Reference to the installation resource itself.

Note that the Popup Types, Configurations, Environment, and Analyzer Web Services are not described in this document and their functionality may change in the future.

For example:

```
$ curl -i
       -H 'Accept-Language: en-US'
       -H 'Accept: application/vnd.deltek.maconomy.root+json; charset=utf ↩
    -8; version=2.1'
       'http://SERVER/maconomy-api/installations/macoprod'

HTTP/1.1 200 OK
Content-Type: application/vnd.deltek.maconomy.root+json; charset=utf-8;  ↩
    version=2.1

{
  "authentication": {
    "useDomainCredentialsForBasicAuthentication": true,
    "kerberos": {
      "kdc": "PSO-DC.PSO.COM",
      "realm": "PSO.COM",
      "realms": {
        "PSO.COM": {
          "kdc": "PSO-DC.PSO.COM",
          "name": "PSO.COM"
        }
      },
```

```
      "serviceName": "MACONOMYSSO/PSO.COM"
    },
    "schemes": {
      "basic": {
        "name": "basic"
      },
      "x-changepassword": {
        "name": "x-changepassword"
      },
      "x-reconnect": {
        "name": "x-reconnect"
      },
      "x-cookie": {
        "name": "x-cookie"
      }
    }
  },
  "links": {
    "containers": {
      "href": "http://SERVER/maconomy-api/containers/macoprod",
      "rel": "containers"
    },
    "popups": {
      "href": "http://SERVER/maconomy-api/popups/macoprod",
      "rel": "popups"
    },
    "filedrop": {
      "href": "http://SERVER/maconomy-api/filedrop/macoprod",
      "rel": "filedrop"
    },
    "logging": {
      "href": "http://SERVER/maconomy-api/logging/macoprod",
      "rel": "logging"
    },
    "configurations": {
      "href": "http://SERVER/maconomy-api/configurations/macoprod",
      "rel": "configurations"
    },
    "environment": {
      "href": "http://SERVER/maconomy-api/environment/macoprod",
      "rel": "environment"
    },
    "usersettings": {
      "href": "http://SERVER/maconomy-api/usersettings/macoprod",
      "rel": "usersettings"
    },
    "authentication": {
      "href": "http://SERVER/maconomy-api/auth/macoprod",
      "rel": "authentication"
```

```
    },
    "messages": {
        "href": "http://SERVER/maconomy-api/messages/macoprod",
        "rel": "messages"
    },
    "analyzer": {
      "href": "http://SERVER/maconomy-api/analyzer/macoprod",
      "rel": "analyzer"
    },
    "self": {
      "href": "http://SERVER/maconomy-api/installations/macoprod",
      "rel": "self"
    }
  }
}
```

In the system of this example, all the Maconomy RESTful web services have been enabled, and hyperlinks allowing the client program to discover these services are available in the `links` property.

The following chapters address the RESTful web services possibly discoverable from an installation resource.

Chapter 4

Containers Web Service

The Maconomy RESTful Containers Web Service exposes data and functionality through so-called *containers* which are an abstraction giving a uniform interface to all functionality within the Maconomy system. Each container is made up of a number of *panes* where the following three types of panes are available:

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

Table panes which contain zero or more records. A table pane where the ordering of the records is based on assigned line numbers is said to have *line-number control*. When the records of such a table pane are hierarchically organized, the pane is called a *tree table pane*. In Maconomy, the expense sheet lines part of the `ExpenseSheets` container is an examples of a line-number controlled table pane, and the job budget lines part of the `JobBudgets` container is an example of a tree table pane.

Filter panes which, like tables, contain zero or more records. Filter panes present available data entries and allow the client program to select subsets of the potential content by applying certain restrictions. See Filtering.

The containers currently supported by the Containers Web Service are the ones whose panes structure fulfills the following conditions:

- At least one pane must be defined.
- At most one pane of each type must be defined.
- If a table pane is defined, then a card pane must also be defined.

That is, a container having, for example, a filter and a table pane but no card pane defined will not be accessible through the Containers Web Service.

The following complementary terms are used throughout this document:

Data container denotes any container for which a card and possibly a table pane are defined. Most data containers also have a filter pane defined.

Popup container denotes any container for which only a filter pane is defined and where the records of this filter pane are the values of some enum type.

Search container denotes any container for which only a filter pane is defined and which is used for searching in Maconomy (see Foreign Keys).

Now, to begin the survey of the functionality of the Containers Web Service, this is the custom media type covering the JSON representations within the encompassed version of the service (see Media Types):

```
application/vnd.deltek.maconomy.containers+json; charset=utf-8; version ↩
    =6.0
```

Earlier versions of the service are not addressed in any further detail in this document.

As mentioned in the previous chapter (see Installation), the root resource of the Containers Web Service can be accessed by following the hyperlink with link relation `containers` available from an installation resource:

```
{
  "href": "http://SERVER/BASEPATH/containers/SHORTNAME",
  "rel": "containers"
}
```

These are the properties that are present in an acquired JSON representation of such root resource:

Property	Explanatory text
`containerNames`	Names of the containers accessible through the Containers Web Service. Each of these names can replace the `{container}` placeholder in the template URL of the `container` hyperlink mentioned in the `links` property.
`links`	Hyperlinks available from the root resource of the Containers Web Service. Find the list of link relations below.

The purposes of the hyperlinks available through the `links` property listed above are these:

Link relation	Explanatory text
`container`	Reference to a specific container resource. Find a description of this below.
`self`	Reference to the root resource of the Containers Web Service itself.

For example, acquiring a representation of the `macoprod` system's Containers Web Service root resource:

```
$ curl -i
       -H 'Accept-Language: en-US'
       -H 'Accept: application/vnd.deltek.maconomy.containers+json;  ↩
    charset=utf-8; version=6.0'
       'http://SERVER/maconomy-api/containers/macoprod'

HTTP/1.1 200 OK
Content-Type: application/vnd.deltek.maconomy.containers+json; charset=utf ↩
    -8; version=6.0

{
  "containerNames": [
    ...,
    "employees",
    ...,
    "expensesheets",
    ...,
    "jobs",
    ...
  ],
  "links": {
    "container": {
      "template": "http://SERVER/maconomy-api/containers/macoprod/{ ↩
    container}",
      "rel": "container"
    },
    "self": {
      "href": "http://SERVER/maconomy-api/containers/macoprod",
      "rel": "self"
    }
  }
}
```

Here, `Employees`, `ExpenseSheets`, and `Jobs` are a few examples of Maconomy containers accessible through the Containers Web Service in the system approached.

To interact with a specific container among the ones listed in the `containerNames`

property, one should follow the hyperlink with link relation `container`, substituting the name of the chosen container into the `{container}` placeholder:

```
{
  "template": "http://SERVER/BASEPATH/containers/SHORTNAME/{container}",
  "rel": "container"
}
```

These are the properties of a JSON object representing a container resource:

Property	Explanatory text
`meta`	`containerName`: Name of the container.
`links`	Hyperlinks available from the container resource. Find the list of link relations below.

The purposes of the hyperlinks available through the `links` property listed above are these:

Link relation	Explanatory text
`specification`	Reference to the specification resource of the container. See Specification.
`access`	Reference to the access resource of the container. See Access.
`data:filter`	Reference to the filter resource of the container. See Filtering.
`instance:create`	Reference to the action of creating an instance of the container. See Container Instances.
`self`	Reference to the container resource itself.

For example, for the `ExpenseSheets` container:

```
$ curl -i
       -H 'Accept-Language: en-US'
       -H 'Accept: application/vnd.deltek.maconomy.containers+json;  ↩
    charset=utf-8; version=6.0'
       'http://SERVER/maconomy-api/containers/macoprod/expensesheets'

HTTP/1.1 200 OK
Content-Type: application/vnd.deltek.maconomy.containers+json; charset=utf ↩
    -8; version=6.0
```

```
{
  "meta": {
    "containerName": "expensesheets"
  },
  "links": {
    "specification": {
      "href": "http://SERVER/maconomy-api/containers/macoprod/ ↩
    expensesheets/specification",
      "rel": "specification"
    },
    "access": {
      "href": "http://SERVER/maconomy-api/containers/macoprod/ ↩
    expensesheets/access",
      "rel": "access"
    },
    "data:filter": {
      "href": "http://SERVER/maconomy-api/containers/macoprod/ ↩
    expensesheets/filter",
      "rel": "data:filter"
    },
    "instance:create": {
      "href": "http://SERVER/maconomy-api/containers/macoprod/ ↩
    expensesheets/instances",
      "rel": "instance:create"
    },
    "self": {
      "href": "http://SERVER/maconomy-api/containers/macoprod/ ↩
    expensesheets",
      "rel": "self"
    }
  }
}
```


The following four sections each relates to one of the hyperlinks available from a container resource, namely to the `specification`, the `access`, the `data:filter`, and the `instance:create` hyperlink respectively.

4.1 Specification

Every container accessible through the Containers Web Service interface has a *specification* resource.

The specification resource can be used to programmatically determine the following:

- Names, titles, and entities of the panes in the container.
- Whether a table pane is a tree table pane with hierarchically organized records.

- Whether the records of a line-number controlled table pane can be moved around only inside its current context or across the entire table pane.
- Names and titles of the actions supported by each pane.
- Names, titles, and data types of the fields comprising the records in each pane.
- Foreign keys of each pane relating their records to other records.
- Relevant hyperlinks for other containers somehow related to the container.

For example, to correctly interpret and manipulate records in the panes of a container, a client program must read the specification resource to gain the necessary knowledge about the data types of the record fields.

A representation of the specification resource for a container is acquired by following the hyperlink with link relation `specification` available from the container's root resource:

```
{
  "href": "http://SERVER/BASEPATH/containers/SHORTNAME/CONTAINER/ ↩
    specification",
  "rel": "specification"
}
```

The response to a `specification` request is called a *specification response*, and the JSON object held in its body has the following properties:

Property	Explanatory text
`meta`	`containerName`: Name of the container.
`panes`	Specification of the panes defined for the container. The properties of the object held for each pane are described in the table below.
`relatedContainers`	References to other containers that are considered related to the container. See Related Containers.
`links`	`self`: Hyperlink referring to the container specification resource itself.

The possible properties of the value of the `panes` property are `card`, `table`, and `filter`, representing a card, a table and a filter pane respectively. Their values are objects describing a pane of the given type:

Property	Explanatory text
`paneName`	Name of the pane.
`title`	Title of the pane. A human-readable text appropriate for displaying in a user interface.
`entity`	Name of the entity containing the data of the pane. This information about which entity the pane is based on is relevant when doing foreign key look-ups and searches, see Foreign Keys.
`isTreeView`	Only present for table panes and then indicates whether the pane is a tree table pane with hierarchically organized records.
`moveMode`	Only present for table panes for which the move action is defined and then carries one of two values: • `insideContext`: Indicates that it is only allowed to move a record of the table pane to a position inside its current context, that is, to a position leaving its parental relation unchanged. • `full`: Indicates that it is allowed to move a record of the table pane both to a position inside and outside its current context. The move action is identified by `action:move` and as for any other action defined, a specification of the move action can be found in the `actions` property.
`actions`	Specification of the actions defined for the pane. See Actions.
`fields`	Specification of the fields comprising the data records of the pane. See Fields.
`foreignKeys`	Specification of the foreign keys defined for the pane. See Foreign Keys.

For example, for the `ExpenseSheets` container:

```
$ curl -i
       -H 'Accept-Language: en-US'
       -H 'Accept: application/vnd.deltek.maconomy.containers+json;  ↩
    charset=utf-8; version=6.0'
       'http://SERVER/maconomy-api/containers/macoprod/expensesheets/ ↩
    specification'

HTTP/1.1 200 OK
Content-Type: application/vnd.deltek.maconomy.containers+json; charset=utf ↩
    -8; version=6.0

{
  "meta": {
    "containerName": "expensesheets"
```

```
  },
  "panes": {
    "filter": {
      "paneName": "filter",
      "title": "List of Expense Sheets",
      "entity": "expensesheetheader",
      "actions": { ... },
      "fields": { ... },
      "foreignKeys": { ...}
    },
    "card": {
      "paneName": "card",
      "title": "Expense Sheets",
      "entity": "expensesheetheader",
      "actions": { ... },
      "fields": { ... },
      "foreignKeys": { ...}
    },
    "table": {
      "paneName": "table",
      "title": "Expense Sheet Lines",
      "entity": "expensesheetline",
      "isTreeView": false,
      "actions": { ... },
      "fields": { ... },
      "foreignKeys": { ...}
    },
  },
  "relatedContainers": { ... },
  "links" : {
    "self": {
      "href": "http://SERVER/containers/macoprod/expensesheets/ ↩
    specification",
      "rel": "self"
    }
  }
}
```

This JSON tells us that the `ExpenseSheets` container has all three types of panes: a filter, a card, and a table pane. Obviously, only the high-level structure of the specification response has been shown here. The full response is substantially larger and some of the omitted parts are revealed during the elaboration taking place in the following sections.

4.1.1 Actions

Specifying a card or a table pane (does not apply to filter panes) includes specifying the gross list of actions available for that pane. Whether a specific action among these actions

can actually be invoked during interaction with a particular data entry is determined by examining if a hyperlink with a matching link relation is available from the current state of that resource. More specifically, enabled pane actions can be discovered in the `links` property held in the JSON objects representing pane data within a full data response, see Data Resource. Such action discovery is a good example of the principles of HATEOAS put into practice.

Specifications of actions supported by a given container data pane can be found in the `actions` property of the JSON object representing that pane's specification. For any but the actions `action:init-row`, `action:init-create-row`, and `action:move`, these are the properties of an action specification JSON object:

Property	Explanatory text
`title`	Human-readable text describing the action. This text is appropriate for displaying in a user interface.
`rel`	Uniquely identifies the action within the container and specifies the link relation associated with any hyperlink representing the action.

The properties of a JSON object representing the specification of one of the actions `action:init-row`, `action:init-create-row`, or `action:move` that may be defined for a line-number controlled table pane are slightly different:

For the actions `action:init-row` and `action:init-create-row`, these are the properties of a JSON object representing their specification:

Property	Explanatory text
`insertTitle`	Human-readable text describing the action of adding a record at an existing position in the line-number controlled table pane.
`appendTitle`	Human-readable text describing the action of adding a record to the end of the line-number controlled table pane.
`rel`	Either the value `action:init-row` or `action:init-create-row`, specifying the link relation associated with any hyperlink representing the action.

For the action `action:move`, these are the properties of a JSON object representing its specification:

Property	Explanatory text
`upTitle`	Human-readable text describing the action of moving a record upwards in the line-number controlled table pane.
`downTitle`	Human-readable text describing the action of moving a record downwards in the line-number controlled table pane.
`indentTitle`	Human-readable text describing the action of indenting a record in the line-number controlled table pane. This property is only present when a `full` mode move action is defined, see the `moveMode` property described above.
`outdentTitle`	Human-readable text describing the action of outdenting a record in the line-number controlled table pane. This property is only present when a `full` mode move action is defined, see the `moveMode` property described above.
`rel`	The value `action:move`, specifying the link relation associated with any hyperlink representing the action.

Defined actions which can be discovered through a container's specification resource each falls into one of the following two groups of actions:

Standard actions are a collection of predefined actions that seem natural for data organized in panes of records:

Link relation	Explanatory text
`action:init`	Initializing a record in a data pane *without* line-number control. See Creating a Data Entry and Adding a Table Record.
`action:init-create`	Initializing and creating a record in a data pane *without* line-number control. See Creating a Data Entry and Adding a Table Record.
`action:init-row`	Initializing a record in a table pane *with* line-number control. See Adding a Table Record.
`action:init-create-row`	Initializing and creating a record in a table pane *with* line-number control. See Adding a Table Record.
`action:create`	Creating an already initialized data pane record. See Creating a Data Entry and Adding a Table Record.
`action:update`	Updating a data pane record. See Updating a Record.
`action:delete`	Deleting a data pane record. See Deleting a Record.

Link relation	Explanatory text
`action:move`	Moving a record in a table pane *with* line-number control to another position. See Moving a Table Record.
`action:print`	Printing a data pane record. See Printing.

Application actions relate to the specific business logic implemented by the container to which the data pane belongs. The value of their `rel` property can be any `action:`*xxx* where the *xxx* does not cause any overlap with the standard actions. See Applying an Application Action for more details on the usage of application actions.

Here is a fuller version of the `table` pane's `actions` property which was omitted from the `ExpenseSheets` specification JSON above:

```
"actions": {
  "action:init-row": {
    "insertTitle": "Insert Expense Sheet Line",
    "appendTitle": "Add Expense Sheet Line",
    "rel": "action:init-row"
  },
  "action:init-create-row": {
    "insertTitle": "Insert Expense Sheet Line",
    "appendTitle": "Add Expense Sheet Line",
    "rel": "action:init-create-row"
  },
  "action:create": {
    "title": "Save Expense Sheet Line",
    "rel": "action:create"
  },
  "action:update": {
    "title": "Save Expense Sheet Line",
    "rel": "action:update"
  },
  "action:delete": {
    "title": "Delete Expense Sheet Line",
    "rel": "action:delete"
  },
  "action:move": {
    "title": "Move Expense Sheet Line",
    "rel": "action:move"
  },
  "action:print": {
    "title": "Print Expense Sheet Line",
    "rel": "action:print"
  },
  "action:createjobfavorite": {
```

```
    "title": "Create favorite",
    "rel": "action:createjobfavorite"
  },
  ...
}
```

Due to the `table` pane of the `ExpenseSheets` container being subject to line-number control, the `action:init-row` and `action:init-create-row` actions are here present instead of the `action:init` and `action:init-create` actions. Also notice how the JSON objects for these two actions hold the two properties `insertTitle` and `appendTitle` instead of just a `title` property. The specified `action:createjobfavorite` action is an example of one of the application actions specifically related to expense sheets.

4.1.2 Fields

Another important part of specifying a container pane is specifying the fields comprising the records of that pane. The specification of the record's fields is held in the `fields` property of the JSON object representing the pane's specification and these are the possible properties of the JSON object holding a field specification:

Property	Explanatory text
`name`	Identifier used to reference the field in representations. This is intended for use by software and is normally not displayed in a user interface.
`title`	Human-readable name of the field. The title is an appropriate label for the field in a user interface.
`key`	Indicates whether the field is a key field. The key fields of a record uniquely identifies it.
`type`	Data type of the field. This is important information as it tells the client program how to interpret and represent values of the field when interacting with records. The data type is one of these: `integer`, `real`, `amount`, `boolean`, `string`, `date`, `time`, `enum`, `timeduration`, or `autotimestamp`. The specifics of each format are detailed in the Data Types section.
`enumType`	Only present for fields having the `enum` data type and then contains the name of the enum type. The value may, for example, be used when client programs need to construct expressions used in filter restrictions.

Property	Explanatory text
`subtypeContainer`	Only present for fields having the `enum` data type and then contains the name of the container supplying the possible values for the enum type. To find hyperlinks relevant for the subtype container, client programs should go through the `relatedContainers` property, see Related Containers.
`maxLength`	Only present for string fields and then specifies the maximum length of the field value.
`multiLine`	Indicates whether a string field can contain newline characters (`\u000a`), thus spanning multiple lines.
`create`	Indicates whether the field is editable at the time a record is created. If `false`, a client program is not permitted to change the value of the field in the template record obtained from the initialization operation.
`update`	Indicates whether the field can be modified by the client program after the record has been created.
`autoSubmit`	Indicates to the client program whether it should automatically update the resource when a user finishes editing the field.
`mandatory`	Indicates whether the field is mandatory and must be filled out. If mandatory, string, date, and time fields cannot be blank and numeric fields must be non-zero.
`secret`	Indicates whether the contents of the field must not be displayed unmasked in a user interface. This could be the case for a password field, for example.
`unfilterable`	Indicates whether the field cannot act as part of a filter restriction.
`suggestions`	Indicates how the client should present inline searches from the field in a user interface. These are the possible values: • `onDemand`: Inline search on demand. • `automatic`: A search-as-you-type style inline search. • `none`: No inline search. • `standard`: The client program should apply its own preferred default and use the behavior of either `onDemand`, `automatic`, or `none`.
`references`	Lists which foreign keys the field participates in. See Foreign Keys.

Here, for example, are some of the JSON objects specifying record fields of the `table` pane in the `ExpenseSheets` container:

```
"fields": {
  ...,
  "activitynumber": {
    "title": "Activity No.",
    "name": "activitynumber",
    "type": "string",
    "key": false,
    "create": true,
    "autoSubmit": false,
    "mandatory": false,
    "maxLength": 255,
    "multiLine": false,
    "secret": false,
    "suggestions": "onDemand",
    "update": true,
    "unfilterable": false,
    "references": [
      "activitynumber_expensemileageactivity",
      "activitynumber_activity"
    ]
  },
  "text": {
    "title": "Description",
    "name": "text",
    "type": "string",
    "key": false,
    "create": true,
    "autoSubmit": false,
    "mandatory": false,
    "maxLength": 255,
    "multiLine": false,
    "secret": false,
    "suggestions": "none",
    "update": true,
    "unfilterable": false,
    "references": []
  },
  "currency": {
    "title": "Currency",
    "name": "currency",
    "type": "enum",
    "key": false,
    "subtypeContainer": "popup_currencytype",
    "create": true,
    "autoSubmit": false,
    "mandatory": true,
```

```
    "multiLine": false,
    "secret": false,
    "suggestions": "none",
    "update": true,
    "unfilterable": false,
    "enumType": "CurrencyType",
    "references": []
  },
  ...
}
```

The fields `activitynumber` and `text` are both string fields, whereas the field `currency` is a field of enum type having the container `popup_currencytype` as the source of its values.

4.1.3 Foreign Keys

Foreign keys describe associations between data in the system and can be used to navigate to related resources and/or to provide suggestions for values for one or more record fields through so-called *foreign key searching*.

As opposed to a simple filter search (see Filtering) where the client program is interacting with just a single filter pane, two container panes are in play when doing a foreign key search:

1. The host pane from where the search is launched.
2. The search pane providing the search result.

The role of the host pane is to provide the server with a restriction based on the uncommitted value of the record from where the search was launched. For example, when searching for a task on an expense sheet line, then only tasks related to the job currently selected on that expense sheet line should be included in the search result.

For each pane of a given container, the `foreignKeys` property of that pane's specification JSON holds the specifications of the foreign keys defined. These are the possible properties of a foreign key JSON object:

Property	Explanatory text
`name`	Name of the foreign key.
`title`	Title of the foreign key. A text appropriate for displaying in a user interface.

Property	Explanatory text
`incomplete`	Indicates whether the foreign key is incomplete. Incomplete foreign keys can be used for searching only and cannot be navigated like complete foreign keys can. Find further explanation below.
`searchContainer`	Only present for searchable foreign keys and then holds the name of the container providing the search results. Note that a `self` foreign key will never have this property. Find further explanation of `self` foreign keys below.
`searchPane`	Only present for searchable foreign keys and then holds the name of the pane within the search container providing the search results. The search pane will always be the filter pane and its name is included here for convenience only. Note that a `self` foreign key will never have this property. Find further explanation of `self` foreign keys below.
`fieldReferences`	List of JSON objects specifying the field references comprising the foreign key relationship. Find the properties of a field reference JSON object below.
`switchField`	Only present for conditional foreign keys and then holds the name of the enum type field whose value determines whether the foreign key is enabled. Find further explanation of conditional foreign keys below.
`switchValue`	Only present for conditional foreign keys and then holds the enum literal value that must be assigned to the switch field in order for the foreign key to be enabled. Find further explanation of conditional foreign keys below.
`links`	Hyperlinks available for the foreign key. Find the list of link relations below.

A field reference JSON object listed in the above mentioned `fieldReferences` property contains the following properties:

Property	Explanatory text
`field`	Name of the field in the container pane that maps to a field in a foreign container pane.
`foreignField`	Name of the field in a foreign container pane being referred.

Property	Explanatory text
`supplement`	Indicates if the field reference is only a supplement and thus not really participating in the foreign key relationship. Find further description of a supplement field reference below.

These are the purposes of the hyperlinks possibly available from a foreign key JSON object:

Link relation	Explanatory text
`data:search`	Reference to the action of performing a foreign key search. Find further description below.
`data:key`	Reference to the action of navigating a complete foreign key. Find further description below.

For example, the following JSON object specifying the `activitynumber_activity` foreign key for the `table` pane was left out from the `ExpenseSheets` specification JSON presented earlier:

```
{
  "name": "activitynumber_activity",
  "title": "Activity",
  "incomplete": false,
  "searchContainer": "find_activity",
  "searchPane": "filter",
  "fieldReferences": [
    {
      "field": "activitynumber",
      "foreignField": "activitynumber",
      "supplement": false
    },
    {
      "field": "activitytextvar",
      "foreignField": "activitytext",
      "supplement": true
    }
  ],
  "links": {
    "data:search": {
      "href": "http://SERVER/maconomy-api/containers/macoprod/ ↩
  expensesheets/search/table;foreignkey=activitynumber_activity",
      "rel": "data:search"
    },
```

```
      "data:key": {
        "template": "http://SERVER/maconomy-api/containers/macoprod/{ ↩
    container}/instances/{instance}/data;activitynumber={0}",
        "fields": [
          "activitynumber"
        ],
        "rel": "data:key"
      }
    }
  }
```

Here, the first of the two field references specified in the `fieldReferences` property shows that the `activitynumber` field on expense sheet lines references the `activitynumber` field found on activities. For the second field reference, you can see that this, with the boolean value `true` in its `supplement` property, is marked as a *supplement*. A supplement field reference is not directly participating in the foreign key relationship, but is included only as a signal to the client program to assign the value back during a foreign key search. In this particular case, whenever the user has chosen an activity among some `activitynumber_activity` foreign key search result, the client program must assign the value of this activity's `activitytext` field back to the `activitytextvar` field on the expense sheet line.

That the `searchContainer` property for the `activitynumber_activity` foreign key holds the container name `find_activity` indicates that the foreign key can be used for searching and that these searches are performed by use of the filter pane within the `find_activity` container. Being a search container, the `find_activity` container is among the containers mentioned in the `table` pane's `relatedContainers` property, and there a hyperlink leading to its specification can be found (see Related Containers). Besides being useful in discovering the entity to which the foreign fields of the field references refer, the search container's specification is also important for the client program's ability to interpret foreign key search results.

In order to actually perform a foreign key search from a record within some pane, the client program must follow the `data:search` hyperlink available from the `links` property of the foreign key's JSON object:

```
{
  "href": "http://SERVER/BASEPATH/containers/SHORTNAME/CONTAINER/search/ ↩
    PANE;foreignkey=FOREIGN_KEY",
  "rel": "data:search"
}
```

The client program must authenticate (see Authentication) and apply the HTTP verb `POST` with the uncommitted value of the record from where the search was initiated in the request body. Like with a filter request, the client program can have any Filter Paging, Filter Sorting, Filter Fields Slicing, and Filter Restriction applied to the search.

The response to a successful `data:search` request is called a *search response* and except from a missing `self` hyperlink its structure is similar to that of a search container filter response, see Filtering.

For the `activitynumber_activity` foreign key, for example:

```
$ curl -i
       -u 'Administrator:123456'
       -H 'Accept-Language: en-US'
       -H 'Accept: application/vnd.deltek.maconomy.containers+json;  ↩
    charset=utf-8; version=6.0'
       -H 'Content-Type: application/vnd.deltek.maconomy.containers+json;  ↩
    charset=utf-8; version=6.0'
       -d '{
         "data": {
           "expensesheetnumber": "10760001",
           "description": "",
           "employeenumber": "11",
           ...
         }
       }'
       'http://SERVER/maconomy-api/containers/macoprod/expensesheets/ ↩
    search/table;foreignkey=activitynumber_activity'

HTTP/1.1 200 OK
Content-Type: application/vnd.deltek.maconomy.containers+json; charset=utf ↩
    -8; version=6.0

{
  "meta": {
    "containerName": "find_activity"
  },
  "panes": {
    "filter": {
      "meta": {
        "paneName": "filter",
        "rowCount": 25,
        "rowOffset": 0
      },
      "records": [
        {
          "data": {
            "activitynumber": "101",
            ...,
            "activitytext": "Consulting",
            ...
          },
          "links": {
            "data:same-key-some-container": {
```

```
            "template": "http://SERVER/maconomy-api/containers/macoprod ↩
    /{container}/instances/{instance}/data;activitynumber=101",
            "rel": "data:same-key-some-container"
          }
        }
      },
      {
        "data": {
          "activitynumber": "102",
          ...,
          "activitytext": "Preparation",
          ...
        },
        "links": {
          "data:same-key-some-container": {
            "template": "http://SERVER/maconomy-api/containers/macoprod ↩
    /{container}/instances/{instance}/data;activitynumber=102",
            "rel": "data:same-key-some-container"
          }
        }
      },
      ...
    ]
  }
 }
}
```

As described in Filtering, the `links` property of each record represented in this search response mentions a `data:same-key-some-container` hyperlink. These hyperlinks are referencing the action of loading the given container data entry and their usage is further described in Loading a Data Entry.

A foreign key is said to be *complete* whenever the composition of the field values participating in its references (supplement fields excluded) uniquely identifies a foreign resource. With the number being a unique identifier of an activity, the `activitynumber_activity` foreign key is an example of such complete foreign key, and this is indicated by the boolean value `false` in its `incomplete` property.

Not surprisingly, a complete foreign key can be used for navigating from one resource to another, and this is done by following the `data:key` hyperlink available from its specification:

```
{
  "template": "http://SERVER/BASEPATH/containers/SHORTNAME/{container}/ ↩
    instances/{instance}/data;KEY_FIELD_NAME_1={0};KEY_FIELD_NAME_2 ↩
    ={1};...",
  "fields": [
    "KEY_FIELD_NAME_1",
```

```
    "KEY_FIELD_NAME_2",
    ...
  ],
  "rel": "data:key"
}
```

The client program must replace the `{container}` placeholder in the template URL by the name of the container it wishes to navigate to (this container must of course be based on the same entity as referred to by the foreign fields of the field references). Also, the `{instance}` placeholder must be replaced by the id of an instance of the container chosen (see Container Instances). Finally, the field values making up the foreign key reference (excluding supplement fields) must be substituted into the template URL. The indices of the field names in the array held in the `fields` property of the hyperlink object indicate which of the numbered placeholders each field value should replace.

As with any of the hyperlinks described in Loading a Data Entry, the client program must authenticate using a reconnect token (see Maconomy Reconnect Authentication) valid for the container instance and apply the HTTP verb `POST` with a valid `Maconomy-Concurrency-Control` header enclosed (see Concurrency Tags).

The response to a successful `data:key` request is a full data response (see Data Resource).

For example, for the `activitynumber_activity` foreign key, the `data:key` hyperlink for navigating to the activity referenced by an expense sheet line looks like this:

```
{
  "template": "http://SERVER/maconomy-api/containers/macoprod/{container}/ ↩
    instances/{instance}/data;activitynumber={0}",
  "fields": [
    "activitynumber"
  ],
  "rel": "data:key"
}
```

Choosing `Activities` as the target container, the substituted `data:key` template URL could look something like this:

```
http://SERVER/maconomy-api/containers/macoprod/activities/instances/ ↩
    c953cdd8-2806-4e61-b3cd-3022ab8aea50/data;activitynumber=102
```

As opposed to the foreign key `activitynumber_activity` discussed above, the foreign key `taskname_expensemileagetasklistline` is an example of an *incomplete* foreign key defined for the `table` pane of the `ExpenseSheets` container:

```
"taskname_expensemileagetasklistline": {
  "name": "taskname_expensemileagetasklistline",
  "title": "Task",
```

```
  "incomplete": true,
  "searchContainer": "find_tasklistline",
  "searchPane": "filter",
  "fieldReferences": [
    {
      "field": "taskname",
      "foreignField": "taskname",
      "supplement": false
    },
    {
      "field": "tasknamevar",
      "foreignField": "description",
      "supplement": true
    }
  ],
  "links": {
    "data:search": {
      "href": "http://SERVER/maconomy-api/containers/macoprod/ ↩
    expensesheets/search/table;foreignkey= ↩
    taskname_expensemileagetasklistline",
      "rel": "data:search"
    }
  }
}
```

The incompleteness of the `taskname_expensemileagetasklistline` foreign key reflects the fact that an expense sheet line relates to a task which cannot be identified solely from information stored directly on the expense sheet line. Aside from the name of the task, the identification takes the name of the list to which it belongs, and the latter is only indirectly available via the job pointed to by the expense sheet line's `jobnumber` field. Such incomplete foreign keys can only be used for searching and therefore the `links` property does not mention any `data:key` hyperlink.

This is the `self` foreign key also defined for the `table` pane of the `ExpenseSheets` container:

```
"self": {
  "name": "self",
  "title": "Expense Sheet",
  "incomplete": false,
  "fieldReferences": [
    {
      "field": "instancekey",
      "foreignField": "instancekey",
      "supplement": false
    }
  ],
  "links": {
```

```
    "data:key": {
      "template": "http://SERVER/maconomy-api/containers/macoprod/{ ↩
    container}/instances/{instance}/data;instancekey={0}",
      "fields": [
        "instancekey"
      ],
      "rel": "data:key"
    }
  }
}
```

As it occurs from this JSON, the `self` foreign key cannot be used for searching. A `self` foreign key expresses the fact that the key of any entity can be seen as a foreign key to that entity itself, and its purpose is to enable the client program to relate a given record of the hosting container pane with the exact same record in the card pane of some other container. Obviously, the foreign card pane must be based on the same entity as the hosting container pane. A `self` foreign key can be particularly useful when working with multiple containers whose card panes are based on the same entity.

Conditional Foreign Keys

Any foreign key whose JSON object includes the two properties `switchField` and `switchValue` is called a *conditional foreign key*. This because the validity of these foreign keys is conditional on whether the value of the record field mentioned in the `switchField` property equals the literal value mentioned in the `switchValue` property.

As the `ExpenseSheets` container is not able to deliver any conditional foreign keys, we instead turn our eyes towards the `GeneralJournal` container to find examples. Now, in the `GeneralJournal` container, one of the fields defined for the `table` pane is the `accountnumber` field:

```
{
  "name": "accountnumber",
  "title": "Account No.",
  "key": false,
  "type": "string",
  "maxLength": 255,
  "multiLine": false,
  "create": true,
  "update": true,
  "autoSubmit": false,
  "mandatory": false,
  "secret": false,
  "unfilterable": false,
  "suggestions": "automatic",
  "references": [
    "accountnumber_account",
    "accountnumber_customer",
```

```
    "companycustomer",
    "accountnumber_vendor",
    "companyvendor"
  ]
}
```

Here all five of the referenced foreign keys are conditional foreign keys whose validity depends on the value of the record's `typeofentry` field. In this example, take a look at the specification for the `accountnumber_account` foreign key:

```
{
  "name": "accountnumber_account",
  "title": "Account",
  "incomplete": false,
  "searchContainer": "find_account",
  "searchPane": "filter",
  "fieldReferences": [
    {
      "field": "accountnumber",
      "foreignField": "accountnumber",
      "supplement": false
    }
  ],
  "switchField": "typeofentry",
  "switchValue": "g",
  "links": {
    "data:search": {
      "href": "http://SERVER/maconomy-api/containers/macoprod/ ↩
    generaljournal/search/table;foreignkey=accountnumber_account",
      "rel": "data:search"
    },
    "data:key": {
      "template": "http://SERVER/maconomy-api/containers/macoprod/{ ↩
    container}/instances/{instance}/data;accountnumber={0}",
      "fields": [
        "accountnumber"
      ],
      "rel": "data:key"
    }
}
```

From this it appears that the `accountnumber_account` foreign key is only applicable in cases where a record holds the literal value `g` in its `typeofentry` field. If a different value is filled into that field, the `accountnumber_account` foreign key is no longer valid. Similarly, the foreign keys `accountnumber_customer` and `companycustomer` have the switch value `r`, and the foreign keys `accountnumber_vendor` and `companyvendor` the switch value `p`.

When searching from the `accountnumber` field, the client program must in turn consider the conditional foreign keys involving the `accountnumber` field and apply the first (if any) foreign key for which the switch value matches the (uncommitted) value of the record's `typeofentry` field.

4.1.4 Related Containers

The purpose of the `relatedContainers` property in a data container's specification is to provide the client program with some handy references for each container considered related to the container. Container *c* is considered related if it matches at least one of two descriptions:

1. *c* is a popup container supplying values for an enum type and is mentioned in the `subtypeContainer` property of some field specification within the container specification.
2. Some foreign key searches are performed through *c* as it is mentioned in the `searchContainer` property of some foreign key specification within the container specification.

A related container JSON object held in a `relatedContainers` property has the following two properties:

Property	Explanatory text
`meta`	`containerName`: Name of the related container.
`links`	Hyperlinks available for the related container. Find the list of link relations below.

These are the purposes of the hyperlinks available from a related container JSON object:

Link relation	Explanatory text
`specification`	Reference to the specification resource of the related container, see Specification.
`data:enumvalues`	Only available for subtype containers in which case it is a reference to be followed in order to gain knowledge on the possible enum values. Find further details below.

For the `ExpenseSheets` container, for example, this is some of the contents of the `table` pane's `relatedContainers` property earlier omitted:

```
"relatedContainers": {
  ...,
  "popup_currencytype": {
    "meta": {
      "containerName": "popup_currencytype"
    },
    "links": {
      "specification": {
        "href": "http://SERVER/maconomy-api/containers/macoprod/ ↩
    popup_currencytype/specification",
        "rel": "specification"
      },
      "data:enumvalues": {
        "href": "http://SERVER/maconomy-api/containers/macoprod/ ↩
    popup_currencytype/filter",
        "rel": "data:enumvalues"
      }
    }
  },
  ...,
  "find_activity": {
    "meta": {
      "containerName": "find_activity"
    },
    "links": {
      "specification": {
        "href": "http://SERVER/maconomy-api/containers/macoprod/ ↩
    find_activity/specification",
        "rel": "specification"
      }
    }
  },
  ...
}
```

As expected, both a `specification` and a `data:enumvalues` hyperlink are available for `popup_currencytype` since this refers to the subtype container of the `table` pane's `currency` field. For `find_activity` which the foreign key `activitynumber_activity` specifies as the name of its search container, only a `specification` hyperlink is available.

To get the list of possible values for some field of a given enum type, the `data:enumvalues` hyperlink available from the subtype container's entry within the `relatedContainers` property can be followed:

```
{
  "href": "http://SERVER/BASEPATH/containers/SHORTNAME/POPUP_CONTAINER/ ↩
    filter",
```

```
  "rel": "data:enumvalues"
}
```

The `data:enumvalues` link relation can be seen as an alias for the popup container's `data:filter` link relation and its usage is thus similar to what is described for `data:filter` in Filtering.

4.2 Access

No `data:any-key` hyperlink will ever be available from a container instance resource (see Container Instances), if the authenticated user is not granted read access to the given container. Similarly, no `action:init` hyperlink will ever be available, if the user is not granted create access.

For a given container, a client program is able to retrieve information about an authenticated user's CRUD access rights by following the hyperlink with link relation `access` available from the container's root resource:

```
{
  "href": "http://SERVER/BASEPATH/containers/SHORTNAME/CONTAINER/access",
  "rel": "access"
}
```

The client program must authenticate (see Authentication) and apply the HTTP verb `GET`.

The JSON object held in the body of a response to an `access` request has the following properties:

Property	Explanatory text
`meta`	`containerName`: The name of the container.
`access`	`read`: Indicates whether the authenticated user is allowed to read data from the container. `create`: Indicates whether the authenticated user is allowed to create new container entries. `update`: Indicates whether the authenticated user is allowed to update container entries and execute application actions. `delete`: Indicates whether the authenticated user is allowed to delete entries from the container.
`links`	`self`: Hyperlink referring to the access resource itself.

Note that if the authenticated user is not granted read access to a container, it is implied

that the user is not granted create, update, or delete access either – that is, if the `read` property carries the value `false`, then so are the remaining access properties.

For the `ExpenseSheets` container, for example:

```
$ curl -i
       -u 'Administrator:123456'
       -H 'Accept-Language: en-US'
       -H 'Accept: application/vnd.deltek.maconomy.containers+json;  ↩
    charset=utf-8; version=6.0'
       'http://SERVER/maconomy-api/containers/macoprod/expensesheets/ ↩
    access'

HTTP/1.1 200 OK
Content-Type: application/vnd.deltek.maconomy.containers+json; charset=utf ↩
    -8; version=6.0

{
  "meta": {
    "containerName": "expensesheets"
  },
  "access": {
    "read": true,
    "create": true,
    "update": true,
    "delete": true
  },
  "links": {
    "self": {
      "href": "http://SHORTNAME/containers/macoprod/expensesheets/access",
      "rel": "self"
    }
  }
}
```

This response tells us that the Administrator user is granted full CRUD access to expense sheets.

4.3 Filtering

For a **data container**, the purpose of defining a filter pane is to allow for client programs to search for data entries within the container itself.

For a **popup container**, each record of its filter pane comprises an enum value. All relevant information is available through these entries and none of them form a basis for further interaction.

For a **search container**, its filter pane allows for client programs to search for data

entries within other data containers whose card pane is based on the same entity. Interactions with any of these data entries have to happen against some instance of a data container.

Now, a client program interacts with the filter pane of a container by following the hyperlink with link relation `data:filter` available from the container's resource:

```
{
  "href": "http://SERVER/BASEPATH/containers/SHORTNAME/CONTAINER/filter",
  "rel": "data:filter"
}
```

In order to access a filter resource, the client program must authenticate (see Authentication) and apply the HTTP verb `POST`. Relevant filter parameters are communicated to the server by inclusion of appropriate JSON properties in the request body, see Filter Paging, Filter Sorting, Filter Fields Slicing, and Filter Restriction.

The response to a `data:filter` request is called a *filter response*, and the JSON object held in its body has the following properties:

Property	Explanatory text
`meta`	`containerName`: The name of the container.
`panes`	`filter`: JSON object representing filter pane contents. Find its properties below.
`links`	`self`: Hyperlink referring to the filter resource itself.

These are the properties of the JSON object within a filter response representing filter pane contents:

Property	Explanatory text
`meta`	`paneName`: The name of the filter pane. `rowCount`: The number of filter pane records returned (see Filter Paging). `rowOffset`: The offset of the returned filter pane records (see Filter Paging).
`records`	List of filter pane record JSON objects each exposing a container data entry. Find the possible properties of these objects below.

The properties of the filter pane record JSON objects listed in the `records` property, each exposing a container data entry, are these:

Property	Explanatory text
`data`	A selection of field values appropriate for displaying in the user interface from where the filter request was initiated. The fields mentioned will be the key fields plus any extra fields explicitly specified in a `fields` parameter on the request (see Filter Fields Slicing).
`links`	Only present for search and data containers, at which point it presents a hyperlink referencing the action of loading the container data entry. For search containers, this is a `data:same-key-some-container` hyperlink, and for data containers, a `data:same-key-some-instance` hyperlink. See Loading a Data Entry for further details on how to use these two kinds of hyperlinks.

For the `ExpenseSheets` container, for example:

```
$ curl -i
       -u 'Administrator:123456'
       -H 'Accept-Language: en-US'
       -H 'Accept: application/vnd.deltek.maconomy.containers+json;  ↩
    charset=utf-8; version=6.0'
       'http://SERVER/maconomy-api/containers/macoprod/expensesheets/ ↩
    filter'

HTTP/1.1 200 OK
Content-Type: application/vnd.deltek.maconomy.containers+json; charset=utf ↩
    -8; version=6.0

{
  "meta": {
    "containerName": "expensesheets"
  },
  "panes": {
    "filter": {
      "meta": {
        "paneName": "filter",
        "rowCount": 25,
        "rowOffset": 0
      },
      "records": [
        {
          "data": {
            "expensesheetnumber": "10760001"
          },
          "links": {
            "data:same-key-some-instance": {
```

```
              "template": "http://SERVER/maconomy-api/containers/macoprod/ ↩
    expensesheets/instances/{instance}/data;expensesheetnumber=10760001",
              "rel": "data:same-key-some-instance"
            }
          }
        },
        ...
      ]
    }
  },
  "links": {
    "self": {
      "href": "http://SHORTNAME/containers/macoprod/expensesheets/filter",
      "rel": "self"
    }
  }
}
```

Here we see information about expense sheet number `10760001` accompanied by the `data:same-key-some-instance` hyperlink for the client program to follow in order to interact with that particular expense sheet (see Loading a Data Entry).

As another example, for the `popup_currencytype` popup container:

```
$ curl -i
       -u 'Administrator:123456'
       -H 'Accept-Language: en-US'
       -H 'Accept: application/vnd.deltek.maconomy.containers+json;  ↩
    charset=utf-8; version=6.0'
       'http://SERVER/maconomy-api/containers/macoprod/popup_currencytype/ ↩
    filter'

HTTP/1.1 200 OK
Content-Type: application/vnd.deltek.maconomy.containers+json; charset=utf ↩
    -8; version=6.0

{
  "meta": {
    "containerName": "popup_currencytype"
  },
  "panes": {
    "filter": {
      "meta": {
        "paneName": "filter",
        "rowCount": 28,
        "rowOffset": 0
      },
      "records": [
        {
```

```
          "data": {
            "value": "nil"
          }
        },
        {
          "data": {
            "value": "dkk"
          }
        },
        {
          "data": {
            "value": "eur"
          }
        },
        ...
      ]
    }
  },
  "links": {
    "self": {
      "href": "http://SERVER/maconomy-api/containers/macoprod/ ↩
    popup_currencytype/filter",
      "rel": "self"
    }
  }
}
```

A `nil` value like the one listed here is defined for all enum types in order to allow for the cases where no value has been chosen for an enum field. In user interfaces, it makes sense to display possible enum values as a dropdown.

Besides a `value` property carrying a representation of the corresponding enum value (see Enum), the `data` JSON object of a popup filter pane record may include zero or more among three other properties depending on the `fields` parameter passed along with the filter request (see Filter Fields Slicing):

Property	Explanatory text
`value`	The value of the enum value represented by the filter pane record (see Enum).
`ordinal`	The integer ordinal of the enum value represented by the filter pane record. This uniquely identifies the enum value among the others of the same type.
`title`	A human-readable name of the enum value represented by the filter pane record.

Property	Explanatory text
`hidden`	Indicates whether the enum value represented by the filter pane record should be available for selection in a user interface or not.

A specification of the available popup record fields can of course be obtained by following the `specification` hyperlink for the given popup container (see Specification).

In the four sub-sections below, the following supported filter pane searching features are described:

- Paging
- Sorting
- Fields slicing
- Restriction

4.3.1 Filter Paging

You may have noticed that your client program receives exactly 25 records in every filter response even though the system being polled contains more than 25 data entries. This is because the filter resource splits the results into pages. Filter paging can be controlled by two *paging parameters* supplied as JSON properties in the request body:

Parameter	Explanatory text
`limit`	If $n > 0$, the amount of records included in the response will not exceed n. If `0`, *all* records will be included in the response.
`offset`	If $n \geq 0$, the first n records will be skipped.

For data and search containers, the default `limit` applied is `25`, and for popup containers it is `0`. The default `offset` applied is `0` for all containers.

For example, providing the following paging parameters with a filter request, the response will contain up to 11 records, starting from record number 8 when counting from 0:

```
{
  "limit": 11,
  "offset": 8
}
```

As another example, the following paging parameters will make a filter response contain any amount of records, starting from record number 20 when counting from 0:

```
{
  "limit": 0,
  "offset": 20
}
```

4.3.2 Filter Sorting

The filter resource allows the client program to control the order in which record objects are presented in a filter response. A preferred sorting can be specified by the client program through an `orderBy` parameter supplied as a JSON property in the request body.

A preferred sorting must be passed as a list of JSON objects with the following properties:

Property	Explanatory text
`field`	Name of field that should participate in a multi-column sorting of the records.
`ascending`	If `true`, the sorting order applied for the field of the given name will be ascending, otherwise, if `false`, it will be descending. If left out, the ascending order will be applied.

For example, the client program will receive an `ExpenseSheets` filter response with the represented records sorted first descending by the value in their `DateSubmitted` field and second ascending by the value in their `EmployeeName` field when passing the following in the request body:

```
{
  "orderBy": [
    {
      "field": "DateSubmitted",
      "ascending": false
    },
    {
      "field": "EmployeeName"
    }
  ]
}
```

4.3.3 Filter Fields Slicing

The filter resource also allows the client program to control which fields are included in the record representations within the filter response. A key field is always part of the

response, but otherwise a field is included only if it is mentioned among the list of field names passed via a `fields` JSON property in the request body. Performance benefits can be achieved by limiting the amount of fields included to the ones that are actually relevant to the client program.

For example, the record JSON objects received in an `ExpenseSheets` filter response will at least contain the properties `Description` and `EmployeeName`, if the client program passes the following `fields` parameter:

```
{
  "fields": [
    "EmployeeName",
    "Description"
  ]
}
```

As `ExpenseSheetNumber` is a key field, the record JSON objects will also hold an `expensesheetnumber` property.

4.3.4 Filter Restriction

Finally, the filter resource allows the client program to supply an expression restricting the filter record JSON objects returned in a way similar to how a `WHERE` clause works in SQL. The syntax used for these expressions is the Expression Language also used in MDML and other XML specification languages in Maconomy (see the MDML Language Reference [5] for a full description of the Expression Language). A restricting expression is provided by the client program through a `restriction` parameter supplied as a JSON property in the request body.

For example, by passing the following `restriction` parameter along with an `ExpenseSheets` filter request, the filter records represented in the response will all be pointing to sheets that were created some time after July 1, 2014:

```
{
  "restriction": "CreateDate > date(2014,7,1)"
}
```

As another example, the following `restriction` parameter should be supplied, if only filter records pointing to expense sheets that have been submitted for approval and are related to employees whose name begins with “Bob” are of interest:

```
{
  "restriction": "Submitted and EmployeeName like \"Bob*\""
}
```

4.4 Container Instances

To interact with the data entries within some data container, the client program must first create an *instance* of that container. A container instance is a resource holding the state of a container (variable values, whether the different actions are currently enabled or disabled, and so on), and by letting the client program carry out its data interactions against such an instance, the server does not have to spend time on expensive state recalculations on each request (this is in fact the main difference between the first version of the Containers Web Service and later versions).

A new container instance is created by following the hyperlink with link relation `instance:create` available from the container resource:

```
{
  "href": "http://SERVER/BASEPATH/containers/SHORTNAME/CONTAINER/instances ↩
    ",
  "rel": "instance:create"
}
```

Container instance creation requires authentication, and since an instance only lives within the scope of a login session, the client program is required to authenticate using Maconomy Reconnect Authentication in order to obtain a reconnect token that can be used for authentication on subsequent requests towards the created container instance. Also, the client program must apply the HTTP verb `POST` and possibly include an instance configuration JSON object (see Configuring an Instance) in the request body.

The body contained in the response to a container instance creation request representing the created instance is a JSON object with the following properties:

Property	Explanatory text
`meta`	`containerName`: Name of the container. `containerInstanceId`: The identifier of the container instance.
`links`	Hyperlinks available from the container instance resource. Find the list of link relations below.

These are the purposes of the hyperlinks possibly available from a container instance JSON object:

Link relation	Explanatory text
`action:init`	Reference to the action of initializing a card pane record, see Creating a Data Entry.

Link relation	Explanatory text
`action:init-create`	Reference to the action of initializing and creating a card pane record, see Creating a Data Entry.
`data:any-key`	Reference to the action of loading an unspecified (yet deterministic) container data entry, see Loading a Data Entry.
`data:some-key`	Reference to the action of loading a specific container data entry, see Loading a Data Entry.
`instance:data`	Reference to the container data entry currently loaded, see Data Resource.
`instance:data-refresh`	Reference to the action of re-loading the already loaded container data entry, see Data Resource.
`instance:configuration`	Reference to the configuration currently applied to the container instance, see Configuring an Instance.
`instance:configuration-update`	Reference to the action of updating the configuration applied to the container instance, see Configuring an Instance.
`instance:delete`	Reference to the action of deleting the container instance, see Deleting an Instance.
`self`	Reference to the container instance resource itself.

For example, for the `ExpenseSheets` container:

```
$ curl -i
       -u 'Administrator:123456'
       -H 'Maconomy-Authentication: X-Reconnect'
       -H 'Accept-Language: en-US'
       -H 'Accept: application/vnd.deltek.maconomy.containers+json;  ↩
    charset=utf-8; version=6.0'
       -X POST
       'http://SERVER/maconomy-api/containers/macoprod/expensesheets/ ↩
    instances'

HTTP/1.1 200 OK
Maconomy-Reconnect: Zjk1YjUzMT...JMTAJMTYwMjY4MDE4NQ==
Maconomy-Concurrency-Control: d2a39243-a63f-4bd5-8eab-676952009e93
Content-Type: application/vnd.deltek.maconomy.containers+json; charset=utf ↩
    -8; version=6.0
```

```
{
  "meta": {
    "containerName": "expensesheets",
    "containerInstanceId": "7dc0b114-ecf3-4441-940e-cf93dcf58620"
  },
  "links": {
    "action:init": {
      "href": "http://SERVER/maconomy-api/containers/macoprod/ ↩
    expensesheets/instances/7dc0b114-ecf3-4441-940e-cf93dcf58620/data/ ↩
    panes/card/inits",
      "rel": "action:init"
    },
    "action:init-create": {
      "href": "http://SERVER/maconomy-api/containers/macoprod/ ↩
    expensesheets/instances/7dc0b114-ecf3-4441-940e-cf93dcf58620/data/ ↩
    panes/card",
      "rel": "action:init-create"
    },
    "data:any-key": {
      "href": "http://SERVER/maconomy-api/containers/macoprod/ ↩
    expensesheets/instances/7dc0b114-ecf3-4441-940e-cf93dcf58620/data;any ↩
    ",
      "rel": "data:any-key"
    },
    "instance:data": {
      "href": "http://SERVER/maconomy-api/containers/macoprod/ ↩
    expensesheets/instances/7dc0b114-ecf3-4441-940e-cf93dcf58620/data",
      "rel": "instance:data"
    },
    "instance:data-refresh": {
      "href": "http://SERVER/maconomy-api/containers/macoprod/ ↩
    expensesheets/instances/7dc0b114-ecf3-4441-940e-cf93dcf58620/data/ ↩
    refresh",
      "rel": "instance:data-refresh"
    },
    "instance:configuration": {
      "href": "http://SERVER/maconomy-api/containers/macoprod/ ↩
    expensesheets/instances/7dc0b114-ecf3-4441-940e-cf93dcf58620/ ↩
    configuration",
      "rel": "instance:configuration"
    },
    "instance:configuration-update": {
      "href": "http://SERVER/maconomy-api/containers/macoprod/ ↩
    expensesheets/instances/7dc0b114-ecf3-4441-940e-cf93dcf58620/ ↩
    configuration",
      "rel": "instance:configuration-update"
    },
    "instance:delete": {
```

```
      "href": "http://SERVER/maconomy-api/containers/macoprod/ ↩
    expensesheets/instances/7dc0b114-ecf3-4441-940e-cf93dcf58620",
      "rel": "instance:delete"
    },
    "self": {
      "href": "http://SERVER/maconomy-api/containers/macoprod/ ↩
    expensesheets/instances/7dc0b114-ecf3-4441-940e-cf93dcf58620",
      "rel": "self"
    }
  }
}
```


Here the reconnect token `Zjk1YjUzMT...JMTAJMTYwMjY4MDE4NQ==` (dots denoting that only an abbreviation of the token has been included here) held in the response header `Maconomy-Reconnect-Token` is what needs to be used for Maconomy Reconnect Authentication on a later request towards the created container instance, as mentioned above. The concurrency tag `d2a39243-a63f-4bd5-8eab-676952009e93` held in the response header `Maconomy-Concurrency-Control` will be explained next, whereas the sections covering the topics of the hyperlinks in the `links` property can be found in the table above.

4.4.1 Concurrency Tags

The Maconomy system has a concurrency control mechanism that prior to each update to the database compares the values known to the client program with the values stored in the database. If the values differ, the update is rejected and the client program is told that data has been changed by someone else. This is known as the *`DataChanged` check* and is meant to protect against unintended overwrites of updates made by others and to ensure that update decisions are never based on outdated information.

Now, how does this `DataChanged` check suffice in preventing undesirable database updates in the Containers Web Service setup where updates are performed towards container instances and not directly towards the database? Entailing a comparison of the state of the container instance with that of the database, the `DataChanged` check seems sufficient as long as client programs are in sync with their container instances.

It may happen that a client program comes out of sync with a container instance though. This either as a consequence of the container instance accidentally being shared between different interaction flows (for example, in two different browser tabs) or because the client program has missed some relevant update information communicated back in an earlier response.

To ensure that a client program and a server agree on the state of a container instance, *concurrency tags* are being exchanged through `Maconomy-Concurrency-Control` HTTP headers. The response to each request involving a container instance contains a `Maconomy-Concurrency-Control` header carrying the concurrency tag uniquely

identifying the container instance state that resulted from the request. Similarly, each possibly state modifying request towards a container instance must include a `Maconomy-Concurrency-Control` header, passing the most recently received concurrency tag. The request will proceed only if the actual state of the container instance on the server matches the concurrency tag received from the client program.

In those cases where a client program provides an invalid concurrency tag when a `Maconomy-Concurrency-Control` header is required or where the `DataChanged` check fails, the request is rejected by the server and a `409 Conflict` response is returned.

Hyperlinks mentioning one of the following link relations are all referencing some action for which a valid concurrency tag must be supplied in a `Maconomy-Concurrency-Control` header on the request:

`data:any-key`

`data:key`

`data:same-key`

`data:same-key-some-instance`

`data:same-key-some-container`

`instance:configuration-update`

`instance:data-refresh`

`action:`*xxx*

The `Maconomy-Concurrency-Control` header is not allowed with any other requests. If applied anyway, the server will respond with a `400 Bad Request` response.

4.4.2 Data Resource

A main component of a container instance is its data resource where all information related to the data entry currently in focus is gathered.

How a data entry gets loaded into the data resource of a container instance is covered in Creating a Data Entry and Loading a Data Entry. In the sections Adding a Table Record, Updating a Record, Deleting a Record, Moving a Table Record, Printing, and Applying an Application Action you can find descriptions on how to perform different actions on a data entry once loaded.

The current state of the data resource of a container instance can be acquired by following the `instance:data` hyperlink available from a JSON object representing the container instance:

```
{
  "href": "http://SERVER/BASEPATH/containers/SHORTNAME/CONTAINER/instances ↩
    /INSTANCE/data",
  "rel": "instance:data"
}
```

Authentication is required and, as explained in the beginning of Container Instances, a reconnect token valid for the container instance in question must be used for this (see Maconomy Reconnect Authentication). The HTTP verb applied may be either `GET` or `POST` where in the latter case, the client program is able to include paging parameters in the request body (see Table Paging).

If no data entry is currently held in the data resource, the response to an `instance:data` request is a `204 No Content`. Otherwise, the client program receives a so-called *data response* with a data entry JSON representation in the body holding the following properties:

Property	Explanatory text
`meta`	`containerName`: The name of the container. `containerInstanceId`: The identifier of the container instance.
`panes`	Representation of the data entry's card and table pane contents held in a `card` and a `table` property respectively. Find the properties of a JSON object representing pane contents below.
`links`	Hyperlinks available from the data resource. Find the list of link relations below.

These are the properties of a JSON object within a data response representing pane contents:

Property	Explanatory text
`meta`	`paneName`: The name of the card or table pane. `rowCount`: The number of data pane records returned. This will always be 1 for a card pane, but for a table pane, the number depends on the actual amount of records and any Table Paging applied. `rowOffset`: The offset of the data pane records returned. This will always be 0 for a card pane, but for a table pane, the number depends on any Table Paging applied.

Property	Explanatory text
`records`	List of JSON objects each representing a record within the data pane. Each JSON object has a `data` property holding a name/value entry for every record field included. Note that only key fields are included, unless otherwise specified by the current Data Fields Slicing. For table panes, the order in which the JSON objects are listed depends on any Table Sorting applied. For tree table panes with hierarchically organized records, each JSON object in the list may also hold a `records` property with the same characteristics as this outermost `records` property. Since a card pane always carries exactly one record, the list contains exactly one JSON object in case of a card pane. Note that the `records` property often is replaced by a `recordsPatch` property in Partial Data Responses.
`links`	Hyperlinks available for the data pane. Each of these hyperlinks is a reference to one of the actions described in Actions. Note that the `links` property often is replaced by a `linksPatch` in Partial Data Responses.

The purposes of the hyperlinks available from the outermost `links` property of a data response are these:

Link relation	Explanatory text
`data:same-key`	Reference to the action of loading the data entry exposed in the data response into the data resource, see Loading a Data Entry.
`data:restore`	Reference to the action of loading the data entry exposed in the data response into the data resource of some other container instance, see Loading a Data Entry. This may be relevant in cases where an auto log-out forces restoration.
`self`	Hyperlink referring to the data resource itself.

To refresh the data entry currently held in the data resource of a container instance, the client program must follow the hyperlink with link relation `instance:data-refresh` also available from the JSON object representing the container instance:

```
{
  "href": "http://SERVER/BASEPATH/containers/SHORTNAME/CONTAINER/instances ↩
    /INSTANCE/data/refresh",
  "rel": "instance:data-refresh"
```

```
}
```

Again, the client program must authenticate using a reconnect token valid for the container instance in use (see Maconomy Reconnect Authentication), and since an `instance:data-refresh` request may cause a change in the state of that container instance, the client program must provide a valid concurrency tag (see Concurrency Tags). The client program must apply the HTTP verb `POST` and possibly include paging parameters in the request body (see Table Paging).

If no data entry has been loaded, the response to an `instance:data-refresh` request is a `204 No Content` one. Otherwise, a data response reflecting the refreshed state of the loaded data entry is returned. If contracted as described in Partial Data Responses, this may be a partial data response.

4.4.3 Data Fields Slicing

For a lot of data containers, the amount of fields defined for their data panes is huge, and since in many cases only a smaller subset of these fields are of actual interest to the client program, by default, only key fields are included in data responses. This to avoid that a lot of unnecessary data is potentially being transmitted.

If some non-key fields within a data pane are of interest to the client program, these fields should be picked out at the time when the client program is acquiring the container instance. Key fields are always considered of interest and will be included, no matter if they have been picked out by the client program or not.

Except for key fields, fields that are not specifically picked out by the client program cannot be mentioned in later update requests (see Updating a Record) and they are all left out of any data response (see Data Resource). Record fields thus excluded are said to be victims of *data fields slicing*.

See Configuring an Instance for further details on how a data fields slicing is actually configured.

4.4.4 Table Sorting

If the client program wishes to receive the JSON objects representing table pane records in a certain order in data responses, a multi-column sorting can be specified at the time of container instance creation. Further details on how this is done can be found in Configuring an Instance.

Note that for tree table panes, any configured multi-column sorting will be applied on each level of records.

Note that whenever a multi-column sorting of table pane records has been configured for a container instance, Adding a Table Record or Moving a Table Record using that

container instance may be a bit confusing, since the added or moved record will show up at the position resulting from the sorting being applied.

4.4.5 Configuring an Instance

In Data Fields Slicing and Table Sorting it is described how, for a container instance, the client program is able to configure which record fields are exposed in data responses and which multi-column sorting is applied to any table pane record JSON objects of these.

Such data fields slicing and table sorting can be specified during container instance creation by passing a so-called *instance configuration* JSON object of the following structure in the body of an `instance:create` request:

```
{
   "panes" : {
     "card" : {
       "fields" : CARD_FIELD_NAMES
     },
     "table" : {
       "fields" : TABLE_FIELD_NAMES,
       "orderBy" : SORT_ORDERS
     }
   }
}
```

Here, `CARD_FIELD_NAMES` and `TABLE_FIELD_NAMES` denote a list of field names for the card and table pane respectively and indicate the fields of interest in each pane. The `SORT_ORDERS` denotes a value similar to the one described in Filter Sorting.

For example, if the client program was in fact only interested in the `Description` field of expense sheets and the `Text` field of their lines and wanted the latter ordered first descending by `EntryDate` and then ascending by `Text`, the following JSON object could have been included in the body of the `instance:create` request for the `ExpenseSheets` container we saw above:

```
{
   "panes" : {
     "card" : {
       "fields" : [ "Description" ]
     },
     "table" : {
       "fields" : [ "Text" ],
       "orderBy" : [
         { "field": "EntryDate", "ascending": false },
         { "field": "Text" }
       ]
     }
```

```
    }
}
```

If necessary, the client program can see the data fields slicing and table sorting currently applied for a given container instance by following the `instance:configuration` hyperlink available from the container instance resource:

```
{
  "href": "http://SERVER/BASEPATH/containers/SHORTNAME/CONTAINER/instances ↩
    /INSTANCE/configuration",
  "rel": "instance:configuration"
}
```

As any other container instance request, such a request requires the client program to authenticate using a reconnect token valid for the given container instance (see Maconomy Reconnect Authentication). Since the request does not cause any changes to the state of the container instance, no concurrency tag is required nor allowed (see Concurrency Tags).

The response to an `instance:configuration` request is called an *instance configuration response*, and the JSON object held in its body is an instance configuration JSON object extended with the following two properties:

Property	Explanatory text
`meta`	`containerName`: Name of the container. `containerInstanceId`: The identifier of the container instance.
`links`	Hyperlinks available from the instance configuration response. Find the list of link relations below.

These are the purposes of the hyperlinks available from an instance configuration response:

Link relation	Explanatory text
`instance:configuration-update`	Reference to the action of updating the configuration of the container instance. Find further explanation below.
`self`	Reference to the instance configuration resource itself.

For example, with the above instance configuration JSON object passed on `ExpenseSheets`

instance creation, a succeeding `instance:configuration` request would unfold something like this:

```
$ curl -i
       -H 'Authorization: X-Reconnect Zjk1YjUzMT...JMTAJMTYwMjY4MDE4NQ==='
       -H 'Accept-Language: en-US'
       -H 'Accept: application/vnd.deltek.maconomy.containers+json;  ↩
    charset=utf-8; version=6.0'
       'http://SERVER/maconomy-api/containers/macoprod/expensesheets/ ↩
    instances/7dc0b114-ecf3-4441-940e-cf93dcf58620/configuration'

HTTP/1.1 200 OK
Maconomy-Reconnect: YTNkNTAzN2...JMTAJMTYwNjgyNjUxNg==
Maconomy-Concurrency-Control: 9d497fb1-3533-4927-963b-ee204aa837be
Content-Type: application/vnd.deltek.maconomy.containers+json; charset=utf ↩
    -8; version=6.0

{
  "meta": {
    "containerName": "expensesheets",
    "containerInstanceId": "7dc0b114-ecf3-4441-940e-cf93dcf58620"
  },
  "panes": {
    "card": {
      "fields": [
        "description",
        "expensesheetnumber"
      ]
    },
    "table": {
      "fields": [
        "instancekey",
        "text"
      ],
      "orderBy" : [
         {
           "field": "entrydate",
           "ascending": false
         },
         {
           "field": "text",
           "ascending": true
         }
      ]
    }
  },
  "links": {
    "instance:configuration-update": {
      "href": "http://SERVER/maconomy-api/containers/macoprod/ ↩
```

```
    expensesheets/instances/7dc0b114-ecf3-4441-940e-cf93dcf58620/ ↩
    configuration",
      "rel": "instance:configuration-update"
    },
    "self": {
      "href": "http://SERVER/maconomy-api/containers/macoprod/ ↩
    expensesheets/instances/7dc0b114-ecf3-4441-940e-cf93dcf58620/ ↩
    configuration",
      "rel": "self"
    }
  }
}
```

Besides the field `Description` mentioned during creation of the container instance, the card pane's fields list mentions the key field `ExpenseSheetNumber`. Similarly, the key field `InstanceKey` appears together with `Text` in the table pane's fields list.

The client program can update the configuration of an existing container instance by following the `instance:configuration-update` hyperlink available from either the container instance resource or some instance configuration response:

```
{
  "href": "http://SERVER/BASEPATH/containers/SHORTNAME/CONTAINER/instances ↩
    /INSTANCE/configuration",
  "rel": "instance:configuration-update"
}
```

As always, the client program must authenticate using a valid reconnect token (see Maconomy Reconnect Authentication), and since an `instance:configuration-update` request may cause a change to the state of the container instance, a valid concurrency tag must also be provided (see Concurrency Tags). The client program must apply the HTTP verb `POST`, and an instance configuration JSON object representing the new data fields slicing and table sorting must be included in the request body.

The response to an `instance:configuration-update` request is an instance configuration response reflecting the new configuration.

4.4.6 Deleting an Instance

In order to avoid occupying unnecessary space on the server, it is very important that client programs delete their container instances as soon as these are no longer needed. Even though there will be arrangements in place on the server attending the task of cleaning out container instances that have been lying around untouched for too long, it is of course much preferred if client programs have the matters taken care of instantly.

A container instance can be deleted by following the hyperlink with link relation `instance:delete` available from the container instance resource:

```
{
  "href": "http://SERVER/BASEPATH/containers/SHORTNAME/CONTAINER/instances ↩
    /INSTANCE",
  "rel": "instance:delete"
}
```

Such a request requires authentication using a reconnect token appropriate for the container instance (see Maconomy Reconnect Authentication), but no concurrency tag is needed. The HTTP verb to apply is `DELETE`.

The response to an `instance:delete` request is a `204 No Content` response.

4.5 Working with Data

This section covers all the different kinds of interactions a client program can do with data within data containers reachable through the Containers Web Service. Common for these interactions is that they all go via a container instance, as explained in Container Instances.

4.5.1 Record Positions

In order to be able to update (see Updating a Record), delete (see Deleting a Record), move (see Moving a Table Record), print (see Printing) or otherwise process (see Applying an Application Action) a record within a data container pane, a way of pointing out existing records is needed. Furthermore, in case of a line-number controlled table pane, the client program also needs a way of specifying the target position of a record that is about to be either added (see Adding a Table Record) or moved (see Moving a Table Record).

Position of an Existing Record

Regardless of whether the records of some pane are hierarchically organized or not (a card pane's single record is never hierarchically organized, but the records within a tree table pane always are), the position of an existing record is pointed out by a so-called *dot index*:

Letting i_r denote an existing record r's flat index among its sibling records, then in the cases where r is residing at the outermost level, r's dot index I_r is simply given by i_r. If instead r is a child of some other record r', then I_r is given by the following:

$$I_{r'}.i_r$$

For example, having a records structure like the following, the dot indices of the records will be as noted on the right:

```
A                           0
     B                      0.0
     C                      0.1
          D                 0.1.0
     E                      0.2
F                           1
G                           2
```

Dot indices that only differ in their last index are called *sibling dot indices*, and for two sibling dot indices I_a and I_b with last index i_a and i_b respectively, I_a is said to be *less than* I_b, if $i_a < i_b$. Similarly, I_a is said to be *greater than* I_b, if $i_a > i_b$. For example, `0.1` and `0.4` are sibling dot indices where the first is less than the second and the second is greater than the first.

Dot index *incrementation* is the operation where the last index of the dot index is incremented by one. For dot indices whose last index is greater than zero, *decrementation* refers to the operation where the last index of the dot index is decremented by one. For example, incrementing the dot index `0.1` results in the dot index `0.2`, and decrementing `0.2` results in `0.1`.

A Target Position

Target positions are also pointed out by dot indices. Aside from dot indices pointing out positions where other records are already residing, the following are also valid target position dot indices:

- In case of an empty pane, the dot index `0`.
- Any dot index pointing out the position just after some last record r.

A dot index pointing out a position where no record resides is called an *end dot index*, and the client program is allowed to replace its last index by the token `end`. For example, the position as the above record `E`'s first child can be targeted by supplying either the end dot index `0.2.0` or the end dot index `0.2.end`.

Notice that in cases of move operations (see Moving a Table Record), the target position relates to the records structure as it looks prior to any changes. For example, if record `C` in the above records structure is requested to be moved to `0.2.end`, it will actually end up at the position with dot index `0.1.0`:

```
A                           0
     B                      0.0
     E                      0.1
          C                 0.1.0
               D            0.1.0.0
F                           1
G                           2
```

4.5.2 Creating a Data Entry

Creating a new data entry in a data container corresponds to creating a new card pane record.

Creating a new card pane record comprises the following two steps:

1. Submit an initialization request and obtain a record template holding default field values.
2. Submit a creation request supplying the possibly adjusted version of the record template.

In order to allow the client program to do an initialization request without proceeding with a creation request (one could, for example, imagine a user never reaching the point of hitting the save button in a client interface), the initialization state triggered by the initialization request will not affect the state of the container instance. Instead the initialization state is stored in a separate temporary resource which is then presented to the client program for further interaction.

The hyperlink to follow in order to acquire a new temporary initialization resource for a card pane record is the one with link relation `action:init` available from the container instance resource:

```
{
  "href": "http://SERVER/BASEPATH/containers/SHORTNAME/CONTAINER/instances ↩
    /INSTANCE/data/panes/card/inits",
  "rel": "action:init"
}
```

As always when working with a container instance, the client program must authenticate using a valid reconnect token (see Maconomy Reconnect Authentication), and since an `action:init` request may rely on the state of the container instance, the client program must also supply a valid concurrency tag (see Concurrency Tags). The client program must apply the HTTP verb `POST` and the body of the request must be kept empty.

The response to an `action:init` request has a JSON object in the body representing the created temporary initialization resource. These are the properties of such a JSON object:

Property	Explanatory text
`meta`	`containerName`: Name of the container. `containerInstanceId`: Identifier of the container instance. `paneName`: Name of the container pane.
`data`	JSON object holding a name/default value entry for each field of a record within the container pane.

Property	Explanatory text
`links`	Hyperlinks available from the record initialization resource. Find the list of link relations below.

The purposes of the hyperlinks available from a card pane record initialization resource are these:

Link relation	Explanatory text
`action:create`	Reference to the action of creating the record just initialized. Find further details about this below.
`self`	Hyperlink referring to the record initialization resource itself.

For example, to initialize a new expense sheet:

```
$ curl -i
       -H 'Authorization: X-Reconnect Zjk1YjUzMT...JMTAJMTYwMjY4MDE4NQ=='
       -H 'Accept-Language: en-US'
       -H 'Accept: application/vnd.deltek.maconomy.containers+json;  ↩
    charset=utf-8; version=6.0'
       -H 'Maconomy-Concurrency-Control: d2a39243-a63f-4bd5-8eab-676952009 ↩
    e93'
       -X POST
       'http://SERVER/maconomy-api/containers/macoprod/expensesheets/ ↩
    instances/7dc0b114-ecf3-4441-940e-cf93dcf58620/data/panes/card/inits'

HTTP/1.1 200 OK
Maconomy-Reconnect: MmYxZGUyNz...JMTAJMTYwNjE3MTM5Mw==
Maconomy-Concurrency-Control: d2a39243-a63f-4bd5-8eab-676952009e93
Content-Type: application/vnd.deltek.maconomy.containers+json; charset=utf ↩
    -8; version=6.0

{
  "meta": {
    "containerName": "expensesheets",
    "containerInstanceId": "7dc0b114-ecf3-4441-940e-cf93dcf58620",
    "paneName": "card"
  },
  "data": {
    "expensesheetnumber": "",
    "description": "",
    "employeenumber": "11",
    "companynumber": "",
    "createdby": "",
```

```
    "createddate": "",
    ...
  },
  "links": {
    "action:create": {
      "href": "http://SERVER/maconomy-api/containers/macoprod/ ↩
    expensesheets/instances/7dc0b114-ecf3-4441-940e-cf93dcf58620/data/ ↩
    panes/card/inits/0cf5c664-0fdd-4720-8505-8e15a3b6c8e9",
      "rel": "action:create"
    },
    "self": {
      "href": "http://SERVER/maconomy-api/containers/macoprod/ ↩
    expensesheets/instances/7dc0b114-ecf3-4441-940e-cf93dcf58620/data/ ↩
    panes/card/inits/0cf5c664-0fdd-4720-8505-8e15a3b6c8e9",
      "rel": "self"
    }
  }
}
```

Here the concurrency tag `d2a39243-a63f-4bd5-8eab-676952009e93` supplied by the client program being the same as the one received from the server witnesses that the expense sheet initialization has not changed the state of the container instance, but just triggered the creation of a new initialization resource.

Once the record template received as part of the initialization resource representation has been adjusted to reflect the characteristics of the new card pane record, the actual record creation can be accomplished by making the client program follow the `action:create` hyperlink available from the initialization resource:

```
{
  "href": "http://SERVER/BASEPATH/containers/SHORTNAME/CONTAINER/instances ↩
    /INSTANCE/data/panes/card/inits/INIT",
  "rel": "action:create"
}
```

Besides authenticating using a valid reconnect token (see Maconomy Reconnect Authentication), the client program must apply the HTTP verb `POST` with the updated card pane record template in the body and the valid `Maconomy-Concurrency-Control` header enclosed (see Concurrency Tags). If relevant, paging parameters are passed through extra JSON properties in the body of the request (see Table Paging).

The response to a successful `action:create` request is a data response (see Data Resource) and subsequently the newly created data entry is the one referenced by the data resource of the container instance.

Note that any temporary initialization resource vanishes as soon as the state of the container instance changes (like if the record creation is completed, for example) or if the client program is instructed to perform another record initialization.

Accomplishing the creation of the expense sheet initialized above:

```
$ curl -i
       -H 'Authorization: X-Reconnect MmYxZGUyNz...JMTAJMTYwNjE3MTM5Mw==='
       -H 'Accept-Language: en-US'
       -H 'Accept: application/vnd.deltek.maconomy.containers+json;  ↩
    charset=utf-8; version=6.0'
       -H 'Maconomy-Concurrency-Control: d2a39243-a63f-4bd5-8eab-676952009 ↩
    e93'
       -H 'Content-Type: application/vnd.deltek.maconomy.containers+json;  ↩
    charset=utf-8; version=6.0'
       -d '{
         "data": {
           "expensesheetnumber": "",
           "description": "Yet another expense sheet",
           "employeenumber": "11",
           "companynumber": "",
           "createdby": "",
           "createddate": "",
           ...
         }
       }'
       'http://SERVER/maconomy-api/containers/macoprod/expensesheets/ ↩
    instances/7dc0b114-ecf3-4441-940e-cf93dcf58620/data/panes/card/inits/0 ↩
    cf5c664-0fdd-4720-8505-8e15a3b6c8e9'

HTTP/1.1 200 OK
Maconomy-Reconnect: NDZhZmNiNG...JMTAJMTYwNjE3NTAxMw==
Maconomy-Concurrency-Control: 8a610472-b72a-4369-a124-6c1057c32f0c
Content-Type: application/vnd.deltek.maconomy.containers+json; charset=utf ↩
    -8; version=6.0

{
  "meta": {
    "containerName": "expensesheets",
    "containerInstanceId": "7dc0b114-ecf3-4441-940e-cf93dcf58620"
  },
  "panes": {
    "card": {
      "meta": {
        "paneName": "card",
        "rowCount": 1,
        "rowOffset": 0
      },
      "records": [
        {
          "data": {
            "expensesheetnumber": "10760004",
            "description": "Yet another expense sheet",
```

```
            "employeenumber": "11",
            "companynumber": "1",
            "createdby": "Administrator",
            "createddate": "2020-11-24",
            ...
          }
        }
      ],
      "links": {
        "action:init": {
          "href": "http://SERVER/maconomy-api/containers/macoprod/ ↩
    expensesheets/instances/7dc0b114-ecf3-4441-940e-cf93dcf58620/data/ ↩
    panes/card/inits",
          "rel": "action:init"
        },
        "action:init-create": {
          "href": "http://SERVER/maconomy-api/containers/macoprod/ ↩
    expensesheets/instances/7dc0b114-ecf3-4441-940e-cf93dcf58620/data/ ↩
    panes/card",
          "rel": "action:init-create"
        },
        ...
      }
    },
    "table": {
      "meta": {
        "paneName": "table",
        "rowCount": 0,
        "rowOffset": 0
      },
      "records": [],
      "links": {
        "action:init-row": {
          "href": "http://SERVER/maconomy-api/containers/macoprod/ ↩
    expensesheets/instances/7dc0b114-ecf3-4441-940e-cf93dcf58620/data/ ↩
    panes/table/inits",
          "rel": "action:init-row"
        },
        "action:init-create-row": {
          "href": "http://SERVER/maconomy-api/containers/macoprod/ ↩
    expensesheets/instances/7dc0b114-ecf3-4441-940e-cf93dcf58620/data/ ↩
    panes/table",
          "rel": "action:init-create-row"
        }
      }
    }
  },
  "links": { ... }
}
```

As you can see, the new expense sheet has been assigned the number `10760004` and the updated concurrency tag witnesses the fact that the new sheet is now the one in focus within the container instance.

Since expense sheet `10760004` does not yet have any expense sheet lines, the `records` property of the table pane's JSON object holds an empty list. Both hyperlinks available from the `links` property there relate to expense sheet line creation and is further described in Adding a Table Record.

The `action:init` and the `action:init-create` hyperlink available from the `links` property of the card pane's JSON object are similar to the ones available from the container instance resource. The appearance of these hyperlinks in both places is a general thing and nothing particular for expense sheets.

In situations without any great need of a card pane record template, initialization and creation can be combined into a single request by following the `action:init-create` hyperlink available from the container instance resource:

```
{
  "href": "http://SERVER/BASEPATH/containers/SHORTNAME/CONTAINER/instances ↩
    /INSTANCE/data/panes/card",
  "rel": "action:init-create"
}
```

The usage of the `action:init-create` hyperlink is similar to what was just described for the `action:create` hyperlink. Aside from a `data` JSON property carrying appropriate field name/values for the new card pane record, relevant paging parameter JSON properties (see Table Paging) should be included in the request body.

4.5.3 Loading a Data Entry

A client program is able to interact with a data entry within a data container only if that entry is the one currently being referenced by the data resource of a container instance.

In Creating a Data Entry it was described how new data entries go into focus once created, and in this section it is disclosed where hyperlinks to follow in order to load an already existing data entry can be discovered.

If the client program follows the hyperlink with link relation `data:any-key` available from a container instance resource, an unspecified (yet deterministic) data entry is loaded into the data resource of the container instance:

```
{
  "href": "http://SERVER/BASEPATH/containers/SHORTNAME/CONTAINER/instances ↩
    /INSTANCE/data;any",
  "rel": "data:any-key"
}
```

The client program must authenticate using a valid reconnect token (see Maconomy Reconnect Authentication), and since a `data:any-key` request affects the state of the container instance, the client program must also supply a valid concurrency tag (see Concurrency Tags). The client program must apply the HTTP verb `POST` and paging parameters may be included in the request body (see Table Paging).

The response to a `data:any-key` request is a data response holding the JSON representation of the current state of the data entry put into focus (see Data Resource).

For example, doing a `data:any-key` request against the `ExpenseSheets` container instance we acquired in Container Instances:

```
$ curl -i
       -H 'Authorization: X-Reconnect NDZhZmNiNG...JMTAJMTYwNjE3NTAxMw=='
       -H 'Accept-Language: en-US'
       -H 'Accept: application/vnd.deltek.maconomy.containers+json;  ↩
    charset=utf-8; version=6.0'
       -H 'Maconomy-Concurrency-Control: 8a610472-b72a-4369-a124-6 ↩
    c1057c32f0c'
       -X POST
       'http://SERVER/maconomy-api/containers/macoprod/expensesheets/ ↩
    instances/7dc0b114-ecf3-4441-940e-cf93dcf58620/data;any'

HTTP/1.1 200 OK
Maconomy-Reconnect: YTg1ZGY3Ym...wCTEwCTE2MDYyMzE3Mzg=
Maconomy-Concurrency-Control: 8db720b5-015d-4cfe-926a-3edf96f3e724
Content-Type: application/vnd.deltek.maconomy.containers+json; charset=utf ↩
    -8; version=6.0

{
  "meta": {
    "containerName": "expensesheets",
    "containerInstanceId": "7dc0b114-ecf3-4441-940e-cf93dcf58620"
  },
  "panes": {
    "card": {
      "meta": {
        "paneName": "card",
        "rowCount": 1,
        "rowOffset": 0
      },
      "records": [
        {
          "data": {
            "expensesheetnumber": "10760001",
            "description": "Expenses, expenses, expenses...",
            "employeenumber": "11",
            "companynumber": "1",
            ...
```

```
          }
        }
      ],
      "links": {
        ...,
        "action:submitexpensesheet": {
          "href": "http://SERVER/maconomy-api/containers/macoprod/ ↩
    expensesheets/instances/7dc0b114-ecf3-4441-940e-cf93dcf5862/data/panes ↩
    /card/0/action;name=submitexpensesheet",
          "rel": "action:submitexpensesheet"
        },
        ...
      }
    },
    "table": {
      "meta": {
        "paneName": "table",
        "rowCount": 32,
        "rowOffset": 0
      },
      "records": [ ... ],
      "links": { ... }
    }
  },
  "links": {
    "data:same-key": {
      "href": "http://SERVER/maconomy-api/containers/macoprod/ ↩
    expensesheets/instances/7dc0b114-ecf3-4441-940e-cf93dcf58620/data; ↩
    expensesheetnumber=10760001",
      "rel": "data:same-key"
    },
    "data:restore": {
      "template": "http://SERVER/maconomy-api/containers/macoprod/ ↩
    expensesheets/instances/{instance}/data;expensesheetnumber=10760001",
      "rel": "data:restore"
    },
    "self": {
      "href": "http://SERVER/maconomy-api/containers/macoprod/ ↩
    expensesheets/instances/7dc0b114-ecf3-4441-940e-cf93dcf58620/data",
      "rel": "self"
    }
  }
}
```

Besides an updated concurrency tag, here are examples of the `data:same-key` and the `data:restore` hyperlink always available from a full data response.

The `data:same-key` hyperlink is meant for the client program to follow if it at some point becomes relevant to load the data entry represented in the data response into the

data resource of the container instance again:

```
{
  "href": "http://SERVER/BASEPATH/containers/SHORTNAME/CONTAINER/instances ↩
    /INSTANCE/data;KEY",
  "rel": "data:same-key"
}
```

The `data:restore` hyperlink, however, is useful, if the container instance gets lost for some reason (for example, if the user gets logged out due to inactivity) and the client program needs to restore its state using a new container instance:

```
{
  "template": "http://SERVER/BASEPATH/containers/SHORTNAME/CONTAINER/ ↩
    instances/{instance}/data;KEY",
  "rel": "data:restore"
}
```

Once the `{instance}` placeholder in the `data:restore` template URL has been replaced by the id of some new container instance, the usage of the `data:same-key` and `data:restore` hyperlinks is similar to what was described for the `data:any-key` hyperlink above.

Unless the container contains exactly one entry, a `data:any-key` hyperlink is normally not very useful, as the client program typically wants to interact with a specific data entry and not just some unspecified (yet deterministic) entry. In cases where the client program already knows the key of some specific data entry, the `data:some-key` hyperlink also available from a container instance resource may come in handy:

```
{
  "template": "http://SERVER/BASEPATH/containers/SHORTNAME/CONTAINER/ ↩
    instances/INSTANCE/data;KEY_FIELD_NAME_1={0};KEY_FIELD_NAME_2 ↩
    ={1};...",
  "fields": [
    "KEY_FIELD_NAME_1",
    "KEY_FIELD_NAME_2",
    ...
  ],
  "rel": "data:some-key"
}
```

Here the `fields` property holds an array listing the names of the fields participating in a data key for the container, and the index of a field's name within this array indicates which of the numbered placeholders in the template URL its value should replace. Once an entire key has been substituted into the template URL, the usage of the `data:some-key` hyperlink is similar to what was described for the `data:any-key` hyperlink above.

In cases where the client program does not know the keys of desired data entries beforehand, filter panes are very useful. In Filtering it is described how these panes allow the client program to discover hyperlinks referencing specific data entries. For data containers where the records of a filter pane can be thought of as thumbnails describing the container's data entries, a `data:same-key-some-instance` hyperlink is available from each record presented in a filter response:

```
{
  "template": "http://SERVER/BASEPATH/containers/SHORTNAME/CONTAINER/ ↩
    instances/{instance}/data;KEY",
  "rel": "data:same-key-some-instance"
}
```

Just like with the `data:restore` hyperlink described above, a container instance id must be substituted into the `{instance}` placeholder, but besides from that, the usage of a `data:same-key-some-instance` hyperlink is similar to what was described for the `data:any-key` hyperlink above.

Since no search container ever hosts any actual data (remember that a search container is a container with only a filter pane defined), Filtering describes how the hyperlink available from each record presented in a search container filter response is not a `data:same-key-some-instance` but instead a `data:same-key-some-container` hyperlink:

```
{
  "template": "http://SERVER/BASEPATH/containers/SHORTNAME/{container}/ ↩
    instances/{instance}/data;KEY",
  "rel": "data:same-key-some-container"
}
```

Here, the `{container}` placeholder must be replaced by the name of a data container whose card pane is based on the same entity as the search container, and the `{instance}` placeholder must be replaced by an id of an instance of that data container. Aside from this, the usage of a `data:same-key-some-container` hyperlink is similar to what was described for the `data:any-key` hyperlink above.

As mentioned in Foreign Keys, the data entry navigation hyperlinks available from any search response are `data:same-key-some-container` hyperlinks as well. There you can also see how a client program can navigate to data entries by use of the `data:key` hyperlink available for any complete foreign key.

4.5.4 Adding a Table Record

Similar to Creating a Data Entry, adding a new record to a table pane of the data entry currently in focus comprises a step of initialization and a step of actual creation.

For table panes *without* line-number control, the link relation of the hyperlink to follow in order to perform a record initialization is `action:init`, just like for card panes:

```
{
  "href": "http://SERVER/BASEPATH/containers/SHORTNAME/CONTAINER/instances ↩
    /INSTANCE/data/panes/table/inits",
  "rel": "action:init"
}
```

Whenever applicable, an `action:init` hyperlink will be available from the table pane's JSON object within a full data response (see Data Resource) and its usage is entirely as described in Creating a Data Entry. The same goes for the `action:init-create` hyperlink to be followed in order to perform a merged action of table pane record initialization and creation:

```
{
  "href": "http://SERVER/BASEPATH/containers/SHORTNAME/CONTAINER/instances ↩
    /INSTANCE/data/panes/table",
  "rel": "action:init-create"
}
```

Obviously, the client program has no influence on where a new record ends up in a table pane without line-number control.

For a table pane *with* line-number control, specifically specifying where in the pane the new record shall appear is part of the game. In such line-number control cases, the hyperlinks to look for in the table pane's JSON object within a full data response is an `action:init-row` and an `action:init-create-row` hyperlink instead of an `action:init` and an `action:init-create` hyperlink respectively:

```
{
  "href": "http://SERVER/BASEPATH/containers/SHORTNAME/CONTAINER/instances ↩
    /INSTANCE/data/panes/table/inits",
  "rel": "action:init-row"
}

{
  "href": "http://SERVER/BASEPATH/containers/SHORTNAME/CONTAINER/instances ↩
    /INSTANCE/data/panes/table",
  "rel": "action:init-create-row"
}
```

The usage of an `action:init-row` and an `action:init-create-row` hyperlink is similar to that of an `action:init` and an `action:init-create` hyperlink except from a requirement that the request body carries a JSON property `row` specifying the preferred position for the new record.

In accordance with what is described in Record Positions, the target position of the new record must be one of the following:

- A dot index pointing out an already existing record r.
- A valid end dot index.

In the first case, the record r and all its sibling records positioned at larger dot indices are shifted one down (their dot index is incremented by one), and then the new record is inserted where record r used to reside. In the second case, the new record is appended at the position pointed out by the given end dot index.

For example, in order to insert a new first line into the non-empty expense sheet currently selected, the following `row` parameter should be provided:

```
{
  "row": 0
}
```

Just like for `action:init`, the response to an `action:init-row` request presents an `action:create` hyperlink, that the client program needs to follow in order to actually create the new record.

If an invalid position is supplied when adding a new record, the server responds with a `400 Bad Request` response. If the request is successfully processed, a data response is returned, and, if contracted as described in Partial Data Responses, this may be a partial one.

4.5.5 Updating a Record

A common need when working with a data container is to be able to update certain field values within some record belonging to either a card or a table pane.

The hyperlink to follow in order to update the card pane record within the data entry currently in focus is the `action:update` hyperlink available from the card pane's JSON object within a full data response (see Data Resource):

```
{
  "href": "http://SERVER/BASEPATH/containers/SHORTNAME/CONTAINER/instances ↩
    /INSTANCE/data/panes/card/0",
  "rel": "action:update"
}
```

Aside from authenticating using a reconnect token valid for the container instance in use (see Maconomy Reconnect Authentication), the client program must apply the HTTP verb `POST` with a JSON object in the request body carrying a `data` property describing the field values to be updated and possibly also an `offset` and a `limit` property (see

Table Paging). Of course an appropriate `Maconomy-Concurrency-Control` header needs to be enclosed in the request as well (see Concurrency Tags).

The response to a successful update request on a card pane record is a data response, and, if contracted as described in Partial Data Responses, this may be a partial one.

Here is an example of updating the description of expense sheet number `10760001` currently in focus:

```
$ curl -i
       -H 'Authorization: X-Reconnect N2U1NWY4YT...JMTAJMTYwNzYwMDk5MA=='
       -H 'Accept-Language: en-US'
       -H 'Accept: application/vnd.deltek.maconomy.containers+json;  ↩
    charset=utf-8; version=6.0'
       -H 'Maconomy-Concurrency-Control: 915ed0db-8fc5-450b-93c7- ↩
    f03b1b91345c'
       -d $'{
         "data": {
           "description": "Expenses, expenses, expenses, expenses..."
         }
       }'
       'http://SERVER/maconomy-api/containers/macoprod/expensesheets/ ↩
    instances/7dc0b114-ecf3-4441-940e-cf93dcf58620/data/panes/card/0'

HTTP/1.1 200 OK
Maconomy-Reconnect: ODI2NjJhNz...JMTAJMTYwNzYwNDk4OQ==
Maconomy-Concurrency-Control: 386508e9-0f1b-4d7d-8fb4-927b76de1303
Content-Type: application/vnd.deltek.maconomy.containers+json; charset=utf ↩
    -8; version=6.0

{
  "meta": { ... },
  "panes": {
    "card": {
      "meta": { ... },
      "records": [
        {
          "data": {
            "expensesheetnumber": "10760001",
            "description": "Expenses, expenses, expenses, expenses...",
            ...
          }
        }
      ],
      "links": { ... }
    },
    "table": { ... }
  },
  "links": { ... }
}
```

Now, for a record belonging to a table pane of the data entry currently in focus, the hyperlink to follow in order to update some of its field values is the `action:update` hyperlink available from the table pane's JSON object within a full data response:

```
{
  "template": "http://SERVER/BASEPATH/containers/SHORTNAME/CONTAINER/ ↩
    instances/INSTANCE/data/panes/table/{row}",
  "rel": "action:update"
}
```

Once the `{row}` placeholder has been replaced by the dot index pointing out the table pane record to be updated (see Record Positions), the usage is similar to what was described for the card pane record above.

If an invalid dot index is substituted into the `{row}` placeholder of the `action:update` template URL, the server responds with a `400 Bad Request` response. Otherwise, the response is similar to the one received on a card pane record update request.

4.5.6 Deleting a Record

Deleting the record from the card pane of the data entry currently in focus corresponds to deleting the entire data entry (for example, if you delete an expense sheet, all of its lines are also deleted), and the hyperlink to follow in order to do so is the one with link relation `action:delete` available from the card pane's JSON object within a full data response (see Data Resource):

```
{
  "href": "http://SERVER/BASEPATH/containers/SHORTNAME/CONTAINER/instances ↩
    /INSTANCE/data/panes/card/0/delete",
  "rel": "action:delete"
}
```

Aside from authenticating using a reconnect token valid for the container instance in use (see Maconomy Reconnect Authentication), the client program must apply the HTTP verb `POST` and include a valid concurrency tag in a `Maconomy-Concurrency-Control` header (see Concurrency Tags). The request body must be kept empty.

The response to a successful delete request for a card pane record is a `204 No Content` response. Subsequently, until a new data entry has been loaded, this is also the response when acquiring the data resource of the container instance (see Data Resource).

To have a record deleted from a table pane of the data entry currently in focus, the client program must follow the `action:delete` hyperlink available from the table pane's JSON object within a full data response:

```
{
  "template": "http://SERVER/BASEPATH/containers/SHORTNAME/CONTAINER/ ↩
    instances/INSTANCE/data/panes/table/{row}/delete",
  "rel": "action:delete"
}
```

Once the `{row}` placeholder has been replaced by the dot index pointing out the table pane record to be deleted (see Record Positions), the usage is similar to what was described for the card pane record above, except from paging parameters being allowed in the request body (see Table Paging).

If an invalid dot index is substituted into the `{row}` placeholder of the `action:delete` template URL, the server responds with a `400 Bad Request` response. Otherwise, the response to a successful delete request for a table pane record is a data response. If contracted as described in Partial Data Responses, this data response may be a partial one.

4.5.7 Moving a Table Record

Sometimes it makes sense to have a record within a line-number controlled table pane moved to a different position. The hyperlink to follow in order to move a table pane record around within the data entry currently in focus has the link relation `action:move` and is available from the table pane's JSON object within a full data response (see Data Resource):

```
{
  "template": "http://SERVER/BASEPATH/containers/SHORTNAME/CONTAINER/ ↩
    instances/INSTANCE/data/panes/table/{row}/move",
  "rel": "action:move"
}
```

Letting r denote the record to be moved, the `{row}` placeholder within the template URL mentioned here must be replaced by the dot index currently pointing out record r (see Record Positions).

In accordance with what is described in Record Positions, the new position for record r must be chosen among the target positions whose dot index fulfills one of the following two conditions:

- A dot index pointing out a record r' different from record r and any of r's descendant records.
- A valid end dot index.

In the first case, record r (with all its descendant records) is moved to the position just above record r'. In the second case, record r (again with all its descendant records) is

moved to the position pointed out by the given end dot index. Record indices are then adjusted in the way naturally closing the gap left by r.

Note that if record r is moved to a target position corresponding to an outdent, then for some tree table panes (the one defined for the `JobBudgets` container, for example), records that used to be siblings of r and positioned at larger dot indices will be moved, one by one, to the end position among r's child records.

Note that for other tree table panes (the one defined for the `PeriodicSumJobBudgets` container, for example), the target dot index supplied with the move request must point to a position within record r's current context. That is, for some tree table panes, a record cannot be moved to a position changing its parental relation. Whether the move action is restricted in such ways can be discovered by inspection of the property `moveMode` found in the pane's specification, see Specification.

Besides authenticating using a reconnect token valid for the container instance in use (see Maconomy Reconnect Authentication), the client program must include a valid concurrency tag in a `Maconomy-Concurrency-Control` header (see Concurrency Tags) and apply the HTTP verb `POST`. A `row` JSON property carrying the dot index pointing out r's target position and any relevant paging parameter JSON properties (see Table Paging) must be included in the request body.

If either the dot index substituted into the `{row}` placeholder of the `action:move` template URL or the dot index carried by the `row` JSON property in the request body is found invalid, the server responds with a `400 Bad Request` response. Otherwise, the response to a successful move request is a data response (if contracted as described in Partial Data Responses, this data response may be a partial one).

For example, using the following substituted `action:move` template URL and request body JSON object, the second line of the expense sheet currently in focus would be converted into the last:

```
http://SERVER/maconomy-api/containers/macoprod/expensesheets/instances/7 ↩
    dc0b114-ecf3-4441-940e-cf93dcf58620/data/panes/table/1/move

{
  "row": "end"
}
```

4.5.8 Printing

Printing is one of the standard actions (see Actions) and is available from both card and table pane records (in most cases, printing from any table pane record is equivalent to printing from the card pane record). When a printing request is received on the server, generation of a PDF file with contents appropriate for the pane record in focus is triggered. The generated PDF file can be retrieved by following the URL shared with the client program through a `Link` header on the response.

The hyperlink to follow in order to initiate printing from the card pane record of the data entry currently in focus is the `action:print` hyperlink available from the card pane's JSON object within a full data response (see Data Resource):

```
{
  "href": "http://SERVER/BASEPATH/containers/SHORTNAME/CONTAINER/instances ↩
    /INSTANCE/data/panes/card/0/print",
  "rel": "action:print"
}
```

Besides authenticating using a reconnect token valid for the container instance in use (see Maconomy Reconnect Authentication), the client program must include a valid concurrency tag in a `Maconomy-Concurrency-Control` header (see Concurrency Tags). The HTTP verb to apply is `POST` and paging parameters can be included in the request body (see Table Paging).

The response to a successful `action:print` request from the card pane record is a data response carrying a `Link` header with a URL as described above. If contracted as described in Partial Data Responses, this data response may be a partial one.

This example shows how to acquire the expense sheet currently in focus as a PDF file by submitting a printing request from its header:

```
$ curl -i
       -H 'Authorization: X-Reconnect OTg1MDE1MD...JMTAJMTYwNzYyNjY1NA=='
       -H 'Accept-Language: en-US'
       -H 'Accept: application/vnd.deltek.maconomy.containers+json;  ↩
    charset=utf-8; version=6.0'
       -H 'Maconomy-Concurrency-Control: 8fd63a4a-c8e3-485b-b330- ↩
    dc599e729daa'
       -X POST
       'http://SERVER/maconomy-api/containers/macoprod/expensesheets/ ↩
    instances/7dc0b114-ecf3-4441-940e-cf93dcf58620/data/panes/card/0/print ↩
    '

HTTP/1.1 200 OK
Maconomy-Reconnect: MzA0NjI2Nz...JMTAJMTYwNzYyODc0NQ==
Maconomy-Concurrency-Control: eb6c18e7-bcfb-4144-baa4-e2ab5f763a98
Link: <http://SERVER/maconomy-api/filedrop/macoprod/download ↩
    /5125049317486823446>;rel=file;type=application/pdf
Content-Type: application/vnd.deltek.maconomy.containers+json; charset=utf ↩
    -8; version=6.0

{
  "meta": { ... },
  "panes": {
    "card": { ... },
    "table": { ... }
  },
```

```
  "links": { ... }
}
```

According to the value of the `Link` header displayed here, the expense sheet as a PDF file can be downloaded using the following URL:

```
http://SERVER/maconomy-api/filedrop/macoprod/download/5125049317486823446
```

Now, to initiate printing from a table pane record instead, the client program must follow the `action:print` hyperlink available from the table pane's JSON object within a full data response:

```
{
  "template": "http://SERVER/BASEPATH/containers/SHORTNAME/CONTAINER/ ↩
    instances/INSTANCE/data/panes/table/{row}/print",
  "rel": "action:print"
}
```

Once the `{row}` placeholder has been replaced by the dot index pointing out the table pane record from which the printing should be initiated (see Record Positions), the usage is similar to what was described for the card pane record above.

If an invalid dot index is substituted into the `{row}` placeholder of the `action:print` template URL, the server responds with a `400 Bad Request` response. Otherwise, the response is similar to the one received when printing from a card pane record.

4.5.9 Applying an Application Action

As described in Actions, a varying amount of so-called application actions are defined for the panes of a data container. These actions relate to the specific business logic implemented by the container.

Some application actions may require that one or several *arguments* are provided by the client program (these requirements are not discoverable through the Containers Web Service but must be known from out-of-band means). The client program provides such arguments by means of an `arguments` JSON property in the request body:

```
{
  "arguments": {
    "ARGUMENT_1_NAME": "ARGUMENT_1_VALUE",
    "ARGUMENT_2_NAME": "ARGUMENT_2_VALUE",
    ...
  }
}
```

All argument values must be expressed using the Expression Language also used for filtering restrictions (see the MDML Language Reference [5] for a full description of the

Expression Language). For example, if an application action takes a date as argument and the name of this argument is `day`, then the client program would have to include a JSON object like the following in the body of every request for that action:

```
{
  "arguments": {
    "day": "date(2020,12,31)"
  }
}
```

Some application actions may also consume one or several *file resources*. Such file resources must be provided to the server through appropriate `Maconomy-File-Callback` headers on the request using the following format:

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

In cases where multiple file resources are required, the client program has the choice of either including one `Maconomy-File-Callback` header per resource or including all of the URI references comma-separated within the same header (see section 4.2 in [7]).

For any application action that consumes one or more file resources, it is mandatory that resources are first uploaded to file drops on the server using the File Drop Web Service and then URLs pointing to these created file drops can be passed in `Maconomy-File-Callback` headers on the actual application action request. For example, imagine that a receipt has been uploaded to a file drop identified by the following URL:

```
http://SERVER/maconomy-api/filedrop/macoprod/upload/3404797840542625411
```

Then, if needed, this receipt can be provided to the server by inclusion of the following header on the request:

```
Maconomy-File-Callback: <http://SERVER/maconomy-api/filedrop/upload/ ↩
    macoprod/3404797840542625411>
```

Now, for the card pane record of the data entry currently in focus, the card pane's JSON object within a full data response will present an `action:APP_ACTION` hyperlink for each application action enabled at that point (see Data Resource):

```
{
  "href": "http://SERVER/BASEPATH/containers/SHORTNAME/CONTAINER/instances ↩
    /INSTANCE/data/panes/card/0/action;name=APP_ACTION",
  "rel": "action:APP_ACTION"
}
```

Besides authenticating using a reconnect token valid for the container instance in use (see Maconomy Reconnect Authentication), the client program must include a valid concurrency tag in a `Maconomy-Concurrency-Control` header (see Concurrency Tags) and apply the HTTP verb `POST`. As described above, an `arguments` property carrying the required arguments must be included in a JSON object in the request body and file resources to be consumed must be appropriately described in `Maconomy-File-Callback` headers. Paging parameters may also be included in the request body (see Table Paging).

The response to a successful `action:APP_ACTION` request for the card pane record is a data response that, if contracted as described in Partial Data Responses, may be a partial one. Also, for those of the application actions producing one or several file resources, the response will carry `Link` headers providing pointers to these resources.

For an example of an application action hyperlink, see the `action:submitexpensesheet` hyperlink included in the card pane JSON object within the `data:any-key` response in Loading a Data Entry:

```
{
  "href": "http://SERVER/maconomy-api/containers/macoprod/expensesheets/ ↩
    instances/7dc0b114-ecf3-4441-940e-cf93dcf58620/data/panes/card/0/ ↩
    action;name=submitexpensesheet",
  "rel": "action:submitexpensesheet"
}
```

As the link relation indicates, this hyperlink must be followed in order to submit the expense sheet currently in focus.

Also for the table pane records of the data entry currently in focus, the table pane's JSON object within a full data response presents an `action:APP_ACTION` hyperlink for each application action enabled at that point (see Data Resource):

```
{
  "template": "http://SERVER/BASEPATH/containers/SHORTNAME/CONTAINER/ ↩
    instances/INSTANCE/data/panes/table/{row}/action;name=ACTION",
  "rel": "action:APP_ACTION"
}
```

Once the `{row}` placeholder has been replaced by the dot index pointing out the table pane record to which the application action should be applied (see Record Positions), the usage is similar to what was described for the card pane record above.

If an invalid dot index is substituted into the `{row}` placeholder of the `action:APP_ACTION` template URL, the server responds with a `400 Bad Request` response. Otherwise, the response is similar to one received on a card pane record application action request.

4.5.10 Table Paging

Since table pane records are often presented in pages in user interfaces with limited space, the client program is able to limit the table pane records described in a data response (see Data Resource) to a subrange of the available records. Like with Filter Paging, the client program specifies such a subrange by including appropriate paging parameter JSON properties in the request body. Both for `offset` and `limit`, the default value applied is `0`.

When loading a data entry by either creating a new one (see Creating a Data Entry) or following a hyperlink loading an existing data entry (see Loading a Data Entry), paging parameters pointing out the first page of table pane records are typically passed in the request body. Once the data entry has been loaded, the subsequent pages of table pane records can then be acquired by repeatedly following the `instance:data` hyperlink available from a JSON object representing the container instance (see Container Instances), applying the HTTP verb `POST` and including appropriate paging parameters in the request body. For example, implementing pages of 10 table pane records, the following paging parameters should be passed with the request loading the data entry:

```
{
  "limit": 10,
  "offset": 0
}
```

Then, to acquire the next 10 table pane records, the following paging parameters should be passed with a subsequent `instance:data` request:

```
{
  "limit": 10,
  "offset": 10
}
```

When performing an action on a loaded data entry (see Adding a Table Record, Updating a Record, Deleting a Record, Moving a Table Record, Printing, Applying an Application Action), the range of table pane records described in the data response should not just cover some current table pane records page but rather must cover the entire range of pages acquired by the client program up until then. This in order to enable correct patching of the client program's locally maintained data entry state. Continuing the above example, assuming that the client program has acquired only two pages by now, the following paging parameters should be included in the body of any data entry action request:

```
{
  "limit": 20,
  "offset": 0
}
```

Note that any record position passed by the client program must always be relative to the complete list of table pane records and not to any subrange pointed out by paging parameters.

4.5.11 Partial Data Responses

Where Data Fields Slicing applies to data responses in general, the concept of *partial data responses* helps economize the amount of data sent back in responses to requests potentially modifying the state of the data entry currently loaded. A full data response carries a representation of the current state of the data entry loaded, whereas a partial data response only carries a representation of the state changes that were triggered by the request. With a partial data response at hand, the client program should be able to patch the changes into its locally maintained data entry state.

In order to indicate to the server that a partial data response is preferred, the client program must include the following header on the request:

```
Maconomy-Response-Type: patch
```

If, in fact, the client program does not even care to receive a partial response, including the following request header makes the server send back a `204 No Content` in response to a successful request:

```
Maconomy-Response-Type: none
```

This `none` option may be useful, for example, during imports where the client program is often indifferent to actual state changes but interested in knowing whether each request was successfully completed or not.

A full data response is always sent back by the server, if either the request includes no `Maconomy-Response-Type` header or if the partial response requested by the client program cannot be derived.

A full and a partial data response differ in that the JSON object representing data pane contents in the latter may have a `recordsPatch` property instead of the `records` property and a `linksPatch` property instead of the `links` property.

The purpose of the properties `recordsPatch` and `linksPatch` is to feed the client program with the patches needed to be carried out on the given data pane's records and hyperlinks respectively in order to bring the client program's state up to date. Record patches must always be applied in the order they are received, and it is important to understand that each record patch modifies the client program's state and that the subsequent patch must be applied to the modified state. That is, given the record patches rp_1, rp_2, ..., and rp_n and the client state cs, then apply rp_1 to cs and get cs_1, apply rp_2 to cs_1 and get cs_2, ..., and apply rp_n to cs_{n-1} and get cs_n.

The JSON objects listed in a `recordsPatch` property each represents a data pane record patch and follows one of the four properties patterns described below.

A JSON object representing an *update record patch* has the following properties:

Property	Explanatory text
`operation`	The value `update`, specifying that the record found at the dot index mentioned in `row` must have its fields updated according to the changes described in `record`.
`row`	Dot index of the record to be updated (see Record Positions).
`record`	JSON object with a `data` property holding field name/value pairs describing the record changes to be applied.

A JSON object representing a *delete record patch* has the following properties:

Property	Explanatory text
`operation`	The value `delete`, specifying that first the record found at the dot index mentioned in `row` must be deleted, and then any sibling record residing at a larger dot index must be shifted one up (its dot index is decremented by one).
`row`	Dot index of the record to be deleted (see Record Positions).

A JSON object representing an *insert record patch* has the following properties:

Property	Explanatory text
`operation`	The value `insert`, specifying that first any record found at the dot index mentioned in `row` or at any greater sibling dot index must be shifted one down (its dot index is incremented by one), and then the record for which a representation is found in `record` must be inserted at the dot index mentioned in `row`.
`row`	Dot index at which a record must be inserted (see Record Positions).
`record`	JSON object representing the record to be inserted. The structure of the JSON object is equal to the structure of the JSON objects held in the `records` property of a full data response (see Data Resource).

A JSON object representing a *move record patch* has the following properties:

Property	Explanatory text
`operation`	The value `move`, specifying that the client program must apply what corresponds to a delete record patch followed by an insert record patch. First, the client program must apply what corresponds to a delete record patch where the dot index mentioned in `rowDelete` specifies which record to delete. Keeping the record just deleted, the client program must then apply what corresponds to an insert record patch where the record to insert is the one just deleted and the dot index mentioned in `rowInsert` specifies where to insert the record.
`rowDelete`	Dot index of the record to be deleted (see Record Positions).
`rowInsert`	Dot index at which the record deleted must be inserted (see Record Positions).

Note that a dot index mentioned in some JSON object representing a table pane record patch will always be relative to the complete list of table pane records and not to any sub-range pointed out by paging parameters passed on the request (see Table Paging).

The JSON objects listed in a `linksPatch` property each represents a data pane hyperlink patch and follows one of the two properties patterns described below.

A JSON object representing a *delete hyperlink patch* has the following properties:

Property	Explanatory text
`operation`	The value `delete`, specifying that the hyperlink with the link relation mentioned in `rel` must be deleted.
`rel`	Link relation to which the delete hyperlink patch relates.

A JSON object representing an *insert hyperlink patch* has the following properties:

Property	Explanatory text
`operation`	The value `insert`, specifying that the hyperlink held in `link` must be inserted under the link relation mentioned in `rel`.
`rel`	Link relation to which the insert hyperlink patch relates.
`link`	Hyperlink to be inserted.

Note that if the server wants the client program to replace all the records or hyperlinks of a data pane, these new values are communicated via the usual data pane properties `records` and `links`.

For example, deleting the top most line from the expense sheet currently in focus, requesting a partial data response:

```
$ curl -i
       -H 'Authorization: X-Reconnect M2E4YjcyMD...JMTAJMTYwNzQ3MDUxMw=='
       -H 'Accept-Language: en-US'
       -H 'Accept: application/vnd.deltek.maconomy.containers+json;  ↩
    charset=utf-8; version=6.0'
       -H 'Maconomy-Response-Type: patch'
       -H 'Maconomy-Concurrency-Control: bbb50752-b063-4a26-b01c-44 ↩
    fb728c4ea0'
       -X DELETE
       'http://SERVER/maconomy-api/containers/macoprod/expensesheets/ ↩
    instances/7dc0b114-ecf3-4441-940e-cf93dcf58620/data/panes/table/0'

HTTP/1.1 200 OK
Maconomy-Reconnect: M2E4YjcyMD...JMTAJMTYwNzQ3MTg5MA==
Maconomy-Concurrency-Control: 82391514-59b3-4e79-b66a-7b447a68c0e0
Content-Type: application/vnd.deltek.maconomy.containers+json; charset=utf ↩
    -8; version=6.0

{
  "meta": { ... },
  "panes": {
    "card": {
      "meta": { ... },
      "recordsPatch": [],
      "linksPatch": []
    },
    "table": {
      "meta": { ... },
      "recordsPatch": [
        {
          "operation": "delete",
          "row": 0
        },
        {
          "operation": "update",
          "row": 0,
          "record": {
            "data": {
              "linenumber": 1
            }
          }
        },
```

```
        {
          "operation": "update",
          "row": 1,
          "record": {
            "data": {
              "linenumber": 2
            }
          }
        },
        {
          "operation": "update",
          "row": 2,
          "record": {
            "data": {
              "linenumber": 3
            }
          }
        },
        ...
      ],
      "linksPatch": []
    }
  },
  "links": { ... }
}
```

This partial response tells the client program to first delete the record at index `0`, to then update `linenumber` to `1` on the record now at index `0`, to then update `linenumber` to `2` on the record now at index `1`, to then update `linenumber` to `3` on the record now at index `2`, and so on.

4.6 Warnings and Notifications

The Maconomy server may raise warnings and notifications during execution.

Whereas any notification message is included as a `Maconomy-Notification` header on the response, there is a bit more to warnings.

A Maconomy warning is meant to allow the user to continue or abort an operation, and in traditional Maconomy clients this has been implemented by a synchronous callback where the server sits waiting for a signal to either continue or abort. Besides the potential performance impact of possibly keeping entries locked in the database while waiting for the user's answer, such a protocol is just not naturally implemented in an HTTP-based interface, and therefore a different successive warnings acceptance mechanism has been implemented in the Containers Web Service.

By default, the Containers Web Service automatically accepts any warning from the Ma-

conomy server and includes the messages of the accepted warnings as `Maconomy-Warning` headers on the response. This default behavior, where the operation is continued no matter what warning occurs, is also the behavior obtained when the client program includes the following request header:

```
Maconomy-Warning-Callback: accept
```

If it is desired that any application warning should cause the operation to abort, the following header should be included:

```
Maconomy-Warning-Callback: reject
```

With this request header, the status `422 Unprocessable Entity` is responded if a warning is raised, and the message of the warning is included as a `Maconomy-Warning` response header.

Now, in between the two ends of `accept` and `reject`, we have a successive warnings acceptance variant triggered by the following request header:

```
Maconomy-Warning-Callback: reject-but
```

The `reject-but` variant's function can be described by the following steps:

1. The client program submits a request holding zero or more `Maconomy-Warning` headers.
2. The server starts/continues execution. One of the following two happens next:
    a. The server reaches warning w_i ($i - 1$ being the amount of warnings already reached). If the value of the ith `Maconomy-Warning` header on the request turns out to match the message of the observed warning, execution continues, taking us back to 2. Otherwise, the server rolls back execution and returns a response holding the messages of the warnings w_1, ..., w_i as `Maconomy-Warning` headers. Depending on whether no or some conflicting ith `Maconomy-Warning` header was found on the request, the response carries the status `422 Unprocessable Entity` or `409 Conflict` respectively. If the client program afterwards adopts the responded `Maconomy-Warning` headers and resubmits the request, you are taken back to 1.
    b. Execution completes. Depending on whether there are zero or more unvisited `Maconomy-Warning` headers on the request, the server returns a response carrying a Success status or the status `409 Conflict` respectively. The response includes `Maconomy-Warning` headers representing the warnings accepted during execution. See 2a for further details on the warnings handling.

In essence, when the `reject-but` variant has been chosen, a `422 Unprocessable Entity` response signals that the next warning is ready for the user to accept, whereas

a `409 Conflict` response signals that the client program has provided at least one `Maconomy-Warning` header that is not in sync with how things actually work.

For example, a happy path of successive warnings acceptance deleting a job from the `Jobs` container:

First request, including `reject-but` in a `Maconomy-Warning-Callback` header:

```
$ curl -i
       -H 'Authorization: X-Reconnect YzcyZWUxZm...JMTAJMTYwNzk4MTYzMw==='
       -H 'Accept-Language: en-US'
       -H 'Accept: application/vnd.deltek.maconomy.containers+json;  ↩
    charset=utf-8; version=6.0'
       -H 'Maconomy-Warning-Callback: reject-but'
       -H 'Maconomy-Concurrency-Control: ad139f04-b298-43eb-a902-6 ↩
    dcc1164338b'
       -X DELETE
       'http://SERVER/maconomy-api/containers/macoprod/expensesheets/ ↩
    instances/7dc0b114-ecf3-4441-940e-cf93dcf58620/data/panes/card/0'

HTTP/1.1 422 Unprocessable Entity
Maconomy-Reconnect: YzcyZWUxZm...JMTAJMTYwNzk4MTYzMw==
Maconomy-Warning: "Budgets exist for this job. Delete it anyway?"
Content-Type: application/json; charset=utf-8

{
  "errorMessage": "Budgets exist for this job. Delete it anyway?",
  "errorFamily": "application",
  "errorSeverity": "warning",
  "focus": {
    "fieldName": "jobnumber",
    "paneName": "card"
  }
}
```

Second request, accepting the first warning by including it in a `Maconomy-Warning` header:

```
$ curl -i
       -H 'Authorization: X-Reconnect YzcyZWUxZm...JMTAJMTYwNzk4MTYzMw==='
       -H 'Accept-Language: en-US'
       -H 'Accept: application/vnd.deltek.maconomy.containers+json;  ↩
    charset=utf-8; version=6.0'
       -H 'Maconomy-Warning-Callback: reject-but'
       -H 'Maconomy-Warning: Budgets exist for this job. Delete it anyway?
       -H 'Maconomy-Concurrency-Control: ad139f04-b298-43eb-a902-6 ↩
    dcc1164338b'
       -X DELETE
       'http://SERVER/maconomy-api/containers/macoprod/expensesheets/ ↩
    instances/7dc0b114-ecf3-4441-940e-cf93dcf58620/data/panes/card/0'
```

```
HTTP/1.1 422 Unprocessable Entity
Maconomy-Reconnect: YzcyZWUxZm...JMTAJMTYwNzk4MTYzMw==
Maconomy-Warning: "Budgets exist for this job. Delete it anyway?"
Maconomy-Warning: "The job is referenced from one or more other jobs.  ↩
    These references will be blanked - proceed?"
Content-Type: application/json; charset=utf-8

{
  "errorMessage": "The job is referenced from one or more other jobs.  ↩
    These references will be blanked - proceed?",
  "errorFamily": "application",
  "errorSeverity": "warning",
  "focus": {
    "fieldName": "jobnumber",
    "paneName": "card"
  }
}
```

Third and final request, also accepting the second warning by including it in another `Maconomy-Warning` header:

```
$ curl -i
       -H 'Authorization: X-Reconnect YzcyZWUxZm...JMTAJMTYwNzk4MTYzMw==='
       -H 'Accept-Language: en-US'
       -H 'Accept: application/vnd.deltek.maconomy.containers+json;  ↩
    charset=utf-8; version=6.0'
       -H 'Maconomy-Warning-Callback: reject-but'
       -H 'Maconomy-Warning: Budgets exist for this job. Delete it anyway?
       -H 'Maconomy-Warning: The job is referenced from one or more other  ↩
    jobs. These references will be blanked - proceed?
       -H 'Maconomy-Concurrency-Control: ad139f04-b298-43eb-a902-6 ↩
    dcc1164338b'
       -X DELETE
       'http://SERVER/maconomy-api/containers/macoprod/expensesheets/ ↩
    instances/7dc0b114-ecf3-4441-940e-cf93dcf58620/data/panes/card/0'

HTTP/1.1 204 No Content
Maconomy-Reconnect: YzcyZWUxZm...JMTAJMTYwNzk4MTYzMw==
Maconomy-Concurrency-Control: 13e6cb3f-0a62-4906-a1ff-fad2b2046a2a
Maconomy-Warning: "Budgets exist for this job. Delete it anyway?"
Maconomy-Warning: "The job is referenced from one or more other jobs.  ↩
    These references will be blanked - proceed?"
```

As done in this example, `Maconomy-Warning` headers must be provided by the client program in the exact same order as they were received in the response from the server.

4.6.1 HTML Entity Escaping

Before including a Maconomy warning or notification message in a `Maconomy-Warning` or `Maconomy-Notification` response header respectively, the Containers Web Service HTML entity escapes the message using the following rules:

Character	Decimal value	Escape entity
"	34	`&quot;`
&	38	`&`
'	39	`&apos;`
,	44	`&comma;`
<	60	`<`
>	62	`>`
c	Below 32 or above 126	`&#x`x`;`, x being the hexadecimal value of c
c	None of the above	c

In order to be recognizable to the server, the `Maconomy-Warning` headers included in a request by a client program should be identical to the ones received from the server.

The standards says that an HTTP header value should never contain any non-ASCII characters, and with the HTML entity escaping just described, it is ensured that at least `Maconomy-Warning` and `Maconomy-Notification` HTTP headers will live up to this requirement.

4.7 Web Access Configuration

Just like any Maconomy client, the Containers Web Service is subject to the core access control setup in Maconomy. However, since any of the additional data filtering provided by the screen layouts of other clients is lacking, some containers or record fields normally not exposed may in fact be available through the Containers Web Service.

To address this issue, a REST API-specific access control mechanism has been introduced. This mechanism is based on web access rules specified inside a `webaccess.ini` file located in a `Definitions` folder in the Maconomy server's custom search path (in order for changes to the `webaccess.ini` to take effect, the Coupling Service needs to be restarted):

```
/CustomizationDir/Custom.<shortname>/Definitions/
/CustomizationDir/Custom/Definitions/
```

```
/CustomizationDir/Solution/Definitions/
```

The web access rules specified are appropriately matched against containers and record fields accessed through the Containers Web Service. Accessing a container for which access is not granted according to the web access rules results in a `403 Forbidden` response. For the censored record fields, none of these appear in any of the responses returned by the Containers Web Service.

The following two sections contain further details on how web access rules are specified inside a `webaccess.ini` file.

4.7.1 Access Lists

Web access rules are specified inside a `webaccess.ini` file by use of so-called *access lists*. These access lists come in pairs of an `include` and an `exclude` list and with the following semantics:

- If only an `include` list is specified, access is granted only if the container/field name matches one of the listed patterns.
- If only an `exclude` list is specified, access is granted only if the container/field does not match any of the listed patterns.
- If both an `include` and an `exclude` list have been specified, access is granted only if the container/field name either does not match any of the patterns listed in the `exclude` list or matches one of the patterns in the `include` list. The patterns in the `include` list are thus exemptions from the `exclude` list.

Now, an access list may contain zero or more patterns delimited by a whitespace, and it may be distributed across multiple lines by putting a backslash (`\`) at the end of each line except for the last, for example:

```
<access-list> = <pattern#1> <pattern#2> \
  <pattern#3> \
  <pattern#4> <pattern#5>
```

Each pattern in an access list is one of three kinds:

- A *literal pattern*.
- A *wildcard pattern* consisting of literal string segments separated by wildcards in the form of an asterisk (`*`).
- A *regular expression pattern* surrounded by forward slashes (`/`) and conforming to the Java regex pattern syntax [2].

Note that all access list patterns are case-insensitive.

Since managing long lists of access list patterns can be a challenge, it is possible to create *named lists*. The syntax of a named list is similar to that of access lists described above, for example:

```
<list-name> = <pattern#1> <pattern#2>
```

The contents of a named list can be referenced from some access list by prefixing the name of the list by `$`.

For example:

List declaration	Explanatory text
`my-list = <pattern#1> <pattern#2>`	Declares the named list `my-list`.
`another-list = $my-list <pattern#3>`	Declares another named list `another-list` which includes the patterns of `my-list`.
`data.exclude = $my-list`	Assigns the patterns of `my-list` to the `exclude` list `data.exclude`.
`filter.include = $another-list`	Assigns the patterns of `another-list` to the `include` list `filter.include`.

Container Level

The patterns in container level access lists must match container names formatted as this:

```
<namespace>:<container>
```

Here are a few examples of patterns that can be found in container level access lists:

Pattern	Explanatory text
`maconomy:jobs`	The literal pattern matching the Maconomy `Jobs` container.
`maconomy:*`	The wildcard pattern matching all Maconomy containers.
`/maconomy\:.*/`	The regular expression pattern matching all Maconomy containers.
`*:Jobs`	The wildcard pattern matching the `Jobs` container in any namespace.

Field Level

The patterns of field level access lists must match field names formatted as this:

```
<namespace>:<container>/<pane>.<field>
```

Field level access rules can be specified either for a specific container or across all containers.

Note that since a client program must have access to all key fields of a container, field level access rules must grant access to these.

Here are a few examples of patterns that can be found in field level access lists:

Pattern	Explanatory text
`maconomy:jobs/filter.JobNumber`	The literal pattern matching the field `JobNumber` in the filter pane of the Maconomy `Jobs` container.
`maconomy:jobs/*`	The wildcard pattern matching all fields in all panes of the Maconomy `Jobs` container.
`*/*.nameofuser`	The wildcard pattern matching the field `NameOfUser` in any container pane.

4.7.2 Web Access Contract

The contract around the web access control mechanism is that the Containers Web Service makes any container specification, filtering, or data request subject to the web access rules derivable from the following pairs of container level access lists respectively:

- `specification.include`
- `specification.exclude`

- `filter.include`
- `filter.exclude`

- `data.include`
- `data.exclude`

Also, record fields are censored according to the web access rules derivable from the following pair of field level access lists:

- `field.include`
- `field.exclude`

For example, to disallow filtering access to all `Find_` containers except from the Maconomy `Find_Activity` container, the following `filter` access lists could be included:

```
filter.exclude = maconomy:find_*
filter.include = maconomy:find_activity
```

As another example, inclusion of the following will make the fields `BankAccountNumber` and `BasicSalary` invisible for all containers:

```
field.exclude = */*.bankaccountnumber */*.basicsalary
```

4.7.3 Diagnostic Logging

In order to get diagnostic logging of which access rules are evaluated during Containers Web Service requests, add the following logger to `configuration/logback.xml` on the Coupling Service:

```
<logger name=" com.maconomy.webservices.common.access" additivity="false">
  <level value="DEBUG" />
  <appender-ref ref="FILE" />
</logger>
```

With the above logger, each access rule evaluation results in a line being written to `log/coupling/maconomy.log` with the following structure:

```
DEBUG  c.m.w.c.access.McAccessConfiguration     {Data|Filter|Specification ↩
    |Field} access {granted|denied} to '{name}' by access rules: { ↩
    accessRules}
```

Chapter 5

Popup Types Web Service

The purpose of the Maconomy RESTful Popup Types Web Service is to allow client programs to get hold of the name of the popup container providing the values available for a given enum type (see Containers Web Service for a definition of a popup container). As described in the Fields section, the `subtypeContainer` property of a specification JSON object for an enum type field holds exactly this information, but the values of an enum type may also come in handy in situations where no field of that type is at hand. The Popup Types Web Service facilitates a uniform way of gaining knowledge of the popup container corresponding to a given enum type. Once knowing the popup container name, the enum values can be acquired by following the `data:filter` hyperlink available from the popup container's resource (see Filtering).

This is the custom media type covering the JSON representations within the encompassed version of the Popup Types Web Service (see Media Types):

```
application/vnd.deltek.maconomy.popups+json; charset=utf-8; version=1.1
```

The root resource of the Popup Types Web Service can be accessed by following the hyperlink with link relation `popups` available from an installation resource (see Installation):

```
{
  "href": "http://SERVER/BASEPATH/popups/SHORTNAME",
  "rel": "popups"
}
```

For example, for the `macoprod` system:

```
$ curl -i
       -H 'Accept-Language: en-US'
       -H 'Accept: application/vnd.deltek.maconomy.popups+json; charset= ↩
    utf-8; version=1.1'
```

```
       'http://SERVER/maconomy-api/popups/macoprod'

HTTP/1.1 200 OK
Content-Type: application/vnd.deltek.maconomy.popups+json; charset=utf-8;  ↩
    version=1.1

{
  "links": {
    "popup": {
      "template": "http://SERVER/maconomy-api/popups/macoprod/{popup}",
      "rel": "popup"
    },
    "self": {
      "href": "http://SERVER/maconomy-api/popups/macoprod",
      "rel": "self"
    }
  }
}
```

The hyperlink with link relation `popup` available from the JSON object of a `popups` response is the one to follow in order to acquire knowledge of some enum type:

```
{
  "template": "http://SERVER/BASEPATH/popups/SHORTNAME/{popup}",
  "rel": "popup"
}
```

Besides substituting the name of the enum type of interest into the `{popup}` placeholder of the hyperlink's template URL, the client program must authenticate (see Authentication) and apply the HTTP verb GET.

If an invalid enum type name is substituted into the `popup` template URL, the server responds with a `404 Not Found`. Otherwise, the client program receives a `200 OK` response with a JSON object in the body holding the following properties:

Property	Explanatory text
`meta`	`popupTypeName`: Name of the enum type. `popupContainerName`: Name of the popup container providing the values available for the enum type.
`links`	`self`: Hyperlink referring to the enum type resource itself.

For example, for the currencies enum type:

```
$ curl -i
```

```
       -H 'Accept-Language: en-US'
       -H 'Accept: application/vnd.deltek.maconomy.popups+json; charset= ↩
    utf-8; version=1.1'
       'http://SERVER/maconomy-api/popups/macoprod/currencytype'

HTTP/1.1 200 OK
Content-Type: application/vnd.deltek.maconomy.popups+json; charset=utf-8;  ↩
    version=1.1

{
  "meta": {
    "popupTypeName": "currencytype",
    "popupContainerName": "popup_currencytype"
  },
  "links": {
    "self": {
      "href": "http://SERVER/maconomy-api/popups/macoprod/currencytype",
      "rel": "self"
    }
  }
}
```

This response reveals that the `Popup_CurrencyType` container is the one providing values in the system for the enum type `CurrencyType`.

Chapter 6

File Drop Web Service

The purpose of the Maconomy RESTful File Drop Web Service is to facilitate that a client program can hand over files to the Maconomy server. This may, for example, be relevant for some application actions in the Containers Web Service (see Applying an Application Action).

The crux of the File Drop Web Service is the so-called *file drops*, each being a temporary file store on the Maconomy server to which a single file can be uploaded.

The state space of a file drop has the following two options:

Unresolved which denotes the case where *no file has yet been uploaded* to the file drop.

Resolved which denotes the case where *some file has been uploaded* to the file drop. Note that each file drop can only be resolved once and that a resolved file drop can never go back to being unresolved.

Now, this is the custom media type covering the JSON representations within the encompassed version of the File Drop Web Service (see Media Types):

```
application/vnd.deltek.maconomy.filedrop+json; charset=utf-8; version=1.1
```

The root resource of the File Drop Web Service can be accessed by following the hyperlink with link relation `filedrop` available from an installation resource (see Installation):

```
{
  "href": "http://SERVER/BASEPATH/filedrop/SHORTNAME",
  "rel": "filedrop"
}
```

For example, for the `macoprod` system:

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

```
       -H 'Accept: application/vnd.deltek.maconomy.filedrop+json; charset= ↩
    utf-8; version=1.1'
       'http://SERVER/maconomy-api/filedrop/macoprod'

HTTP/1.1 200 OK
Content-Type: application/vnd.deltek.maconomy.filedrop+json; charset=utf ↩
    -8; version=1.1

{
  "links": {
    "new": {
      "href": "http://SERVER/maconomy-api/filedrop/macoprod/new",
      "rel": "new"
    }
  }
}
```

In general, the JSON representation of the root resource returned in response to a `filedrop` request has a `links` property presenting a `new` hyperlink:

```
{
  "href": "http://SERVER/BASEPATH/filedrop/SHORTNAME/new",
  "rel": "new"
}
```

This `new` hyperlink is the one to follow in order to create a new file drop on the server.

File drop creation requires authentication (see Authentication) and the client program must apply the HTTP verb `POST`. The request body must be kept empty.

For example:

```
$ curl -i
       -u 'Administrator:123456'
       -H 'Accept-Language: en-US'
       -H 'Accept: application/vnd.deltek.maconomy.filedrop+json; charset= ↩
    utf-8; version=1.1'
       -X POST
       'http://SERVER/maconomy-api/filedrop/macoprod/new'

HTTP/1.1 201 Created
Location: http://SERVER/maconomy-api/filedrop/macoprod/upload ↩
    /3404797840542625411
Content-Type: application/vnd.deltek.maconomy.filedrop+json; charset=utf ↩
    -8; version=1.1

{
  "location": "http://SERVER/maconomy-api/filedrop/macoprod/upload ↩
    /3404797840542625411"
```

```
}
```

In general, the response to a successful file drop creation request has status `201 Created` and a JSON object in the body with the following structure:

```
{
  "location": "http://SERVER/BASEPATH/filedrop/SHORTNAME/upload/FILEDROP"
}
```

Here, the `location` property holds a URL identifying the created file drop. This location URL is also included in a `Location` header on the response, as seen in the example above.

When a new file drop has been created, the client program can upload a file to it using either a binary or a `multipart/form-data` data format. In both cases it takes submission of an authenticated (see Authentication) `POST` request towards the file drop's location URL.

If the client program decides to `POST` the *binary data* comprising the contents of some file towards a file drop's location URL, the following two HTTP headers must also be included on the request:

```
Content-Type: application/octet-stream

Content-Disposition: attachment; filename="FILENAME"
```

The first of these headers tells the server that the request body contains unstructured binary data, whereas the second brings the client program's suggestion for a file name to be used by the server when storing the file. For example:

```
$ 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/maconomy-api/filedrop/macoprod/upload ↩
    /3404797840542625411'

HTTP/1.1 204 No Content
```

The `204 No Content` response means that the request was successful and that the file drop is now resolved.

Note that if the client program tries to upload another file to a resolved file drop, then the request fails with a `409 Conflict` response. For example:

```
$ 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/maconomy-api/filedrop/macoprod/upload ↩
    /3404797840542625411'

HTTP/1.1 409 Conflict
Content-Type: application/json; charset=utf-8

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

Since the File Drop Web Service also supports the `multipart/form-data` media type, the client program is able to upload a file to a file drop using a classical *HTML form* [see 9, for technical details] as well. In that case, it is mandatory that the file part of the form is named `file`. For the file drop from the example above, an HTML form for uploading a file to it could have looked something like this:

```
<form action="http://SERVER/maconomy-api/filedrop/macoprod/upload ↩
    /3404797840542625411"
      method="post"
      enctype="multipart/form-data">
  <input type="file" name="file"><!-- name must be "file" -->
  <input type="submit" value="Upload file">
</form>
```

Chapter 7

Logging Web Service

The Maconomy RESTful Logging Web Service provides the client program with the possibility of contributing additional log entries to the Application Performance Monitoring (APM) framework. Such log entries can be used to provide additional insight into the performance of the client program.

This is the custom media type covering the JSON representations within the encompassed version of the Logging Web Service (see Media Types):

```
application/vnd.deltek.maconomy.logging+json; charset=utf-8; version=1.1
```

The root resource of the Logging Web Service can be accessed by following the hyperlink with link relation `logging` available from an installation resource (see Installation):

```
{
  "href": "http://SERVER/BASEPATH/logging/SHORTNAME",
  "rel": "logging"
}
```

For example, for the `macoprod` system:

```
$ curl -i
       -H 'Accept-Language: en-US'
       -H 'Accept: application/vnd.deltek.maconomy.logging+json; charset= ↩
    utf-8; version=1.1'
       'http://SERVER/maconomy-api/logging/macoprod'

HTTP/1.1 200 OK
Content-Type: application/vnd.deltek.maconomy.logging+json; charset=utf-8; ↩
     version=1.1

{
  "enabled": true,
```

```
  "links": {
    "entries": {
      "href": "http://SERVER/maconomy-api/logging/macoprod/entries",
      "rel": "entries"
    }
  }
}
```

In general, the JSON representation of the root resource returned in response to a `logging` request has an `enabled` property indicating whether the APM framework is enabled plus a `links` property presenting an `entries` hyperlink:

```
{
  "href": "http://SERVER/BASEPATH/logging/SHORTNAME/entries",
  "rel": "entries"
}
```

This `entries` hyperlink is the one to follow in order to contribute a log entry to the APM framework.

Log entry contribution requires authentication (see Authentication) and the client program must apply the HTTP verb `POST`. The body of the request must contain the log entry to be contributed as a JSON object with the following properties:

Property	Explanatory text
`name`	Name of the log entry. If the name contains no colons (`:`), the namespace `WebService` will be prepended.
`level`	Log level of the log entry. Possible values are `ERROR`, `WARNING`, `INFO`, `DEBUG`, or `TRACE`.
`duration`	Duration of the log entry (in milliseconds). Mutually exclusive with `elapsed`. Find an explanation of when `duration` is the correct choice below.
`elapsed`	Elapsed time of the log entry (in milliseconds). Mutually exclusive with `duration`. Find an explanation of when `elapsed` is the correct choice below.
`start`	Optional. Start time of the log entry (in Unix Epoch milliseconds). If not specified, a start time will be calculated from the start time of any parent log entry and the duration of already logged sibling log entries (see the `children` property below).
`audit`	Optional. Set it to the boolean value `true` if the log entry should be marked as an audit event, otherwise leave it out or set it to the boolean value `false`.

Property	Explanatory text
`entries`	JSON object with additional custom key-value pairs relevant for the log entry.
`children`	Optional. List of JSON objects with properties as described in this table. Each object represents a so-called child log entry and will be logged with a parent reference pointing to the log entry.

Here, if the time reported in a log entry was spent entirely within the client program (no time was spent interacting with some Maconomy RESTful web service), then it should be specified as a `duration`. Otherwise it should be specified as `elapsed` time.

Also, it is strongly recommended that the client program repeats the `Maconomy-RequestId` header received on the response to the latest web service request to which the contributed log entry relates:

```
Maconomy-RequestId: REQUEST_ID
```

This ensures that the log entry (and its children) is associated with the current interaction flow (see Request Identification in APM Logs).

To control the client name stored with the log entry (and its children), a `Maconomy-Client` request header should be included (see Client Identification in APM Logs):

```
Maconomy-Client: CLIENT_NAME
```

The response to a successful log entry contribution request is a `204 No Content` response.

For example:

```
$ curl -i
       -H 'Maconomy-RequestId: 5fa7f168-5e89-4043-87cb-fe79883d0f57'
       -H 'Accept-Language: en-US'
       -H 'Accept: application/vnd.deltek.maconomy.logging+json; charset= ↩
    utf-8; version=1.1'
       -H 'Content-Type: application/vnd.deltek.maconomy.logging+json;  ↩
    charset=utf-8; version=1.1'
       -d $'{
         "name" : "Test",
         "level" : "INFO",
         "elapsed" : 1234,
         "audit" : true,
         "entries" : {
           "Message" : "Something happened"
         }
```

```
      }'
      'http://SERVER/maconomy-api/logging/macoprod/entries'

HTTP/1.1 204 No Content
```

Chapter 8

User Settings Web Service

The Maconomy RESTful User Settings Web Service offers a simple mechanism for storing user-specific settings on the server. Settings are stored as JSON documents identified by document keys chosen freely by the client program. The server only accepts and produces valid JSON, but the schema is otherwise unconstrained. A user settings document can only be accessed by the Maconomy user who created it.

This is the custom media type covering the JSON representations within the encompassed version of the User Settings Web Service (see Media Types):

```
application/vnd.deltek.maconomy.usersettings+json; charset=utf-8; version ↩
    =1.1
```

The root resource of the User Settings Web Service can be accessed by following the hyperlink with link relation `usersettings` available from an installation resource (see Installation):

```
{
  "href": "http://SERVER/BASEPATH/usersettings/SHORTNAME",
  "rel": "usersettings"
}
```

As the root resource carries user-specific information, accessing it requires authentication (see Authentication).

The JSON object contained in the response to a successful `usersettings` request has a `links` property from where the following kinds of hyperlinks are available:

Link relation	Explanatory text
`user-settings:key-template`	Reference to the action of creating a new user settings document. Find further details below.

Link relation	Explanatory text
`user-settings:key:DOCUMENT_KEY`	Reference to the user settings document identified by the alphanumeric key `DOCUMENT_KEY`. Depending on the amount of settings documents created by the authenticated user, the root resource state may contain zero or more hyperlinks of this kind. Find further details below.

For example, for the `macoprod` system:

```
$ curl -i
       -u 'Administrator:123456'
       -H 'Accept-Language: en-US'
       -H 'Accept: application/vnd.deltek.maconomy.usersettings+json;  ↩
    charset=utf-8; version=1.1'
       'http://SERVER/maconomy-api/usersettings/macoprod'

HTTP/1.1 200 OK
Content-Type: application/vnd.deltek.maconomy.usersettings+json; charset= ↩
    utf-8; version=1.1

{
  "links": {
    "user-settings:key-template": {
      "template": "http://SERVER/maconomy-api/usersettings/macoprod/{ ↩
    document-key}",
      "rel": "user-settings:key-template"
    }
  }
}
```

Since only a `user-settings:key-template` hyperlink is available from this response, apparently, no settings documents have yet been created for the `Administrator` user.

In order to create a new user settings document, the client program must follow the `user-settings:key-template` hyperlink available form the root resource:

```
{
  "template": "http://SERVER/BASEPATH/usersettings/SHORTNAME/{document-key ↩
    }",
  "rel": "user-settings:key-template"
}
```

Here, the `{document-key}` placeholder within the template URL must be replaced by an alphanumeric key identifying the new settings document.

Besides authenticating (see Authentication), the client program must apply the HTTP verb `PUT` and include a valid JSON representation of the new settings document in the request body.

The response to a successful `user-settings:key-template` request is a `204 No Content` response. If the JSON representation in the request body is malformed, a `400 Bad Request` is returned.

For example, adding a `helloworld` settings document:

```
$ curl -i
       -u 'Administrator:123456'
       -H 'Accept: application/vnd.deltek.maconomy.usersettings+json;  ↩
    charset=utf-8; version=1.1'
       -H 'Content-Type: application/vnd.deltek.maconomy.usersettings+json ↩
    ; charset=utf-8; version=1.1'
       -d $'{
         "key": 42,
         "hello": "world"
       }'
       -X PUT
       'http://SERVER/maconomy-api/usersettings/macoprod/helloworld'

HTTP/1.1 204 No Content
```

With a `helloworld` settings document thus added, a hyperlink specifically referencing this document is available from the root resource:

```
$ curl -i
       -u 'Administrator:123456'
       -H 'Accept-Language: en-US'
       -H 'Accept: application/vnd.deltek.maconomy.usersettings+json;  ↩
    charset=utf-8; version=1.1'
       'http://SERVER/maconomy-api/usersettings/macoprod'

HTTP/1.1 200 OK
Content-Type: application/vnd.deltek.maconomy.usersettings+json; charset= ↩
    utf-8; version=1.1

{
  "links": {
    "user-settings:key-template": {
      "template": "http://SERVER/maconomy-api/usersettings/macoprod/{ ↩
    document-key}",
      "rel": "user-settings:key-template"
    },
    "user-settings:key:helloworld": {
      "template": "http://SERVER/maconomy-api/usersettings/macoprod/ ↩
    helloworld",
```

```
      "rel": "user-settings:key:helloworld"
    }
  }
}
```

An existing user settings document can be acquired by submitting an authenticated (see Authentication) `GET` request towards the URL provided with its hyperlink available from the root resource:

```
{
  "href": "http://SERVER/BASEPATH/usersettings/SHORTNAME/DOCUMENT_KEY",
  "rel": "user-settings:key:DOCUMENT_KEY"
}
```

For example, retrieving our `helloworld` settings document:

```
$ curl -i
       -u 'Administrator:123456'
       -H 'Accept: application/vnd.deltek.maconomy.usersettings+json;  ↩
    charset=utf-8; version=1.1'
       'http://SERVER/maconomy-api/usersettings/macoprod/helloworld'

HTTP/1.1 200 OK
Content-Type: application/vnd.deltek.maconomy.usersettings+json; charset= ↩
    utf-8; version=1.1

{
  "key": 42,
  "hello": "world"
}
```

If instead the HTTP verb `PUT` is applied like described for the hyperlink with link relation `user-settings:key-template`, the user settings document is replaced by the JSON object provided in the request body.

If instead the HTTP verb `DELETE` is applied, the user settings document is deleted from the system. For example, deleting our `helloworld` settings document:

```
$ curl -i
       -u 'Administrator:123456'
       -H 'Accept: application/vnd.deltek.maconomy.usersettings+json;  ↩
    charset=utf-8; version=1.1'
       -X DELETE
       'http://SERVER/maconomy-api/usersettings/macoprod/helloworld'

HTTP/1.1 200 OK
    Content-Type: application/vnd.deltek.maconomy.usersettings+json;  ↩
    charset=utf-8; version=1.1
```

Chapter 9

Authentication Web Service

Aside from authenticating towards third party systems, the Maconomy RESTful Authentication Web Service offers the client program to change the role in the current session of the Maconomy authenticated user or create a new session for the user with a certain role. In this document, only the user roles functionality of the web service is covered in details.

This is the custom media type covering the JSON representations within the encompassed version of the Authentication Web Service (see Media Types):

```
application/vnd.deltek.maconomy.authentication+json; charset=utf-8; ↩
    version=1.3
```

The root resource of the Authentication Web Service can be accessed by following the hyperlink with link relation `authentication` available from an installation resource (see Installation):

```
{
  "href": "http://SERVER/BASEPATH/auth/SHORTNAME",
  "rel": "authentication"
}
```

As the root resource carries system-specific information, accessing it requires authentication (see Authentication).

The JSON object contained in the response to a successful `authentication` request has a `links` property from where the following kinds of hyperlinks are available:

Link relation	Explanatory text
`auth:maconomy`	Hyperlink referring to resource exposing information about the Maconomy authenticated user. This is described in further details below.
`auth:THIRD_PARTY_SYSTEM`	Hyperlink referring to the action of authenticating towards the third party system identified by `THIRD_PARTY_SYSTEM`. This is not described any further in this document.
`auth:renew`	Hyperlink referring to the action of renewing some authentication towards a third party system. This is not described any further in this document.
`auth:logout`	Hyperlink referring to the action of logging out of some third party system. This is not described any further in this document.

For example, for the `macoprod` system:

```
$ curl -i
       -u 'Administrator:123456'
       -H 'Accept-Language: en-US'
       -H 'Accept: application/vnd.deltek.maconomy.authentication+json;  ↩
    charset=utf-8; version=1.3'
       'http://SERVER/maconomy-api/auth/macoprod'

HTTP/1.1 200 OK
Content-Type: application/vnd.deltek.maconomy.authentication+json; charset ↩
    =utf-8; version=1.3

{
  "links": {
    "auth:maconomy": {
      "href": "http://SERVER/maconomy-api/auth/macoprod/maconomy",
      "rel": "auth:maconomy"
    },
    "auth:business-objects": {
      "href": "http://SERVER/maconomy-api/auth/macoprod/business-objects",
      "rel": "auth:business-objects"
    },
    "auth:renew": {
      "href": "http://SERVER/maconomy-api/auth/macoprod/renew",
      "rel": "auth:renew"
    },
    "auth:logout": {
      "href": "http://SERVER/maconomy-api/auth/macoprod/logout",
```

```
      "rel": "auth:logout"
    }
  }
}
```

Here, the presence of an `auth:business-objects` hyperlink signals that Business Objects have been enabled for the `macoprod` system and the client program will be able to authenticate towards Business Objects by following this hyperlink. We also see an `auth:maconomy` hyperlink and, contrary to other `auth` hyperlinks, the usage of this will be described in the following.

9.1 Maconomy User Resource

For a Maconomy authenticated user (see Authentication), the role currently assigned to the user's session impacts which data and actions are made available to the user. Acquiring the Maconomy user resource by following the `auth:maconomy` hyperlink available from the root resource of the Authentication Web Service, the client program is presented with information about the currently authenticated user:

```
{
  "href": "http://SERVER/BASEPATH/auth/SHORTNAME/maconomy",
  "rel": "auth:maconomy"
}
```

The client program must of course authenticate (see Authentication) and the HTTP verb to apply is `GET`.

The response to a successful `auth:maconomy` request is called a *Maconomy user response* and carries a JSON object in the body with the following properties:

Property	Explanatory text
`name`	Name of the authenticated user.
`roles`	List of JSON objects each representing a role available for the authenticated user. Find their properties below.
`role`	Instance key of the role currently assigned to the authenticated user's session.
`links`	Hyperlinks available from the Maconomy user resource. Find the list of link relations below.

A JSON object listed in the above mentioned `roles` property represents a role available for the authenticated user and contains the following two properties:

Property	Explanatory text
`name`	Name of the user role.
`key`	Instance key of the user role.

The purposes of the hyperlinks available from the links property of a Maconomy user response are these:

Link relation	Explanatory text
`maconomy:change-role-in-session`	Reference to the action of changing the role of the authenticated user's session, see Change Role in User Session.
`maconomy:new-session-with-role`	Reference to the action of creating a new session for the authenticated user and have some given role assigned, see Create New User Session With Role.
`self`	Reference to the Maconomy user resource itself.

For example:

```
$ curl -i
       -u 'Administrator:123456'
       -H 'Accept-Language: en-US'
       -H 'Accept: application/vnd.deltek.maconomy.authentication+json;  ↩
    charset=utf-8; version=1.3'
       'http://SERVER/maconomy-api/auth/macoprod/maconomy'

HTTP/1.1 200 OK
Content-Type: application/vnd.deltek.maconomy.authentication+json; charset ↩
    =utf-8; version=1.3

{
  "name": "Administrator",
  "roles": [
    {
      "name": "Standard",
      "key": "RoleInstanceKeyAdministratorStandard"
    },
    {
      "name": "Advanced",
      "key": "8074a573-3a1e-4f1d-98f2-bdbed8292f8f"
```

```
    }
  ],
  "role": "RoleInstanceKeyAdministratorStandard",
  "links": {
    "maconomy:change-role-in-session": {
      "template": "http://SERVER/maconomy-api/auth/macoprod/maconomy/ ↩
    change-role-in-session;role={role}",
      "rel": "maconomy:change-role-in-session"
    },
    "maconomy:new-session-with-role": {
      "template": "http://SERVER/maconomy-api/auth/macoprod/maconomy/new- ↩
    session-with-role;role={role}",
      "rel": "maconomy:new-session-with-role"
    },
    "self": {
      "href": "http://SERVER/maconomy-api/auth/macoprod/maconomy",
      "rel": "self"
    }
  }
}
```

With this Maconomy user response, we are informed that the user authenticated is `Administrator` and that the role currently assigned to the authenticated user is the one with instance key `RoleInstanceKeyAdministratorStandard`, that is, the `Standard` role. Aside from the currently assigned `Standard` role, the `Advanced` role is listed as an available role for the `Administrator`.

Just as expected, the JSON delivered within the above Maconomy user response also carries a `links` property exposing a `maconomy:change-role-in-session` and a `maconomy:new-session-with-role` hyperlink. How following these hyperlinks can make the `Advanced` role the one assigned to the authenticated user is described in the next two sub-sections.

9.1.1 Change Role in User Session

By following the `maconomy:change-role-in-session` hyperlink available from the JSON of a Maconomy user response, the client program is able to change the role assigned to the authenticated user's session:

```
{
  "template": "http://SERVER/BASEPATH/auth/SHORTNAME/maconomy/change-role- ↩
    in-session;role={role}",
  "rel": "maconomy:change-role-in-session"
}
```

The `{role}` placeholder within the template URL mentioned here must be replaced by the instance key of the user role to be assigned. As described above, the names and

instance keys of the roles available for the authenticated user are exposed in the `roles` property of the JSON received in the user response.

Once the template URL has been resolved, the client program must authenticate using Maconomy Reconnect Authentication and apply the HTTP verb `POST`.

If either an invalid instance key is substituted into the `{role}` placeholder of the template URL or if an authentication scheme different from a Maconomy reconnect one is applied, the server responds with a `400 Bad Request`. Otherwise, the response to a successful `maconomy:change-role-in-session` request is a Maconomy user response exposing the accomplished role assignment.

The new role will be in play when the reconnect information held in the response to the `maconomy:change-role-in-session` request is applied on a subsequent request.

9.1.2 Create New User Session With Role

By following the `maconomy:new-session-with-role` hyperlink available from the JSON of a Maconomy user response, the client program is able to create a new session for the authenticated user with a certain role assigned without deleting the existing user session:

```
{
  "template": "http://SERVER/BASEPATH/auth/SHORTNAME/maconomy/new-session- ↩
    with-role;role={role}",
  "rel": "maconomy:new-session-with-role"
}
```

The `{role}` placeholder within the template URL mentioned here must be replaced by the instance key of the user role to be assigned in the new session. As described above, the names and instance keys of the roles available for the authenticated user are exposed in the `roles` property of the JSON received in the user response.

Once the template URL has been resolved, the client program must authenticate using Maconomy Reconnect Authentication and apply the HTTP verb `POST`.

If either an invalid instance key is substituted into the `{role}` placeholder of the template URL or if an authentication scheme different from a Maconomy reconnect one is applied, the server responds with a `400 Bad Request`. Otherwise, the response to a successful `maconomy:new-session-with-role` request is a Maconomy user response.

The new user session can be exploited by applying the reconnect information held in the received Maconomy user response on a subsequent request. The previous user session will still be available through the previous reconnect information.

Chapter 10

Messages Web Service

The Maconomy RESTful Messages Web Service offers a way for web-based clients to view and interact with the current set of system messages in the Maconomy system. System messages are server defined messages in text or HTML format that can be used to communicate relevant information to the users such as upcoming maintenance windows or important business-related deadlines.

This is the custom media type covering the JSON representations within the encompassed version of the Messages Web Service (see Media Types):

```
application/vnd.deltek.maconomy.messages+json; charset=utf-8; version=1.0
```

Whenever a system message is returned from the Messages Web Service, its text is returned in the language being the best match between the locale specified in an `Accept-Language` header on the request and the languages in which the message is available. If no such match is found, a default version of the message is returned. It is up to the author of a system message to choose the language of its default version.

In addition to the Messages Web Service, the following header will be present on a response from a Maconomy RESTful web service whenever new system messages appear for the user logged in:

```
Maconomy-Messages-Changed: system=<Comma-separated ids of new messages>
```

The messages with the ids included in such a response header can be subsequently retrieved from the Messages Web Service as described later in this chapter.

Note that system messages are an optional feature. It is therefore always safe for a client program to ignore any `Maconomy-Messages-Changed` response header and never read system messages from the Messages Web Service.

In addition, note that the information about new messages is only kept on the server for a limited period of time. A user returning after a prolonged period of inactivity without

having to log in again might not receive any notifications about system messages that appeared while the user was away. The minimum guaranteed lifetime of these notifications is controlled by the setting `cache.mailbox-lifetime` in `settings.ini`.

Finally, note that on load balanced multi-node systems, it is possible for a user to receive multiple notifications for the same system message. A client program can avoid notifying the user of the same message more than once by only fetching messages in the `new` scope and updating the scope of each message to `visible` or `hidden` as soon as it has been shown to the user. Find further details on how to do this later in this chapter.

Now, the root resource of the Messages Web Service can be accessed by following the hyperlink with link relation `messages` available from an installation resource (see Installation):

```
{
  "href": "http://SERVER/BASEPATH/messages/SHORTNAME",
  "rel": "messages"
}
```

These are the properties present in an acquired JSON representation of such root resource:

Property	Explanatory text
`scopes`	List of possible values for the optional `scope` parameter to be substituted into the template URL of the `system` hyperlink mentioned in the `links` property. Find the `scope` parameter values and their semantics listed below.
`links`	Hyperlinks available from the root resource of the Messages Web Service. Find the list of link relations below.

The supported `scope` parameter values and their semantics are as follows:

Scope	Explanatory text
`all`	Fetch all system messages, regardless of their scope.
`new`	Fetch only new system messages.
`visible`	Fetch all visible system messages, including new ones. When no `scope` parameter is supplied, this is the value applied by default.
`hidden`	Fetch only hidden system messages.

The purposes of the hyperlinks available through the `links` property of the root resource representation are as follows:

Link relation	Explanatory text
`system`	Reference to the system messages resource. Find a description of this below.
`self`	Reference to the root resource of the Messages Web Service itself.

For example, for the `macoprod` system:

```
$ curl -i
       -H 'Accept-Language: en-US'
       -H 'Accept: application/vnd.deltek.maconomy.messages+json; charset= ↩
    utf-8; version=1.0'
       'http://SERVER/maconomy-api/messages/macoprod'

HTTP/1.1 200 OK
Content-Type: application/vnd.deltek.maconomy.messages+json; charset=utf ↩
    -8; version=1.0'

{
  "scopes": [
    "all",
    "new",
    "visible",
    "hidden"
  ],
  "links": {
    "system": {
      "template": "http://SERVER/maconomy-api/messages/macoprod/system{? ↩
    scope}",
      "rel": "system"
    },
    "self": {
      "template": "http://SERVER/maconomy-api/messages/macoprod",
      "rel": "self"
    }
  }
}
```

A hyperlink with link relation `system` is the one to follow in order to get hold of the system messages currently available within some scope:

```
{
  "template": "http://SERVER/BASEPATH/messages/SHORTNAME/system{?scope}",
```

```
  "rel": "system"
}
```

Besides substituting the `{?scope}` placeholder of the hyperlink's template URL in accordance with the semantics of the possible `scope` parameter values described above, the client program must authenticate (see Authentication) and apply the HTTP verb `GET`.

If an invalid `scope` parameter value is substituted into the `{?scope}` placeholder, the server responds with a `400 Bad Request`. Otherwise, the client program receives a `200 OK` response with a JSON object in the body holding the following properties:

Property	Explanatory text
`messages`	List of JSON objects representing the system messages matching the selected scope for the authenticated user. Find the properties of each such message object below.
`links`	Hyperlinks available from the system messages resource. Find the list of link relations below.

These are the properties of a system message JSON object:

Property	Explanatory text
`text`	Message text, possibly in HTML format.
`language`	Language of the message text.
`id`	Id of the message. This can replace the `{id}` placeholder in the template URL of the `message` hyperlink mentioned in the `links` property.
`validFrom`	Date/time in ISO 8601 format from when the message is valid.
`validTo`	Date/time in ISO 8601 format to when the message is valid.
`scope`	Current scope of the message. Either `new`, `visible`, or `hidden`.
`availableLanguages`	List of languages in which the message is available.

The purposes of the hyperlinks available through the `links` property of a system messages resource representation are as follows:

Link relation	Explanatory text
`message`	Reference to a specific system message resource. Find a description of this below.
`self`	Reference to the system messages resource itself.

For example:

```
$ curl -i
       -u 'Administrator:123456'
       -H 'Accept-Language: en-US'
       -H 'Accept: application/vnd.deltek.maconomy.messages+json; charset= ↩
    utf-8; version=1.0'
       'http://SERVER/maconomy-api/messages/macoprod/system?scope=visible'

HTTP/1.1 200 OK
Content-Type: application/vnd.deltek.maconomy.messages+json; charset=utf ↩
    -8; version=1.0

{
  "messages": [
    {
      "text": "This is the english system message",
      "language": "en",
      "id": "ID",
      "validFrom": "1970-01-01T00:00:00Z",
      "validTo": "2030-01-01T00:00:00Z",
      "scope": "new",
      "availableLanguages": [
        "en",
        "da"
      ]
    }
  ],
  "links": {
    "message": {
      "template": "http://SERVER/maconomy-api/messages/macoprod/system/{id ↩
    }",
      "rel": "message"
    },
    "self": {
      "href": "http://SERVER/maconomy-api/messages/macoprod/system?scope= ↩
    visible",
      "rel": "self"
    }
  }
}
```

A hyperlink with link relation `message` is the one to follow in order to interact with a specific system message:

```
{
  "template": "http://SERVER/BASEPATH/messages/SHORTNAME/system/{id}",
  "rel": "message"
}
```

Besides substituting the id of a system message into the `{id}` placeholder of the hyperlink's template URL, the client program must authenticate (see Authentication) and apply the HTTP verb `GET`.

If an invalid message id substituted into the `message` template URL, the server responds with a `404 Not Found`. Otherwise, the client program receives a `200 OK` response with a JSON object in the body holding the following properties:

Property	Explanatory text
`message`	A message JSON object like the one described above representing the system message acquired.
`links`	Hyperlinks available from the system message resource. Find the list of link relations below.

The purposes of the hyperlinks available through the `links` property of a system message resource representation are as follows:

Link relation	Explanatory text
`make-new`	Reference to the action of changing the scope of the system message to `new`. Find a description of this below.
`make-visible`	Reference to the action of changing the scope of the system message to `visible`. Find a description of this below.
`make-hidden`	Reference to the action of changing the scope of the system message to `hidden`. Find a description of this below.
`self`	Reference to the system message resource itself.

For example, to fetch the message identified by `ID`:

```
$ curl -i
       -u 'Administrator:123456'
```

```
       -H 'Accept-Language: en-US'
       -H 'Accept: application/vnd.deltek.maconomy.messages+json; charset= ↩
    utf-8; version=1.0'
       'http://SERVER/maconomy-api/messages/macoprod/system/ID'

HTTP/1.1 200 OK
Content-Type: application/vnd.deltek.maconomy.messages+json; charset=utf ↩
    -8; version=1.0

{
  "message": {
    "text": "This is the english system message",
    "language": "en",
    "id": "ID",
    "validFrom": "1970-01-01T00:00:00Z",
    "validTo": "2030-01-01T00:00:00Z",
    "scope": "new",
    "availableLanguages": [
      "en",
      "da"
    ]
  },
  "links": {
     "make-visible": {
      "href": "http://SERVER/maconomy-api/messages/macoprod/system/ID/make ↩
    -visible",
      "rel": "make-visible"
    },
    "make-hidden": {
      "href": "http://SERVER/maconomy-api/messages/macoprod/system/ID/make ↩
    -hidden",
      "rel": "make-hidden"
    },
    "self": {
      "href": "http://SERVER/maconomy-api/messages/macoprod/system/ID",
      "rel": "self"
    }
  }
}
```

A hyperlink with link relation `make-SCOPE` is the one to follow in order to change the scope of a system message to `SCOPE`:

```
{
    "href": "http://SERVER/BASEPATH/messages/SHORTNAME/system/ID/make- ↩
    SCOPE",
    "rel": "make-SCOPE"
}
```

Besides authenticating (see Authentication), the client program must apply the HTTP verb `POST`.

If the system message turns out to have the scope indicated by the link relation already, the server responds with a `304 Not Modified`. Otherwise, the client program receives a `204 No content` response.

Of course only hyperlinks referring to applicable message scope actions are included in the representation of a system message resource. In the above example with a new system message, only a `make-visible` and a `make-hidden` hyperlink will be available from the `links` property.

To change the scope of the above message identified by `ID` to `visible`:

```
$ curl -i
       -u 'Administrator:123456'
       -H 'Accept-Language: en-US'
       -H 'Accept: application/vnd.deltek.maconomy.messages+json; charset= ↩
    utf-8; version=1.0'
       -X POST
       'http://SERVER/maconomy-api/messages/macoprod/system/ID/make- ↩
    visible'

HTTP/1.1 204 No Content
```

Chapter 11

Diagnostics Web Service

The purpose of the Maconomy RESTful Diagnostics Web Service is to facilitate a way of easily judging whether the system seems to be correctly configured.

This is the custom media type covering the JSON representations within the encompassed version of the Diagnostics Web Service (see Media Types):

```
application/vnd.deltek.maconomy.diagnostics+json; charset=utf-8; version ↩
    =1.0
```

The root resource of the Diagnostics Web Service can be accessed by following the hyperlink with link relation `diagnostics` available from the root resource of the Root Web Service:

```
{
  "href": "http://SERVER/BASEPATH/diagnostics",
  "rel": "diagnostics"
}
```

For example:

```
$ curl -i
       -H 'Accept-Language: en-US'
       -H 'Accept: application/vnd.deltek.maconomy.diagnostics+json;  ↩
    charset=utf-8; version=1.0'
       'http://SERVER/maconomy-api/diagnostics'

HTTP/1.1 200 OK
Content-Type: application/vnd.deltek.maconomy.diagnostics+json; charset= ↩
    utf-8; version=1.0

{
  "links": {
```

```
    "paths": {
      "href": "http://SERVER/maconomy-api/diagnostics/paths",
      "rel": "paths"
    },
    "cookies:server": {
      "href": "http://SERVER/maconomy-api/diagnostics/cookies/server",
      "rel": "cookies:server"
    },
    "timing": {
      "href": "http://SERVER/maconomy-api/diagnostics/timing",
      "rel": "timing"
    },
    "self": {
      "href": "http://SERVER/maconomy-api/diagnostics",
      "rel": "self"
    }
  }
}
```

For further description of the `paths`, the `cookies:server`, and the `timing` hyperlink available from the JSON object of a `diagnostics` response, see the next three sections respectively.

11.1 Paths

The hyperlink with link relation `paths` available from the diagnostics root resource is the one to follow in order to verify whether certain paths are being handled correctly by the reverse proxy:

```
{
  "href": "http://SERVER/BASEPATH/diagnostics/paths",
  "rel": "paths"
}
```

For example:

```
$ curl -i
       -H 'Accept-Language: en-US'
       -H 'Accept: application/vnd.deltek.maconomy.diagnostics+json;  ↩
    charset=utf-8; version=1.0'
       'http://SERVER/maconomy-api/diagnostics/paths'

HTTP/1.1 200 OK
Content-Type: application/vnd.deltek.maconomy.diagnostics+json; charset= ↩
    utf-8; version=1.0

{
```

```
  "links": {
    "paths:path-0": {
      "href": "http://SERVER/maconomy-api/diagnostics/paths/0/slashes/ ↩
    first;matrix=x%2Fy/second",
      "rel": "paths:path-0"
    },
    "self": {
      "href": "http://SERVER/maconomy-api/diagnostics/paths",
      "rel": "self"
    }
  }
}
```

Each hyperlink with link relation `paths:path-INDEX` available from the JSON object of a `paths` response is the one to follow in order to carry out a certain path test:

```
{
  "href": "http://SERVER/BASEPATH/diagnostics/paths/INDEX/PATH",
  "rel": "paths:path-INDEX"
}
```

Which `paths:path-INDEX` hyperlinks are included can be configured via a setting in `settings.ini`, see Test Paths. Note that no restart of the Coupling Service is needed in order for an updated setting value to catch on.

If the path submitted when following some `paths:path-INDEX` hyperlink survives all the way to the server without getting malformed, the server sends back a `204 No Content` response. Otherwise, the client program receives a `400 Bad Request` response.

For example, following the above `paths:path-0` hyperlink:

```
$ curl -i
       -H 'Accept-Language: en-US'
       -H 'Accept: application/vnd.deltek.maconomy.diagnostics+json;  ↩
    charset=utf-8; version=1.0'
       'http://SERVER/maconomy-api/diagnostics/paths/0/slashes/first; ↩
    matrix=x%2Fy/second'

HTTP/1.1 204 No Content
```

In this case, the successful response means that an encoded slash (`%2F`) is being handled correctly by the reverse proxy.

11.2 Cookies

The hyperlink with link relation `cookies:server` available from the diagnostics root resource is the one to follow in order to initiate the verification of whether cookies are transmitted correctly back and forth through the reverse proxy:

```
{
  "href": "http://SERVER/BASEPATH/diagnostics/cookies/server",
  "rel": "cookies:server"
}
```

Besides a `cookies` property listing information about the cookies included on the response in the form of `Set-Cookie` headers, a `links` property in the JSON object carried in the body of a `cookies:server` response presents the hyperlink to be followed next in the cookies test.

For example:

```
$ curl -i
       -H 'Accept-Language: en-US'
       -H 'Accept: application/vnd.deltek.maconomy.diagnostics+json;  ↩
    charset=utf-8; version=1.0'
       'http://SERVER/maconomy-api/diagnostics/cookies/server'

HTTP/1.1 200 OK
Set-Cookie: Maconomy-Diagnostics-Cookie="MjgyOTIzMD...TEwCTE1NTMyNDg5NTI="
Content-Type: application/vnd.deltek.maconomy.diagnostics+json; charset= ↩
    utf-8; version=1.0

{
  "cookies": [
    {
      "name": "Maconomy-Diagnostics-Cookie",
      "value": "MjgyOTIzMD...2CTEwCTE1NTMyNDg5NTI=",
      "maxAge": -1,
      "secure": false,
      "version": 0,
      "httpOnly": false
    }
  ],
  "links": {
    "cookies:client": {
      "href": "http://SERVER/maconomy-api/diagnostics/cookies/client",
      "rel": "cookies:client"
    },
    "self": {
      "href": "http://SERVER/maconomy-api/diagnostics/cookies/server",
      "rel": "self"
    }
  }
}
```

Following the hyperlink with link relation `cookies:client` available from the JSON object of a `cookies:server` response will finish the cookies test:

```
{
  "href": "http://SERVER/BASEPATH/diagnostics/cookies/client",
  "rel": "cookies:client"
}
```

The client program must include `Cookie` headers corresponding to the `Set-Cookie` headers received from the server when carrying out the initial part of the cookies test.

If the expected cookies are received by the server, the server responds with a `204 No Content`. Otherwise, the client program receives a `400 Bad Request` response. In the latter case, the cookie descriptions held in the above `cookies` property can help unravel whether the problem lies in submitting the cookies from or to the server.

For example:

```
$ curl -i
       -H 'Accept-Language: en-US'
       -H 'Accept: application/vnd.deltek.maconomy.diagnostics+json;  ↩
    charset=utf-8; version=1.0'
       -H 'Cookie: Maconomy-Diagnostics-Cookie="MjgyOTIzMD... ↩
    TEwCTE1NTMyNDg5NTI='
       'http://SERVER/maconomy-api/diagnostics/cookies/client'

HTTP/1.1 204 No Content
```

Note that it is possible to configure which `Set-Cookie` headers are included on a `cookies:server` response via a setting in `settings.ini`, see Set-Cookie Values. Note that no restart of the Coupling Service is needed in order for an updated setting value to catch on.

11.3 Timing

The hyperlink with link relation `timing` available from the diagnostics root resource is the one to follow in order to trigger a response time measurement:

```
{
  "href": "http://SERVER/BASEPATH/diagnostics/timing",
  "rel": "timing"
}
```

The server will send back a `204 No Content` response with the following two custom HTTP headers:

- A `Maconomy-Diagnostics-Time-Enter` header carrying an ISO-8601 representation of the point in time at which the request arrived at the Coupling Service.

- A `Maconomy-Diagnostics-Time-Exit` header carrying an ISO-8601 representation of the point in time at which the response left the Coupling Service.

For example:

```
$ curl -i
       -H 'Accept-Language: en-US'
       -H 'Accept: application/vnd.deltek.maconomy.diagnostics+json;  ↩
    charset=utf-8; version=1.0'
       'http://SERVER/maconomy-api/diagnostics/timing'

HTTP/1.1 204 No Content
    Maconomy-Diagnostics-Time-Enter: 2021-10-29T09:04:03.744Z
    Maconomy-Diagnostics-Time-Exit: 2021-10-29T09:04:04.117Z
```

Note that `Maconomy-Diagnostics-Time-Enter` and `Maconomy-Diagnostics-Time-Exit` headers can be requested on any web service response by submitting the following header along with the request:

```
Maconomy-Diagnostics-Time: on
```

Chapter 12

Configuration

In this chapter, the settings relevant for each of the Maconomy RESTful web services covered in this document are described.

Some of these settings are set by MConfig in `server.ini` during installation of the Maconomy system. The settings in `server.ini` should never be changed manually and any additional settings should instead be set in `settings.ini` located in the `settings` folder.

An overall setting set by MConfig is the one specifying the port at which the web services are exposed:

```
web.port
```

By default, the web services are exposed at port `8080`.

MConfig also configures whether the Maconomy web services are in overall enabled or not. This is done using the following setting:

```
web.services.enabled
```

In addition to this common setting, the enablement of each individual web service can be controlled manually in `settings.ini` using settings of the following form:

```
web.services.<web-service>.enabled
```

These enablement settings are described in the below sub-sections.

Whether or not the web services accept secure requests (HTTPS protocol) only can be controlled by setting the following to either `true` or `false`:

```
web.https-only
```

The HTTPS-only option is enabled by default only if proxy encryption has been enabled:

```
web.proxy.encryption
```

Consult the `settings.ini` file for further information.

12.1 Root Web Service Configuration

The Root Web Service is enabled by default. To disable it, the following setting must be set to `false`:

```
web.services.root.enabled
```

12.1.1 Version Information

By default, version information regarding the system installed is part of the representation of the Root Web Service's root resource. To exclude the version information, the following setting must be set to `false`:

```
web.services.root.versions.visible
```

12.1.2 Shortnames

By default, installation shortnames are part of a representation of the Root Web Service's root resource. In order to hide these shortnames, the following setting must be set to `false`:

```
web.services.root.installations.visible
```

12.2 Containers Web Service Configuration

The Containers Web Service is disabled by default. To enable it, the following setting must be set to `true`:

```
web.services.containers.enabled
```

Whenever the Containers Web Service is enabled, the very first version as well as any later version of the service (for example, the version 3 covered in this document) are enabled by default.

The very first version of the Containers Web Service can be disabled by setting the following setting to `false`:

```
web.services.containers.v1.enabled
```

All later versions of the Containers Web Service can be disabled by setting the following setting to `false`:

```
web.services.containers.v2.enabled
```

12.2.1 Container Instances Cache Mode

Container Instances can be cached either in the database, in-memory, or in Redis. This is controlled by assigning the value `database`, `memory`, or `redis` to the following setting:

```
web.services.containers.instances.cache.mode
```

The default cache mode is `database` and this should *only* be changed after advice from Maconomy Development. Further details on the configuration of each of the three cache modes can be found in the three sub-sections below.

Whether the bytes representing a container instance are compressed before they are put into the container instances cache can be controlled by the following setting:

```
web.services.maconomy-containers.instances.cache.compression.enabled
```

Compression is enabled by default and this should *only* be changed after advice from Maconomy Development.

Database

If `database` is the cache mode selected, two cleaning settings become relevant. The first of these must be set if another privileged user than the one used by the batch framework should be used when cleaning out expired container instances from the database cache:

```
web.services.containers.instances.cache.database.cleaner.login.name
```

The other setting must be set if the seconds between cleanings of the database instances cache should be different from `60`:

```
web.services.containers.instances.cache.database.cleaner.interval.secs
```

In-memory

If `memory` is the cache mode selected, container instances are cached in-memory, making it very important that requests from a particular client always arrive at the same server node. When multiple server nodes exist, a load balancer configured to use sticky sessions is required.

For the in-memory container instances cache mode, the concurrency level of the underlying in-memory cache implementation can be controlled by the following setting:

```
web.services.containers.instances.cache.memory.concurrency-level
```

The default value `none` as well as any integer less than zero mean that the default value of the underlying in-memory cache implementation is used. This setting should *only* be changed after advice from Maconomy Development.

Also, for an in-memory container instances cache, the following eight metrics (six histograms and two timers) are maintained:

Identifier	Explanatory text
`sessions-count`	A *histogram* describing the distribution of the amount of user sessions.
`instances-per-session-count`	A *histogram* describing the distribution of the amount of container instances registered within each user session in the cache.
`instances-total-count`	A *histogram* describing the distribution of the total amount of container instances in the cache.
`bytes-per-instance-count`	A *histogram* describing the distribution of the amount of bytes taken up by each container instance in the cache.
`bytes-per-session-count`	A *histogram* describing the distribution of the amount of bytes taken up by each user session in the cache.
`bytes-total-count`	A *histogram* describing the distribution of the total amount of bytes taken up by the cache.
`instance-read-time`	A *timer* tracking how many seconds it took to read some container instance from the cache.
`instance-write-time`	A *timer* tracking how many seconds it took to write some container instance to the cache.

The data for the metric with identifier *xxx* can be found on the server in a CSV file

named `McMemoryInstancesCache.`*xxx*`.csv`.

The more frequent the histograms are updated, the more accurate statistics they provide. However, increasing the frequency of histogram updates also increases the base CPU load. The amount of seconds between updates of the histograms can be controlled by the following setting:

```
web.services.containers.instances.cache.memory.metrics.interval.secs
```

If you set the update interval to `-1`, no histogram updates are done at all. The default update interval is `1` second.

Redis

If `redis` is the cache mode selected, container instances are cached in Redis [3]. The Redis URI used will be `redis://localhost:6379` unless another is stated using the following setting:

```
web.services.containers.instances.cache.redis.uri
```

12.2.2 Container Instance Expiry

By default, a container instance expires when it has lied untouched in the cache for more than `10` minutes. If a different expiry is desired, the following setting must be set:

```
web.services.containers.instances.cache.instance-expiry.minutes
```

12.2.3 Container Instances Limit

By default, `25` container instances are allowed per user session. If a different limit is desired, the following setting must be set:

```
web.services.containers.instances.cache.max-instances.per-session
```

Whenever the limit is exceeded, the least recently used container instance registered for the given user session will be removed.

If the setting is set to `none` or some negative integer, there is no limit on the amount of container instances per user session.

12.2.4 Auto Position Fields

By default, any auto position field which is not a key field is left out of table pane specifications and table pane records. If no auto position fields should be left out, the following setting must be set to `false`:

```
web.services.containers.filter-out.auto-position-fields.enabled
```

This setting affects the efficiency of Partial Data Responses and should *only* be changed after advice from Maconomy Development.

12.3 Popup Types Web Service Configuration

The Popup Types Web Service is disabled by default. To enable it, the following setting must be set to `true`:

```
web.services.popups.enabled
```

12.4 File Drop Web Service Configuration

The File Drop Web Service is disabled by default. To enable it, the following setting must be set to `true`:

```
web.services.filedrop.enabled
```

If the File Drop Web Service is not enabled, the client program will not be able to hand over files to the Maconomy Server in connection with execution of application actions in the Containers Web Service (see Applying an Application Action).

12.5 Logging Web Service Configuration

The Logging Web Service is disabled by default. To enable it, the following setting must be set to `true`:

```
web.services.logging.enabled
```

12.6 User Settings Web Service Configuration

The User Settings Web Service is disabled by default. To enable it, the following setting must be set to `true`:

```
web.services.user-settings.enabled
```

12.7 Authentication Web Service Configuration

The Authentication Web Service is disabled by default. To enable it, the following setting must be set to `true`:

```
web.services.authentication.enabled
```

12.8 Messages Web Service Configuration

The Messages Web Service is disabled by default. To enable it, the following setting must be set to `true`:

```
web.services.messages.enabled
```

12.9 Diagnostics Web Service Configuration

The Diagnostics Web Service is disabled by default. To enable it, the following setting must be set to `true`:

```
web.services.diagnostics.enabled
```

12.9.1 Test Paths

The `paths:path-INDEX` hyperlinks described in Paths reflects the comma separated paths assigned to the following setting:

```
    web.services.diagnostics.paths
```

Its default value is `slashes/first;matrix=x%2Fy/second`, leading to the exposure of just a single `paths:path-INDEX` hyperlink:

```
{
  "href": "http://SERVER/BASEPATH/diagnostics/paths/0/slashes/first;matrix ↩
    =x%2Fy/second",
  "rel": "paths:path-0"
}
```

The purpose of this hyperlink is to allow verification of an encoded slash (`%2F`) being handled correctly by the reverse proxy.

12.9.2 `Set-Cookie` Values

A response received by the client program when following the `cookies:server` hyperlink described in Cookies, by default, includes exactly one `Set-Cookie` header which has been known to cause problems in the past. This can be overridden by assigning comma-separated `Set-Cookie` values to the following setting:

```
web.services.diagnostics.cookies
```

Chapter 13

Versions

For each Maconomy RESTful web service encompassed in this document, this chapter contains a section providing short descriptions of the different service versions.

13.1 Root Web Service Versions

13.1.1 Root Web Service Version 1

Version 1.0 covers the first version of the Root Web Service which was introduced in Maconomy 2.5.2.

Note that version 1 is deprecated from Maconomy 2.5.4 and will no longer be accessible from Maconomy 2.7.

13.1.2 Root Web Service Version 2

From version 2.0, which was introduced in Maconomy 2.5.4, the version information exposed as part of a root resource representation has been made similar for APU and TPU and thus an application build number is also included.

From version 2.1, which was introduced in Maconomy 2.6, an `errorIds` property is introduced in Error Responses.

13.2 Containers Web Service Versions

13.2.1 Containers Web Service Version 1

Version 1.0 covers the first version of the Containers Web Service which was introduced in Maconomy 2.1.3 With the interaction model used in version 1, the server has to do a large amount of recalculations for each interaction and hence later versions are recommended.

Note that version 1 is deprecated from Maconomy 2.5.2 and will no longer be accessible from Maconomy 2.7.

13.2.2 Containers Web Service Version 2

From version 2.0, which was introduced in Maconomy 2.5.2, a completely different interaction model is being used. Interaction with container data using version 2 is done through so-called Container Instances holding important parts of the container's state and this eliminates the need for the large amount of recalculations necessary in version 1. Version 2 has a greatly improved performance over version 1 and in fact performs on par with other APIs used by Maconomy clients. iAccess for Maconomy 2.5.2 uses version 2 and sees significant performance improvements.

Note that version 2 is deprecated from Maconomy 2.5.3 and will no longer be accessible from Maconomy 2.7.

13.2.3 Containers Web Service Version 3

From version 3.0, which was introduced in Maconomy 2.5.3, data containers having tree table panes with hierarchically organized records are fully supported. This means that JSON objects received as representations of table pane records reflect any hierarchical structure and Record Positions are pointed out by so-called dot indices. A move record patch now occurs in Partial Data Responses returned for move requests.

Note that version 3 is deprecated from Maconomy 2.5.4 and will no longer be accessible from Maconomy 2.7.

13.2.4 Containers Web Service Version 4

From version 4.0, which was introduced in Maconomy 2.5.4, Filtering parameters may be supplied as properties of a JSON object in a `POST` request body instead of as query parameters. Also, when the client program does not supply any `fields` parameter, only key fields (instead of all fields) are included in the filter response.

Also from version 4.0, the `moveMode` property exposed as part of a table pane's specification allows a client program to discover in advance if records may only be moved around inside their current context. Furthermore, the targeted title properties `upTitle`, `downTitle`, `indentTitle`, and `outdentTitle` in `action:move` specifications facilitate more accurate action descriptions in user interfaces.

From version 4.1, which was introduced in Maconomy 2.6, an `errorIds` property is introduced in Error Responses.

Note that version 4 is deprecated from Maconomy 2.6 and will no longer be accessible from Maconomy 2.7.

13.2.5 Containers Web Service Version 5

From version 5.0, which was introduced in Maconomy 2.6, an `access` hyperlink is included in container resources, allowing the client program to retrieve information about the authenticated user's CRUD access rights.

Also from version 5.0, only key fields can be mentioned in update requests and only key fields are included in data responses, unless otherwise specified in the Data Fields Slicing JSON object submitted on container instance creation.

Also from version 5.0, a `data:some-key` hyperlink is included in container instance resources and can be followed in cases where the client program wants to load the data entry corresponding to some already known key (see Loading a Data Entry).

Note that version 5 is deprecated from Maconomy 2.6.1 and will no longer be accessible from Maconomy 2.7.

13.2.6 Containers Web Service Version 6

From version 6.0, which was introduced in Maconomy 2.6.1, the Filtering has changed for popup containers. Aside from the `value`, `ordinal`, and `title` field previously available for popup containers, a `hidden` field has been made available, providing the client with information about whether an enum value should be selectable in a user interface or not. Also, like for other containers, the filtering parameters `fields`, `restriction`, and `orderBy` are now supported for popup containers, and if no `fields` parameter is supplied in a popup container filter request, only the `value` field will be included in the response.

Also from version 6.0, Table Paging is available for data responses and the client program is able to control the order of table pane records by configuring multi-column Table Sorting for container instances. As a consequence of query parameters being eradicated and request body parameters being used instead, the following stand:

- Only the HTTP verb `POST` is applicable when following a `data:filter` (or a `data:enumvalues`) hyperlink (see Filtering).
- Instead of `DELETE`, the HTTP verb `POST` must be applied when following a `data:delete` hyperlink (see Deleting a Record).
- The HTTP verb `POST` may also be applied when following an `instance:data` hyperlink (see Container Instances).

13.3 Popup Types Web Service Versions

13.3.1 Popup Types Web Service Version 1

Version 1.0 covers the first version of the Popup Types Web Service which was introduced in Maconomy 2.5.2.

From version 1.1, which was introduced in Maconomy 2.6, an `errorIds` property is introduced in Error Responses.

13.4 File Drop Web Service Versions

13.4.1 File Drop Web Service Version 1

Version 1.0 covers the first version of the File Drop Web Service which was introduced in Maconomy 2.1.3.

From version 1.1, which was introduced in Maconomy 2.6, an `errorIds` property is introduced in Error Responses.

13.5 Logging Web Service Versions

13.5.1 Logging Web Service Version 1

Version 1.0 covers the first version of the Logging Web Service which was introduced in Maconomy 2.5.1.

From version 1.1, which was introduced in Maconomy 2.6, an `errorIds` property is introduced in Error Responses.

13.6 User Settings Web Service Versions

13.6.1 User Settings Web Service Version 1

Version 1.0 covers the first version of the User Settings Web Service which was introduced in Maconomy 2.1.3.

From version 1.1, which was introduced in Maconomy 2.6, an `errorIds` property is introduced in Error Responses.

13.7 Authentication Web Service Versions

13.7.1 Authentication Web Service Version 1

Version 1.0 covers the first version of the Authentication Web Service which was introduced in Maconomy 2.1.3.

From version 1.1, which was introduced in Maconomy 2.6, a `401 Unauthorized` is only returned when the Maconomy authentication cannot be carried through. When something regarding a third party authentication goes wrong, a `500 Server Error` is returned instead.

From version 1.2, which was also introduced in Maconomy 2.6, an `errorIds` property is introduced in Error Responses.

From version 1.3, which was introduced in Maconomy 2.6.1, an `auth:maconomy` hyperlink is available from the root resource, enabling the client program to administer the role assigned to the authenticated user.

13.8 Messages Web Service Versions

13.8.1 Messages Web Service Version 1

Version 1.0 covers the first version of the Messages Web Service which was introduced in Maconomy 2.6.

13.9 Diagnostics Web Service Versions

13.9.1 Diagnostics Web Service Version 1

Version 1.0 covers the first version of the Diagnostics Web Service which was introduced in Maconomy 2.6.

Bibliography

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

[2] Regular expressions in JDK 8. URL https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html.

[3] Redis. URL https://redis.io/.

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

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

[6] 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.

[7] 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.

[8] 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.

[9] L. Masinter. Returning Values from Forms: multipart/form-data. RFC 2388 (Proposed Standard), August 1998. URL https://tools.ietf.org/rfc/rfc2388.txt.

[10] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, and C. Mortimore. OpenID Connect Core 1.0 incorporating errata set 1, 2014. URL http://openid.net/specs/openid-connect-core-1_0.html.

[11] 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.

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

Index