Deltek

Deltek People Planner

RESTful API Developer Guide

July 1, 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 publication contains proprietary information that is protected by copyright. All rights are reserved. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, or translated into another language, without the prior written consent of Deltek, Inc.

This edition published July 2022.

© Deltek, Inc.

Deltek's software is also protected by copyright law and constitutes valuable confidential and proprietary information of Deltek, Inc. and its licensors. The Deltek software, and all related documentation, is provided for use only in accordance with the terms of the license agreement. Unauthorized reproduction or distribution of the program or any portion thereof could result in severe civil or criminal penalties.

All trademarks are the property of their respective owners.

Contents

Introduction

The People Planner RESTful Web Service API (the REST API or the API) is a programmatic interface that provides access to data and business functionality in the Deltek People Planner product.

REST

REST is a specific design of a web service and stands for REpresentational State Transfer. A web service that is built on REST principles is said to be RESTful.

It is useful to know a little about what REST is, and the concepts and terminology associated with it. Two important concepts for a RESTful web service are entities and hyperlinks.

Naming Conventions

A *resource* is a central concept in a RESTful web service. However, there already exists a concept in the People Planner domain with the same name. Therefore, the term *entity* is often used in this guide. The meaning of the entity term is identical to that of a resource, only the word is different.

Entities

An entity is a domain object that is uniquely identified by a URL. For example, each booking in a People Planner system has a unique URL. When you access the URL for an entity, you get a representation of the current state of that entity. For a booking, this representation contains properties such as start date, finish date, and value.

Entities are manipulated (read, updated, deleted, and so on) by a fixed set of HTTP verbs. The verbs used in the People Planner REST API are GET, POST, and DELETE.

Each entity has a *landing page* with general information related to that entity, such as metadata and hypermedia links.

Logging

Base configuration can be found in the `appSettings.json` file. Normally, you do not need to make changes to this file, but should instead use the `appSettings.Production.json` file.

Configuring logging in the People Planner REST API consists of two parts: configuring what the application logs and configuring the PP file logging provider.

Configuring *what* the application logs is specified in the `Root→Logging→PPFileLogger` section. This follows standard .NET (Core and later) logging syntax[1].

Configuration of the PP file logging provider is specified in the `Root→PPFileLogger` section. This section controls what is actually writing to the log file and the location of the log file.

Enabling Verbose Logging

To enable verbose logging, set `Root→Logging→PPFileLogger→LogLevel` to `Debug` and `Root→PPFileLogger→LogSourceLevel` to `Verbose`.

---

[1] https://docs.microsoft.com/en-us/aspnet/core/fundamentals/logging/#configure-logging-1

The `appSettings.Production.json` file should look something like this:

```json
{
  "Logging": {
    "PPFileLogger": {
      "LogLevel": {
        "Default": "Debug"
      }
    },
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Error"
    }
  },
  "PPFileLogger": {
    "LogSourceLevel": "Verbose",
    "Filename": "..\\logs\\REST.log"
  },
  "AppConfiguration": {
    "DataConnectionFileName": "..\\DataConnection.xml",
    "IgnoreETags": false
  }
}
```

Reference Documentation

Refer to the section on [Swagger Documentation](#).

Further Reading

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

Basics

Authentication

The People Planner RESTful API only supports bearer token authentication, originally created as part of the OAUTH2 described in RFC6750[2]. The token representation used is JSON Web Token (JWT) as described in RFC7519[3].

The JWT token must be sent by the client in, and only in, the `Authorization` HTTP request header using the `Bearer` authentication scheme, described in RFC2617[4].

Token validation configuration, such as *issuer* and *audience*, can be found and configured in the REST API `appSettings.json` configuration file. The token must be signed using HMAC SHA-256 with a private key using the `Silent Sign In (SSI) Maconomy Secret Key` from the People Planner settings, which must be padded (by repeating it) to 256 bits in length. The token must match the configured *issuer* and *audience*. The token must have a subject claim (`sub`) with a People Planner network username and network domain name in the form `networkusername@networkdomainname`.

When integrating with Deltek Maconomy, the Maconomy RESTful API can issue a People Planner JWT token. The URL for the People Planner JWT authorization endpoint can be found by navigating to the `auth` endpoint for the desired Maconomy shortname and following the `auth:people-planner-jwt` link relation.

OAUTH2, OIDC, and other authorization protocols and flows are currently *not* supported.

Media types and API versions

The REST API uses both standard and custom media types in the responses to provide the client with information on the data representation in the payload. The API only uses JSON serialization, so all content types reflect that. The response content types for the different endpoints in the API can be found in the Swagger . The `Content-Type` HTTP response header also contains the API version number of the representation returned.

The API supports multiple versions of endpoints and actions. A client can request a specific version of an endpoint by providing the version number in the Accept HTTP request header, for example:

```
Accept: application/json; v=1.0
```

Endpoints that use versioning also reply with an `api-supported-versions` custom HTTP response header, listing the supported versions for the endpoint. If you request an unsupported or incorrect version value, the API returns an error.

> **Note:** Currently, it is not mandatory to send a request version, but that will change in later versions of the API—always send a request version.

For more details about the supported media types and versions, please refer to the Swagger that can be found at http://<server>:<port>/RestApi/api-doc/index.html.

---

[2] https://datatracker.ietf.org/doc/html/rfc6750

[3] https://datatracker.ietf.org/doc/html/rfc7519

[4] https://datatracker.ietf.org/doc/html/rfc2617

Languages

The REST API supports a number of different languages, used to provide localized phrases for actions, entity fields, and so on.

You can specify the preferred language using the `Accept-Language` HTTP request header, for example:

`Accept-Language: da-DK`

At the moment, there is no programmatic way of getting a list of supported languages from the REST API. This will be included in a future versions.

OpenAPI Specification

The OpenAPI Specification[5] (OAS) is an open standard, technical specification that describes a REST API. The OAS is maintained by the OpenAPI Initiative and they describe OAS like this:

The OAS defines a standard, programming language-agnostic interface description for REST APIs, which allows both humans and computers to discover and understand the capabilities of a service without requiring access to source code, additional documentation, or inspection of network traffic. When properly defined via OAS, a consumer can understand and interact with the remote service with a minimal amount of implementation logic. Similar to what interface descriptions have done for lower-level programming, the OAS removes guesswork in calling a service.

In the People Planner REST API, the OAS is automatically generated based on what capabilities exists and are available at run-time. For more information, please refer to the following online resources: https://oai.github.io/Documentation and https://openapi.tools.

> **Note:** One limitation in the current version of the People Planner REST API is that, we do not have hypermedia navigation links in the specification (HATEOAS, see https://restfulapi.net/hateoas).

Swagger Documentation

The People Planner REST API uses the OpenAPI specification to generate a run-time web user interface using Swagger UI.

This user interface can be used to read the documentation and explore the different endpoints and data types exposed by the API. It also serves as a basic interactive tool for interacting with the API.

The Swagger documentation URL is: http://<server>:<port>/restapi/api-doc/index.html.

Entity Specification

Every entity in the People Planner RESTful web service interface has a *specification* resource endpoint.

> **Note:** This specification is similar to what is used in the Maconomy RESTful API. More information can be found in the specifications section of the *Deltek Maconomy Web Services Programmers Guide*.

---

[5] https://www.openapis.org

The specification is used to programmatically determine the following

1. The names, titles, and data types of the fields exposed by the entity

2. The names and titles of the actions supported by the entity

To correctly interpret and manipulate records of a specific entity, a client program must read the specification resource to obtain the field names and their data types.

An example of an entity specification link:

```
"specification": {

    "href": "http://server/api/bookings/time/specification",

    "rel": "specification"

}
```

By looking at the `rel` property and discovering that the relation "specification" is present, a client program can determine that this particular hyperlink points to the specification resource of that entity. The link relation is simply an identifier that tells client programs about the meaning of a particular hyperlink. When writing client applications, you should only rely on the link relations and consider the links as opaque.

> **Warning:** You should not attempt to guess the pattern for particular kinds of entities because only the link relation is guaranteed to be stable between People Planner releases.

If you follow the specification link, you acquire the specification for the entity.

```
{

    "entityName": "timebookings",

    "actions": {...},

    "fields": {...},

    "foreignKeys": {...}

}
```

The preceding example shows the high-level structure of the specification resource for the timebookings entity. If you try this request yourself, you will see that the full response is substantially larger. The omitted parts are discussed in the following sections.

This JSON object holds the specification for the entity named `timebookings` and it contains *actions*, *fields* and *foreign keys* for the entity.

Actions

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

```
"actions": {

    "action:init": {

        "title": "Init timebookings",

        "rel": "action:init"

    },

    "action:read": {

        "title": "Read timebookings",

        "rel": "action:read"

    },
```

```
    "action:create": {
          "title": "Create new timebookings",
          "rel": "action:create"
    },
    "action:delete": {
          "title": "Delete timebookings",
          "rel": "action:delete"
    },
    "action:update": {
          "title": "Update timebookings",
          "rel": "action:update"
    }
}
```

Each action is represented by an object that contains a `rel` property. The value of this property uniquely identifies the action within the entity. The object also contains a language specific title property, appropriate for displaying in a user interface.

The list of actions found in a specification is the full list of actions. When interacting with a particular entity, the client program can determine if an action can be invoked, in the current state of the entity, by examining whether a hyperlink with the link relation corresponding to the action's `rel` property is present. For example, if the currently authenticated user does not have the required privileges for deleting a specific entity, the action:delete link relation is not present in the actions list.

Fields

The following is an example of some of the contents of a `fields` property:

```
"fields": {
     "subject": {
          "name": "subject",
          "title": "Subject",
          "type": "string",
          "mandatory": false,
          "key": false,
          "update": true,
          "create": true,
          "maxLength": 200,
          "multiLine": false,
          "references": []
     },
     "start": {
          "name": "start",
          "title": "Start",
          "type": "datetime",
          "mandatory": true,
```

```
            "key": false,
            "update": true,
            "create": true,
            "references": []
    },
    "resourceid": {
            "name": "resourceid",
            "title": "Resource ID",
            "type": "guid",
            "mandatory": false,
            "key": false,
            "update": false,
            "create": true,
            "references": [
                    "resourceid_resource"
            ]
    },
    ...
}
```

These are examples of field objects that contain metadata for the *subject*, *start,* and *resourceid* fields.

The field objects describe all the fields that are present in a record of an entity. An important property of a field object is its *type*. The type determines how client programs must interpret and represent values for that field when interacting with an entity. The specifics of each format are detailed in Data Types.

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

Property	Description
name	[string] The identifier used to refer to the field in representations. This is intended for use by the software, and is normally not visible in a user interface.
title	[string] The human-readable language specific name for the field. The title is an appropriate, localized label for the field in a user interface.
type	[string] The data type of the field. The data type is one of the types described in the section Data Types, where there is also a description of each type.
enumType	[string] This property is defined for fields that have the enum data type and it contains the name of the enumeration type.
mandatory	[bool] Indicates whether the field is mandatory. Mandatory fields are always included in entity representations sent by the server, even if the field is not part of a filtered request.
key	[bool] Indicates whether the field is a primary key field.

Property	Description
update	[bool] Indicates whether the field can be updated after the record is created. Some fields are immutable after the record is created in the system.
create	[bool] Indicates whether the field is editable when a record is created. If this value is false, the service ignores any value provided by client program. Thus, the field need not be part of entity representation on create action.
maxLength	[bool] Indicates the maximum length of a string field.
multiLine	[bool] Indicates if a string field is considered multi-line. This is useful when generating user interfaces.
references	[string array] Indicates which foreign keys this field participates in.

Foreign Keys

The following is an example of the `foreignKeys` property:

```
"foreignKeys": {
    "assignmentid_assignment": {
        "name": "assignmentid_assignment",
        "title": "Assignment",
        "incomplete": false,
        "rel": "data:key:assignmentid_assignment",
        "entity": "assignments",
        "fieldReferences": [
            {
                "field": "assignmentid",
                "foreignField": "id",
                "supplement": false
            },
            {
                "field": "assignmenttext",
                "foreignField": "assignmenttext",
                "supplement": true
            }
        ],
        "links": {}
    },
    "resourceid_resource": {
        "name": "resourceid_resource",
        "title": "Resource",
        "incomplete": false,
```

```
            "rel": "data:key:resourceid_resource",
            "entity": "resources",
            "fieldReferences": [
                {
                    "field": "resourceid",
                    "foreignField": "id",
                    "supplement": false
                },
                {
                    "field": "resourcename",
                    "foreignField": "name",
                    "supplement": true
                }
            ],
            "links": {
                "data:search": {
                    "href":
"http://server/restapi/api/bookings/time/search?foreignkey=resourceid_resourc
e",
                    "rel": "data:search"
                }
            }
        },
        ...
}
```

Each foreign key describes an association between entities in the system. In this example, the *foreign key* `resourceid_resource` is a reference between a timebooking and a People Planner resource (planning resource). Foreign keys are used to search for a value for one or more fields and for navigating between related entities.

A foreign key has a number of field references, for example the field `resourceid` on the *timebookings* entity references to the field `id` on the *resource* entity. If a field reference is marked as `supplement` it does not directly participate in the foreign key relationship, but is included as a signal to a client program to assign the value back during a foreign key search. In this example, the client program should assign the `name` from the *resource* back to the `resourcename` on the booking entity when performing a search for the `resourceid_resource` foreign key relation.

The link with the link relation `data:search` is used to perform the foreign key search.

A foreign key can be either *complete* or *incomplete* as indicated by the `incomplete` property. Incomplete foreign keys can only be used for searching. If the foreign key is complete, then the combination of the values of the fields that participates in the foreign key (excluding supplement fields) uniquely identifies another resource. A client program can navigate a complete foreign key. For example, a client program can follow a link from the timebooking to the resource. The property `rel` indicates to the client that links on booking entities with the link relation `data:key:resourceid_resource` is a link to the resource of that booking. The `rel` will only be present for complete foreign keys.

Here is an example of the links that will be available on a *timebookings* entity record:

```json
{
    "meta": {
        "entityName": "timebookings"
    },
    "data": {
        "id": "8b18291a-6ebd-49ab-a3c8-f1ee313c4e1f",
        "subject": "Solution construction",
        "start": "2017-08-14 00:00:00",
        "finish": "2017-08-20 23:59:59",
        "value": 7.0,
        "description": "",
        "eventkindcolor": "#FF008000",
        "approvalstatus": "none",
        "assignmentid": "b80be9de-9c46-4c47-add6-0526454a32b8",
        "assignmenttext": "",
        "resourceid": "10c226d6-744e-43a2-9048-fe4d11b5fd61",
        "resourcename": "Gina Ford",
        "eventid": "1d2089d5-7c88-4111-b695-43c25f21a2d7",
        "eventname": "Solution construction",
        "eventerp": 100.0,
        "bookingcategoryid": "00000000-0000-0000-0000-000000000000",
        "bookingcategoryname": "",
        "bookingcategorydisplaywarningbeforechange": false,
        "bookingcategorywarningmessage": "",
        "bookingcategorycolor": "#00000000",
        "lastmodified": "2021-05-26 08:39:06"
    },
    "links": {
        "self": {
            "href":  "http://server/api/bookings/time/8b18291a-6ebd-49ab-a3c8-
f1ee313c4e1f",
            "rel": "self"
        },
        "action:update": {
            "href":  "http://server/api/bookings/time/8b18291a-6ebd-49ab-a3c8-
f1ee313c4e1f",
            "rel": "action:update"
        },
        "action:delete": {
            "href":  "http://server/api/bookings/time/8b18291a-6ebd-49ab-a3c8-
f1ee313c4e1f",
            "rel": "action:delete"
        },
```

```
        "data:key:assignmentid_assignment": {
            "href":     "http://server/api/assignments/b80be9de-9c46-4c47-add6-
0526454a32b8",
            "rel": "data:key:assignmentid_assignment"
        },
        "data:key:resourceid_resource": {
            "href":      "http://server/api/resources/10c226d6-744e-43a2-9048-
fe4d11b5fd61",
            "rel": "data:key:resourceid_resource"
        },
        "data:key:eventid_event": {
            "href":        "http://server/api/events/1d2089d5-7c88-4111-b695-
43c25f21a2d7",
            "rel": "data:key:eventid_event"
        }
    }
}
```

Data Types

Integer[6], float[7], and double[8] data types are represented as a JSON number and must conform to the number grammar rule. String, DateTime, color, GUID, and enum values are represented as JSON string values and must conform to the string grammar rule[9].

Integer

The integer is a signed 32-bit integer, which allows values in the following range: -2,147,483,648 to 2,147,483,647.

Examples of acceptable values are 1000, 0, and -549.

Float

The float represents a 32-bit floating-point value, which allows the following precision and approximate range: $\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$. Therefore, the C# constants *Single.PositiveInfinity*, *Single.NegativeInfinity,* and *Single.NaN* are not allowed.

Examples of acceptable value are 100, 0.892, 314159e16, and -2e-3

Double

The double represents a 64-bit floating-point value, which allows the following precision and approximate range: $\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$. Double values are represented as a JSON number and must conform to the number grammar rule. Therefore, the C# constants *Double.PositiveInfinity*, *Double.NegativeInfinity* and *Double.NaN* are not allowed.

---

[6] https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/int

[7] https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/float

[8] https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/double

[9] (ECMA-404: the json data interchange format)

Examples of acceptable value are 100, 0.892, 314159e64, and -2e3

Boolean

The boolean data type consists of the values true and false. JSON Booleans are represented as the JSON values true and false.

String

The string data type is used to represent text. The character set UTF-8 is the default. Note that Unicode characters may be escaped using \uXXXX where X is a hexadecimal digit, for example, \u0066 for the letter 'f'.

The API supports string values up to 3.0×10^5 characters in length.

Examples of acceptable values are "" and "Hello world".

DateTime

The DateTime data type is used to represent a date and a time. DateTime values are stored without any information about the timezone. That is, a client creating a DateTime for 1st of January 00:00:00, will appear as 1st of January 00:00:00 regardless of where the DateTime is later requested from. Therefore, no time zone indication should be part of the serialized date time format, which is: "yyyy-MM-dd HH:mm:ss".

Examples of acceptable values are "2000-01-01 00:00:00" and "2000-12-24 00:00:00".

Color

The color data type is used to represent colors. The color is declared in the generic ARGB format in hexadecimal values: #AARRGGBB. AA – alpha component, RR – red component, GG – green component, BB – blue component.

JSON: Color values are encoded in strings using the above format. Examples of acceptable values are "#0014AA88" and "#883273FF".

GUID

The GUID data type is a **G**lobally **U**nique **ID**entifier. A GUID is represented as 16 bytes in hexadecimal format, separated by hyphens: "XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX". A new GUID is always created by the server, e.g. when creating a new entity. GUID values can still be submitted by a client, for example, when pointing to an existing entity in an entity relation.

> **Note:** GUID value are encoded in strings. Example of acceptable values are "65d7a983-32d4-4d17-aa41-dd980ae32a54" and "97df0be7-acaa-4663-bb03-ba4e462b7467".

Enum

Enum is an enumerated type that defines a set of named constants. To obtain a list of the used enums in the API and the possible values for each enum, the *enum_overview* link can be followed from the entities endpoint, for example:

```
{
    "meta": {
```

```
        "version": "1.0"
    },
    "links": {
        "enum_overview": {
            "href": "http://server/resapi/api/enums",
            "rel": "enum_overview"
        },
            ...
}
```

The enum endpoint responds with a JSON document containing a single JSON object (enums) which has references to all enums (for example, `eventkind`) and its possible values (project, amount, and so on).

An enum object contains a property for each possible value. The name of the property represents the fixed name of the property. The value of the property is the language specific title of the enum property.

```
{
    "enums": {
        "eventkind": {
            "project": "Project",
            "milestone": "Milestone",
            "task": "Task",
            "summary": "Summary",
            "absence": "Absence",
            "amount": "Amount"
        },
            ...
    }
}
```

For example, the enum `eventkind` is used by the *event* entity. Thus, the specification of an event entity references this particular enum type:

```
{
    "entityName": "events",
    "actions": {
        "action:read": {
            "title": "Read events",
            "rel": "action:read"
        }
    },
    "fields": {
        "kind": {
            "name": "kind",
            "title": "Kind",
            "type": "enum",
            "enumType": "eventkind",
```

```
        "mandatory": false,
        "key": false,
        "update": false,
        "create": false,
        "references": []
    },
    ...
},
...
}
```

An entity `enum` field is always paired with an `enumType` parameter. More information about the `enumType` can be retrieved from the *enum_overview* endpoint as described above.

JSON: Enum values are encoded in strings. Examples of acceptable values for an enum of type `eventkind` is: are "project" and "task".

Default Values

The default values for the data types (listed in Data Types) are defined in the following table.

Data type	Default value	Comment
Integer	0 (zero)	
Float	0 (zero)	
Double	0 (zero)	
Boolean	false	
String	"" / null	Internally 'null' but always formatted during de/serialization as an empty string "".
DateTime	0001-01-01 00:00:00	Serialization formatting depends on settings.
Color	#00000000	
GUID	00000000-0000-0000-0000-000000000000	
Enum	(E)0	The value produced by casting 0 to the Enum type E.

For more details regarding default values of .NET data types, see https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/default-values-table.

Hyperlinks

Hyperlinks work just like links on a web page and point to related entities or actions available.

Hyperlinks are also used to represent available state transitions. For example, to update a booking, the client application needs to follow a specific hyperlink. Entities have hyperlinks for all available state

transitions. Each link has an associated link relation, which is a value that defines what the link can be used for (like updating or deleting an entity).Hyperlinks are used for two purposes in the REST API

1. Referencing related entities, for example, a booking that has a link to the assigned resource.

2. State transitions, for example, a booking can have a link to its *delete* action.

Link Relations

self: The self-link relation indicates a hyperlink to the context resource (the resource related to the current endpoint you are interacting with). This is useful when a client program interacts with one resource and the web service responds with the state of another resource.

specification Indicates a reference to the specification resource for the context resource (an entity).

action:init Indicates a link that is used to get a "blank record" of the context resource.

action:create Indicates a link that is used to create a new resource (entity). Clients must use the HTTP POST method with an entity record structure as the request payload.

action:update Indicates a link that is used to perform the update state transition. This state transition changes the values of one or more fields in a record. Clients must the HTTP POST method with an (updated) entity record structure as the request payload.

> **Note:** POST is used instead of PUT; this is to align with the way the Maconomy REST API works. Also notice that, only a full entity update is supported for now. Partial updates, using the HTTP PATCH method, might be implemented in future releases.

action:delete Indicates a link that is used to perform the delete state transition. This state transition deletes a record. Clients must use the HTTP DELETE method. Note that deleting some entities might cascade delete associated entities.

HTTP Response Status Codes and Error Messages

For a detailed guide to response status codes returned by the different endpoints, refer to the Swagger reference documentation provided by the REST API.

Status Codes

Each response from the REST API has a HTTP status code. The status code indicate whether the request was successful or failed. If the request was unsuccessful, the status code indicates what kind of failure occurred which is used by client applications to decide how to proceed. The People Planner REST API uses the common[10] HTTP response status codes.

Error Messages

An error response from the REST API is sent a JSON object in the body of the HTML response. The API uses the standard Problem Details format[11].

---

[10] https://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml
[11] https://datatracker.ietf.org/doc/html/rfc7807

Filtering

Entity Landing Pages contain a link to a filter resource. The filter endpoint can be used to make queries for multiple entities and supports paging, sorting, field selection and data restrictions.

Remember to never rely on the URL structure, but always rely on link relations.

The filter responds with appropriate payload using default parameters. It is possible to use custom parameters by specifying those in the URL (query string parameters).

Parameter	Description
Skip	[integer] The number of records to skip in the result set. This can be used, in combination with the 'take' parameter, for implementing paging.
Take	[integer] The number of records to return in the result set. This can be used, in combination with the 'skip' parameter, for implementing paging.
Sort	[array] Specifies the sorting of the record in the result set.
Filter	[array] Specifies a restriction. Only records matching the filter restriction are included in the result set.
Select	[array] Specifies which fields are included in the records of the result set. Note, that some fields are always included, such as "ID".

An example of a filtering query looks like this:

```
http://server/restap/api/bookings/time/filter?take=3&skip=1&filter=["resource
name","=","Jean-Luc Picard"]
&select=["id","value","resourceid","resourcename"]
```

This query skips the first record, retrieves subsequent three time booking entity records with the specific resource name, and only returns the fields id, value, resourceid and resourcename.

For more information on how to construct sorting and filter parameters, refer to the Swagger documentation found at http://<server>:<port>/RestApi/api-doc/index.html.

You can also refer to the following:

https://js.devexpress.com/Documentation/Guide/Data_Binding/Data_Layer/#Reading_Data/Sorting

https://js.devexpress.com/Documentation/Guide/Data_Binding/Data_Layer/#Reading_Data/Filtering/Binary_Filter_Operations

Create, Update and Delete

Creating an entity

Creating a new entity record is accomplished by sending an HTTP POST request to the endpoint specified by the `action:create` link relation. This link can be obtained from the *entity landing page* of the current resource context.

For example, the following is part of the JSON response from the entity landing page of the "timebookings" resource context:

```
{
    "meta": {
        "entityname": "timebookings"
    },
    "links": {
        ...
        "action:create": {
            "href": "http://server/restapi/api/bookings/time",
            "rel": "action:create"
        },
        ...
    }
}
```

The payload of the create request should contain JSON serialization of the entity which should be created, for example:

```
{
    "subject": "New Booking",
    "start": "2021-01-28 09:00:00",
    "finish": "2021-01-28 10:00:00",
    "value": 1,
    "description": "This is my booking"
    "eventid" : "878aed6a-ad19-461e-b4d7-517651b61df0",
    "resourceid" : "b35f6dec-7ddb-4d6c-b3fd-3fd43e1ec969"
}
```

If the resource was successfully created, the API returns with an HTTP response status code `200 OK` and the full entity representation in the body. The response also includes an HTTP `Location` response header with an URL to the newly created resource.

Concurrency control

To ensure that the API can resolve possible conflicts when concurrent users attempt to modify or delete the same entity at the same time, the API supports *e-tags* and `If-Match` request headers.

E-tag

For every mutable entity, the representation includes an e-tag in the response.

When looking at a single entity, the e-tag is found as the `ETag` HTTP response header variable.

However, when using the filter endpoint, the e-tag is transferred in the meta data section of the JSON serialized response, for example:

```
"records": [
    {
        "meta": {
            "rowNumber": 0,
            "etag": "1991524367c441559f0b0015932d06e0-63757615140577…"
        },
        "data": {
            "id": "19915243-67c4-4155-9f0b-0015932d06e0",
            ...
        },
        ...
    }
]
```

The e-tag is updated on the server side each time a change is made. For example, User A and B obtains e-tag "X" from the server. User A makes an update and the e-tag is updated to "Y" on the server. When user B tries to commit changes, the server rejects the changes, since user B is trying to commit changes with e-tag "X" and the current state of the resource is e-tag "Y".

If user B still wants to update the entity, user B must first obtain a new representation of the entity (which contains the updated e-tag: "Y"). The user is now able to determine which previous changes they wants to keep and which it wants to override. By using the new e-tag, the user should be able to update the entity (assuming no other has changed it again in the meanwhile).

If-Match request header

To enforce this behavior, the server only accepts *conditional requests* for updating and deleting entities.

A conditional request is a request containing an `If-Match` HTTP request header. For example, the following is an example of the headers in an update request:

```
POST /restapi/api/bookings/time/19915243-67c4-4155-9f0b-0015932d06e0 HTTP/1.1
Host: server:80
Content-Type: application/json
If-Match: 1991524367c441559f0b0015932d06e0-637576151405770000
Accept: application/json; v=1.0
```

If no e-tag is supplied the API responds with a `428 Precondition Required` HTTP response status code. If an incorrect or outdated e-tag is supplied, the API responds with a `412 Precondition Failed` status code.

After a successful update, the API returns with a status code `200 OK` result containing a full representation of the updated entity and its corresponding new e-tag.

After as successful delete, the API returns with a `204 No Content` result, and no e-tag

Updating an entity

Updating an existing entity record is accomplished by sending an HTTP POST request to the endpoint specified by the `action:update` link relation. This link can, for example, be obtained from a specific *entity resource endpoint*.

For example, the following is part of the JSON response from the entity resource endpoint of a "timebookings" resource:

```
{
    "meta": {
        "entityName": "timebookings"
    },
    "data": {
        "id": "19915243-67c4-4155-9f0b-0015932d06e0",
        "subject": "New Booking",
        "start": "2021-01-28 09:00:00",
        "finish": "2021-01-28 10:00:00",
        "value": 1,
        "description": "This is my booking"
        "eventid" : "878aed6a-ad19-461e-b4d7-517651b61df0",
        "resourceid" : "b35f6dec-7ddb-4d6c-b3fd-3fd43e1ec969"
    },
    "links": {
            ...,
        "action:update": {
            "href":    "http://server/restapi/api/bookings/time/19915243-67c4-
4155-9f0b-0015932d06e0",
            "rel": "action:update"
        },
            ...
    }
}
```

The payload of the HTTP POST request should be the full entity representation. If possible, any fields omitted in the request payload might be reset with the default for the datatype. The `specification` of the entity contains meta data on what fields are mandatory and not.

For example, to update the finish time and value of a booking, the following request payload could be sent:

```
{
    "subject": "New Booking Updated",
    "start": "2021-01-28 09:00:00",
    "finish": " 2021-01-28 11:00:00",
    "value": 2,
    "description": "This is my updated booking"
    "eventid" : "878aed6a-ad19-461e-b4d7-517651b61df0",
```

```
    "resourceid" : "b35f6dec-7ddb-4d6c-b3fd-3fd43e1ec969"
}
```

When making an update request, remember that it must be in the form on a conditional request as described in concurrency section.

Deleting an entity

Deleting an existing entity is accomplished by sending an HTTP DELETE request to the endpoint specified by the `action:delete` link relation. This link can, for example, be obtained from a specific *entity endpoint.*

```
{
    "meta": {
        "entityName": "timebookings"
    },
    "data": {
        "id": "19915243-67c4-4155-9f0b-0015932d06e0",
            ...
    },
    "links": {
            ...,
        "action:delete": {
            "href":    "http://server/restapi/api/bookings/time/19915243-67c4-
4155-9f0b-0015932d06e0",
            "rel": "action:delete"
        },
            ...
    }
}
```

When making a delete request, remember that it must be in the form of a conditional request as described in concurrency section.

References

ECMA-404: the json data interchange format. December 2017. <http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf>.

About Deltek

Better software means better projects. Deltek delivers software and information solutions that enable superior levels of project intelligence, management and collaboration. Our industry-focused expertise makes your projects successful and helps you achieve performance that maximizes productivity and revenue. www.deltek.com