RSI Tutorial

Author	in-tech Architecture Team
E-Mail	project_vw_eemf_rsi_architektur@in-tech.com
First release	23.06.2017
Version	1.2.2

Table of contents

1 Introduction

The Hyper Text Transport Protocol (HTTP) is the most-used exchange protocol for distributed collaborative information systems today. Providing and accessing resources via HTTP(S) is common in all major domains and the field of application is ever-growing. With a great diversity of applications, services, and systems, the automotive sector has been actively participating in the expansion of the field for years.

The Volkswagen Infotainment Web Interface (ViWi) [1] specifies a standardized interface for HTTP-based services to access a vehicle's system resources and functionalities. Consumers of the interface include applications running on in-vehicle infotainment (IVI) systems as well as any device connected via TCP/IP. Alongside HTTP(S), ViWi considers WebSockets to support PULL-, as well as PUSH-based services. ViWi is a RESTful service interface and as such offers a convenient and familiar API to developers. It promotes micro service system design patterns, encourages interconnection between services, and utilizes standardized web security mechanisms like TLS encryption [2]. In short, ViWi transforms a vehicle into a web service and opens up a whole new world of real-time applications, while at the same time catering for security, authentication, and authorization.

In addition to ViWi 1.6.0 [1], this document serves four goals:

1. It describes conceptual details for a concrete implementation of the ViWi specification.
2. It describes extensions of the ViWi protocol that support an actual implementation.
3. It complements the specification by explaining technical details of the submission in an intelligible manner.
4. It includes a continuously updated FAQ of questions that have been directed at the RSI Architecture group at in-tech.

2 Terms and Definitions

Within the RSI context, several terms exist which have a clearly defined meaning. Hence, the following table lists the corresponding definitions to get an overview of the frequently used terms.

Terms	Definitions
undefined	Used to describe an unset property value of an element
unknown	Used to describe a value that is not known by the service

3 The ViWi Protocol

3.1 Request Types

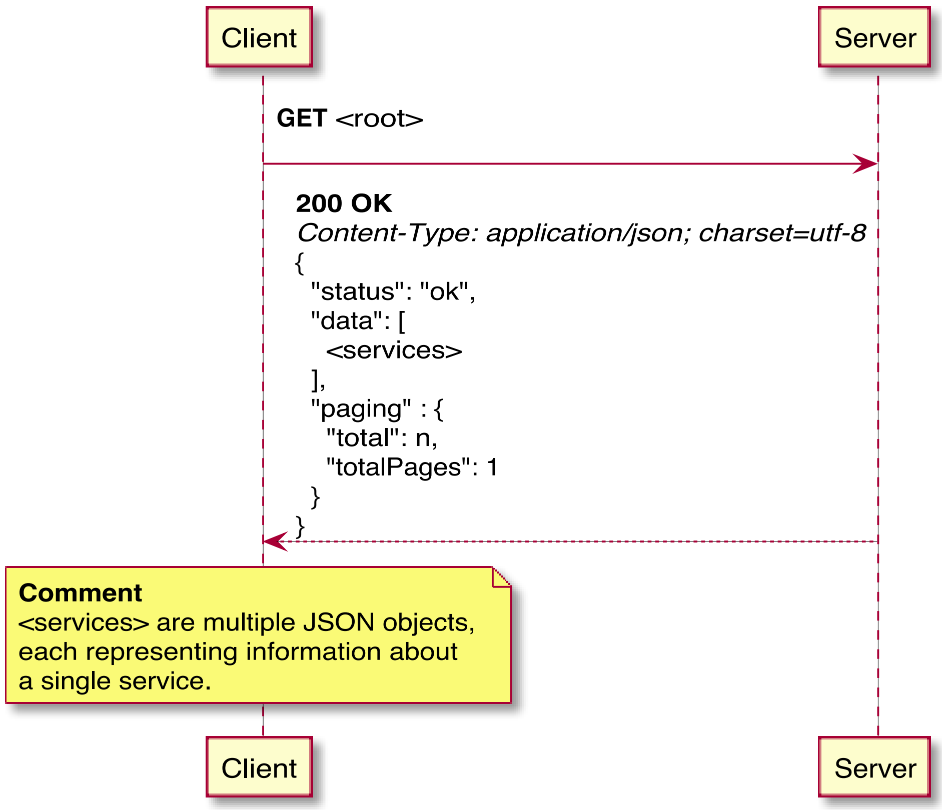
RSI allows clients to send several different request types to the server. Moreover, each request type can be send on different request levels. Since the final effect depends on the type as well as the level of the request, this section describes the resulting response for each request. Each response to a request contains at least the *status* property, that indicates if the request was successful or if any error has occurred. However, this section only handles successful requests, as HTTP error codes still have to be determined and standardized for responses with errors.

3.1.1 GET

GET requests are used to retrieve specific elements. The general workflow is build up in such a way that the client performs a read operation and gets the requested data from the server in return. Still, even though this workflow is unambiguous, the read operation may return different results, depending on the request level. Hence this section describes how a GET request will operate on each request level.

Root:

On root level a GET request will return a list of all available services. If the request was successful, the response will return an HTTP header with status code '200 OK' and contains a JSON Object with properties *status*, *data* and *paging*. The property *status* is needed to indicate if the request was successful and does not contain any further information about the root. However, *data* contains the actual payload of the response, as it holds an array with information about all services of the root. The property *paging* contains paging information about the returned data and has a JSON Object with properties *total* and *totalPages*. The property *total* contains the total number of values that can be returned by the server. Since a common GET request always returns all existing values that match the query, in this case the value of *total* always corresponds to the number of elements contained in the *data* array. On the other hand *totalPages* represents an integer value that is calculated by the total amount of elements divided by the number of elements contained in the data array. Since this ratio is always 1 in a common GET request, the value of *totalPages* is also always 1 in a common GET request. In addition to these two properties, the response may optionally also have the property *timestamp*, which contains the time of the request. The following diagram illustrates the general process of this GET request.



```
{
  "status": "ok",
  "data": [
    <services>
  ],
  "paging" : {
    "total": n,
    "totalPages": 1
  }
}
```

Comment
<services> are multiple JSON objects, each representing information about a single service.

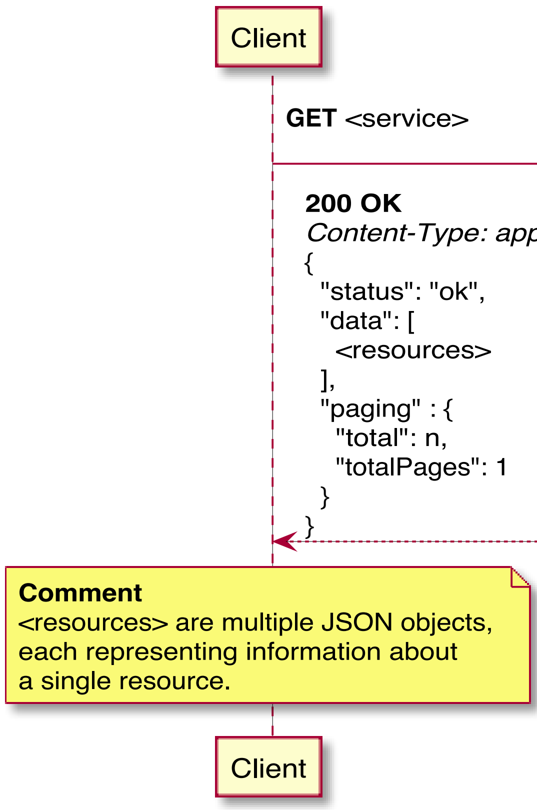
Each returned service is described by its properties *id*, *name*, *uri* and *description*. So if a client sends a GET request on http://localhost:9999/, the response might return the following values:

```
{
        "status": "ok",
        "data": [
            {
                "id": "f9a1073f-e90c-4c56-8368-f4c6bd1d8c96",
                "name": "media",
                "uri": "/media/",
                "description": "The media service"
            },
            {
                "id": "ea65d5eb-d5fb-4ceb-a568-ed24fcf37e20",
                "name": "medialibrary",
                "uri": "/medialibrary/",
                "description": "The medialibrary service"
            }
        ],
        "paging" : {
            "total": 2,
            "totalPages": 1
        }
    }
```

As can be seen above, in this example the root contains two services ‘f9a1073f-e90c-4c56-8368-f4c6bd1d8c96’ and ‘ea65d5eb-d5fb-4ceb-a568-ed24fcf37e20’ which are described by the corresponding properties. Both services are described within the property “data”, as they represent the actual payload of the response. The property *status* sends the value ‘ok’, since the request was successful and the corresponding information about the services could be transmitted from the server.

Service:

A GET request on service level lists data about all contained resources, such that the locations of the resources can be identified. The server response is similar to the one transferred on root level. According to this, the HTTP header sends the status code ‘200 OK’, if the request was successful. Moreover, also in this case the returned JSON Object contains the properties *status*, *data*, *paging* and optionally also *timestamp*. However, in contrast to a GET request on root level, here also the property *service* will be returned, which contains the serviceObject. The diagram below shows how this request is generally used to retrieve the corresponding data.



In addition to that the following example shows a GET request on the service ‘medialibrary’ to illustrate how data are returned by the service. As a result here the path to the resource ‘tracks’ is returned.

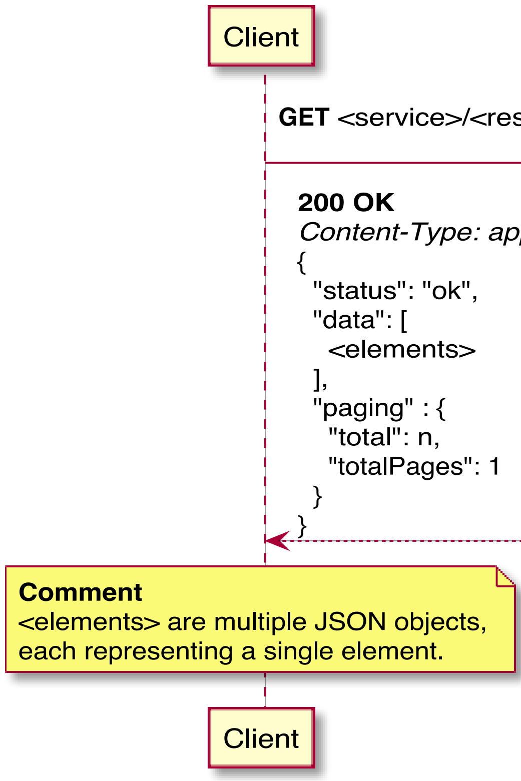
Request: GET on `http://localhost:9999/medialibrary/`

Result:

```
{
      "status": "ok",
      "data": [
        {
          "id": "<UUID>",
          "name": "tracks",
          "uri": "/medialibrary/tracks/"
        }
      ],
      "paging": {
        "total": 1,
        "totalPages": 1
      }
    }
```

Resource:

When performing a GET request on resource level, all elements of the resource will be returned by a JSON Object. The HTTP header contains the status code ‘200 OK’ if the request was successful. The returned JSON-Object contains the properties *status*, *data*, paging and optionally *timestamp*. Each element contained in *data* has properties whose values describe the object. If a resource does not have any elements, *data* just contains an empty array. A property value contained within the *data* array can either be a primitive data type or a reference to another resource element. References are generally represented by their XObject representation which consist of a JSON-Object having the properties *id*, *name* and *uri*. The property *id* describes the UUID of the element whereas *name* represents the name of the element. Finally the property *uri* contains the URI of the element that can be used to access the element directly. A generalized schema of a GET request on resource level is visualized below.


Client
Service
GET <service>/<resource>
200 OK
Content-Type: application/json; charset=utf-8
Comment
<elements> are multiple JSON objects, each representing a single element.
Client
Service

```
{
  "status": "ok",
  "data": [
    <elements>
  ],
  "paging" : {
    "total": n,
    "totalPages": 1
  }
}
```

As a result, if a resource ‘tracks’ exists, a GET request on it will return information about all individual tracks contained by this resource. The concrete request and the returned data for such a request are illustrated in the following:

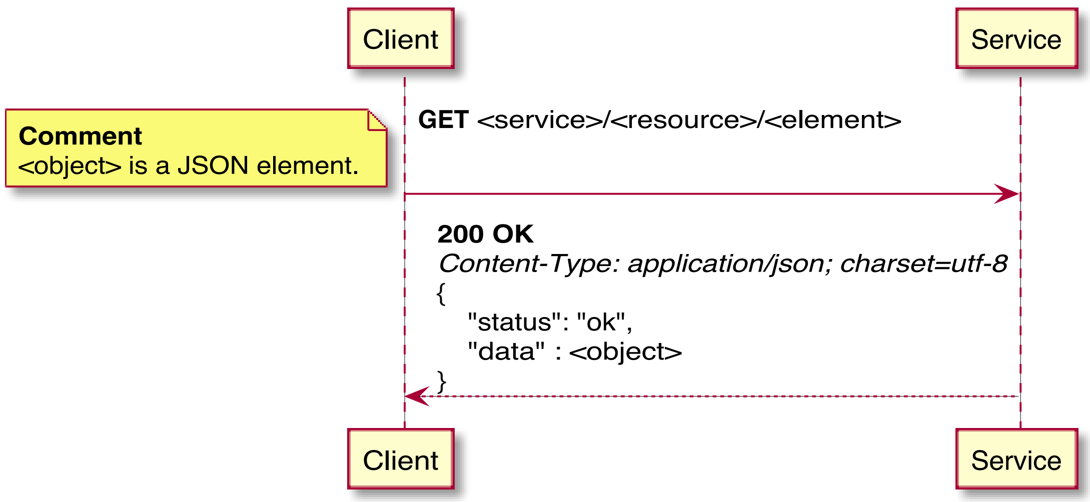
A GET request on `http://localhost:9999/medialibrary/tracks/`

Result:

```
{
      "status": "ok",
      "data": [
        {
          "uri": "/medialibrary/tracks/4b247930-a2ab-49bf-b8f4",
          "id": "4b247930-a2ab-49bf-b8f4",
          "name": "Me and my empty wallet",
          "image": "/cdn/images/hills.jpg",
          "rating": 5,
          "disc": 0,
          "duration": 240
        },
        {
          "uri": "/medialibrary/tracks/976a2844-862e-48f8-bc68-8f2f86613228",
          "id": "976a2844-862e-48f8-bc68-8f2f86613228",
          "name": "The louder, the better",
          "image": "/cdn/images/loud.jpg",
          "rating": 3,
          "disc": 0,
          "duration": 175
        }
      ],
      "paging": {
        "total": 2,
        "totalPages": 1
      }
    }
```

Element:

Performing a GET request on an element will return its JSON representation. Since a GET request on resource level returns an array of element representations, the outcome of a GET request on element level corresponds to a single array value of the associated resource. Thus, the returned JSON Object also contains attributes describing the element by primitive data types or references to other JSON-Objects which are described by their XObject values. So if information about a specific track is required, a GET request needs to be executed on it. As displayed below, this GET request results in an HTTP response with the `200` status code and a JSON object as body, containing a `status` property (which in a successful scenario will always have the value `ok`), the JSON-Object which contains all information about this element in the property `data` and an optional timestamp. Below, a general GET request on element level illustrates a schematic representation of this process to get an overview of the client-server interaction needed to retrieve the desired data.


Client
Service
Comment
<object> is a JSON element.
GET <service>/<resource>/<element>
200 OK
Content-Type: application/json; charset=utf-8
{
"status": "ok",
"data" : <object>
}
Client
Service

GET on `http://localhost:9999/medialibrary/tracks/4b247930-a2ab-49bf-b8f4`

Result:

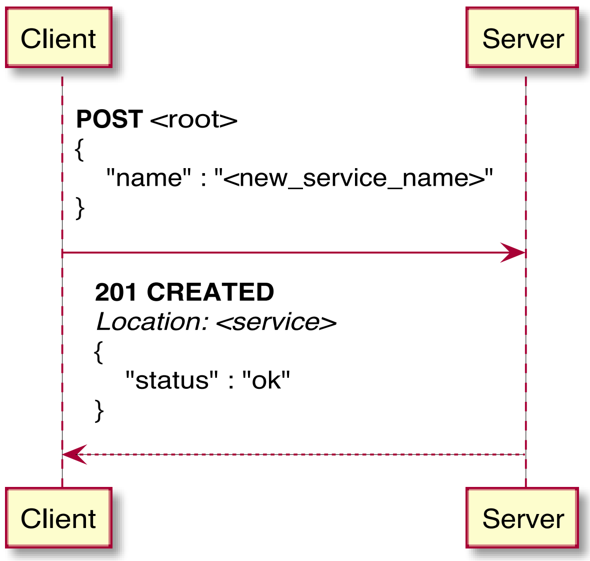
```
{
      "status": "ok",
      "data": {
        "uri": "/medialibrary/tracks/4b247930-a2ab-49bf-b8f4",
        "id": "4b247930-a2ab-49bf-b8f4",
        "name": "Me and my empty wallet",
        "image": "/cdn/images/hills.jpg",
        "rating": 5,
        "disc": 0,
        "duration": 240
      }
    }
```

3.1.2 POST

A POST request can be used to create services, resources and elements or to update property values of elements. Also here the final effect depends on the level where the operation has been performed. So calling a post request on root level will create a new service. Moreover, the same request will create a new resource when calling it on service level, whereas a POST request on resource level will create a new element within the resource. Finally, when performing a POST request on an element, the element itself will be updated by the new data.

Root

On root level a POST request can be used to add new services to the root. To create a new service, the client has to transmit the name and optionally the description for the new service. The values for the properties *id* and *uri* should be generated by the server to ensure the *id* is unique and the URI will be generated appropriately. The information needed for creating a new service are transferred by a JSON Object that holds the necessary property values. Afterwards, the server will send the status code '201 CREATED' in the HTTP header if the request was successful. The corresponding JSON Object transferred by the response then contains the location of the new service as well as the status to indicate if the request was successful. The client-server communication of this process is illustrated in the following diagram.



As an example for a POST request on root level, a root with the services 'media' and 'medialibrary' is given. So a GET request on root `http://localhost:9999/` returns the following output:

```
{
        "status": "ok",
        "data": [
            {
                "id": "f9a1073f-e90c-4c56-8368-f4c6bd1d8c96",
                "name": "media",
                "uri": "/media/",
                "description": "The media service"
            },
            {
                "id": "ea65d5eb-d5fb-4ceb-a568-ed24fcf37e20",
                "name": "medialibrary",
                "uri": "/medialibrary/",
                "description": "The medialibrary service"
            }
        ],
        "paging" : {
            "total": 2,
            "totalPages": 1
        }
    }
```

The following POST request on `http://localhost:9999/` can then be used to add a new service ‘newMediaService’ to the root:

```
{
      "name": "newMediaService",
      "description": "The new media service"
    }
```

If the new service could be added to the root, the subsequent server response notifies the client about the successful operation by sending the following message:

```
{
      "status": "ok"
    }
```

Afterwards the client can send another GET request on `http://localhost:9999/` to receive the updated data. Since the service ‘newMediaService’ has been added to the root before, in this example the GET request will return a list of three services. Consequently, the output for this GET request will then look as follows:

```
{
        "status": "ok",
        "data": [
            {
                "id": "f9a1073f-e90c-4c56-8368-f4c6bd1d8c96",
                "name": "media",
                "uri": "/media/",
                "description": "The media service"
            },
            {
                "id": "ea65d5eb-d5fb-4ceb-a568-ed24fcf37e20",
                "name": "medialibrary",
                "uri": "/medialibrary/",
                "description": "The medialibrary service"
            },
            {
                "id": "ea65d5eb-d5fb-4ceb-a568-ae35fbf21e11",
                "name": "newMediaService",
                "uri": "/newMediaService/",
                "description": "The new media service"
            }
        ],
        "paging" : {
            "total": 3,
            "totalPages": 1
        }
    }
```

Service:

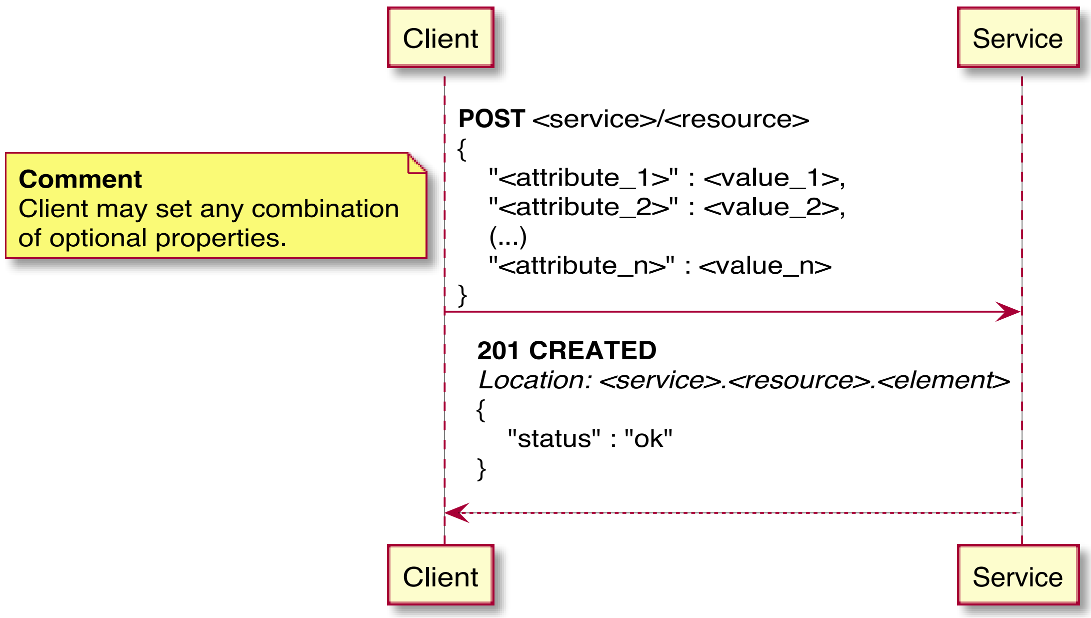
Clients are not allowed to create a resource.

Resource:

A POST request on resource level adds a new element to this resource. The POST request results in an HTTP response with the `201` or `202` status code and a JSON object as body, containing a `status` property (which in a successful scenario will always have the value `ok`) and an optional timestamp.

Status code `201` is used if creation has been successfully completed.

Status code `202` may be used if the request has been accepted but not completed, meaning that it is still being processed. In this case the HTTP response must have a `Location` header, containing information about the future URI where the results will be in case of a successful operation. The following diagram illustrates this functionality in more detail.


Client
Service
Comment
Client may set any combination of optional properties.
POST <service>/<resource>
{
"<attribute_1>" : <value_1>,
"<attribute_2>" : <value_2>,
(...)
"<attribute_n>" : <value_n>
}
201 CREATED
Location: <service>.<resource>.<element>
{
"status" : "ok"
}
Client
Service

A concrete example for a POST request on a resource is visualized in the following. It shows how to use this request for adding a new collection item. So assumed the following resource 'collections' is given which only contains one element,

```
{
      "status": "ok",
      "data": [
        {
          "uri": "/media/collections/deadbeef-d2c1-11e6-9376-df943f51f0d8",
          "id": "deadbeef-d2c1-11e6-9376-df943f51f0d8",
          "name": "default",
          "items": [ "item1" ]
        }
      ],
      "paging": {
        "total": 1,
        "totalPages": 1
      }
    }
```

then the following POST request can be used to add the new element to the resource.

POST on `http://localhost:9999/media/collections/` with mime type `application/json` and body

```
{
      "name": "newCollectionItem"
    }
```

```
{
      "status": "ok"
    }
```

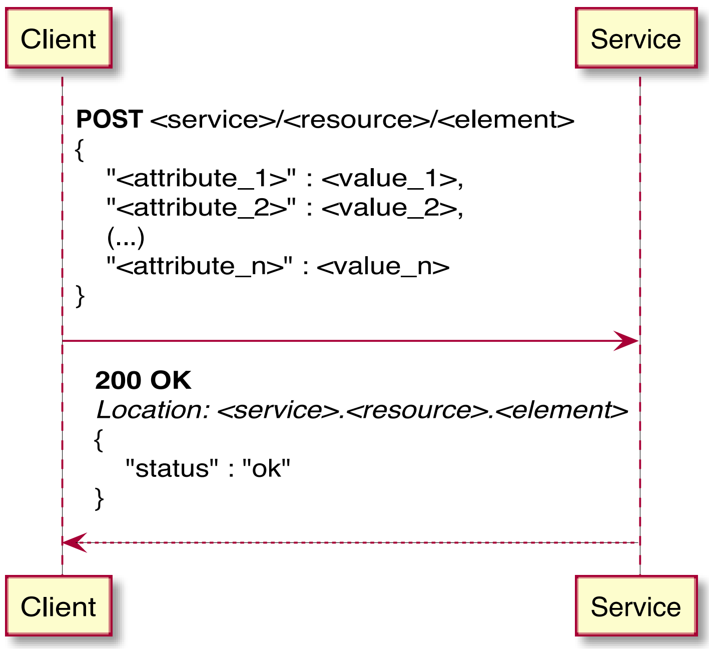
The posted element is not part of the response. As a result, when performing a new GET request on the collection also the newly added element will be displayed as depicted below.

```
{
      "status": "ok",
      "data": [
        {
          "uri": "/media/collections/deadbeef-d2c1-11e6-9376-df943f51f0d8",
          "id": "deadbeef-d2c1-11e6-9376-df943f51f0d8",
          "name": "default",
          "items": [ "item1" ]
        },
        {
          "uri": "/media/collections/1e367a60-5ccc-11e7-b53a-43e196b8eb83",
          "id": "1e367a60-5ccc-11e7-b53a-43e196b8eb83",
          "name": "newCollectionItem"
        }
      ],
      "paging": {
        "total": 2,
        "totalPages": 1
      }
    }
```

Element:

A POST request on an element can be used to update its property values. The POST request results in an HTTP response with the `200` or `202` status code and a JSON object as body, containing a `status` property (which in a successful scenario will always have the value `ok`), the JSON-Object which contains all information about this element and an optional timestamp.

Status code `200` is used if an update has been successfully completed.
Status code `202` may be used if the request has been accepted but not completed, meaning that it is still being processed. In this case the HTTP response must have a `Location` header, containing information about the future URI where the results will be in case of a successful operation. A general POST request is visualized in the following diagram. Please note that the mandatory properties `id` and `uri` are immutable. However, this does not apply for the property `name`, which is also a mandatory property.


Client
Service
POST <service>/<resource>/<element>
{
"<attribute_1>" : <value_1>,
"<attribute_2>" : <value_2>,
(...)
"<attribute_n>" : <value_n>
}
200 OK
Location: <service>.<resource>.<element>
{
"status" : "ok"
}
Client
Service

So if an element contains the property *items* which has the value '["item1"]', this property value can be updated by a POST request on `http://localhost:9999/media/collections/deadbeef-d2c1-11e6-9376-df943f51f0d8` with body

```
{
        "items": ["item1", "item2", "item3"]
    }
```

As a result the old value ["item1"] of the property 'items' has been replaced by the new one:

```
{
        "items": ["item1", "item2", "item3"]
    }
```

An example response body could look like this:

```
{
        "status": "ok"
    }
```

3.1.3 DELETE

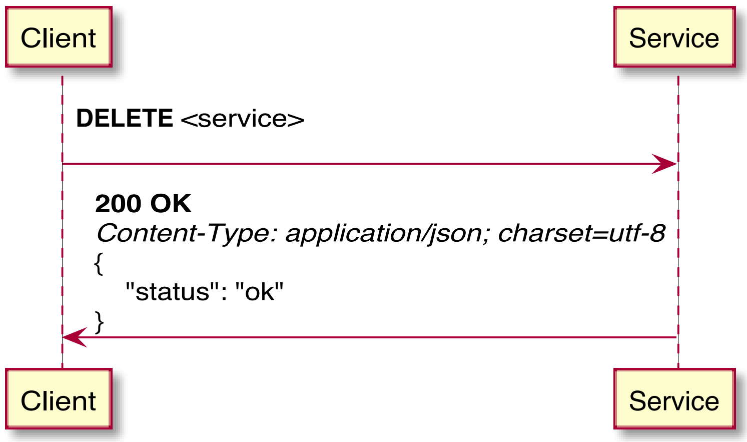
Root:

A `DELETE` request on a root is not valid.

Service:

A `DELETE` request on a service is used to de-register the service from the system. If the request was successful, the server returns ‘200 OK’ and transfers a JSON Object that has the properties *status* and *service*. Moreover, also in this case a property *timestamp* may optionally be transferred by the response to indicate the request time.

In the following a general `DELETE` request on a service is visualized:



A specific example could look like this:

`DELETE` on `http://localhost:9999/media/`

A successful operation will lead to a simple response, that only indicates that the de-registering happened without errors.

```
{
         "status": "ok"
    }
```

Resource:

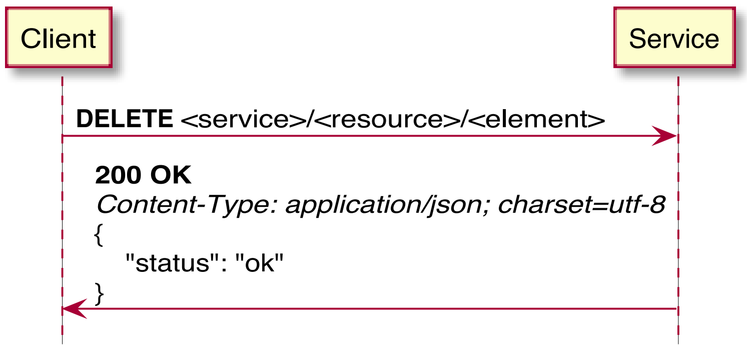
A `DELETE` request on a resource is not valid.

Element:

A `DELETE` request can be used to delete elements or specific properties of that element. An element/field is not available after deletion and cannot be recovered. The DELETE request results in an HTTP response with the `200` or `202` status code and a JSON object as body, containing a `status` property (which in a successful scenario will always have the value `ok`) and an optional timestamp.

Status code `200` is used if deletion has been successfully completed.
Status code `202` may be used if the request has been accepted but not completed, meaning that it is still being processed.

In the following a general `DELETE` request on an element is visualized:


Client
Service
DELETE <service>/<resource>/<element>
200 OK
Content-Type: application/json; charset=utf-8
{
"status": "ok"
}

A specific example could look like this:

`DELETE` on
`http://localhost:9999/media/collections/deadbeef-d2c1-11e6-9376-df943f51f0d8`

The corresponding response body will then return the following data if the request was successful:

```
{
        "status": "ok"
    }
```

3.1.4 SUBSCRIBE

The ViWi protocol uses WebSockets to allow the server to push information to a client. To make sure each client only gets necessary information, the clients have to subscribe for events on the GET URL of a query they are interested in. Then the push mechanism will be used to receive updates. Also here the final effect depends on the request level. To establish a WebSocket connetion use the protocol prefix `ws://` instead of `http://`.

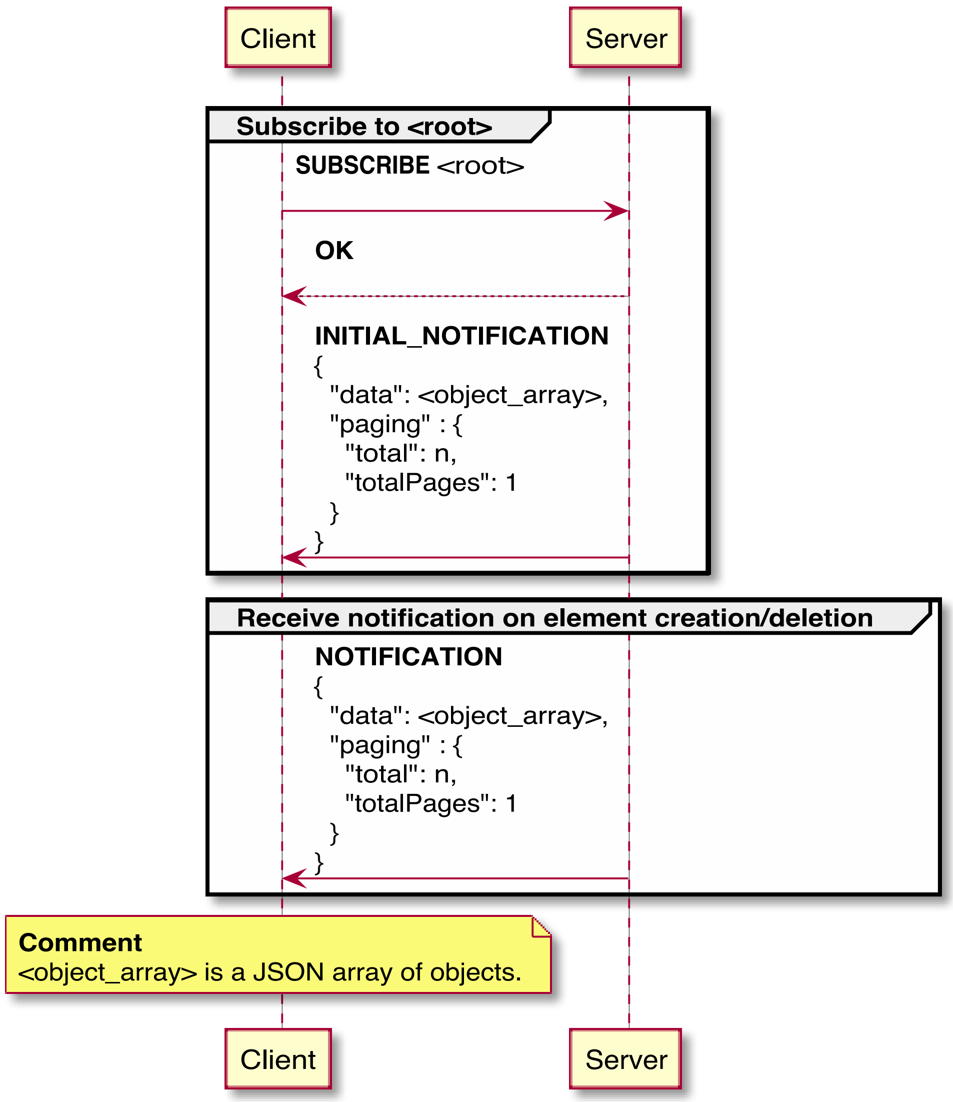
Root:

A subscription on root level will notify the client about changes in the service list. The subscribe request itself will be sent to the server as a JSON Object using the established WebSocket connection. This request contains the following properties:

- type
- event
- interval
- updatelimit

The value of *type* should always be ‘subscribe’, as this object indicates that the corresponding request represents a subscribe request. Furthermore, *event* is needed to inform the server about the location the client wants to subscribe to. Hence the value of *event* always has to represent an URI. However, to allow a client to subscribe to the same server several times with different subscribe parameters, he can add an ID for this request by adding a ‘#’ followed by the ID. Since the ID itself will not be evaluated by the server and is only needed for distinguishing the requests on client side, it is only necessary to keep the ID unique per client. The following two parameters *interval* and *updatelimit* are optional and can be used to control the update frequency. So *interval* determines the time between two updates when periodic updates were requested, whereas *updatelimit* can be used to set the minimum time between two updates when the client should be notified ‘on change’ events, such that if an ‘on change’ event occurred before the time defined by *updatelimit* elapsed, the event will be executed in a delayed way and the client will be informed as soon as the specified time period has been passed. If the state is overridden while waiting for the delay, the updated state will be emitted. Both parameters represent time intervals in milliseconds and have to be numerical. Even though both parameters can be defined within a single request, the client will always be notified about updates periodically when the value for *interval* has been set. Consequently, whenever *interval* is set, the value of *updatelimit* will not have any effect, because then the client will not be informed on changes.

After sending the subscribe request, the server will notify the client about the request status. Moreover, the server also transmits an initial notification that contains the payload. More precisely, a notification generally contains the same data as a common GET request on the subscribed URI would have. Depending on the property values set in the subscribe request, the client will get a notification as soon as the root data change or when the period defined by interval elapsed. The following diagram summarizes the subscription process graphically to get a quick overview of the corresponding interactions.



To subscribe to the root, the client has to send a request to the server that uses the properties explained above. So if the client would like to subscribe to `http://localhost:9999/`, he can send the following request:

```
{
      "type" : "subscribe",
      "event" : "http://localhost:9999/",
      "interval": 100,
      "updatelimit": 100,
      "Authorization": "abcdefg"
    }
```

In this case the server will answer by the following response if the request was successful:

```
{
      "type" : "subscribe",
      "event" : "http://localhost:9999/",
      "status" : "ok"
    }
```

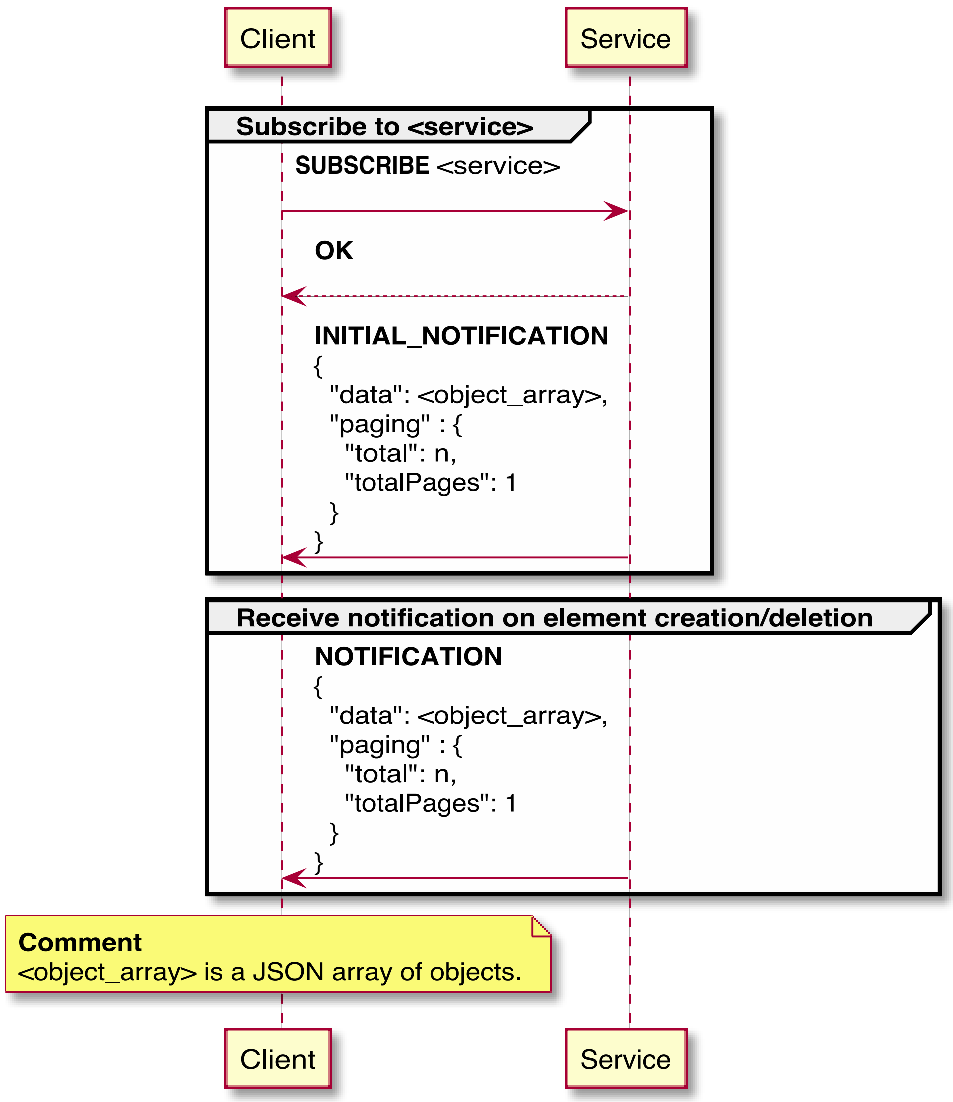
Furthermore, the server will also send an initial notification which contains the payload. If only the two services ‘media’ and ‘medialibrary’ exist within the root, the notification might look as follows:

```
{
        "status": "ok",
        "data": [
            {
                "id": "f9a1073f-e90c-4c56-8368-f4c6bd1d8c96",
                "name": "media",
                "uri": "/media/",
                "description": "The media service"
            },
            {
                "id": "ea65d5eb-d5fb-4ceb-a568-ed24fcf37e20",
                "name": "medialibrary",
                "uri": "/medialibrary/",
                "description": "The medialibrary service"
            }
        ],
        "paging" : {
            "total": 2,
            "totalPages": 1
        }
    }
```

In this example notifications will be sent after 100 milliseconds due to the property value of *interval*. So as long as the data do not change, the client will get the same notification every 100 milliseconds. However, if an update occurs, the returned data will also change accordingly.

Service:

When subscribing to a service, the server will update the client when its collection changes. Possible changes are caused by ‘add’ or ‘delete’ actions or changes of the order or selection of the potential result set. The subscription process itself works in the same way as a subscription on root level. However, in this case it is necessary to subscribe to a service instead of a root. Hence, the subscribe request itself can generally be created in the same way as a subscribe request on root level. Of course, here the value of *event* has to represent the URI to the service the client wants to subscribe to. The subscription process is illustrated in the following diagram.



So if the client wants to subscribe to the service *medialibrary*, the corresponding request that needs to be sent to the server might look as follows:

```
{
        "type" : "subscribe",
        "event" : "/medialibrary",
        "interval": 100,
        "updatelimit": 100
    }
```

After sending the request, the server informs the client about the result using the existing WebSocket connection. So if the request succeeded, the server will send the following response:

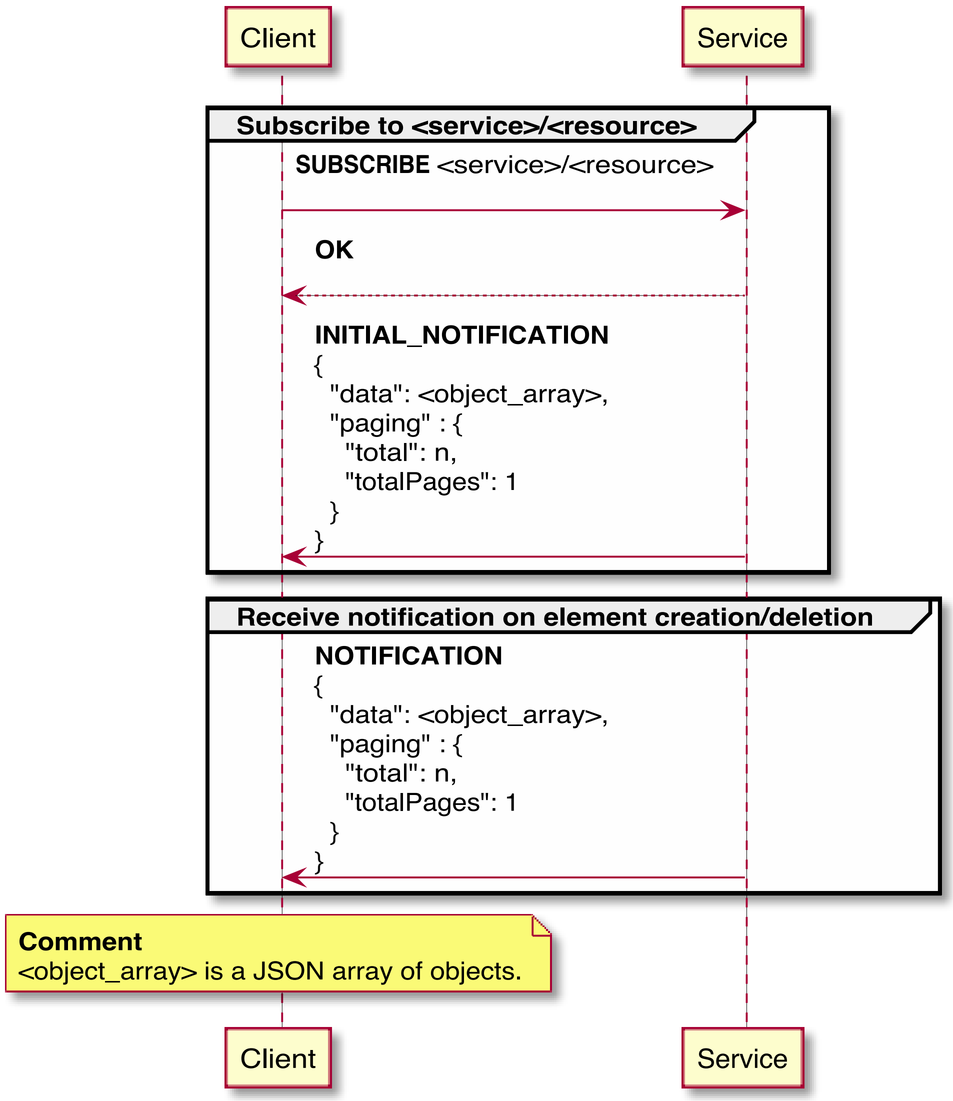
```
{
      "type" : "subscribe",
      "event" : "/medialibrary",
      "status" : "ok"
    }
```

Now the server is ready to notify the client about changes by sending a payload in a periodic way. If in the current example the service ‘medialibrary’ only has the resource ‘tracks’, the notification sent by the server will return the following data:

```
{
        "status": "ok",
        "data": [
            {
                "id": "<UUID>",
                "name": "tracks",
                "uri": "/medialibrary/tracks/"
            }
        ],
      "paging": {
        "total": 1,
        "totalPages": 1
      }
    }
```

Resource:

Like on root or service level a subscription on a resource also updates the client when changes within the resource collection occurred. Since a subscription on resource level generally works in the same way as on root or service level, also the corresponding process is analog to these ones. Hence, also the following sequence diagram for a subscription on resource level only slightly differs from the other ones.



Consequently also the subscribe request is similar to the one for root or service subscriptions and contains the same properties. As previously described, the subscription requests will be sent by using the WebSocket connection. However, also here it has to be considered that the URI of *event* does not point to a root or service but to a resource instead. So if a subscribe request on the resource 'tracks' is desired, the following JSON Object might be sent by the client:

```
{
        "type" : "subscribe",
        "event" : "/medialibrary/tracks/",
        "interval": 100,
        "updatelimit": 100
     }
```

The resulting response will be analog to server responses on other subscription levels. So in this case the following message will be returned through the existing WebSocket connection if the subscription was successful:

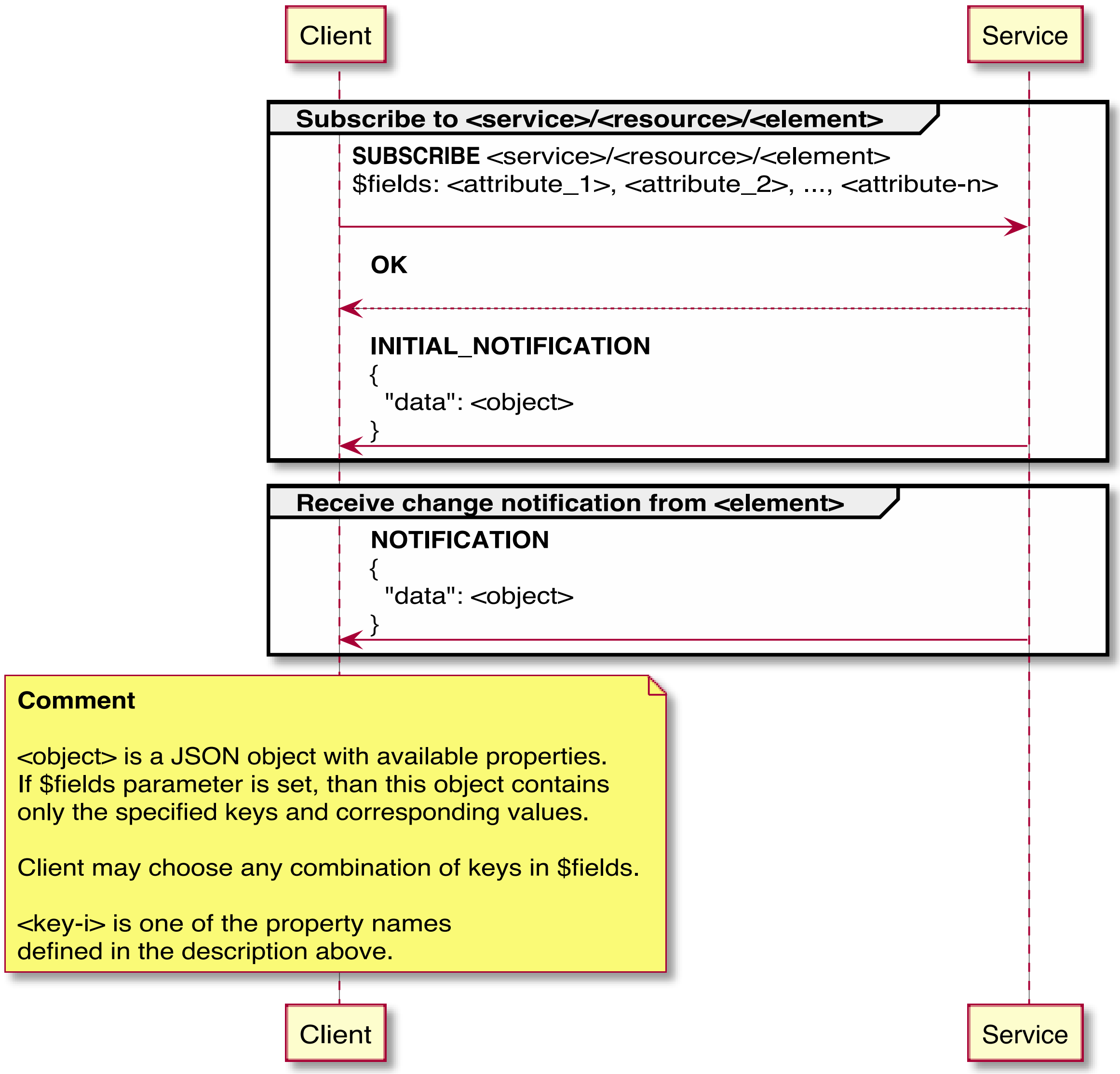
```
{
      "type" : "subscribe",
      "event" : "/medialibrary/tracks/",
      "status" : "ok"
    }
```

The following notifications will also be transferred as on root or service level. If, however, the payload contains a reference to another XObject, this object will be represented as it would be represented by a common GET request:

```
{
      "type" : "subscribe",
      "event" : "/medialibrary/tracks/",
      "status" : "ok",
      "data": [
        {
            "id": "4b247930-a2ab-49bf-b8f4-9ae3b01b3cf2",
            "name": "Me and my empty wallet",
            "uri": "/medialibrary/tracks/4b247930-a2ab-49bf-b8f4-9ae3b01b3cf2",
            "image": "/cdn/images/hills.jpg",
            "rating": 5,
            "playlist": {
              "id": "29eaf68d-89b7-4286-89e2-c5ce4bbdd61f",
              "name": "favorite songs september",
              "uri": "/medialibrary/playlists/29eaf68d-89b7-4286-89e2-c5ce4bbdd61f"
            },
            "disc": 0,
            "duration": 240
        },
        {
            "id": "976a2844-862e-48f8-bc68-8f2f86613228",
            "name": "The louder, the better",
            "uri": "/medialibrary/tracks/976a2844-862e-48f8-bc68-8f2f86613228",
            "image": "/cdn/images/loud.jpg",
            "rating": 3,
            "playlist": {
              "id": "10a131aa-c429-42f1-86d6-ff45a6177acb",
              "name": "my rock music list",
              "uri": "/medialibrary/playlists/10a131aa-c429-42f1-86d6-ff45a6177acb"
            },
            "disc": 0,
            "duration": 175
        }
      ],
      "paging": {
        "total": 2,
        "totalPages": 1
      }
    }
```

Element:

Subscribing on element level means the client will be updated whenever the element itself changes. The following diagram shows how the subscription process works on this level. As stated before, a subscription is requested by using the existent WebSocket connection.


Client
Service
Subscribe to <service>/<resource>/<element>
SUBSCRIBE <service>/<resource>/<element>
$fields: <attribute_1>, <attribute_2>, ..., <attribute-n>
OK
INITIAL_NOTIFICATION
{
"data": <object>
}
Receive change notification from <element>
NOTIFICATION
{
"data": <object>
}
Comment
<object> is a JSON object with available properties.
If $fields parameter is set, than this object contains
only the specified keys and corresponding values.
Client may choose any combination of keys in $fields.
<key-i> is one of the property names
defined in the description above.
Client
Service

Also here the subscribe request looks like the ones on root, service or resource level, such that a subscribe request on a specific track might look as follows:

```
{
       "type" : "subscribe",
       "event" : "/medialibrary/tracks/4b247930-a2ab-49bf-b8f4-9ae3b01b3cf2",
       "interval": 100,
       "updatelimit": 100
    }
```

Comparable to the responses of roots, services or resources, the server will then send the following response through the aldready established WebSocket connection if the request was successful:

```
{
       "type" : "subscribe",
       "event" : "/medialibrary/tracks/4b247930-a2ab-49bf-b8f4-9ae3b01b3cf2",
       "status" : "ok"
    }
```

Notifications will be transferred by sending a payload like on the other subscribe levels. Hence, also here XObjects will be transmitted as illustrated in the following:

```
{
      "type" : "subscribe",
      "event" : "/medialibrary/tracks/4b247930-a2ab-49bf-b8f4-9ae3b01b3cf2",
      "status" : "ok",
      "data":
        {
            "id": "4b247930-a2ab-49bf-b8f4-9ae3b01b3cf2",
            "uri": "/medialibrary/tracks/4b247930-a2ab-49bf-b8f4-9ae3b01b3cf2",
            "name": "Me and my empty wallet",
            "image": "/cdn/images/hills.jpg",
            "rating": 5,
            "playlist": {
              "id": "29eaf68d-89b7-4286-89e2-c5ce4bbdd61f",
              "name": "favorite songs september",
              "uri": "/medialibrary/playlists/29eaf68d-89b7-4286-89e2-c5ce4bbdd61f"
            },
            "disc": 0,
            "duration": 240
        }
    }
```

For additional information about the usage of $fields and property-search please visit the dedicated sections. For information about the `$expand` -behaviour visit the expand-section

When do updates trigger if different expand levels are used?

- All subscriptions besides resource-level-subscriptions trigger as defined in ViWi 1.12 [1].

How do resource subscriptions work?

The definition in chapter 1.12 contained in the ViWi specification[1] regarding the emission of events does only refer to **elements**. In contrast to that, the subscription of **resources** does only emit changes, if the set of returned elements of two subsequent GET-request is different. Therefore, only changes of the selection, quantity (add, remove) or order trigger an event on resource level. This means, if the result set is influenced by query-paramenters for example due to property-search or `$sortBy` the watched result set might change as shown in the following examples:

Example 1 - Change of Quantity:

- The client is subscribed with `SUBSCRIBE /<testService>/<testResource1>/` .
- The client adds a new element to `<testService>/<testResource>/` .

- Server: Triggers event to the subscribed client.

Example 2 - Change of Quantity:

- The client is subscribed with `SUBSCRIBE /<testService>/<testResource1>/` .
- The client removes a element from `<testService>/<testResource>/` .
- Server: Triggers event to the subscribed client.

Example 3 - Change of Selection:

- The client is subscribed with `SUBSCRIBE /<testService>/<testResource1>/?rating=5` .
- The client updates `<testResource1>/<testElement1>` value of `rating` from `5` to `6` , which means that the element will be removed from the corresponding result set.
- Server: Triggers event to the subscribed client.

Example 4 - Change of Order:

- The client subscribes with `SUBSCRIBE /<testService>/<testResource1>/?rating=5&$sortBy=distance`
- `<testResource1>` contains the following elements:

Resource

Element1:
rating = 5
distance = 10

Element2: rating = 5 ~~distance = 11~~ distance = 9

- The client posts `distance=9` to `Element2` as shown in the table above. This update implies a change of the order of the result set, since it is sorted by the numerical value of `distance` .
- The result set will look like this:

Resource

Element2:
rating = 5
distance = 9

Element1:
rating = 5
distance = 10

- Due to the change of the order an update is triggered.

What happens if an element is deleted while a subscription is active?

If an element is deleted while a subscription is active the server sends the following error message to to subscribed client:

```
{
        "type" : "error",
        "code": 410,
        "event": "/<service>/<resource>/<element>#<uniqueid-per-session>",
        "data": "Gone"
    }
```

How are subscription sessions handled?

To create a custom session for the subscription the client can use a `<uniqueid-per-session>`. This allows subscriptions to the same target with different query-parameters or parameter-values. A `<uniqueid-per-session>` can be created by appending it to the URI of the `event` parameter, so that it looks as follows:

```
{
        "type" : "subscribe",
        "event" : "/medialibrary/tracks/4b247930-a2ab-49bf-b8f4-9ae3b01b3cf2#1",
        "interval": 100,
        "updatelimit": 100
    }
```

As a result the service will then return the following response containing the `<uniqueid-per-session>`:

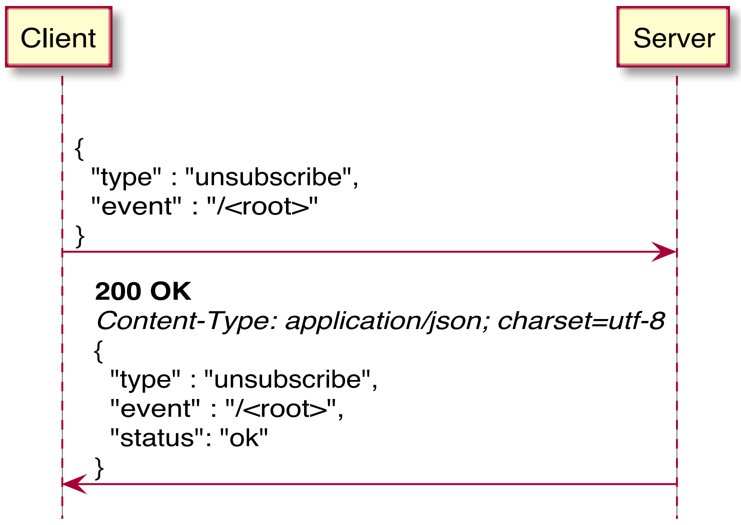
```
{
        "type" : "subscribe",
        "event" : "/medialibrary/tracks/4b247930-a2ab-49bf-b8f4-9ae3b01b3cf2#1",
        "status" : "ok"
    }
```

3.1.5 UNSUBSCRIBE

It is also possible to send an unsubscribe request after a subscription. Such a request will stop the server from sending updates to the client. To get an overview of the corresponding processes, this section describes how unsubscribe requests can be performed on the individual levels and which concrete effect such a request will have.

Root:

If a client has been subscribed to a root, he can unsubscribe from it if he does not want to have notifications anymore. For this purpose the client has to send an unsubscribe request. Such an unsubscribe request will be transmitted by sending a JSON Object to the server with the properties *type*, which always should have the value ‘unsubscribe’ to indicate the purpose of this request, and *event*, which contains the URI the client wants to unsubscribe from. Of course, if the client wants to unsubscribe from a root, the URI also has to represent the location to this root. After transmitting the request, the server will send a response that informs the client about the success of this operation. This response also contains the properties *type* and *event*, so the client can identify the context of this response. However, the actual information about the success of the operation will be transferred by the value of the property *status*. The following diagram describes this process in an abstract way to get a general idea about this procedure.



In a scenario where a client has a subscription on `http://localhost:9999/`, he can use the following request to unsubscribe from this root:

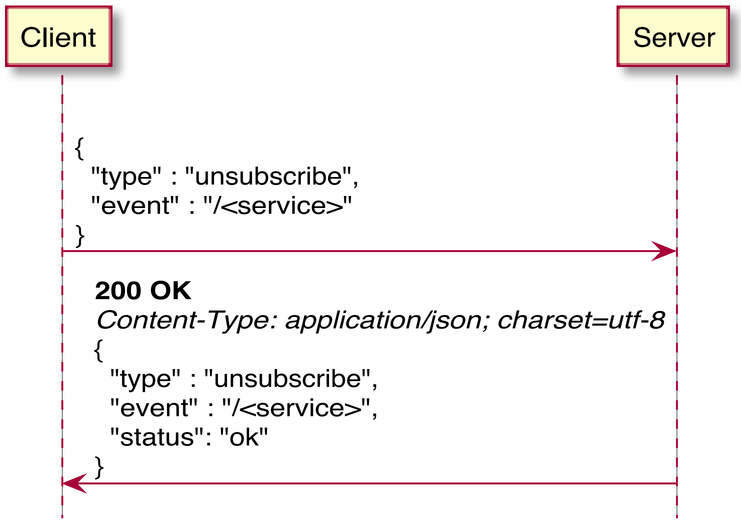
```
{
        "type" : "unsubscribe",
        "event" : "http://localhost:9999/"
    }
```

If the unsubscribe request was successful, the server will then send the following response to let the client know that he is no longer subscribed to the root.

```
{
        "type" : "unsubscribe",
        "event" : "http://localhost:9999/",
        "status" : "ok"
    }
```

Service:

To unsubscribe from service events a client has to send the following JSON message to the server.



```
{
  "type" : "unsubscribe",
  "event" : "/<service>"
}
```

200 OK
Content-Type: application/json; charset=utf-8

```
{
  "type" : "unsubscribe",
  "event" : "/<service>",
  "status": "ok"
}
```

An example for this could look like this:

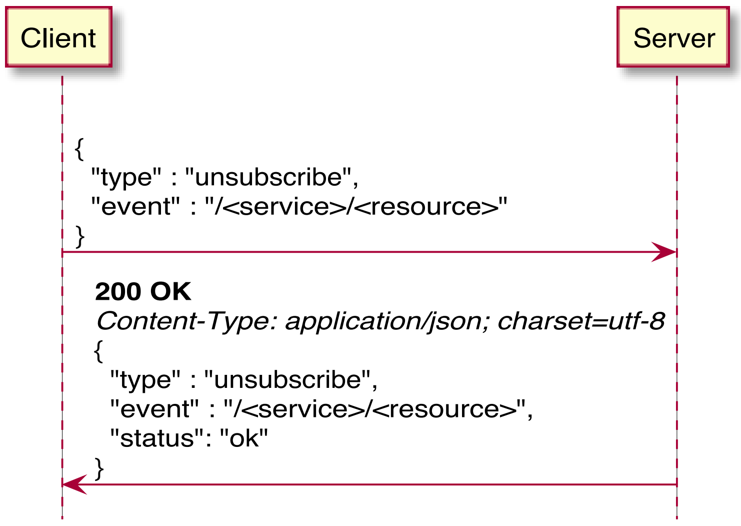
```
{
        "type" : "unsubscribe",
        "event" : "/medialibrary"
    }
```

If the server approved of the unsubscribe request it will send the following response:

```
{
        "type" : "unsubscribe",
        "event" : "/medialibrary",
        "status" : "ok"
    }
```

Resource:

To unsubscribe from resource events a client has to send the following JSON message to the server.


Client
Server
{
"type" : "unsubscribe",
"event" : "/<service>/<resource>"
}
200 OK
Content-Type: application/json; charset=utf-8
{
"type" : "unsubscribe",
"event" : "/<service>/<resource>",
"status": "ok"
}

An example for this could look like this:

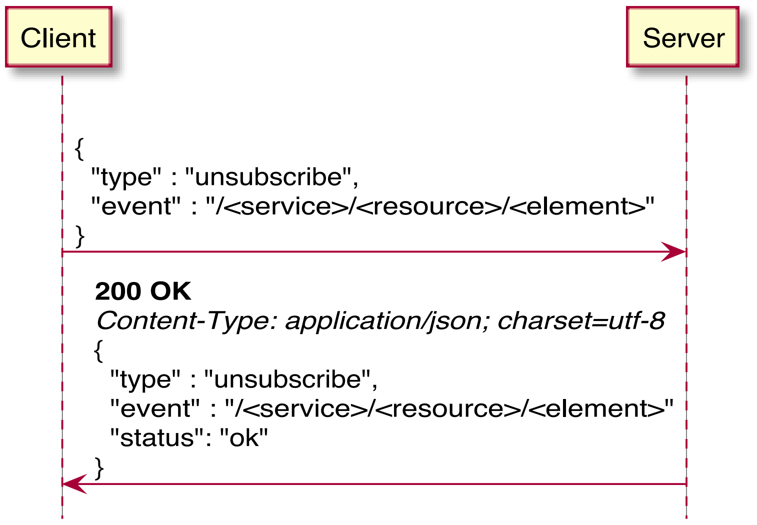
```
{
        "type" : "unsubscribe",
        "event" : "/medialibrary/tracks"
    }
```

If the server approved of the unsubscribe request it will send the following response:

```
{
        "type" : "unsubscribe",
        "event" : "/medialibrary/tracks",
        "status" : "ok"
    }
```

Element:

To unsubscribe from element events a client has to send the following JSON message to the server.


Client
Server
{
"type" : "unsubscribe",
"event" : "/<service>/<resource>/<element>"
}
200 OK
Content-Type: application/json; charset=utf-8
{
"type" : "unsubscribe",
"event" : "/<service>/<resource>/<element>"
"status": "ok"
}

An example for this could look like this:

```
{
        "type" : "unsubscribe",
        "event" : "/medialibrary/tracks/4b247930-a2ab-49bf-b8f4"
    }
```

If the server approved of the unsubscribe request it will send the following response:

```
{
        "type" : "unsubscribe",
        "event" : "/medialibrary/tracks/4b247930-a2ab-49bf-b8f4",
        "status" : "ok"
    }
```

3.2 Query-Parameters

Apart from common requests the ViWi protocol also supports the use of query-parameters. They allow the client to get additional information or to create more complex requests by extending common ones. To get an overview of all these parameters this section explains their functionalities and shows examples to illustrate their usage.

3.2.1 $fields

This parameter can be used to filter GET requests, such that not the complete element will be returned with all its attributes but only those fields of the element that were directly mentioned by the request. If more than one attribute should be displayed, the filtered attributes can be separated by commas. Still, even when filtering a GET request, the fields defined by XObject are always part of the response, since they are mandatory.

As an example assumed a common GET request on `http://localhost:9999/medialibrary/tracks/4b247930-a2ab-49bf-b8f4` returns the following result:

```
{
      "status": "ok",
      "data": {
        "id": "4b247930-a2ab-49bf-b8f4",
        "name": "Me and my empty wallet",
        "uri": "/medialibrary/tracks/4b247930-a2ab-49bf-b8f4",
        "image": "/cdn/images/hills.jpg",
        "rating": 5,
        "disc": 0,
        "duration": 240
      }
    }
```

If instead only the attributes *image* and *rating* are of interest, then the filtered GET request `http://localhost:9999/medialibrary/tracks/4b247930-a2ab-49bf-b8f4?$fields=image,rating` will remove all further attributes which are not mandatory, such that the result will look as follows:

```
{
      "status": "ok",
      "data": {
        "id": "4b247930-a2ab-49bf-b8f4",
        "name": "Me and my empty wallet",
        "uri": "/medialibrary/tracks/4b247930-a2ab-49bf-b8f4",
        "image": "/cdn/images/hills.jpg",
        "rating": 5
      }
    }
```

Combining the parameter *$fields* with a DELETE request on element level will remove specific attribute values instead of the whole element. Due the fact that the keyword *null* is not allowed in ViWi, this approach is the default way to unset / delete attributes. The attribute values of *id*, *name* and *uri* cannot be deleted because they are mandatory and have to be defined for every element. All deleted attributes will not be represented in future GET requests. When extending the previous example by performing the DELETE request `http://localhost:9999/medialibrary/tracks/4b247930-a2ab-49bf-b8f4?$fields=image,rating` the attribute values of *image* and *rating* will be removed and replaced by the value *undefined*, such that a subsequent GET request on `http://localhost:9999/medialibrary/tracks/4b247930-a2ab-49bf-b8f4` will return the following output:

```
{
      "status": "ok",
      "data": {
        "id": "4b247930-a2ab-49bf-b8f4",
        "name": "Me and my empty wallet",
        "uri": "/medialibrary/tracks/4b247930-a2ab-49bf-b8f4",
        "disc": 0,
        "duration": 240
      }
    }
```

Moreover, it is also possible to use the *$fields* attribute in a subscribe request as well. When requesting an on-change subscription in this context, the client will only get an update notification from the server if a mandatory attribute value or a value of an attribute referenced by *$fields* has been updated. So assumed the following subscribe request has been sent by the client:

```
{
      "type": "subscribe",
      "event": "/medialibrary/tracks/4b247930-a2ab-49bf-b8f4?$fields=image",
      "updatelimit": 100
    }
```

Then the client will only get notified if the attribute values of *id*, *name*, *uri* or *image* have been updated. If instead the attribute values for *rating*, *disc* or *duration* get an update, the server will not inform the client about these changes. Moreover, subsequent changes will result in a notification whose payload only contains those attributes that were not filtered out by *$fields* before. So if the property value *image* gets a new value ‘newhills’ in this example, the service might send the following notification to the subscribed client:

```
{
      "type": "data",
      "event": "/medialibrary/tracks/4b247930-a2ab-49bf-b8f4?$fields=image",
      "data": {
        "id": "4b247930-a2ab-49bf-b8f4",
        "name": "Me and my empty wallet",
        "uri": "/medialibrary/tracks/4b247930-a2ab-49bf-b8f4",
        "image": "/cdn/images/newhills.jpg"
      },
      "paging": {
        "total": 1,
        "totalPages": 1
      },
      "timestamp": "2017-06-23T12:00:00+00:00"
    }
```

If instead also an interval had been added to the request, updates would have been transferred periodically. Thus, in such a case the parameter *$fields* would not have any influence on the update conditions. However, notifications about change would then also just return the mandatory properties as well as those attributes that were defined by *$fields*.

The same behavior applies to the resource level. If the request `http://localhost:9999/medialibrary/tracks/?$fields=image` is used to query all tracks and to limit the fields, the following response will be returned by the server.

```
{
      "status": "ok",
      "data": [
        {
          "id": "00000000-0000-0000-0000-000000000001",
          "name": "Me and my empty wallet",
          "uri": "/medialibrary/tracks/00000000-0000-0000-0000-000000000001",
          "image": "/cdn/images/hills.jpg"
        },
        {
          "id": "00000000-0000-0000-0000-000000000002",
          "name": "Title n",
          "uri": "/medialibrary/tracks/00000000-0000-0000-0000-000000000002",
          "image": "/cdn/images/album-n.jpg"
        }
      ],
      "paging": {
        "total": 2,
        "totalPages": 1
      }
    }
```

3.2.2 $sortby

On resource level the parameter *$sortby* can be used to sort results of a GET request by specific attributes. Primitive values are sorted alphabetically or numerically while complex objects are ordered by the mandatory attribute '*name*'. Complex objects can also be sorted in a different way if the service implemented a different behavior. If elements are sorted by a list, the comparison will be performed in such a way that the values having the same indices are compared successively until two compared values are unequal or one of the lists has no further elements. Regardless of the concrete implementation, also a descending sort order is allowed by adding a '-' in front of the attribute that defines the sorting.

To illustrate this concept by example the following common GET request `http://localhost:9999/medialibrary/tracks/` is assumed:

```
{
      "status": "ok",
      "data": [
        {
          "uri": "/medialibrary/tracks/4b247930-a2ab-49bf-b8f4",
          "id": "4b247930-a2ab-49bf-b8f4",
          "name": "Me and my empty wallet",
          "image": "/cdn/images/hills.jpg"
        },
        {
          "uri": "/medialibrary/tracks/976a2844-862e-48f8-bc68-8f2f86613228",
          "id": "976a2844-862e-48f8-bc68-8f2f86613228",
          "name": "The louder, the better",
          "image": "/cdn/images/loud.jpg"
        },
        {
          "uri": "/medialibrary/tracks/adbb974c-bfb3-43d7-a873-546b10ab95b9",
          "id": "adbb974c-bfb3-43d7-a873-546b10ab95b9",
          "name": "Eat, sleep, code, repeat",
          "image": "/cdn/images/eat.jpg"
        },
        {
          "uri": "/medialibrary/tracks/c5c20560-acc8-46d8-88ce-e4d39530752b",
          "id": "c5c20560-acc8-46d8-88ce-e4d39530752b",
          "name": "Only the brave",
          "image": "/cdn/images/brave.jpg"
        }
      ],
      "paging": {
        "total": 4,
        "totalPages": 1
      }
    }
```

When adding a *$sortby* query-parameter and replacing the GET request by `http://localhost:9999/medialibrary/tracks/?$sortby=-image` , the data will be sorted alphabetically in descending order according to the attribute *image* as visualized in the following.

```
{
      "status": "ok",
      "data": [
        {
          "uri": "/medialibrary/tracks/976a2844-862e-48f8-bc68-8f2f86613228",
          "id": "976a2844-862e-48f8-bc68-8f2f86613228",
          "name": "The louder, the better",
          "image": "/cdn/images/loud.jpg"
        },
        {
          "uri": "/medialibrary/tracks/4b247930-a2ab-49bf-b8f4",
          "id": "4b247930-a2ab-49bf-b8f4",
          "name": "Me and my empty wallet",
          "image": "/cdn/images/hills.jpg"
        },
        {
          "uri": "/medialibrary/tracks/adbb974c-bfb3-43d7-a873-546b10ab95b9",
          "id": "adbb974c-bfb3-43d7-a873-546b10ab95b9",
          "name": "Eat, sleep, code, repeat",
          "image": "/cdn/images/eat.jpg"
        },
        {
          "uri": "/medialibrary/tracks/c5c20560-acc8-46d8-88ce-e4d39530752b",
          "id": "c5c20560-acc8-46d8-88ce-e4d39530752b",
          "name": "Only the brave",
          "image": "/cdn/images/brave.jpg"
        }
      ],
      "paging": {
        "total": 4,
        "totalPages": 1
      }
    }
```

3.2.3 $expand

The ViWi protocol allows services to deliver objects that have references to other objects. Although usually referenced objects are just represented by their mandatory properties, the ViWi protocol also allows a requested object to get expanded such that all properties of the referenced objects will be returned as well. In addition to that the concept of resolve level has been introduced, which limits the allowed expansion levels and avoids possible endless loops due to circular references. The expansion level describes how depth a recursive search for expandable objects may be. According to the ViWi protocol expansion levels between 0 and 3 are allowed. To enable expansion, the query-parameter *$expand* has to be added to a GET request on resource or element level. Valid values of this parameter can either be a property name or an integer value between 0 and 3. If the property name is given, only the object referenced by the denoted value will be expanded. If, on the other hand, the value of this query-parameter represents an integer, all properties having a reference to another object will be displayed. Depending on the expansion level, also references within referenced objects can be expanded by such a request.

So a GET request on `http://localhost:9999/medialibrary/albums/6149c270-b528-11e3-a5e2-0800200c9a66` might return the following data:

```
{
      "status": "ok",
      "data": {
        "id": "6149c270-b528-11e3-a5e2-0800200c9a66",
        "name": "its in my pocket",
        "artists": [
          {
            "id": "bb3372f0-b527-11e3-a5e2-0800200c9a66",
            "name": "ich",
            "uri": "/medialibrary/artists/bb3372f0-b527-11e3-a5e2-0800200c9a66"
          },
          {
            "id": "bb3372f0-b500-11e3-a5e2-0800200c9a66",
            "name": "du",
            "uri": "/medialibrary/artists/bb3372f0-b500-11e3-a5e2-0800200c9a66"
          }
        ],
        "image": "/cdn/images/image09720.png",
        "uri": "/medialibrary/albums/6149c270-b528-11e3-a5e2-0800200c9a66"
      }
    }
```

To expand the values of the property ‘artist’, the GET request can be replaced by `http://localhost:9999/medialibrary/albums/6149c270-b528-11e3-a5e2-0800200c9a66?$expand=artist` As a result now not just the mandatory attributes of ‘artist’ will be returned but also the remaining ones as can be seen in the following.

```
{
      "status": "ok",
      "data": {
        "id": "6149c270-b528-11e3-a5e2-0800200c9a66",
        "name": "its in my pocket",
        "artists": [
          {
            "id": "bb3372f0-b527-11e3-a5e2-0800200c9a66",
            "name": "ich",
            "tracks": [
              {
                "id": "1ebe63c0-b528-11e3-a5e2-0800200c9a66",
                "name": "me and my empty wallet",
                "uri": "/medialibrary/tracks/1ebe63c0-b528-11e3-a5e2-0800200c9a66"
              },
              {
                "id": "9df7f840-b528-11e3-a5e2-0800200c9a66",
                "name": "wumpel",
                "uri": "/medialibrary/tracks/9df7f840-b528-11e3-a5e2-0800200c9a66"
              }
            ],
            "image": "/cdn/images/image837943.jpg",
            "uri": "/medialibrary/artists/bb3372f0-b527-11e3-a5e2-0800200c9a66"
          }
        ],
        "image": "/cdn/images/image834543.jpg",
        "uri": "/medialibrary/albums/6149c270-b528-11e3-a5e2-0800200c9a66"
      }
    }
```

Similar to that also the GET request `http://localhost:9999/medialibrary/albums/6149c270-b528-11e3-a5e2-0800200c9a66?$expand=1` will expand all properties in the same way. To keep this example simple, here only the property ‘artist’ can be expanded, such that the corresponding result is the same as the previous one.

Analogously the *$expand* parameter can also be applied to resources. Then all returned JSON Objects will be expanded according to the value of the parameter.

```
{
      "status": "ok",
      "data": [
        {
          "id": "6149c270-b528-11e3-a5e2-0800200c9a66",
          "name": "its in my pocket",
          "artists": [
            {
              "id": "bb3372f0-b527-11e3-a5e2-0800200c9a66",
              "name": "ich",
              "tracks": [
                {
                  "id": "1ebe63c0-b528-11e3-a5e2-0800200c9a66",
                  "name": "me and my empty wallet",
                  "uri": "/medialibrary/tracks/1ebe63c0-b528-11e3-a5e2-0800200c9a66"
                },
                {
                  "id": "9df7f840-b528-11e3-a5e2-0800200c9a66",
                  "name": "wumpel",
                  "uri": "/medialibrary/tracks/9df7f840-b528-11e3-a5e2-0800200c9a66"
                }
              ],
              "image": "/cdn/images/image837943.jpg",
              "uri": "/medialibrary/artists/bb3372f0-b527-11e3-a5e2-0800200c9a66"
            }
          ],
          "image": "/cdn/images/image834543.jpg",
          "uri": "/medialibrary/albums/6149c270-b528-11e3-a5e2-0800200c9a66"
        }
      ],
      "paging": {
        "total": 2,
        "totalPages": 1
      }
    }
```

Due to the *$expand* parameter also in this case all properties of ‘artist’ are returned, such that the result of this request provides more information than a common GET request on the resource ‘albums’. Of course, when using the *$expand* parameter on resource level, it is also possible to use a property value instead of an integer. In this case not all values are expanded but only the ones of the resource elements that describe the corresponding property. As a result the request `http://localhost:9999/medialibrary/albums/?$expand=artist` will also return the same output as before if ‘artist’ represents the only expandable property.

Using the *$expand* query-parameter with a subscription will not have any effect on the event trigger. So even when adding this parameter to a subscribe request, the client will only get a notification when an update occurs on expansion level 0 or the period defined by interval has been elapsed. As a result, if the previous example is still given, the client can send the following subscribe request:

```
{
      "type": "subscribe",
      "event": "/medialibrary/albums/?$expand=1",
      "interval": 100,
      "updatelimit": 100
    }
```

Even though this request represents a valid one, the client will only get a notification if a property value of ‘6149c270-b528-11e3-a5e2-0800200c9a66’ or a mandatory property of ‘bb3372f0-b527-11e3-a5e2-0800200c9a66’ changes. So, if for instance the name of ‘1ebe63c0-b528-11e3-a5e2-0800200c9a66’ has been updated, no notification will be sent to the client, although the *$expand* parameter has been added to the request. However, when using the *$expand* parameter with a subscription, the payload will also contain the expanded properties. As a result, since this subscription example bases on the same resource as the previous GET request example, the payload transferred by a notification would be the same as the returned data of the previous example if no data have been changed before.

What does $expand 0, 1, 2, 3 mean for different situations?

Root-Level

`GET /api/v2/?$expand=0`

According to the ViWi protocol a GET-Request on root for expand level 0 returns a list of the available services as XObjects and the available properties.

```
{
    "status": "ok",
    "data": [
        {
          "id": "<UUID>",
          "name": "cdn",
          "uri": "/api/v2/cdn/",
          "description": "testDescription"
        }
      ],
      "paging": {
        "total": 1,
        "totalPages": 1
      }
    }
```

Service-Level

`GET /api/v2/cdn/?$expand=0`

A GET-Request on a service for expand level 0 returns a list of the available resources as XObjects and the available properties.

```
{
    "status": "ok",
    "data": [
        {
          "id": "<UUID>",
          "name": "testResource1",
          "uri": "/api/v2/cdn/testResource1",
          "description": "testDescription"
        }
      ],
      "paging": {
        "total": 1,
        "totalPages": 1
      }
    }
```

Resource-Level

`GET /api/v2/cdn/testResources1/?$expand=0`

A GET-Request on a resource for expand level 0 returns a list of the available elements as XObjects.

```
{
      "data": [
        {
          "id": "8e2a2d0f-a0bd-41ea-a1d9-14586394aea3",
          "name": "testElement1",
          "uri": "/api/v2/testResources1/8e2a2d0f-a0bd-41ea-a1d9-14586394aea3",
          "testProperty1": "testValue1",
          "nestedTestElement": {
            "id": "dbf90ecf-8063-479e-81b0-9145b249a315",
            "name": "nestedTestElement",
            "uri": "/api/v2/cdn/testResources1/dbf90ecf-8063-479e-81b0-9145b249a315"
          },
          "nestedTestArray": [
            {
              "id": "208c469a-1388-4373-809f-919c3d7ce262",
              "name": "nestedTestElement2",
              "uri": "/api/v2/cdn/testResources1/208c469a-1388-4373-809f-919c3d7ce26
            },
            {
              "id": "1e3b02a8-7845-40c3-8337-cbb29018f275",
              "name": "nestedTestElement3",
              "uri": "/api/v2/cdn/testResources1/1e3b02a8-7845-40c3-8337-cbb29018f27
            }
          ]
        }
      ],
      "paging": {
        "total": 1,
        "totalPages": 1
      }
    }
```

`GET /api/v2/cdn/testResources1/?$expand=1`

A GET-Request on a resource for expand level 1 returns a list of the available elements as XObjects with properties.

```
{
      "data": [
        {
          "id": "8e2a2d0f-a0bd-41ea-a1d9-14586394aea3",
          "name": "testElement1",
          "uri": "/api/v2/cdn/testResources1/8e2a2d0f-a0bd-41ea-a1d9-14586394aea3",
          "testProperty1": "testValue1",
          "nestedTestElement": {
            "id": "dbf90ecf-8063-479e-81b0-9145b249a315",
            "name": "nestedTestElement",
            "uri": "/api/v2/testResources/dbf90ecf-8063-479e-81b0-9145b249a315",
            "testproperty1": "testProperty1Value",
            "testProperty2": "testProperty2Value",
            "testProperty3": {
              "id": "c4444420-f84c-45b1-8c6f-c457710eb180",
              "name": "nestedTestElement",
              "uri": "/api/v2/cdn/testResources1/c4444420-f84c-45b1-8c6f-c457710eb18
            },
            "nestedTestArray": [
              {
                "id": "208c469a-1388-4373-809f-919c3d7ce262",
                "name": "nestedTestElement2",
                "uri": "/api/v2/cdn/testResources1/208c469a-1388-4373-809f-919c3d7ce
                "testproperty1": "testProperty1Value",
                "testproperty2": "testProperty2Value"
              },
              {
                "id": "1e3b02a8-7845-40c3-8337-cbb29018f275",
                "name": "nestedTestElement3",
                "uri": "/api/v2/cdn/testResources1/1e3b02a8-7845-40c3-8337-cbb29018f
                "testproperty1": "testProperty1Value",
                "testproperty2": "testProperty2Value"
              }
            ]
          }
        }
      ],
      "paging": {
        "total": 1,
        "totalPages": 1
      }
    }
```

`GET /api/v2/cdn/testResources1/?$expand=2`

A GET-Request on a resource for expand level 2 returns a list of the available elements as XObjects with properties as well as the XObjects of these properties.

```
{
      "data": [
        {
          "id": "8e2a2d0f-a0bd-41ea-a1d9-14586394aea3",
          "name": "testElement1",
          "uri": "/api/v2/cdn/testResources/8e2a2d0f-a0bd-41ea-a1d9-14586394aea3",
          "testProperty1": "testValue1",
          "nestedTestElement": {
            "id": "dbf90ecf-8063-479e-81b0-9145b249a315",
            "name": "nestedTestElement",
            "uri": "/api/v2/cdn/dbf90ecf-8063-479e-81b0-9145b249a315",
            "testProperty1": "testProperty1Value",
            "testProperty2": "testProperty2Value",
            "testProperty3": {
              "id": "c4444420-f84c-45b1-8c6f-c457710eb180",
              "name": "nestedTestElement",
              "uri": "/api/v2/cdn/testResources/c4444420-f84c-45b1-8c6f-c457710eb180
              "testProperty1": "testPropertyValue",
              "testProperty2": {
                "id": "c4444420-f84c-45b1-8c6f-c457710eb181",
                "name": "nestedTestElement",
                "uri": "/api/v2/cdn/testResources/c4444420-f84c-45b1-8c6f-c457710eb1
              }
            }
          },
          "nestedTestArray": [
            {
              "id": "208c469a-1388-4373-809f-919c3d7ce262",
              "name": "nestedTestElement2",
              "uri": "/testResources/208c469a-1388-4373-809f-919c3d7ce262",
              "testproperty1": "testProperty1Value",
              "testproperty2": "testProperty2Value"
            },
            {
              "id": "1e3b02a8-7845-40c3-8337-cbb29018f275",
              "name": "nestedTestElement3",
              "uri": "/testResources/1e3b02a8-7845-40c3-8337-cbb29018f275",
              "testproperty1": "testProperty1Value",
              "testproperty2": "testProperty2Value"
            }
          ]
        }
      ],
      "paging": {
        "total": 1,
        "totalPages": 1
      }
    }
```

Expand level 3 follows the same principle, which means that the properties of the already expanded properties are defined as XObjects with properties.

Element-Level

The `$expand` requests on element level behave analogously to the resource level expand requests with the only exception that the responses do not return a list of elements but a single element.

Example for the `$expand=0` with a nested Object and a nested Array:

```
{
      "data": {
        "id": "8e2a2d0f-a0bd-41ea-a1d9-14586394aea3",
        "name": "testElement1",
        "uri": "/api/v2/cdn/testResources1/8e2a2d0f-a0bd-41ea-a1d9-14586394aea3",
        "testProperty1": "testValue1",
        "nestedTestElement": {
          "id": "dbf90ecf-8063-479e-81b0-9145b249a315",
          "name": "nestedTestElement",
          "uri": "/api/v2/cdn/testResources1/dbf90ecf-8063-479e-81b0-9145b249a315"
        },
        "nestedTestArray": [
          {
            "id": "208c469a-1388-4373-809f-919c3d7ce262",
            "name": "nestedTestElement2",
            "uri": "/api/v2/cdn/testResources1/208c469a-1388-4373-809f-919c3d7ce262"
          },
          {
            "id": "1e3b02a8-7845-40c3-8337-cbb29018f275",
            "name": "nestedTestElement3",
            "uri": "/api/v2/cdn/testResources1/1e3b02a8-7845-40c3-8337-cbb29018f275"
          }
        ]
      }
    }
```

Example for the `$expand=1` with a nested Object and a nested Array:

```
{
      "data": {
        "id": "8e2a2d0f-a0bd-41ea-a1d9-14586394aea3",
        "name": "testElement1",
        "uri": "/api/v2/cdn/testResources1/8e2a2d0f-a0bd-41ea-a1d9-14586394aea3",
        "testProperty1": "testValue1",
        "nestedTestProperty2": {
          "id": "dbf90ecf-8063-479e-81b0-9145b249a315",
          "name": "nestedTestProperty2",
          "uri": "/api/v2/cdn/testResources1/dbf90ecf-8063-479e-81b0-9145b249a315",
          "testProperty": "testString"
        },
        "nestedTestArray": [
          {
            "id": "208c469a-1388-4373-809f-919c3d7ce262",
            "name": "nestedTestElement2",
            "uri": "/api/v2/cdn/testResources1/208c469a-1388-4373-809f-919c3d7ce262"
            "testProperty": "testPropertyValue1",
            "testProperty2": {
              "id": "dbf90ecf-8063-479e-<-9145b249a315",
              "name": "nestedTestProperty2",
              "uri": "/api/v2/cdn/testResources1/dbf90ecf-8063-479e-81b0-9145b249a31
            }
          },
          {
            "id": "1e3b02a8-7845-40c3-8337-cbb29018f275",
            "name": "nestedTestElement3",
            "uri": "/api/v2/cdn/testResources1/1e3b02a8-7845-40c3-8337-cbb29018f275"
            "testProperty": "testPropertyValue2"
          }
        ]
      }
    }
```

3.2.4 $limit

The parameter *$limit* is needed to limit the number of elements returned by a GET request on resource or service level. So if for instance the common GET request `http://localhost:9999/media/renderers` exists:

```
{
      "status": "ok",
      "data": [
        {
          "uri": "/media/renderers/d6ebfd90-d2c1-11e6-9376-df943f51f0d8",
          "id": "d6ebfd90-d2c1-11e6-9376-df943f51f0d8",
          "name": "Netflux",
          "state": "idle",
          "shuffle": "off",
          "repeat": "off",
          "offset": 0,
          "media": "initialCollection"
        },
        {
          "uri": "/media/renderers/deadbeef-d2c1-11e6-9376-beefdead",
          "id": "deadbeef-d2c1-11e6-9376-beefdead",
          "name": "stpd",
          "state": "idle",
          "shuffle": "off",
          "repeat": "off",
          "offset": 0,
          "media": ""
        }
      ],
      "paging": {
        "total": 2,
        "totalPages": 1
      }
    }
```

Then `http://localhost:9999/media/renderers?$limit=1` will just return the first element instead of the whole set as can be seen below.

```
{
      "status": "ok",
      "data": [
        {
          "uri": "/media/renderers/d6ebfd90-d2c1-11e6-9376-df943f51f0d8",
          "id": "d6ebfd90-d2c1-11e6-9376-df943f51f0d8",
          "name": "Netflux",
          "state": "idle",
          "shuffle": "off",
          "repeat": "off",
          "offset": 0,
          "media": "initialCollection"
        }
      ],
      "paging": {
        "total": 1,
        "totalPages": 1
      }
    }
```

Moreover, the special case in which the total number of elements is less than the denoted limit also represents a valid request, because then the server will simply return all available elements.

Another special case is *$limit=0*. This value is used to retrieve information about the total list size. The response contains an empty list in *data* and a *paging* attribute with the number of list items in the attribute *total*. Subscriptions on this URI will inform the client whenever the list size changes. This is only the case, if $limit is set to zero. Subscribing with other values will not trigger an update, if the list size, more specifically the value of “total”, changes.

```
{
      "status": "ok",
      "data": [],
      "paging": {
        "total": 314159
      }
    }
```

3.2.5 $offset

The parameter *$offset* can be used to create a sublist by defining the first returned element of the original list. The value can either be the index of the element or its ID. So assumed the following GET request on `http://localhost:9999/media/renderers` will return the result illustrated below:

```
{
      "status": "ok",
      "data": [
        {
          "uri": "/media/renderers/d6ebfd90-d2c1-11e6-9376-df943f51f0d8",
          "id": "d6ebfd90-d2c1-11e6-9376-df943f51f0d8",
          "name": "Netflux",
          "state": "idle",
          "shuffle": "off",
          "repeat": "off",
          "offset": 0,
          "media": "initialCollection"
        },
        {
          "uri": "/media/renderers/deadbeef-d2c1-11e6-9376-beefdead",
          "id": "deadbeef-d2c1-11e6-9376-beefdead",
          "name": "stpd",
          "state": "idle",
          "shuffle": "off",
          "repeat": "off",
          "offset": 0,
          "media": ""
        }
      ],
      "paging": {
        "total": 2,
        "totalPages": 1
      }
    }
```

When extending this request by the *$offset* parameter, such that the request will be `http://localhost:9999/media/renderers$?offset=1` , the first element will be skipped and only the second one will be returned, such that the result will then look as follows:

```
{
      "status": "ok",
      "data": [
        {
          "uri": "/media/renderers/deadbeef-d2c1-11e6-9376-beefdead",
          "id": "deadbeef-d2c1-11e6-9376-beefdead",
          "name": "stpd",
          "state": "idle",
          "shuffle": "off",
          "repeat": "off",
          "offset": 0,
          "media": ""
        }
      ],
      "paging": {
        "total": 1,
        "totalPages": 1
      }
    }
```

Since the value of *$offset* can also be an ID, the request `http://localhost:9999/media/renderers$?offset=deadbeef-d2c1-11e6-9376-beefdead` will also return the same result. So due to the fact that adding an ID or the index of an element as parameter value may return the same results, the corresponding context has to be considered when deciding whether the value should be an ID or the index of the element.

If the value of $*offset* exceeds the lists upper bound index, a empty list will be returned. The same behavior applies to a subscription. If the client subscribes to a list and all elements above the specified offset are removed, the client receives an update with an empty list. In all cases, the server may not send error codes.

3.2.6 $q

According to the ViWi protocol it is possible to perform free text searches by adding the parameter *$q*. The associated value can be any string, as it represents the search text. When performing a free text search, only elements having at least one property value which matches the search text will be returned. In general a property value matches the search text, if the search text represents the same sequence of characters as the property value. So when searching for a text ‘foo’, only elements will be returned that also have a property value ‘foo’. If for instance an element has a property value ‘foo_bar’ but no other property whose value is ‘foo’, the element will not be returned. However, if a certain position of the search text should be undefined, the wildcard character ‘%’

(URL encoded: %25) can be added at the corresponding position to allow the use of any character sequence there. Moreover, when using the wildcard character '%', it is also possible to leave the wild card value blank. As a result, when searching for a text %foo%, the following property values would match:

- foo
- _foo
- foo_
- foo_bar
- bar_foo
- bar_foo_bar

Free text searches will generally be performed on expand level 0, which means that all properties of referenced XObjects will also be scanned. If necessary, a service can also implement this feature in such a way that the search can be performed on other expand levels. Regardless of the concrete implementation, the service has to care for performance optimization to ensure a quick searching. For this purpose it is also allowed to define the minimum number of characters within a search text field by a service.

As an example for a free text search, a common GET request on `http://localhost:9999/media/renderers` might return the following values:

```
{
      "status": "ok",
      "data": [
        {
          "uri": "/media/renderers/00000000-0000-0000-0000-000000000001",
          "id": "00000000-0000-0000-0000-000000000001",
          "name": "Netflux",
          "state": "idle",
          "shuffle": "off",
          "repeat": "off",
          "offset": 0,
          "media": "initialCollection"
        },
        {
          "uri": "/media/renderers/00000000-0000-0000-0000-000000000002",
          "id": "00000000-0000-0000-0000-000000000002",
          "name": "stpd",
          "state": "idle",
          "shuffle": "off",
          "repeat": "off",
          "offset": 0,
          "media": ""
        }
      ],
      "paging": {
        "total": 2,
        "totalPages": 1
      }
    }
```

A free text search can then be performed by `http://localhost:9999/media/renderers/?$q=Net%x` . As can be seen below, this request will just return one element of the resource, because the other one does not have a property which matches the query.

```
{
      "status": "ok",
      "data": [
        {
          "uri": "/media/renderers/00000000-0000-0000-0000-000000000001",
          "id": "00000000-0000-0000-0000-000000000001",
          "name": "Netflux",
          "state": "idle",
          "shuffle": "off",
          "repeat": "off",
          "offset": 0,
          "media": "initialCollection"
        }
      ],
      "paging": {
        "total": 1,
        "totalPages": 1
      }
    }
```

Furthermore, also a subscribe request on resource level can be combined with a free text search. As a result when using the query-parameter in this context on resource level, the client will receive a notification from the server when the set of elements, which are selected by this query, changed. So if the previous example is given, a client can subscribe to the resource ‘renderers’ by using the following request:

```
{
      "type" : "subscribe",
      "event" : "/media/renderers/?$q=Net%x",
      "interval": 100,
      "updatelimit": 100,
      "Authorization": "abcdefg"
    }
```

Then the server will notify the client about elements that have a property value which starts by the prefix ‘Net’ and ends with ‘x’.

3.3 Combining Query-Parameters

According to the viwi protocol it is possible to use multiple query-parameters within the same request. However, even though numerous combinations of query-parameters are possible, the viwi protocol itself only explains paging in detail. Other useful combinations are not described by this document. Thus, the following section describes the effect for the most common query-parameter combinations that can be used to create more sophisticated requests. For example, a combination of limit and offset can be used to implement paging.

3.3.1 Processing order on combined parameters

Related Parameters:

- Property Search (<property_name>=<property_value>)
- $fields
- $expand
- $limit
- $offset
- $sortby

If a client subscribes to a resource or element and parameters related to sorting, filtering, paging, projection and expanding are defined, the parameters are processed in the following order:

1. Property Search (<property_name>=<property_value>): The result set is filtered according to the given parameters
2. Sorting ($sortby=…): The list will be sorted by the order specified in $sortby
3. Paging ($limit=… and $offset=…): The parameters $offset and $limit will determine the current page
4. Projection ($fields=…): All fields specified by the parameter $fields will be selected
5. Expand ($expand=…): All items will be expanded to the given level

The implemented processing order can differ from the order mentioned above. However, the returned results must correspond to those obtained by applying the original processing sequence.

3.3.2 Selection on different expand levels

Related Parameters:

- $fields
- $expand

When combining *$fields* and *$expand*, the parameter *$fields* will be evaluated before *$expand*. As a result, in this context the expand operation can only be performed when the corresponding value has not been filtered out by the *$fields* parameter in advance. Nevertheless, even when using the expand operation on an attribute that is not referenced by the *$fields* parameter, the request is still a valid one.

As an example for using these parameters together, a common GET request on `http://localhost:9999/medialibrary/tracks/00000000-0000-0000-0000-000000000001` could be used to create the following output:

```
{
      "status": "ok",
      "data": {
        "id": "00000000-0000-0000-0000-000000000001",
        "name": "its in my pocket",
        "artists": [
          {
            "id": "00000000-0000-0000-0000-000000000002",
            "name": "ich",
            "uri": "/medialibrary/artists/00000000-0000-0000-0000-000000000002"
          },
          {
            "id": "00000000-0000-0000-0000-000000000003",
            "name": "du",
            "uri": "/medialibrary/artists/00000000-0000-0000-0000-000000000003"
          }
        ],
        "image": "/cdn/images/image09720.png",
        "uri": "/medialibrary/albums/00000000-0000-0000-0000-000000000001"
      }
    }
```

Replacing this request by `http://localhost:9999/medialibrary/tracks/...`

`...00000000-0000-0000-0000-000000000001?$expand=artist&$fields=image` will then only return the fields *id*, *name*, *uri* and *image*. The property values of *artist* will not be returned, although the expansion has been requested explicitly, due to the evaluation order of the query-parameters. Hence, in this example the data will be returned as illustrated in the following:

```
{
      "status": "ok",
      "data": {
        "id": "00000000-0000-0000-0000-000000000001",
        "name": "its in my pocket",
        "image": "/cdn/images/image09720.png",
        "uri": "/medialibrary/albums/00000000-0000-0000-0000-000000000001"
      }
    }
```

3.3.3 Paging

Related Parameters:

- $limit
- $offset

Paging allows the viwi protocol to not just return all available elements of a resource at once but also to return limited sections of data. This feature is used to limit the amount of transferred data and increase response times. It becomes available by combining the query-parameters *$limit* and *$offset*, because then it is possible to successively return a certain number of elements until all needed elements have been retrieved. So as an example assumed a common GET request on the resource `http://localhost:9999/medialibrary/tracks` returns the following data:

```
{
      "status": "ok",
      "data": [
        {
          "uri": "/medialibrary/tracks/4b247930-a2ab-49bf-b8f4",
          "id": "4b247930-a2ab-49bf-b8f4",
          "name": "Me and my empty wallet",
          "image": "/cdn/images/hills.jpg"
        },
        {
          "uri": "/medialibrary/tracks/976a2844-862e-48f8-bc68-8f2f86613228",
          "id": "976a2844-862e-48f8-bc68-8f2f86613228",
          "name": "The louder, the better",
          "image": "/cdn/images/loud.jpg"
        },
        {
          "uri": "/medialibrary/tracks/adbb974c-bfb3-43d7-a873-546b10ab95b9",
          "id": "adbb974c-bfb3-43d7-a873-546b10ab95b9",
          "name": "Eat, sleep, code, repeat",
          "image": "/cdn/images/eat.jpg"
        },
        {
          "uri": "/medialibrary/tracks/c5c20560-acc8-46d8-88ce-e4d39530752b",
          "id": "c5c20560-acc8-46d8-88ce-e4d39530752b",
          "name": "Only the brave",
          "image": "/cdn/images/brave.jpg"
        }
      ],
      "paging": {
        "total": 4,
        "totalPages": 1
      }
    }
```

Then these data can be filtered by the GET request `http://localhost:9999/medialibrary/tracks?$offset=1&$limit=2` in such a way that only the second and third entry of the returned data list will be displayed. The final result of this GET request will then look as follows:

```
{
      "status": "ok",
      "data": [
        {
          "uri": "/medialibrary/tracks/976a2844-862e-48f8-bc68-8f2f86613228",
          "id": "976a2844-862e-48f8-bc68-8f2f86613228",
          "name": "The louder, the better",
          "image": "/cdn/images/loud.jpg"
        },
        {
          "uri": "/medialibrary/tracks/adbb974c-bfb3-43d7-a873-546b10ab95b9",
          "id": "adbb974c-bfb3-43d7-a873-546b10ab95b9",
          "name": "Eat, sleep, code, repeat",
          "image": "/cdn/images/eat.jpg"
        }
      ],
      "paging": {
        "previous": "/medialibrary/tracks/?$limit=2&$offset=0",
        "next": "/medialibrary/tracks/?$limit=2&$offset=3",
        "total": 4,
        "totalPages": 2
      }
    }
```

If the next page is needed, the GET request `http://localhost:9999/medialibrary/tracks?$offset=3&$limit=2` can be used. Since the last index of the returned list is always known, it is easy to request the next page by adjusting the offset.

If the sum of $offset and *$limit* is bigger than the list size, only the remaining items will be returned. Hence, if the previous GET request is modified in such a way that *$limit* is set to 10, the result will still just contain the last three elements of the resource. The same behavior applies to a subscription. If the client subscribes to a list and the element order within the subscribed page changes afterwards, the client receives an update which describes the new content of the list. However, in case a resource gets modified in such a way that the element order of the subscribed page is not affected, no notification will be sent.

3.4 Property Search

Apart from common query-parameters, RSI also allows the filtering of result sets by properties in such a way that only those elements will be returned which have a certain attribute value. Such a filter will always be evaluated before query-parameters are interpreted to keep intermediate result sets small and reduce computation costs.

A filter can be added to a request by assigning the filter value to the attribute that should be filtered within the request as follows.

```
<property_name>=<property_value>
```

Analogously to search texts of free text searches, also here an element only gets returned if the search text exactly matches the value of the associated property. If it is still desired to search for a substring of a property value, it is also here possible to use the wildcard character % (URL encoded: %25) as in free text searches.

In a concrete example a common GET request on `http://localhost:9999/media/renderers` might return the following values:

```
{
      "status": "ok",
      "data": [
        {
          "uri": "/media/renderers/d6ebfd90-d2c1-11e6-9376-df943f51f0d8",
          "id": "d6ebfd90-d2c1-11e6-9376-df943f51f0d8",
          "name": "Netflux",
          "state": "idle",
          "shuffle": "off",
          "repeat": "off",
          "offset": 0,
          "media": "initialCollection"
        },
        {
          "uri": "/media/renderers/deadbeef-d2c1-11e6-9376-beefdead",
          "id": "deadbeef-d2c1-11e6-9376-beefdead",
          "name": "stpd",
          "state": "idle",
          "shuffle": "off",
          "repeat": "off",
          "offset": 0,
          "media": ""
        }
      ],
      "paging": {
        "total": 2,
        "totalPages": 1
      }
    }
```

Then the GET request http://localhost:9999/media/renderers/?media=initialCollection will only return the first data item as illustrated below, because it is the only one whose property value for *media* matches the property search of the request.

```
{
      "status": "ok",
      "data": [
        {
          "uri": "/media/renderers/d6ebfd90-d2c1-11e6-9376-df943f51f0d8",
          "id": "d6ebfd90-d2c1-11e6-9376-df943f51f0d8",
          "name": "Netflux",
          "state": "idle",
          "shuffle": "off",
          "repeat": "off",
          "offset": 0,
          "media": "initialCollection"
        }
      ],
      "paging": {
        "total": 1,
        "totalPages": 1
      }
    }
```

Property search can also be used fo find values within an array. So executing a request in combination with a property search that refers to an array property will return all elements where the corresponding array contains at least one entry that matches the query. If the property *media* of the previously returned element would return an array of strings instead of a single one, a common GET request on `http://localhost:9999/media/renderers` might return the following output:

```
{
      "status": "ok",
      "data": [
        {
          "uri": "/media/renderers/d6ebfd90-d2c1-11e6-9376-df943f51f0d8",
          "id": "d6ebfd90-d2c1-11e6-9376-df943f51f0d8",
          "name": "Netflux",
          "state": "idle",
          "shuffle": "off",
          "repeat": "off",
          "offset": 0,
          "media": ["initialCollection", "anotherCollection"]
        },
        {
          "uri": "/media/renderers/deadbeef-d2c1-11e6-9376-beefdead",
          "id": "deadbeef-d2c1-11e6-9376-beefdead",
          "name": "stpd",
          "state": "idle",
          "shuffle": "off",
          "repeat": "off",
          "offset": 0,
          "media": []
        }
      ],
      "paging": {
        "total": 2,
        "totalPages": 1
      }
    }
```

Applying for this example the GET request `http://localhost:9999/media/renderers/?media=initialCollection` would then also return the same output as before, because the first returned element of the common GET request contains an array value ‘initialCollection’ that matches the predefined filter.

Furthermore, XObjects can also be filtered by a property search. Comparable to a property search on arrays, in this case a filter on a property that contains an XObject evaluates all mandatory properties of the XObject. If any of the corresponding values matches the filter condition, the corresponding element will be returned. So assumed in contrast to the previous examples the property *media* does not contain a string value or array of strings, but an XObject and a common GET request on it returns the following output:

```
{
      "status": "ok",
      "data": [
        {
          "uri": "/media/renderers/d6ebfd90-d2c1-11e6-9376-df943f51f0d8",
          "id": "d6ebfd90-d2c1-11e6-9376-df943f51f0d8",
          "name": "Netflux",
          "state": "idle",
          "shuffle": "off",
          "repeat": "off",
          "offset": 0,
          "media": {
            "id": "d6ebfd90-d2c1-11e6-9377",
            "name": "initialCollection",
            "uri": "/media/renderers/d6ebfd90-d2c1-11e6-9377"
          }
        },
        {
          "uri": "/media/renderers/deadbeef-d2c1-11e6-9376-beefdead",
          "id": "deadbeef-d2c1-11e6-9376-beefdead",
          "name": "stpd",
          "state": "idle",
          "shuffle": "off",
          "repeat": "off",
          "offset": 0,
          "media": {
            "id": "d6ebfd90-d2c1-11e6-9378",
            "name": "anythingElse",
            "uri": "/media/renderers/d6ebfd90-d2c1-11e6-9378"
          }
        }
      ],
      "paging": {
        "total": 2,
        "totalPages": 1
      }
    }
```

Analogously to the previous examples, also here the GET request `http://localhost:9999/media/renderers/?media=initialCollection` will just return the element ‘d6ebfd90-d2c1-11e6-9376-df943f51f0d8" but not element ’deadbeef-d2c1-11e6-9376-beefdead’. Of course, property searches can also be performed on arrays of XObjects. In such a case the procedure corresponds to a combination of a property search on an array and a property search on a single XObject. So since a property search on an array matches as soon as one value satisfies the search condition and also a property search on a single XObject is suitable when one of its mandatory properties matches the filter, a property search on an array of XObjects matches if at least one mandatory attribute of at least one XObject within the array satisfies the search condition.

Of course, property searches can also be combined with subscriptions on resources. Even though in such cases the notification conditions do not differ from common subscriptions on resources that do not use a property search, the payload returned by a notification only contains those elements of the resource that suit the condition. If the previous example gets changed in such a way that a client not just sends a GET request but subscribes to `http://localhost:9999/media/renderers/?media=initialCollection`, the following payload will be sent by a notification if the resource elements did not change:

```
{
      "type": "data",
      "event": "/media/renderers/?media=initialCollection",
      "data": [
        {
          "uri": "/media/renderers/d6ebfd90-d2c1-11e6-9376-df943f51f0d8",
          "id": "d6ebfd90-d2c1-11e6-9376-df943f51f0d8",
          "name": "Netflux",
          "state": "idle",
          "shuffle": "off",
          "repeat": "off",
          "offset": 0,
          "media": {
            "id": "d6ebfd90-d2c1-11e6-9377",
            "name": "initialCollection",
            "uri": "/media/renderers/d6ebfd90-d2c1-11e6-9377"
          }
        }
      ],
      "paging": {
        "total": 1,
        "totalPages": 1
      },
      "timestamp": "2017-06-23T12:00:00+00:00"
    }
```

As illustrated above, the payload of the notification only contains element ‘d6ebfd90-d2c1-11e6-

9376-df943f51f0d8', since the other one does not have any property whose value corresponds to 'initialCollection'. Hence, when using a property search in combination with a subscription, the payload will be filtered in the same way as when using a property search and a GET request together.

3.5 Large list handling

To improve the performance and enable paging on large lists, they are not mapped as arrays in attributes. Instead they are defined using foreign keys. This key is used to querying the list elements from the resource. For example an object that represents a playlist is extended with a attribute "trackListId" that contains the foreign key. These attribute does not contain a list of tracks, instead it contains a list id.

```
{
      "uri": "/medialibrary/playlists/00000000-0000-0000-0000-000000000001",
      "id": "00000000-0000-0000-0000-000000000001",
      "name": "My awesome playlist",
      "trackListId": 3
    }
```

To get all tracks of the playlist, the client must use the attribute, that contains the id ("trackListId") as parameter and the id of the requested list as value to query the element uri. See also property-search in the previous chapter. For this example "GET /medialibrary/tracks/?trackListId=3". Which results in the following response:

```
[
      {
        "uri": "/medialibrary/tracks/00000000-0000-0000-0000-000000000002",
        "id": "00000000-0000-0000-0000-000000000002",
        "name": "Title 1",
        "artist": "Artist N",
        "trackListId": [
          3
        ]
      }
    ]
```

If the track is included in multiple lists, the attribute contains all lists.

```
[
      {
        "uri": "/medialibrary/tracks/00000000-0000-0000-0000-000000000002",
        "id": "00000000-0000-0000-0000-000000000002",
        "name": "Title 1",
        "artist": "Artist N",
        "trackListId": [
          3,
          4,
          19
        ]
      }
    ]
```

Because property-search is used to select all elements of a list, the subscription on a large list behaves like the property-search subscription described in chapter 3.4.

4 Architecture of the ViWi Protocol

4.1 Access Operations

The ViWi provides the operations `GET` , `PUT` , `POST` , `DELETE` , `SUBSCRIBE` and `UNSUBSCRIBE` that can be applied to different API levels. Additionally, these operations can be sophisticated using the RSI-Access Operations. The access operations are used to specify the request types for all clients. The operations have been used in former service definitions and are now summed up in this document. They are also supported by the RSI Editor, Excel-Generator and several other tools.

The following subsections are intended to ease the understanding of the different representations of the access operations in the RSI-Editor, the Excel-File and the json-representation.

4.1.1 Definition of Access Operations

To get a brief introduction into RSI Access Operations, the following table lists the name of the access operations, the concerned RSI-Operation and -level as well as a description.

RSI Access Operation	RSI-Operation and -level	Description
CREATE_required	`POST /<service>/<resource>/`	On create/post element this property is required (implies CREATE_settable).
CREATE_settable	`POST /<service>/<resource>/`	On create/post element this property is settable.
UPDATE_required	`POST /<service>/<resource>/<element>`	On update element this property is required (implies UPDATE_settable).
UPDATE_settable	`POST /<service>/<resource>/<element>`	On update element this property is settable.
		Create a new element in a collection by

CREATE_element	`POST /<service>/<resource>/`	sending the entity with at least the required properties. The new element's URI is assigned accordingly.
READ_element	`GET /<service>/<resource>/<element>`	Retrieve a representation of the addressed member of the collection, expressed in an appropriate Internet media type.
UPDATE_element	`POST /<service>/<resource>/<element>`	Update element. The updated element's URI is assigned automatically and is returned by the operation.
DELETE_element	`DELETE /<service>/<resource>/<element`	Delete the referred entity of the collection (or its attributes)
SUB_element	`SUBSCRIBE /<service>/<resource>/<element>`	Get updates on element changes (e.g. playing tracks offset on media player) via Websocket
SUB_resource	`SUBSCRIBE /<service>/<resource>/`	Get updates on collection changes (e.g. add, delete of elements) via Websocket. See more at 3.1.4.
READ_resource	`GET /<service>/<resource>/`	List the URIs and additional details of the collection's

		members. See more at 3.1.1.

4.1.2 Application of Access Operations using the RSI Editor

Once a RSI-API-Designer understood the definition of access operations, the RSI-Editor has to be used to apply the desired access operations to a service and the contained resources, elements and properties.

The following table displays the interaction path to apply the desired access operations:

RSI-Operation	RSI-Editor-Interaction
CREATE_required	Resource Access > POST > Property > settable & mandatory
CREATE_settable	Resource Access > POST > Property > settable
UPDATE_required	Element Access > POST > Property > settable & mandatory
UPDATE_settable	Element Access > POST > Property > settable
CREATE_element	Resource Access > POST
READ_element	Element Access > GET
UPDATE_element	Element Access > POST
DELETE_element	Element Access > DELETE
SUB_element	Element Access > subscribable
SUB_resource	Resource Access > subscribable
READ_resource	Resource Access > GET

4.1.3 The resulting JSON-Representations

The following section explains how the access operations, specified via the RSI-Editor, are transformed into a schema.json file.

Create settable and required

Location: `<resourceName>.endpoints.resource`

To restrict a POST-Request on resource level:

The `post.parameters.properties` object contains the settable properties on post/element creation.

The `post.required` array contains the required properties.

Note: The required properties have to exist in the `post.parameters.properties` array, since required properties have to be settable.

```
{
      "resource": {
        "post": {
          "parameters": {
            "description": "see model description",
            "method": "post",
            "properties": {                     //settable props on create
              "renderer": {
                "description": "see model description",
                "format": "uri",
                "type": "string"
              }
            },
            "required": [                       // required props on create
              "renderer"
            ],
            "resource": "maprenderer.snapshots"
          }
        }
      }
    }
```

Update settable and update required

Location: `<resourceName>.endpoints.element`

The `post.parameters.properties` object contains the settable properties on post/element creation.

The `post.required` array contains the required properties.

Note: The required properties have to exist in the `post.parameters.properties` array, since required properties have to be settable.

```
{
      "element": {
        "post": {
          "parameters": {
            "description": "see model description",
            "method": "post",
            "properties": {                       // settable props on update
              "renderer": {
                "description": "see model description",
                "format": "uri",
                "type": "string"
              }
            },
            "required": [                        // required props on update
              "renderer"
            ],
            "resource": "maprenderer.snapshots"
          }
        }
      }
    }
```

CREATE_element

Location: `<resourceName>.endpoints.resource`

To allow the creation of a element of a resource, a `post` property has to exist in `<resourceName>.endpoints.resource`.

```
    {
      "resource": {
        "post": {
        }
      }
    }
```

READ_element

Location: `<resourceName>.endpoints.element`

To allow the creation of a element of a resource, a `get` property has to exist in `<resourceName>.endpoints.element`.

```
{
  "element": {
    "get": {
    }
  }
}
```

UPDATE_element

Location: `<resourceName>.endpoints.element`

To allow the creation of a element of a resource, a `post` property has to exist in `<resourceName>.endpoints.element`.

```
{
  "element": {
    "post": {
    }
  }
}
```

DELETE_element

Location: `<resourceName>.endpoints.element`

To allow the creation of a element of a resource, a `get` property has to exist in `<resourceName>.endpoints.element`.

```
    {
      "element": {
        "delete": {
        }
      }
    }
```

SUB_element

Location: `<resourceName>.systemTriggeredEvents`

To allow the subscription to an element, a `element` string has to be pushed to the `<resourceName>.systemTriggeredEvent` array.

```
{
      "systemTriggeredEvents": [
        "element"
      ]
    }
```

SUB_resource

Location: `<resourceName>.systemTriggeredEvents`

To allow the subscription to a resource, a `resource` string has to be pushed to the `<resourceName>.systemTriggeredEvent` array.

```
{
      "systemTriggeredEvents": [
        "resource"
      ]
    }
```

READ_resource

Location: `<resourceName>.endpoints.resource`

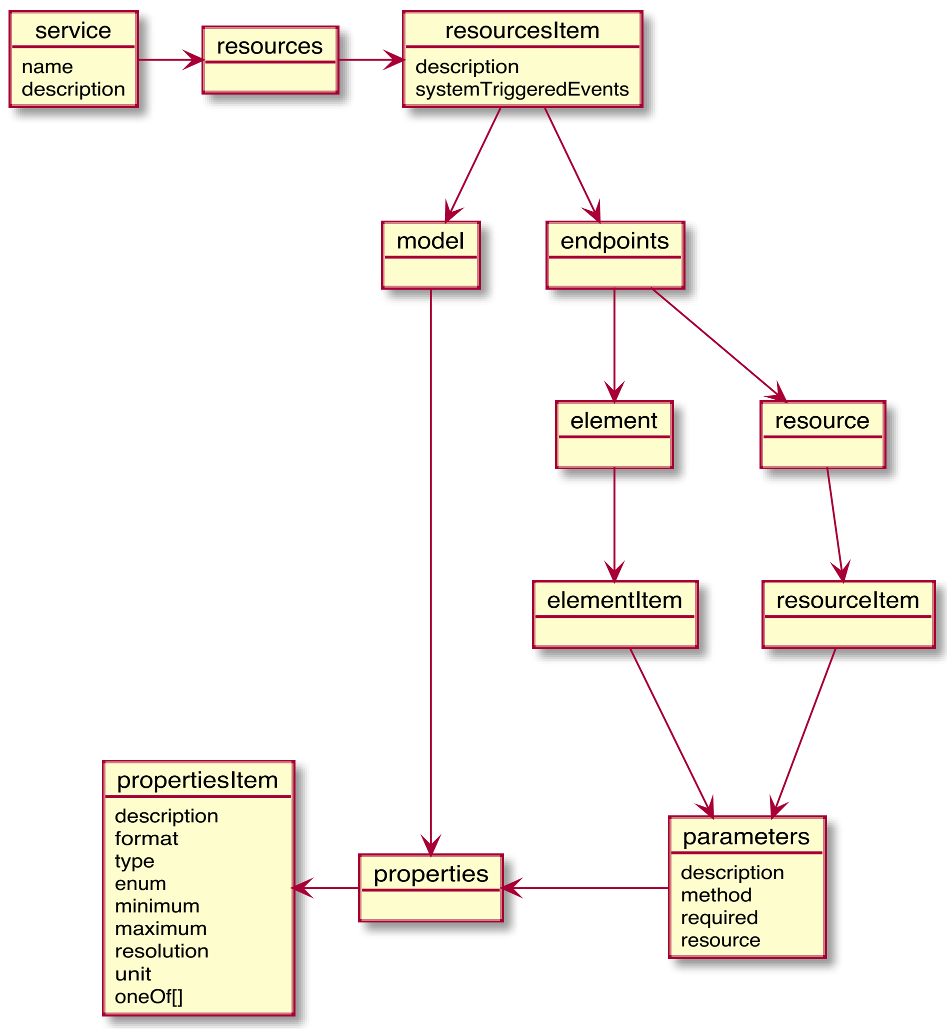
To allow the reading the entire resource, a `get` property has to exist in

`<resourceName>.endpoints.resource` .

```
{
  "resource": {
    "get": {
    }
  }
}
```

4.2 Schema-Structure

To create a service that satisfies the restrictions of the ViWi protocol, it has to follow a suitable schema. For this purpose a JSON schema exists that describes how a service should be built to ensure all features described by the ViWi protocol can be used. Apart from the root, each JSON-Object of the schema is nested within another one. The following diagram depicts the hierarchy of these JSON-Objects to illustrate on which level a certain JSON Object exists and the associated properties. In general the name of the node corresponds to the name of the JSON-Object. However, node names ending by the suffix ‘Item’ represent JSON-Objects that do not have a predefined name but can have a custom one. Moreover, these objects may also have multiple instances that can be identified by their names. In addition to the graph, the table below describes the function of each JSON Object and its included properties that are needed to describe the JSON-Object. So while the graph illustrates the schema structure, the table explains the purpose of each JSON-Object and property.



JSON Object	Description	Properties
element	Contains all available access operations of the individual elements.	
	A concrete instance of an element needed to define the available access operations for the individual element. Only the following element requests are allowed: *get:* Retrieve a representation of the addressed member of the collection, expressed in an	

elementItem	appropriate Internet media type. *post:* Update element. The updated element's URI is assigned automatically and is returned by the operation. *put:* Create new element in collection by sending the entire entity. The new element's URI is assigned accordingly. *delete:* Delete the referred entity of the collection (or its attributes).	
endpoints	Contains the available access operations for the corresponding resource and all its elements. An endpoint may only contain the JSON-Objects 'element' or 'resource'.	
parameters	Contains all properties which can be modified by the corresponding request	**description:** explains the purpose of the corresponding request **method:** shows the request kind. This value should correspond to the name of the element **required:** An array containing all property names whose references have to be updated by the corresponding request.The name must be conform to the name of the referenced property **resource:** the name of the corresponding resource
	A collection of properties, which all together represent an XObject of the corresponding resource. The number of properties may vary, but each XObject must have at least the following properties: *id:* A property which represents the unique identifier of the XObject. This property has to be of type string and the format must	

properties	be of type ‘Custom Format / Regex’ *name:* the name of the object. This property must be of type string or enum *uri:* A link to the corresponding instance of the resource. This property has to be of type string and the format must be of type ‘Custom Format / Regex’ The property names ‘id’, ‘name’ and ‘uri’ are always lowercase.	
propertiesItem	A property of an XObject.	**description:** A text that describes the property. **format:** the format of the property. This value may only be set if type represents a string. **type:** the datatype of the property. **enum:** An array which contains allowed values for the property. This value may only be set if type represents an enum. **minimum:** The minimum value. This value may only be set if type represents a numerical value like ‘integer’ or ‘number’. **maximum:** The maximum value. This value may only be set if type represents a numerical value like ‘integer’ or ‘number’. **resolution:** The step size for a numerical value. Each numerical value of the corresponding type has to be divisible by this value. This value may only be set if type represents a numerical value like ‘integer’ or ‘number’. **unit:** The unit of the value. This value may only be set if type represents a numerical value like ‘integer’ or ‘number’. **oneOf[]:** An array containing

		references to other XObjects. Each object within the array must have the attribute '#refs' which holds the referenced name of the object. This value may only be set if type represents an object.
resource	Contains all available access operations of the individual elements.	
resourceItem	A concrete instance of a resource needed to define the available access operations for the individual resource. Only the following element requests are allowed: *get:* Retrieve a representation of the addressed member of the collection, expressed in an appropriate Internet media type. *post:* Update element. The updated element's URI is assigned automatically and is returned by the operation. *put:* Create new element in collection by sending the entire entity. The new element's URI is assigned accordingly. *delete:* Delete the referred entity of the collection (or its attributes).	
resources	A collection of the individual resources instances.	
resourcesItem	A concrete resources instance. Its name has to be written in plural form.	**description:** A text that describes the resource. **systemTriggeredEvents:** An array that lists whether it is possible to subscribe to resources or elements. This property will be set when using the access operations SUB_element or SUB_resource

service	The service	**name:** The service name **description:** A text that describes the service

4.3 Cross-origin resource sharing (CORS)

Cross-origin resource sharing (CORS) is a mechanism that allows restricted resources (e.g. fonts, JavaScript, etc.) on a web page to be requested from another domain outside the domain from which the resource originated.

A web page may freely embed images, stylesheets, scripts, iframes, videos and some plugin content from any other domain. However embedded web fonts and AJAX(XMLHttpRequest) requests have traditionally been limited to accessing the same domain as the parent web page (as per the same-origin security policy). “Cross-domain” AJAX requests are forbidden by default because of their ability to perform advanced requests (POST, PUT, DELETE and other types of HTTP requests, along with specifying custom HTTP headers) that introduce many cross-site scripting security issues.

CORS defines a way in which a browser and server can interact to safely determine whether or not to allow the cross-origin request. It allows for more freedom and functionality than purely same-origin requests, but is more secure than simply allowing all cross-origin requests. It is a recommended standard of the W3C.

To allow clients (e.g. browsers) which are following these guidelines access to the service, the service must use CORS.

Source: https://en.wikipedia.org/wiki/Cross-origin_resource_sharing

Web Server requirements:

- The server must be configured to allow CORS (http://www.w3.org/TR/cors/)
- The server must send the CORS header Access-Control-Allow-Origin *
- The server must send the CORS header Access-Control-Expose-Headers ‘location’
- The server must send the CORS header Access-Control-Allow-Credentials
- The server must send the CORS header Access-Control-Allow-Methods: GET,HEAD,PUT,PATCH,POST,DELETE
- The server must send the CORS header Access-Control-Allow-Headers: and use as its value the headers it received in the request header Access-Control-Request-Header

For more in-depth content about CORS it’s also recommended to visit the Mozilla Developer Network CORS-Section [6]

5 FAQ

Are multiple HTTP status codes possible?

No, per response only one HTTP status code is possible [3].

How are binary data transmitted according to the ViWi protocol?

Binary data is transmitted with regular HTTP headers according to its MIME-type.

How many websocket connections are possible?

Exactly one websocket connection per client is possible.

Why are PUT requests not used?

The ViWi protocol generally only supports PUT requests on root and element level. However, PUT requests are not used on element level, because they would have the same effect as a POST request on resource level apart from the fact that a POST request on resource level creates the ID automatically, whereas a PUT request on element level would not create an ID such that it would have had to be defined by the user. To ensure the IDs are always valid, POST request on resource level should be preferred over PUT requests on element level.

A PUT request on root level creates new services. Since the root level cannot be accessed by services, this PUT request also cannot be called when creating services.

How does a ViWi server manage data?

This section describes how ViWi objects are handled on client and service side. Due to the complexity of this topic, other behaviors are also possible if the rules have been described in the service definition and were coordinated with the RSI Architecture team.

General rules: A client may never delete an object that was created by a service or another client. A service may never delete an object that has at least one subscription to it. A service must implement a garbage collection that manages all existing objects inside the service following the rules defined in “Objects created by a service” and “Object created by a client”.

1.1 Objects created by a service

Each resource is classified into one of three lifecycle types, which means that every object created inside the resource by the service follows one of the three following lifecycle rules. This definition is done by the service and is unknown to the client during runtime. As a general rule an object created by a service always has a minimum length of survival, which is 10 seconds (note: this is subject to change).

1.1.1 Objects that exist forever

A service may create an object (e.g. at startup) that exists forever or until the service is terminated. These are objects that always have to exist in order for the service to fulfil its purpose.

Example: A systemDescriptions resource of any ViWi service may have an active systemDescriptions instance at all times in order to publish system description to the client.

1.1.2 Objects that exist as long as a physical equivalent exists

A service may create an objects that exists as long as a physical equivalent exists, which means that the objects represents a physically existing element. After the the physical representation does not exist any more the service must delete the object unless it has at least one subscription to it. If the object cannot be deleted due to a subscription, the service must try to delete the object in a recurring manner.

Example 1: A traffic incident is decoded by a traffic receiver that is attached to a ViWi traffic service. Whenever a traffic incident exists, the service creates a corresponding incidents object. The incidents object survives as long as the physical traffic incident exists.

Example 2: A personalization layer that is attached to a navigation service holds a number of favorite destinations. As long as the personalization layer holds the favorites, the navigation service will hold corresponding favorites objects.

1.1.3 Objects that exist as long as a client is using them

A service may create an object that exists to be used by one or more clients. After the minimum length of survival the service must delete the object right away unless at least one client is subscribed to it. If the object cannot be deleted due to a subscription, the service must try to delete the object in a recurring manner.

1.2 Objects created by a client

If a client creates an object, the client must delete the object as soon it is not needed any more. If the object has at least one subscription to it, the deletion will fail on service side. If this happens the client must leave the object to the service’s garbage collection. The service must grant an object created by a client a minimum survival length, which is 10 seconds (note: this is subject to change). After that minimum survival length the service must delete the object unless it has at least one subscription to it. If the object cannot be deleted due to a subscription, the service must try to delete the object in a recurring manner.

When is it necessary to escape control characters?

According to the common JSON specification [4] the most control characters exist in JSON must be escaped. For detailed information see the “string” section on http://www.json.org/.

Is it possible to send JSON as a property value

It is possible to send JSON as a value by encoding it as a string. Note that you need to escape certain JSON properties in the string representation. Please refer to the JSON specification for details. [4]

Encrypted communication - technical layer

Please Note: *This subject is currently under debate and should be understood as a suggestion and not as a final specification!*

Which participant decides if the communication is cryptographically secured?

Definitions:

- Server: The ViWi server providing the ViWi services
- Internal client: A ViWi client which is running on the same ECU as the ViWi server
- Vehicle client: A ViWi client which is running on a different ECU inside the same vehicle
- Vehicle-external client: A ViWi client which is running outside the vehicle such as an app on a smart phone

For the communication security concept distinctions are made based on the type of the communication client [5]:

1. For internal clients the communication is not cryptographically secured.
2. For vehicle clients the communication is secured using pre-shared keys which are distributed during the vehicle production process.
3. For vehicle-external clients the communication is secured using certificates which are established through a pairing process.

In order to determine wether a client is an internal, vehicle or vehicle-external client with regards to a ViWi server, the respective IP adresses may be used:

First, let us have a look at how a client distinguishes which cryptography method is used: A client establishing a direct connection to a server compares its own IP address to the server’s IP address. If both IP addresses are equal, the client is internal and the communication is not encrypted. If the server’s as well as the client’s IP addresses are inside the vehicle’s IP range but are different, the client is a vehicle client and the communication is encrypted using pre-shared keys. If the server’s IP address is inside the vehicle’s IP range and the client’s IP address is outside, the client is an external client and the communication is secured using certificates.

The server may distinguish which cryptography method is used analogously: It compares its own

IP address with the client's incoming request's IP address. If both IP addresses are equal, the client is an internal client and the communication remains unencrypted. If the client's IP address is inside the vehicle's IP range, pre-shared keys are used. If the client's IP is outside the vehicle's IP range, the client is a vehicle-external client and certificates are used.

Following the described process, both participants, the client as well as the server, can determine which encryption method is used.

ViWi communication inside a ECU

If the ViWi client is running on the same ECU as the ViWi server (internal client), no cryptographically secured channel (HTTPS) is needed [5]. Instead, HTTP requests are issued.

Error Handling

Error handling uses status codes to communicate what happened. Status codes are currently defined in three documents:

1. ViWi specification [1]
2. The Navigation Specific Status Code Handling document
3. Service Concept (i.e. Places Concept, MapRenderer Concept etc.)

Each document may overwrite specific codes in the previous document. For a comprehensive description of all available HTTP status codes, see the ViWi documentation version 1.6.0 [1]. Each status code may have a use case specific meaning, which is described in the corresponding "Service Concept" respectively.

However, since some error codes in the ViWi documentation version 1.6.0 are not conform to common HTTP status codes, the error codes 42, 1337, 1895 and 31415 may not be used anymore. Instead, the error codes defined in the ViWi documentation version 1.7.0 have to be used when necessary, such that the following additional HTTP error codes are valid:

Code	Meaning
400	syntax error
403	access denied (token invalid or expired)
404	subscription uri invalid
503	maximum number of event subscriptions reached

The client's behavior on errors depends on the specific implementation. However, in some cases exceptions have to be handled in a predefined manner to ensure a consistent behavior for each service. To get an overview of these cases, the following table lists all error scenarios that need to be handled in a specific way and the corresponding server responses.

Error Description	Server Response
The client sends a malformed request	HTTP 400 (bad request)
The client unsubscribes from a non-existing subscription	HTTP 404 (subscription URI invalid)
The client requests a function that is not provided by the server	HTTP 501 (not implemented)

If no error is encountered, the server sends a valid response and a result is send back.

How should UUIDs be created?

Since version 1.6.0 of the ViWi protocol is very ambiguous in this context, several additional rules exist that have to be considered when defining how UUIDs are created by a service. So at first it is necessary to ensure all UUIDs are unique and no services have the same UUID. Hence, the calculation of a UUID has to consider the URI of the corresponding service. Moreover, it has been determined that UUIDs must be written in lower case. The calculation of the UUIDs also has to be described in the service definition to keep the corresponding algorithm comprehensible and identify possible errors quickly.

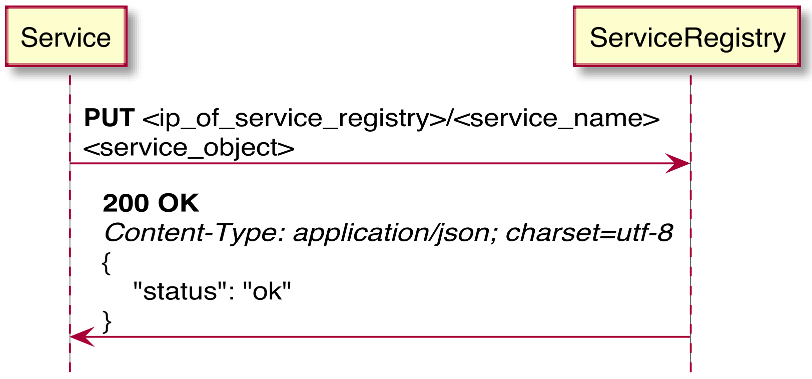
How should an URI be represented within a serviceObject?

The URI should point to the actual service. Since it is necessary to distinguish between local and remote clients when defining whether encrypted connection has to be used, references in the serviceObject always have to be represented by absolute paths. However, it has not yet been determined how to inform the service about the IP address that is needed for building the absolute URI.

What if the service is unstable and breaks off? What is the reconnect strategy, especially for subscriptions?

If a service starts up, it registers itself by creating a serviceObject. For the registration, the service uses the default way by sending a HTTP PUT to the service registry.

An example of this could look like this:


Service
ServiceRegistry
PUT <ip_of_service_registry>/<service_name>
<service_object>
200 OK
Content-Type: application/json; charset=utf-8
{
"status": "ok"
}

`PUT` on `http://localhost:9999/media` with the request body:

```
{
      "id": {
        "type": "string",
        "description": "service id (if sent with registration request, it will be ig
        "format": "uuid"
      },
      "name": {
        "type": "string",
        "description": "service name"
      },
      "uri": {
        "type": "string",
        "description": "service uri (the desired <service_path> relative to the root
        "format": "uri"
      },
      "description": {
        "type": "string",
        "description": "human readable description of the micro service"
      },
      "port": {
        "type": "integer",
        "description": "TCP port the service is running on"
      },
      "serviceCategories": {
        "type": "array",
        "items": {
          "type": "string",
          "description": "predefined key words"
        }
      },
      "privileges":  {
        "type": "array",
        "items": {
          "type": "string",
          "description": "relative path to the service or resource"
        }
      },
      "versions":  {
        "type": "array",
        "items": {
          "type": "string",
          "description": "supported version in semVer notation"
        }
      }
    }
```

A successful operation will lead to a simple response, that only indicates that the registration happened successfully:

```
{
        "status": "ok"
    }
```

Now the registry can create a subscription on the root level of the new service with `$limit=0`. This enables the registry to supervise the service. To give the client the ability to observe the service status too, the client must subscribe itself to the newly created serviceObject.

If the service crashes, the WebSocket stream, that was created by the subscription to the service, throws an error and registry knows that the appropriate service are not longer available. The registry now deletes the corresponding serviceObject, which informs the client too, that the service is gone. The client should now try to reconnect to the service and reestablish all previous subscriptions. That doesn't unbind the service implementers from the requirement to provide stable and non-crashing services. Deviations for each case including the behavior have to be agreed separately and have to be described in the technical concept document.

Subscribe on service & root: When does paging information come?

Paging information must always be provided with a GET request for a resource, service or root.

Representation of references within a POST-Request

As mentioned above, POST requests contain references as absolute URIs. The server answers with an XObject if no additional `$expand` -level is defined, since the default `$expand` level is 0. For more in-depth information about `$expand` -resolution visit the expand-section.

How should connections be implemented?

Connections between client and server have to be established by the client and may only be canceled if a timeout or a server error (HTTP 5xx) occurs. To ensure timeouts are generally handled equally by services, it has been determined that keep-alive messages have to be used as defined in HTTP 1.1 [7]. Moreover, it also has been defined that redirects are only allowed in a temporary way and must have a predefined timeout.

6 References

[1]	ViWi protocol definition Version 1.6	
[2]	W3C ViWi protocol definition submission request	(last visit 22.08.2017)
[3]	IETF RFC 2616: Hypertext Transfer Protocol – HTTP/1.1 (Chapter 6.1 Status-Line)	(last visit 22.08.2017)
[4]	ECMA Standard 404: The JSON Data Interchange Format	(last visit 22.08.2017)
[5]	ViWi Schutzkonzept (Chapter 2.1)	(last visit 22.08.2017)
[6]	MDN: Cross-Origin Resource Sharing (CORS)	(last visit 04.09.2017)
[7]	MDN web docs definition Keep-Alive	(last visit 05.09.2017)

7 Appendix

7.1 FAQ – Overview

- Erstellen eines Beispiels für Erstellen eines Objektes (durch client)
- Erstellen eines Beispiels für Löschen eines Objektes (durch client)
- Sind mehrere http Statuscodes möglich?
- Sequenzdiagramm zur Visualisierung von GET request erstellen
- Client-Server-Interaktion 4-gliedrig beschreiben
- Strukturanpassung des rsi_tutorials - Referenzierung der Fragen/Todos anpassen
- Strukturanpassung des rsi_tutorials - Bezug zu “Excel-Operationen” wieder entfernen
- FAQ erweitern mit Fragen vom 22.06.2017 - Paging erklären
- FAQ erweitern mit Fragen vom 22.06.2017 - Wie werden Binärdaten nach ViWi/HTTP protokollgerecht transportiert?
- FAQ erweitern mit Fragen vom 22.06.2017 - Wie viele WS Verbindugen sind möglich?
- FAQ erweitern mit Fragen vom 22.06.2017 - Warum wird PUT nicht verwendet?
- FAQ erweitern mit Fragen vom 22.06.2017 - Warum wird PUT nicht verwendet?
- FAQ erweitern mit Fragen vom 22.06.2017 - An welchen Stellen müssen in einem RSI Request /Response “Steuerzeichen” wie z.B. \t \n escaped werden?
- Wie werden nicht gesetzte Felder repräsentiert? Null oder UNDEFINED?
- ViWi: Wie ist der Unterschied UNDEFINED und unbekannt?
- Welche Form der UUID muss der Service akzeptieren?
- Was passiert, wenn ein Element gelöscht wird, auf das sich ein Client subscribed hat?
- Was passiert, wenn ein Element gelöscht wird, auf das sich ein Client subscribed hat?
- Dokumentation der ViWi JSON Struktur und der dort verwendeten Parameter erstellen
- Feedback-1.0.0-draft: Access Rights umbenennen in Access Operations sowie Zweck und Spezifikationsort hinzufügen
- $expand-Verhalten in verschiedenen Situationen
- Feedback-1.0.0-draft: $expand um root-, service- und resource-level erweitern
- Feedback-1.0.0-draft: $expand um Erläuterung erweitern
- Wann triggern Updates auf unterschiedlichen $expand-Levels?
- Zusammenspiel $fields und $expand
- Reihenfolge der Auswertung von Sortierung, Filter und Paging bei Subscriptions
- Updates bei Änderungen der Länge der von Listen
- Wie sehen Referenzen aus, die durch GET/SUBSCRIBE an den Client geliefert werden?
- Was passiert bei einer Subscription mit Paging, wenn in der Ressource nicht mehr genügend Elemente drin sind
- Was ist mit Connections und Redirects? Connection-Abbrüche? KeepAlive/StillAlive?
- Was ist, wenn der Service instabil ist und wegbricht? Wie ist dann die reconnect-Strategie, insbesondere für Subscriptions?

- Unter welchen Umständen wird 31415 at least one field name unknown zurückgeliefert (siehe Viwi 1.6.0 Kapitel 1.13)?
- Wie soll ein Client auf 1895 maximum number of event subscriptions reached reagieren(siehe Viwi 1.6.0 Kapitel 1.13), ist diese maximum number definiert?
- Wird eine Freetext Search nur auf den Elementen des Level 0 der Objekte ausgeführt oder auf allen Werten die bei einem GET mit dem Expand Level geliefert werden?
- Was soll der Service antworten wenn auf eine Subscription, die bereits unsubscribed wurde, erneut ein unsubscribe aufgerufen wird?
- Abschnitt zu Filtern von Properties
- Feedback-1.0.0-draft: subscribe auf service & root: Wann kommt paging Information?
- Feedback-1.0.0-draft: Hinweis auf Verwendbarkeit der Fehlercodes aus 1.7.0 hinzufügen
- Feedback-1.0.0-draft: Schema Schena Grafik näher erklären (inhaltliche Bedeutung (vllt auf Tabelle hinweisen)Pfeile, Relationen, Kardinalitäen,)
- Feedback-1.0.0-draft: Schema Schema Tabelle anpassen
- Feedback-1.0.0-draft: $q (Freetextsearch) Abschnitt anpassen
- Wie sehen Referenzen aus, die durch POST an den Service geliefert werden?
- CORS-Sektion aus ViWi 1.8.0 in RSI-Tutorial 1.1.0-draft übernehmen
- 20170912-02: Entspricht die Freetext- oder Property-Search einem “contains” oder einem “equals”
- 20170912-02: Entspricht die Freetext- oder Property-Search einem “contains” oder einem “equals”
- 20170912-01: Einarbeitung der Tabelleninhalte in den Request-Types Abschnitt

8 Changelog

Version 1.2.0 (2017-09-05):

Features:

- VWEERA-175: Subscribe on service & root: When does paging information come?
- Describe search text handling for free text searches
- Describe search text handling for property searches
- Describe returned status codes and properties for the individual request types

Version 1.1.0 (2017-09-05):

Features:

- VWEERA-159: Enhance definition of systemTriggeredEvents and endpoints
- VWEERA-167: How should connections be implemented?
- VWEERA-168: What does relative uris in a serviceObject mean?
- VWEERA-169: What if the service is unstable and breaks off? What is the reconnect strategy, especially for subscriptions?
- VWEERA-172: Free-text search on level 0 elements
- VWEERA-173: Double unsubscribe on a subscription
- VWEERA-195: Representation of references within a POST-Request
- VWEERA-196: CORS-Sektion aus ViWi 1.8.0 in RSI-Tutorial 1.1.0-draft übernehmen
- VWEERA-197: Add changelog section