X-Road Message Protocol for REST

Table of Contents

- Table of Contents
- 1 Introduction
  - 1.1 Overview
- 2 Definitions
  - 2.1 Key Words
  - 2.2 X-Road Terminology
  - 2.3 Versioning
  - 2.4 References
- 3 Scope
  - 3.1 Scope and Requirements
  - 3.2 Objectives
- 4 Message Format
  - 4.1 REST Interface
  - 4.2 URI Sanitation
  - 4.3 Use of HTTP Headers
  - 4.4 Use of Query Parameters
  - 4.5 Error handling
- 5 Services
  - 5.1 Describing Services
  - 5.2 Service Implementation
- 6 Examples
  - 6.1 General
  - 6.2 GET Request and Response
  - 6.3 PUT Request and Response
  - 6.4 POST Request and Response
  - 6.5 DELETE Request and Response
  - 6.6 POST Request with Attachments and Response
- Appendix 1 Example Service Definition
- Appendix 2 Document Change History

1 Introduction

1.1 Overview

Representational State Transfer [REST] is an architectural style that defines a set of constraints to be used for creating web services. Web services that conform to the [REST] architectural style, or RESTful web services, provide interoperability between computer systems on the Internet. REST-compliant web services allow the requesting systems to access and manipulate textual representations of web resources by using a uniform and predefined set of stateless operations.

This document describes the X-Road Message Protocol for [REST] version 0.1.0. The protocol is used in X-Road infrastructure between information systems and X-Road security servers to consume and produce REST services.

2 Definitions

2.1 Key Words
The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document (in uppercase, as shown) are to be interpreted as described in [RFC2119].

2.2 X-Road Terminology

The X-Road specific terminology such as instance, member class, member and subsystem are described in the document [XROAD-TERMS].

2.3 Versioning

The X-Road Message Protocol for REST uses [SEMANTIC-VERSIONING] rules. After the initial development phase (0.x) there must be a strongly justified reason for amending or updating the protocol. Especially new major versions of the protocol SHOULD be extremely rare.

2.4 References

[RFC2119] https://www.ietf.org/rfc/rfc2119.txt
[URL-ENCODE] https://en.wikipedia.org/wiki/Percent-encoding
[OPENAPI-INITIATIVE] https://www.openapis.org/
[XROAD-TERMS] https://github.com/nordic-institute/X-Road/blob/develop/doc/terms_x-road_docs.md
[SEMANTIC-VERSIONING] https://semver.org/
[SSRF] https://www.owasp.org/index.php/Server_Side_Request_Forgery
[REST-BEST-PRACTISES] https://github.com/WhiteHouse/api-standards

3 Scope

3.1 Scope and Requirements

- Only synchronous request-response messages SHALL be supported. Asynchronous or one-way operations SHALL NOT be possible.
- Any payload type over REST SHALL be supported. The payload MUST NOT be restricted to just JSON or XML.
- The protocol SHALL support any message size. In practice the message size is limited by the security server's memory and disk sizes. For security reasons it is RECOMMENDED to introduce a configurable "maximum message size" parameter in the security server implementation.

3.2 Objectives

- The objective of the draft versions (0.x) of the protocol is to receive feedback from the audience.
- The objective of the protocol version 1.x is to be future proof so that no protocol level changes are required for X-Road version 7.
- The protocol must have a mechanism for updating it so that a new protocol version can be published when necessary.
4 Message Format

4.1 REST Interface

HTTP version 1.1 is used by the protocol as described in [RFC2616]. The service to be called is encoded as part of the HTTP/HTTPS request. Here is the generic form of the REST service call.

```
{http-request-method} /rest/{protocol-version}/{consumer-subsystem}/
    {provider-subsystem}/{serviceId}
```

- `{http-request-method}` can be one of the request methods defined in [RFC7231]. For example GET, POST, PUT and DELETE.
- `rest` is a constant prefix defined by the protocol (with it the security server implementation can easily distinguish between SOAP and REST calls).
- `{protocol-version}` is the version of the message protocol for REST. Use v1 for the initial version.
- `{consumer-subsystem}` is the subsystem/REST endpoint assigned to consumer security server's subsystem.
- `{provider-subsystem}` is the subsystem to be called on the provider security server.
- `{serviceId}` is the service path.

Here is a practical example of an X-Road REST call.

```
GET /rest/v1/INSTANCE1/CLASS1/MEMBER1/SUBSYSTEM1/INSTANCE2/CLASS2/MEMBER2
    /SUBSYSTEM2/v1/bar/zyggy?quu=1
```

Secure REST services MUST only provide HTTPS endpoints. This protects authentication credentials in transit, for example passwords, API keys or JSON Web Tokens. It also allows clients to authenticate the service and guarantees integrity of the transmitted data. It is RECOMMENDED to use mutually authenticated client-side certificates to provide additional protection for highly privileged web services.

4.2 URI Sanitation

The REST URIs are composed of different parts (e.g. INSTANCE/CLASS/MEMBER/SUBSYSTEM) and some of them may contain characters that can not be used in URIs directly. The URI syntax is described in [RFC3986]. Because of this the consumer information system MUST [URL-ENCODE] the request.

On the security server side the incoming request URIs MUST be strictly validated. Input strings from the user can't be trusted. Lengths of the strings need to be checked and maximum length or the request URI needs to be limited. Although the URI standard does not specify a maximum size of the URL, most clients enforce an arbitrary limit of 2000 characters. The security server implementation MAY do this as well. Sending data that is difficult to express in a hierarchical manner, and especially data that is larger than this 2000 character limit, should be transmitted in the body of the request.

The REST URI parsing scenario is particularly vulnerable to [HPPP] (HTTP Parameter Pollution) and [SSRF] (Server-Side Request Forgery) attacks. The security server SHOULD explicitly be prepared for them.

4.3 Use of HTTP Headers

The use of HTTP headers in X-Road REST service calls is OPTIONAL. The most common header types and their operation are described next.

**Content-Type header**

- With REST messages that include the request body it is RECOMMENDED that the content's media type is indicated with this header. Additionally it is RECOMMENDED to use the charset parameter to indicate the character encoding used the REST message.
- The REST messages originating from the security server (e.g. error messages) MUST include the header and indicate the content's type and character encoding with it.
Content-Type examples

- Content-Type: application/json; charset=utf-8
- Content-Type: multipart/form-data; boundary=something

Accept header

- With Accept header it is RECOMMENDED that the client advertises the content types it is able to understand in the response message.

Accept examples

- Accept: application/xml

Optional X-Road HTTP headers

- X-Road-ID (if ID is not provided, it SHALL be generated by the consumer security server (e.g. [UUID]))
- X-Road-UserID
- X-Road-Issue

Optional X-Road header examples

- X-Road-ID: fa2e18a5-c2cb-4d09-b994-f57727f7c3fb
- X-Road-UserID: A872323190
- X-Road-Issue: MT324223MSD

User defined HTTP headers

- User defined HTTP headers MUST be passed to recipient unmodified by X-Road security server.

User defined HTTP headers examples

- X-Powered-By: PHP/5.2.17
- X-Pingback: https://example.com/xmlrpc.php

4.4 Use of Query Parameters

The use of query parameters in X-Road REST service calls is OPTIONAL. The query parameters MUST be passed unmodified through the X-Road security servers to the provider service.

4.5 Error handling

In normal situations the requests reach the provider service and a response is returned to the consumer information system. However, the security server may encounter technical errors and in these cases it must respond in a predictable manner. When technical error occurs, the security server MUST use HTTP status codes as defined in [RFC7231] to communicate it back to the consumer information system.

[RFC7231] defines over 70 HTTP status codes. Most of the developers do not have them memorized so they have to go to the Internet and look them up. To make it simpler for the developers the X-Road Message Protocol for REST uses only a small subset of HTTP status codes. When it is boiled down, there are really only 3 categories of errors between the client and the security server.

1. Everything worked (in security server's perspective, the service may still return an error)
2. The client did something wrong
3. The security server did something wrong

We map these error cases to the HTTP status codes and response bodies.
1. The status code, response body and HTTP headers are returned by the service.
2. 400 - Bad Request. The response body and HTTP headers are returned by the security server.
3. 500 - Internal Server Error. The response body and HTTP headers are returned by the security server.

In cases 2 and 3 this is the response body format.

```json
{
    "code": 500,
    "type": "DatabaseError",
    "message": "Error accessing database"
}
```

The security server implementation MUST respect the Accept header specified by the consumer information system and try to respond with suggested content type. Additionally the security server MUST include Content-Type header in the response to indicate the media type of the response.

5 Services

5.1 Describing Services

The services provided by the information systems are described with [OPENAPI2]. It is a specification for machine-readable interface files for describing, producing, consuming, and visualizing RESTful web services. The OpenAPI Specification is a community-driven open specification within the [OPENAPI-INITIATIVE], a Linux Foundation Collaborative Project.

At the time of writing (2018), a new version of the OpenAPI specification has just come out. 4 years after the [OPENAPI2] the [OPENAPI3] adds a simpler and more reusable structure. Other features include content negotiation support, enhanced security definitions, updated parameter types, OpenAPI links and support for describing callbacks. Because the tool support for [OPENAPI3] is still somewhat immature the X-Road Message Protocol for REST uses the older [OPENAPI2] instead.

An example [OPENAPI2] service definition file is listed in Appendix 1. This service is used in the examples of the next chapter.

5.2 Service Implementation

The protocol RECOMMENDS that the [REST-BEST-PRACTISES] are followed when implementing [REST] services. However, the requirements are not strict and also non-compliant services are supported by the protocol.

6 Examples

6.1 General

The pet store service used in the following examples has an [OPENAPI2] service description file available in appendix 6.1. The most important aspects of the service are described in the text but for more details please refer to the aforementioned service description file.

6.2 GET Request and Response

REQUEST

<table>
<thead>
<tr>
<th>Service</th>
<th>Method</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>/pet/{petId}</td>
<td>GET</td>
<td>Finds pet by ID</td>
<td>* petId - ID of pet to return</td>
</tr>
</tbody>
</table>

Service called directly
curl -X GET "https://petstore.swagger.io/v2/pet/1124" -H "accept: application/json"

Service called through X-Road

curl -X GET "https://{securityserver}/rest/v1/{consumer-subsystem}/(provider-subsystem)/pet/1124" -H "accept: application/json"

Service response

```json
{
   "id": 1124,
   "name": "Siddu",
   "photoUrls": [],
   "tags": [],
   "status": "Offline"
}
```

Service response code

200

Service response headers

`content-type: application/json`

### 6.3 PUT Request and Response

#### REQUEST

<table>
<thead>
<tr>
<th>Service</th>
<th>Method</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>/pet</td>
<td>PUT</td>
<td>Update an existing pet</td>
<td>• body - Pet object that needs to be added to the store</td>
</tr>
</tbody>
</table>

Service called directly

curl -X PUT "https://petstore.swagger.io/v2/pet" -H "accept: application/json" -H "Content-Type: application/json" -d "{"id": 0, "category": { "id": 0, "name": "string" }, "name": "doggie", "photoUrls": [ "string" ], "tags": [ { "id": 0, "name": "string" } ], "status": "available"}"

Service called through X-Road
curl -X PUT "https://{securityserver}/rest/v1/{consumer-subsystem}/(provider-subsystem)/pet" -H "accept: application/json" -H "Content-Type: application/json" -d "{{"id": 0, "category": { "id": 0, "name": "string" }, "name": "doggie", "photoUrls": [ "string" ], "tags": [ { "id": 0, "name": "string" } ], "status": "available"}}"

Service response

{
    "id": 5657082955040009,
    "category": {
        "id": 0,
        "name": "string"
    },
    "name": "doggie",
    "photoUrls": [
        "string"
    ],
    "tags": [
        {
            "id": 0,
            "name": "string"
        }
    ],
    "status": "available"
}

Service response code

200

Service response headers

content-type: application/json

6.4 POST Request and Response

REQUEST

<table>
<thead>
<tr>
<th>Service</th>
<th>Method</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>/pet</td>
<td>POST</td>
<td>Add a new pet to the store</td>
<td>- body - Pet object that needs to be added to the store</td>
</tr>
</tbody>
</table>

Service called directly
curl -X POST "https://petstore.swagger.io/v2/pet" -H "accept: application/json" -H "Content-Type: application/json" -d "{"id": 0, "category": { "id": 0, "name": "string" }, "name": "doggie", "photoUrls": [ "string" ], "tags": [ { "id": 0, "name": "string" } ], "status": "available"}"

Service called through X-Road

curl -X POST "https://{securityserver}/rest/v1/{consumer-subsystem}/{provider-subsystem}/pet" -H "accept: application/json" -H "Content-Type: application/json" -d "{"id": 0, "category": { "id": 0, "name": "string" }, "name": "doggie", "photoUrls": [ "string" ], "tags": [ { "id": 0, "name": "string" } ], "status": "available"}"

Service response

{
   "id": 5657082955040122,
   "category": {
      "id": 0,
      "name": "string"
   },
   "name": "doggie",
   "photoUrls": [
      "string"
   ],
   "tags": [
      {
         "id": 0,
         "name": "string"
      }
   ],
   "status": "available"
}

Service response code

200

Service response headers

content-type: application/json

6.5 DELETE Request and Response
### DELETE Request with Attachment and Response

#### REQUEST

<table>
<thead>
<tr>
<th>Service</th>
<th>Method</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>/pet/{petId}</td>
<td>DELETE</td>
<td>Deletes a pet</td>
<td>petId - Pet id to delete</td>
</tr>
</tbody>
</table>

#### Service called directly

```
curl -X DELETE "https://petstore.swagger.io/v2/pet/1124" -H "accept: application/json"
```

#### Service called through X-Road

```
curl -X DELETE "https://{securityserver}/rest/v1/{consumer-subsystem}/(provider-subsystem)/pet/1124" -H "accept: application/json"
```

#### Service response

#### Service response code

200

#### Service response headers

```
content-type: application/json
```

### 6.6 POST Request with Attachments and Response

#### REQUEST

<table>
<thead>
<tr>
<th>Service</th>
<th>Method</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>/pet/{petId}/uploadImage</td>
<td>POST</td>
<td>uploads an image</td>
<td>petId - ID of pet to update additionalMetadata - Additional data to pass to server file - file to upload</td>
</tr>
</tbody>
</table>

#### Service called directly

```
curl -X POST "https://petstore.swagger.io/v2/pet/1124/uploadImage" -H "accept: application/json" -H "Content-Type: multipart/form-data" -F "file=@A-fluffy-cat-looking-funny-surprised-or-concerned.jpg;type=image/jpeg"
```

#### Service called through X-Road


curl -X POST "https://{securityserver}/rest/v1/{consumer-subsystem}/
(provider-subsystem)/pet/1124/uploadImage" -H "accept: application/json" -
H "Content-Type: multipart/form-data" -F "file=@A-fluffy-cat-looking-funny-
surprised-or-concerned.jpg;type=image/jpeg"

Service response

{
    "code": 200,
    "type": "unknown",
    "message": "additionalMetadata: null\nFile uploaded to ./A-fluffy-cat-
looking-funny-surprised-or-concerned.jpg, 170025 bytes"
}

Service response code

200

Service response headers

content-type: application/json

Appendix 1 Example Service Definition

swagger: "2.0"
info:
    description: "This is a sample server Petstore server. You can find out
more about Swagger at [http://swagger.io](http://swagger.io) or on
[irc.freenode.net, #swagger](http://swagger.io/irc/). For this
sample, you can use the api key `special-key` to test the
authorization filters."
    version: "1.0.0"
    title: "Swagger Petstore"
    termsOfService: "http://swagger.io/terms/"
    contact:
        email: "apiteam@swagger.io"
    license:
        name: "Apache 2.0"
        url: "http://www.apache.org/licenses/LICENSE-2.0.html"
host: "petstore.swagger.io"
basePath: "/v2"
tags:
    - name: "pet"
        description: "Everything about your Pets"
- name: "store"
  description: "Access to Petstore orders"
- name: "user"
  description: "Operations about user"
  externalDocs:
    description: "Find out more about our store"
    url: "http://swagger.io"

schemes:
- "https"
- "http"

paths:
/pet:
  post:
    tags:
      - "pet"
    summary: "Add a new pet to the store"
    description: ""
    operationId: "addPet"
    consumes:
      - "application/json"
      - "application/xml"
    produces:
      - "application/xml"
      - "application/json"
    parameters:
      - in: "body"
        name: "body"
        description: "Pet object that needs to be added to the store"
        required: true
        schema:
          $ref: "#/definitions/Pet"
    responses:
      405:
        description: "Invalid input"
        security:
          - petstore_auth:
            - "write:pets"
            - "read:pets"

  put:
    tags:
      - "pet"
    summary: "Update an existing pet"
    description: ""
    operationId: "updatePet"
    consumes:
      - "application/json"
      - "application/xml"
    produces:
      - "application/xml"
      - "application/json"
    parameters:
- in: "body"
  name: "body"
  description: "Pet object that needs to be added to the store"
  required: true
  schema:
    $ref: "#/definitions/Pet"
responses:
  400:
    description: "Invalid ID supplied"
  404:
    description: "Pet not found"
  405:
    description: "Validation exception"
security:
- petstore_auth:
  - "write:pets"
  - "read:pets"
/pet/findByStatus:
  get:
    tags:
    - "pet"
    summary: "Finds Pets by status"
    description: "Multiple status values can be provided with comma separated strings"
    operationId: "findPetsByStatus"
    produces:
    - "application/xml"
    - "application/json"
    parameters:
    - name: "status"
      in: "query"
      description: "Status values that need to be considered for filter"
      required: true
      type: "array"
      items:
        type: "string"
        enum:
        - "available"
        - "pending"
        - "sold"
        default: "available"
        collectionFormat: "multi"
    responses:
      200:
        description: "successful operation"
        schema:
          type: "array"
          items:
            $ref: "#/definitions/Pet"
      400:
        description: "Invalid status value"
security:
- petstore_auth:
/pet/findByTags:
  get:
    tags:
      - "pet"
    summary: "Finds Pets by tags"
    description: "Multiple tags can be provided with comma separated strings. Use tag1, tag2, tag3 for testing."
    operationId: "findPetsByTags"
    produces:
      - "application/xml"
      - "application/json"
    parameters:
      - name: "tags"
        in: "query"
        description: "Tags to filter by"
        required: true
        type: "array"
        items:
          type: "string"
          collectionFormat: "multi"
    responses:
      200:
        description: "successful operation"
        schema:
          type: "array"
          items:
            $ref: "/definitions/Pet"
      400:
        description: "Invalid tag value"
        security:
          - petstore_auth:
            - "write:pets"
            - "read:pets"
            deprecated: true
/pet/{petId}:
  get:
    tags:
      - "pet"
    summary: "Find pet by ID"
    description: "Returns a single pet"
    operationId: "getPetById"
    produces:
      - "application/xml"
      - "application/json"
    parameters:
      - name: "petId"
        in: "path"
        description: "ID of pet to return"
        required: true
        type: "integer"
        format: "int64"
responses:
  200:
    description: "successful operation"
    schema:
      $ref: "#/definitions/Pet"
  400:
    description: "Invalid ID supplied"
  404:
    description: "Pet not found"
security:
- api_key: 
post:
  tags:
- "pet"
  summary: "Updates a pet in the store with form data"
  description: ""
  operationId: "updatePetWithForm"
  consumes:
- "application/x-www-form-urlencoded"
  produces:
- "application/xml"
- "application/json"
  parameters:
  - name: "petId"
    in: "path"
    description: "ID of pet that needs to be updated"
    required: true
    type: "integer"
    format: "int64"
  - name: "name"
    in: "formData"
    description: "Updated name of the pet"
    required: false
    type: "string"
  - name: "status"
    in: "formData"
    description: "Updated status of the pet"
    required: false
    type: "string"
  responses:
  405:
    description: "Invalid input"
security:
- petstore_auth:
  - "write:pets"
  - "read:pets"
delete:
  tags:
  - "pet"
  summary: "Deletes a pet"
  description: ""
  operationId: "deletePet"
  produces:
parameters:
- name: "api_key"
  in: "header"
  required: false
  type: "string"
- name: "petId"
  in: "path"
  description: "Pet id to delete"
  required: true
  type: "integer"
  format: "int64"
responses:
  400:
    description: "Invalid ID supplied"
  404:
    description: "Pet not found"
security:
- petstore_auth:
  - "write:pets"
  - "read:pets"
/pet/{petId}/uploadImage:
  post:
    tags:
    - "pet"
    summary: "uploads an image"
    description: ""
    operationId: "uploadFile"
    consumes:
    - "multipart/form-data"
    produces:
    - "application/json"
    parameters:
    - name: "petId"
      in: "path"
      description: "ID of pet to update"
      required: true
      type: "integer"
      format: "int64"
    - name: "additionalMetadata"
      in: "formData"
      description: "Additional data to pass to server"
      required: false
      type: "string"
    - name: "file"
      in: "formData"
      description: "file to upload"
      required: false
      type: "file"
    responses:
    200:
      description: "successful operation"
schema:
  $ref: "/definitions/ApiResponse"

security:
  - petstore_auth:
    - "write:pets"
    - "read:pets"

/store/inventory:
  get:
    tags:
      - "store"
    summary: "Returns pet inventories by status"
    description: "Returns a map of status codes to quantities"
    operationId: "getInventory"
    produces:
      - "application/json"
    parameters: []
    responses:
      200:
        description: "successful operation"
        schema:
          type: "object"
          additionalProperties:
            type: "integer"
            format: "int32"
        security:
          - api_key: []

/store/order:
  post:
    tags:
      - "store"
    summary: "Place an order for a pet"
    description: ""
    operationId: "placeOrder"
    produces:
      - "application/xml"
      - "application/json"
    parameters:
      - in: "body"
        name: "body"
        description: "order placed for purchasing the pet"
        required: true
        schema:
          $ref: "/definitions/Order"
    responses:
      200:
        description: "successful operation"
        schema:
          $ref: "/definitions/Order"
      400:
        description: "Invalid Order"

/store/order/{orderId}:
  get:
    tags:
- "store"
  summary: "Find purchase order by ID"
  description: "For valid response try integer IDs with value >= 1 and
  <= 10. Other values will generated exceptions"
  operationId: "getOrderById"
  produces:
  - "application/xml"
  - "application/json"
  parameters:
  - name: "orderId"
    in: "path"
    description: "ID of pet that needs to be fetched"
    required: true
    type: "integer"
    maximum: 10.0
    minimum: 1.0
    format: "int64"
  responses:
  200:
    description: "successful operation"
    schema:
      $ref: "#/definitions/Order"
  400:
    description: "Invalid ID supplied"
  404:
    description: "Order not found"

delete:
  tags:
  - "store"
  summary: "Delete purchase order by ID"
  description: "For valid response try integer IDs with positive
  integer value. Negative or non-integer values will generate API
  errors"
  operationId: "deleteOrder"
  produces:
  - "application/xml"
  - "application/json"
  parameters:
  - name: "orderId"
    in: "path"
    description: "ID of the order that needs to be deleted"
    required: true
    type: "integer"
    minimum: 1.0
    format: "int64"
  responses:
  400:
    description: "Invalid ID supplied"
  404:
    description: "Order not found"
/user:
  post:
  tags:
Create user

This can only be done by the logged in user.

operationId: createUser

produces:
- "application/xml"
- "application/json"

parameters:
- in: "body"
  name: "body"
  description: "Created user object"
  required: true
  schema:
    $ref: "#/definitions/User"

responses:
  default:
    description: "successful operation"

/post:
  tags:
  - "user"
  summary: "Creates list of users with given input array"
  description: ""
  operationId: "createUsersWithArrayInput"
  produces:
  - "application/xml"
  - "application/json"

parameters:
- in: "body"
  name: "body"
  description: "List of user object"
  required: true
  schema:
    type: "array"
    items:
      $ref: "#/definitions/User"

responses:
  default:
    description: "successful operation"

/post:
  tags:
  - "user"
  summary: "Creates list of users with given input array"
  description: ""
  operationId: "createUsersWithListInput"
  produces:
  - "application/xml"
  - "application/json"

parameters:
- in: "body"
  name: "body"
  description: "List of user object"
required: true
schema:
  type: "array"
  items:
    $ref: "#/definitions/User"
responses:
  default:
    description: "successful operation"
/user/login:
  get:
    tags:
      - "user"
    summary: "Logs user into the system"
    description: ""
    operationId: "loginUser"
    produces:
      - "application/xml"
      - "application/json"
    parameters:
      - name: "username"
        in: "query"
        description: "The user name for login"
        required: true
        type: "string"
      - name: "password"
        in: "query"
        description: "The password for login in clear text"
        required: true
        type: "string"
    responses:
      200:
        description: "successful operation"
        schema:
          type: "string"
        headers:
          X-Rate-Limit:
            type: "integer"
            format: "int32"
            description: "calls per hour allowed by the user"
          X-Expires-After:
            type: "string"
            format: "date-time"
            description: "date in UTC when token expires"
      400:
        description: "Invalid username/password supplied"
/user/logout:
  get:
    tags:
      - "user"
    summary: "Logs out current logged in user session"
    description: ""
    operationId: "logoutUser"
    produces:
parameters: []
responses:
  default:
    description: "successful operation"

/user/{username}:
  get:
    tags:
      - "user"
    summary: "Get user by user name"
    description: ""
    operationId: "getUserByName"
    produces:
      - "application/xml"
      - "application/json"
    parameters:
      - name: "username"
        in: "path"
        description: "The name that needs to be fetched. Use user1 for testing."
        required: true
        type: "string"
    responses:
      200:
        description: "successful operation"
        schema:
          $ref: "#/definitions/User"
      400:
        description: "Invalid username supplied"
      404:
        description: "User not found"

  put:
    tags:
      - "user"
    summary: "Updated user"
    description: "This can only be done by the logged in user."
    operationId: "updateUser"
    produces:
      - "application/xml"
      - "application/json"
    parameters:
      - name: "username"
        in: "path"
        description: "name that need to be updated"
        required: true
        type: "string"
      - in: "body"
        name: "body"
        description: "Updated user object"
        required: true
        schema:
          $ref: "#/definitions/User"
responses:
  400:
    description: "Invalid user supplied"
  404:
    description: "User not found"
delete:
  tags:
  - "user"
  summary: "Delete user"
  description: "This can only be done by the logged in user."
  operationId: "deleteUser"
  produces:
  - "application/xml"
  - "application/json"
  parameters:
  - name: "username"
    in: "path"
    description: "The name that needs to be deleted"
    required: true
    type: "string"
  responses:
  400:
    description: "Invalid username supplied"
  404:
    description: "User not found"

securityDefinitions:
  petstore_auth:
    type: "oauth2"
    authorizationUrl: "http://petstore.swagger.io/oauth/dialog"
    flow: "implicit"
    scopes:
      write:pets: "modify pets in your account"
      read:pets: "read your pets"
  api_key:
    type: "apiKey"
    name: "api_key"
    in: "header"
definitions:
  Order:
    type: "object"
    properties:
      id:
        type: "integer"
        format: "int64"
      petId:
        type: "integer"
        format: "int64"
      quantity:
        type: "integer"
        format: "int32"
      shipDate:
        type: "string"
        format: "date-time"
status:
  type: "string"
  description: "Order Status"
  enum:
    - "placed"
    - "approved"
    - "delivered"
complete:
  type: "boolean"
  default: false
xml:
  name: "Order"
Category:
  type: "object"
  properties:
    id:
      type: "integer"
      format: "int64"
    name:
      type: "string"
xml:
  name: "Category"
User:
  type: "object"
  properties:
    id:
      type: "integer"
      format: "int64"
    username:
      type: "string"
    firstName:
      type: "string"
    lastName:
      type: "string"
    email:
      type: "string"
    password:
      type: "string"
    phone:
      type: "string"
    userStatus:
      type: "integer"
      format: "int32"
      description: "User Status"
xml:
  name: "User"
Tag:
  type: "object"
  properties:
    id:
      type: "integer"
      format: "int64"
      name:
### Appendix 2 Document Change History

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Author</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1.0</td>
<td>2.10.2018</td>
<td>Ilkka Seppälä</td>
<td>Initial draft version</td>
</tr>
</tbody>
</table>