INFORMATION TECHNOLOGIES FOR SHIFT2RAIL

D1.2 – Semantic Web Service Registry

Due date of deliverable: 31/08/2017

Actual submission date: 28/11/2017

Leader of this Deliverable: Cefriel

Reviewed: Y

Document status		
Revision	Date	Description
1	19/07/2017	Draft
2	17/10/2017	Revision after ThalesPT comments
3	28/11/2017	Final Version after TMC approval

Project funded from the European Union's Horizon 2020 research and innovation programme		
Dissemination Level		
PU	Public	X
CO	Confidential, restricted under conditions set out in Model Grant Agreement	
CI	Classified, information as referred to in Commission Decision 2001/844/EC	

Start date of project: 01/05/2015 Duration: 36 months

EXECUTIVE SUMMARY

The Interoperability Framework plays a central role in IT²RAIL as it enables different actors from several companies to exchange data. The IT²RAIL Semantic Assets Manager is the component devoted to govern the process of maintaining a shared repository of assets such as ontologies, data schemas and service descriptors. It is an organised collection and storage of these assets, enhanced by tools to support a workflow process for review, versioning, approval and publishing of the assets.

The IT²RAIL Semantic Assets Manager is built on top of the open-source WSO2 Framework that provides: (i) a web application (the Publisher) through which owners/contributors make informational assets available to the community; (ii) a web application (the Management Console) including a workflow process tool that supports the collaborative management of the published assets, and (iii) an organised web repository (the Store) of digital assets accessible by any participant organisation, human actor or application. The IT²RAIL Ontology Repository and Semantic Web Service Registry will be particular sections of the Store, in which ontology files and semantically annotated web service descriptors are stored for use after having completed the approval/versioning process supported by the management console, which operates on digital inputs provided by their owners / contributors through the publisher function.

This deliverable aims at:

- Provide a detailed description of the WSO2 Framework (Section 1);
- Describe the architecture of the IT²RAIL Semantic Assets Manager and provide details about how the WSO2 framework has been extended to support the definition and visualisation of new asset types through JSON schemas and the definition of custom asset lifecycles (Section 2);
- Describe the assets (i.e., *ontologies, travel expert descriptions, JSON schemas* and *RDF dataset*) currently managed by the IT²RAIL Semantic Assets Manager and provide a list of potential additional assets to be managed (Section 3).

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

API: Application Programming Interface

CSA: Coordination and Support Actions

DCAT: Data Catalog

DCAT-AP: Data Catalog Application Profile IF: Interoperability Framework

JSON: JavaScript Object Notation

RDF: Resource Description Framework

REST: REpresentational State Transfer

RXT: Configurable Governance Artifacts

SCXML: State Chart XML

SOAP: Simple Object Access Protocol

SPARQL: SPARQL Protocol and RDF Query Language

SQL: Structured Query Language

ST4RT: Semantic Transformation for Rail Transportation

TE: Travel Expert

URI: Uniform Resource Identifier

URL: Uniform Resource Locator

XML: eXtensible Markup Language

XSD: XML Schema Definition

WADL: Web Application Description Language

WSDL: Web Service Description Language

1. THE WSO2 FRAMEWORK

The Assets Manager developed by IT²RAIL is built on top of the open-source WSO2 Governance Registry[1] that provides three primary functions:

1. **Management Console**: a web application including a workflow process tool that supports the collaborative management of the published assets, i.e. reviews, discussions, versioning, approval, distribution, etc.;
2. **Publisher**: a web application through which owners/contributors make informational assets available to the community;
3. **Store**: an organised web repository of digital assets accessible by any participant organisation, human actor or application.

1.1 THE MANAGEMENT CONSOLE

The Management Console of the WSO2 Framework supports the **definition of asset types** by using Configurable Governance Artefacts (RXT)[2].

RXT definitions consist of a set of tables containing typed fields. In case of multiple choice fields, the values can be obtained by dynamically calling external Java code, therefore allowing populating fields values using data obtained by running an SQL or SPARQL query. The example reported in Figure 1 is related to the definition of an Ontology asset type, showing the definition of two tables and their fields. The first table, named *Overview*, contains the following metadata fields for characterising the ontology: *name, version, author, institution, domain, description* and *expected validity*. The second table, named *Content*, contains the following fields for locating and sharing the ontology: *ontology URL, upload file* and *follow imported ontologies.*

In addition to the definition of an asset type, the Management Console supports also the **definition of custom asset lifecycles** as finished state machines expressed using State Chart XML[3]. Each lifecycle contains a set of states and the allowed transitions, plus additional constraints to be enforced while changing status and actions to be triggered upon successful status transition. The “actions” are implemented with Java code, therefore allowing the execution of arbitrary code.

[1] http://wso2.com/products/governance-registry
[2] https://docs.wso2.com/pages/viewpage.action?pageId=48284982
[3] https://www.w3.org/TR/scxml/

```
<?xml version="1.0" encoding="UTF-8"?>
<artifactType type="application/vnd.wso2-ontology+xml" shortName="ontology" singularLabel="Ontology" pluralLabel="Ontologies"
    hasNamespace="false" iconSet="27">
    <storagePath>/ontology/@{name}</storagePath>
    <nameAttribute>overview_name</nameAttribute>
    <ui>
        <list>
            <column name="Name">
                <data type="text" value="overview_name"/>
            </column>
            <column name="Version">
                <data type="path" value="overview_version" href="@{storagePath}"/>
            </column>
        </list>
    </ui>
    <content>
        <table name="Overview">
            <field type="text" required="true" >
                <name label="Name">Name</name>
            </field>
            <field type="text">
                <name label="Version">Version</name>
            </field>
            <field type="text">
                <name label="Author">Author</name>
            </field>
            <field type="text">
                <name label="Institution">Institution</name>
            </field>
            <field type="options">
                <name label="Domain">Domain</name>
                <values>
                    <value>Infrastructure</value>
                    <value>Customer services</value>
                </values>
            </field>
            <field type="text">
                <name>Createdtime</name>
            </field>
            <field type="text-area">
                <name label="Description">Description</name>
            </field>
            <field type="date">
                <name label="Expected validity">Expected validity</name>
            </field>
        </table>
        <table name="Content">
            <field type="text">
                <name label="Ontology URL">URL</name>
            </field>
            <field type="text">
                <name label="Upload file">File</name>
            </field>
            <field type="options">
                <name label="Follow imported ontologies">Follow imports</name>
                <values>
                    <value>Yes</value>
                    <value>No</value>
                </values>
            </field>
        </table>
```

Figure 1: Sample RXT definition of an asset type

This function is fundamental for IT²RAIL since the editing process of an asset is split among several actors and companies. Therefore, before deciding to publish an artefact, an approval process must take place, featuring a series of "lifecycle stages". For instance, an asset may start off as "created", then after quality assurance has confirmed that the asset is consistent should be moved to the "tested" status. Upon testing, the asset can then move to a "deployed" status at which point it is released to production. Eventually, the asset can be taken down or replaced with another as it moves to a "deprecated" status.

An example of lifecycle definition is shown in Figure 2.

```
<aspect name="IT2RailLifeCycle" class="org.wso2.jaggery.scxml.aspects.JaggeryTravellingPermissionLifeCycle">
    <configuration type="literal">
        <lifecycle>
            <scxml xmlns="http://www.w3.org/2005/07/scxml"
                version="1.0"
                initialstate="Initial">
                <state id='Initial'>
                    <datamodel>
                        <data name="transitionExecution">
                            <execution forEvent="Promote" class="org.wso2.jaggery.scxml.generic.GenericExecutor">
                                <parameter name="PERMISSION:get"
                                    value="http://www.wso2.org/projects/registry/actions/get"/>
                                <parameter name="PERMISSION:add"
                                    value="http://www.wso2.org/projects/registry/actions/add"/>
                                <parameter name="PERMISSION:delete"
                                    value="http://www.wso2.org/projects/registry/actions/delete"/>
                                <parameter name="PERMISSION:authorize" value="authorize"/>

                                <parameter name="STATE_RULE1:Created"
                                    value="Internal/private_{asset_author}:+get,+add,+delete,+authorize"/>
                                <parameter name="STATE_RULE2:Created"
                                    value="Internal/reviewer:-get,-add,-delete,-authorize"/>
                                <parameter name="STATE_RULE3:Created"
                                    value="Internal/everyone:-get,-add,-delete,-authorize"/>
                                </execution>
                        </data>
                    </datamodel>
                    <transition event="Promote" target="Created"/>
                </state>
                <state id="Created">
                    <datamodel>
                        <data name="transitionExecution">
                            <execution forEvent="Promote" class="org.wso2.jaggery.scxml.generic.GenericExecutor">
                                <parameter name="PERMISSION:get"
                                    value="http://www.wso2.org/projects/registry/actions/get"/>
                                <parameter name="PERMISSION:add"
                                    value="http://www.wso2.org/projects/registry/actions/add"/>
                                <parameter name="PERMISSION:delete"
                                    value="http://www.wso2.org/projects/registry/actions/delete"/>
                                <parameter name="PERMISSION:authorize" value="authorize"/>


                                <parameter name="STATE_RULE1:In-Review"
                                    value="Internal/private_{asset_author}:+get,-add,-delete,-authorize"/>
                                <parameter name="STATE_RULE2:In-Review"
                                    value="Internal/reviewer:+get,+add,-delete,+authorize"/>
                                <parameter name="STATE_RULE3:In-Review"
                                    value="Internal/everyone:-get,-add,-delete,-authorize"/>

                                </execution>
                            </data>
                        </datamodel>
                        <transition event="Promote" target="In-Review"/>
                    </state>
```

Figure 2: An example of lifecycle definition using SCXML

1.2 THE PUBLISHER

Once the asset type is defined, the Publisher web application allows authorised users to add new assets or to modify existing ones. The WSO2 framework proposes default asset types: Policies, REST Services, SOAP Services, Swaggers, WADLs and WSDLs. Since the WSO2 Framework is not limited to these asset types, new form-based interface for editing information about an asset can be auto-generated from the asset type definition. Figure 3 shows the form allowing the editing of a SOAP Service description.

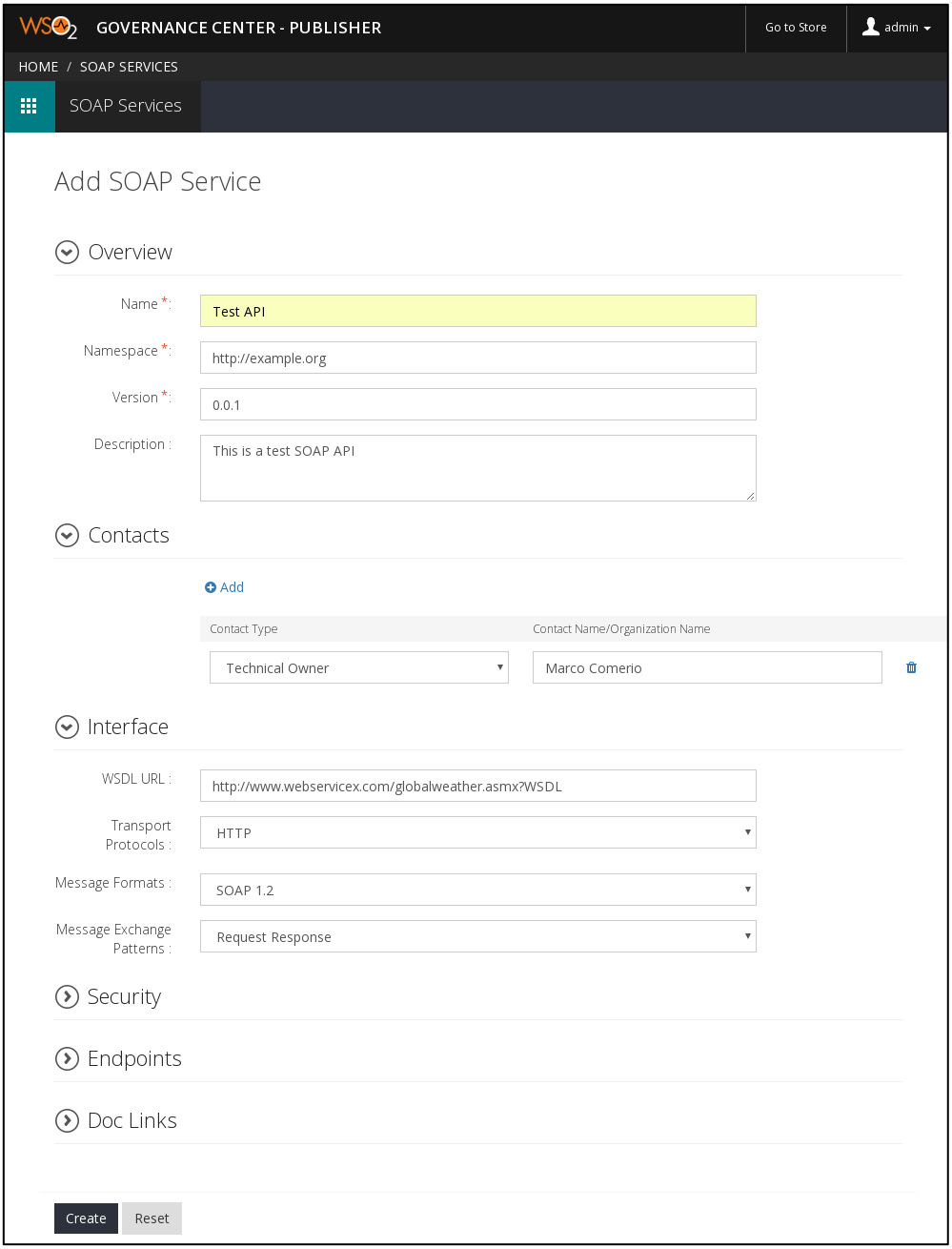


Figure 3: Form for editing a SOAP Service description.

Once the lifecycle workflow has been designed and deployed, the Publisher application allows the users to modify the status of an asset. Figure 4 shows the lifecycle editing interface for the SOAP Service described using the form in Figure 3.

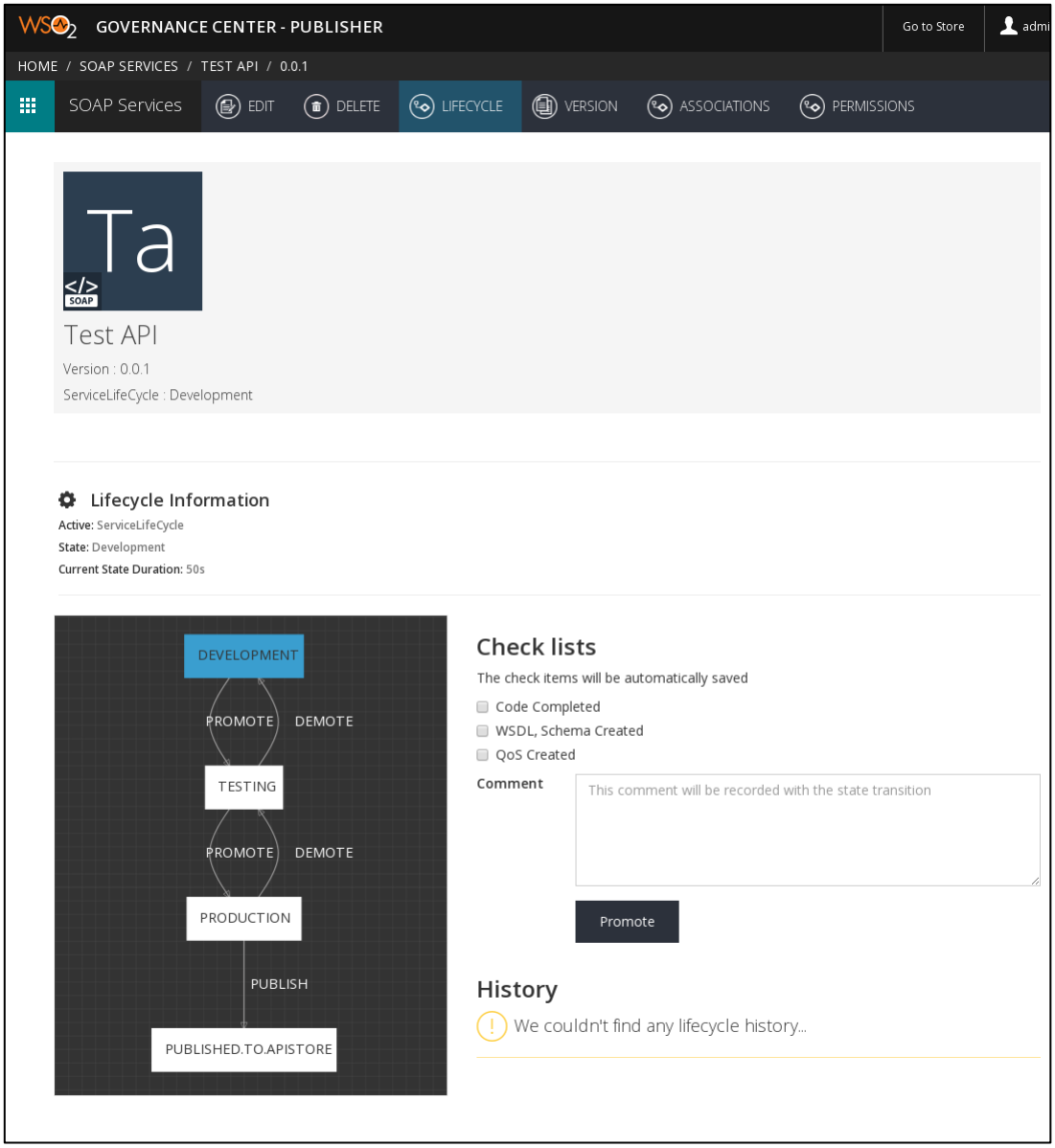
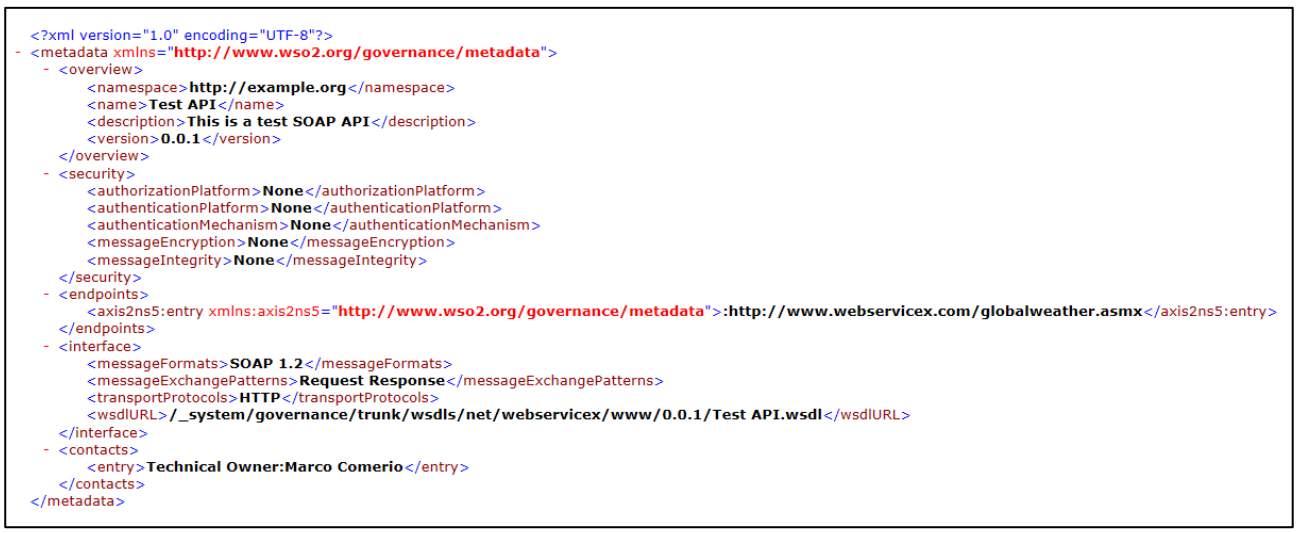


Figure 4: An example of the asset lifecycle editing interface inside the Publisher

After the filling of the form, data related to the asset are stored inside the database used by WSO2 as an XML document. Figure 5 shows the document resulting from filling the form in Figure 3.



```
<?xml version="1.0" encoding="UTF-8"?>
- <metadata xmlns="http://www.wso2.org/governance/metadata">
  - <overview>
      <namespace>http://example.org</namespace>
      <name>Test API</name>
      <description>This is a test SOAP API</description>
      <version>0.0.1</version>
    </overview>
  - <security>
      <authorizationPlatform>None</authorizationPlatform>
      <authenticationPlatform>None</authenticationPlatform>
      <authenticationMechanism>None</authenticationMechanism>
      <messageEncryption>None</messageEncryption>
      <messageIntegrity>None</messageIntegrity>
    </security>
  - <endpoints>
      <axis2ns5:entry xmlns:axis2ns5="http://www.wso2.org/governance/metadata">:http://www.webservicex.com/globalweather.asmx</axis2ns5:entry>
    </endpoints>
  - <interface>
      <messageFormats>SOAP 1.2</messageFormats>
      <messageExchangePatterns>Request Response</messageExchangePatterns>
      <transportProtocols>HTTP</transportProtocols>
      <wsdlURL>/_system/governance/trunk/wsdls/net/webservicex/www/0.0.1/Test API.wsdl</wsdlURL>
    </interface>
  - <contacts>
      <entry>Technical Owner:Marco Comerio</entry>
    </contacts>
</metadata>
```

Figure 5: XML content of a SOAP Service description

1.3 THE STORE

The store is the frontend allowing the users for accessing and searching existing assets according to the defined authorisation policy. The “User reviews” tab allows inserting ratings and comments about assets, therefore enabling the possibility of collecting hints to improve the content of the assets.

The IT²RAIL ***Ontology Repository*** and ***Semantic Web Service Registry*** will be particular sections of the Store, in which ontology files and semantically annotated web service descriptors are stored for use after having completed the approval/versioning process supported by the management console, which operates on digital inputs provided by their owners / contributors through the publisher function.

Figure 6 shows an overview of the Store that, among the available assets, includes the SOAP Service description resulting from filling the form in Figure 3.

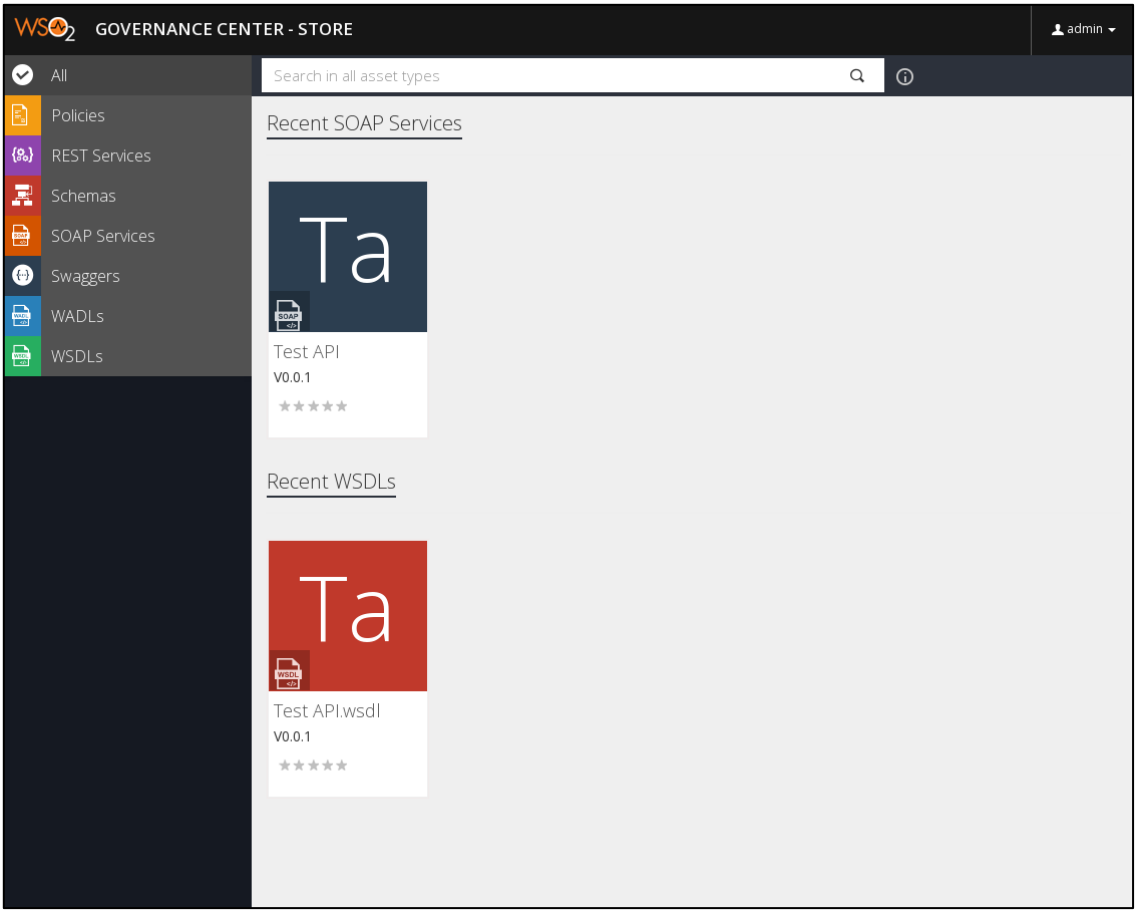


Figure 6: An overview of the Store

2. THE IT²RAIL SEMANTIC ASSETS MANAGER

The architecture of the IT²RAIL solution for the Semantic Assets Manager is depicted in Figure 7. It is composed of three components:

- **Assets Manager**: an extension and configuration of the WSO2 framework to cover the management of the IT²RAIL Interoperability Framework assets;
- **Transformation Server**: a new component, external to the WSO2 framework, developed to add semantics to the asset descriptions. It enables the transformation of "Asset XML contents" to RDF triples;
- **RDF Repository**: a semantic graph database used as RDF triplestore. The GraphDB[4] has been adopted.

[4] http://graphdb.ontotext.com/graphdb/

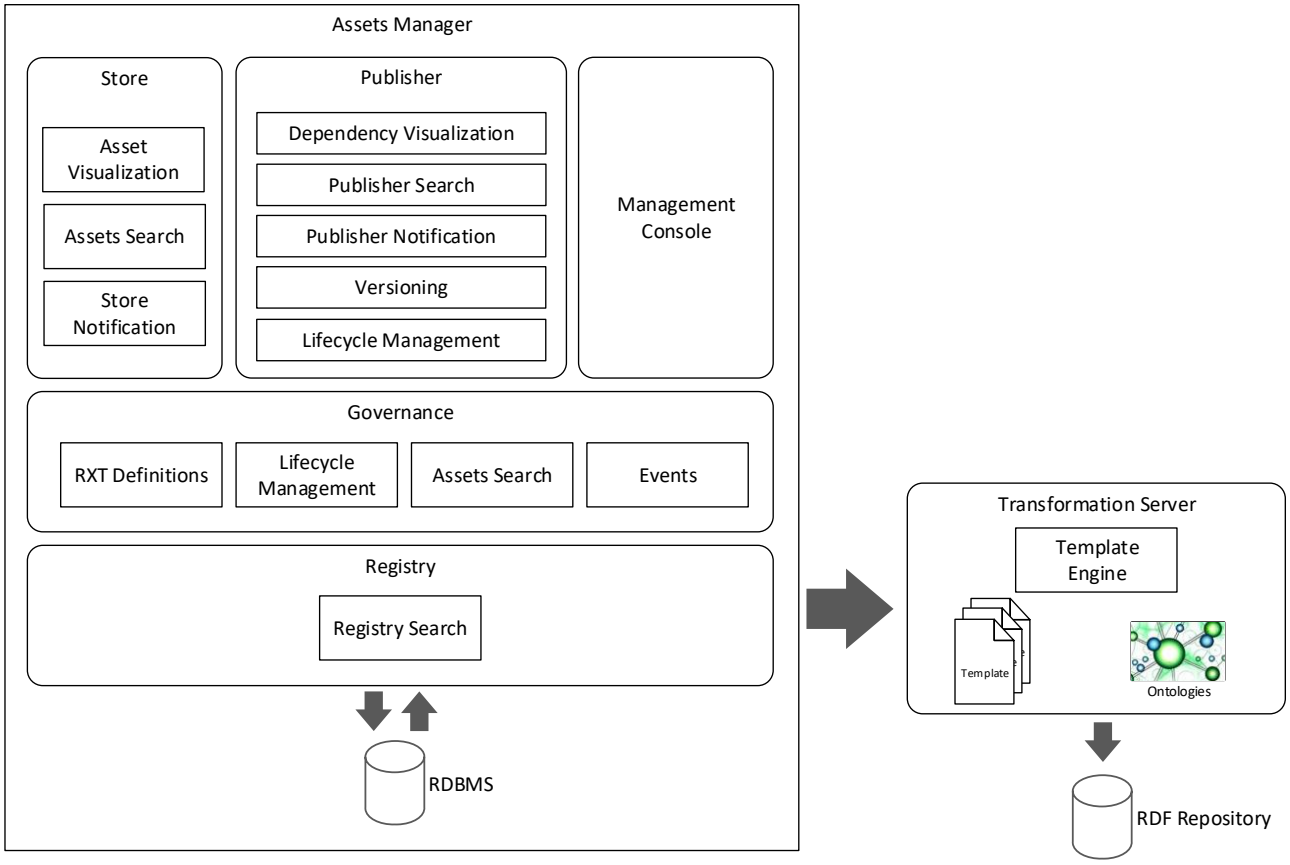


Figure 7: The architecture of the IT²RAIL Semantic Assets Manager

2.1 WSO2 EXTENSIONS

In order to realise the features required by the Assets Manager, the following main extensions of the WSO2 framework have been made:

- Definition and visualisation of new asset types through JSON schemas;
- Definition of custom asset lifecycles.

2.1.1 Definition and visualisation of new asset types through JSON schemas

The WSO2 Store is a generic store, which allows any type of resource defined by a RXT to be deployed and managed. It consists of a Store Front and Back Office, which respectively provides a consumer facing view and a management view, for asset publishers and curators. The Store provides a set of standard views and generic business logic implementations. However, most often developers may need to customise the Enterprise Store by changing its default behaviour and appearance to suit their personal needs. Among others, the Store supports **asset extension** that allows the definition of a new asset type with a set of custom behaviours and views. The behaviours defined within an asset extension are only applicable for that specific asset type. As an example, an asset extension could consist in changing an asset view, i.e., modifying the way the asset listing page appears.

The *asset.js script* is the core component of any asset extension. This script allows business logic of an asset type to be altered with the use of several callback methods, which are:

- ***asset.configure***: used to alter RXT properties and to define meta properties (i.e., thumbnail, banner and category);
- ***asset*.server**: used to create or override APIs and pages;
- ***asset.renderer***: used to intercept calls to render operations such as, control page decorators (e.g., tag cloud, social meta information, rating);
- ***asset.manager***: used to override the existing methods to handle the CRUD (create, read, update and delete) business logic of an asset type.

The *asset.js* script that corresponds to an asset extension inherits callback methods from the default asset extension. The Store uses the generic asset manager implementation, which is in the RXT module, as the parent implementation. Therefore, as the callback methods are inherited from the default asset extension, a developer can choose to implement only some of the callback methods, or even choose not to define any of the callback methods in the asset extension. Thereby, if a developer does not define any callback methods, the callback methods in the default asset extension are invoked.

RXT asset definitions are based upon a “database-centric” philosophy. A “table” tag inside the asset type definition is meant to be stored inside a single database table. As such, it is difficult to represent deeply nested field and complex data structures (see, as an example, the Travel Expert Description Asset Type detailed in Section 3.2). JSON schemas appears more suitable to define articulated asset type. Therefore, to overcome those expressivity limitations, we added the possibility to specify an asset schema through JSON Schemas. A JSON Schema asset type has been created to let developers of the Asset Manager have a consistent view over the existing data structure, and to ease integration with other tools (e.g. SOFIA2 broker also used in the IT²RAIL Interoperability Framework). Inside the newly created asset definitions, we link to the existing JSON Schema asset, and then we exploit a client-side form generation library to handle the form rendering and the collection of the data gathered through the form itself, as depicted in Figure 8.

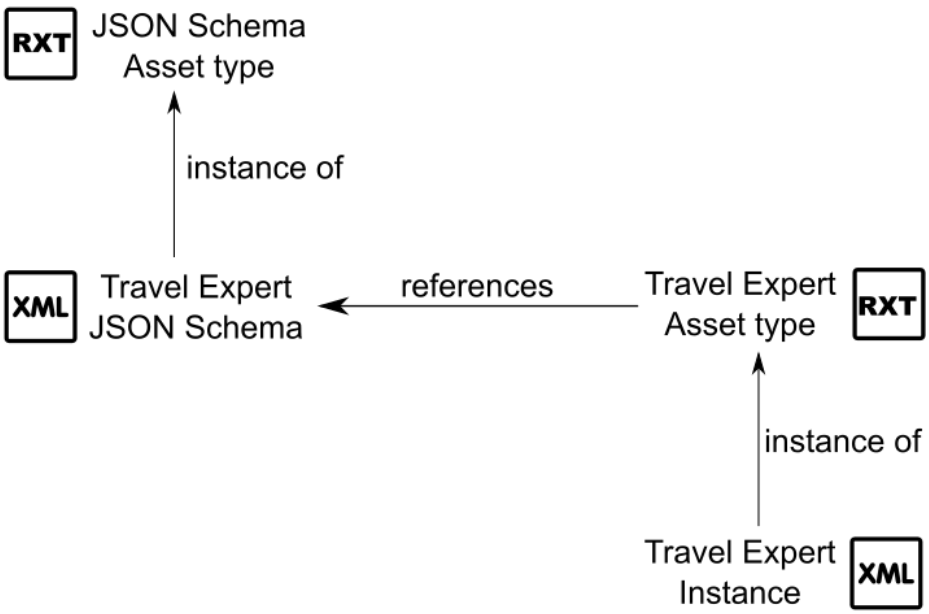


Figure 8 WSO2 extension to manage JSON Schemas

The integration with the form generation library has been carried out by customising the Asset Manager via an asset extension. A new asset type (i.e., Travel Expert asset type) has been created and its *asset.js* script has been configured to create a new page dedicated to JSON Schema visualisation by means of Alpaca libraries[5].

2.1.2 Definition of custom asset lifecycles

Another WSO2 extension performed to realise the features required by the IT²RAIL Assets Manager consists in the definition of **custom asset lifecycles** as finished state machines expressed using State Chart XML[6]. Each lifecycle contains a set of states and the allowed transitions, plus additional constraints to be enforced while changing status and actions to be triggered upon successful status transition. The "actions" are implemented with Java code, therefore allowing the execution of arbitrary code.

[5] www.alpacajs.org
[6] www.w3.org/TR/scxml/

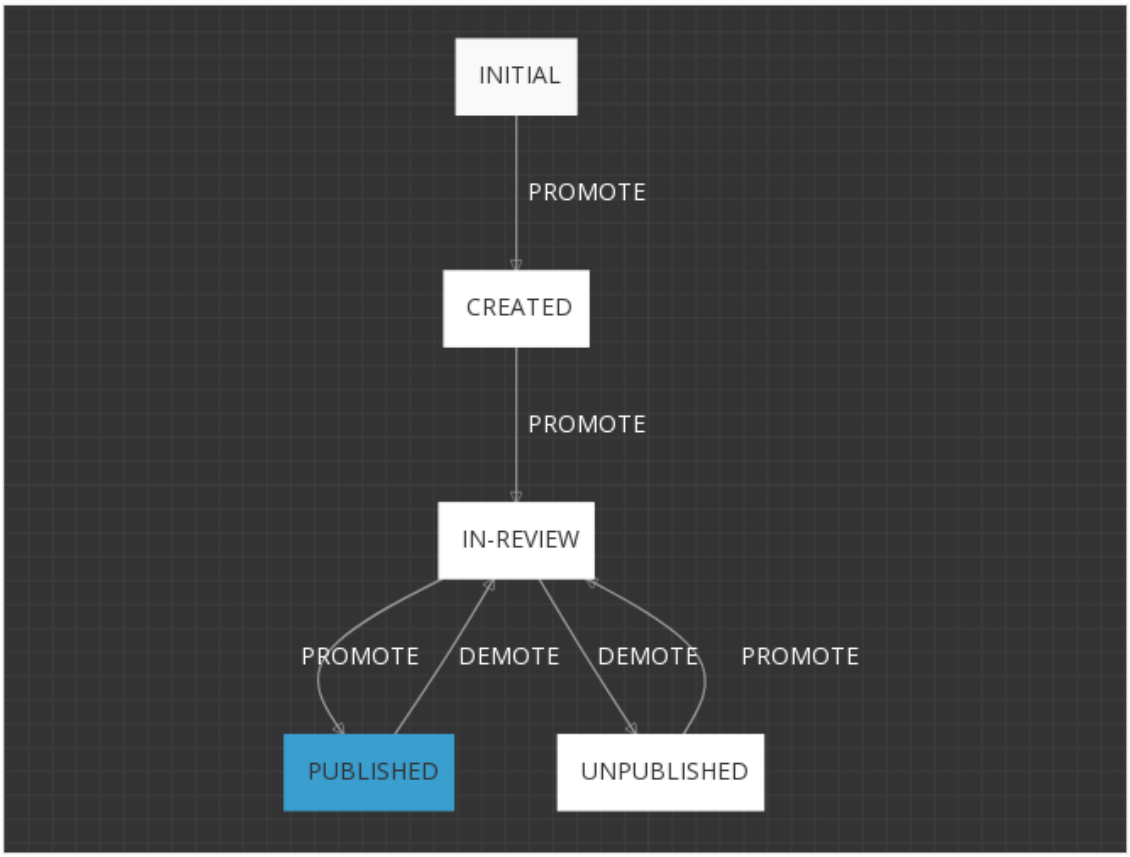


Figure 9: IT²RAIL Asset Lifecycle

Starting from the standard lifecycle for IT²RAIL Assets described in Figure 9, a custom lifecycle can be defined. Figure 10 shows the custom lifecycle for the ontology asset expressed using SCXML. Lines 14-28 (on Figure 10) state that:

- Along the transition “IN-REVIEW -> PUBLISHED” triggered by the event “PROMOTE”, the Java code OntologyConnector must be called with two parameters: (i) the URL of the destination and (ii) the operation (“NEW”) to be performed;
- Along the transition “IN-REVIEW -> UNPUBLISHED” triggered by the event “DEMOTE”, the Java code OntologyConnector must be called with two parameters: (i) the URL of the destination and (ii) the operation (“DELETE”) to be performed.

```
<aspect name="OntologyLifeCycle" class="org.wso2.carbon.governance.registry.extensions.aspects.DefaultLifeCycle">
    <configuration type="literal">
        <lifecycle>
            <scxml xmlns="http://www.w3.org/2005/07/scxml"
                   version="1.0"
                   initialstate="Initial">
                <state id='Initial'>
                    <transition event="Promote" target="Created"/>
                </state>
                <state id="Created">
                    <transition event="Promote" target="In-Review"/>
                </state>
                <state id="In-Review">
                    <datamodel>
                        <data name="transitionExecution">
                            <execution forEvent="Promote" class="it.cefriel.wso2.lifecycle.OntologyConnector">
                                <parameter name="DEST_URL" value="http://localhost:15001/assets/"/>
                                <parameter name="OPERATION" value="new"/>
                            </execution>
                            <execution forEvent="Demote" class="it.cefriel.wso2.lifecycle.OntologyConnector">
                                <parameter name="DEST_URL" value="http://localhost:15001/assets/"/>
                                <parameter name="OPERATION" value="delete"/>
                            </execution>
                        </data>
                    </datamodel>
                    <transition event="Promote" target="Published"/>
                    <transition event="Demote" target="Unpublished"/>
                </state>
                <state id="Published">
                    <transition event="Demote" target="In-Review"/>
                </state>
                <state id="Unpublished">
                    <transition event="Promote" target="In-Review"/>
                </state>
            </scxml>
        </lifecycle>
    </configuration>
</aspect>
```

Figure 10: custom lifecycle for the ontology asset

2.2 TRANSFORMATION SERVER

The Transformation Server is a new component, external of the WSO2 framework, developed to enable the transformation of “Asset XML contents” to “RDF triples”. Asset XML contents are XML data from WSO2 concerning the description of a specific asset (e.g., the IT²RAIL ontology) resulting from the filling of the form auto-generated from the asset JSON schema.

The Transformation Server makes use of the JinJa Template Engine[7] and available N3/Turtle Templates to perform the XML2RDF transformation. An example of Turtle template for the ontology asset is shown in Figure 11. The template is made of conditions guiding the creation of RDF triples. As an example, Lines 8-10 (on Figure 11) states that if the “Asset XML contents” contains “overview.name”, then the triple “*asset_id rdfs:label overview.name*” is created.

[7] http://jinja.pocoo.org/

```
{% set prefix="http://www.it2rail.eu/instances#"%}
{% set asset_name = slugify(root.overview.name.text) %}
{% set asset_id = '<'+prefix+slugify(uid)+'>' %}
# baseURI: http://www.it2rail.eu/ontology
@prefix dcterms: <http://purl.org/dc/terms/> .
@prefix gr: <http://purl.org/goodrelations/v1#> .

{%if root.overview.name%}
    {{asset_id}} rdfs:label "{{root.overview.name}}".
{%endif%}
{%if root.overview.version%}
  {{asset_id}} schema:versio"{{root.overview.version}}".
{%endif%}
{%if root.overview.description%}
  {{asset_id}} rdfs:comment """{{root.overview.description}}""".
{%endif%}
{%if root.overview.author%}
  {{asset_id}} dcterms:creator """{{root.overview.author}}""".
{%endif%}
{%if root.overview.institution%}
  {{asset_id}} dcterms:publisher """{{root.overview.institution}}""".
{%endif%}
{%if root.overview.expectedvalidity %}
  {{asset_id}} :validity :validity_{{asset_name}}.
  :validity_{{asset_name}} rdf:type service:TimeInterval;
  gr:validThrough
  "{{ root.overview.expectedvalidity.text.split("/")[2]
  {{ root.overview.expectedvalidity.text.split("/")[0] }}-
  {{ root.overview.expectedvalidity.text.split("/")[1] }}"^^xsd:date.
{%endif%}
{%if root.overview.createdtime%}
  {{asset_id}} dcterms:dateAccepted
  "{{ root.overview.createdtime.text.split("/")[2] }
  {{ root.overview.createdtime.text.split("/")[0] }}-
  {{ root.overview.createdtime.text.split("/")[1] }}"^^xsd:date.
{%endif%}
```

Figure 11: Turtle template of the ontology asset.

The Template Engine performs the following process:

1. Retrieve the template associated to the asset specified in the “Asset XML content”;
2. Fill the retrieved template with data in the “Asset XML content”;
3. Perform the RDF transformation according to ontology annotation specified in the template;
4. Store the RDF triples in the RDF Repository.

3. ASSETS MANAGED BY THE SEMANTIC ASSETS MANAGER

In this section, the assets (i.e., *ontologies, travel expert descriptions, JSON schemas* and *RDF dataset*), currently managed by the IT²RAIL Semantic Assets Manager, are described in terms of their descriptors (i.e., the form auto-generated from their JSON schema). In Section 3.5, a list of potential additional assets to be managed by the Assets Manager is provided.

Each asset is associated with a lifecycle. The IT²RAIL project associates the same lifecycle (shown in Figure 9) to all the managed asset types. Such lifecycles will be revised and differentiated within the context of the Shift2Rail IP4 CSA project GoF4R (Governance of the Interoperability Framework for Rail and Intermodal Mobility).

The lifecycle is made of five states. When the publisher starts filling the asset descriptor, the asset is in the INITIAL status. When the publisher submits the descriptor, the status is promoted to CREATED. When the publisher decides to send the asset to the reviewer in order to start the review process, he/she changes the status to IN-REVIEW. After the review process, the reviewer has two options:

- (i) accept the asset and change the status to PUBLISHED;
- (ii) discard the asset and change the status to UNPUBLISHED.

When requests for changes in published/unpublished assets come, the publisher or the reviewer can demote the status to IN-REVIEW and re-start the review process.

3.1 ONTOLOGIES

Ontology is the main asset type managed by the Assets Manager. Figure 12 shows the form allowing the editing of an Ontology asset description that is auto-generated from its JSON schema.

The form is divided into 4 sections. In the *Overview* section, the following attributes are specified:

- **name**: the name given to the ontology (e.g., IT²RAIL Ontology);
- **version***:* a specific historical description of the ontology;
- **author, author email** and **institution***:* entity responsible for making the ontology available and his/her contact;
- **domain**: the reference domain of the ontology;
- **description**: a brief explanation of the ontology content;
- **expected validity**: the date until when the ontology is supposed to be valid.

In the *Content* section, the **Ontology URL** (i.e., an unambiguous reference to the ontology) is specified together with the identification of the related graph (i.e., **Existing Graph URI**) and the statement about the usage of imported ontologies (i.e., **Follow imported ontologies**). Finally, the sections *Images* and *Tags* support the association of images and metadata/tags to the ontology.

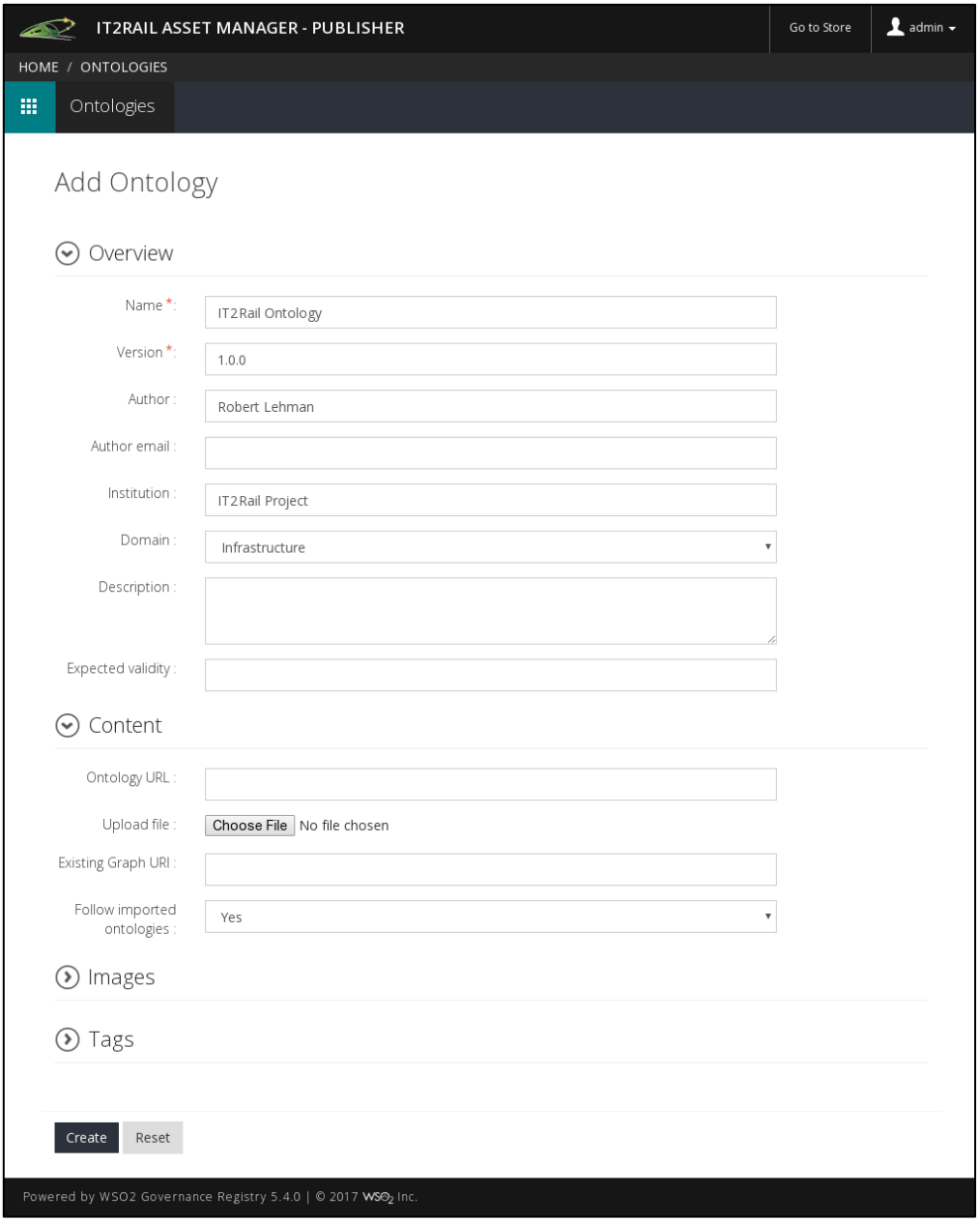


Figure 12: The form for editing an *Ontology* asset

3.2 TRAVEL EXPERT DESCRIPTIONS

The Travel Expert Description is an asset type characterised by a structured JSON schema descriptor covering the many facets of a travel expert service. After the *Overview* attributes used to identify internally the asset and its publisher (i.e., *name, version, author* and *institution*), the descriptor is articulated into five sections: *Images, Tags, Basic Info, Policies* and *Methods. Images* and *Tags* support the association of images and metadata/tags to the Travel Expert Description.

The *Basic Information* section contains the following attributes:

- **name**: commercial name of the Travel Expert;
- **URL endpoint**: the URL where the Travel Expert can be accessed;
- **service provider**: name of the Travel Expert provider;

The *Policy* section includes the following attributes:

- **security protocol**: security protocol used at service level;
- **supported language**: natural languages supported by the Travel Expert;
- **complementary services**: list of complementary services (e.g., car rental, event booking, visa request) offered by the Travel Expert;
- **supported modes of travel**: list of travel mode supported by the Travel Expert. Each supported travel mode is characterised by:
  - **served area**: area (a polygon identified by geographical coordinates) covered by the travel expert which may contain a list of the stop places;
  - **schedules**: dates/periods/hours of operation.
- **supported high level functions**: list of high-level functions (e.g., Planning, Shopping, Booking, Tapping, Tracking) supported by the Travel Expert. Each supported high-level function is further characterised by:
  - **high-level function name**: the name identifying the function (e.g., Planning, Shopping, Booking, Tapping, Tracking);
  - **high-level function category**: category defining the pattern/style used to realise the high-level function. Examples of categories for the high-level function *Shopping* are *one shot*, *journey planning + pricing*, *location resolving + journey planning + pricing*. Examples of categories for the high-level function *Tapping* are *back-end based, application based* and *cloud storage based.*
  - **availability:** dates/periods/hours of operation;
  - **procedure**: callable method(s) to be invoked to realise the high-level function. This is NOT a specification of how to orchestrate methods, it's just a set of methods to be invoked to complete the high-level function;
  - **dependencies**: additional existing applications which is needed / which supports the high-level function (e.g. an existing Tapping Application);
  - **supported ticket medium type**(s) (e.g., physical ticket, SMS, email): to be specified if the high-level function "produces" a ticket;

- **supported passenger category**: list of passenger category supported by the high-level function. Each supported passenger category is characterised by:
  - **label**: a string that identifies the category;
  - **category type** (Single Passenger or Group of Passengers);
  - **travel purpose** (Tourism, Leisure, Business, Not Specified, …);
  - **required service(s)**: services that are required by the specific category of passenger. The required service is defined by:
    - **service type**: the type of requested service (e.g., assistance);
    - **constraint expression**: in case, can be used to characterise the request using an operator (max, min, exact value) and a measurement unit;
    - **required/supported (flag)**: used to specify if the service is considered mandatory or preferred for the passenger.

  In case *category type* is equal to *Single Passenger*, the supported passenger category is further characterised by:

  - **passenger characteristic**: a characteristic of the passenger (e.g., age) that is used to identify the category. It is defined through:
    - **attribute name**;
    - **constraint expression**: in case, can be used to specify the required characteristic using an operator (max, min, exact value) and a measurement unit. An example is *age < 65 years old*;
    - **required/supported (flag)**: used to specify if the characteristic is considered mandatory or preferred.
  - **resource**: a resource owned by the passenger (e.g., a fidelity card) that is used to identify the category. It is defined by:
    - **resource type**: the type of owned resource;
    - **constraint expression**: in case, can be used to characterise the requested resource using an operator (max, min, exact value) and a measurement unit;
    - **required/supported (flag)**: used to specify if the resource is considered mandatory or preferred.
  - **travel equipment**: an equipment owned by the passenger (e.g., a luggage, a bicycle) that is used to identify the category. It is defined by:
    - **equipment type**: the type of owned travel equipment;

- **constraint expression**: in case, can be used to characterise the owned equipment using an operator (max, min, exact value) and a measurement unit;
- **required/supported (flag)**: used to specify if the equipment is considered mandatory or preferred.

In case *category type* is equal to *Group of Passenger*, the supported passenger

category is further characterised by:

- **group definition**: the description of the group in terms of type (e.g., Family) and size (min, max, exact value).

The *Methods* section contains the following attributes:

- **description location**: URL where the method's documentation (e.g., WSDL URL, Swagger URL) can be downloaded;
- **service type**: used to identify the Travel Expert service type (i.e., SOAP service or web API);
- **security protocol:** security protocol associated to the method. It is functional dependent and it overrides the service level security protocol (specified in the Policy section);
- **list of supported passenger categories**: the subset of supported passenger categories specified in the Policy section that are supported by the method;
- **method name**: the name of the method;
- **required parameters**: set of parameters required for the method invocation;
- **optional parameters**: set of optional parameters for the method invocation (e.g., supported search options);
- **output**: a concept (e.g., ticket, product list) that identifies the output produced by the method. It is not the actual data schema, just the concept from the ontology;
- **data mapping**: URL of the translation service OR code snippet (its URL) to be executed when translating messages from the IT²RAIL model to the specific implementation model (and back).

  In case *service type* is equal to web API, the method description is further characterised by:

  - **input format**: supported input format (e.g., JSON, XML);
  - **input schema**: String containing the input schema (XSD or JSON-Schema) OR URL pointing to the XSD/JSON-Schema;
  - **output format**: supported output format (e.g., JSON, XML);
  - **output schema**: String containing the output schema (XSD or JSON-Schema) OR URL pointing to the XSD/JSON-Schema;

- **fault format**: supported fault format (e.g., JSON, XML);
- **fault schema**: String containing the fault schema (XSD or JSON-Schema) OR URL pointing to the XSD/JSON-Schema

Basically, if the Travel Expert service is SOAP-based, then its methods' description just contains a pointer to its WSDL specification and each method is just described by a name, required/optional parameters, output, supported passenger categories, security protocol and (if available) the pointer to the data mapping service. If the Travel Expert service is Web API or REST-based, then the full set of input/output/fault message structures and formats of each callable method must be specified.

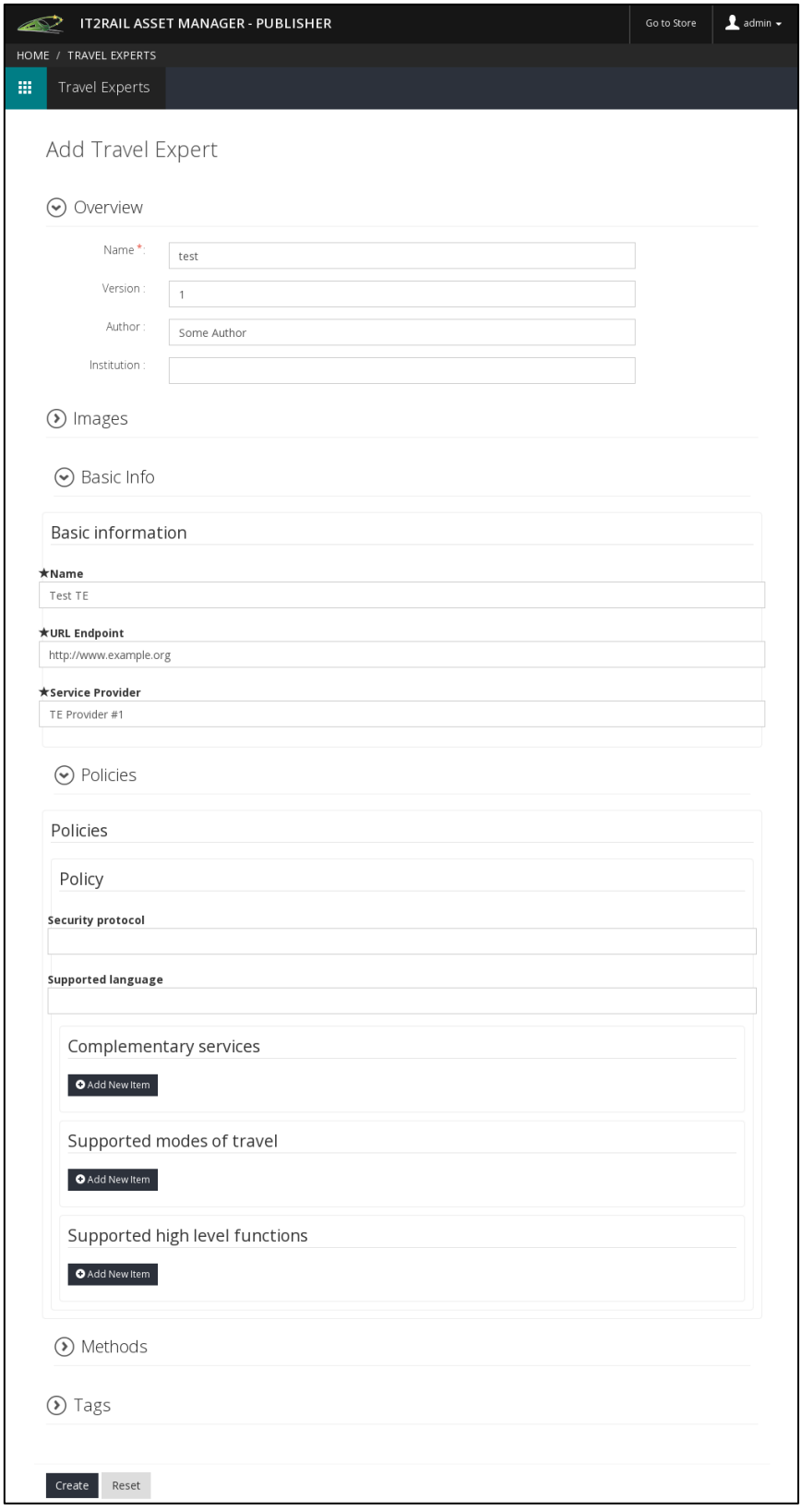


Figure 13: The form for editing a *Travel Expert description* asset

As emerge from the descriptor, a Travel Expert (TE) description is not meant to provide data for automatic orchestration of the callable methods inside a high-level function. During the design phase, developers explore the semantic Web service registry and decide which Travel Experts are going to be integrated inside their applications. Then, they write the code required to implement the orchestrations needed by the high-level functions. During runtime, the TE Resolver is called with the aim of not being forced to invoke all the known Travel Experts. The TE Resolver therefore applies a "server-side" filtering, and the caller can then apply a "client-side" filtering using the descriptions of the matching Travel Experts to further reduce the number of Travel Expert invocations.

Figure 13 above shows an excerpt of the form allowing the editing of a Travel Expert description asset.

3.3 JSON SCHEMAS

A JSON schema is another asset type managed by the Asset Manager. Basically, the JSON Schema, used to define an asset descriptor (e.g., the Travel Expert descriptor) and auto-generate its editing form, is in turn an asset to be managed.

Figure 14 shows the form for editing a JSON Schema asset. After the attributes used to identify internally the asset and its publisher (i.e., **name, version, author, author email, institution, description** and **expected validity**), the *Content* section supports the upload of the **Schema** and the specification of **AlpacaJS options**, **AlpacaJS view options** and **AlpacaJs Layout**. Such specifications represent the configuration of the Alpaca Javascript libraries that enable the auto-generation of the asset editing form from the JSON Schema (see Section 2.1.1).

The *Images* and *Tags* sections support the association of images and metadata/tags to the JSON Schema Description.

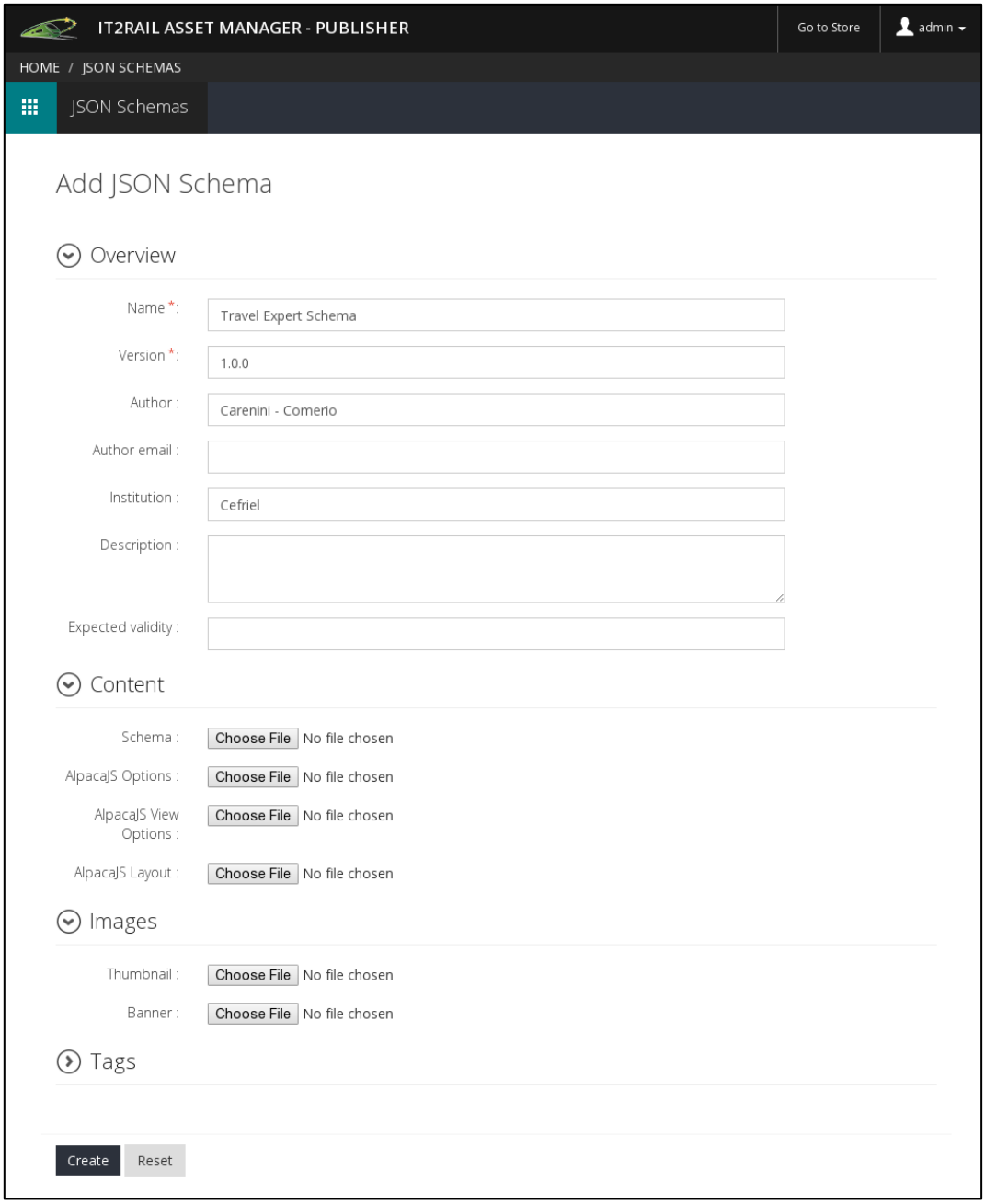


Figure 14: The form for editing a *JSON Schema* asset

3.4 RDF DATASETS

RDF Datasets are assets managed by the Asset Manager. Figure 15 shows the form for editing a RDF Dataset asset. Similarly to the Ontology asset, in the *Overview* section of the RDF Dataset editing form, the following attributes are specified:

- **name**: the name given to the RDF dataset;
- **version***:* a specific historical description of the RDF dataset;
- **author, author email** and **institution***:* the entity responsible for making the RDF dataset available;
- **domain**: the reference domain of the RDF dataset;
- **description**: a brief explanation of the RDF dataset;
- **expected validity**: the date until when the dataset is supposed to be valid.

The *Content* section supports the RDF dataset upload and the specification of:

- **graph name**: the name associated to the RDF graph;
- **existing Graph URI:** the identification of related existing graph;
- **follow imported ontologies:** the statement about the usage of imported ontologies.

The *Images* and *Tags* sections support the association of images and metadata/tags to the RDF Dataset Description.

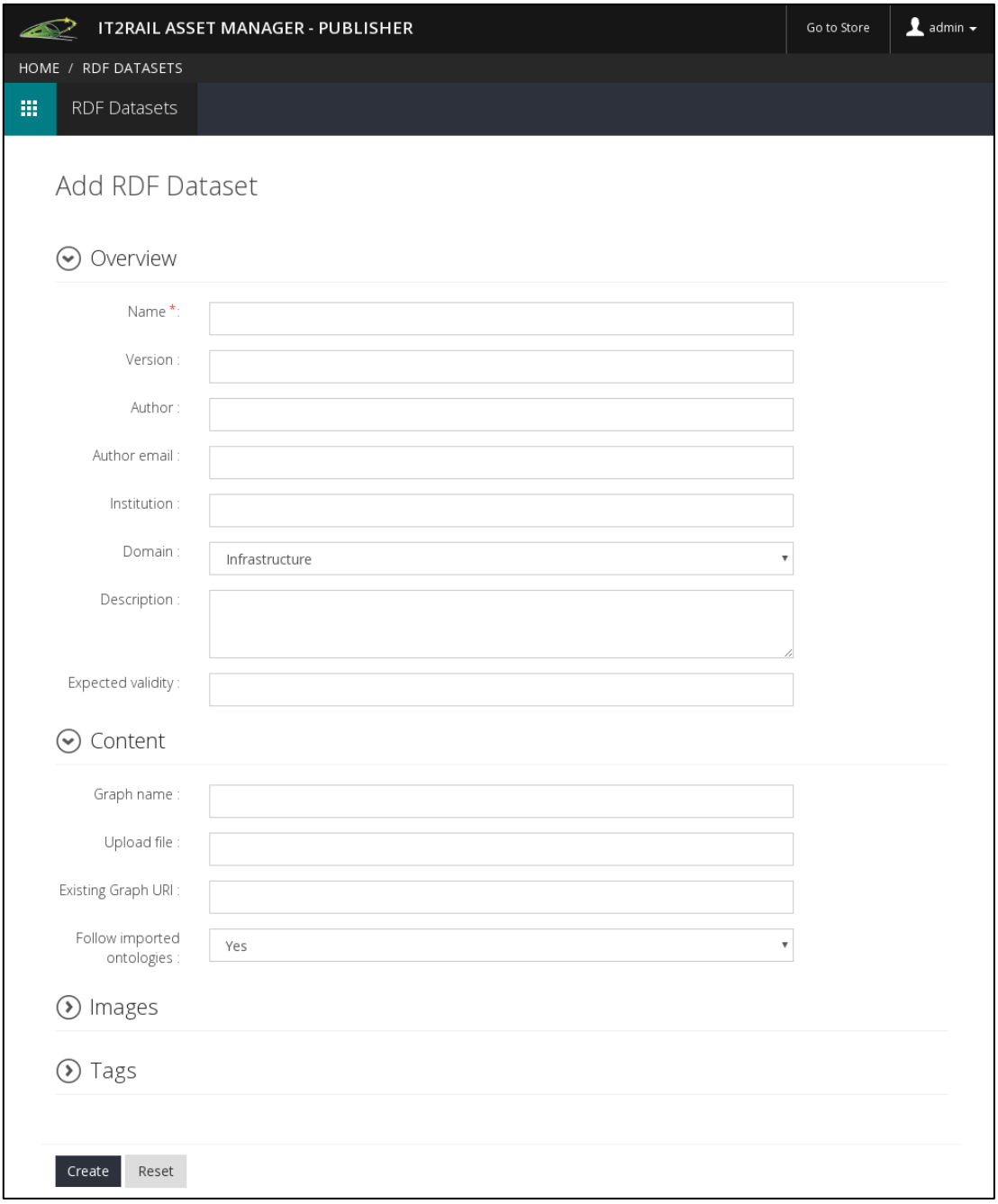


Figure 15: The form for editing a *RDF Dataset* asset

3.5 OTHER ASSETS

Since the Assets Manager is the IT²RAIL Interoperability Framework component devoted to govern the process of maintaining a shared repository of assets enabling different actors from several companies to exchange data, the list of such asset types can change over time. New asset types can be defined when become useful to be shared and available ones can be remove when unused or outdated.

In particular, the Asset Manager can be dynamically enriched with new asset types when new requirements come from ongoing Shift2Rail projects. In the following, a brief list of potential additional assets that can be managed through the Asset Manager is proposed:

- **Booking engine descriptions**: ongoing or incoming Shift2Rail projects could be interested in using the Assets Manager for sharing descriptions of available booking engines. This asset, together with the already available Travel Expert descriptions, could be a specification of a more generic asset type: the *Service Description* asset.
- **Tapping module descriptions**: similarly, ongoing or incoming Shift2Rail projects could be interested in another service description type covering the main features of a Tapping module application.
- **Resolvers**: since the Resolvers (e.g., *location resolver* and *travel expert resolver*) are fundamental components of the Interoperability Framework, ongoing or incoming Shift2Rail projects could be interested in sharing specific resolver configurations.
- **Datasets (DCAT-AP)**: The DCAT Application profile (DCAT-AP) [1] is a specification based on the Data Catalogue vocabulary (DCAT) [2] for describing public sector datasets in Europe. Its basic use case is to enable cross-data portal search for data sets and make public sector data better searchable across borders and sectors. This can be achieved by the exchange of descriptions of datasets among data portals. Similarly, ongoing or incoming Shift2Rail projects could be interested in using DCAT-AP (or an extended application profile) to share public datasets relevant for the railway and public transport sectors.
- **Semantic transformation rules**: since the Interoperability Framework should support the transformation between messages defined according to different data models, the adopted semantic transformation rules could be of interested for ongoing or incoming Shift2Rail projects (e.g., the Semantic Transformations for Rail Transportation (ST4RT) project).

4. REFERENCES

[1] EU Commission. DCAT application profile for data portals in Europe. Available at: https://joinup.ec.europa.eu/asset/dcat_application_profile/description

[2] F. Maali and J. Erickson. Data Catalog Vocabulary (DCAT). W3C Recommendation. Available at: http://www.w3.org/TR/vocab-dcat/