Deliverable 2.1

Design Document

Verein zur Förderung der selbstständigen Nutzung von Daten
2540 Bad Vöslau
ZVR: 789007092

Contact: office@ownyourdata.eu

Content

1 Introduction

Semantic Containers enable secure and traceable data exchange between multiple parties. The solution is open source, standards-based, and offers a lightweight infrastructure to make open and commercial data available in an auditable and reproducible manner.

1.1 Goals

This section summarizes the design goals and principles of the Semantic Container architecture.

Goal	Description
decentralized	The architecture should eliminate the requirement for centralized authorities or single points of failure in data exchange, including the registration of globally unique identifiers, public verification keys, service endpoints, and other metadata.
provisioned	The architecture should provide a defined, reproducible, and automatically verifiable data status.
open source	The solution should be available to the general public for use or modification from its original design.
discoverable	The architecture should make it easy and quick to find suitable data and allow to identify and compare similar data sets.
packaged	The architecture should provide mechanisms to combine data, semantic description and program logic in one distribution mechanism.
composable	The architecture should allow to combine semantic containers in sequence for creating data processing pipelines.
tradeable	The architecture should make it easy to trade data in a secure manner and unambiguously describe the usage rights for the data.
simple	The architecture should be (to paraphrase Albert Einstein) "as simple as possible but no simpler". This includes a developer friendly environment and extensive documentation for users.
portable	The architecture should be system- and network-independent.
standards-based	The architecture should make use of available standards where possible and sensible.

2 A Motivating Scenario

Scenario Description

An insurance company wants to check if damage reports about earthquake damage match with actual seismic events in the vicinity of the reporter. ZAMG (Central Institute for Meteorology and Geodynamics in Austria) provides seismic events as Open Data.

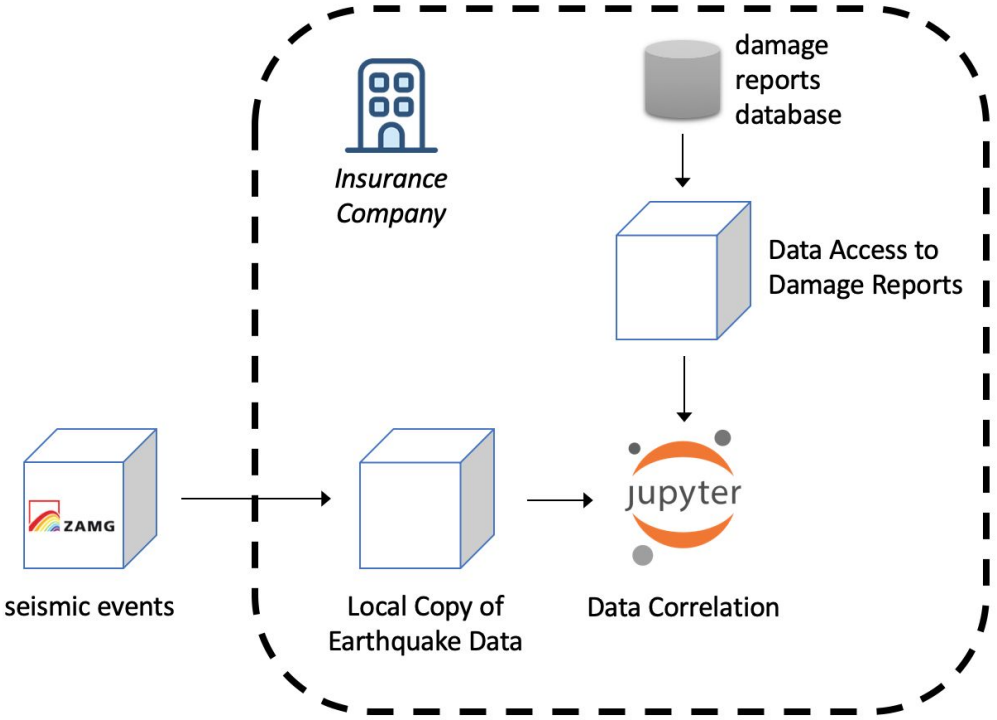


Figure 2.1. An Overview of a Semantic Container Scenario for an Insurance Company

Role Description

- Data Collector: Insurance Company
- Data Provider:
  - ZAMG: seismic events provided as dynamic Semantic Container at https://vownyourdata.zamg.ac.at:9500/api/data
    note: a dynamic container redirects any requests to the original data source outside of the container; by contrast, a static container responds with data that is stored within the container.
  - Insurance Company: damage reports from an internal database made available via an internal IP.
- Container Developer: the following containers are developed by the Insurance company:
  - Local Copy of Earthquake Data: since ZAMG seismic events are only a snapshot of the last month, the insurance company wants to create a local archive of all seismic events; therefore, a static Semantic Container is set up and daily queries write the current ZAMG data into the local data store (the static Semantic Container).
  - Data Access to Damage Reports: to provide easy access to the damage reports database, a dynamic Semantic Container is also set up (other benefits of the Semantic Container in this scenario are caching mechanisms to reduce the number of accesses to the damage report database, provide only the relevant fields, and log who accesses the database)

- Data User: a data scientist combines earthquake data and damage reports in a Jupyter Notebook and visualizes results (highlighting seismic events and damages as well as possible fraud)

Workflow / Data Flow

1. A recurring task (performed daily) reads from the ZAMG seismic events container the current data and writes it into the local store (static Semantic Container "Local Copy of Earthquake Data" available at `http://localhost:4000/api/data`)
   ```
   $ curl -s "https://vownyourdata.zamg.ac.at:9500/api/data?
              duration=1&lat=47.28&long=16.34&radius=1000" | \
     curl -H "Content-Type: application/json" -d "$( cat - )"
          -X POST http://localhost:4000/api/data
   ```
2. The dynamic Semantic Container *"Data Access to Damage Reports"* provides access to damage reports and is available at `http://localhost:4100/api/data`
3. In the Jupyter Notebook, the data scientists correlates data from the two Semantic Containers.

Usage Policies for each container

- *ZAMG seismic events:* usage policy allows to use and distribute the seismic information to other recipients, as long as the given policies allow to do so.
- *Local Copy of Earthquake Data:* usage policy, stating that the data is only used by the specific stakeholder.
- *Data Access to Damage Reports:* no usage policy is provided

Tutorials

The following tutorials will provide insight into developing Semantic Container using the scenario outlined above:

1. Setup Static Semantic Container as local data store
   this tutorial covers the topics:
   - basic commands to start and stop a Semantic Container
   - configure a container (`init.trig`)
   - defining and using usage policies
2. Develop Dynamic Semantic Container
   this tutorial covers the topics:
   - the Semantic Container base image
   - developing and integrating custom functions
   - publishing Semantic Containers
3. Access Data from Semantic Containers in Jupyter Notebooks
   this tutorial covers the topics:
   - accessing data in Semantic Containers
   - checking the Provenance of data

3 Semantic Container Architecture

The following sections provide a detailed description of all components, necessary to operate a Semantic Container.

3.1 An Overview

Figure 1 provides an overview of the components in a Semantic Container architecture. We categorized the components based on the four types of involved stakeholders: (1) **Maintainer**, (2) **Container Developer**, (3) **Data Provider/Collector**, and (4) **Data User**. We will explain each category in the following paragraphs.

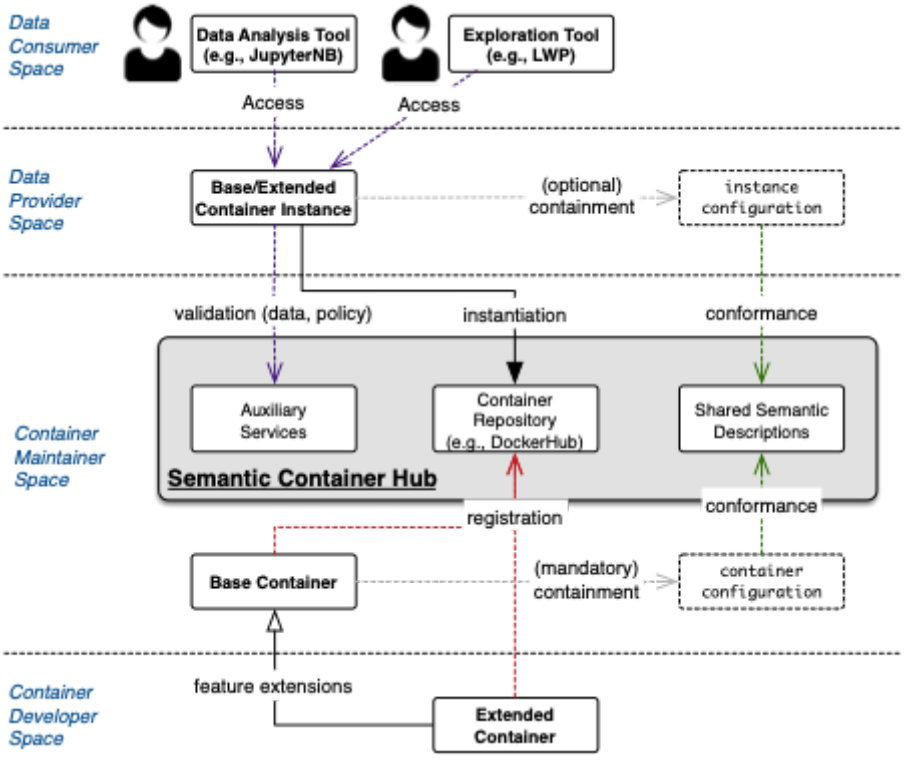


Figure 1. Initial Semantic Container Architecture

The central part (or **the Container Maintainer Space**) of the Semantic Container mainly concerns its maintainer and consists of two components: **Semantic Container Hub** and **Base Container**.

- **Semantic Container Hub**. The Semantic Container Hub provides interfaces between all stakeholders of Semantic Container Platform and consists of three sub-components: (1) *Container Repository*, where all semantic containers are stored and shared, (2) *Auxiliary Services* that provides relevant services that is used by semantic container instances, and (3) *Shared Semantic Descriptions* that defines vocabularies used in the containers to enable interoperability between various semantic containers.

- **Base Container**. The base container can be viewed as the "basic template" for all semantic containers. It defines the core functionality/services of a semantic container[1] (e.g., read, write, delete, get metadata, etc). The Base Container contains a ***container configuration*** document that includes *instance configuration* specification which will be used to validate instances of the semantic container in Data Provider space.

The Container Developer Space is where container developers could develop his/her own containers that extends the base container to include specific functions that are necessary for specific application or domains. **The Extended Container**, as we call it, would need to implement all functions of base container, while adding its own methods/functions as required. Both base container and extended containers need to be registered in the Hub, making sure that they can be accessed by the Data Provider, and eventually Data User.

The containers stored in the Hub will not be useful unless users start instantiating and putting data into them. To this end, *Data Provider* (who wants to share or sell its data) or *Data Collector* (who wants to collect data from users) can instantiate the container from the Hub in their own environment within **Data Provider Space**. In this space, the stakeholders can instantiate one or more (base and/or extended) **Semantic Container instances**. Each instance can (optionally) contain an ***instance configuration*** containing semantic description to describe contained data and its corresponding policy.

The semantic description makes them interoperable and facilitates seamless integration and processing of data from different sources and providers. Semantic containers use a set of standard vocabularies for *content descriptions* that specify the semantics of their respective input and output. The use of shared vocabularies also allows auxiliary services of a Semantic Container Hub to validate the content of semantic containers.

In the end, data consumers works in **Data Consumer Space**, utilizing data gathered from semantic container instances and other sources to conduct analysis using existing data analysis tools, such as Jupyter Notebook, or utilizing data integration platform such as Linked Widget Platform (LWP) to orchestrate data processing pipeline for data analysis and integration.

3.2 Interoperability Layers

Information about the data to be processed can be available on various levels. For some use cases, it is sufficient to have only limited metadata available, for others there is also a need for a well-specified syntax for all available records. Ideally, though, it is possible to describe the exact semantics of the data at hand. The Semantic Container concept acknowledges that data sources are not necessarily specified in a well-defined manner and the proposed solution enables users to enrich the available data in a step-by-step approach.

A Semantic Container describes its input and output on three levels:

1. **Metadata Level:** every container includes the publisher of the data as well as a verbal description of the content; additionally, the allowed usage, provenance, and optional billing information is provided.

[1] the full list of the functionality is listed in: https://api-docs.ownyourdata.eu/semcon/

2. **Syntax Level:** at this level, it is possible to specify and validate the data format (syntax) of the data processed and stored by the container; this information can be used to check compatibility between containers
3. **Semantic Level:** the highest level of interoperability includes metadata and syntax information as well as a semantic description about the individual data attributes; on this level, it is possible to automatically validate data and ensure a defined level of data quality.

3.3 Semantic Descriptions

This section explains the semantic descriptions used in the context of Semantic Containers. We use the Resource Description Framework (RDF), Web Ontology Language (OWL) and Shape Constraints Language (SHACL) to describe semantic containers.

3.3.1 Image Configuration

The image configuration (usually provided in the file "image-constraints.trig") consists of several RDF graphs:

(1) The **default graph**, which consists of
  (a) Ontology Version: **`http://w3id.org/semcon/ns/ontology/0.0.1`**
  (b) SHACL constraints for validating the semantic container configuration (see next section)
(2) The named graph **`:ImageConfiguration`**
  (a) Listing all available services, e.g.,
    (i) `:usagePolicyValidationService` (for validating usage-policy)
    (ii) `:initValidationService` (for validating the container configuration)
  (b) A placeholder for holding global configuration of a Semantic Container
(3) The named graph **`:ImageModel`**
  (a) Stores the **Semantic Container** ontology;
  (b) The Semantic Container ontology contains all classes and properties used within the semantic container context.

Appendix A lists an example `image-constraints.trig` file.

3.3.2 Container Configuration

Whereas the image configuration contains generic information that universally holds for all semantic containers derived from an image, the container configuration (usually provided in a file named **`init.trig`**) holds the container-specific configuration for each instance. Note that providing a container configuration is optional.

The container configuration file consist of several named RDF Graphs as follows:

(1) The named graph `:BaseConfiguration`, which consists of two resources:
  (a) `:ContainerConfigurationInstance`, which contains the general information about the container, such as: title, description, creator, contributor; providing this resource is mandatory.
  (b) `:DataConfigurationInstance`, which contains the information about data contained within a semantic container; providing this resource is optional.
(2) The named graph `:UsagePolicy`

(a) Lists the usage policy of the container.
(b) Output data from this container will be delivered with this usage policy.
(c) All incoming/contributing data must conform with this policy.

(3) The named graph `:DataModel`
   (a) Provides information on the RDF data model used within the container
   (b) **Can be empty**, only used if the data is in RDF serializations (or can be mapped to its RDF format using mapping in named graph: `:DataMapping`) and the container interoperability is in Semantic level.

(4) The named graph `:DataConstraint`
   (a) Provides constraints of the RDF data model used within the container.
   (b) To make sure that the incoming data conforms with the data model described in `:DataModel`.
   (c) **Can be empty,** only used if the data is in RDF serializations (or can be mapped to its RDF format using mapping in named graph: `:DataMapping`) and for containers with Semantic level interoperability.

(5) The named graph `:DataMapping`
   (a) Provides the mapping between the original data format (e.g., CSV) to the RDF data model described in `:DataModel`
   (b) **Can be empty,** only used if the data is in non-RDF serializations and the container interoperability is in Semantic level.

Appendix B lists an example `init.trig` file.

3.3.3 Usage Policy

According to the SPECIAL project[2], "*a Usage Policy is meant to specify a set of authorized operations*". Within a Semantic Container, such authorized operations are characterized by:

- *Data Categories:* the data processed by the operation
- *Purpose:* the purpose of the operation
- *Processing:* a description of the operation itself
- *Recipient:* the entities that can access the result of the operation
- *Storage:* a description of
  - *Location:* where the result is stored and
  - *Duration:* for how long

Appendix C lists the available attributes for each of the elements above.

3.3.4 Provenance

The Provenance information in a Semantic Container provides an audit trail for the data stored in the container. Based on to the PROV-Ontology (https://www.w3.org/TR/prov-o/), we use the following three classes to specify provenance:

- **Entity:** used to describe data stored in a Semantic Container:
  - sha256 hash value of the data
    (sha256hash12 are the first twelve digits of the sha256 hash value)

[2] https://www.specialprivacy.eu

- reference to Semantic Container where the data is stored (containerId12 are the first 12 digits of the container unique id)
- timestamp when the provenance information was generated

```
scr:data_{sha256hash12} a prov:Entity;
    rdfs:label "data set from {current date & time}"^^xsd:string;
    sc:dataHash "{sha256 hash of data set}"^^xsd:string;
    prov:wasAttributedTo scr:container_{containerId12};
    prov:generatedAtTime "{current date & time}"^^xsd:dateTime.
```

- **Agent:** provides information about the container:
  - general information such as title and description
  - unique id of the container
  - sha256 hash value of image
  - reference to the container operator

```
scr:container_{containerId12} a prov:softwareAgent;
    sc:containerInstanceId "{unique id from container}"^^xsd:string;
    sc:imageHash "{sha256 hash of image}"^^xsd:string;
    rdfs:label "{title from container configuration}"^^xsd:string;
    rdfs:comment "{description from container configuration}"^^xsd:string;
    prov:actedOnBehalfOf scr:operator_{sha256hash12};
.
scr:operator_{sha256hash12} a foaf:Person, prov:Person;
                    # or: foaf:Organization, prov: Organization;

    sc:operatorHash "{sha256 hash of operator 'name <email>'}"^^xsd:string;
    foaf:name "{operator name from container configuration}";
    foaf:mbox <{operator email from container configuration}>;
.
```

- **Activity:** provides information about the data source (if available):
  - reference to entity (dataset) of imported data by using sha256 hash this in turn can include a complete set of provenance data following the same structure
  - begin and end data & time for reading data
  - reference to entity this data source belongs to

```
scr:input_{sha256hash12} a prov:Activity;
    sc:inputHash "{complete sha256 hash of serialized data}"^^xsd:string;
    rdfs:label "input data from {current date & time}"^^xsd:string;
    prov:used scr:data_{sha256hash12}; # optional, point to new data below
    prov:startedAtTime "{start time of data input}"^^xsd:dateTime;
    prov:endedAtTime "{end time of data input}"^^xsd:dateTime;
    prov:generated data_{sha256hash12}. # point to existing/main data above
.
```

3.4 Services

Semantic Containers require a number of services to function properly. The source code for an implementation of these services is available on the public Github repository: https://github.com/sem-con/srv-semantics

3.4.1 Container Configuration Validation

This service is referred to in the :ImageConfiguration as :initValidationService and is used for validating the container configuration (`init.trig`) file.

- Location: https://semantics.ownyourdata.eu/api/validate/init
- Swagger documentation: https://api-docs.ownyourdata.eu/semcon-validation/
- Input: the input file is of type application/json to wrap a turtle file using JSON key
  - `container-config` for the content of the `init.trig` file
  - `image-constraints` for the content of the `image-constraints.trig` file
- Output:
  - HTTP response code 200: if the `init.trig` file conforms to the `image-constraints.trig`
  - HTTP response code 500: otherwise (plus the explanation of the violation as defined in https://www.w3.org/TR/shacl/#validation-report)

3.4.2 Usage Policy Matching

This service is referred to in the `:ImageConfiguration` as `:usagePolicyValidationService` and is used for **validating usage-policies.**

- Location: https://semantics.ownyourdata.eu/api/validate/usage-policy
- Swagger documentation: https://api-docs.ownyourdata.eu/semcon-validation/
- Input: the input file is of type application/json to wrap a turtle file using JSON key "usage-policy" and must contain the following resources:
  - `sc:DataSubjectPolicy` representing the usage policy of user, and
  - `sc:DataControllerPolicy` representing the usage policy of the data controller.
- Output:
  - HTTP response code 200: if the policy of controllers is not violating the policy of users
  - HTTP response code 500: otherwise

3.4.3 Data Validation

This service is referred to in the `:ImageConfiguration` as `:dataValidationService`; it is used for validating input data according to the data specification (provided through the `init.trig` file as SHACL definition).

- Location: https://semantics.ownyourdata.eu/api/validate/data
- Swagger documentation: https://api-docs.ownyourdata.eu/semcon-validation/
- Input: the input file is of type application/json to wrap input data and constraints using keys:
  - "`data`" for the RDF Graph to be validated
  - "`constraints`" for the SHACL constraints for the RDF graph
- Output:
  - HTTP response code 200: if the data conforms with the constraints

- HTTP response code 500: otherwise (plus the explanation of the constraint violation as defined in https://www.w3.org/TR/shacl/#validation-report).

3.4.4 Blockchain

This service is referred to in the `:ImageConfiguration` as `:blockchainService` and is used for storing hash values in a distributed ledger for immutability.

- Location: https://blockchain.ownyourdata.eu/api/doc
- Swagger documentation: https://api-docs.ownyourdata.eu/notary
- Input: the input is a hash (string with 64 characters
- Output:
  - HTTP response code 200: if the hash was successfully processed together with information how to validate immutability of the hash value
  - HTTP response code 500: otherwise

3.4.5 Billing

This service is referred to in the `:ImageConfiguration` as `:billingService` and is used for providing the following information and services to a Semantic Container.

- Location: due to the individual nature of the service a user has to run his/her own instance of this service
- Swagger documentation: https://api-docs.ownyourdata.eu/semcon-billing/
- Input for starting the service
  - Private GPG Key including Passphrase for specified Email
  - Private Key to access Ethereum account
  - API Key from Etherscan.io

The following API endpoints are provided by the service:

- /api/encrypt - encrypt string for given email with public key
- /api/decrypt - decrypt string with private key
- /api/sign - sign string with private key
- /api/verify - verify signature for given email
- /api/payment_info - all necessary payment details in the course of the billing workflow
- /api/payment_terms - price calculation and validity for a given request
- /api/transaction - query information about a given transaction

3.4.6 DMA Integration

To register a Semantic Container in the Metadata-Store of the Data Market Austria (DMA) the corresponding Swagger file has to be submitted to the DMA Metadata-Store.

3.5 Authorization and Billing

Containers are secured against remote access through the API. Note that the current architecture does not provide access control for local access to the container.

3.5.1 Authorization

Access control through the API is optional and the implementation is based on OAuth 2.0. It is activated by setting the environment variable `AUTH` to `true` on container startup. The necessary credentials to access the container are written to `STDOUT` and are available in the container logs.

Example:

```
$ docker run -d -e AUTH=true -p 4000:3000 semcon/sc-base
```

console output:

```
APP_KEY: 1de268f3478a152ae48f393842ed83488e8a1a9e0f4f5b475828de00005410ca
APP_SECRET: e77d313a84c5e346e4588d532d74ed9168246e71c842988c59aa1102df4e6feb
```

When authentication is enabled for a container, then any access to the container requires first requesting a TOKEN and afterwards providing the token in the header of a request.

Example:

```
$ curl -d grant_type=client_credentials \
    -d client_id={APP_KEY} -d client_secret={APP_SECRET} \
    http://localhost:4000/oauth/token
```

response:

```
{"access_token":"197a8130efe8933c8a99683c02f6bb46aa53146d18528350df
d03f6df6fd3e12","token_type":"bearer","expires_in":7200,"created_at
":1542362558}
```

reading data requires providing the access token in the header:

```
$ curl -H "Content-Type: application/json" \
    -H "Authorization: Bearer {ACCESS_TOKEN}" \
    http://localhost:4000/api/data
```

The following shell script provides a way to get the access token in a single statement by reading client-id and client-secret from the container logs:

```
$ export APP_KEY=`docker logs test_auth | grep APP_KEY | \
          awk -F " " '{print $NF}'`; \
export APP_SECRET=`docker logs test_auth | grep APP_SECRET | \
          awk -F " " '{print $NF}'`;
export TOKEN=`curl -s -d grant_type=client_credentials \
     -d client_id=$APP_KEY \
     -d client_secret=$APP_SECRET \
     -d scope=admin \
     -X POST http://localhost:4000/oauth/token | \
          jq -r '.access_token'`
```

The only exception of required authentication for secured containers is the API endpoint `/api/active` which also provides information about the authentication mode of the container.

To provide different levels of access, the following “scopes” are available

- admin: allows any access including creation, updating and deletion of new credentials

- write: allows reading and writing data
- read: allows only reading of data

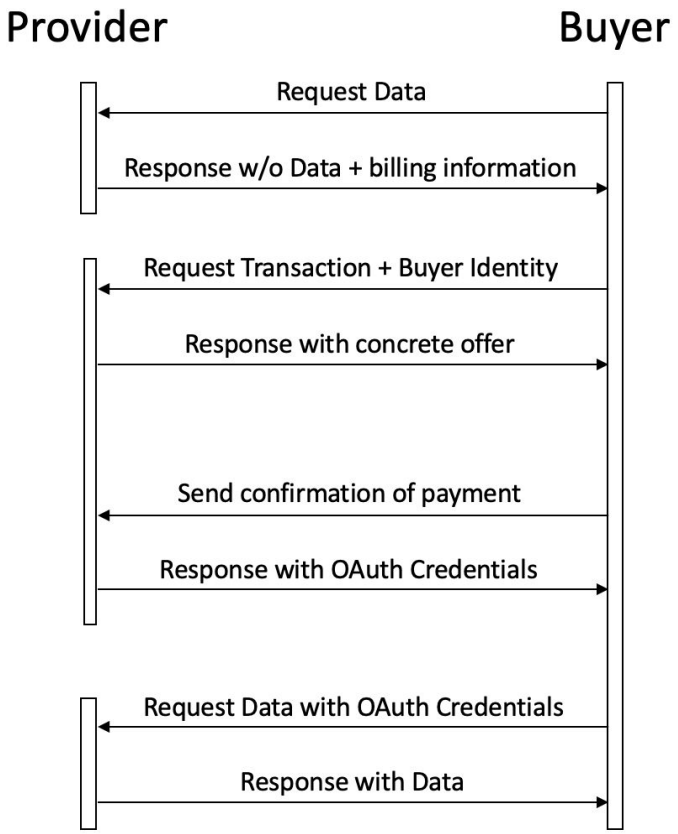
In the following list, the necessary administrative API endpoints are described through examples:

- create new applications (requires admin scope)
```
$ curl -H "Authorization: Bearer $TOKEN" \
    -d name=app_name -d scopes=admin_write_read \
    -X POST http://localhost:4000/oauth/applications
```
- remove existing application (requires admin scope)
```
$ curl -H "Authorization: Bearer $TOKEN" \
    -X DELETE http://localhost:4000/oauth/applications/2
```
- get information about token
```
$ curl -H "Authorization: Bearer {ACCESS_TOKEN}" \
    http://localhost:4000/oauth/token/info
```
- revoke token
```
$ curl -F token={ACCESS_TOKEN} \
    -X POST localhost:4000/oauth/revoke
```

The documentation for authorization is also available in the online Semantic Container API description on https://api-docs.ownyourdata.eu/semcon in the "Authorization" section.

3.5.2 Billing

A container is marked for billing by setting the environment variable `AUTH` to `billing` on startup. Any access to /api/data is then required to provide an OAuth2 token. The following process describes how to request OAuth2 credentials and a token through the billing workflow.



Step 1: Buyer submits initial request and Provider responds with billing information

The response for a request to `GET /api/data` provides the following information:

- billing
  - payment-info: free text information about data and payment
  - methods: list of payment options
  - provider: email address of seller
  - provider-pubkey-id: public key ID for GPG key of seller email address
- provision
  - usage-policy
  - provenance
- validation
  - hash
  - dlt-reference

Step 2: Buyer requests to buy data and receives concrete offer

The request to buy data is sent to `POST /api/buy` and shall include the following information:

- buyer: email address of buyer
- buyer-pubkey-id: public key ID for GPG key of buyer email address
- buyer-info: additional information about buyer as hash
- request: query string to specify data request in the format key1=value1&key2=value2
- usage-policy: usage policy to be used by the buyer for the intended data
- payment-method: chosen payment method
- signature: request string signed with private key from buyer

The response from the Semantic Container to the POST request above provides the following information:

- billing
    - uid: unique transaction ID
    - payment-info: free text information about data and payment
    - payment-method: used payment method
    - provider: email address of seller
    - provider-pubkey-id: public key ID for GPG key of seller email address
    - offer-timestamp: date & time of offer
    - offer-validity: date & time until offer is valid
    - cost: price for requested data
    - payment-address: address / account where price should be paid
    - signature: uid signed by private key of provider
- provision
    - usage-policy
    - provenance
- validation
    - hash
    - dlt-reference

With the provided information the seller can make a payment for the requested data. To unambiguously mark the transaction the UID must be written into the input field (intended use).

Step 3: Buyer confirms payment and receives encrypted OAuth2 credentials

After paying the requested amount at the specified address the buyer sends the transaction hash to the provider via `GET /api/paid?tx={transaction hash}` and receives upon successful verification the OAuth2 credentials:

- key: id for OAuth2 Client Credentials Flow
- secret: secret for OAuth2 Client Credentials Flow encrypted with public key from buyer

Step 4: Buyer accesses data

Buyer decrypts the secret, requests an OAuth2 token through `GET /oauth/token` and accesses data via `GET /api/data?options` and providing the OAuth2 token in the header.

3.6 API Description

This section lists the default API endpoints provided by Semantic Containers. Note that container developers provide additional endpoints that should be described as HTML under the address /api-docs as Swagger documentation.

Verb	URI Pattern	Description
GET	/api-docs	Swagger documentation of the API endpoints
GET	/api/active	check if container is active and secured
GET	/api/info	get container overview
GET	/api/log	show log information (for details see section 3.8)
POST	/api/meta	set container configuration
GET	/api/meta	show container configuration (meta data)
GET	/api/meta/{detail}	show specific container configuration (info, usage policy or example data)
POST	/oauth/token	request access token
GET	/oauth/token/info	show token info
POST	/oauth/revoke	revoke access token
POST	/oauth/applications	create access account
DELETE	/oauth/applications/{id}	remove account
GET	/api/data{/option}	read data (for details see section 3.7.2)
POST	/api/data	write data
POST	/api/buy	request to buy data
GET	/api/paid	confirm payment for data
GET	/api/payments	list all payments

The complete API description is also available in the online Semantic Container API description on https://api-docs.ownyourdata.eu/semcon.

3.7 Data Format

This section describes the data format to be exchanged with a Semantic Container, i.e., data sent to the container for storage and data read from the container.

The data processed by a Semantic Container is segmented into records and each record can be in one of three formats according to the interoperability layers described in section 3.2:

- *byte string* - without any defined structure
- *defined syntax* - currently, CSV, JSON and Turtle are supported and defined during the container setup (see section 3.3.2)
- *defined semantic* - as described in the Data Model graph using SHACL and provided during container setup (see section 3.3.2)

3.7.1 Structure of Input Data

Data sent via a POST request to the API endpoint `/api/data` is validated with regards to the format as specified in the container definition (made available during container setup) in :BaseConfiguration > :DataConfigurationInstance > :hasNativeSyntax

- for `:hasNativeSyntax w3c-format:Turtle` - the syntax is validated using the following parser: https://www.rubydoc.info/github/ruby-rdf/rdf-turtle/ additionally, the semantics are also validated against the :DataModel provided during container setup using the defined SHACL validation service (default: https://semantic.ownyourdata.eu/api/validate/data)
- for `:hasNativeSyntax <http://w3id.org/semcon/ns/ontology#JSON>` - the syntax is validated using the following parser: https://ruby-doc.org/stdlib-2.5.3/libdoc/csv/rdoc/CSV.html
- for `:hasNativeSyntax <http://w3id.org/semcon/ns/ontology#CSV>` - the syntax is validated using the following parser: http://ruby-doc.org/stdlib-2.5.3/libdoc/json/rdoc/JSON.html

If no native syntax is specified, the validation is omitted. If the validation fails, the data is not stored and an error message together with response code 422 is returned.

Data sent via a POST request to the API endpoint `/api/data` is processed in the following way:

1. if it is a JSON array: write each element as separate record into the container storage
    a. otherwise: write the complete string as a single new record into the container storage
2. if it is CSV: write each line as separate record into the container storage
3. if it is a valid JSON object
    a. test if it has a "content" attribute: write data as a new record into the container storage
    b. test if it has a "provision" attribute with a "content" attribute inside: write data as a new record into the container storage
4. otherwise: write the complete string as a single new record into the container storage

3.7.2 Structure of Output Data

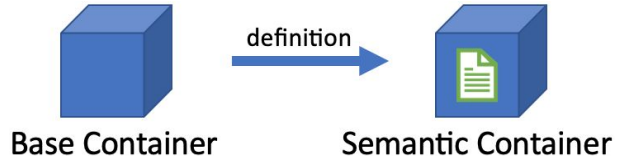
Accessing data in a Semantic Container is enabled through a number of API endpoints providing different types of information:

- complete (`/api/data/full`)
    - provision
        - content
        - content-constraints
        - usage-policy
        - provenance
    - validation
        - hash
        - trusted-timestamp
        - dlt-reference
- default (`/api/data`)
    - provision
        - content
        - usage-policy
        - provenance
    - validation
        - hash
        - dlt-reference
- provision only (`/api/data/provision`)
    - content
    - usage-policy
    - provenance
- plain: data JSON array only (`/api/data/plain`)

3.8 Container Lifecycle

The following steps describe the typical life cycle of a Semantic Container:

1) **Container setup**
   The simplest way to create a semantic container is through starting the base container (https://hub.docker.com/r/semcon/sc-base/) and providing a definition of data and usage policy.



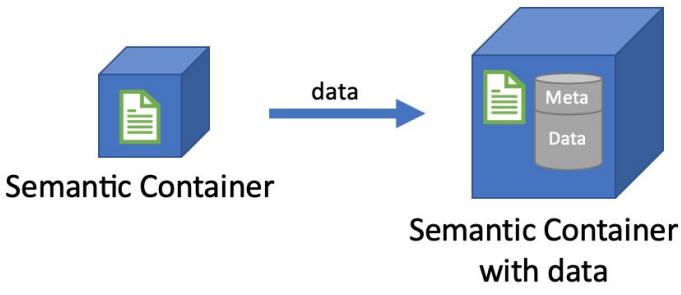
   example command:

```
$ docker run -d --name my_container -p 4000:3000 \
         semcon/sc-base /bin/init.sh "$(< init.trig)"
```

   Alternatively, you can create your own container by performing the following steps:

   a) clone the base container source repository (https://github.com/sem-con/sc-base)
      `$ git clone git@github.com:sem-con/sc-base.git`
   b) implement functionality and update Dockerfile
   c) build the container
      `$ docker build -t repo/my_container .`

2) **Data Upload**
   Data is copied into the container through the API endpoint `POST /api/data`. This method performs automatic data validation and generates a complete audit trail.

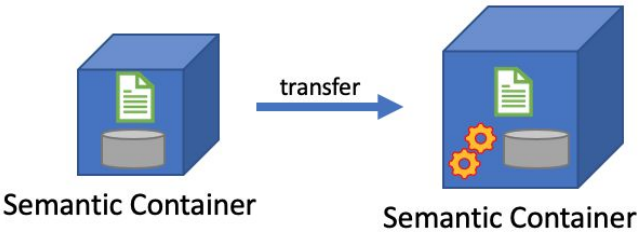


   example command:

```
$ curl -H "Content-Type: application/json" \
    -d '[{"date":"2018-05-14","value":"8753"},
         {"date":"2018-05-15","value":"10192"}]' \
    -X POST http://localhost:4000/api/data
```

3) **Data Processing**
   To perform reproducible processing steps on the data in semantic containers with specific functionality it is possible to string together containers. Best practice for Semantic Containers

is to read data from STDIN, accept options as parameters, and write the results to STDOUT. Using this approach, process pipelines can be built.



Each container should first validate incoming data (for syntax, semantics, usage policy), thereafter perform the respective processing step, and finally store the results in the internal database. All steps are again tracked in the provenance description.

example command:
apply differential privacy to data from a semantic container

```
$ curl http://localhost:4000/api/data | \
    docker run -i --name anon_data -p 4001:3000 \
           semcon/diffpriv /bin/init.sh "$(< init.trig)"
```

example command for processing pipeline:
sometimes it is necessary to perform transformation steps to use existing containers

```
$ curl http://localhost:4000/api/data | \
  docker run -i semcon/transform /bin/process.sh \
    "$(< mapping.json)" | \
  docker run -i --name anon_data -p 4001:3000 \
    semcon/diffpriv /bin/init.sh "$(< definition.json)"
```

4) Data Sharing

There are two options to make data publicly available:

a) running containers can be accessed through the API and the default endpoint is GET /api/data; to keep track who requests data and apply billing, access can be restricted via OAuth2 requiring data users to acquire Key and Secret before downloading the data (see section 3.5 Authentication and Security and section 3.4 Services, Billing)
example command to access data in a semantic container:

```
$ curl http://localhost:4000/api/data
```

b) containers can also be stopped and distributed through images stored in the respective repositories (e.g., https://www.dockerhub.com); for data access it is necessary to start the image
example commands to stop, publish and run a container / image:

```
$ docker commit my_conainer repo/my_image
$ docker push repo/my_image
$ docker run -d -p 4000:3000 repo/my_image
```

3.9 Logging

Semantic Containers log the following events by default:

- Container creation
  `{ "type":"create",`
  `"scope":"`*IMAGE_NAME (IMAGE_SHA256)*`",`
  `"request":"`*invocation script*`" }`
- Configure container (providing init.trig)
  `{ "type":"write",`
  `"scope":"meta information",`
  `"request":"`*request IP*`" }`
- Writing data into a container
  `{ "type":"write",`
  `"scope":"[`*list of unique IDs of created records*`]",`
  `"request":"`*request IP*`" }`
- Reading data from the container
  `{ "type":"read",`
  `"scope":"all (# records)",`
  `"request":"`*request IP*`" }`

Each log entry additionally contains a unique ID and a timestamp (UTC based), indicating when the record was created. The complete list of log entries can be queried through the API endpoint `/api/log`.

3.10 Data Structure

The following section describes the models used in a Semantic Container to store the relevant data.

3.10.1 AsyncProcesses

`AsyncProcesses` stores information about asynchronous processes initiated by long running requests.

The model has the following attributes:

- rid - unique request ID
- status - current status of request
- request - normalized request string, i.e., all request parameters in defined order including default values
- file_list - list of processed files
- error_list - list of errors while executing request

3.10.2 Billings

`Billings` stores information about billing requests submitted to a container that has billing activated.

The model has the following attributes:

- uid - unique billing id
- buyer - email address of buyer
- buyer_info - additional information about buyer
- buyer_address - FROM address used in transaction by buyer
- address_path - used path for address in hierarchical deterministic wallet according to BIP-32

3.10.3 Logs

`Logs` stores information about all log records for the container.

The model has the following attributes:

- item - json object with log information

3.10.4 Provenances

`Provenances` stores the provenance chain of data in the container.

The model has the following attributes:

- startTime - timestamp at begin of new input data
- endTime - timestamp at end of new input data
- input_hash - hash value of new input data
- prov - provenance statement according to PROV-O

3.10.5 Semantics

`Semantics` stores the container definition.

The model has the following attributes:

- uid - unique id of the container
- validation - RDF graph with container configuration provided on startup

3.10.6 Stores

`Stores` holds the data in the container.

The model has the following attributes:

- item - content of a single record
- prov_id - reference to provenance information for this record

3.11 Linked Widget Platform

The Linked Widget Platform (LWP) is a platform that combines semantic web and mashup concepts to support non-expert users in efficiently making use of the growing number of (linked) open and non-open data sources. In particular, the platform allows users to collaboratively and interactively integrate data in an ad-hoc and distributed manner. Each stakeholder can contribute data and computing resources to a shared data processing flow.

The LWP serves as a prime candidate to provide users with an interface to orchestrate the workflow among semantic containers. We are investigating the requirements to adapt the current LWP to allow such orchestration, and we come up with the following requirements:

Separation of GUI components from data processing logic. Currently, the interface and data processing parts in the LWP are not modularized, since the processing logic is embedded as part of the platform. In order to be able to support the orchestration of Semantic Containers, we refactor the original LWP to allow clear separation of concerns of the GUI for the users (LWP) and the data processing logic (Semantic Container).

Supports for multi-level interoperability. The original idea of the LWP is to support ad-hoc integration of only RDF Graph data. In the scope of SEMCON project, we extend the LWP with the capability to support non RDF-Graph data, in line with the three layers of interoperability proposed within the SEMCON project (see Section 3.2).

An extended semantic metadata. The original LWP focuses on providing users with the provenance information of processed data, which is reflected in the semantic metadata addressing this issue. We extend the metadata with specific metadata, required in the SEMCON project, such as privacy policies and definitions of constraints .

4 Evaluation Setup

This chapter describes a set of use cases and KPIs to evaluate performance and practicability of Semantic Containers.

4.1 Use Cases

In the course of the nine-month funding period, three use cases were selected to showcase various aspects of Semantic Containers.

4.1.1 Seismic Data

Purpose: provide a list of seismic activities within a radius for a given coordinate in a given timespan
Semantic Container URL: https://vownyourdata.zamg.ac.at:9500

General information:

- Data Provider: ZAMG
- publicly available raw data: http://geoweb.zamg.ac.at/static/event/lastday.json

The use case demonstrates:

- semantic description of data
- confirmed source through signature in provenance
- confirmed data integrity through blockchain
- integration in OwnYourData

4.1.2 Weather Data

Purpose: access to hourly data from TAWES (teilautomatische Wetterstationen) in the last 30 years
Semantic Container URL: https://vownyourdata.zamg.ac.at:9600

General information:

- Data Provider: ZAMG
- subset of data which is publicly available: https://www.data.gv.at/katalog/dataset/9b40a0af-a6fe-47ff-9624-2ea8f40c746f

The use case demonstrates:

- filter and aggregate existing data
- billing workflow

4.1.3 EODC Satellite Data

Purpose: show time lapse satellite images for a given coordinate form EU Copernicus Program (data from Sentinel 2 satellites is used)
Semantic Container URL:

- Data download: https://vownyourdata.zamg.ac.at:9700
- Atmospheric correction: https://vownyourdata.zamg.ac.at:9701
- Image clipping: https://vownyourdata.zamg.ac.at:9702

General information:

- Data Provider: EODC
- publicly available raw data: ftp://galaxy.eodc.eu

The use case demonstrates:

- test performance with large data sets
- container pipelines
- visualization of the container pipeline in Linked Widget platform

4.2 Evaluation Criteria

The use cases described in section 4.1 are evaluated against the following criteria:

- continuous operation for 1 week and statistics about
    - number of data records made available
    - number of data accesses
- stress test data: based on typical usage from continuous operation stress test scenarios are designed, simulated and evaluated
- documented data flow from source to end user
  note: the OwnYourData data vault acts as end user / data consumer (although FFG stated that this work is not funded)
- description of benefits of Semantic Containers in contrast to current situation

4.3 Prototype Development Plan

Container Roadmap:

1. BASIC: container w/o configuration - done
    - general functionality - done
        - dump any data into container; format: array, each item is 1 record
        - create logging information for any data request
        - process pipeline for simple use cases
    - scripts
        - init.sh
        - init.rb
    - APIs
        - POST /api/data
        - GET /api/data
        - GET /api/info
        - GET /api/logs
    - testing infrastructure
        - size of image
        - time to write 1e[3..6] records
        - test case to check logging
    - required for ACCESS, META
2. ACCESS: container with OAUTH2 for access restriction - done

- prerequisite: BASIC
- environment variables
  - AUTH
- scripts
  - /bin/apps.sh - print credentials
- APIs
  - POST /oauth/token - ok
  - GET /oauth/token/info - ok
  - POST /oauth/revoke - ok
  - POST /oauth/applications/new
  - DELETE /oauth/applications/{id}
- testing
  - access container only with correct credentials
- required for BILLING

3. META: start container and set meta information - done
   - prerequisite: BASIC
   - scripts
     - /bin/init.sh - read configuration file and set meta data
       (also possible with POST /api/meta)
   - APIs
     - POST /api/meta
     - GET /api/meta
     - GET /api/meta/info
     - GET /api/meta/usage
     - GET /api/meta/example
   - services
     - init.trig validation service
   - testing
     - validate meta information
   - required for USAGE, SYNTAX, PROVENANCE
4. USAGE: provide usage policy and validation - done
   - prerequisite: META
   - scripts
     - /bin/init.sh - extend to accept meta data and usage policy
   - APIs
     - GET /api/meta/usage
     - POST /api/data - extend to accept usage policy together with data and validate this usage policy against policy of container
   - services
     - usage policy validation service
   - required for BILLING
5. SYNTAX: enable syntax interoperability layer - done
   - prerequisite: META
   - APIs
     - GET /api/data/plain
   - services

        - syntax validation of input data
    - supported syntaxes
        - CSV
        - JSON
        - RDF (Turtle)
    - required for SEMANTIC
6. SEMANTIC: enable semantic interoperability layer - done
    - prerequisite: SYNTAX
    - APIs
        - GET /api/data/full
        - GET /api/data/provision
    - services
        - SHACL validation of input data
    - required for: D5.2
        - ZAMG Seismic
        - ZAMG TAWES
7. PROVENANCE: include complete provenance trail - done
    - prerequisite: META
    - services
        - blockchain
    - required for: D5.2
        - ZAMG Seismic
        - ZAMG TAWES
        - EODC
8. BILLING:
    - prerequisite: ACCESS and USAGE
    - services
        - billing
    - required for D5.2
        - ZAMG TAWES
9. LINKED-WIDGETS: visualization of pipelines in Linked Widgets Platform
    - prerequisite: SYNTAX & PROVENANCE
    - required for D4.2

5 Related Work

The following related initiatives and solutions were identified during the work on Semantic Containers:

- **BEST:** Achieving the benefits of SWIM by making smart use of semantic technologies
  Web: http://project-best.eu/
- **Data Market Austria (DMA):** The Data Market Austria Project is creating a Data-Services Ecosystem in Austria by advancing technology foundations for secure data markets and cloud interoperability, and creating an environment encouraging data-centred innovation.
  Web: https://datamarket.at/en/
- **Data Package:** how to share data, open data, and research objects
  Web: https://frictionlessdata.io/data-packages/
- **IHAN:** building an ecosystem for a fair data economy, together with citizens and different organisations
  Web: https://www.sitra.fi/en/topics/fair-data-economy/
- **Open Algorithms (OPAL):** The goal of the OPAL project is to make a broad array of data available for inspection and analysis without violating personal data privacy
  Web: https://openalgorithms.mit.edu/
- **Social Linked Data (SOLID):** The project aims to radically change the way Web applications work today, resulting in true data ownership as well as improved privacy.
  Web: https://solid.mit.edu/
- **VALENCIA.DATA:** aims to start a digital ecosystem for management and use of relevant personal information and is being carried out by spanish leading-technology institutions
  Web:
  https://www.ibv.org/en/news/valenciadata-digital-ecosystem-empowers-citizens-in-the-use-of-their-data-and-allows-them-to-become-potential-entrepreneurs-en

6 Conclusion and Future Work

This design document provided an overview of the high-level architecture, layers, major components, descriptions, and services of the proposed Semantic Container environment which we envision to provide a technical infrastructure for vibrant data markets.

In particular, we specified the semantic descriptions associated with each semantic container, including metadata such as usage policies and provenance information, outlined the services for validation, blockchain-based verification, and billing available in the Semantic Container environment. Furthermore, we specified data structures and formats, explained the lifecycle of a semantic container. Together, these components provide an infrastructure for self-contained packaging in data exchange. Finally, the document introduced three real-world validation use cases on Seismic Data, Weather Data, and EODC Satellite data. It also provides pointers to the actual implementation of the three evaluation use cases.

There are still a number of areas not addressed in the course of this project that should be investigated further in future work in order for Semantic Containers to cover new use cases and apply the concept in additional areas:

- **Streaming Data:** currently, only static and dynamic Semantic Containers are supported and streaming data would require a new type that handles the relevant metadata for a specific time period
- **Hydra API Description:** Hydra is a lightweight vocabulary to create hypermedia-driven Web APIs and would fit nicely into the overall concept of semantic annotation (currently, APIs are documented in the Swagger notation)
- **Semantic Metadata:** data quality (e.g., freshness, completeness, accuracy) is a multi-dimensional concept which varies depending on the data usage scenario; using W3C's Data Quality Vocabulary[3] as a generic framework will allow to represent quality measurements for the contents of a container along various quality dimensions and according to various quality metrics
- **Registry:** a central registry would allow users to search for semantic containers that satisfy a certain information or processing need; it enables searching Semantic Containers based on usage policies, provenance, and semantics of the content as well as provided data processing capabilities
- **Billing:** currently, only singular payments via the cryptocurrency Ether are supported; it will be necessary to support SEPA based transactions in Euro and also cover various subscription models common in current data markets
- **Hosting:** for wider adoption of Semantic Containers a publicly available hosting platform needs to be established for running Semantic Containers in a scalable environment; this includes running containers as well as access mechanisms for Docker images

[3] The Data Quality Vocabulary (**DQV**) and the corresponding W3C working group note (https://www.w3.org/TR/vocab-dqv/) provide a framework for the description of a dataset's quality.

7 Final Remarks

In this design document, we described the architecture for a data exchange mechanism between multiple parties.

did:sov:TVRonD5tR4h3JDcEJruBY9

You can find the latest version of this document at https://www.ownyourdata.eu/semcon/design.

Additionally, a reference is stored in the Sovrin blockchain at the address shown on the left-hand side. There you can always find the complete history and further information about this document.

Use https://uniresolver.io/#did=did:sov:TVRonD5tR4h3JDcEJruBY9 to resolve the address.

Semantic Containers is funded in the program “IKT der Zukunft” by the Federal Ministry for Transport, Innovation and Technology (bmvit) under grant number 869781.

Please don’t hesitate to contact us with any comments and feedback via semcon@ownyourdata.eu.

Appendices

Appendix A - Example Base-Container Configuration: image-constraints.trig

The following is an example `image-constraints.trig` file.

```
@prefix : <http://w3id.org/semcon/ns/ontology#> .
@prefix scr: <http://w3id.org/semcon/resource/> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix w3c-format: <http://www.w3.org/ns/formats/> .
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

#################################################################
#       Version of the ontology (and the base container)
#################################################################

<http://w3id.org/semcon/ns/ontology> rdf:type owl:Ontology ;
        owl:versionIRI <http://w3id.org/semcon/ns/ontology/0.1.0> .

#################################################################
#       DEFAULT GRAPH: SHACL constraints for the overall structure of data /
#       aggregated data (to validate container description: “INIT.TRIG”)
#################################################################
scr:ContainerConfigurationConstraints
        a               sh:NodeShape ;
        sh:closed       true ;
        sh:description  "ContainerConfiguration validation" ;
        sh:name         "ContainerConfiguration validation" ;
        sh:property     [ sh:maxCount  1 ;
                          sh:minCount  1 ;
                          sh:path         rdf:type
                          ] ;
        sh:property     [ sh:maxCount  1 ;
                          sh:minCount  1 ;
                          sh:path         dc:title
                          ] ;
        sh:property     [ sh:minCount  1 ;
                          sh:path         dc:description
                          ] ;
        sh:property     [ sh:minCount  0 ;
                          sh:path         dc:creator
                          ] ;
        sh:property     [ sh:minCount  0 ;
                          sh:path         dc:contributor
                          ] ;
        sh:property     [ sh:dataType :DataConfiguration ;
                          sh:maxCount  1 ;
                          sh:minCount  0 ;
                          sh:path         :hasDataConfiguration
                          ] ;
        sh:targetClass  :ContainerConfiguration .


scr:DataConfigurationConstraints
        a               sh:NodeShape ;
        sh:closed       true ;
```

```
        sh:description  "DataConfiguration validation" ;
        sh:name         "DataConfiguration validation" ;
        sh:property     [ sh:maxCount  1 ;
                          sh:minCount  1 ;
                          sh:path      rdf:type
                        ] ;
        sh:property     [ sh:minCount  1 ;
                          sh:path      :hasTag
                        ] ;
        sh:property     [ sh:maxCount  1 ;
                          sh:minCount  0 ;
                          sh:path      :hasNativeSyntax
                        ] ;
        sh:property     [ sh:maxCount  1 ;
                          sh:minCount  0 ;
                          sh:path      :hasExampleData
                        ] ;
        sh:targetClass  :DataConfiguration .

#################################################################
#       NAMED GRAPH ==> :ImageConfiguration
#       To describe the basic configuration of the Semantic Container
#       Including the locations of the services
#################################################################
:ImageConfiguration {
        :ImageConfigurationInstance rdf:type :ImageConfiguration ;
            # usage policy service
            :usagePolicyValidationService
                "https://semantic.ownyourdata.eu/api/validate/usage-policy" ;
            # usage policy service
            :initValidationService
                "https://semantic.ownyourdata.eu/api/validate/init" ;
        .
}

#################################################################
#       NAMED GRAPH ==> :ImageModel
#       Contains all classes and properties used within Semantic Container Context
#################################################################
:ImageModel {
        ###################################################################
        #       Object Properties
        ###################################################################
        ###  http://purl.org/dc/elements/1.1/contributor
        dc:contributor rdf:type owl:ObjectProperty ;
                rdfs:range foaf:Agent .
        ###  http://purl.org/dc/elements/1.1/creator
        dc:creator rdf:type owl:ObjectProperty ;
                rdfs:range foaf:Agent .
        ###  http://w3id.org/semcon/ns/ontology#hasDataConfiguration
        :hasDataConfiguration rdf:type owl:ObjectProperty ;
                        rdfs:domain :ContainerConfiguration ;
                        rdfs:range :DataConfiguration .
        ###  http://w3id.org/semcon/ns/ontology#hasDataLayer
        :hasDataLayer rdf:type owl:ObjectProperty ;
                rdfs:domain :ContainerConfiguration ;
                rdfs:range :DataLayer .
        ###  http://w3id.org/semcon/ns/ontology#hasNativeSyntax
        :hasNativeSyntax rdf:type owl:ObjectProperty ;
                        rdfs:domain :DataConfiguration ;
                        rdfs:range :NativeSyntax .

        ###################################################################
        #       Data properties
        ###################################################################
```

```
###  http://purl.org/dc/elements/1.1/description
dc:description rdf:type owl:DatatypeProperty .
###  http://purl.org/dc/elements/1.1/title
dc:title rdf:type owl:DatatypeProperty .
###  http://w3id.org/semcon/ns/ontology#hasExampleData
:hasExampleData rdf:type owl:DatatypeProperty ;
        rdfs:domain :DataConfiguration .
###  http://w3id.org/semcon/ns/ontology#isDataConstraintExist
:isDataConstraintExist rdf:type owl:DatatypeProperty ;
                       rdfs:domain :DataConfiguration .
###  http://w3id.org/semcon/ns/ontology#isDataMappingExist
:isDataMappingExist rdf:type owl:DatatypeProperty ;
                rdfs:domain :DataConfiguration .
###  http://w3id.org/semcon/ns/ontology#isDataModelExist
:isDataModelExist rdf:type owl:DatatypeProperty ;
                rdfs:domain :DataConfiguration .
###  http://w3id.org/semcon/ns/ontology#hasTag
:hasTag rdf:type owl:DatatypeProperty ;
rdfs:domain :DataConfiguration .
###  http://xmlns.com/foaf/0.1/mbox
foaf:mbox rdf:type owl:DatatypeProperty ;
        rdfs:domain foaf:Agent .
###  http://xmlns.com/foaf/0.1/name
foaf:name rdf:type owl:DatatypeProperty ;
        rdfs:domain foaf:Agent .

#################################################################
#       Classes
#################################################################
###  http://w3id.org/semcon/ns/ontology#ContainerConfiguration
:ContainerConfiguration rdf:type owl:Class .
###  http://w3id.org/semcon/ns/ontology#DataConfiguration
:DataConfiguration rdf:type owl:Class .
###  http://w3id.org/semcon/ns/ontology#DataLayer
:DataLayer rdf:type owl:Class .
###  http://w3id.org/semcon/ns/ontology#NativeSyntax
:NativeSyntax rdf:type owl:Class .
###  http://w3id.org/semcon/ns/ontology#RDFSyntax
:RDFSyntax rdf:type owl:Class ;
        rdfs:subClassOf :NativeSyntax .
###  http://xmlns.com/foaf/0.1/Agent
foaf:Agent rdf:type owl:Class .
###  http://xmlns.com/foaf/0.1/Organization
foaf:Organization rdf:type owl:Class ;
                rdfs:subClassOf foaf:Agent .
###  http://xmlns.com/foaf/0.1/Person
foaf:Person rdf:type owl:Class ;
        rdfs:subClassOf foaf:Agent .

#################################################################
#       Individuals
#################################################################
###  https://ownyourdata.eu/semcon/ontology
###  http://w3id.org/semcon/ns/ontology#JSON
:JSON rdf:type owl:NamedIndividual ,
        :NativeSyntax .
###  http://w3id.org/semcon/ns/ontology#CSV
:CSV rdf:type owl:NamedIndividual ,
        :NativeSyntax .
###  http://w3id.org/semcon/ns/ontology#LayerMetadata
:LayerMetadata rdf:type owl:NamedIndividual ,
                       :DataLayer ;
        rdfs:label "Layer: Metadata" .
###  http://w3id.org/semcon/ns/ontology#LayerSemantics
```

```
        :LayerSemantics rdf:type owl:NamedIndividual ,
                                :DataLayer ;
                        rdfs:label "Layer: Semantics" .
        ###  http://w3id.org/semcon/ns/ontology#LayerSyntax
        :LayerSyntax rdf:type owl:NamedIndividual ,
                        :DataLayer ;
                rdfs:label "Layer: Syntax" .
        ###  http://www.w3.org/ns/formats/JSON-LD
        w3c-format:JSON-LD rdf:type owl:NamedIndividual ,
                                :RDFSyntax .
        ###  http://www.w3.org/ns/formats/TriG
        w3c-format:TriG rdf:type owl:NamedIndividual ,
                                :RDFSyntax .
        ###  http://www.w3.org/ns/formats/Turtle
        w3c-format:Turtle rdf:type owl:NamedIndividual ,
                                :RDFSyntax .
        ###  Generated by the OWL API (version 4.5.6.2018-09-06T00:27:41Z)
        ###      https://github.com/owlcs/owlapi
}
```

Listing A.1: The initial image-container configuration, i.e., the "`image-container.trig`" file.

Appendix B - Example Initial Container Configuration: init.trig

The following is an example `init.trig` file.

```
@prefix : <http://w3id.org/semcon/ns/ontology#> .
@prefix scr: <http://w3id.org/semcon/resource/> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix w3c-format: <http://www.w3.org/ns/formats/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix step: <http://w3id.org/semcon/ns/stepcount#> .
@prefix svpr:  <http://www.specialprivacy.eu/vocabs/processing#> .
@prefix svpu:  <http://www.specialprivacy.eu/vocabs/purposes#> .
@prefix svd:   <http://www.specialprivacy.eu/vocabs/data#> .
@prefix svdu:  <http://www.specialprivacy.eu/vocabs/duration#> .
@prefix svl:   <https://www.specialprivacy.eu/vocabs/locations#> .
@prefix svr:   <https://www.specialprivacy.eu/vocabs/recipients#> .
@prefix spl:   <https://www.specialprivacy.eu/langs/usage-policy#> .
@prefix sh: <http://www.w3.org/ns/shacl#> .

###################################################################
#    NAMED GRAPH ==> :BaseConfigurations
###################################################################
:BaseConfiguration {

  :ContainerConfigurationInstance rdf:type :ContainerConfiguration ;
     # generic - dublin core
     dc:title "Step Count Container" ;
     dc:description """
           This container contains information about step count
        """ ;
     dc:creator [
        rdf:type foaf:Person ;
        foaf:name "Christoph Fabianek" ;
        foaf:mbox <mailto:christoph.fabianek@gmail.com> ;
     ] ;
     dc:contributor [
        rdf:type foaf:Organization ;
        foaf:name "OwnYourData" ;
        foaf:mbox <mailto:christoph.fabianek@gmail.com> ;
     ] , [
        rdf:type foaf:Person ;
        foaf:name "Fajar Ekaputra" ;
        foaf:mbox <mailto:fajar@juang.id> ;
     ] ;
     :hasDataConfiguration :StepCountData .
     # :hasDataLayer scr:LayerSemantic ;       # NON-EDITABLE - assigned

  :DataConfigurationInstance rdf:type :DataConfiguration ;
     # :isDataModelExist true ;          # NON-EDITABLE - assigned
     # :isDataConstraintExist true ;          # NON-EDITABLE - assigned
     # :isDataMappingExist false ;        # NON-EDITABLE - assigned
     # :isUsagePolicyExist true ;         # NON-EDITABLE - assigned
     # data - classifications
     :hasTag "Step Count", "Health", "Gyroscope" ;
     # data format - inspired from void
     :hasNativeSyntax w3c-format:Turtle ;
     # example data in plain string
     :hasExampleData """
        @prefix step: <http://w3id.org/semcon/ns/stepcount#> .
        @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
```

```
        [] rdf:type step:StepCount ;
           step:date "2018-06-17" ;
           step:value "2345" .

        [] rdf:type step:StepCount ;
           step:date "2018-06-18" ;
           step:value "7502" .
        """    .
}

#################################################################
#    NAMED GRAPH ==> :UsagePolicies
#################################################################
:UsagePolicy {
  :ContainerPolicy rdf:type owl:Class ; # this line should not be changed!
     owl:equivalentClass [
        owl:intersectionOf (
           [
              rdf:type owl:Restriction ;
              owl:onProperty spl:hasData ;
              owl:someValuesFrom svd:Financial
           ]
           [
              rdf:type owl:Restriction ;
              owl:onProperty spl:hasProcessing ;
              owl:someValuesFrom spl:AnyProcessing
           ]
           [
              rdf:type owl:Restriction ;
              owl:onProperty spl:hasPurpose ;
              owl:someValuesFrom spl:AnyPurpose
           ]
           [
              rdf:type owl:Restriction ;
              owl:onProperty spl:hasRecipient ;
              owl:someValuesFrom spl:AnyRecipient
           ]
           [
              rdf:type owl:Restriction ;
              owl:onProperty spl:hasStorage ;
              owl:someValuesFrom spl:AnyStorage
           ]
        ) ;
        rdf:type owl:Class
     ] .
}

#################################################################
#    NAMED GRAPH ==> :DataModels
#################################################################
:DataModel {
  # I have a single class, called StepCount
  step:StepCount a owl:Class .

  # each step count has a maximum of one date property of type xsd:date
  step:date a owl:DatatypeProperty, owl:FunctionalProperty ;
     rdfs:domain step:StepCount ;
     rdfs:range xsd:date .

  # each step count has a maximum of one value property of type xsd:integer
  step:value a owl:DatatypeProperty, owl:FunctionalProperty ;
     rdfs:domain step:StepCount ;
     rdfs:range xsd:integer .
}
```

```
#####################################################################
#    NAMED GRAPH ==> :DataConstraints
#####################################################################
:DataConstraint {
  scr:StepCountShape a sh:NodeShape ;
     sh:name "StepCount data constraints" ;
     sh:targetClass: step:StepCount ;
     sh:description "Example data constraints for StepCount data" ;
     sh:closed true ;
     sh:property [
        sh:path step:date ;
        sh:dataType xsd:dateTime ;
        sh:maxCount 1 ;
        sh:minCount 1 ;
     ] ;
     sh:property [
        sh:path step:value ;
        sh:dataType xsd:integer ;
        sh:maxCount 1 ;
        sh:minCount 1 ;
        sh:minValue 0 ;
     ] ;
     sh:property [
        sh:path rdf:type ;
        sh:maxCount 1 ;
        sh:minCount 1 ;
     ] .
}

#####################################################################
#    NAMED GRAPH ==> :DataMappings (Future)
#####################################################################
:DataMapping {

}

#####################################################################
#    NAMED GRAPH ==> :UserFunctions - Hydra function definitions (Future)
#####################################################################
:UserFunction {

}
```

Listing B.1: The initial container configuration for step-count data, i.e., the “`init.trig`” file.

Appendix C - Usage Policies

The following lists define the available attributes for each element of a Usage Policy as defined in the SPECIAL Policy Language V1[4].

Data Categories:

- Activity
- Anonymized
- AudiovisualActivity
- Computer
- Content
- Demographic
- Derived
- Financial
- Government
- Health
- Interactive
- Judicial
- Location
- Navigation
- Online
- OnlineActivity
- Physical
- PhysicalActivity
- Political
- Preference
- Profile
- Purchase
- Social
- State
- Statistical
- TelecomActivity
- UniqueId

Purposes:

- Account
- Admin
- AnyContact
- Arts
- AuxPurpose
- Browsing
- Charity
- Communicate
- Current
- Custom
- Delivery

[4] https://www.specialprivacy.eu/images/documents/SPECIAL_D2.1_M12_V1.0.pdf

- Develop
- Downloads
- Education
- Feedback
- Finmgt
- Gambling
- Gaming
- Government
- Health
- Historical
- Login
- Marketing
- News
- OtherContact
- Payment
- Sales
- Search
- State
- Tailoring
- Telemarketing

Processing Methods:

- Aggregate
- Analyze
- Anonymize
- Collect
- Copy
- Derive
- Move
- Query
- Transfer

Recipients:

- Delivery - delivery services
- Other Recipient - others with a different policy
- Ours - ourselves
- Public - public fora
- Same - others with the same policy
- Unrelated - unrelated third parties

Locations:

- Controller Servers - on the servers of the data controller
- EU - on servers of countries in the EU
- EULike - on servers of countries with EU-like legislation
- ThirdCountries - on servers outside the EU
- OurServers - only on our own servers
- ProcessorServers - on servers of the data processor

- ThirdParty - on the servers of third party service providers

Durations:

- BusinessPractices
- Indefinitely
- LegalRequirement
- StatedPurpose