DataVera

EKG Platform Implementation Guide

Platform setup and performance tuning.
Setting up data processing rules

Summary

1. Deploy the platform

1.1. Architecture overview

DataVera EKG Platform is a sophisticated enterprise software solution optimized for high-load environments. Its architecture for on-premise installation includes several components. The principal deployment schema is shown in the Fig. 1. The components distribution by the virtual machines and the minimum hardware requirements are listed in the Table 1.

- Kubernetes/OpenShift cluster nodes are used to deploy the DataVera application components:

  – DataVera EKG Provider – the data access middleware, a core system component.

  – DataVera EKG Explorer – the web-based user interface.

  – Incoming (ekg-worker) and outgoing data exchange adapters.

- Apache Fuseki – the RDF triple store used by the platform to keep the data model.
- PostgreSQL – the database cluster used by the platform to store the data.
- etcd+haproxy – the configuration storage and load balancer for PostgreSQL cluster.
- Keycloak – the authorization and authentication provider.
- Kafka – the message broker used for data exchange between the platform components and with the external data suppliers or consumers.
- Elasticsearch and Kibana – logs storage and components monitoring.

Looks complex, right? But we are sure that the proposed architecture is highly adaptive, scalable and functional for the real-world data processing tasks.

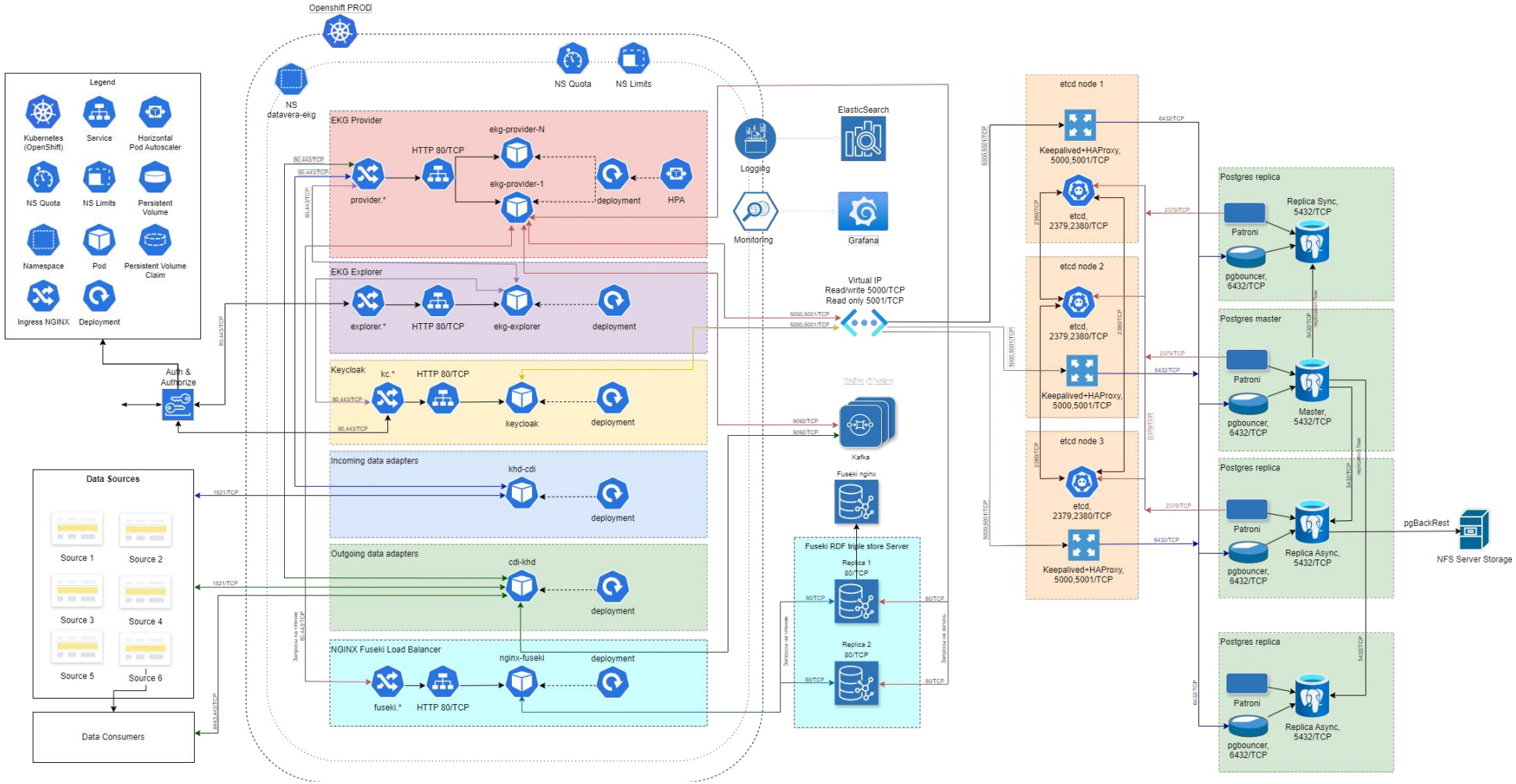


Fig. 1. DataVera EKG Platform deployment diagram

In this guide we describe how to set up platform environment step by step, referring to separate technical instructions for some steps to avoid excessive details.

Table 1. **DataVera EKG Platform on-premise deployment: the list of the components and virtual machines, with minimal hardware requirements.**

#	*Server role*	*vCPUs*	*RAM, Gb*	*SSD, Gb*	*Quantity*
1	Kubernetes/OpenShift cluster, master node	4	4	50	1
2	Kubernetes/OpenShift cluster, worker node	8	16	100	2
3	GitLab server	4	8	100	1
4	Elasticsearch/Kibana	4	8	100	1
5	Apache Fuseki RDF triple store (replica 1)	4	4	50	1
6	Apache Fuseki RDF triple store (replica 2)	4	4	50	1
7	PostgreSQL (master)	8	32	500	1
8	Apache Fuseki RDF triple store (replica 1)	4	4	50	1

#	*Server role*	*vCPUs*	*RAM, Gb*	*SSD, Gb*	*Amount*
9	Apache Fuseki RDF triple store (replica 2)	4	4	50	1
10	PostgreSQL (master)	8	32	500	1
11	PostgreSQL (read only replica)	8	32	500	1
12	PostgreSQL (read only replica)	8	32	500	1
13	PostgreSQL (standby)	8	32	500	1
14	PostgreSQL (replica)	8	32	500	1
15	etcd+haproxy (node 1)	2	4	20	1
16	etcd+haproxy (node 2)	2	4	20	1
17	etcd+haproxy (node 3)	2	4	20	1
Total		**66**	**184**	**2510**	**14**

To install the software, you need to prepare all the above-listed servers and allocate resources in the Kubernetes/OpenShift cluster. To improve platform performance, we recommend using drives and network storage with minimal access rates (Low Latency) and maximum I/O performance (High IOPS), based on NVMe or SSD. After installing the operating system, it is recommended to install all the released updates and security patches, following the OS vendor's instructions. On all servers, you need to install and configure time synchronization software via the NTP protocol in accordance with the OS vendor's instructions.

Then install GitLab and import the projects provided by DataVera in the installation package (see the GitLab set-up instruction), or use the HELM charts. These projects contain configuration files and scenarios used in the further installation steps.

1.2. Install PostgreSQL and set up databases

We recommend clustered Postgres deployment using Partoni. However, for the demonstration purposes or in the low load projects the standalone Postgres can be used.

Deploy the PostgreSQL cluster using the Ansible playbook provided by DataVera, following the detailed instructions. As a result of this step, you should obtain a working PostgreSQL cluster (see *Fig. 2*).

A multi-threaded connection pooler pgcat is used to distribute SQL queries across cluster replicas and split read and write queries. The pooler runs as a container in a Kubernetes/OpenShift cluster. See the separate instruction on how to deploy pgcat using GitLab CI/CD pipeline.

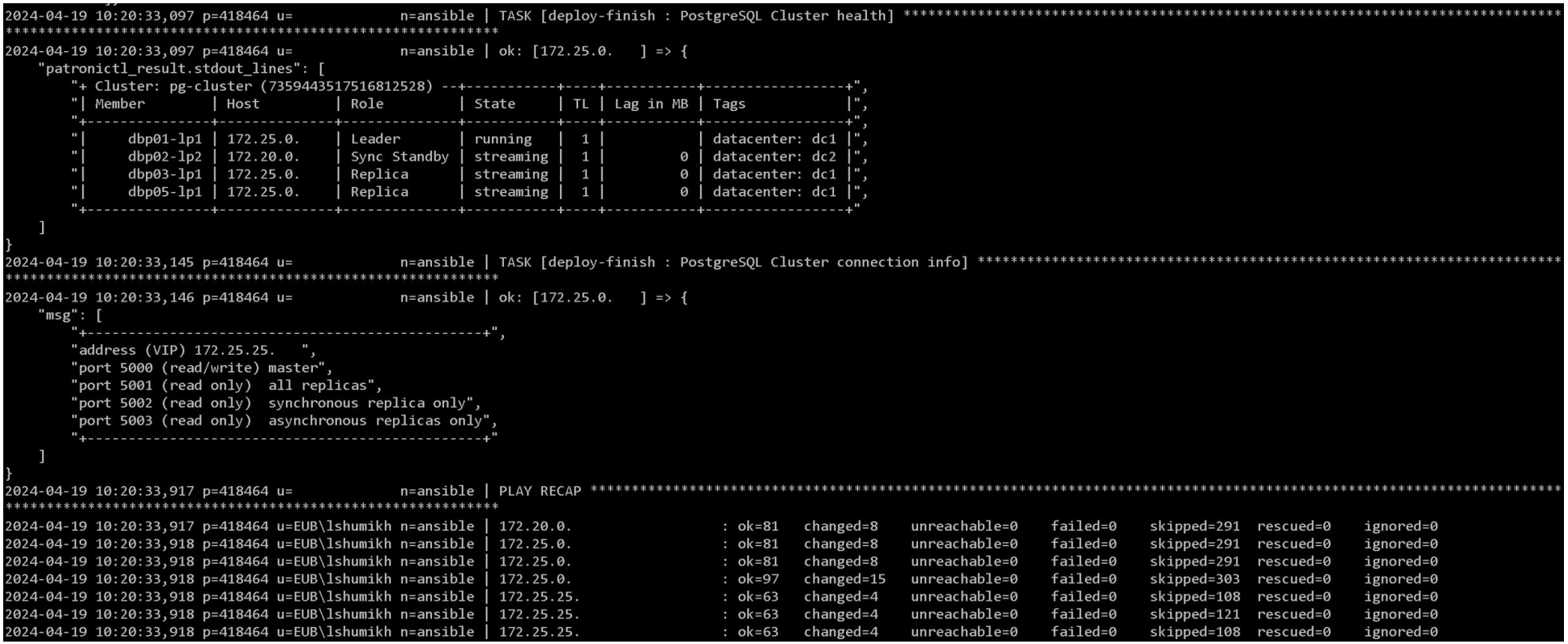


Fig. 2. Ansible playbook output demonstrating the successful PostgreSQL cluster deployment

After that, you need to import the initial database dumps provided in the dumps folder of the datavera-ekg/postgresql_cluster GitLab project.

This can be done from the console of any VM with the postgresql client installed and the network access to the PostgreSQL cluster, for example from any cluster node.

Copy the dump files ekg_config_prod.psql.gz and ekg_data_prod.psql.gz, unzip them and load into the database using the following shell commands:

```
psql -h [master IP] -p 5000 -U ekg -d ekg_config < ekg_config_prod.psql
psql -h [master IP] -U ekg -d ekg_data_prod < ekg_data_prod.psql
```

You can use any PostgreSQL client application to connect to these databases, for example DBeaver.

1.3. Install Apache Fuseki and initialize dataset

Apache Jena Fuseki is an open-source RDF triple store which DataVera EKG Platform uses as a data model storage. It is deployed using datavera-ekg/update-artefacts GitLab project. The image update CI/CD pipeline includes update-image job that fetches images from the vendor repository and uploads them to the customer's repository. You should run this job before starting deployment.

The Fuseki server is deployed as a Docker container on two hosts that duplicate each other. To distribute requests between the hosts, a NGINX traffic balancer shall be deployed in Kubernetes/OpenShift. It is an entry point for applications reading from Fuseki. Fuseki write operations are performed by the platform on both servers directly and simultaneously. Please follow the Fuseki deployment instruction to deploy it on the appropriate virtual servers.

After the deployment is done, you shall load the initial dataset from the file "dataset.nq.gz" in the datavera-ekg/fuseki GitLab project into both Fuseki instances, using the "Add data" button in its user interface.

1.4. Set up Keycloak

Keycloak is an open-source authorization and authentication provider used by DataVera EKG Platform. It can be federated with Active Directory domains or used as a Kerberos authentication provider.

The YAML manifests and configuration files for Keycloak are located in the datavera-ekg/keycloak GitLab project. Keycloak runs in Kubernetes. It stores configuration in PostgreSQL. Please refer to the Keycloak deployment instruction to go through the setup steps.

When Keycloak is up, log into its interface. Open the "master" realm management page. Open the datavera-ekg/keycloak project in GitLab and save the eub-prod/realm-export.json file to the workstation where the Keycloak web interface is available.

Go to the http://[keycloak address]/auth/admin/master/console/#/realms/master/partial-import page, choose the "If a resource exists = Skip" option, and import the saved file.

Keycloak deployment is complete. Now you can create user groups and accounts.

1.5. Deploy DataVera applications as Kubernetes pods

The datavera-ekg/ekg-provider GitLab project contains the core YAML manifests and configuration files for application deployment. The EKG Provider stores its configuration in the ekg_config PostgreSQL database. The access credentials for this database, as well as some other default settings, are defined in the config.json file, which you can edit in GitLab:

```
{"postgres":"user=ekg password=<db_password> dbname=ekg_config host=pgcat-primary port=6432
target_session_attrs=read-write","ekg_host":"127.0.0.1","ekg_port":"80","ekg_url":"/
rest/","ekg_client":"ekg","internal_broker":"Postgres","broker_queue":"EKG_SYNC","broker_host
":"<broker
host>","broker_port":"9092","broker_user":"ekg","broker_password":"<broker_password>"}
```

The description of the parameters and their default values are listed on the next page.

Table 2. **DataVera EKG Provider core configuration parameters**

Parameter	*Default value*	*Descriprion*
postgres		PostgreSQL connection string
ekg-host	127.0.0.1	EKG Provider host
ekg-port	80	EKG Provider port
ekg_url	/rest/	EKG Provider URL
ekg_client	ekg	EKG Provider client ID
internal_broker	Postgres	Internal broker used by EKG Provider, may be either PostgreSQL or Kafka
broker_queue	EKG_SYNC	Internal broker topic name
broker_host		External broker used by EKG Provider to exchange data with the data sources or consumers, host
broker_port		External broker port
broker_user		External broker port login
broker_password		External broker port password

When the config.json is ready, you have to run the "update-image" and "deploy" tasks in the GitLab project, as shown in *Fig. 3*.

Run these jobs for all the DataVera applications: ekg-provider, ekg-explorer and ekg-worker. If the pods start successfully, the deployment is done. Now you have to configure the data storages and access rights.

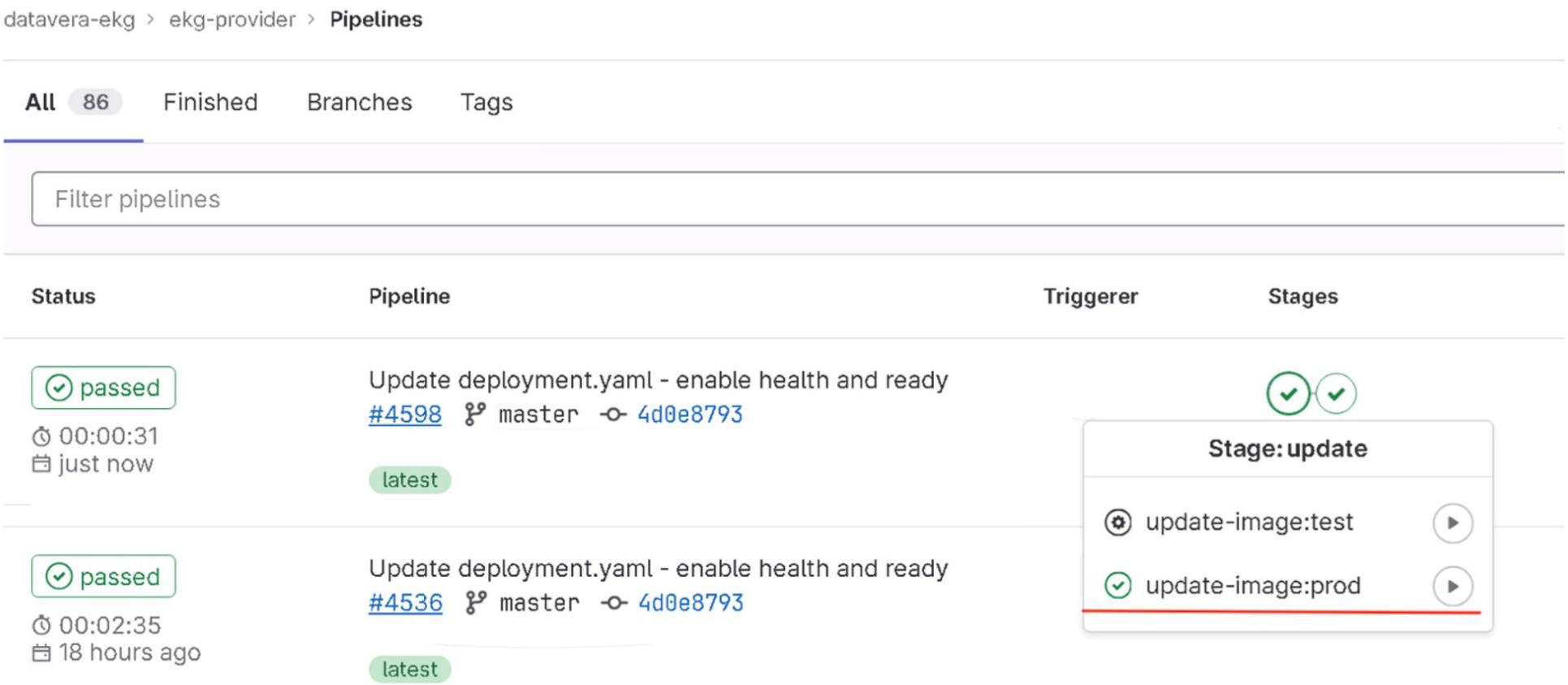


Fig. 3. CI/CD tasks for the ekg-provider application deployment

1.6. Set up endpoint, data storages and access rights

The remaining platform configuration steps are performed using the "ekg-provider" shell command. You shall log into Kubernetes/OpenShift cluster, and get the ekg-provider pod identifier by running the “kubectl get pods” command. Then you can run configuration commands prepending them with "kubectl exec" (or "oc exec", if using OpenShift). For example, run the following command to list the available endpoints:

```
kubectl exec [pod id] -c main ./ekg-provider show endpoints
```

We will further omit the left part of the commands.

Our first task is to create an endpoint. Endpoint can be viewed as a separate dataset, or a logical database. Each endpoint has its own data model, rules and access rights.

To create a new endpoint or change the properties of an existing one, run the following command:

```
ekg-provider add | change endpoint [readable name] [API code] [ontology prefix] [default
storage]
```

Each endpoint shall have at least one assigned storage – a database which will store its data. The first storage of each endpoint must be a Fuseki dataset: it will be used to keep store the model and rules. The other storages may be PostgreSQL tables intended to keep store the data objects (individuals) of the model classes.
The storages management command has the following syntax:

```
ekg-provider add | change storage [name] [endpoint name or code] fuseki | postgresql [host] [port] [dbname or _ for Fuseki] [table or dataset for Fuseki] [login] [password] [history (0/1)] [logic (0/1)] [query_path for Fuseki] [update_path for Fuseki]
```

Since an endpoint contains a reference to the default storage, and each storage is bound to an endpoint, you shall perform the following steps:

1. Create an endpoint without specifying a value for the default storage parameter:

   ```
   ekg-provider add endpoint Demo demo http://yourdomain.com/prefix/
   ```

2. Create a Fuseki storage for this endpoint. We have initialized a Fuseki dataset previously, now we have to register it in the platform configuration database:

```
ekg-provider add storage Fuseki_Demo
demo fuseki 127.0.0.1 8080 _ demo
admin [Fuseki password] 1 1 /fuseki4/
demo /fuseki4/demo/update
```

3. Run the change endpoint command, specifying the name of the created storage as the default storage

```
ekg-provider change endpoint Demo demo
http://yourdomain.com/prefix/
Fuseki_Demo
```

After running these commands, you can restart the ekg-provider application using GitLab CI/CD "deploy" task, log into the EKG Explorer web interface and see your first working endpoint. But as this endpoint will have only one Fuseki storage, it will save all the data objects in it, which is rather slow. Therefore, it would be better to create a PostgreSQL storage right from the start, and map our model classes to it. At this moment, we need to come up with our ontology structure (we can consider it as a data model, in some approximation). Let us decide now that all the classes we create shall be the subclasses of an http://www.w3.org/ns/prov#Entity (prov:Entity) class. This class generally describes all the entities that may have associated provenance information, and all our domain ontology objects shall have it. Now we proceed without discussion, and return to it later. Let us create a PostgreSQL storage and map our class to it.

1. Connect to our PostgreSQL server. Create the database "postgres_demo". Create the first table having the following structure:

```
CREATE TABLE public.entity (
    uri varchar NULL,
   "type" varchar NULL,
   "name" varchar NULL,
   "data" jsonb NULL,
   "sameas" varchar NULL,
   CONSTRAINT entity_un UNIQUE (uri)
);

CREATE UNIQUE INDEX entity_idx ON
public.entity USING btree (uri);

CREATE INDEX data_idx ON public.entity
USING gin (data);

CREATE INDEX sameas_idx ON
public.entity USING gin (sameas);
```

2. Create a PostgreSQL storage in the EKG Provider configuration:

```
ekg-provider add storage PG_Entity
demo postgresql 127.0.0.1 5432
postgres_demo entity ekg [Postgres ekg
password] 1 1
```

3. Assign the http://www.w3.org/ns/prov#Entity class (it will be the root of our ontology) to this storage using the command:

```
ekg-provider bind demo PG_Entity
http://www.w3.org/ns/prov#Entity
```

4. Set up standard columns mapping for this storage:

```
ekg-provider map PG_Entity uri uri 0

ekg-provider map PG_Entity data data 0

ekg-provider map PG_Entity name
http://www.w3.org/2000/01/rdf-
schema#label 0

ekg-provider map PG_Entity type
http://www.w3.org/1999/02/22-rdf-
syntax-ns#type 1

ekg-provider map PG_Entity sameas
http://www.w3.org/2002/07/owl#sameAs 1
```

These commands define the correspondence between the individuals of the http://www.w3.org/ns/prov#Entity class and the table columns. You can create additional columns, index them and map them to the ontology properties to improve SELECT queries performance. The properties that are rarely used in the queries should not be mapped to the separate columns, and by default their values will be stored in the jsonb column named "data". The four columns: uri, type, name and data are mandatory for all PostgreSQL storage tables, except of some very special cases. You can see the detailed explanation of the mentioned commands syntax in our User Guide.

There are three special classes in the platform metamodel: the “Consolidation errors” class stores the information about consolidation errors occurred during the reference data assembly, the “Validation result” class keeps the constraint violations, and the “Derivation” class stores data provenance information for the reference objects. All these classes shall be mapped to PostgreSQL tables to use the full platform functionality, especially for bulk validation and consolidation operations.

Below, we list the commands for setting up the standard mapping for these classes.

```
ekg-provider add storage PG_Validation demo postgresql 127.0.0.1 5432 postgres_demo \
validation ekg [Postgres ekg password] 1 1

ekg-provider bind demo PG_Validation http://www.w3.org/ns/shacl#ValidationResult

ekg-provider map PG_Validation uri uri 0

ekg-provider map PG_Validation data data 0

ekg-provider map PG_Validation name http://www.w3.org/2000/01/rdf-schema#label 0

ekg-provider map PG_Validation type http://www.w3.org/1999/02/22-rdf-syntax-ns#type 1

ekg-provider map PG_Validation focus_node \

http://www.w3.org/ns/shacl#focusNode

ekg-provider add storage PG_Consolidation demo postgresql 127.0.0.1 5432 postgres_demo \

consolidation ekg [Postgres ekg password] 1 1

ekg-provider bind demo PG_Consolidation http://datavera.kz/ekg/ConsolidationError

ekg-provider map PG_Consolidation uri uri 0
```

```
ekg-provider map PG_Consolidation data data 0

ekg-provider map PG_Consolidation name http://www.w3.org/2000/01/rdf-schema#label 0

ekg-provider map PG_Consolidation type http://www.w3.org/1999/02/22-rdf-syntax-ns#type 1

ekg-provider map PG_Consolidation data_source_object \

http://datavera.kz/ekg/hasDataSourceObject 1

ekg-provider add storage PG_Provenance demo postgresql 127.0.0.1 5432 postgres_demo \

provenance ekg [Postgres ekg password] 1 1

ekg-provider bind demo PG_Provenance http://www.w3.org/ns/prov#Derivation

ekg-provider map PG_Provenance uri uri 0

ekg-provider map PG_Provenance data data 0

ekg-provider map PG_Provenance name http://www.w3.org/2000/01/rdf-schema#label 0

ekg-provider map PG_Provenance type http://www.w3.org/1999/02/22-rdf-syntax-ns#type 1
```

You shall also create PostgreSQL tables to store the individuals of these classes:

```
CREATE TABLE public.violation (
   uri varchar NULL,
   "type" varchar NULL,
   "name" varchar NULL,
   "data" jsonb NULL,
   "focus_node" varchar NULL,
   CONSTRAINT violation_un UNIQUE (uri)
);

CREATE UNIQUE INDEX violation_uri_idx ON public.violation USING btree (uri);

CREATE INDEX violation_data_idx ON public.violation USING gin (data);

CREATE INDEX violation_focus_node_idx ON public.violation USING btree (focus_node);

CREATE TABLE public.consolidation (
   uri varchar NULL,
   "type" varchar NULL,
   "name" varchar NULL,
   "data" jsonb NULL,
   "data_source_object" _varchar NULL,
```

```
    CONSTRAINT consolidation_un UNIQUE (uri)
);

CREATE UNIQUE INDEX consolidation_uri_idx ON public.consolidation USING btree (uri);

CREATE INDEX consolidation_data_idx ON public.consolidation USING gin (data);

CREATE INDEX consolidation_data_source_object_idx ON public.consolidation USING gin
(data_source_object);

CREATE TABLE public.provenance (
    uri varchar NULL,
    "type" varchar NULL,
    "name" varchar NULL,
    "data" jsonb NULL,
    CONSTRAINT prov_un UNIQUE (uri)
);

CREATE UNIQUE INDEX provenance_uri_idx ON public.provenance USING btree (uri);

CREATE INDEX prov_provenance_idx ON public.provenance USING gin (data);
```

The only remaining task is to set up EKG Platform clients, map them to the Keycloak user groups, and define access rights.

1. First, let us create a platform client named "Managers" with the code "managers" for endpoint Demo:

```
ekg-provider add client Managers
managers Demo
```

2. Grant managers read and write rights to the http://www.w3.org/ns/prov#Entity class:

```
ekg-provider grant Demo Managers
http://www.w3.org/ns/prov#Entity 1 1
```

   Please note that it means that Managers will not have access to the other model classes unless we will explicitly grant it. We have to grant them read-only access to the SHACL classes container to make them able to interact with the platform's Data Quality, Mapping rules and Data provenance features (we will discuss it later in details):

```
ekg-provider grant Demo Managers
http://datavera.kz/ekg/shaclContainer 1 1

ekg-provider grant Demo Managers
http://datavera.kz/ekg/
MappingRulesElements 1 1

ekg-provider grant Demo Managers
http://datavera.kz/ekg/provRoot 1
```

3. Now open the Keycloak realm management console and create the Managers group. In order to map it to the "Managers" platform client, you need to create an attribute named "providerId" having the following JSON value (see *Fig. 4*):

```
{"clientId": "managers", "priority": 0
```

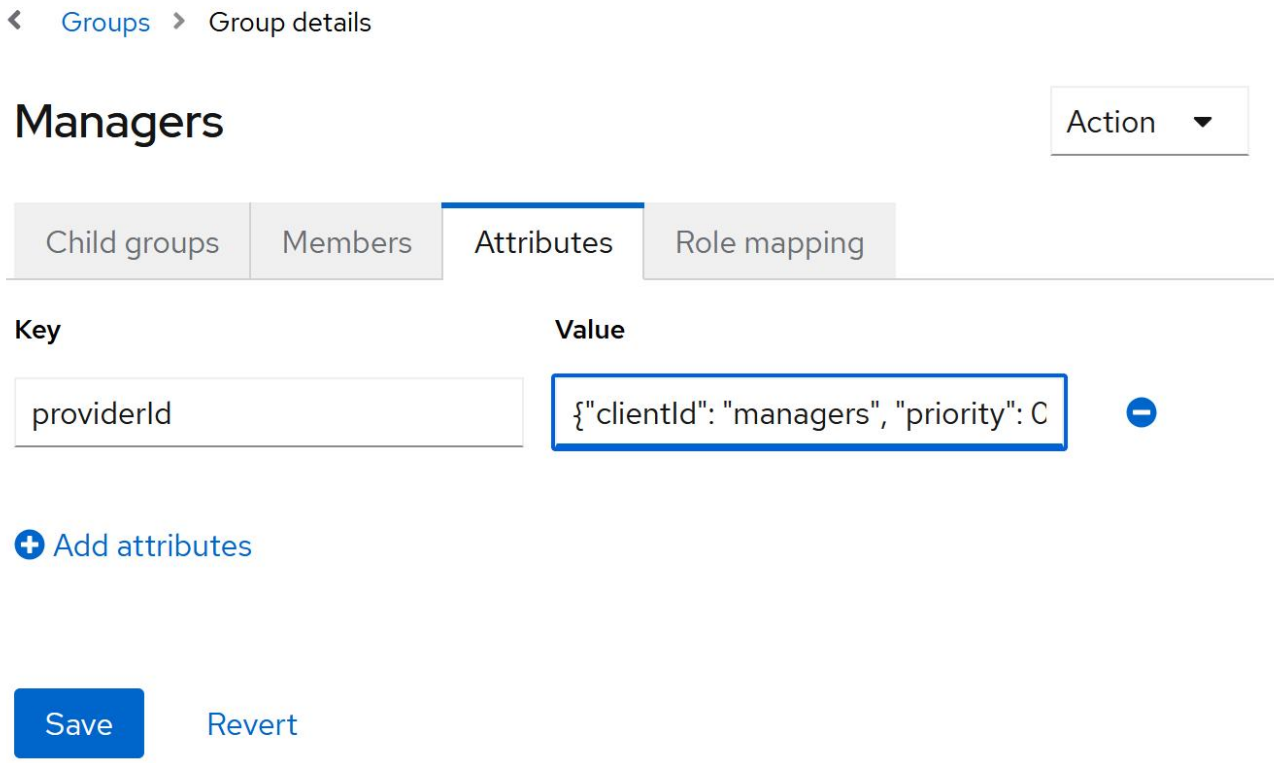


Fig. 4. Creating a Keycloak group attribute to map it with the EKG Provider client

Congratulations, we have completed the initial EKG Platform setup! Now you can restart the ekg-provider pod and log into the EKG Explorer web interface using an account from the "Managers" group. You shall see the tree of the default model classes to which you have granted access. It's time for us to start designing the data model.

2. Compose the Data Model

2.1. Classes

EKG Platform uses the RDFS/OWL ontologies to represent the data model. Don't worry; you won't need to dig into the nuances of logic and philosophical concepts to work with it. If you want to know a little more about the technical details of ontology modeling – for example, why URIs are used as the object identifiers – refer to the books covering this topic. Some information to start is also provided in our EKG Platform User Guide.

Let us also note that EKG Platform does not fully implement all the features of the RDFS/OWL specifications and does not pretend to be the ontology management software. It is a data management platform focused on processing the large data sets – the task which many of the ontology management platforms struggle with. It uses ontologies to simplify the data model and data processing rules management, and the main customer's benefit is the high speed of new data processing requirements implementation.

A rather simple open-source tool, Protégé, will help us create a sample model and load it into the platform. Download and run its desktop version. Change the ontology prefix to yours: we use http://yourdomain.com/prefix/ as an example, as mentioned above when creating endpoint. You can define another prefix, for example, using URL of your organization's website.

In the same Protégé window, open the Ontology Prefixes tab below. Edit the default prefix ":" and change it to http://yourdomain.com/prefix/

Switch to the Entities tab. You will see the Classes sub-tab with the only superclass, owl:Thing. It is a root of an ontology. The classes in ontologies define the entity (or data object) types, such as the Customers, Products etc. You may compare them with the tables in a relational database, with one significant difference: classes may contain subclasses.

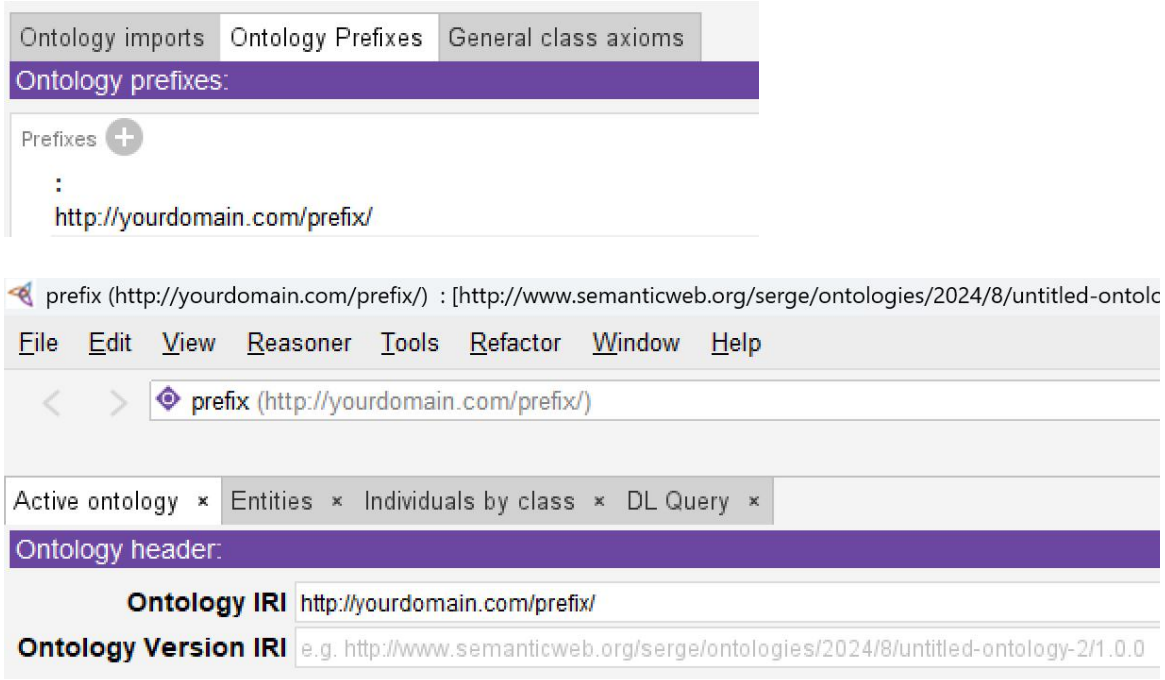


Fig. 5. Ontology prefix in Protégé

A subclass is a subset of some class. For example, our company works with the counterparties (Agents), which can be Persons or Organizations. The latter two are the subclasses of the former. This means that in our model, every Person is an Agent, and every Organization is an Agent, but not vice versa: not every Agent is a Person or Organization. Let us create the superclass http://www.w3.org/ns/prov#Entity, its subclass Agent, and its subclasses Person and Organization. The result should look as shown in *Fig. 6*.

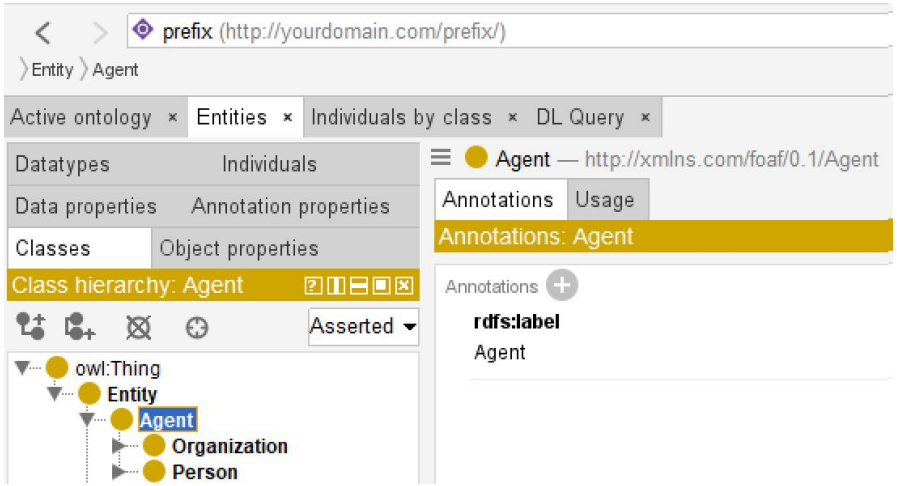


Fig. 6. Sample classes tree

We need to give readable names to the elements of our ontology. Select a class in the tree, click "+" button in the Annotations tab on the right, and add the rdfs:label value. You can define several labels for each ontology element in multiple languages.

At this moment let us mention that there exist a lot of widespread and well-known ontologies. To do the serious ontology engineering, you shall learn the upper-level and domain-level ontologies applicable to your business area. In our example, let us take some classes from the well-known FOAF (friend-of-a-friend) ontology. Let us refactor our model by importing the Agent branch from the FOAF ontology.

Another data objects classification basis is technological. DataVera EKG Platform works in the following way: it transfers data from the data sources into its storage, applies all kinds of rules (we will set them up later), and forms the reference objects (etalon). So we have to distinguish the objects acquired from data sources and the reference objects generated by the EKG. Let's create "Data source object" and "Reference object" subclasses for each of the lower-level classes. Let us also add the "Broker" subclass.

We need also the Product class to represent our company's products, and the Reference data class to store enumerations like Gender and Countries. After creating them, the classes taxonomy tree shall looks as shown in *Fig. 7*.

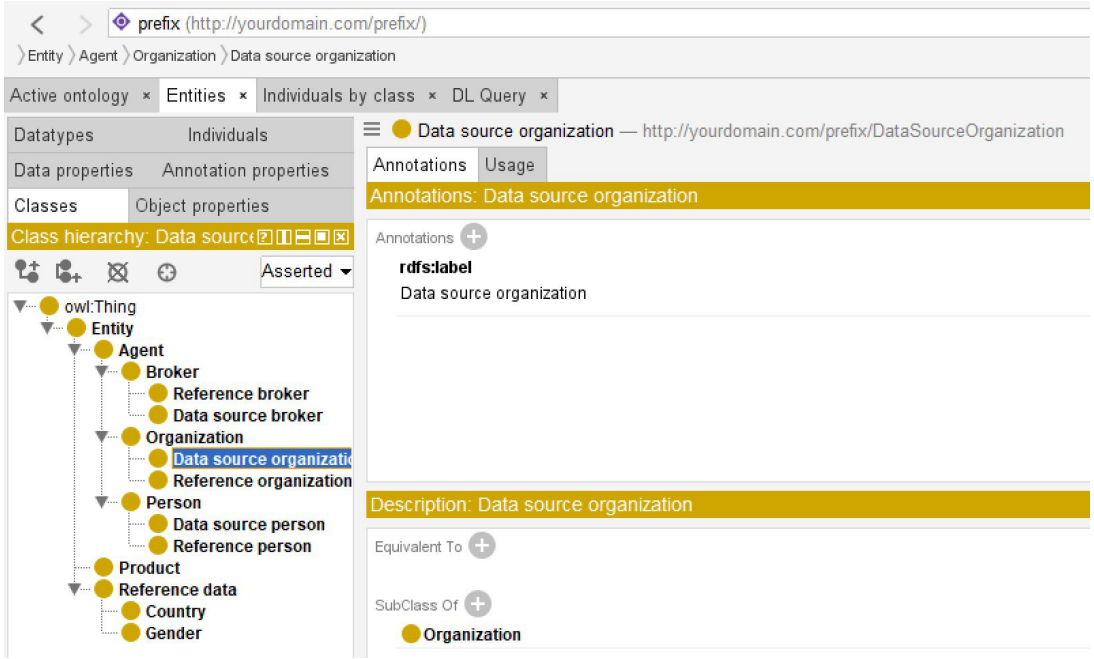


Fig. 7. Sample ontology classes tree

2.2. Properties

Now let's turn to the properties. Unlike relational databases, we can create properties applicable to several classes at once. Each property has a domain, which can be a combination of classes whose individuals can possess this property values. Each property also has a range, which defines the kind of values this property can have. Generally, there are two main kinds of properties: Data properties have literal values (strings, numbers, dates, etc.), while Object properties point to the objects of the other classes. Let us create several properties according to the *Table 3*.

Table 3. **The properties of the sample model**

Property	*Kind*	*Domain*	*Range*
Tax ID	Data property	Person or Broker	xsd:string
VAT number	Data property	Organization	xsd:string
Phone	Data property	foaf:Agent	xsd:string
Birth date	Data property	Person or Broker	xsd:string
Gender	Object property	Person or Broker	Gender
Product of interest	Object property	foaf:Agent	Product
Country	Object property	foaf:Agent	Country

As you can see, properties may be assigned to the upper class, such as foaf:Agent. In this case, individuals of all its subclasses will be able to possess this property value. It may be assigned to a combination of classes, such as the “Tax ID” property assigned to the Person or Broker classes, but not to the Organization class. Or it can be assigned to a particular subclass. Similarly, the object property range can be a combination of several classes, or a single class of any level.

This demonstrates the flexibility of ontologies. A Person can have Birth date and Gender properties, while an Organization can’t. Organizations have a VAT number which the Persons

does not have. Keep in mind that any individual, by default, can have several values for the same property. For example, an Agent can have several products of interest.

You can limit the number of possible property values using the maxCardinality and minCardinality restrictions.
If maxCardinality is set to 1, it means that a property can have only one value. If minCardinality is set to 1, it means that the property must have at least one value for any object which can possess it.

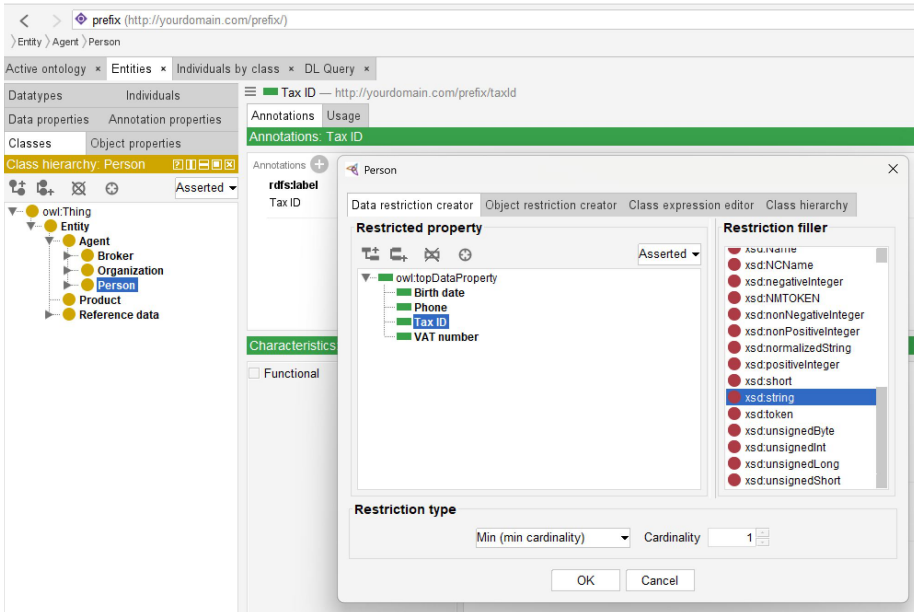


Fig. 8. Min cardinality restriction creator

Creating restrictions may seem a bit messy. To specify the cardinality restriction, you shall make the class that is this property's domain a subclass of a special restriction class. For example, let us make the taxId property mandatory (see *Fig. 8*).

To do this, select the Person class in the classes tree, then click "+" button in the "SubClass Of" section. Open "Data restriction creator" tab, choose taxId in the Restricted property list, and set minCardinality = 1 in the input below. After the restriction is created, it shold look as shown in *Fig. 9*.

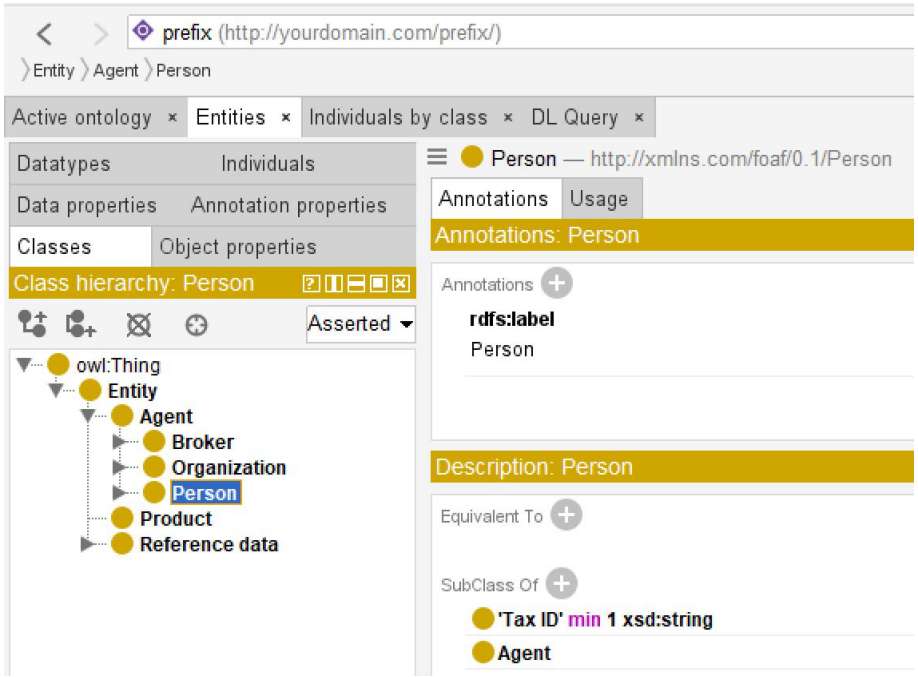


Fig. 9. The cardinality restriction in the class properties

For Object properties, we need to create reference data classes to enumerate their possible values. Let us create the Product class to keep kinds of our products, and the Reference data class with the Gender and Country subclasses. We shall create these class individuals right here in Protégé, using the Individuals by class tab – or you can do it later in the EKG Explorer interface.

Let us create all the properties listed in the Table 3 using the Data property and Object property tabs. The result shall look as shown in *Fig. 10* for the "Tax ID" property.

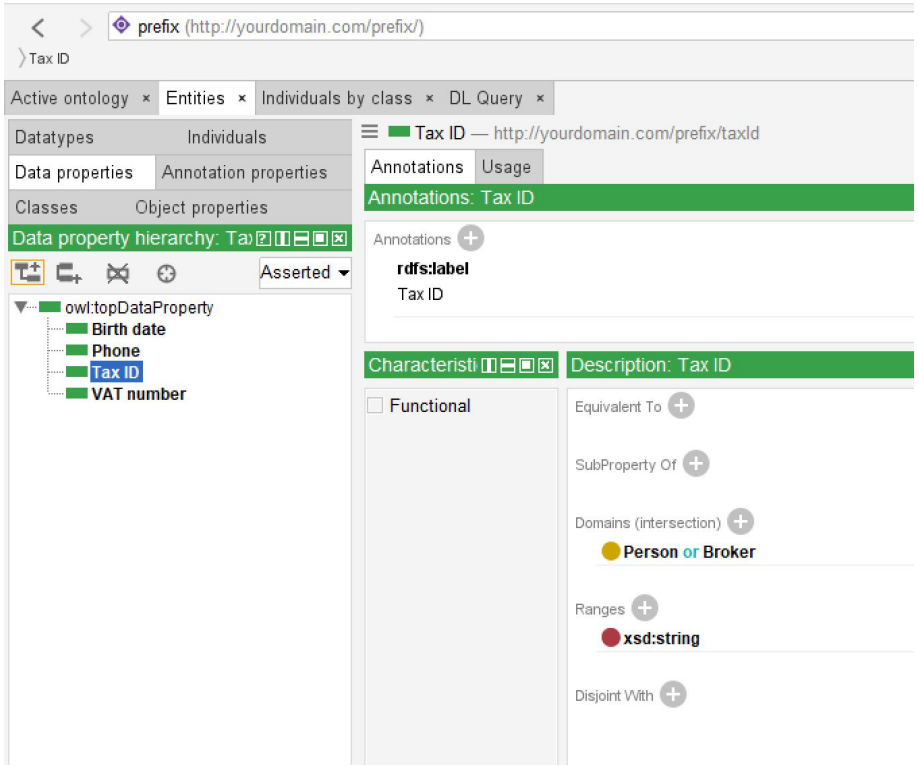


Fig. 10. "Tax ID" data property

Now save the resulting ontology in the Turtle syntax (a file shall have .ttl extension) and upload it to the EKG Platform dataset. You can do this through the Fuseki web interface using the “Add data” button, or by using the “Data model (TBox) Import” button on the DataVera EKG Explorer “Preferences” page. Click on the gear icon in the top-right corner of the screen to access it.

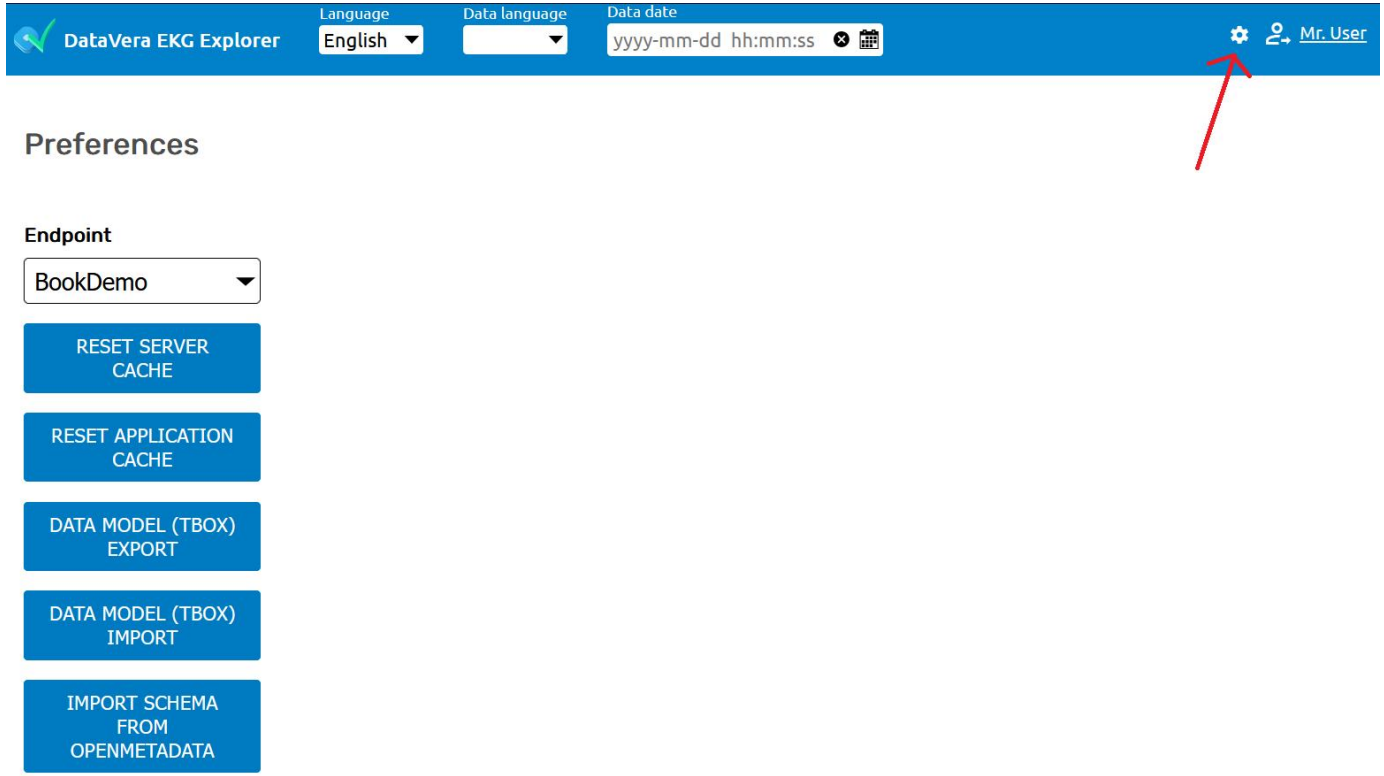


Fig. 11. EKG Explorer Preferences page with data model import/export tools

When you need to update the ontology structure in the future, you can export the model, edit it in Protégé, and import it back to the EKG Platform using these tools.

After the successful import, click the “Reset server cache” button, then the “Reset application cache” button, and return to the main user interface. Click on the class selector in the top left corner of the screen. The class selection tree shall look as follows:

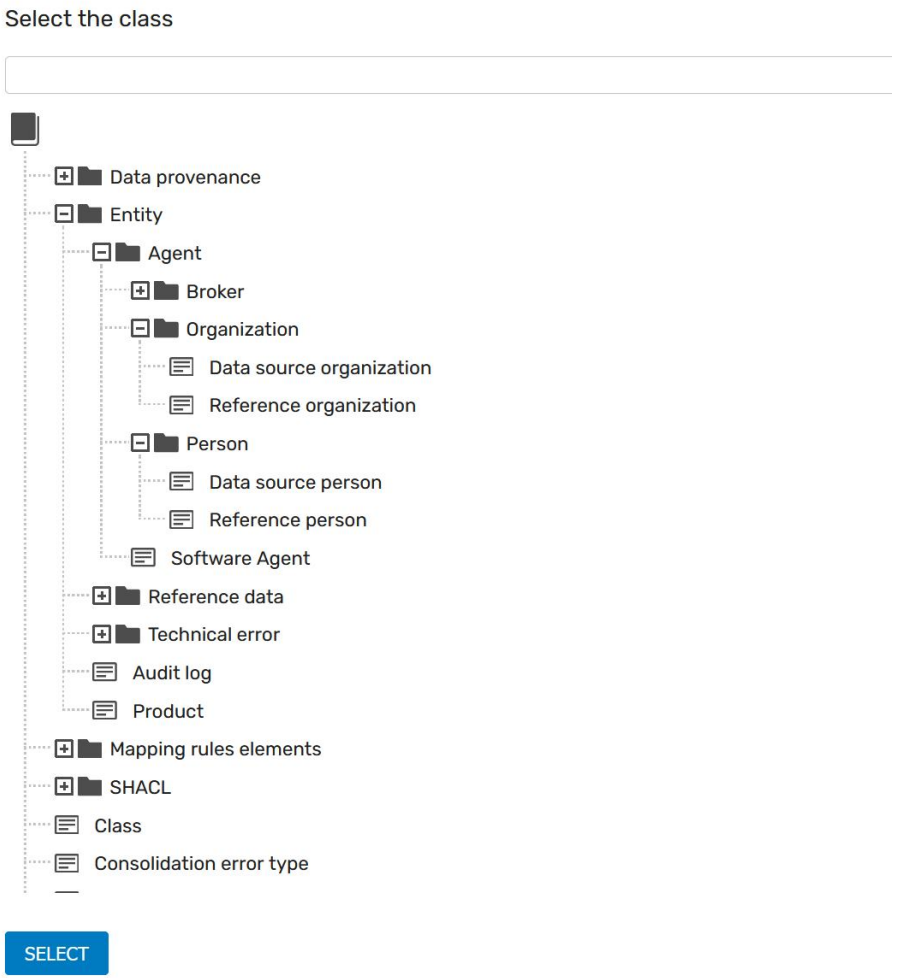


Fig. 11. EKG Explorer Preferences page with data model import/export tools

3. Create mapping rules and connect data sources

3.1. Data sources

Now we are ready to create mapping rules to populate our model. There is a hierarchy of these rules represented as subclasses of the “Mapping rules elements” class:

- Data source. The objects of this class represent the databases or services supplying data to the EKG Platform.
- Class map. Each of these objects represents a correspondence between a data source structure element (a database table, in our example) and an ontology class. Using these objects, we can assert that the objects from the “customers” table shall be imported to EKG as the instances of the Customer class.
- Predicate map. These objects represent a correspondence between the database table columns and the ontology properties. For example, we can say that the “t_id” column of the “customers” table corresponds to the taxId ontology property.
- Object map. This kind of objects is used to map the particular database row identifiers or enumeration elements to the ontology individuals. For example, we can say that the “F” letter in the “gender” column of the “customer” table corresponds to the “Female” gender.

The general schema of the last three mapping rule types is given in *Fig. 13*.

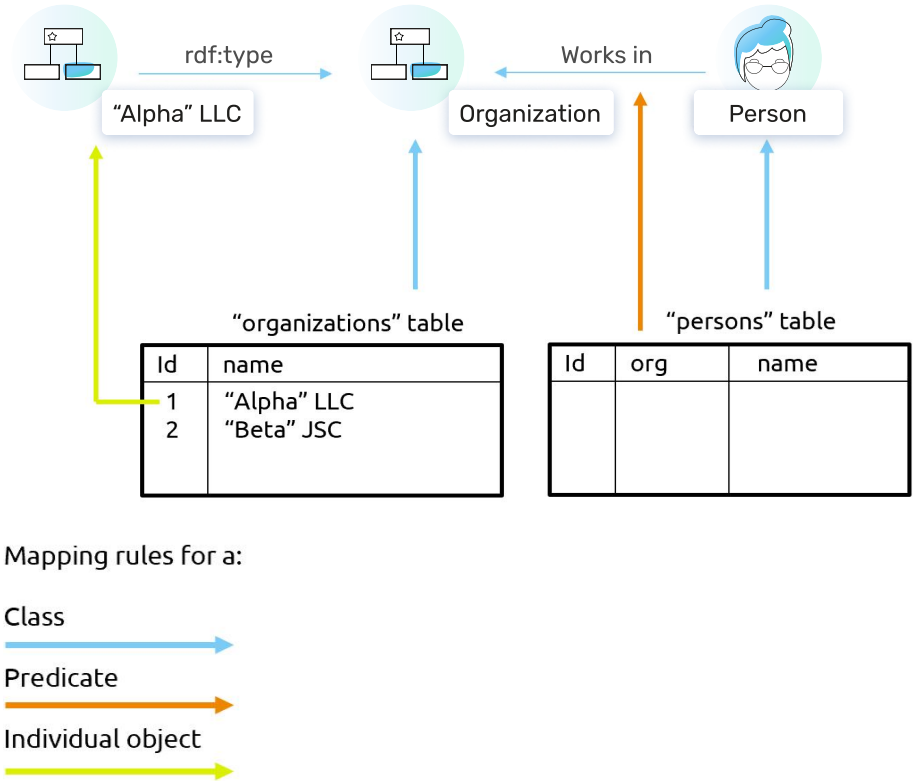


Fig. 13. The general mapping rules schema

As a first step, we have to define a Data source. In our example we will use the PostgreSQL database named "sample". Its structure is shown in *Fig. 14*.

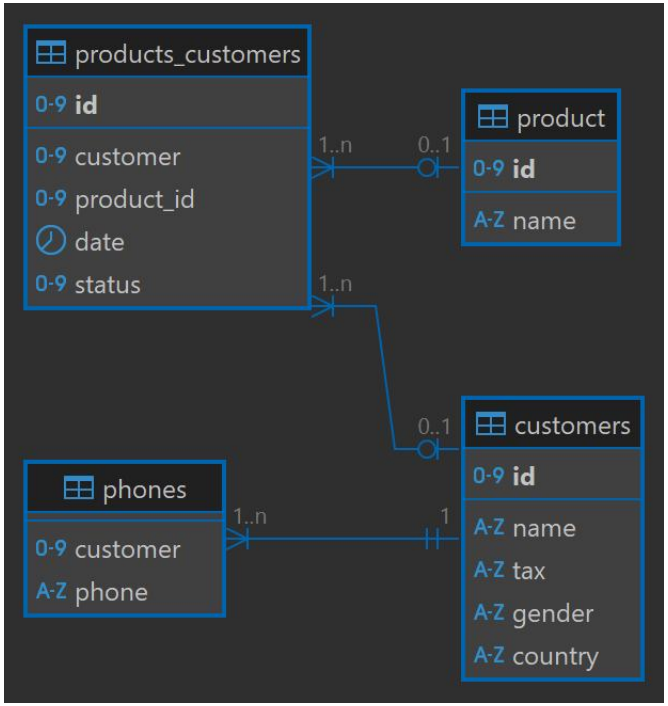


Fig. 14. Sample database structure

To create a data source, choose “Data source” class in the classes tree (it is located in the “Mapping rules elements” superclass), and click the “+” button. This will create a new row in the objects list. Double-click on the randomly generated object URI and change it to the “DataSource_Sample”. Then fill in the “Connection string” box, and type “postgresql” in the “Data source type” box. Assign it the name “Sample”. The screen should look as shown in *Fig. 15*. Then click the “Save” button indicated by the red arrow.

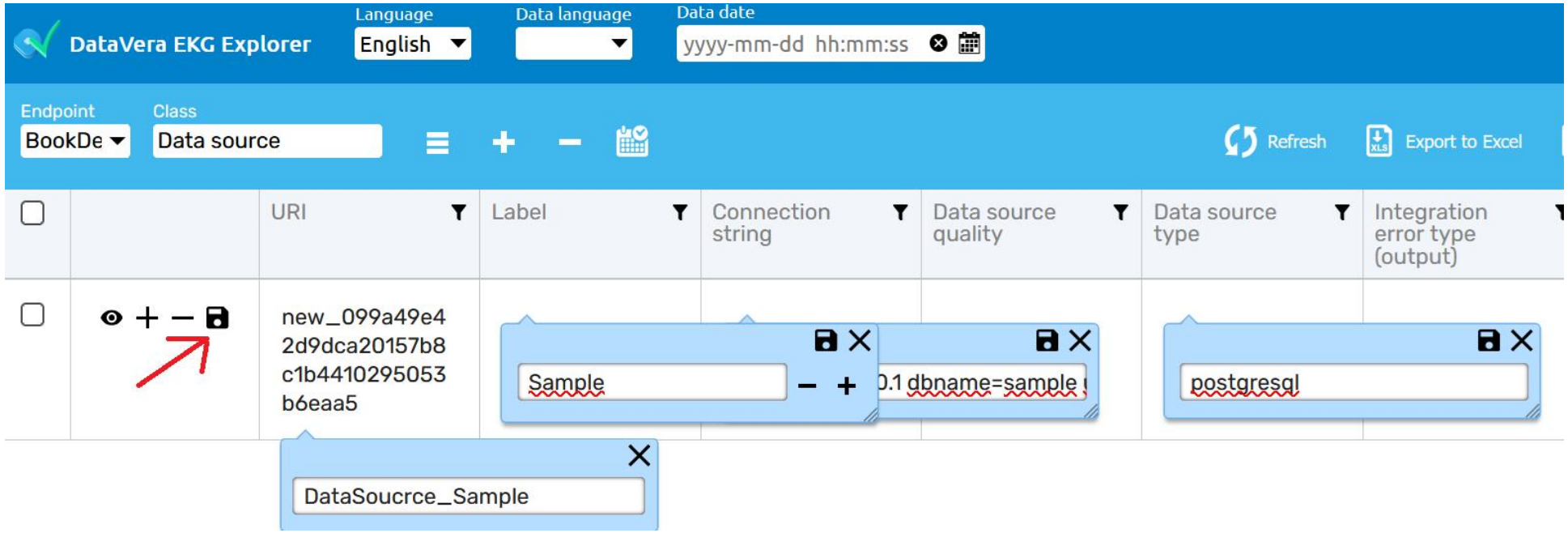


Fig. 15 Creating data source

Alternatively, you can simply change URI, fill in something in the mandatory fields, create an object, and then double-click the URI column to open the object editor dialog shown in *Fig. 16*.

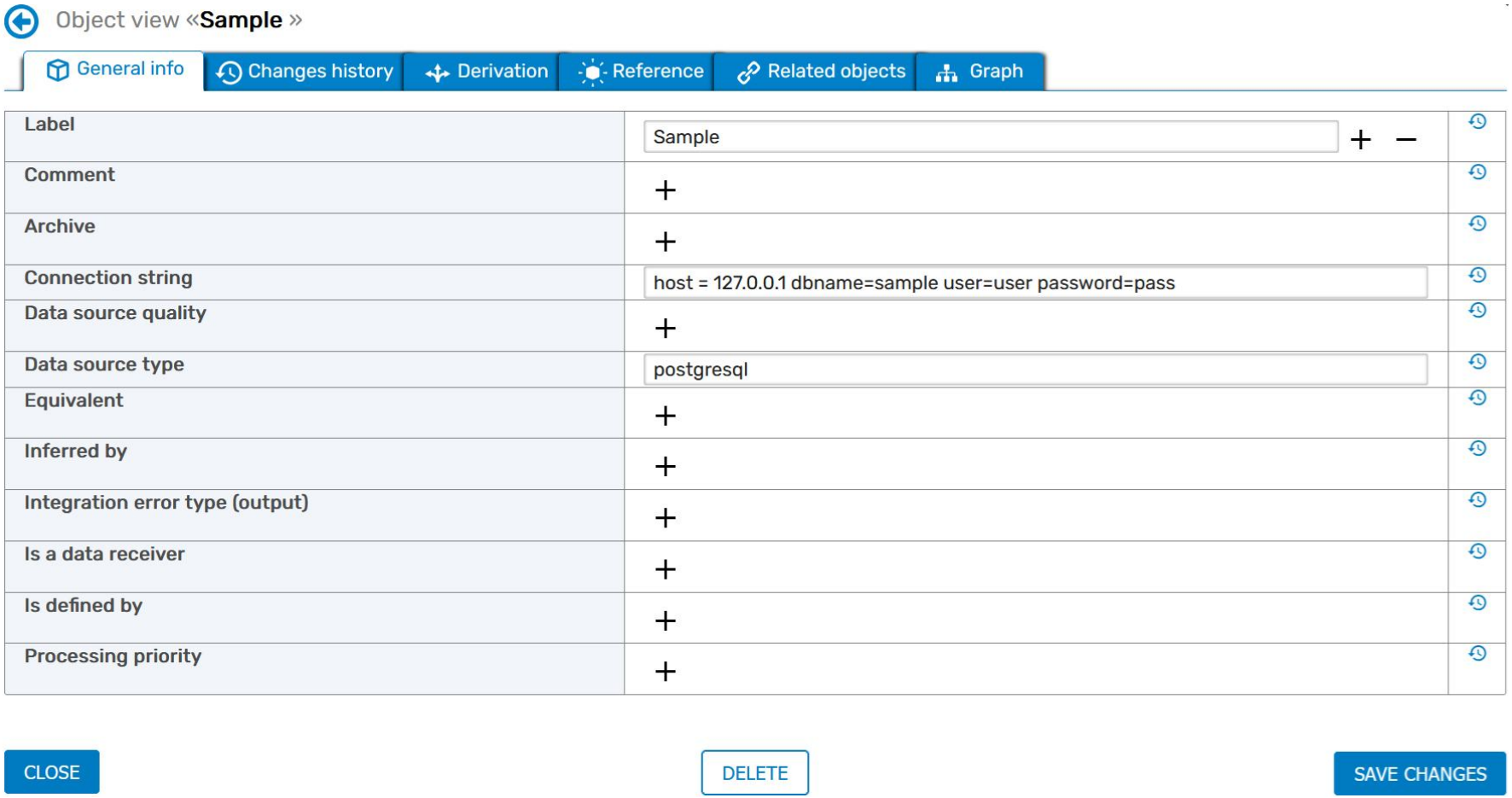


Fig. 16. Object editing dialog

3.2. Class and property maps. OpenMetadata integration

Now we need to create Class maps and Predicate maps. This can be done manually in the user interface, or better using the Excel import tool, which we will explore later. Also, we can use OpenMetadata integration feature to initially populate these maps with the source database description, and then just fill in the corresponding classes and properties of the ontology.

Go to the “Preferences” page. Choose an endpoint and click the “Import schema from OpenMetadata” button. Choose the Data source, then the database, schema, and data table. The system will display a list of the columns of the selected table. Click on the “Load Metadata” button to proceed. Then the list of created PredicateMaps will be displayed.

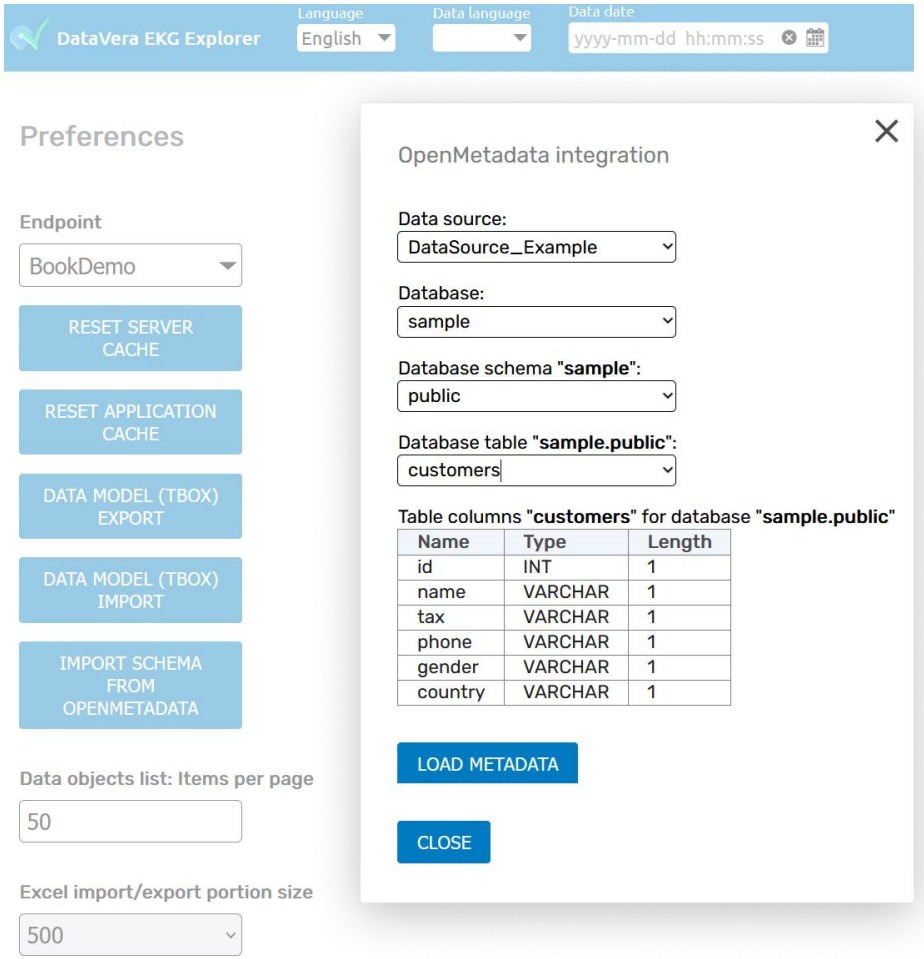


Fig. 17. Using OpenMetadata integration tool

After using the OpenMetadata import tool on the "customers" table of our data source and navigating to the "Class map" class, we will see one row in the objects list. It defines the rule of importing objects from the public.customers table into the EKG platform storage. To complete this rule, we need to select a class in the "Mapped class" column. We have three subclasses of the Agent class, let us start with the first one – the Person. Start typing this word and select an item from the drop-down menu in the field value editor pop-up. But the "customers" table in our database contains not only the Persons, but the Organizations too. We have to restrict the set of the records which will be imported to the Person class by defining the SQL WHERE clause. Input the following string in the "Select condition" cell: "kind = 1", as shown in *Fig. 18*. Save the record.

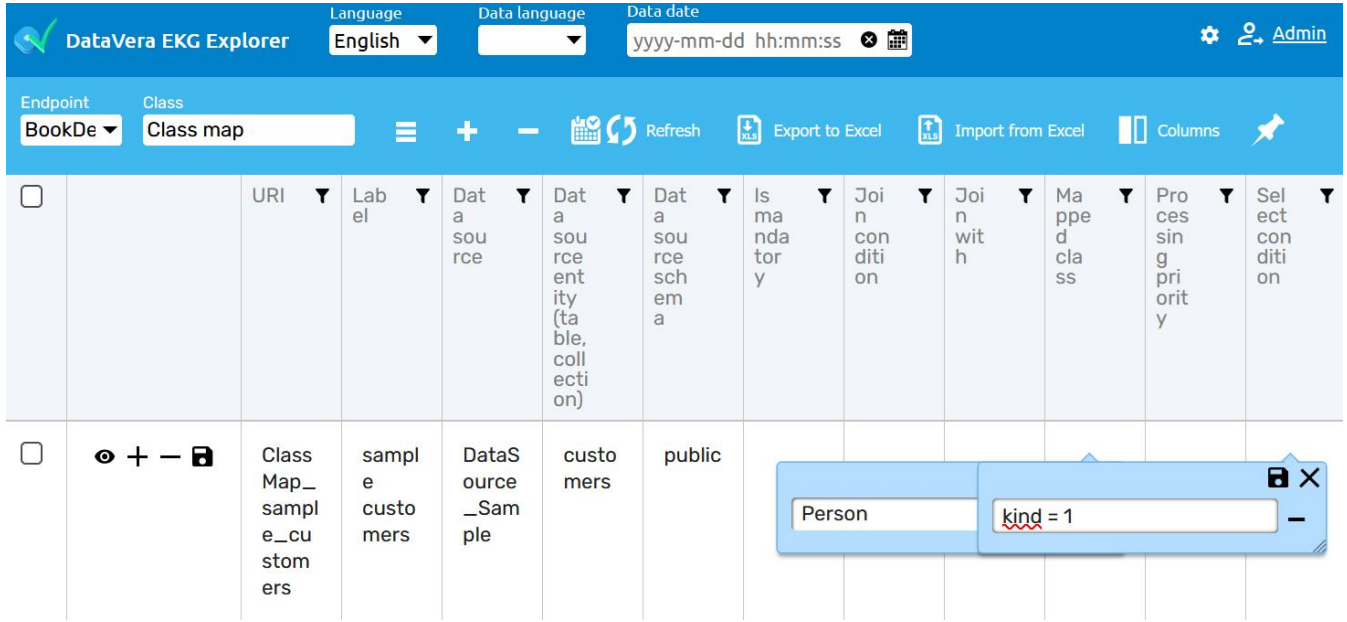


Fig. 18. Editing the Class map rule

Now, duplicate this entry and create a mapping for the Organization class with the URI ClassMap_sample_organizations and the Select condition set to “kind = 2”. Real select conditions can be much more complex. For example, they may include subqueries.

Switch to the Predicate map class. Several rows should appear, each representing a column in the “customers” table. Let us edit them in Excel. Click the “Export to Excel” button on the toolbar. You shall see the Excel file as shown in *Fig. 19*, but with the last three columns empty:

	A	B	C	D	E	F	G	H
1	**Class**	**Predica**	**http://datavera.kz/ekg/PredicateMap**					
2	**URI**	**Label**	**Belongs to Class map**	**Data source**	**e property (co**	**Mapped property**	**Is a primary key property**	**Use objects map**
3	Predic	sample cu	sample customers (ClassMap_sample_cu	Sample (Da	id	http://datavera.kz/ekg/hasOriginCode	true	
4	Predic	sample cu	sample customers (ClassMap_sample_cu	Sample (Da	name	http://www.w3.org/2000/01/rdf-schema#label		
5	Predic	sample cu	sample customers (ClassMap_sample_cu	Sample (Da	tax	taxId		
6	Predic	sample cu	sample customers (ClassMap_sample_cu	Sample (Da	gender	hasGender		ClassMap_Gender
7	Predic	sample cu	sample customers (ClassMap_sample_cu	Sample (Da	country	hasCountry		ClassMap_Country
8	Predic	sample or	sample organizations (ClassMap_sample	Sample (Da	id	http://datavera.kz/ekg/hasOriginCode	true	
9	Predic	sample or	sample organizations (ClassMap_sample	Sample (Da	name	http://www.w3.org/2000/01/rdf-schema#label		
10	Predic	sample or	sample organizations (ClassMap_sample	Sample (Da	tax	VATNumber		
11	Predic	sample or	sample organizations (ClassMap_sample	Sample (Da	country	hasCountry		ClassMap_Country

Fig. 19. Excel file ready for upload

Now, we will fill some information into this file, and upload it back to EKG. First, we have to choose a Mapped property for each column of the table. Here, we shall should enter the URIs (identifiers) of the ontology properties, which you can view in Protégé. Please note the two properties that have a full prefix starting with http://:

- http://datavera.kz/ekg/hasOriginCode - this property belongs to the technical ontology used by EKG Platform. It contains the identifier of each loaded data object as it appears in the data source. We shall map the “id” column to this property. We will also mark these properties as primary keys by typing “true” in the “Is a primary key property” column.
- http://www.w3.org/2000/01/rdf-schema#label – this property, defined in the RDFS specification, serves as the readable label of the object. We have already used rdfs:label when creating our ontology.

We omit the prefix for other properties since they have our ontology’s default prefix.

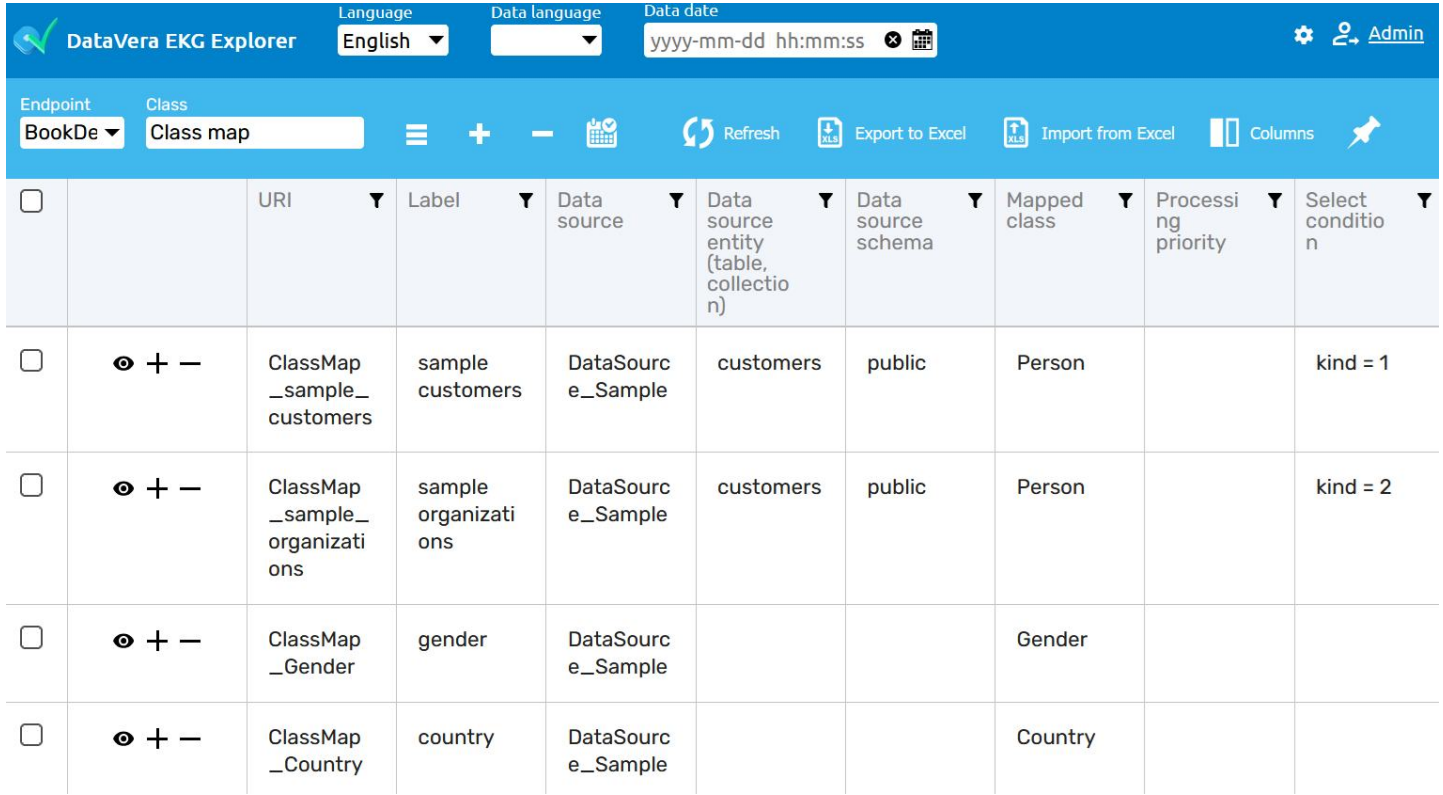
When you open the downloaded file, it will contain only five rows, which were created during the metadata import from OpenMetadata. They are all bound to the “sample customers (ClassMap_sample_customers)” ClassMap, which was also created automatically. Remember that this ClassMap works only with Persons. We have created another ClassMap, ClassMap_sample_organizations, to import our organizations. We shall make a copy of our PredicateMaps for the Organization class. Copy all the rows except hasGender. Replace the word “customers” with the “organization” in these rows. Voila, you have created four more PredicateMaps! Please make sure that you have changed their URIs: each ontology entity must have a unique URI. We did this with our replace operation, which changed URIs from PredicateMap_sample_customers_id to the PredicateMap_sample_organizations_id, etc.

We have a few more details here. The “tax” column has to be mapped to the VATNumber property for Organizations instead of taxId. Also, we have to handle the property values for enumerations: gender and country.

The gender is indicated in our sample database by the letters “M” and “F” contained in the “gender” column. When loading into EKG, we shall convert these letters to the URIs of the individuals we have created in Protégé: Male and Female. To do this, we have to define the ObjectMaps. But let us finish with the PredicateMaps first. Enter the URIs that we will create on the next step, “ClassMap_Gender” and “ClassMap_Country”, into the “Use objects map” column for the hasGender and hasCountry properties. Save the edited file, click the “Import from Excel” button in the EKG interface, choose the saved file and check the “replace values of the existing ones” option, then click the “Import” button. After successful upload, you shall see the updated set of PredicateMaps in the interface.

3.3. Object maps (enumerations)

As a first step, we have to return to the Class map list and create two new class maps: ClassMap_Gender and ClassMap_Country. The resulting set of ClassMaps shall look as shown in *Fig. 20*.



	URI	Label	Data source	Data source entity (table, collection)	Data source schema	Mapped class	Processing priority	Select condition
	ClassMap_sample_customers	sample customers	DataSource_Sample	customers	public	Person		kind = 1
	ClassMap_sample_organizations	sample organizations	DataSource_Sample	customers	public	Person		kind = 2
	ClassMap_Gender	gender	DataSource_Sample			Gender		
	ClassMap_Country	country	DataSource_Sample			Country		

Fig. 20. Class map list

Then go to the Object map class. Click “Export to Excel” button to obtain the empty Excel template for creating Object maps. Fill the first row with the following values:

- URI = ObjectMap_Gender_M
- Label = gender M
- Belongs to Class map = ClassMap_Gender
- Data source = DataSource_Sample
- Data source object id = M
- Mapped object = Male

Copy this row for the Female gender. Then do the same for the list of countries. You shall obtain the file looking as shown in *Fig. 21.*

Load this file into the ObjectMap class and check the result in EKG Explorer UI.

	A	B	C	D	E	F
1	**Classes**	**Object ma**	**http://datavera.kz/ekg/ObjectMap**			
2	**URI**	**Label**	**Belongs to Class map**	**Data source**	**Data source object id**	**Mapped object**
3	ObjectMap_Gender_M	gender M	ClassMap_Gender	DataSource_Sample	M	Male
4	ObjectMap_Gender_F	gender F	ClassMap_Gender	DataSource_Sample	F	Female
5	ObjectMap_Country_KAZ	country KAZ	ClassMap_Country	DataSource_Sample	KAZ	Kazakhstan
6	ObjectMap_Country_SAU	country KSAU	ClassMap_Country	DataSource_Sample	SAU	Saudi_Arabia
7	ObjectMap_Country_TUR	country TUR	ClassMap_Country	DataSource_Sample	TUR	Turkey
8	ObjectMap_Country_UAE	country UAE	ClassMap_Country	DataSource_Sample	UAE	UAE
9	ObjectMap_Country_IRQ	country IRQ	ClassMap_Country	DataSource_Sample	IRQ	Iraq
10	ObjectMap_Country_EGY	country EGY	ClassMap_Country	DataSource_Sample	EGY	Egypt

Fig. 21. Excel file for ObjectMap objects import

3.4. Complex mapping rules

A typical task is to combine information from multiple data source tables into one ontology class. In our example, we have the products_customers table, which links customers to the products table containing the list of product types. Here, we want to save the products into the “Product of interest” customer property. To achieve this, we need to join three tables when selecting customers from data source.

We will create a ClassMap object for each table and then join them together. The resulting Excel file shall look as follows:

	A	D	F	G	H	
1	**Classes**	**http://datavera.kz/ekg/ClassMap**				
2	**URI**	**ource entity (table, colle**	**Mapped class**	**Join with**	**Join condition**	**Select**
3	ClassMap_sample_customers	customers	Person (http://xmlns			kind = 1
4	ClassMap_sample_products_customers	products_customers	Person (http://xmlns	ClassMap_sample_customers	customers.id = product_customers.customer	
5	ClassMap_sample_product	product	Person (http://xmlns	ClassMap_sample_customers	product_customers.product_id = product.id	
6	ClassMap_sample_organizations	customers	Person (http://xmlns		customers.id = product_customers.customer	kind = 2
7	ClassMap_sample_products_customers_orga	products_customers	Person (http://xmlns	ClassMap_sample_organizations	product_customers.product_id = product.id	
8	ClassMap_sample_product_organizations	product	Person (http://xmlns	ClassMap_sample_organizations		
9	ClassMap_Gender	(DataSource_Sample)	Gender (Gender)			
10	ClassMap_Country	(DataSource_Sample)	Country (Country)			

Fig. 22. Joining tables using Class maps

In the “Join with” column we insert the URI of the root ClassMap – “ClassMap_sample_customers”, which maps the class to the main table (“customers” in our case). In the “Join condition” column we place the SQL expression from the ON clause of the join.

In the “Join with” column we insert the URI of the root ClassMap – “ClassMap_sample_customers”, which maps the class to the main table (“customers” in our case). In the “Join condition” column we place the SQL expression from the ON clause of the join.

When compiling SQL query from the rules, EKG always uses LEFT JOIN. This is necessary to avoid missing records from the main table which does not have the corresponding records in the joined tables – for example, customers who do not have the products of interest. If the sequence of the joined table matters, as in our case (we have to join with products_customers, then with the product table), we shall also set the values for the Processing priority property of PredicateMaps. Objects with the higher priorities are processed first, so the tables will be joined in the right order if the ClassMap_sample_products_customers have Processing priority = 100, and ClassMap_sample_product has Processing priority = 90.

After setting up the ClassMaps, we have to map the properties. These properties shall be bound to the ClassMap of the table that contains the value we want to extract. Since we want to obtain the names of the products of interest, we shall assign our predicate maps to the ClassMap_sample_product and ClassMap_sample_product_organizations respectively, as shown in *Fig. 23*.

As every customer can have multiple products of interest, the respective Agent class objects in the EKG will have multiple values for the “Product of interest” property. We can think of it as of a property with an array type.

If we want to link the Customers to the appropriate Products expressed as ontology objects, we need to create the ClassMap and PredicateMaps to populate the Product class. Then, we shall create the first joined ClassMap for our Agents, linking “customers” and “products_customers” tables. We should map the “products_customers .product_id” column to the “Product of interest” ObjectProperty without specifying the value of the “Use objects map” property. This will signal the system to generate the link to other objects loaded from the data source.

		URI	Label	Belongs to Class map	Data source	Data source property (column, key)	Is a primary key property	Mapped property
☐	👁 + −	PredicateMap_sample_customers_product	sample customers product	sample product	DataSource_Sample	name		Product of interest
☐	👁 + −	PredicateMap_sample_organization_product	sample organization product	sample product organizations	DataSource_Sample	name		Product of interest

Fig. 23. Predicate maps for the product names

The link between the loaded data objects is established due to the URI generation rules. When loading data, EKG automatically creates the URIs using the following template: [data source URI]_[class URI]_[unique key from the data source]. For example, a person with id = 1 in the “customers” table will have the following URI: DataSource_Sample_Person_1.

Respectively, the product with id = 4 will have URI DataSource_Sample_Product_4. When resolving the value for the “Product of interest” property of the DataSource_Sample_Person_1, the system will generate id for the referenced product using the same rule, obtaining DataSource_Sample_Product_4, and will create a valid link to the product.

The mapping rules syntax is quite rich. Let us highlight some mapping rules features without going into details.

- When joining tables, you can create several join rules for the same joined table at once (this will result in only one join in the SQL query). This is useful when there is a table containing several related data pieces that must be saved in different properties. Imagine a table named “codes” with the “person_id”, “code_type” and “code_value” fields. There are two records in this table for each person: the record with code_type = 1 contains VAT number in the code_value field, while the record with code_type = 2 contains the internal customer code. In this case you should create two Class maps, indicating code_type = [1 or 2] in the “Select condition” property. Both Class maps will join the same table “codes” with the main table. Then create two Predicate maps referring to each of the Class maps. Both Predicate maps will have “Data source property (column, key)” = “code_value”, but they will map it to different data model properties.
- Predicate maps can have more than one row marked as the Primary key property. This happens when there is no unique ID column in the table, and you need to use several columns combination as the synthetic key. In this case, the values of all the key properties will be used to generate a unique key. For example, if there are the documents, which we identify by the combination of document number and the owner ID, their URIs may look like DataSource_Sample_Document_0123456_12. In this case it is very important to set the Processing priority value of the Predicate maps with “Is a primary key property” = true, to ensure that the key values in the URI will always have the right order: in our example the property map for the “Document number” shall have priority = 100, for the “Document owner” priority = 90.
- The Predicate maps can map a model property to a Constant value.
- The Predicate maps can extract values from data source columns using regular expressions. This is useful when a data source column contains several property values concatenated into one string. For example, document validity dates may be stored as “2024-01-15 – 2034-01-14”. In this case we have to split these values:

the first date shall be stored as the “Issue date” property value, while the second one is the “Valid to” property value. To achieve this, create two Predicate maps for this field, and fill in the “Use regular expression” property with the values like “^(?:\d\d[\.\-,]\d\d[\.\-,]\d{2,4})” and “\d\d[\.\-,]\d\d[\.\-,]\d{2,4}\s*\-*\s*(\d\d[\.\-,]\d\d[\.\-,]\d{2,4})\s*$”. You can specify several regular expressions in one Predicate map.

- The “Take the value related to the biggest value of field” property of the Property map can be used when there are several values in the joined table, and you need to take only the last of these. For example, there can be a table containing the customer visit records. Each record has the “date” and “branch” fields. You need to store the branch which the customer has visited last as the value of the “Preferred branch” property. Map this property to the “branch” field, and indicate the “date” field in the “Take the value related with the biggest value of field” property of the PredicateMap.
- The http://datavera.kz/ekg/lastChanged data model property is especially important. It shall be mapped to the data source column containing the record update time. If there is no such column, you will not be able to track the changes and load them into the EKG instantly (this topic is discussed below). The column may contain data in the YYYY-MM-DD HH24:MI:SS format, or a linux timestamp, or a timestamp in the YYYYMMDDHHMISS000000 format. For joined tables, each table can contain its own timestamp: in this case, the system will pick the latest timestamp value from all the joined tables for each record.

When you have finished composing the Mapping rules, you shall restart the ekg-worker container. It will import the new rules and start processing the incoming data from data source (for the tables that have the ekg:lastChanged property mapped). You can monitor this process using the container logs.

Also, you can perform the initial full data import from each mapped source using the load_data script. Please contact us for assistance in this case.

4. Define normalization rules

The normalization rules are intended for data cleaning. They can remove unnecessary symbols, split strings, and so on. They can be implemented as the regular expressions or use the EKG platform built-in functions.

These rules are stored as instances of "Normalization function" class. Each object of this class describes a single normalization procedure. Every procedure can be applied to several properties. This is represented by individuals of the "Normalization function application" class.

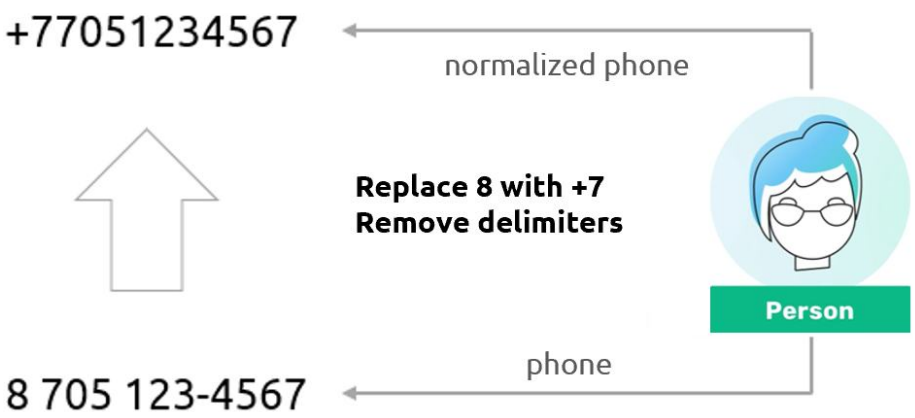


Fig. 24. Normalization rule's structure

For example, CamelCase is a built-in normalization procedure. It is declared in the EKG platform ontology as an instance of the "Normalization function" class. You can bind this procedure to pairs of the data model properties. The recommended practice is to create a property that will store the initial value acquired from the data source, such as nameBeforeNorm, and the property for the normalized value – foaf:name in our example. Create an instance of the "Normalization function application" class with the following property values:

- "Apply normalization function" = CamelCase
- "Apply to property" = nameBeforeNorm
- "Save result as property" = foaf:name

In this case, you should map the data source column to the nameBeforeNorm property. When saving objects acquired from data sources, EKG Platform will save the value "as is" in the nameBeforeNorm property, then apply the CamelCase function and save the result to the foaf:name property.

4. Define normalization rules

The CamelCase function is a built-in and does not have a body declaration. You can create your own functions and define them using the following properties:

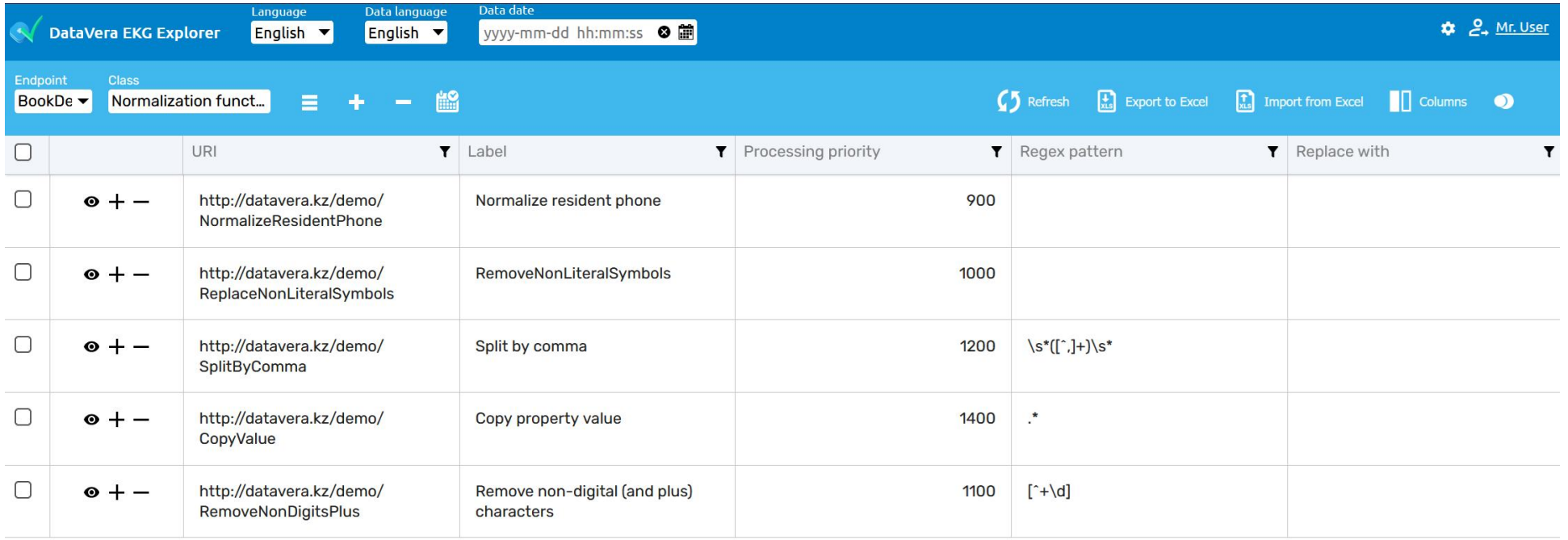
- "Regex pattern" – regular expression search
- "Replace with" – regular expression replace

The “Replace with” parameter can be omitted. In this case, each match of the “Regex pattern” will be saved as the separate property value. For example, the “Split by comma” function has the “Regex pattern” = “\s*([^,]+)\s*“ and no “Replace with”. This pattern includes parentheses to indicate the captured pattern, which captures everything except for the comma. If there are several commas in the string, each part of the string between commas will be captured as a regular expression result. Each result will be stored as the separate value of the resulting property. This function is useful for separating multiple phone numbers divided by commas, etc.

Another example is the RemoveDigits function. It has the “[\d]” pattern and an empty replacement. This means that it will replace each digit in the input with an empty value. Please note that in ontologies, the absence of a property value is significantly different from the presence of an empty value.

4. Define normalization rules

The normalization functions can be viewed and edited using the EKG Explorer UI:



	URI	Label	Processing priority	Regex pattern	Replace with
	http://datavera.kz/demo/NormalizeResidentPhone	Normalize resident phone	900		
	http://datavera.kz/demo/ReplaceNonLiteralSymbols	RemoveNonLiteralSymbols	1000		
	http://datavera.kz/demo/SplitByComma	Split by comma	1200	\s*([^,]+)\s*	
	http://datavera.kz/demo/CopyValue	Copy property value	1400	.*	
	http://datavera.kz/demo/RemoveNonDigitsPlus	Remove non-digital (and plus) characters	1100	[^+\d]	

Fig. 25. Normalization functions in the EKG Explorer UI

After updating the functions or their applications, you shall reset the server cache using the Preferences page or restart the EKG Provider to apply the changes.

There is a limitation in the current implementation of the normalization functions: the currently used regex library does not support UTF-8 characters. However, the built-in normalization functions do support UTF-8.

4. Define normalization rules

Let us list the built-in normalization functions that you might find useful:

Table 4. **Built-in normalization functions**

Function URI	*Function purpose*
CamelCase	Convert the string to camel case – i.e. capitalize the first letter of each word and make the following letters lowercase
LCase, UCase	Convert the entire string to lowercase or uppercase
ReplaceNonLiteralSymbols	Remove all characters except letters, spaces and hyphens (this may be used to normalize names)
ReplaceStrangeQuots	Replace all types of quotes with regular straight quotes
RemoveCrLf	Remove CR, LF and tabulation
SwitchAlphabet	Replace the Latin characters in the Cyrillic string, and vice versa. If the most characters in the string are Latin, it is considered Latin. In this case, if the string contains Cyrillic characters common to both Latin and Cyrillic alphabets (such as a, o, etc.), they are converted to their Latin analogues.

When splitting strings, you can place the first value in one property, and the following values to another. It is useful when separating phones: the first phone can be considered as principal and written to the foaf:phone property, the other phones are considered supplementary and saved in the extraPhone property. In this case you can use the “Save first result as property” property of the “Normalization function application” instance.

4. Define normalization rules

Multiple normalization functions can be applied to each pair of the properties. The application order may be important, so it can be set by defining the “Processing priority” property value. Functions with a higher priority will be applied first.

By default, the normalization functions are applied to all the values of the defined properties, regardless of which classes instances possess them. In some cases, you might need to apply the normalization only to the instances of the specified classes. This can be done by setting the “Mapped class” (ekg:hasClass) property value for a “Normalization function application” class object.

You can easily debug normalization functions. Open the class to which individuals you have defined the functions. Choose the pair of columns for the properties before and after normalization. Enter a value to the property before normalization of some individual and save the record. You will instantly see that the normalized property value is written into the normalized property value:

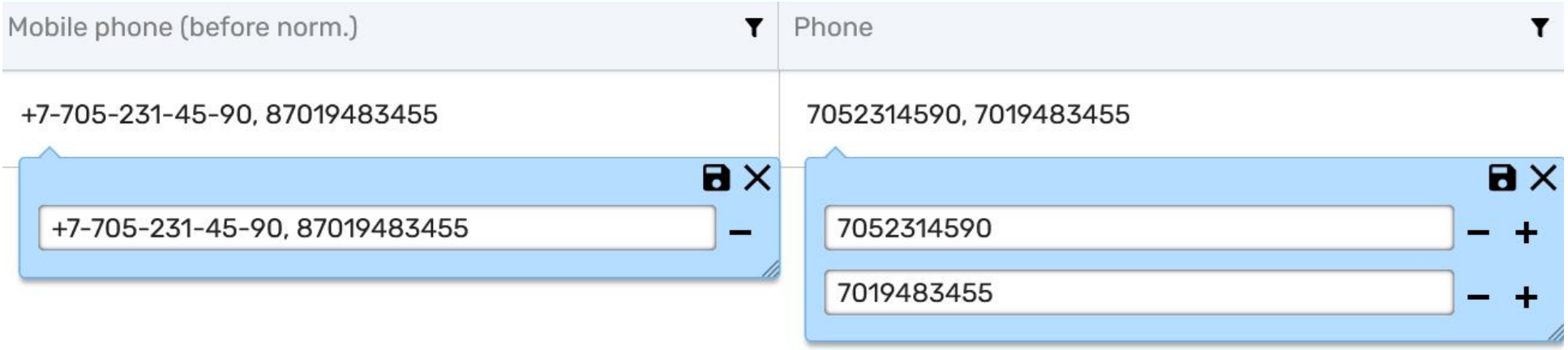


Fig. 25. Normalization functions in the EKG Explorer UI

As you can see in Fig. 26, a set of normalization rules applied to the “Mobile phone (before normalization)” property transforms several comma-separated values into separate values of the “Phone” property. The extra symbols are removed, and the normalized phone numbers contain exactly 10 digits starting with the operator code, without the country code. Of course, the logic of these rules can be changed as needed.

5. Set up validation rules and assess data quality

5.1. Constraints representation in the ontology

Validation rules, or constraints, are stored in the ontology according to the W3C SHACL specification. This specification describes several ways of expressing the fact that some part of the knowledge graph must comply with some certain requirements. EKG Platform implements one of these methods, the SPARQL Constraints. The restriction is defined by creating two ontology objects: instances of the shacl:PropertyShape and shacl:SPARQLConstraint classes.

The first object has the following properties:

- Target class – specifies the class to which the rule applies for instances
- SPARQL expression – points to a shacl:SPARQLConstraint object
- Has severity – can be set to either shacl:Violation (default value) or shacl:Warning. The Violation level constraints
- Reject properties – an extension added by DataVera. If the values of this property are set, only the values of the specified properties will be marked as invalid if constraint fails. By default, the rule marks all properties used in the SPARQL expression as invalid.

The shacl:SPARQLConstraint class objects have the following properties:

- Message – the violation message text
- Select – ASK or SELECT SPARQL query that checks the restriction. If the ASK query returns true or SELECT query returns anything, then the check is passed, and there is no violation
- Inverse logic – a Boolean flag. If set to true, the above-described logic is inverted: the constraint holds if ASK returns false or the SELECT returns no result. This feature is an extension specific to DataVera EKG Platform.

You can prepare constraints in Protégé and import them to the Fuseki endpoint, or create them in the Excel file and import it, or create constraints directly in the user interface.

When creating constraints in the user interface, create a SPARQL constraint first:

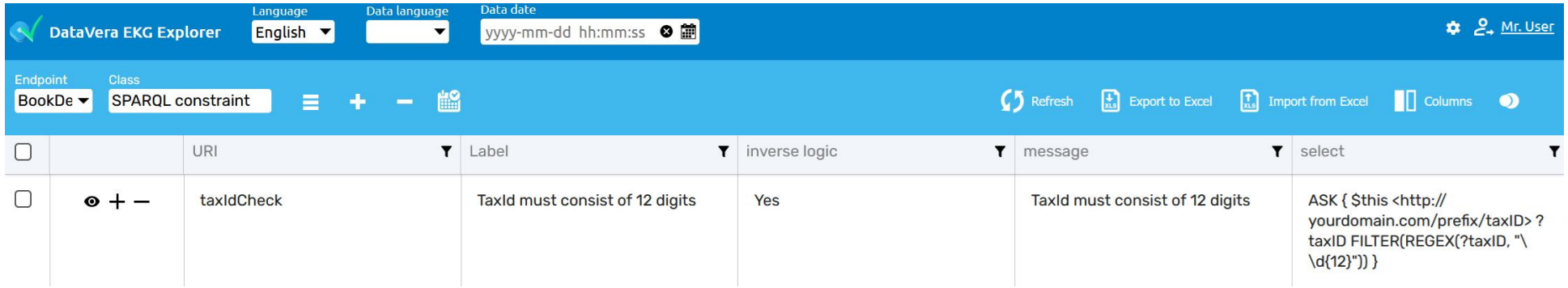


Fig. 27. SPARQL constraint in the EKG Explorer user interface

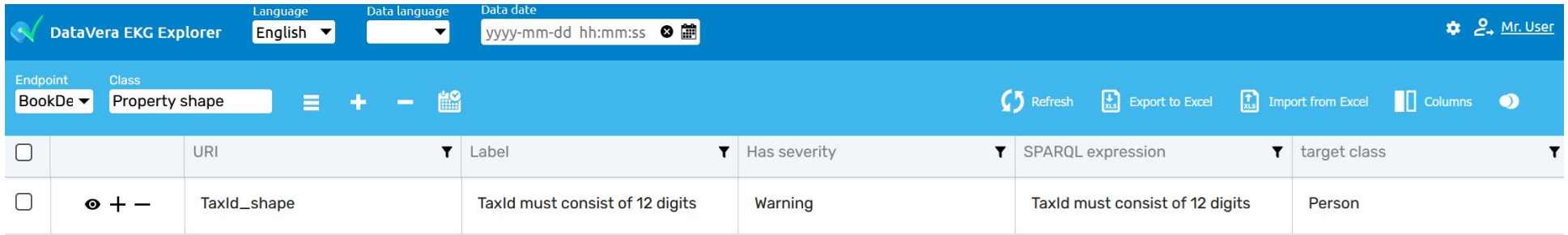


Fig. 28. Property shape in the EKG Explorer user interface

5.2. Create constraints

Creating constraints is a rather complex task. We will present several examples here, although the variety of possible use cases is vast.

Let us start with the simple rule which checks that the Tax ID consists of 12 digits. The SPARQL query is:

```
ASK {
    $this <http://yourdomain.com/prefix/
taxId> ?taxID
    FILTER(REGEX(?taxID, "\\d{12}"))
}
```

The SPARQL query syntax explanation is beyond the scope of this guide. There are several sources available on the Internet. However, let us note that the $this expression is a variable representing the object being checked – a Person class instance. http://yourdomain.com/prefix/taxID is the full URI (identifier) of the taxId property – we must write it including our ontology's default prefix here. ?taxID is the variable which will store the property value. The first line reads as "take the taxId property of the checked object and place it into the ?taxID variable".

The next line contains FILTER clause. This filter checks that the ?taxID variable contains 12 digits using a regular expression function. This means that if the checked object has the value of taxID property that contains 12 digits, the ASK query will return true, and our constraint holds.

But there is a problem. If our Person does not have a taxId value at all, the query will return false, and constraint fails. We don't want to mark every Person without taxId as invalid (for example, we might not know the taxId of the prospective customers). To avoid this, we shall use the above-mentioned "Inverse logic" flag. When we set it to true, EKG will expect

that the ASK query returns false if everything is ok. Let us add negation to the filter:

```
ASK {
    $this <http://yourdomain.com/prefix/
taxId> ?taxID
    FILTER(!REGEX(?taxID, "\\d{12}"))
}
```

Now everything is fine. If our object does not have a taxId value or the taxId value consists of 12 digits, the query will return false. Because the “Inverse logic” flag is set, the constraint holds. We will use the “Inverse logic” flag in all the rules we will discuss below.

Consider a more complex example. Let us assume that the first 6 digits of taxId represent the year, month, and day of birth (this holds for Kazakhstan). Now the SPARQL query will use two properties of the Person, and use several built-in SPARQL function to process their values:

```
ASK {
    $this <http://yourdomain.com/prefix/
taxId> ?taxID.
    $this <http://yourdomain.com/prefix/
birthDate> ?birth
    FILTER(CONCAT(SUBSTR(STR(?birth), 3,
2), SUBSTR(STR(?birth), 6, 2),
SUBSTR(STR(?birth), 9, 2)) != SUBSTR(?
taxID, 1, 6))
}
```

The rule is quite clear: it extracts the last two digits of year, the month, and the day from the date in YYYY-MM-DD format, concatenates these 6 digits, and compares the result with the first 6 digits of taxId. Since we are using inverse logic for the same reason as in the previous example, we assert that they don’t match.

This example demonstrates the usability of the “Reject properties” property of the shape. By default, this rule will

mark both taxId and birthDate values as invalid. However, if we trust the taxId property, it indicates that only birthDate value is invalid. We can set “Reject properties” = birthDate in our shape, ensuring that only this property value will be marked as invalid if the constraint fails.

Along with the SPARQL functions defined in the W3C specifications (with some limitations, see User Guide) you can use the DataVera EKG Platform-specific built-in function. An example:

```
ASK {
    $this <http://yourdomain.com/prefix/
taxId> ?taxid .
    $this <http://yourdomain.com/prefix/
hasResidencyStatus> ?status
    FILTER(STR(?status) == "NonResident"
&& DV_REPEATING_DIGITS_LEN(?taxid) > 11)
}
```

This rule checks that the taxId of non-residents contains no more than 11 repeating digits (it means that the taxId cannot be set to a fake value like 000000000000). The DV_REPEATING_DIGITS_LEN built-in function is used to obtain the length of the maximum repeating sequence within taxId value.

Another example checks that a document has a valid issue date between 1920 and today:

```
SELECT * WHERE {
    $this <http://yourdomain.com/prefix/
issueDate> ?date
    FILTER(DV_INVALID_DATE(?date) ==
'true' || ?date < '1920-12-31' || ?date
> NOW())
}
```

Note that the comparison operations – greater than and less than – are inverted in the filter conditions. We have to invert the entire Filter expression to use the “Inverse logic” flag.

NOW() is the SPARQL built-in function that returns the current date and time, while DV_INVALID_DATE is a DataVera-specific built-in that checks whether the given date has a valid format. The main built-in functions are listed in Table 5.

Table 5. **DataVera-specific SPARQL built-in functions**

Function	*Arguments*	*Description*
DV_REPEATING_SYMBOLS_LEN	?str – a string value to check	Return the length of the maximal sequence of repeating symbols
DV_REPEATING_DIGITS_LEN	?str – a string value to check	Return the length of the maximal sequence of repeating digits
DV_ALPHABET_MIX	?str – a string value to check	Returns true, if a string contains the mix of latin and non-latin characters
DV_INVALID_DATE	?date – a date to check	Returns true, if date has a valid format YYYY-MM-DD HH24:MI:SS
DV_ALLOWED_CHARS	?str – a string to check ?chars – a set of allowed characters	Returns true, if a string contains only the allowed characters. This function works with the multi-byte characters.

The rules can be used to prevent duplicates. For example, if we want to ensure that two customers do not share the same document number, the rule shall look as follows:

```
SELECT * WHERE {
    $this <http://yourdomain.com/prefix/documentNumber> ?number.
    $this <http://yourdomain.com/prefix/taxId> ?taxId.
    $this <http://datavera.kz/ekg/hasOrigin> ?origin.
    ?dupl <http://yourdomain.com/prefix/documentNumber> ?number.
    ?dupl rdf:type <http://yourdomain.com/prefix/DataSourceIdentifyingDocument>.
    ?dupl <http://datavera.kz/ekg/hasOrigin> ?origin.
    ?dupl <http://yourdomain.com/prefix/taxId> ?taxId
    FILTER($this != ?dupl)
    HINT(?number, "mandatory")
}
```

Note that in this rule the $this and ?dupl objects possess the same value of the taxId and hasOrigin properties, indicated by the same variables ?taxId and ?origin. This is one of the ways to assert the property values equality. The equality of the taxId property ensures that the ?dupl document is

possessed by the same customer as $this. Another way to assert this is to write the following lines: ?dupl <http://yourdomain.com/prefix/hasOwner> ?owner . $this http://yourdomain.com/prefix/hasOwner> ?owner (but it would not work if our task is to check the document numbers acquired from the different data sources, as the document from every data source would have its own owner).

As the ?dupl variable in this query contains the URI of another object, we have to identify its class – this is done with the ?dupl rdf:type <http://yourdomain.com/prefix/DataSourceIdentifyingDocument> expression.

The last line of expression uses the DataVera-specific extension. The HINT function tells engine that the ?number is the key variable. This optimizes the rule execution: the system will first look for the documents with the same number as $this has. This will give a small (or empty) set of objects, and limit the further calculations. Without the hint, the system could decide to search the documents with the same origin, which would give an order of magnitude larger data set.

The next rule checks that if the customer has a local phone number starting with 77 (the phone code of Kazakhstan), this customer's document must be either an IdentityCard, ResidentCard or BirthCertificate (these are the URIs of the DocumentKind class instances).

```
SELECT * WHERE {
    $this <http://xmlns.com/foaf/0.1/phone> ?phone .
    ?document rdf:type <http://yourdomain.com/prefix/DataSourceIdentifyingDocument> .
    ?document <http://yourdomain.com/prefix/hasOwner> $this
    { ?document <http://yourdomain.com/prefix/hasDocumentKind> <http://yourdomain.com/prefix/
Passport> }

    UNION { ?document <http://eubank.kz/ns/customer/hasDocumentKind> <http://yourdomain.com/
prefix/IdentityCard> }

    UNION { ?document <http://eubank.kz/ns/customer/hasDocumentKind> <http://yourdomain.com/
prefix/ResidentCard> }

    UNION { ?document <http://eubank.kz/ns/customer/hasDocumentKind> <http://yourdomain.com/
prefix/BirthCertificate> }

    FILTER(SUBSTR(?phone, 1, 2) != "77")
}
```

We have used a UNION clause to express the fact that the document kind must be equal to one of the URIs of enumerated members. The UNION clause can contain multiple patterns within.

Please note also that in this rule we are dealing with the properties of two objects of different classes. $this is a Person instance, while ?document represents a document.

When we have discussed the data model creation above, we have noted that it is possible to impose cardinality constraints on the property values. For example, we can state that taxId property is mandatory, and every object must have a value of it. But in some cases we can accept creating data objects not having a value of some property, but then highlight them with the warning-level violations.

This can be done by a rule like this:

```
ASK {
    OPTIONAL { $this <http://
yourdomain.com/prefix/taxId> ?taxId }
    FILTER(?taxId == "")
}
```

Please note that not all SPARQL features are supported by the platform at the moment. Refer to the EKG Platform User Guide to learn details.

After creating or editing constraints you have to restart the EKG Provider or clean its cache. If you need to re-apply the amended rules to the data objects of certain class, push the “sandwich” menu button over the list and choose “Apply constraints” item. This will initiate an asynchronous task. You can watch its progress using the progress indicator which will be displayed instead of the “sandwich” menu button.

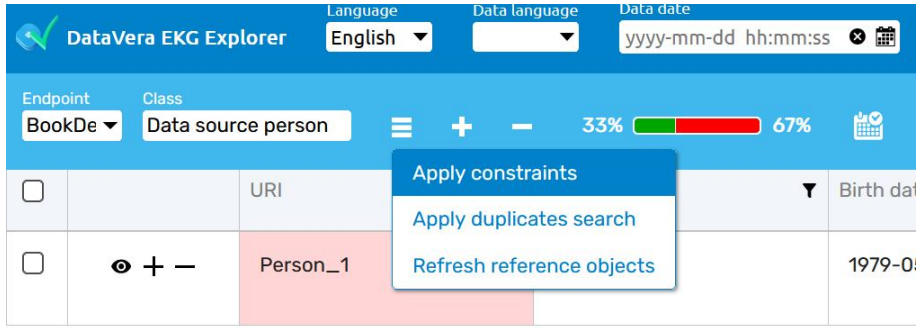


Fig. 29. Applying new constraints

Please note that it is necessary to map the shacl:ValidationResult class to the EKG storage (PostgresPostgreSQL table) to use this operation. The shacl:focusnode property must be mapped to a table column

5.3. Visualize violations and monitor data quality

When some constraint is violated, the shacl:ValidationResult object is created in the database. Each violation refers to a constraint violated and to an object which has caused it.

The constraints are applied automatically when a data object is updated. If there were previously the recorded violations of the constraints which now hold, the old violations are deleted. If the new violated constraints are found, the new violations are created. The shacl:ValidationResult may be observed as the data objects in the EKG Explorer UI. However, it is much more useful to see them in the list of the objects being checked. There are several ways of displaying violations. First, they are shown in the data objects grid: the cells containing invalid property values are highlighted red (for Violations) or yellow (for Warnings).

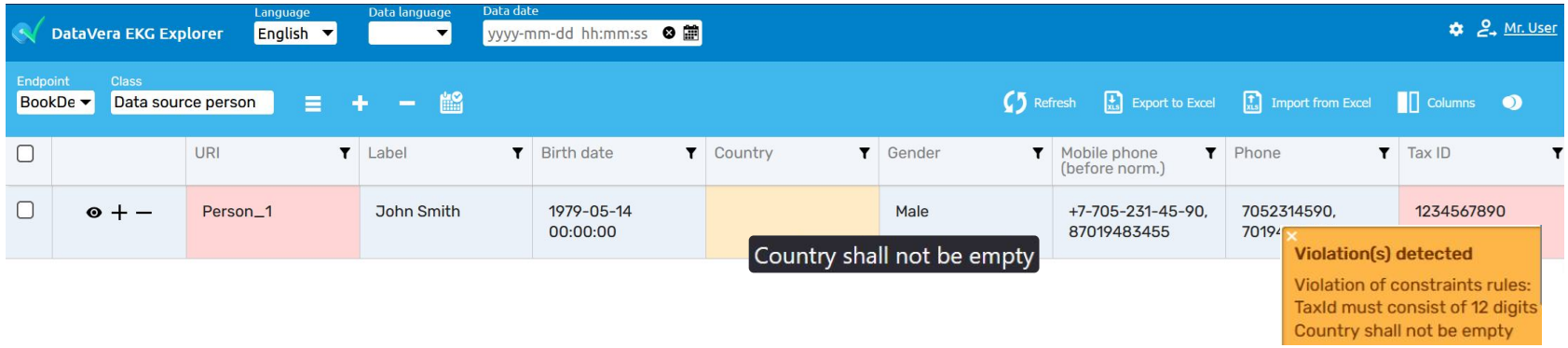


Fig. 30. Validation results display in the EKG Explorer UI

When you move the mouse over highlighted cell, the tooltip is displayed containing the constraint description. Hovering mouse over the URI cell will display all the violations, including those associated with invisible columns.

In the data object edit dialogue, the exclamation signs are displayed near the fields containing property values which violate constraints. Move the mouse over a sign to see the violation description.

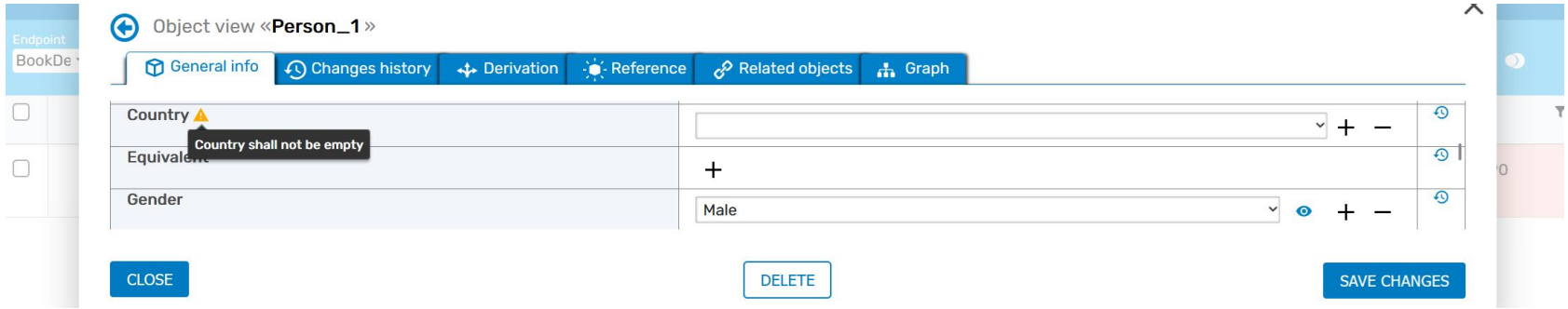


Fig. 31. Violations display in the data object editor box

The Data quality monitor tool is displayed over the data objects list. It shows the overall percentage of data objects having any associated validation error. Hovering mouse over this tool shows some details:

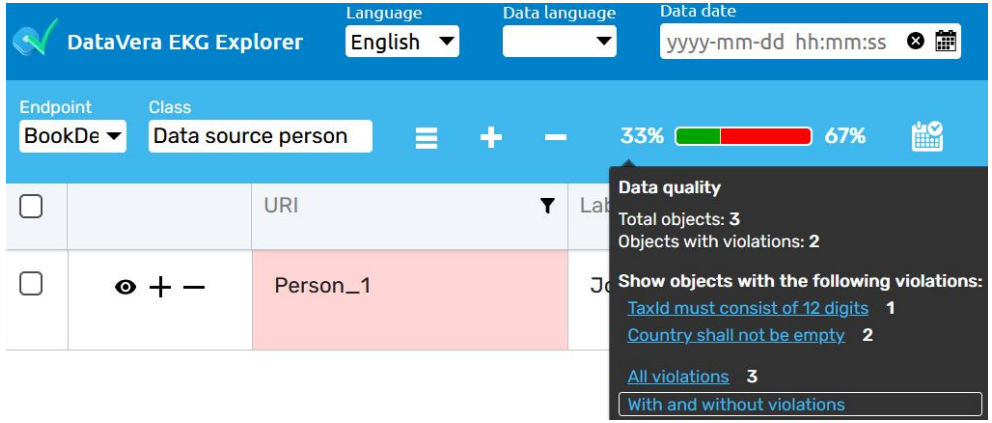


Fig. 32. Data quality monitor

Here you can see the total number of data records, the number of objects having at least one violation, and the violated constraints list with the number of violations of each of them. Clicking on the constraint name you can filter list to display only the data objects which have violated this constraint.

You can navigate to the Data Quality Dashboard to see more detailed metrics. This dashboard contains several widgets shown in the *Figures 33 – 35*.

The diagram in the top left part of the dashboard shows the percentage and amount of data objects having blocking and informative constraint violations. You can choose a particular data source to see this statistic only for data source objects originated from this system. The diagram at the right indicates the consolidation results: how many data source objects were bound with the other system's data source objects, how many of them have produced reference objects or caused a consolidation error.

The second row of the diagram shows the reference object distribution by the number of data source objects used to produce them.

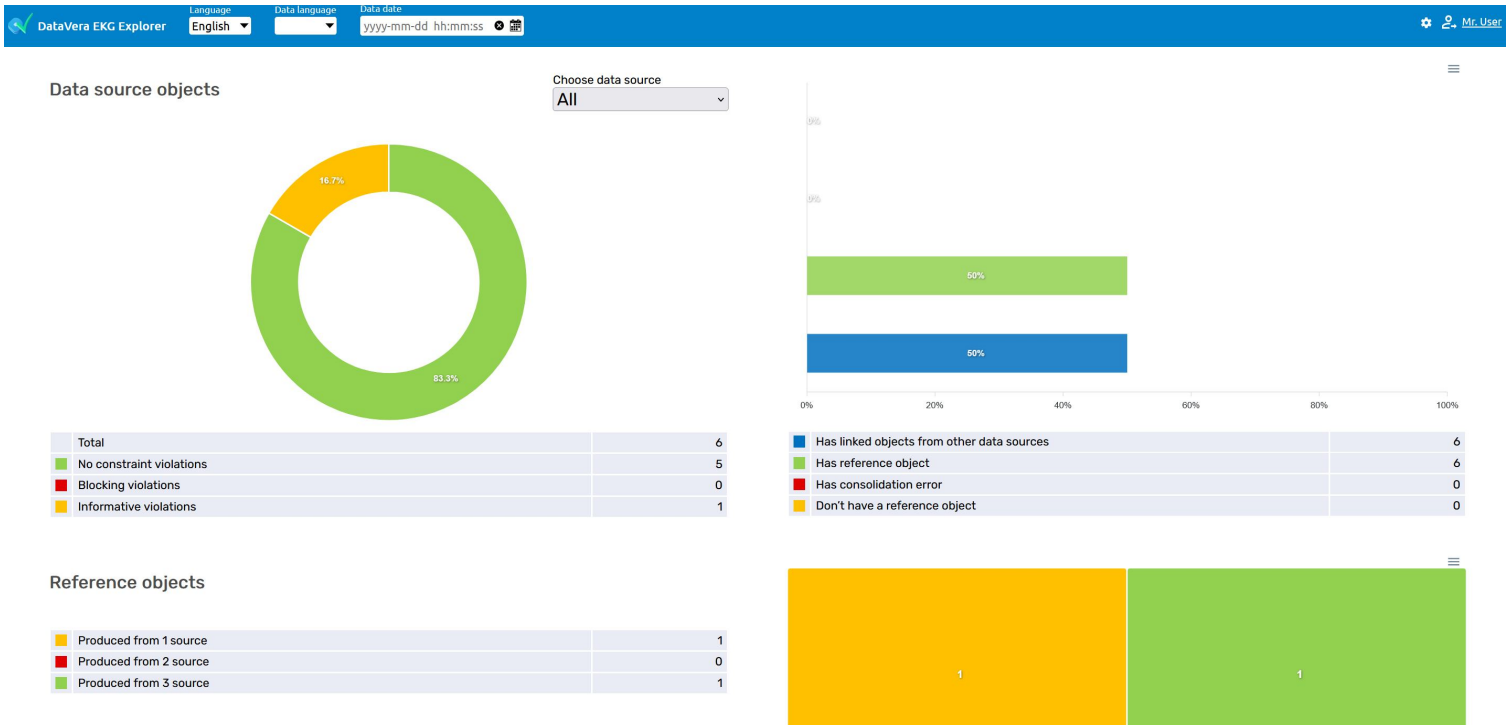


Fig. 33. Data Quality Dashboard, upper part

Fig. 34. Data Quality Dashboard, middle part

In the middle part of the dashboard, the widget shows data quality details for the properties applicable to the chosen class. The left part shows how many data source objects has a non-empty value of every property. The right part shows which share of the property values has successfully passed logical control, or has the blocking and informative violations.

Fig. 35. Data Quality Dashboard, lower part

The lower part of the dashboard shows the properties statistics for the reference objects. The left part shows how many reference objects have non-empty values of each property. The right part displays the share of the data source systems from which the values of each property was taken.

5.4. Use control ratios in constraints

The above constraint examples were focused on the logical control of the certain property values. The constraints may also be used to calculate the control ratios between the numeric property values.

The arithmetic operations right in the FILTER clause are not supported by the platform at the moment. Instead, you shall create a SPARQL function which performs some calculation, and then use its result to compare with desired values.

Consider an example. Let us add a “Deposit” class into our model, having the “Amount”, “Term”, “Rate” and “Income due” properties. We want to check by constraint that Amount * Term * Rate / (365 * 100) is equal to Income due.

First, we have to create three SPARQL function parameters (see “Parameter” class in the “SHACL” branch of the model) with the URIs: daysParam, rateParam and amountParam. rateParam and amountParam must have shacl:datatype property value equal to xsd:double, daysParam shacl:datatype equal to xsd:integer. The special shacl:path property must contain a synthetic URI having http://datavera.kz/ekg/ prefix. The unique part of this URI represents the name of function parameter which will be used in its body. Assign our parameters shacl:path values equal to http://datavera.kz/ekg/days, http://datavera.kz/ekg/rate and http://datavera.kz/ekg/amount.

Second, we must create a SHACL function instance. Open this class in the tree and create a function with URI and name equal to depositIncome. Select all three parameters as the shacl:parameter property values, and assign “return type” equal to xsd:double. Finally, set the “select” property equal to the SPARQL query:

```
SELECT ( $amount*$days*$rate/(365*100)
AS ?result ) WHERE {}
```

This query has a special format which must be followed. Note the three variables used in the formula: $amount, $days and $rate. Their names match the defined function parameters.

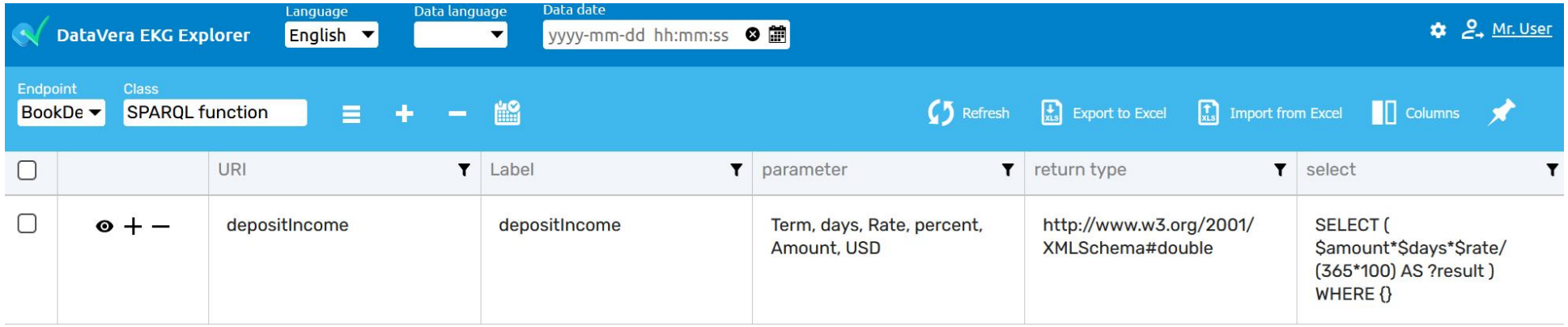


Fig. 36. DepositIncome SPARQL function in the EKG user interface

Now let us create a constraint. It shall be created in the way described above. The constraint body shall be:

```
SELECT * WHERE {  ?dep rdf:type <http://yourdomain.com/prefix/Deposit> .
                  ?dep <http://yourdomain.com/prefix/termDays> ?term .
                  ?dep <http://yourdomain.com/prefix/amount> ?amount .
                  ?dep <http://yourdomain.com/prefix/rate> ?rate .
                  ?dep <http://yourdomain.com/prefix/incomeDue> ?income
                  BIND ( depositIncome( ?amount, ?term, ?rate ) as ?calculated_income )
                  FILTER( ?income != ROUND(?calculated_income, 2) )
}
```

The query extracts three parameters required to calculate the deposit income: the deposit amount, term and rate. Then the depositIncome SPARQL function is called, using these variables as parameters. The function result is saved to the ?calculated_income variable. Finally, this variable is compared with the incomeDue parameter stored in the Deposit class individual's properties. Let us see the violation in the EKG Explorer interface – please note that all the properties involved in calculation are highlighted in red, as any of these values may be incorrect:

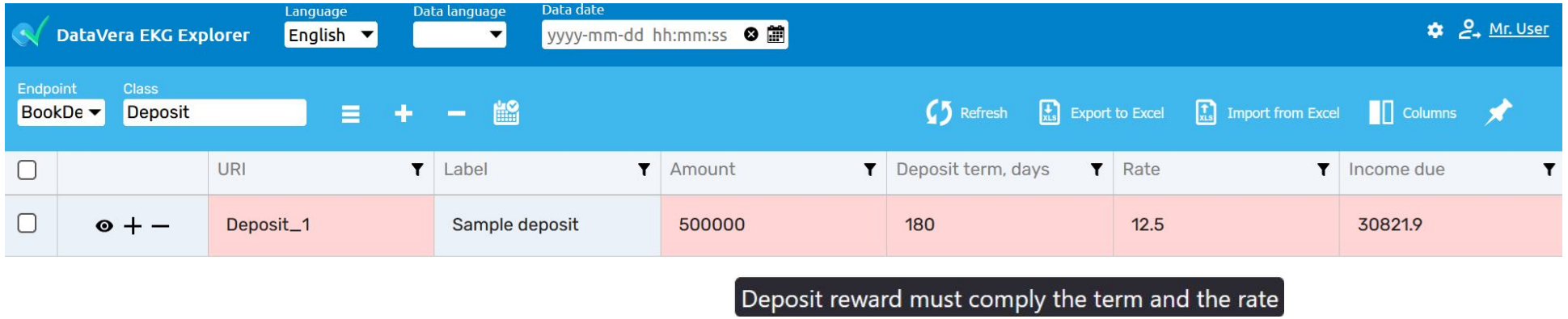


Fig. 37. Constraint violation in the EKG Explorer interface

6. Create consolidation rules

6.1. Duplicates search and markup

The next data processing stage is consolidation. It consists of two steps, the first of which is the duplicates search. As each business object (i.e., a customer, document, address, etc.) can be represented by several records in the various data sources, we first need to bind together these records, which we call duplicates. At the second step, one reference object will be generated from every group of the data source records.

To mark up duplicates, the SHACL rules are used. SHACL rules are similar to the above-described constraints, but with the significant difference: they produce new facts recorded in the database. SPARQL CONSTRUCT query is used in the rule body. The queries of this kind consist of two parts: the WHERE clause defines a pattern to search for data, while the CONSTRUCT part specifies the new inferred facts which will be inserted into database. When using rules for duplicates search, the CONSTRUCT part infers relation between the duplicates, while the WHERE part specifies duplicates search criteria. The query for our sample looks as follows:

```
CONSTRUCT {
    $this owl:sameAs ?pair.
    ?pair owl:sameAs $this

} WHERE {
    $this rdf:type <http://
yourdomain.com/prefix/DataSourcePerson>.
    $this <http://yourdomain.com/prefix/
taxId> ?taxID.
    ?pair rdf:type <http://
yourdomain.com/prefix/DataSourcePerson>.
    ?pair <http://yourdomain.com/prefix/
taxId> ?taxID
    FILTER ( $this != ?pair )
}
```

The WHERE part produces the pairs of $this and ?pair objects which represent persons with the same taxId. The CONSTRUCT part infers owl:sameAs relation between these objects. For other object classes, the CONSTRUCT part usually stays the same, while the WHERE part can contain various conditions similar to the those discussed above.

Please note that the owl:sameAs property (its full URI is http://www.w3.org/2002/07/owl#sameAs) must be mapped to the EKG storage column having array type. The properties used as the duplicates search criteria (taxId in our case) also must be mapped to the separate columns. In our example it means that the “entity” table must have the “sameas” column having _varchar type (we have done this mapping in the first part of this Guide), and the “taxid” varchar column. You shall run the following command at the system configuration stage:

```
ekg-provider map PG_Entity taxid taxId 0
```

It’s worth creating btree index on taxid column.

We have to create the SPARQL Construct rule individual which has no other properties than the above query in the “construct” field. Then switch to the “Node shape” class and create an individual which links the rule to the “Data source person” class. Reload server cache, go to the “Data source person” class page, and click on the “Apply duplicates search” menu item. After the rule execution you will see the inferred facts in the object properties, as shown in *Fig. 38:*

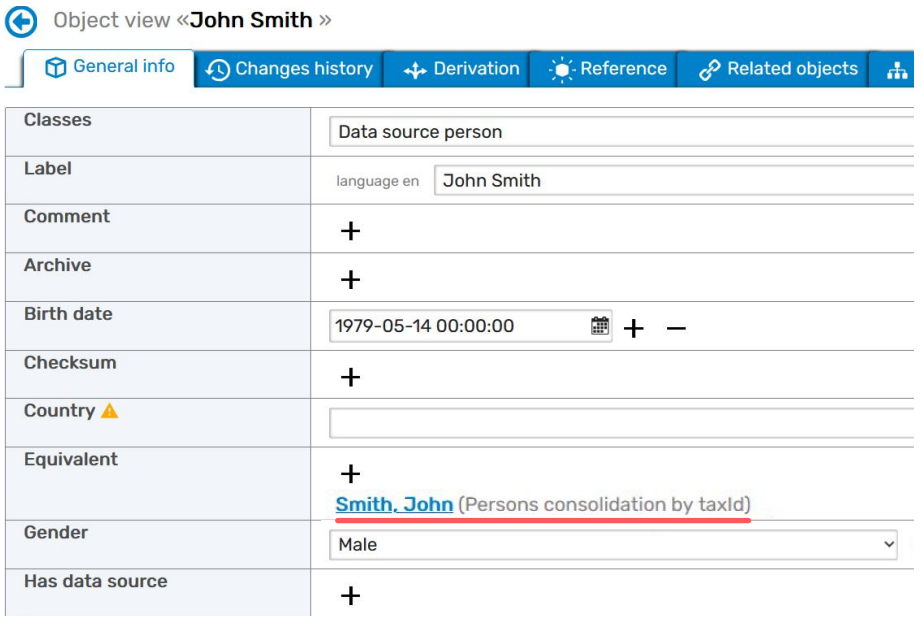


Fig. 38. Inferred facts in the object properties

You can use this kind of rules for any data augmentation tasks. One of the examples is the new relations inference according to the graph chain relation search patterns, such as grouping affiliated persons or companies. The objects can be assigned with the classification properties basing on their relations or literal property values.

6.2. Consolidation rules

Now we need to create the Consolidation rule which will define the reference objects generation logic. Open the “Consolidation rules” class and create an individual with the following property values:

- Mapped class = Data source person. It is the class of the records loaded from data source.
- Reference class = Reference person
- Has mandatory property = taxId. You can select more than one property here. The reference objects will not be created without having the values of these properties.
- Create reference objects from single data objects = Yes. In some special cases this switch may be turned off, to prevent creating reference objects from the records existing in only one data source.

The remaining properties choose the logic of selecting property values for the reference object:

- Prefer newer records = No. If set to Yes, the system will take the most recent record from the group of consolidated objects and copy its properties to the reference object. Then it will take the second-newest object and use it to fill the properties which values are not yet determined, etc. The special “Last change date” (http://datavera.kz/ekg/lastChanged) property is used to determine the data source object update time. You shall

map it to the timestamp or date column of the data source object to use this logic.

- Prefer newer values = Yes. If this option is selected, the system will determine the latest source for each property value. The property value update time is determined by the EKG Platform using its history records. It means that this option has no effect at the initial data load and consolidation, but can be used in the continuous data update process.
- Use origin priority rules = No. If set to Yes, the system will use the “Consolidation priority map” class individuals to determine the preferred data source for each property value. These individuals shall have the following properties:

  – Mapped class = the data source objects class, “Data source person” in our case

  – Mapped property = the property for which the priority is defined, for example “Birth place”

  – Data source = the data source, for example DataSource_Sample

  – Processing priority = a number defining the priority of this data source for the defined class and property. The 0 value indicates the highest priority

  – Mandatory = Yes, if this property value is mandatory at consolidation.

If neither of these options is set, the system will rely on the “Data quality” property of the data source to determine the preferred data source for each record. In the rare case when the reference object was previously created and has some property value, but then all the data source objects are cleared from this property value so the new consolidation is impossible, the existing reference property value will be kept at consolidation.

The property values violating the “Violation” level constraints are ignored at consolidation. The system acts as if these property values are not set.

The EKG Provider restart is required after consolidation rules editing to apply changes.

7. Build the reference data set and check data provenance

We have learned the following types of the rules used by the platform:

- Data source mapping rules allow loading data from external sources
- Normalization rules automatically correct incoming data
- Validation rules are marking incorrect property values
- Duplicates search rules are forming the groups of data records corresponding to the same business object
- Consolidation rules define the logic of the reference objects creation.

All these rules are applied by default when a data object is created or updated in the platform. However, in the bulk data load mode the rules of the last three types are skipped to speed up data loading. This affects both the initial data loading using the scripts, which we discuss below, and the data import from Excel. After populating the platform with data using these ways, you have to apply rules manually using the “Apply constraints”, “Apply duplicates search” and “Refresh reference objects” items of the data class menu. These operations are asynchronous and may take a long time on the large data sets.

When importing data from Excel file, you can check “apply rules” checkbox. This will cause the platform to apply all kinds of rules instantly, but will significantly slow down the loading process.

7. Build the reference data set and check data provenance

After all the rules are constructed, you have to run the initial data loading from data sources. This process has to be run separately for each class and data source. There is a script in the ekg-worker container that can be used to perform this task. Its command line syntax is:

```
php load_data.php [data source] [class
URI or ClassMap individual URI]
[property URI or "all"] ["test" if you
want to test the rule]
```

The last two parameters may be omitted.

Let us run this command for the DataSource_Sample source and DataSourcePerson class:

```
kubectl exec [ekg-worker pod id] php
load_data.php DataSource_Sample
DataSourcePerson all test
```

You can see in the output the resulting SQL query to the data source. If this query produces an error, it will be displayed in the log. In our case the query was successful. Then the "test" parameters cause our script to extract one record from the data source and display its properties processed with the mapping rules, as shown in the end of the output. Here you can see if the mapping rules are set up correctly. Pay attention to the list of properties, check out their values. In our case we have to ensure that the ObjectMaps are applied correctly to obtain the right values of the hasGender and hasCountry properties, as shown in *Fig. 34*.

When everything is fine, you can run the data population commands. If we have two data sources having DataSource_Sample and DataSource_Example URIs, we have to run the following commands to populate the DataSourcePerson class:

```
[serge@datavera datacons]$ php load_data.php DataSource_Sample DataSourcePerson all test
{"time":1727351498,"readable_time":"2024-09-26 11:51:38", "pid":"99517", "message":"Loading model and mapping rules"}
{"time":1727351501,"readable_time":"2024-09-26 11:51:41", "pid":"99517", "message":"Run parameters:
        Data source             DataSource_Sample
        Class or ClassMap       DataSourcePerson
        Property:               all
        Identifiers:            test
"}
{"time":1727351501,"readable_time":"2024-09-26 11:51:41", "pid":"99517", "message":"
Class DataSourcePerson:
SELECT PUBLIC.CUSTOMERS.tax COL_0, PUBLIC.CUSTOMERS.gender COL_1, PUBLIC.CUSTOMERS.country COL_2, PUBLIC.CUSTOMERS.name
 COL_3, PUBLIC.CUSTOMERS.id COL_4, PUBLIC.PRODUCT.name COL_5
FROM PUBLIC.CUSTOMERS
LEFT JOIN PUBLIC.PRODUCTS_CUSTOMERS ON customers.id = products_customers.customer
LEFT JOIN PUBLIC.PRODUCT ON products_customers.product_id = product.id
WHERE kind = 1
ORDER BY PUBLIC.CUSTOMERS.id"}
{"time":1727351501,"readable_time":"2024-09-26 11:51:41", "pid":"99517", "message":"Requesting data objects of DataSour
cePerson class from DataSource_Sample data source: 0.005 sec"}
{"time":1727351501,"readable_time":"2024-09-26 11:51:41", "pid":"99517", "message":"Data object sample:
URI = DataSource_Sample_DataSourcePerson_1
        taxId                                   123456789012
        http://www.w3.org/2000/01/rdf-schema#la John Smith
        http://datavera.kz/ekg/hasOriginCode    1
        http://datavera.kz/ekg/productOfInteres Loan
        http://datavera.kz/ekg/lastChanged      2024-09-26 11:51:41
        http://datavera.kz/ekg/checksum         ffe4f2e12651ef4c9009c7dc4795016c
        hasOrigin                               DataSource_Sample
        hasGender                               Male
        hasCountry                              Kazakhstan
        http://datavera.kz/ekg/hasReferenceObje 844f71af-c805-4d80-b6db-570d07bbebc8
"}
```

Fig. 39. Testing data source connection

When everything is fine, you can run the data population commands. If we have two data sources having DataSource_Sample and DataSource_Example URIs, we have to run the following commands to populate the DataSourcePerson class.

You can run both commands simultaneously using nohup.

```
kubectl exec [ekg-worker pod id] php
load_data.php DataSource_Sample DataSourcePerson

kubectl exec [ekg-worker pod id] php
load_data.php DataSource_Example DataSourcePerson
```

7. Build the reference data set and check data provenance

The data loading takes a long time, it can be several hours or days for large tables having millions of records. The scripts output logs which you can use to monitor the data loading process. The data loading script output looks as follows – please note the processing time and created objects quantity displayed in the end:

The data loading takes a long time, it can be several hours or days for large tables having millions of records. The scripts output logs which you can use to monitor the data loading process. The data loading script output looks as follows – please note the processing time and created objects quantity displayed in the end:

```
{"time":1727350564,"readable_time":"2024-09-26 11:36:04", "pid":"91707", "message":"
Class DataSourcePerson:
SELECT PUBLIC.CUSTOMERS.tax COL_0, PUBLIC.CUSTOMERS.gender COL_1, PUBLIC.CUSTOMERS.country COL_2, PUBLIC.CUSTOMERS.name
 COL_3, PUBLIC.CUSTOMERS.id COL_4, PUBLIC.PRODUCT.name COL_5
FROM PUBLIC.CUSTOMERS
LEFT JOIN PUBLIC.PRODUCTS_CUSTOMERS ON customers.id = products_customers.customer
LEFT JOIN PUBLIC.PRODUCT ON products_customers.product_id = product.id
WHERE kind = 1
ORDER BY PUBLIC.CUSTOMERS.id"}
{"time":1727350564,"readable_time":"2024-09-26 11:36:04", "pid":"91707", "message":"Requesting data objects of DataSour
cePerson class from DataSource_Sample data source: 0.005 sec"}
{"time":1727350565,"readable_time":"2024-09-26 11:36:05", "pid":"91707", "message":"Total pg:       0 ekg:       0 php:
   0.372 sec"}
{"time":1727350565,"readable_time":"2024-09-26 11:36:05", "pid":"91707", "message":"3 objects processed"}
```

Fig. 40. Data loading log

The load_data.php script applies data validation and consolidation rules after writing each data object. It is relatively slow, but does not require further processing. As an alternative, you can use load_data_bulk.php script, having the same [data source] [class URI or ClassMap individual URI] command line parameters. It always rewrites the whole objects and does not have test mode. This script uses bulk load functionality to speed up data loading. But in the bulk load mode the rules are not applied instantly, except for normalization rules. After the data is loaded, you can use

7. Build the reference data set and check data provenance

“Apply constraints”, “Apply duplicates search” and “Refresh reference objects” items of the data class menu to consequentially apply the validation, duplicates search and consolidation rules to the loaded data. These rules are applicable to the whole classes. There are also the command line tools which gives more control on this process. In general, bulk loading data and further running the above-listed asynchronous processing procedures is faster than the data loading with instant rules execution.

When the consolidation is done, you can observe the reference data set in the appropriate class, the “Reference person” in our case. Click on any of the reference objects to open the properties dialogue, and switch to the “Provenance” tab.

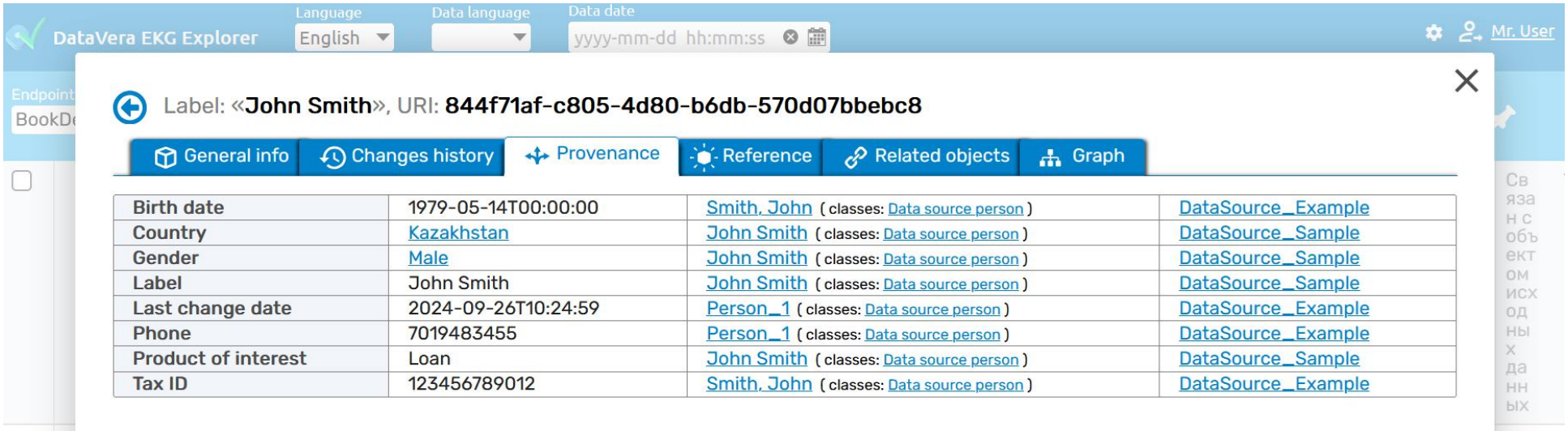


Fig. 41. Reference object provenance

7. Build the reference data set and check data provenance

Looking at the data source object, you can navigate to the “Reference” tab. It shows the linked duplicates and a reference object link, if one was created. If there was a consolidation error, you will see the error information instead:

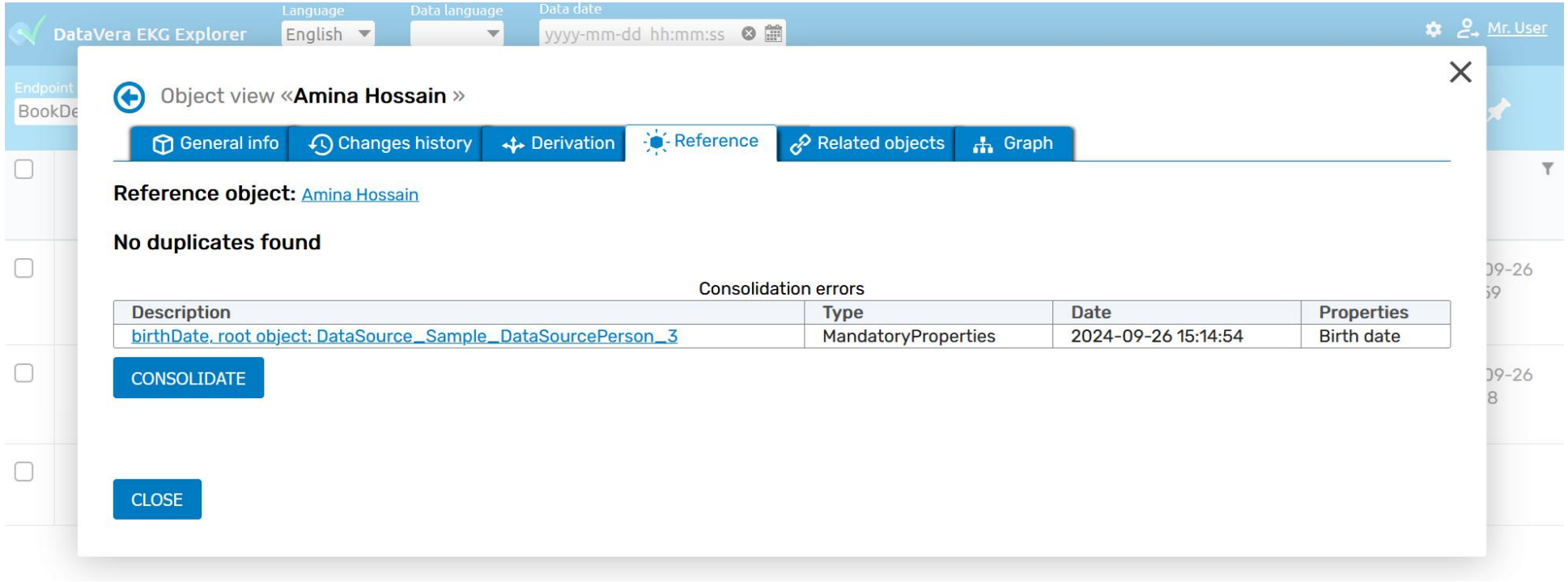


Fig. 42. Consolidation error

The error shown in *Fig. 42* indicates that there is no value for birthDate, a mandatory property. The other reasons may include the missing key property values, duplicates set inconsistency (if the objects from a group of duplicates refer to the different reference objects – this may be caused by the key property values change), etc. The consolidation errors may be also reviewed as the “Consolidation error” class individuals. The good practice is that the data management personnel review this class daily to find out and correct the new errors. The data source objects can be corrected in the EKG as well as in the data sources – in both cases, the new consolidation attempt will occur. The old consolidation error will be deleted, and the reference object (or a new consolidation error) created.

If the reference object exists, but then the data source objects are amended in the way that prevents new successful consolidation, the existing reference object will be marked with “Archive” = true flag. In this case, on the “Reference” tab you will see both the reference object link and the consolidation error information.

The reference objects merging is a rare task which requires manual actions. In this case, you shall delete one of the reference objects, and clear the links pointing to it from the data source objects. Then click “Consolidate” button on the “Reference” tab of one of the data source objects. The rules will be applied to bind together the duplicate objects having the same key property values, and an existing reference object (the one which was not deleted) will be updated accordingly. The data source objects will have a link to this reference object.

8. Set up and monitor continuous data update

When the initial reference data set is created, you have to set up its continuous update. This work is done by the load_data.php script which runs as a default process of the ekg-worker container. At start, this script reads all the configured mapping rules and starts a separate process for acquiring data objects of each class from each source, so the total number of processes is approximately equal to [number of classes] x [number of data sources]. The parent process observes the children and restarts them if needed. This process does not require manual management; however, you can see ekg-worker pod logs to check out that the data update process is going on normally.

The data loading process requires that a timestamp or datetime column exist in every monitored table (or a table joined with it, such as an actions log or history table), containing the last update date for each record. These columns may be populated by the data source system itself, by a mechanism like CDC, or using a trigger that set this column equal to NOW() on every create or update event. The ekg-worker remembers the timestamp of the last processed record for every ClassMap – you can see this value in the properties of any ClassMap individual. Every N second ekg-worker performs a SQL query to obtain the rows which timestamp is more than the last processed record update time. If there are several joined tables, it is possible to have a separate timestamp in each of them: in this case, the maximum timestamp value will be used. The ekg-worker process forms a data object to be written to EKG, and compares its checksum with the already existing object (if it exists). If a checksum does not match, it writes a data object to EKG, which causes all the rules to be applied. As a result, the reference object related with the updated data object is created or updated, or a consolidation error is be recorded.

In case of the massive data changes in the data source, the changes will be processed in the temporal order. An update queue may emerge: the objects which were updated later shall wait while the earlier modified objects are processed.

8. Set up and monitor continuous data update

There is a special script to monitor the queue status:

```
kubectl exec [ekg-worker pod id] php sync_status.php
```

```
[serge@datavera datacons]$ php sync_status.php
DataSourcePerson
        Sample:         2       2024-09-25 15:18:00          0  ClassMap_sample_customers
```

Fig. 43. Data source synchronization monitoring script

The script outputs the list of the data object classes, and data sources configured for each class. It displays the approximate number of records waiting to be processed ("2" at the *Fig. 40*), the last successful object update time, the data source query execution time ("0") and the ClassMap individual URI. If you see a big objects queue, you have to check the log for the errors.

If you have several related classes, for example the Persons and their Documents, you shall keep in mind that creation of the reference Document requires that a corresponding reference Person already exists. So, when the Person and the Document in the source system are updated simultaneously, they can be pushed to EKG in the wrong order. The ekg-worker process handles this situation, but by default it allows only 2 minutes gap between creation of the reference Person and Document. If in your environment the data is updating intensively, and a processing queue of the Person objects may grow, leading to the higher gap, you shall tune the synchronization script with the help of DataVera support team.

When deploying the DataVera EKG platform cluster, it is recommended to set up logs collection from all pods to ELK.

8. Set up and monitor continuous data update

You can also set up EKG Provider metrics collection to Prometheus. Metrics visualization dashboards can be set up in Grafana. The metrics reflect API requests per second, total number of requests, average requests execution time and errors counter. As all the EKG Platform components (including EKG Explorer and ekg-worker data adapter) are working with EKG Provider by querying its API, this reflects the overall platform load.

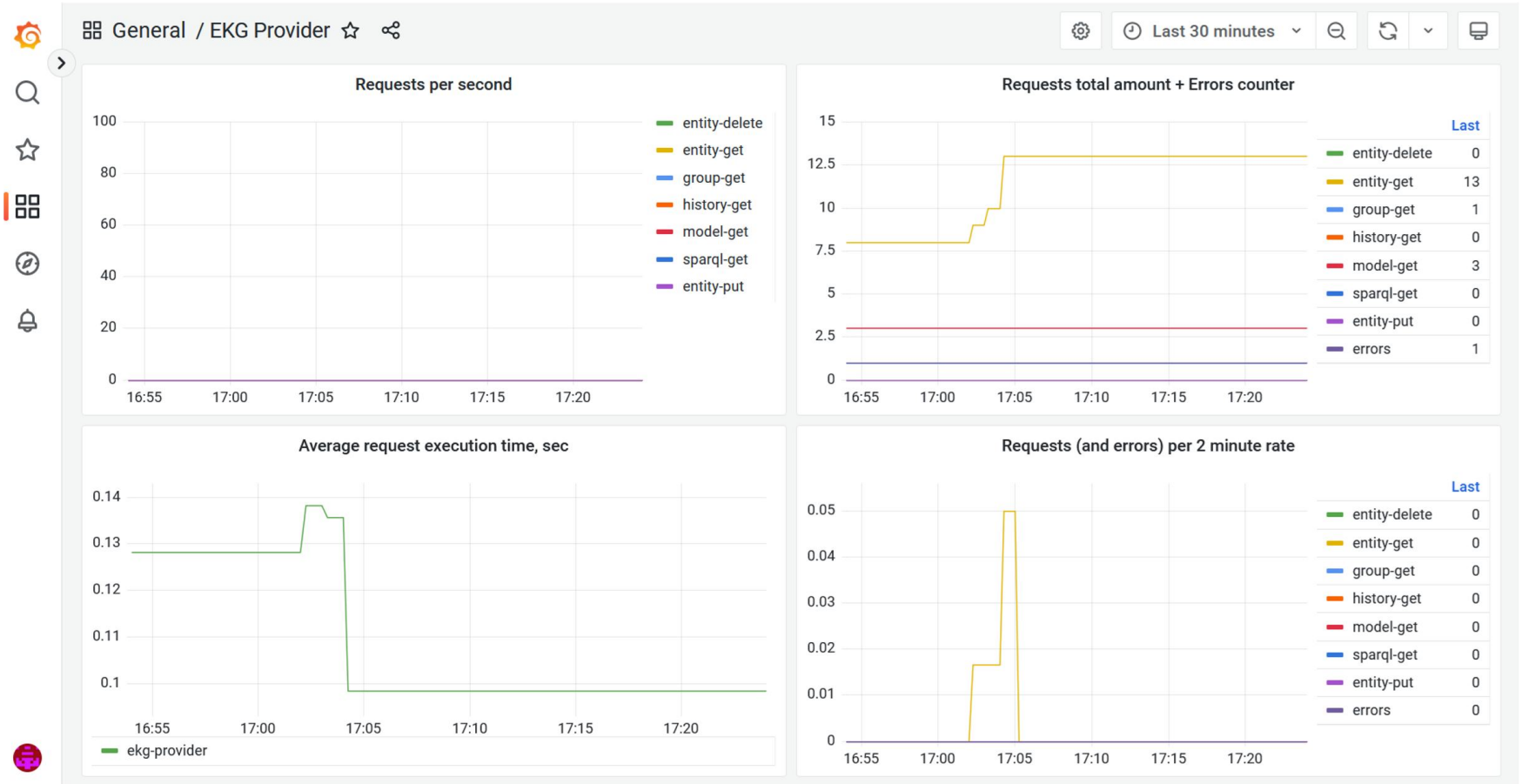


Fig. 44. An example of EKG Provider metrics visualization using Grafana

9. Consume reference data

There are several ways of consuming reference data from EKG. You can query them using REST API which is described in details in the EKG Platform User Guide. For example, you can get all the reference persons who are residents of Kazakhstan:

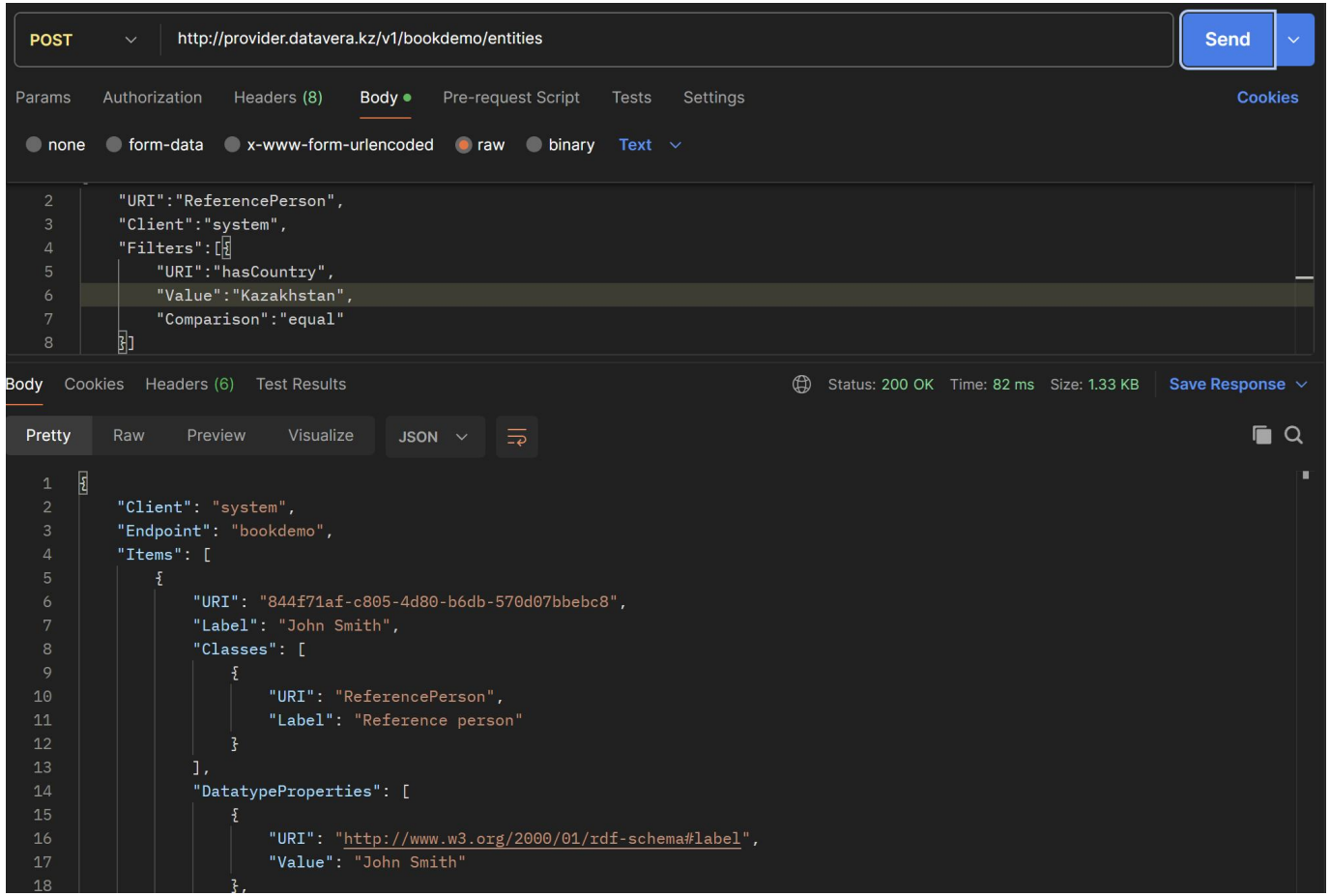


Fig. 45. Using EKG Platform REST API

9. Consume reference data

You can also obtain any metadata (provenance, values history, the rules of all sorts, data quality violations, consolidation errors, etc.) and even the data model using this API. A good idea may be to export some data or metadata to the BI tools. You can also populate the platform with data directly using the API. Subqueries and aggregation functions are available when using data extraction methods.

But for the near real-time data synchronization between EKG and data sources you shall use the subscription mechanism. A data consuming component (an adapter intended for reference data propagation from EKG to the business applications) shall subscribe using the API to obtain a continuous flow of updated data objects through the Kafka topic. This adapter can transform data to the structure appropriate for the business application using the same mapping rules (ClassMap, PredicateMap and ObjectMap), or a separate set of rules which may be created for the outgoing data transformation. You can define the new types of the data receivers and represent them as the DataSource class instances. This gives you the total control over the data integration from one point – the DataVera Platform.

We have a template data adapter script which writes the data objects obtained from Kafka topic to the arbitrary relational databases using the rules configured in the model. This adapter is available by request.

10. Performance tuning tips

EKG Platform performance highly depends on the PostgreSQL cluster performance. The tips below will help you to tune the storages configuration, and then assess the overall platform configuration to maximize performance.

1. Carefully design the number of storages (PostgreSQL tables mapped to the data model classes). We recommend to group the classes having similar properties into the one storage. However, the classes with the very big number of records should not be placed into the same storage with the relatively small classes. You can place the objects from data sources into the same storage with the reference objects. By the other hand, creating too much storages can make slower some operations: when searching an object by URI, the system not always can determine its storage, and in some cases, it will scan all of them.
2. You can use several PostgreSQL clusters with one EKG Platform instance. Sometimes it is useful to use a PostgreSQL cluster to store the reference data and a standalone PostgreSQL for storing logs, namely the instances of ekg:AuditLog, ekg:TechnicalError, prov:Derivation classes. The ekg:MappingRules class probably shall be mapped to this storage too. Please not that the shacl:ValidationResult class must reside in the main PostgreSQL cluster, as the platform joins its table with the data source objects tables in some queries.
3. Manage the properties to columns mapping. Placing the property values into a separate column worth when this property has the relatively high variety of values, and is often used in the SELECT queries. In this case you shall use the btree index (or gin, for arrays) on this column. The owl:sameAs and all the key properties used at consolidation (such as taxId) shall be placed to the separate column in almost all cases.
4. Place shacl:ValidationResult and prov:Derivation into the separate tables. Map shacl:focusNode, shacl:sourceShape, ekg:focusClass and prov:hasEntity to their columns.

10. Performance tuning tips

5. When writing code using EKG Platform API, use the right request flags. If you don’t need the referenced object names in the response, use "NoLabels": true – this will dramatically reduce execution time. Use “URIonly”: true flag when you need to check that object exists. Set "CheckStorages": false in the select and update queries when you are sure that the requested objects storage will never change, and the system shall not check the other storages if the requested or changed objects exist. Use "NoPropagate": true, "NoRefCheck": true when possible in the update requests. Use "NoHistory": true if you don’t need to keep the history for this class objects.