REPORT ON DELIVERABLE D1.4

Final SLIPO System

SLIPO

PROJECT NUMBER: 731581
START DATE OF PROJECT: 01/01/2017
DURATION: 36 months

Horizon 2020

SLIPO is a research project funded by European Commission's Horizon 2020 Programme.

Dissemination Level	Public
Due Date of Deliverable	Month 36, 31/12/2019
Actual Submission Date	20/12/2019
Work Package	WP1 - Requirements, Architecture and Integration
Tasks	1.4 Integration
Type	Demonstrator
Approval Status	Submitted for approval
Version	1.0
Number of Pages	112
Filename	D1.4_Final_SLIPO_System.pdf

Abstract

This report presents an overview of Deliverable D1.4 "Final SLIPO system", i.e., the final version of the SLIPO system, which integrates the latest versions of the SLIPO Toolkit components, as well as manifold new features and improvements resulting from our pilot. First, we provide an overview of the final architecture of the SLIPO system, its subsystems, libraries and external frameworks, as well the characteristics of its deployment. Next, we present the operation of the SLIPO system, its various integrated functionalities, and examples of select POI integration workflows. Finally, we present a benchmark of the system demonstrating its successful and scalable operation on *world-scale* Big POI data assets.

History

version	date	reason	revised by
0.1	30/08/2019	First draft	Spiros Athanasiou
0.2	25/09/2019	Added content to various Sections	Yannis Kouvaras, Michalis Alexakis, Kostas Patroumpas, Dimitris Skoutas, Nikos Karagiannakis
0.5	01/10/2019	Multiple revisions and restructuring	Spiros Athanasiou
0.6	08/11/2019	Added content to various Sections	Yannis Kouvaras, Michalis Alexakis, Nikos Karagiannakis, Mohamed Sherif, Damien Graux
0.7	15/11/2019	Multiple revisions and improvements	Yannis Kouvaras, Spiros Athanasiou
0.9	30/11/2019	Updated benchmarking section	Yannis Kouvaras, Michalis Alexakis, Kostas Patroumpas
0.9	20/12/2019	Internal Review	Kostas Patroumpas, Dimitris Skoutas
1.0	22/12/2019	Final version	Spiros Athanasiou

Author list

organization	name	contact information
ATHENA RC	Spiros Athanasiou	spathan@imis.athena-innovation.gr
ATHENA RC	Dimitris Skoutas	dskoutas@imis.athena-innovation.gr
ATHENA RC	Kostas Patroumpas	kpatro@imis.athena-innovation.gr
ATHENA RC	Yannis Kouvaras	jkouvar@imis.athena-innovation.gr
ATHENA RC	Michalis Alexakis	malex@imis.athena-innovation.gr
ATHENA RC	Nikos Karagiannakis	nkara@imis.athena-innovation.gr
IAIS	Damien Graux	damien.graux@iais.fraunhofer.de
IAIS	Simon Scerri	simon.scerri@iais.fraunhofer.de
InfAI	Mohamed Sherif	sherif@informatik.uni-leipzig.de
InfAI	Kevin Dreßler	dressler@informatik.uni-leipzig.de

Executive Summary

This report presents an overview of Deliverable D1.4 "Final SLIPO system", i.e., the final version of the SLIPO system, which integrates the latest versions of the SLIPO Toolkit components, as well as manifold new features and improvements resulting from our pilot activities.

In Section 1, we present final version of SLIPO system, as it is currently deployed in a production setting. First, we provide an overview for its architecture and detail its major components. Next, we present the production system applied to deploy SLIPO, briefly presenting its capabilities and characteristics, as well as its initialization. Finally, we present how documentation is produced and provide links to the actual documentation of the system.

In Section 2, we present the support key roles and responsibilities of end-users of the SLIPO system, present a concise walkthrough of its entire operation and its various integrated functionalities, and explore examples of select POI data integration workflows.

In Section 3, we present a benchmark of the SLIPO system for an indicative data integration use case demonstrating its successful and scalable operation on *world-scale* Big POI data assets (~92M POIs). This challenging scenario surpasses the *pragmatic* limits of POI data integration and validates the satisfaction of the scalability targets established before the start of the project.

Abbreviations and Acronyms

API	Application Programming Interface
AJAX	Asynchronous JavaScript and XML
CI	Continuous Integration
CORS	Cross-Origin Resource Sharing
CRON	Command Run On
CSRF	Cross-Site Request Forgery
CSS	Cascading Style Sheets
CSV	Comma Separated Values
DI	Dependency Injection
DOM	Document Object Model
ETL	Extract, Transform and Load
HDFS	Hadoop Distributed File System
HTTP	Hypertext Transfer Protocol
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
Java EE	Java Enterprise Edition
JSON	JavaScript Object Notation
MR	MapReduce
MVC	Model View Controller
OGC	Open Geospatial Consortium
OOXML	Open Office Extensible Markup Language
OpenGIS	Open Geographical Information System
ORM	Object Relational Mapper
REST	Representational State Transfer
RPC	Remote Procedure Call
SFTP	SSH File Transfer Protocol
SPA	Single Page Application
SQL	Structured Query Language

SSH	Secure SHell
SMTP	Simple Mail Transfer Protocol
SWM	Smart Water Meter
TCP/IP	Transmission Control Protocol/Internet Protocol
UI	User Interface
VM	Virtual Machine
YAML	Yet Another Markup Language

Table of Contents

1. System Architecture

In this section, we present the final SLIPO system, as it is currently deployed in a production setting. First, we provide an overview of its architecture and detail its major components, with an extended version available in the report deliverable D1.2 "Architecture". Next, we present the design and implementation details for all SLIPO software components. In the following, we present the production system applied to deploy SLIPO, briefly visiting its capabilities and characteristics, as well as its initialization. Finally, we present how documentation is produced and provide links to the actual documentation of the system.

1.1. Architecture

A high-level overview of the architecture of the SLIPO system is depicted in Figure 1. The SLIPO system consists of the following main modules:

- **SLIPO Toolkit**: a collection of individual software components for the quality-assured POI data integration, including transformation, interlinking, fusion, enrichment and analytics. SLIPO toolkit components can either be installed locally or invoked as part of the SLIPO workbench and APIs functionality.
- **SLIPO Workbench**: a web application, which integrates SLIPO Toolkit components to implement POI data integration workflows in a coherent, simple to use, and flexible manner. Workbench allows users to design custom data integration workflows, schedule the execution of workflows, monitor the state of active workflow executions and visualize results.
- **SLIPO APIs**: a collection of RESTful HTTP programmatic interfaces for invoking SLIPO Toolkit component functionality and integrating it into third-party systems. APIs only support the invocation of simple atomic functions (*e.g., transformation*). For composite operations, the Workbench web application must be used. Both, SLIPO Workbench and APIs are exposed by the same web application server.
- **Analytics**: a collection of software libraries and services for data analysis tasks on POI data.
- **Libraries**: a collection of software libraries for exposing **SLIPO API** bindings to external clients, executing POI data integration workflows and rendering workflow execution visualizations.

The SLIPO system implements a **workflow engine** that executes data integration workflows and a **scheduler** for initializing workflow executions. A **workflow** consists of several loosely coupled tasks. A **task** may invoke an operation implemented by a component of the SLIPO Toolkit, or perform secondary operations (*e.g., prepare configuration files, update metadata, copy files*).

The workflow engine and the SLIPO Toolkit components are deployed over a cloud infrastructure (see Section 1.3). Workbench and APIs exchange messages with the scheduler to execute workflows. The *scheduler* propagates requests to the *workflow engine* which subsequently initiates the execution of one or more tasks. A task is executed either in-process locally on the scheduler host, or remotely using Docker containers. Each Toolkit component is responsible for providing a scalable implementation for the

requested operation, inside the context of the running OS process. A Toolkit component which advertises itself as capable of partitioning its input (*and, of course, merging its output*) can also scale to multiple Docker containers. The scheduler only controls the total amount of resources allocated to a container, enforcing CPU/memory quotas derived from component-specific requirements and input size.

A more detailed view of the logical architecture of SLIPO is depicted in Figure 2 and consists of several layers. Every component in each layer depends and interacts only with components from the layers lower in the stack unless stated otherwise. Next, we provide a brief description of each artifact developed or extended during the project, starting from the top layer of the stack and moving downwards.

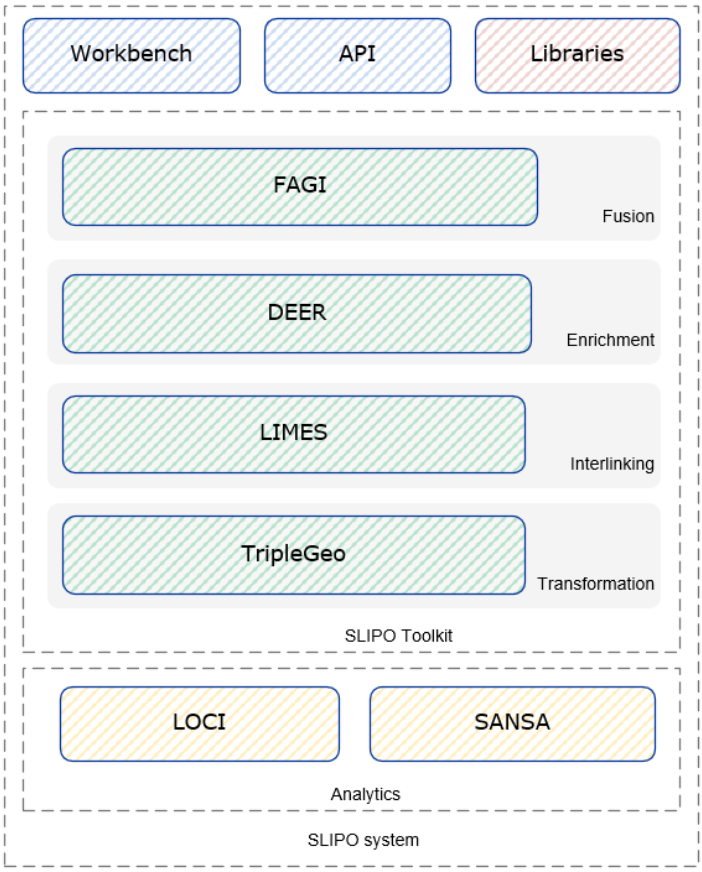


Figure 1: SLIPO system components

The top most layer consists of two modules, namely Workbench and RESTful API.

- **Workbench** is a SPA web application (*Single Page Application*) that integrates the **SLIPO Toolkit** components and allows users to declare POI data integration workflows in a coherent, simple to use, and flexible manner. More specifically, it provides tools for (a) uploading, searching and managing POI datasets in several formats, (b) designing, persisting and managing data integration workflows for POI datasets based on the features provided by the **SLIPO Toolkit**, (c) scheduling and monitoring the execution of the integration workflows, (d) visualizing the final results and interim indicators of workflow executions, and (e) performing quality assurance operations. Workbench implements the user interface of the SLIPO Workbench module.

- **RESTful API** exposes the **SLIPO Service** functionality lower in the stack to third-party applications and services. The API implementation applies a rigid security model that restricts the available features compared to the **Workbench** application. The API supports only the execution of atomic SLIPO Toolkit component operations. Composite operations are only implemented by the **Workbench** application. The **API** defines the method call signatures while the actual implementation is deferred to the **SLIPO Web Application**.
- **Libraries** consists of two software components, namely **SLIPO Python** and **SLIPO Frames**, that provide programmatic bindings for the **RESTful API** to Python and Jupyter notebooks respectively. **SLIPO Frames** depends on **SLIPO Python** and offers strong data-type bindings for API responses and workflow execution visualization features.

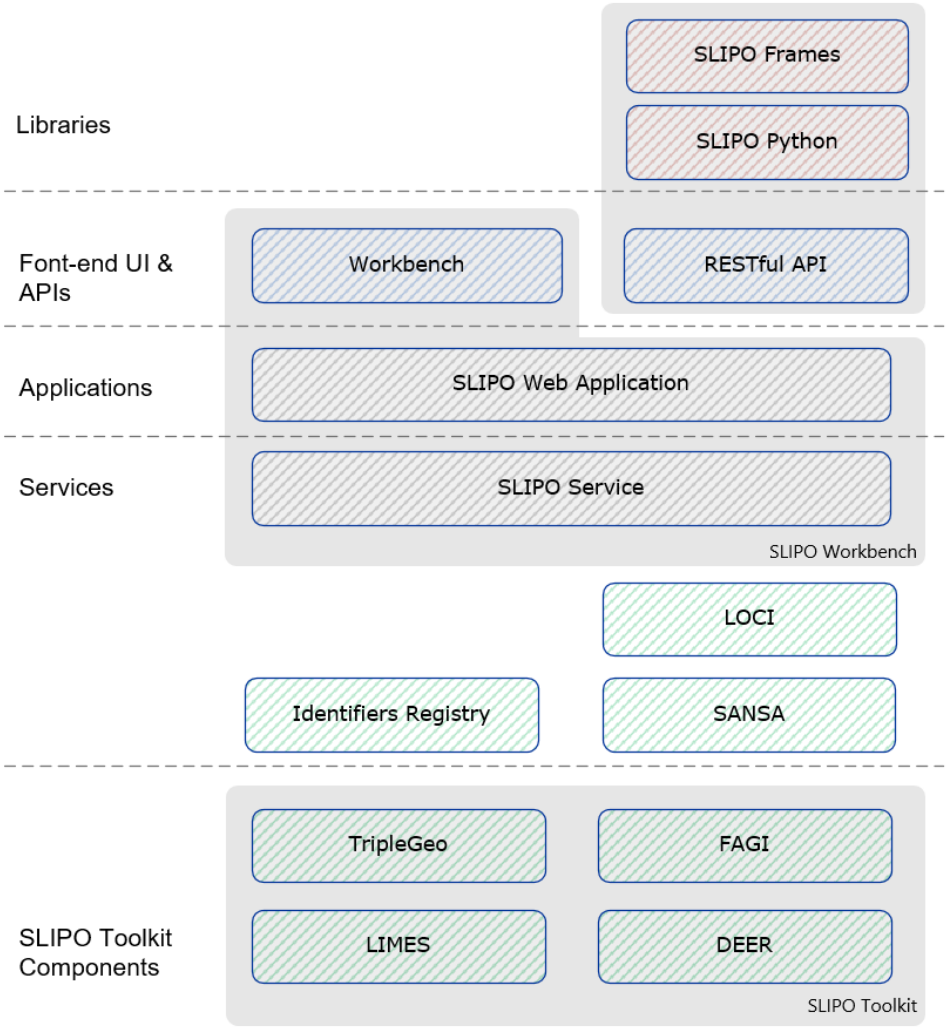


Figure 2: SLIPO system architecture and modules

The **Workbench** and **RESTful API** functionality is implemented by the next layer, which consists of a single module, the **SLIPO Web Application**.

- **SLIPO Web Application** acts as a middleware that exposes the operations of the **SLIPO Service** to either **Workbench** or **RESTful API** clients. It is responsible for enforcing authentication and

authorization constraints. Depending on the client, it may apply additional security rules such as CSRF protection for **Workbench** clients. Moreover, it decouples the operating system account that submits requests to **SLIPO Service** from the account that executes the requests.

The **SLIPO Web Application** propagates requests to the **SLIPO Service** in the next layer.

- The **SLIPO Service** is responsible for orchestrating the execution of POI data integration workflows. A workflow consists of one or more operations implemented by the SLIPO Toolkit components which are executed sequentially, in parallel, or conditionally. For each workflow operation, the SLIPO Service prepares input files, provides configuration options, invokes the appropriate component of the **SLIPO Toolkit**, collects logs and quality assurance data, and stores intermediate results. It also provides logging and tracing facilities for the whole workflow execution. Moreover, it aggregates logs and integrates quality assurance data from all individual operations to compute and visualize quality assurance metrics for the whole workflow.
- The **Identifiers Registry** service supports the registration and fast look-up of multiple POI identifiers, thus, allowing the discovery of duplicate POIs in different linked POI datasets.
- **LOCI** and **SANSA** provide value-added analytics services for POI data assets.

Next are the **SLIPO Toolkit Component Wrappers**.

- The **SLIPO Toolkit Component Wrappers** encapsulate **SLIPO Toolkit** components. The purpose of these components is to *decouple* the implementation details of the **SLIPO Toolkit** components from the **SLIPO Service**.

The **Workbench, SLIPO Web Application** and **SLIPO Service** components comprise the **SLIPO Workbench** module and are presented in detail in Sections 1.2.1, 1.2.4 and 1.2.5 respectively. An overview of the SLIPO APIs is also provided in Section 1.2.2. Finally, a detailed walkthrough of the user interface with examples is provided in Section 2.

1.2. Application Patterns and Design

In the next sub-sections, we present the design and implementation details for all SLIPO software components described in Section 1.1. To facilitate understanding, the reader is invited to consider the following:

- All SLIPO software components use additional external libraries and frameworks as detailed in Deliverable D1.2 "Architecture". During the presentation that follows, we frequently make references to these libraries and frameworks, since they affect the implementation details of each component.
- In SLIPO, we apply the Model View Controller pattern (MVC) and the Single Page Application (SPA) web application design. This pattern and design are used extensively in the SLIPO implementation and they strongly influence the structure of the source code. A short explanation for each follows; a broader coverage of these topics is outside the scope of this document:
    - Model View Controller (MVC). The goal of the Model View Controller pattern is to separate code responsibilities into three parts. The Model, which represents application domain data and logic, the View, which is responsible for the data presentation, and the Controller, who

receives user interactions and updates the Model appropriately. This separation increases code testability and improves a developer team's productivity. Nowadays, there are many variants of the MVC pattern and each MVC framework may implement the pattern in different ways. For the SLIPO implementation we are using the Spring Framework and its corresponding MVC module. A short description of the Spring Framework can be found in Deliverable D1.2 "Architecture". In SLIPO implementation we are using only a single view since the whole **Workbench** application is using the SPA design.

- Single Page Applications (SPAs) offer increased UI usability that is in par with desktop applications. In contrast to traditional web applications, a SPA application is initialized by loading only a single web page. After initialization, any additional resources such as data or JavaScript code files are loaded dynamically on demand using Asynchronous JavaScript and XML (AJAX) requests. Moreover, client-side code is usually implemented using the MVC pattern or some variant of it.

1.2.1. Workbench

To offer increased UI usability that is in par with desktop applications, the Workbench web application, referenced as the 'application' in this sub-section for brevity, is being developed as a Single Page Application (SPA). The application provides the UI elements required or performing tasks related to POI data integration workflow management including:

- Uploading data of several formats;
- Design and persist data integration workflows;
- Schedule and execute data integration workflows;
- Monitor and manage the execution of workflows;
- Visualize the results of a single execution instance;
- Visualize provenance information of individual execution workflows;
- Perform manual quality assurance operations;

The application code is part of the SLIPO Workbench module and is available at the SLIPO-EU/workbench[1] repository on GitHub under the Apache License v2.0. A diagram with the most important components of the application is shown in Figure 3 and includes the following:

[1] https://github.com/SLIPO-EU/workbench/tree/master/webapp/src/main/frontend

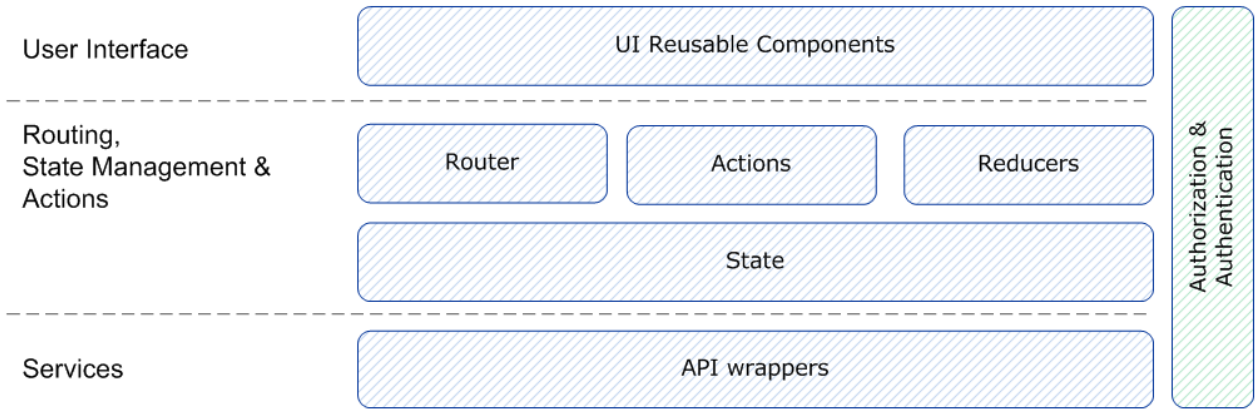


Figure 3: Workbench components

- **UI components**: In SLIPO, we are using the **React** library for building UI components. React builds components using encapsulation of state and composition. Thus, complex user interfaces can be composed from simpler, reusable components. The declarative programming model of React also makes components more predictable and easier to debug. All UI components are located under the components[2] folder in the repository. We separate components in four categories, namely, layout, page, view and helper components. Layout components control the layout of the whole application. Page components represent independent views that replace the layout completely e.g. login or error pages. Views represent pages that are rendered inside the application layout e.g. the workflow designer or the workflow execution browser. A View component may be composed but multiple simpler view components. Finally, helper components are utility components that are used inside other views but never as top-level components.
- **State, Actions and Reducers**: These are the components that implement the core of the application logic and manage application state. React components are using the application state to render views for the user to interact with. In SLIPO we are using the **Redux** pattern for managing state. Redux strictly dictates how the state is updated and attempts to make state mutation predictable by managing state mutation and asynchronicity. To this end, Redux removes two-way interactions between the UI components and the application state. Updating the application state is always a unidirectional process and involves the creation and dispatching of an action, which in turn invokes one or more functions, namely reducers, that update the state. All actions and reducers are in ducks[3] folder in the repository. The state is composed hierarchically with each reducer implementing a specific part of the state tree.
- **Router**: The **Router** component is responsible for handling navigation in the Workbench application and integrates with the Redux state management, thus making available browser location data as part of the state. In the application, all routing data is in routes[4] file which allows easily changing routes for the whole application by updating a single file.

[2] https://github.com/SLIPO-EU/workbench/tree/master/webapp/src/main/frontend/js/workbench/components

[3] https://github.com/SLIPO-EU/workbench/tree/master/webapp/src/main/frontend/js/workbench/ducks

[4] https://github.com/SLIPO-EU/workbench/blob/master/webapp/src/main/frontend/js/workbench/model/routes.js

- **API wrappers**: API wrappers are classes that expose API endpoints and services as a collection of simple functions that can be easily invoked by Redux actions or UI components. They are in service[5] folder and are implemented either as synchronous methods or asynchronous ones using JavaScript Promises. In either case, they share a common implementation that makes error and security handling easier to handle.
- **Authentication & Authorization**: The application defers all security checks to the server. In order to enhance the user experience, two components are provided for displaying content conditionally based on the authenticated user roles, namely, SecureContent[6] and SecureRoute[7]. These components display content and allow navigation respectively based on the user roles.

The application can be built either as part of the SLIPO Workbench module using the Maven[8] software project management tool, or independently using the Grunt[9] JavaScript task runner. In either case, the distributable artifacts are packaged for deployment inside the SLIPO Web Application package.

1.2.2. RESTful API

The RESTful API exposes part of the SLIPO Service functionality to third-party applications and services. Its actual implementation resides in the SLIPO Web Application module. The specification of the API is fully documented using the apiDoc[10] documentation tool and is published at the SLIPO project development web site. The API exposes the following operations:

- Browse and download user files from the server.
- Query and download resources from the catalog.
- Query POI data integration workflows created by the workbench application.
- Enumerate available configuration profiles for SLIPO Toolkit components.
- Invoke an atomic operation implemented by a SLIPO Toolkit component.
- Initialize the execution of an existing POI data integration workflow.
- Query the state of a running operation execution.
- Terminate a running operation execution.
- Retrieve the results of a completed operation.

To invoke the API, a client must provide a valid application key which is linked to an existing SLIPO user account. Application keys are used to apply quota constraints to the respective clients.

[5] https://github.com/SLIPO-EU/workbench/tree/master/webapp/src/main/frontend/js/workbench/service

[6] https://github.com/SLIPO-EU/workbench/blob/master/webapp/src/main/frontend/js/workbench/components/helpers/secure-content.js

[7] https://github.com/SLIPO-EU/workbench/blob/master/webapp/src/main/frontend/js/workbench/components/helpers/secure-route.js

[8] https://maven.apache.org/

[9] https://gruntjs.com/

[10] http://apidocjs.com/

1.2.3. SLIPO Libraries

An external 3rd party client can access SLIPO functionality by directly invoking the SLIPO API. A Python module, namely **slipo**, has also been implemented to provide bindings for SLIPO API to Python code. The library code is available at the project repository on GitHub[11] and can be easily installed from the Python Package Index[12]. Detailed documentation for the library is also published[13].

Based on the **slipo**, a second Python module, namely **SLIPOFrames**, that targets Jupyter notebooks has been developed. In addition to exposing SLIPO API to Jupyter notebooks, **SLIPOFrames** converts API responses to DataFrames[14] for better integration with data analysis tools and offers workflow execution visualization features.

The library source code[15], distribution package[16] and documentation[17] are provided online.

1.2.4. SLIPO Web Application

The SLIPO Web Application is an MVC web application that exposes the operations of the SLIPO Service to the Workbench client and implements the RESTful API. It is implemented using the Spring Boot framework. The source code along with installation instructions is part of the SLIPO-EU/workbench[18] repository and is available under the Apache License v2.0. The main components of the SLIPO Web Application are depicted in Figure 4 and enumerated next:

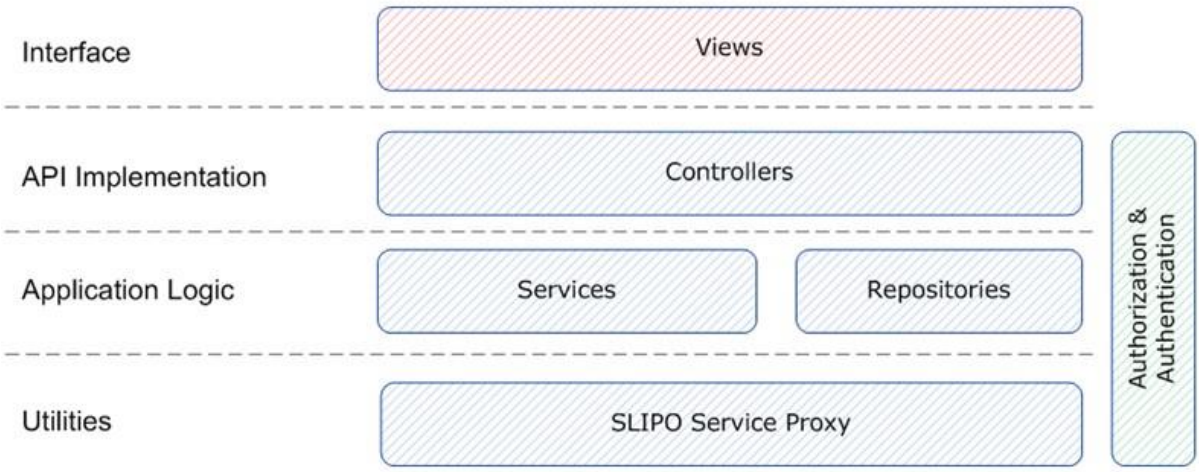


Figure 4: SLIPO Web Application components

[11] https://github.com/SLIPO-EU/slipo-python
[12] https://pypi.org/project/slipo/
[13] https://slipo.readthedocs.io/en/latest/
[14] https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html
[15] https://github.com/SLIPO-EU/slipo-frames
[16] https://pypi.org/project/slipoframes/
[17] https://slipoframes.readthedocs.io/en/latest/
[18] https://github.com/SLIPO-EU/workbench/tree/master/webapp

- **Views**: The application has only a single view with the sole purpose of bootstrapping the Workbench application by including the appropriate JavaScript and CSS resources. The view has no model assigned to it and renders little to none HTML content since rendering and state management is deferred to the Workbench application.
- **Controllers**: The application implements two groups of controllers for publishing two APIs, namely, the HTTP Action API[19] and the RESTful API[20] endpoints. The former is used exclusively by the Workbench application, it is a stateful API and it applies CSRF protection to all state changing methods. The latter is the actual implementation of the SLIPO APIs, it is stateless and is available to third-party consumers. All messages exchanged with the controllers are encoded in JSON format with the only exception of file up/down-loading.
- **Services**: Controllers are implemented as lightweight components and their main responsibilities are to parse incoming requests, manage authentication and handle request routing. All business logic is deferred to service components such as the ProcessService[21], a service component used for managing data integration workflows.
- **SLIPO Service Proxy**: A component that acts as a gateway for invoking SLIPO Service methods. All operations specific to the workflow execution engine and scheduler are propagated to the SLIPO Service through this component.
- **Repositories**: Components that allow controllers or services to access data stored in a PostgreSQL relational database. Such data may include workflow engine and scheduler data if they are just verbatim data from relational tables that require no further processing (*e.g., workflow definition data*). If additional processing or validation is required, the SLIPO Service Proxy is used.
- **Authorization & Authentication**: For the HTTP Action API, the application supports (a) form-based user authentication, (b) role-based user authorization at the controller, service and repository levels and (c) CSRF token protection for all state changing methods. The RESTful API access is restricted using application keys.

1.2.5. SLIPO Service

The SLIPO Service implements the core functionality of the POI data integration lifecycle. It is responsible for scheduling and executing data processing workflows in a scalable manner, maintaining metadata about datasets and jobs, and aggregating statistics generated by individual tool executions. Moreover, it performs other secondary tasks like harvesting datasets from external data sources and performing data analysis on Linked POI datasets.

The SLIPO Service source code is in the SLIPO-EU/workbench[22] repository and is available under the Apache License v2.0. The application is implemented using the Spring Boot framework and shares a similar structure with the SLIPO Web Application.

---

[19] https://github.com/SLIPO-EU/workbench/tree/master/webapp/src/main/java/eu/slipo/workbench/web/controller/action

[20] https://github.com/SLIPO-EU/workbench/tree/master/webapp/src/main/java/eu/slipo/workbench/web/controller/api

[21] https://github.com/SLIPO-EU/workbench/blob/master/webapp/src/main/java/eu/slipo/workbench/web/service/ProcessService.java

[22] https://github.com/SLIPO-EU/workbench/tree/master/rpc-server

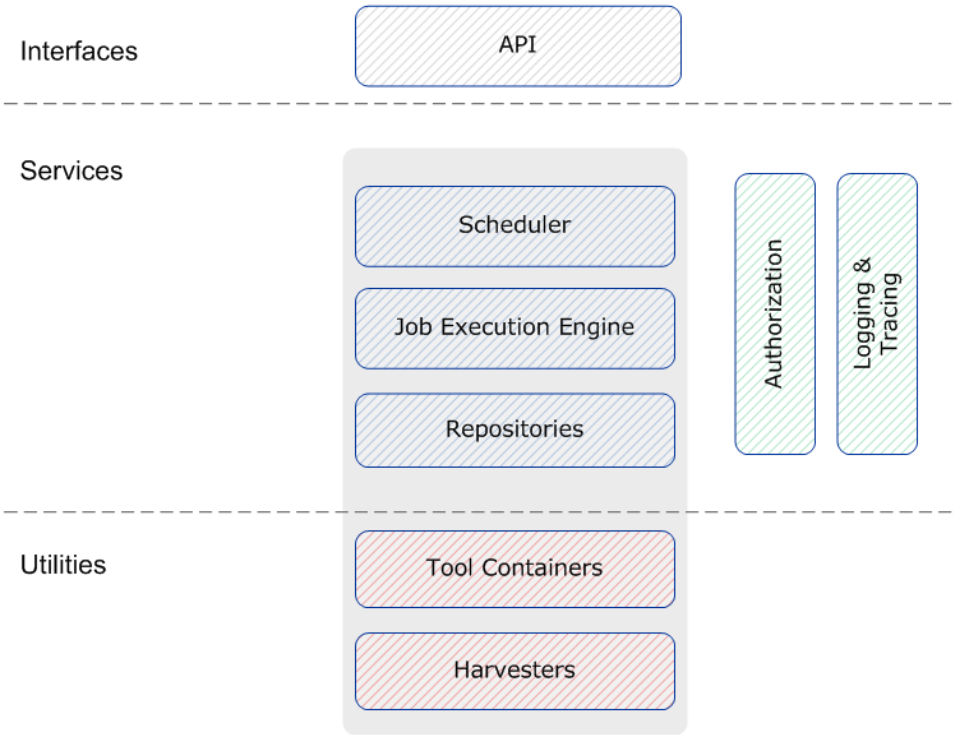


Figure 5: SLIPO Service

The main components of the SLIPO service, as depicted in Figure 5, are:

- **API**: The **API** allows external users to easily query the service status (*e.g., enumerate the running workflow executions or view the state of a single execution instance*). It is implemented as a RESTful API that exchanges JSON messages.
- **Proxy Service**: The proxy service allows the communication between the SLIPO Web Application and Service. It is implemented using Spring's HTTP invoker[23], a special remoting strategy which allows for Java object serialization via HTTP.
- **Scheduler**: A component that initializes the execution of jobs at specific points in time. A job represents any processing task that must be executed asynchronously. Currently, two types of jobs are implemented, namely, data integration workflows and scheduler metadata maintenance tasks. The former is executed on demand while the latter is scheduled for periodic execution using a CRON expression.
- **Job Execution Engine**: A component that orchestrates the execution of data integration workflows. It collects all required input and configuration files, submits units of work (tasks) to remote processing nodes, monitors the execution of each task, optionally retries failed ones and collects output, quality assurance and sample data. A data integration workflow is modeled as a

[23] https://docs.spring.io/spring/docs/current/spring-framework-reference/integration.html#remoting-httpinvoker

graph of Spring Batch[24] flows. A separate project has been developed for modeling workflows and is available at the SLIPO-EU/workflows[25] repository.

- **Repositories**: Components used for storing and updating data. Each repository focuses on a specific task, e.g., the Job Repository is responsible for managing scheduler data. Every component higher in the stack accesses persisted data through the appropriate repository.
- **Tool Adapters:** Components used for abstracting configuration options and execution details of the SLIPO Toolkit components. Each external tool is wrapped with a custom adapter. The Job Execution Engine communicates directly only with an adapter instead of using the tool library or service directly.
- **Harvesters**: Services used for importing data from external sources (e.g., CSW servers, open data catalogues). These datasets may include spatial data, such as POI data, or data that can be used for the enrichment of existing Linked POI datasets.
- **Authentication:** A module that provides a security context to any operation executed inside the SLIPO Service. The security context contains information about the user's identity and roles. The SLIPO Service does not perform any kind of authentication but instead receives the user identity from the SLIPO Web Application.
- **Logging & Tracing**: A module that provides the facilities for logging and tracing operations inside the SLIPO service. It aggregates logs from several sources and supports searching log data for specific operations and modules, e.g., filtering log messages for the scheduler module or reading execution logs for a specific job. The schema of database tables used for logging workflow executions are described in Annex 4.1.

1.2.6. SLIPO Toolkit

The SLIPO Toolkit comprises all components of SLIPO available for local installation and use, addressing the major steps of the SLIPO data integration lifecycle. An overview of the SLIPO Toolkit is presented in D1.2 "Architecture", while the detailed architecture, design, and functionalities of each component are presented in detail in their corresponding deliverables:

- Transformation: D2.4 "Final mapping and transformation software"
- Interlinking: D3.7 "Final release of the POI integration software"
- Enrichment: D3.7 "Final release of the POI integration software"
- Fusion: D3.7 "Final release of the POI integration software"

Every component execution is modeled as a sequential Spring Batch flow. To execute a component, an appropriate job builder has been implemented[26]. A job builder creates all the steps required for collecting input files, composing configuration files, invoking the component's executable or service, extracting the

[24] https://docs.spring.io/spring-batch/trunk/reference/html/

[25] https://github.com/SLIPO-EU/workflows

[26] https://github.com/SLIPO-EU/workbench/tree/master/rpc-server/src/main/java/eu/slipo/workbench/rpc/jobs

results and performing any required post-processing of data. The execution is implemented using Docker containers. A Docker image has been composed for every SLIPO Toolkit component[27]. The configuration of every component is modeled using Java classes[28] that allows the SLIPO Workbench and SLIPO Service to exchange configuration data. At runtime, that configuration objects are serialized to the appropriate file format required by each component e.g., Java property files or XML.

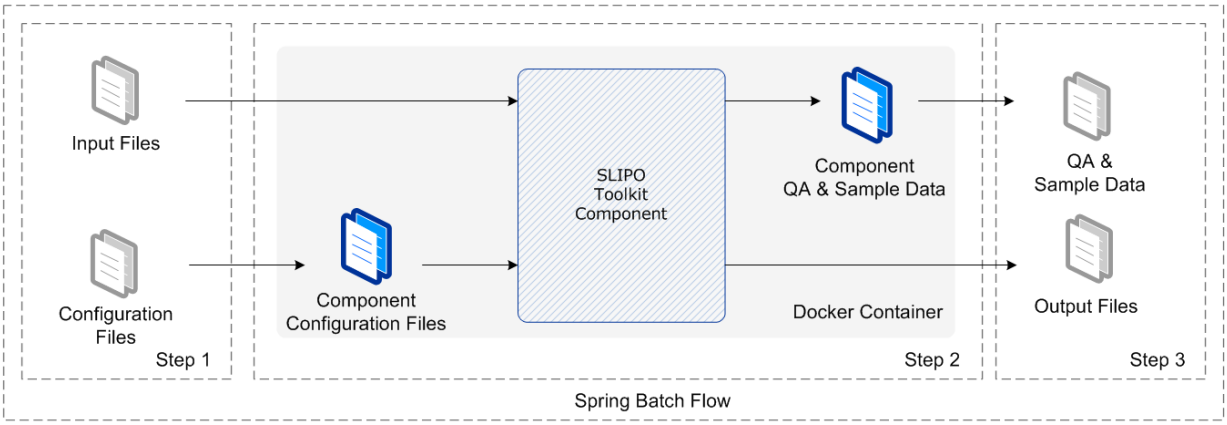


Figure 6: SLIPO Toolkit Component integration

A simple example of a SLIPO Toolkit component Spring Batch flow is displayed in Figure 6.

1.3. Deployment

In this section, we present the production system applied to deploy SLIPO, briefly presenting its capabilities and characteristics, as well as its initialization. Then, we present the actual production architecture in terms of cloud VMs (or groups of VMs), how software is packaged, and deployment is automated. The current versions for all major software components comprising the final SLIPO system are provided in Annex 4.3.

1.3.1. Production Architecture

The SLIPO system is deployed within the Hellenic Data Service[29] (HELIX), the national cloud infrastructure for data intensive research & innovation co-developed by Athena RC, harnessing its highly scalable and compute, network and storage resources. SLIPO is one of the first commercial services hosted by HELIX for Big Data, ensuring its sustainable operation and implementation of our exploitation plans.

HELIX comprises three components offering services to scientists and researchers: Pubs[30], providing search facilities for Open Access publications authored by Greek scientists and researchers, acting as a national OpenAIRE node and harvesting national institutional and thematic scientific repositories; Data[31], a CKAN-based data catalogue and repository for sharing, discovering, assessing, visualizing and downloading

[27] https://github.com/SLIPO-EU/docker-recipes

[28] https://github.com/SLIPO-EU/workbench/tree/master/common/src/main/java/eu/slipo/workbench/common/model/tool

[29] http://www.hellenicdataservice.gr

[30] https://pubs.hellenicdataservice.gr

[31] https://data.hellenicdataservice.gr

scientific data assets; Lab[32], an open-ended collection of services enabling the interactive and scalable manipulation, experimentation and analysis of Big Data assets. SLIPO has been deployed as a service of HELIX Lab, leveraging its underlying IaaS infrastructure for scalable management and execution for its data integration workflows. In addition, this allows the SLIPO services (via the SLIPO API, see 1.2.2) exposure to other HELIX Lab components, and thus the provision of alternate schemes for user interaction and value-added service provision. In Section 2.2.8, fwe demonstrate how users can author, manipulate, invoke, and experiment with POI data integration using Jupyter notebooks.

On a system level, SLIPO is deployed on top of the Synnefo cloud stack[33], within several virtual machines. Synnefo is a complete open source cloud stack written in Python that provides Compute, Network, Image, Volume and Storage services, like those offered by AWS[34]. On a hardware level, the production system is hosted on Athena RC's private IaaS available from okeanos-knossos.gr, currently allocating *256 CPU cores, 256 GB main memory, 20TB storage*. More details on Synnefo are provided in D1.2 "Architecture".

Through the Cyclades system (Synnefo UI), we configured a virtual private network consisting of Debian 8.7 (SMP Debian 3.16.39-1+deb8u2 x86_64 GNU/Linux) virtual machines to host the production system of SLIPO. Moreover, Docker Server Engine has also been installed on every virtual machine (*unless noted otherwise*) since all SLIPO software components are deployed as Docker Images. A Hadoop/Spark cluster is also available, (11 compute nodes, 16GB memory and 8 cores per node) sitting on an HDFS filesystem with a total capacity of 3.5TB (11 data nodes co-located with the compute nodes).

The final SLIPO system is deployed on the following virtual machine groups comprising of one or more nodes:

- **Admin Server**: This server stands as the administrative entry point to the private network. Is also responsible of rolling-out updates and monitoring several aspects of the whole deployment.
- **Web Server**: Hosts the Docker container for the SLIPO Web Application module publishing the SLIPO Workbench application and SLIPO RESTful APIs. This server is the only one having a public IP4/IP6 interface.
- **RPC Server**: Hosts the Docker container for the SLIPO Service module that is responsible for orchestrating the execution of data integration workflows.
- **Worker Server(s)**: Hosts all Docker containers wrapping SLIPO Toolkit components as part of data-integration workflows. The actual execution of data-integration tasks always happens inside these worker containers.
- **Storage Server**: The NFS server providing file storage to the private network. This is a crucial part of the deployment as worker containers exchange input/output data on this storage area.
- **Database**: Hosts a PostgreSQL/PostGIS database. No Docker engine is present on this server.

The general deployment scenario described above in illustrated in Figure 7.

[32] https://lab.hellenicdataservice.gr

[33] https://www.synnefo.org/

[34] https://aws.amazon.com/

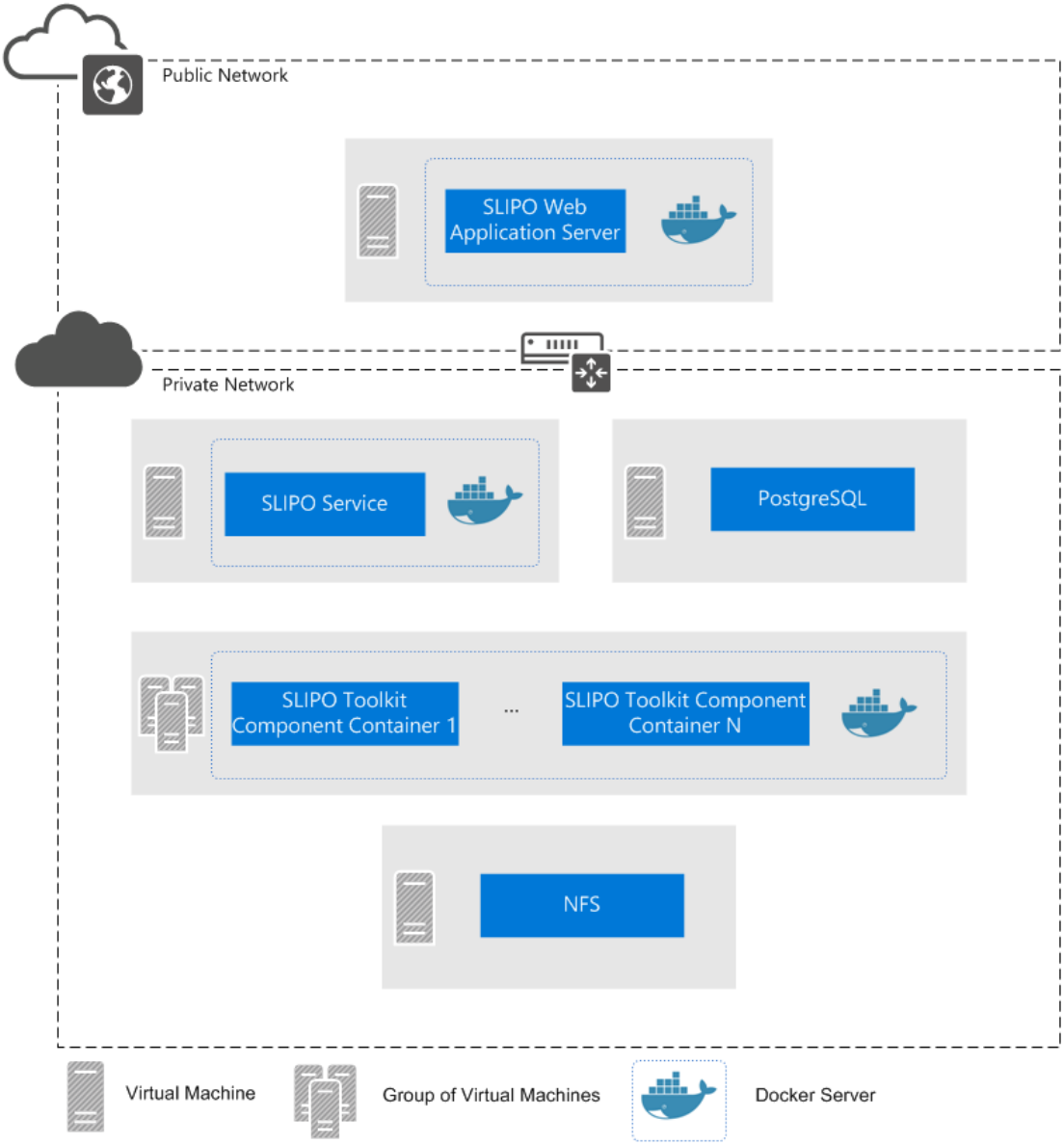


Figure 7: SLIPO system deployment

1.3.2. Packaging

The majority of software components required to deploy SLIPO is packaged as a Docker image. Of course, any site-specific configuration (*production-specific configuration*) is not part of the image and is expected to be supplied at runtime (*either as environment variable or as a bind-mounted filesystem layer*).

In a nutshell, we have two forms of packaging: (a) one for components that comprise the core services of the application, and (b) one for SLIPO Toolkit components. The services that comprise the core SLIPO Workbench application (*i.e., the Web application and the RPC server*) are packaged as Docker images at a post-build step of the core Maven packaging phase. This step prepares a lightweight build context

directory (to be sent to Docker daemon), and then generates the image. The details can be found at the top-level directory of the respective Maven submodule[35].

Each one of the SLIPO Toolkit components is built independently of the main project, and independently of other Toolkit components. The docker image that wraps a Toolkit component aims to provide a relatively common call interface for the RPC server. A dedicated Git repository[36] hosts the collection of all available build recipes. It must be noted that source projects are here considered as read-only "upstream" repositories having an independent development lifecycle. After a successful build, the build artifact is uploaded to Docker Hub under the 'athenarc" namespace (*e.g., athenarc/triplegeo:2.0*). The RPC server references Toolkit components by these exact fully-qualified names, spawning containers as-needed from these base images. Finally, please also note that it *is not technically feasible* to produce a *single* Docker image containing all SLIPO Toolkit components and SLIPO core services. As it is apparent from the corresponding Deliverables, the individual components have entirely different dependencies, execution environments, and scaling requirements.

1.4. Documentation

One of the most important aspects of software development is the creation of detailed and consistent code documentation. The existence of well documented code allows for better collaboration between members of the software development team and parallel development of separate software components that may have dependencies between each other. In this sub-section, we describe the documentation requirements of the project, present the tools used for generating documentation, and provide examples and links to the generated documentation for the SLIPO system.

1.4.1. Introduction

In the SLIPO project there are four distinct cases where documentation is required. First, project documentation is required to easily deploy and configure the SLIPO system in development or production environment. Second, server-side code that manages data, job execution and security, requires documentation to be accessible and extensible. Likewise, client-side code that executes on the browser needs to be documented. Finally, API documentation that allows client-side code to interact with server must be well documented for the developers to access functionality exposed by the SLIPO Web Application.

Depending on the code being documented, several tools have been used for generating documentation. In all cases, the documentation generation process has been integrated in the project build process. Hence, the documentation is updated whenever the code changes. The whole documentation is packaged along with the SLIPO distributable artifacts and becomes accessible once the SLIPO server application has been deployed. Moreover, the documentation artifacts are committed to the GitHub repository and published using GitHub Pages[37] feature.

---

[35] For example, https://github.com/SLIPO-EU/workbench/tree/master/rpc-server#build-docker-image

[36] https://github.com/SLIPO-EU/docker-recipes

[37] https://slipo-eu.github.io/workbench/

1.4.2. Tools

We are using three tools, namely, Javadoc[38] and apiDoc[39], for generating documentation for server-side code, client-side code, and API methods signatures respectively. Javadoc generates documentation files from comments inside the source code. This feature makes code self-documented and easier to understand without having to refer to the complete documentation.

For the API documentation, we followed a different approach in which the documentation is separated from the actual implementation code. The reason for this is that the target audience, i.e., the API consumers, is interested only in the public functionality exposed by the API methods. Still, the full implementation documentation exists as part of the server code documentation. Although there are several tools that allow the generation of API documentation from inside the source code such as JSONDoc[40] or Swagger[41], we avoided using them to reduce the clutter of additional annotations inside the source code.

For the project documentation, we are using the Apache Maven Site Plugin[42]. The generated artifact is a web site with detailed information on how to deploy and configure the SLIPO software.

Finally, all tools described above have full support for templating, thus allowing the configuration of the layout and styling of the generated documentation. Next, we give a brief description of every tool, followed by a simple example and a link to the publicly available actual documentation.

```
/**
 * Downloads a file for the selected execution
 *
 * @param id the process id
 * @param version the process version
 * @param executionId the execution id
 * @param fileId the file id
 * @return an instance of {@link FileSystemResource}
 * @throws IOException  if an I/O exception has occurred
 */
@RequestMapping(value = "/action/process/{id}/{version}/execution/{executionId}/file/{fileId}/download", method = RequestMethod.GET)
public FileSystemResource downloadProcessExecutionFile(
    @PathVariable long id, @PathVariable long version, @PathVariable long executionId, @PathVariable long fileId,
    HttpServletResponse response) throws IOException {

    final File file;
    try {
        file = this.processService.getProcessExecutionFile(id, version, executionId, fileId);
        if ((file != null) && (file.exists())) {
            response.setHeader("Content-Disposition", String.format("attachment; filename=%s", file.getName()));
            return new FileSystemResource(file);
        }
    } catch (ProcessNotFoundException ex) {
        response.sendError(HttpServletResponse.SC_NOT_FOUND, "Process was not found");
    } catch (ProcessExecutionNotFoundException ex) {
        response.sendError(HttpServletResponse.SC_NOT_FOUND, "Process execution was not found");
    }

    response.sendError(HttpServletResponse.SC_GONE, "File has been removed");
    return null;
}
```

Figure 8: Javadoc example

1.4.2.1. Javadoc

The Javadoc tool is used for generating documentation for Java source code. It does so by parsing the documentation comments from all java source files and produces HTML pages for every package, class and

[38] http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javadoc.html

[39] http://apidocjs.com/

[40] http://jsondoc.org/

[41] http://swagger.io/

[42] https://maven.apache.org/plugins/maven-site-plugin/

interface. For each class and interface, documentation is generated for public and protected constructors, methods and fields. An example of Javadoc-compatible documentation comments is depicted in Figure 8.

The generated documentation of SLIPO is available at:

- https://slipo-eu.github.io/workbench

1.4.2.2. apiDoc

apiDoc can create API documentation for several languages using annotations either in the source code itself or separate files. Moreover, it supports generating API history which allows developers to easily detect API changes between different versions. Using JavaScript files, we created documentation for all public API methods and the corresponding messages.

The source code for the API documentation can be found at:

- https://github.com/SLIPO-EU/workbench/tree/master/webapp/apidoc/src/webapp-api

while the actual generated documentation can be browsed at:

- https://app.dev.slipo.eu/docs/webapp-api/index.html

1.4.2.3. Apache Maven site plugin

The Apache Maven Site Plugin generates an HTML web site for the project. In contrast to the previous tools, Site Plugin generates project-wide documentation with information about setting up the SLIPO required databases, configuring application settings, building the SLIPO application from source code, and deploying the application.

The project documentation can be viewed at:

- https://slipo-eu.github.io/workbench

2. SLIPO Workbench

In this section, we present a walkthrough of the final SLIPO system, visiting its various integrated functionalities, and provide examples of selected POI data integration workflows.

2.1. System Walkthrough

SLIPO Workbench is a web application that integrates SLIPO Toolkit components to implement POI data integration workflows in a coherent, simple to use, and flexible manner. Workbench allows users to design custom data integration workflows, schedule the execution of workflows, monitor the state of active workflow executions, and visualize results.

Every functionality is implemented using appropriate user interface components referred to as *views*. For complex operations, more than one views may be combined or collaborate to implement the required functionality. Each view is embedded in the main application layout as it is shown in Figure 9.

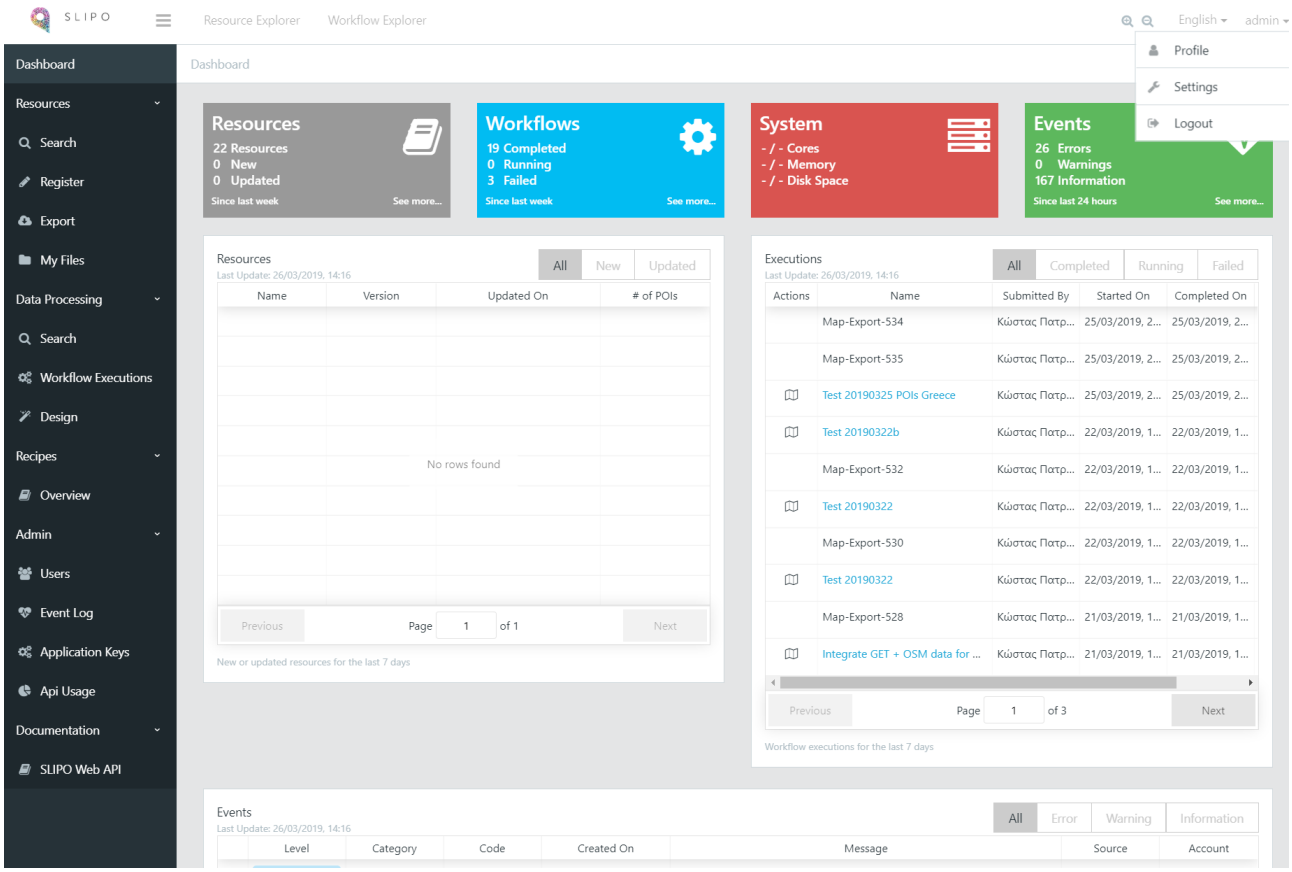


Figure 9: Application Layout

The layout consists of five distinct parts. On the left side, there is a collapsible pane that contains the application main menu that allows users to navigate between the various sections of the application.

On the top, there is a header that contains links to a subset of the main menu sections. These links are context-aware and are dynamically displayed based on the current rendered view. For instance, when the user is browsing the existing workflows, the header provides links to create a new workflow or to switch to the workflow execution browser view.

Under the header, there is a breadcrumb section that allows user to quickly identify the current rendered view and if applicable, provide links to other related views. For instance, when the user is browsing a thematic map with the results of a workflow execution, she can easily switch to the execution viewer without having to navigate to the workflow execution browser and search for the corresponding record.

On the right side, there is a collapsible pane that renders content based on the context of the current view. For instance, the workflow designer is using the right pane for rendering data about resources and displaying validation messages. Both left and right panes are collapsible to increase the screen real-estate. The latter is very important when complex views are displayed such as the workflow designer.

Finally, at the center of the application layout is where the active view is rendered.

Figure 10: Toast examples

In addition to the layout and views, the application uses two more user interface artifacts to provide user feedback and to receive input, namely, toasts and dialogs.

Toasts are transient dialogs that appear on the right top corner of the application and display information about the outcome of most recent user action. Depending of the action result, a toast may indicate an error, warning, or information condition. Two toast examples, a successful and failure one, are displayed in Figure 10.

Dialogs are used to receive user confirmation before performing an action that may cause irreversible changes to data (*e.g., discard a new workflow or deleting a file*). A dialog example is shown in Figure 11.

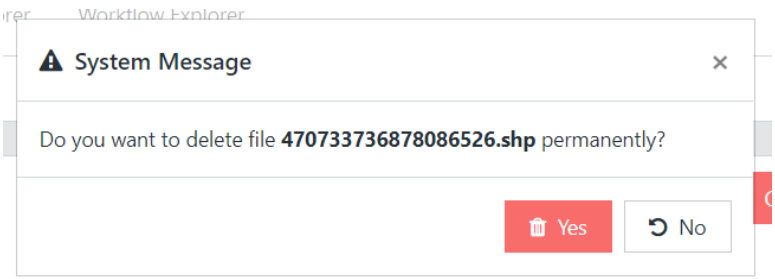


Figure 11: Dialog example

2.1.1. Dashboard

Dashboard, shown in Figure 12, is the default view rendered after the user has been successfully authenticated. It contains aggregated data about resources, workflows and system events as well as records about the most recent system operations.

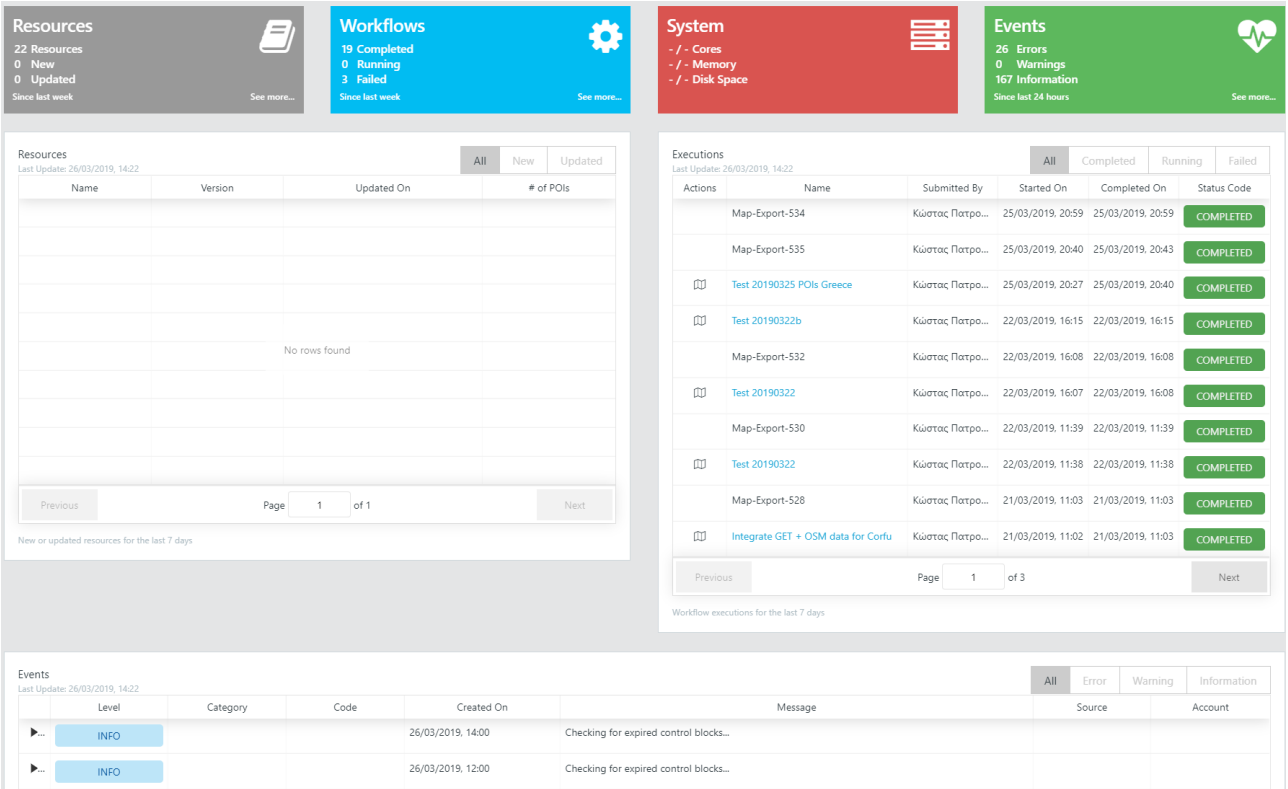


Figure 12: Dashboard view

On the top of the view several data cards are displayed. Each card contains data about a specific functionality of the Workbench (e.g., resources, workflows). An example of two cards in depicted in Figure 13.

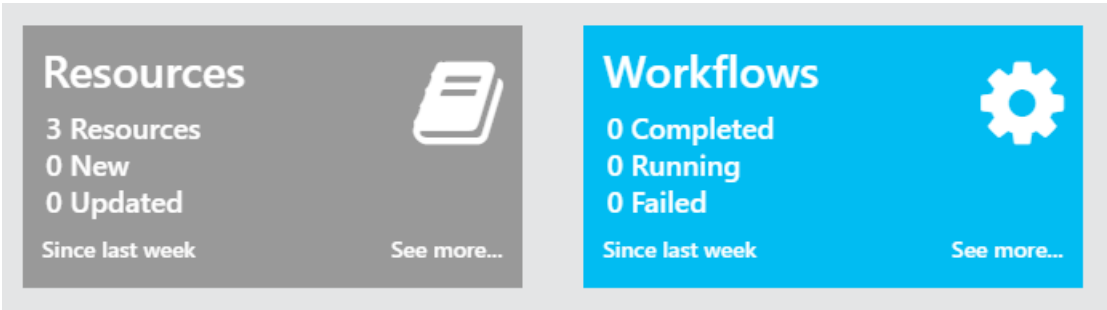


Figure 13: Data card examples

There are four types of cards:

- **Resources**: A card that contains aggregated statistics about catalog resources that have been created or updated during the last week time interval.

- **Workflows**: This card displays several counters for workflow executions grouped by their status e.g. Completed, Running or Failed for the last week time interval.
- **System**: Presents simple statistics about the status of the SLIPO system. That includes CPU cores, memory and disk space usage as well as the status of the RPC server.
- **Events**: Consists of several counters for system events that occurred during the last day interval, grouped by their importance, such as error, warning or information events.

Depending of the type of data displayed in a card, a link may also be displayed for rendering a view with additional details, e.g., from the workflows data card, the user can navigate to the workflow execution browser.

Below the cards, there are three tables with detailed information for workflow executions, resources and system events. The data in these tables refer to the time intervals of the corresponding data cards. An example of each table is illustrated in Figure 14, Figure 15 and Figure 16 respectively. The workflow executions table provides a quick overview of all recent workflow executions. The user can filter the records based on their execution status and drill down to view the details of a specific record. Moreover, if input and output data for a specific execution has been imported to PostGIS, a map can be used for browsing POI data.

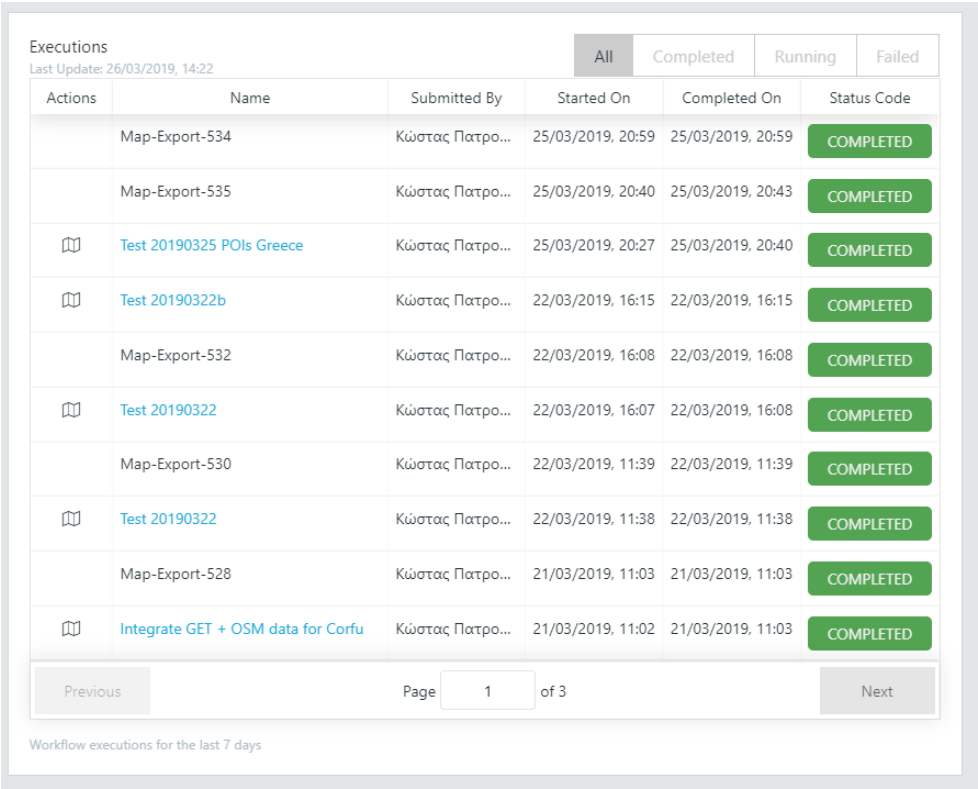


Figure 14: Workflow executions

In a data integration workflow, any file that can be transformed by TripleGeo, can also be used as an input. Every time a workflow executes, these input files are transformed to RDF datasets. If an RDF dataset is used frequently, the user can choose to register it in a catalog, and thus make it available to multiple workflows. The Resources table contains information about all recently registered or updated catalog resources. The user can filter either new or updated resources and drill down to view the details of a specific resource.

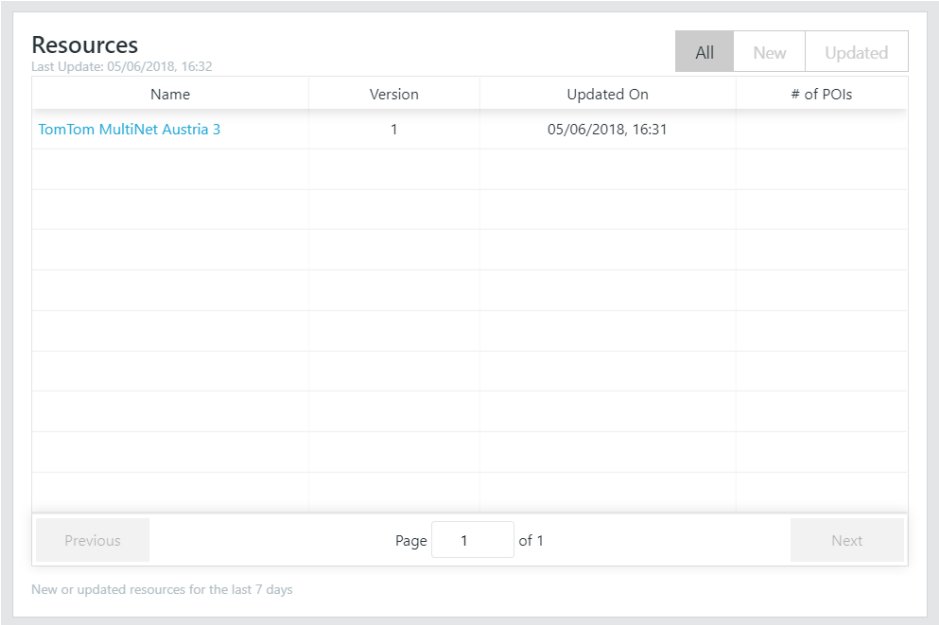


Figure 15: Catalog resources

Finally, the system events table displays records for system messages. This table is rendered only when the authenticated user is an administrator. Rows can be filtered based on the message level (i.e., importance) and each row can be expanded to accommodate longer messages.

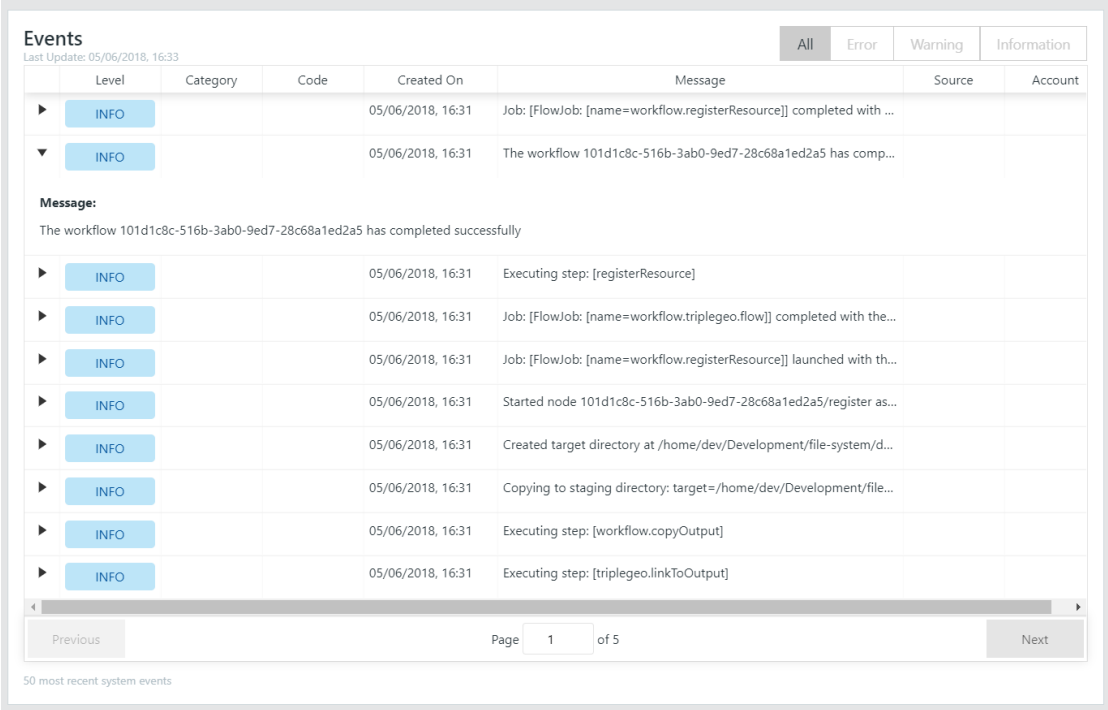


Figure 16: System events

2.1.2. Resources

The resource catalog, shown in Figure 17, stores metadata about RDF datasets frequently used in data integration workflows. The goal of the resource catalog is to avoid executing TripleGeo operations multiple times for the same files. Once a file is registered, it can be selected for use in the workflow designer. Selected

resources are displayed on the right sidebar and can also be filtered based on the dataset type. SLIPO supports two dataset types, namely, POI and Link data.

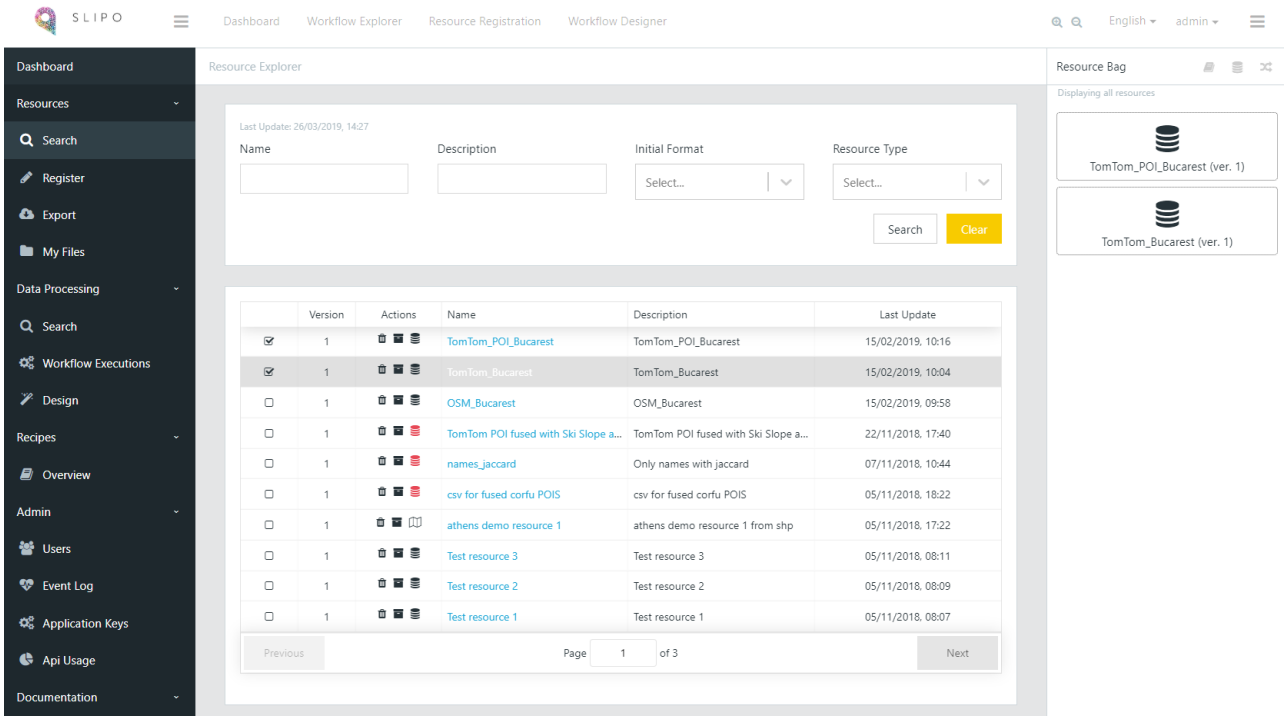


Figure 17: Resource catalog

By selecting a record from the table, the user can view details about the resource metadata as illustrated in Figure 18. If the resource data has been imported to PostGIS, the user can also preview POIs using a map.

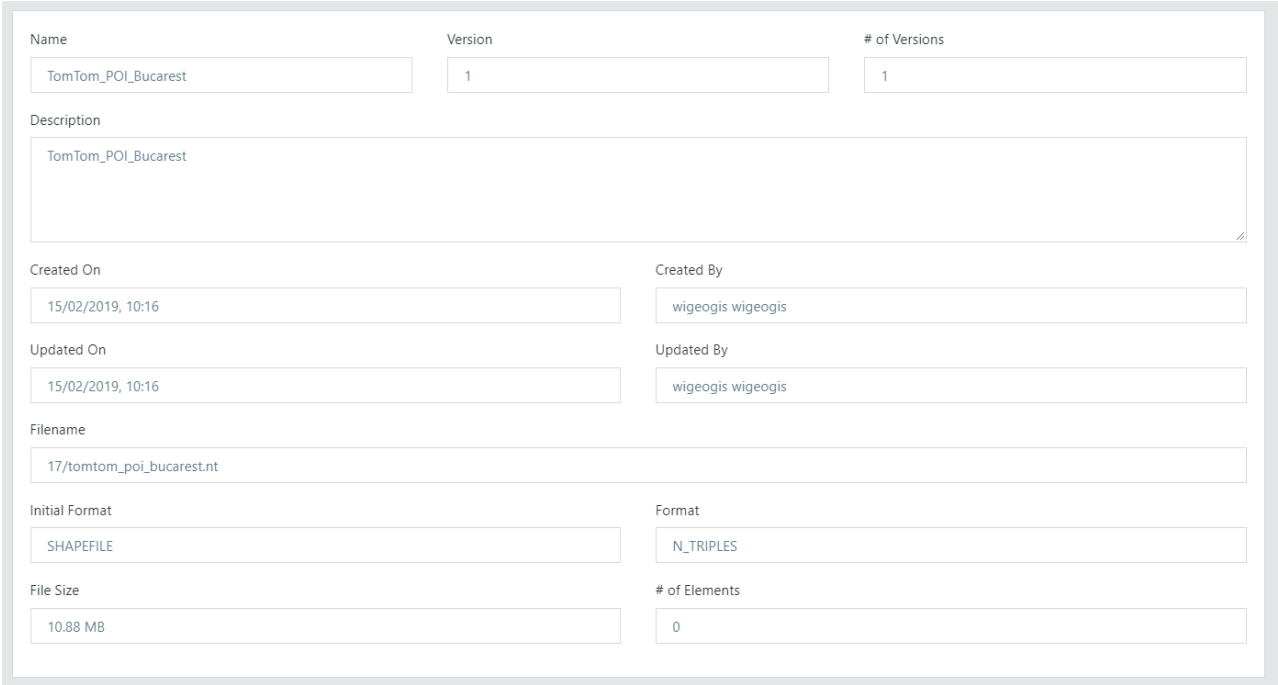


Figure 18: Resource Details

The output of any workflow step, that creates POI data, can be registered to the catalog by creating an appropriate registration step in the workflow designer as described later in Section 2.1.4. Since resource

catalog registration is a common task, a wizard has been implemented to streamline resource registration. The wizard consists of five steps:

- Input mode
- Resource selection
- Resource metadata
- TripleGeo configuration
- Confirmation

The "Input mode" step allows the user to select the data source of the file to register as shown in Figure 19. A data source controls how the input file is transferred to the web server. Currently, the wizard supports two data source types, namely, upload and file system. In the former, the user uploads the file manually while in the latter the user selects an existing file from the server file system.

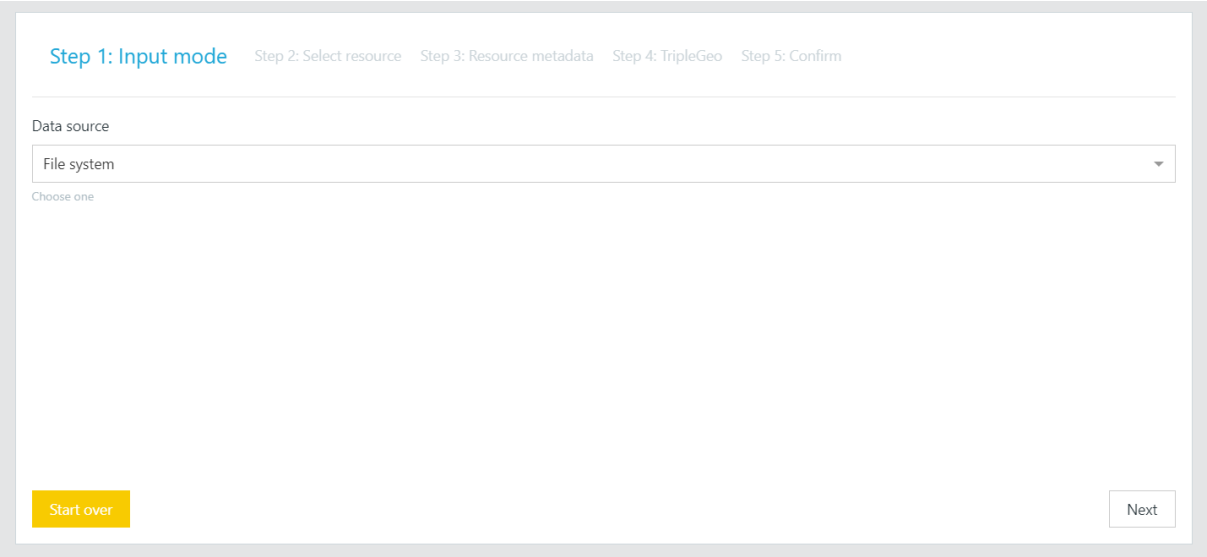


Figure 19: Resource Registration – Input mode

Next, the selected data source type is configured in "Select resource" step. Depending on the data source type, an appropriate view is rendered. In the case of the file system data source, a file browser is shown as illustrated in Figure 20.

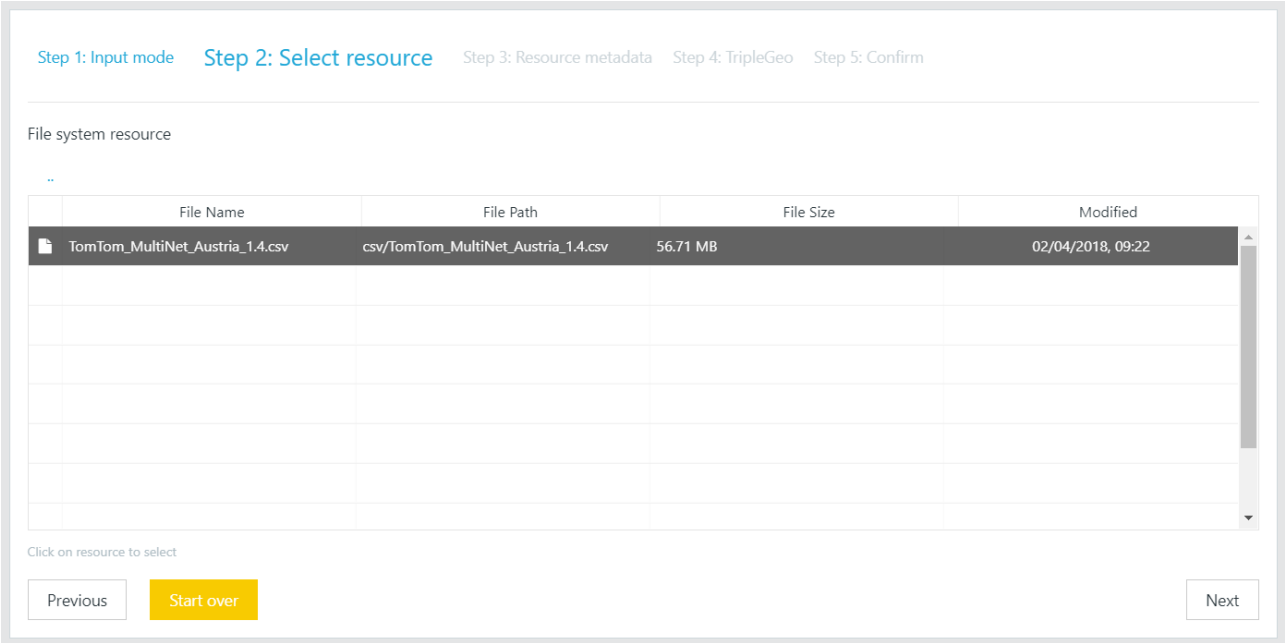


Figure 20: Resource Registration - File selection

Next, a user-friendly name and a description is assigned to the resource in the "Resource metadata" step. The wizard provides appropriate messages to prevent user from entering invalid parameters as depicted in Figure 21.

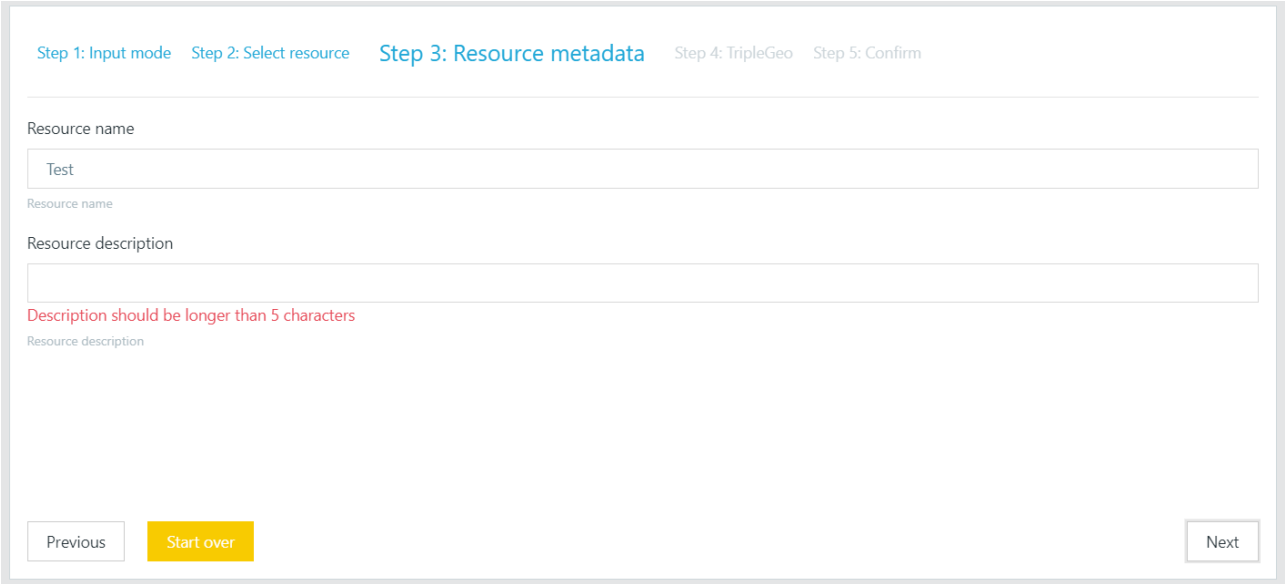


Figure 21: Resource Registration - Resource metadata

In the final step, the user configures the parameters for the TripleGeo operation that will transform the input data file to an RDF dataset that adheres to the SLIPO ontology. The configuration form is the same as the one used by the workflow designer and is displayed in Figure 22.

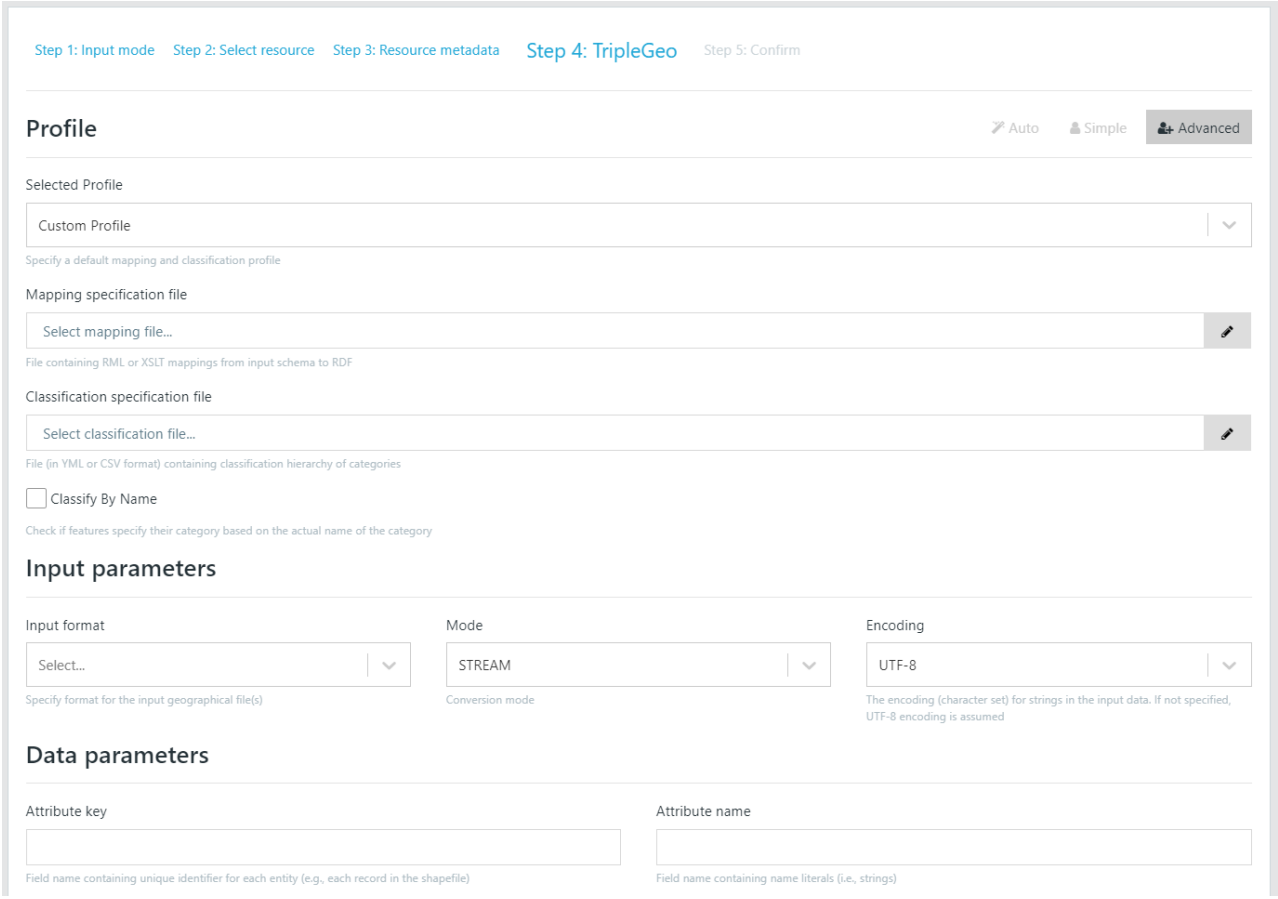


Figure 22: Resource Registration – TripleGeo

Once all parameters are set, a final confirmation step is displayed that summarizes all user input before submitting the request for processing.

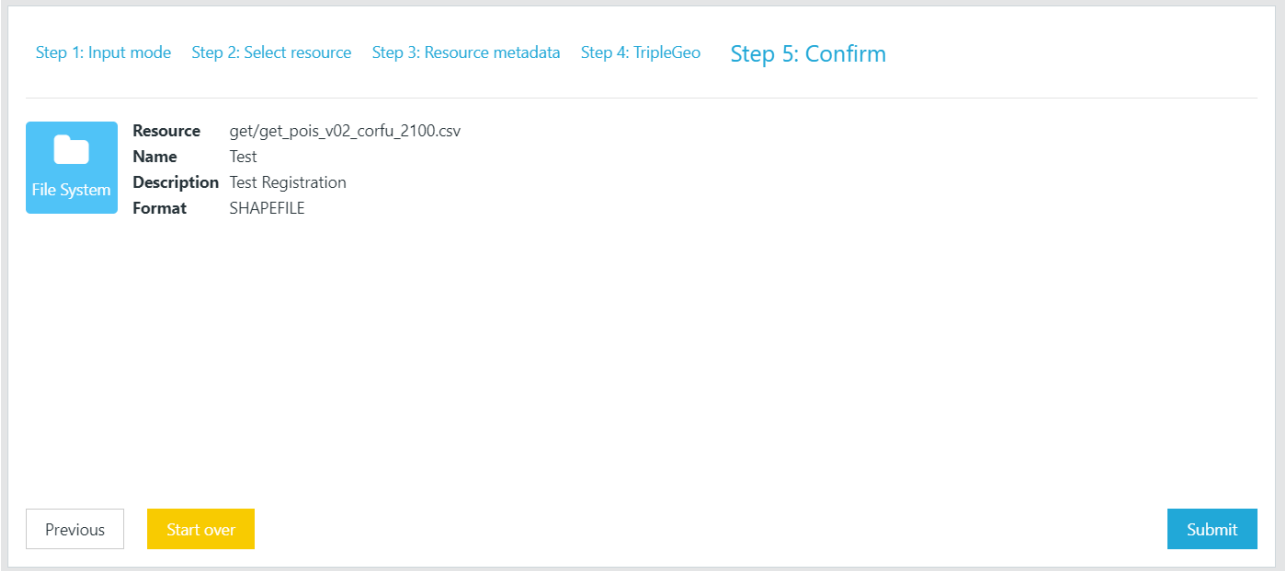


Figure 23: Resource Registration - Overview

Resource catalog data is stored in files encoded with the N-Triples RDF format. The application provides a wizard for exporting resource catalog data to several formats such CSV, Shapefile etc., using the Reverse TripleGeo transformation. The wizard consists of three steps:

- Resource selection
- Reverse TripleGeo configuration
- Confirmation

The "Select resource" step allows the user to select a single resource from the catalog to export. Moreover, the user can invoke the wizard directly from the resource catalog and have the step configured automatically. An example of the step is displayed in Figure 24.

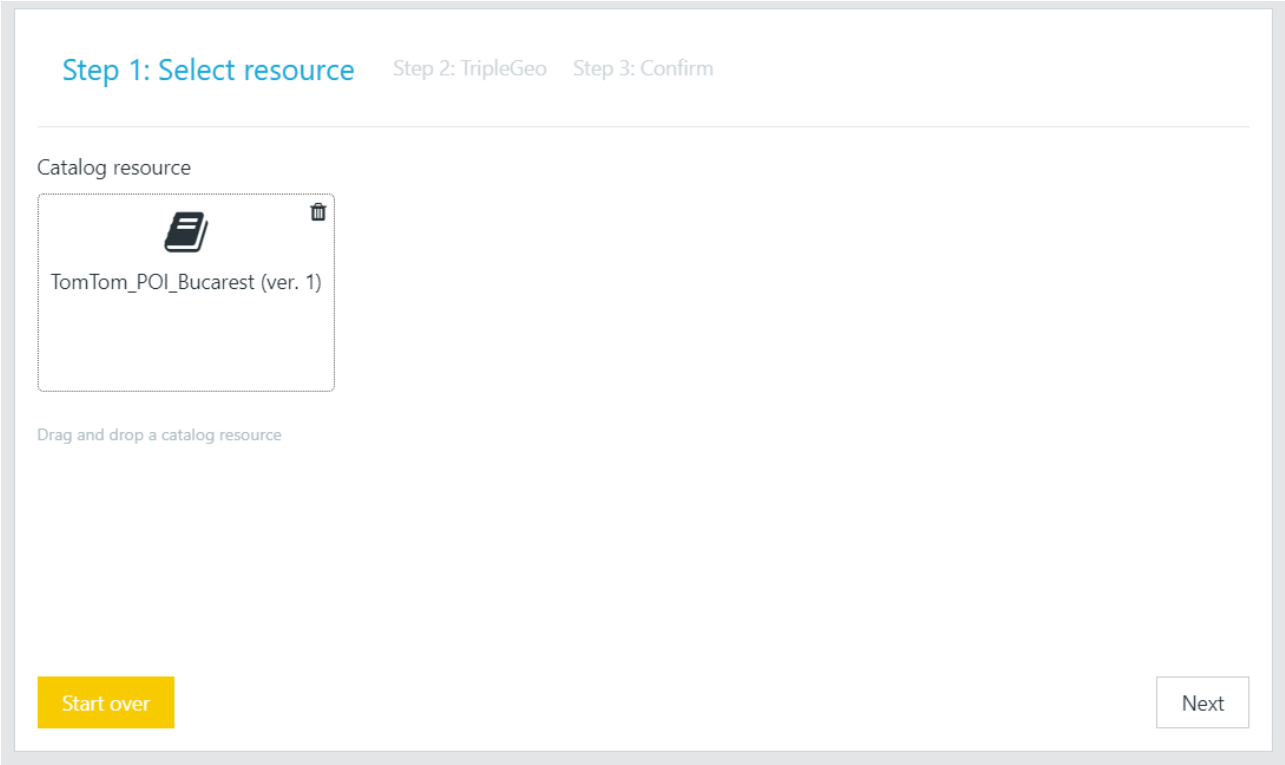


Figure 24: Export Resource – Select Resource

After a resource is selected, the user configures the parameters for the Reverse TripleGeo operation that will transform the input RDF data file to the selected output format as depicted in Figure 25. Finally, a confirmation step is displayed that summarizes all user input before submitting the request for processing.

Step 1: Select resource Step 2: TripleGeo Step 3: Confirm

Profile ▲ Advanced !

Selected Profile

TomTom Multinet

Specify a default SPARQL query

SELECT query

./test/conf/slipo/TomTom_MultiNet_reconstruct.sparql

File containing a user-specified SELECT query (in SPARQL) that will retrieve results from the input RDF triples. This query should conform with the underlying ontology of the input RDF triples.

Output parameters

Output format

CSV

Specify format for the output geographical file(s)

Encoding

UTF-8

The encoding (character set) for strings in the output data. If not specified, UTF-8 encoding is assumed.

Data parameters

Delimiter

|

Specify the character delimiting attribute values

Quote

"

Specify quote character for string values

Figure 25: Export Resource – TripleGeo

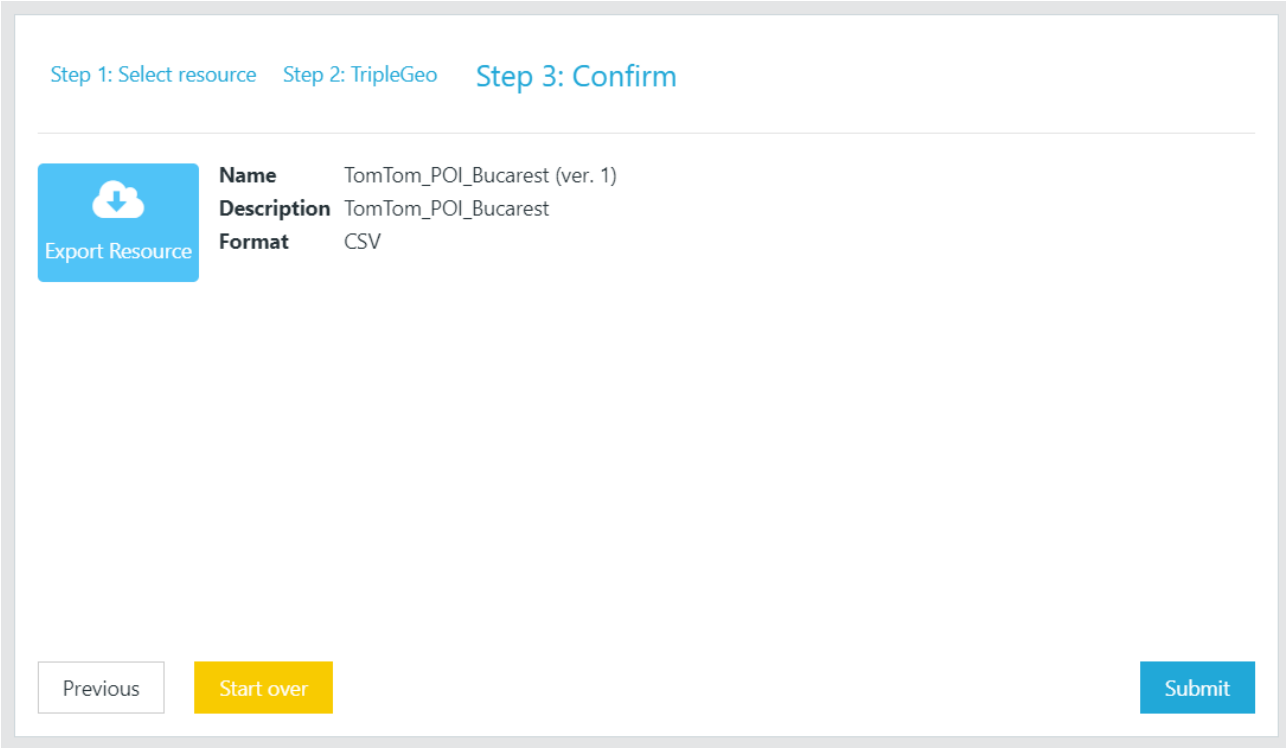


Figure 26: Export Resource - Overview

2.1.3. User File System

Most of the time, while creating a new workflow, the user must upload several files for configuring SLIPO Toolkit component operations. Such files include configuration files as well data files. In order to simplify the file management, the application provides a view for managing the user file system as displayed in Figure 27.

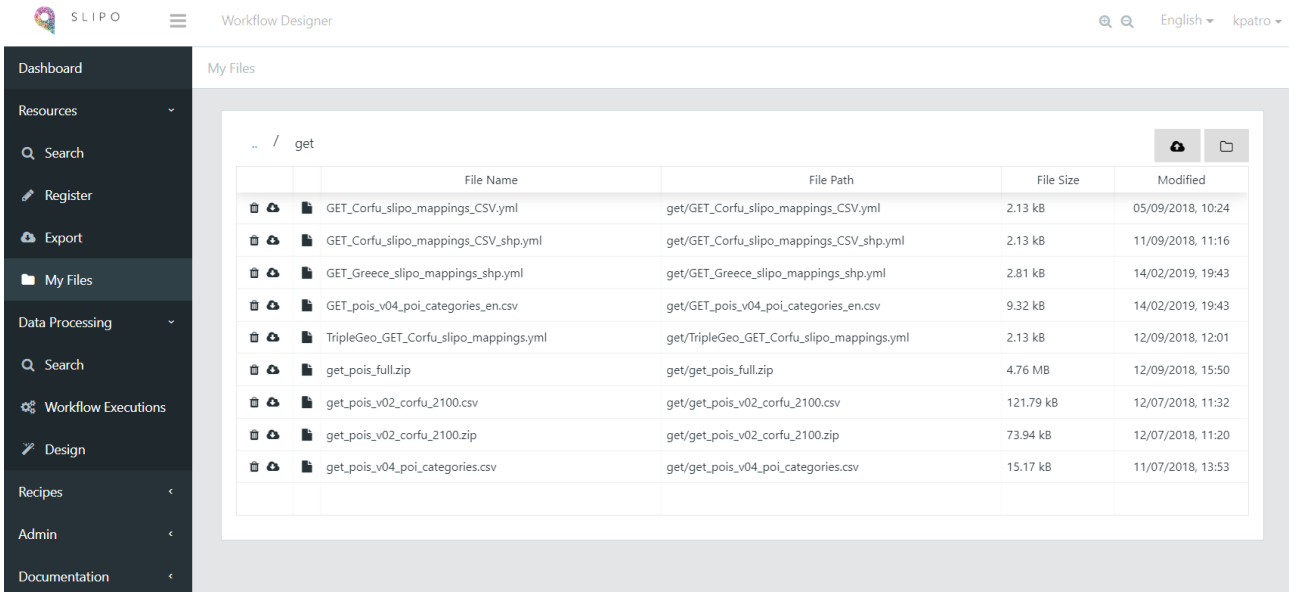


Figure 27: User File System

The file system view allows the user to perform the following actions:

- Browse the file system
- Delete a file or an empty folder
- Download a file
- Create a new folder
- Upload a new file

Once a file is uploaded, it becomes available in the workflow designer and can be selected as input for a SLIPO Toolkit component operation.

2.1.4. Data Integration workflows

The design, execution and monitoring of data integration workflow comprises the main functionality of the Workbench application. In the next sections we present the user interface elements used for browsing existing workflows, monitoring the execution of workflows managed by the SLIPO Service, and designing new data integration workflows.

2.1.4.1. Workflow

The workflow browser, depicted in Figure 28, presents all workflow definitions managed by the Workbench application. Users can filter workflows by name and task type. The SLIPO system supports five task types, namely, "Data Integration", "Registration", "Export Resource", "Export map" and "API". Only the four first options can be selected, with the "API" task type used internally by the SLIPO system for implementing the RESTful API.

Each record can be expanded to display all available versions for a specific workflow. Users can select to view, edit or execute a workflow. If a workflow is already running, the option to stop the execution is also available. Stopping an execution does not guarantee to immediately terminate the underlying process execution but informs the SLIPO Service to stop execution on first chance.

Selecting a single workflow record displays details about the execution history of the specific workflow version as shown in Figure 29. Users can either view additional execution details or browse data on a map by selecting the corresponding actions in the row. The latter option is only available if the execution input and output data has been successfully imported to PostGIS.

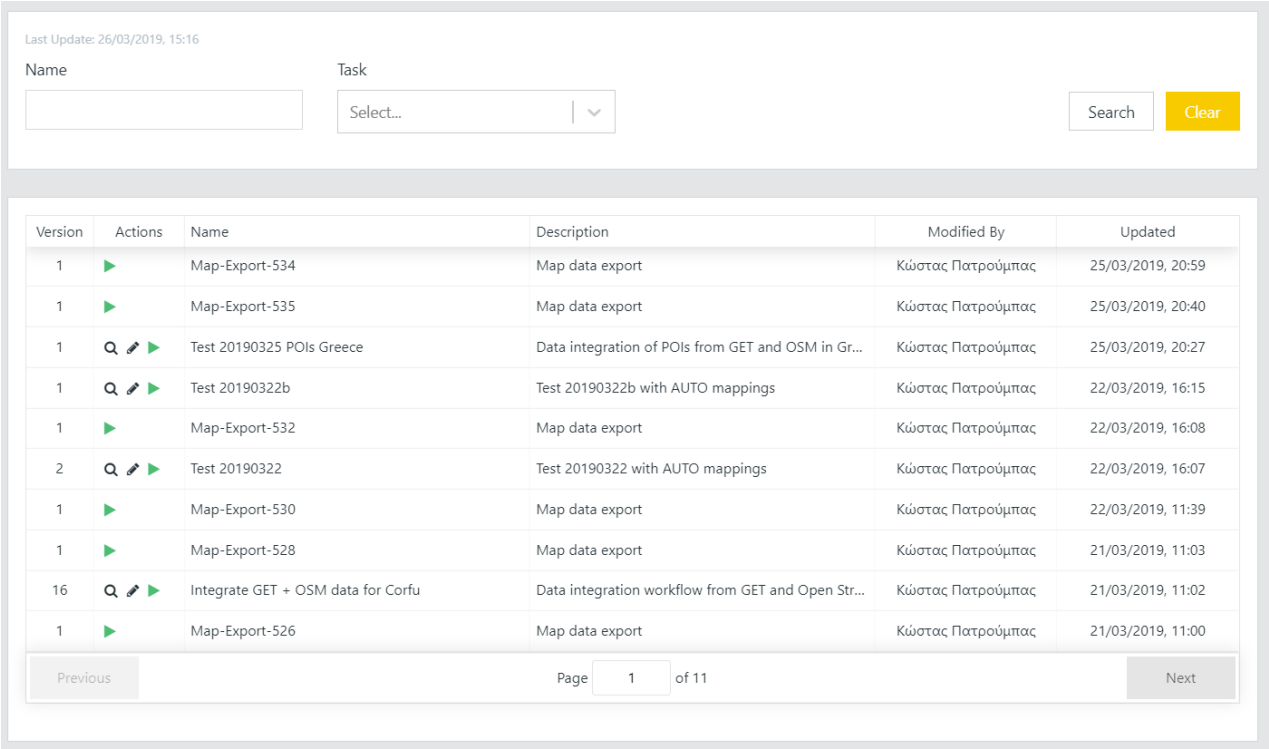


Figure 28: Workflow browser

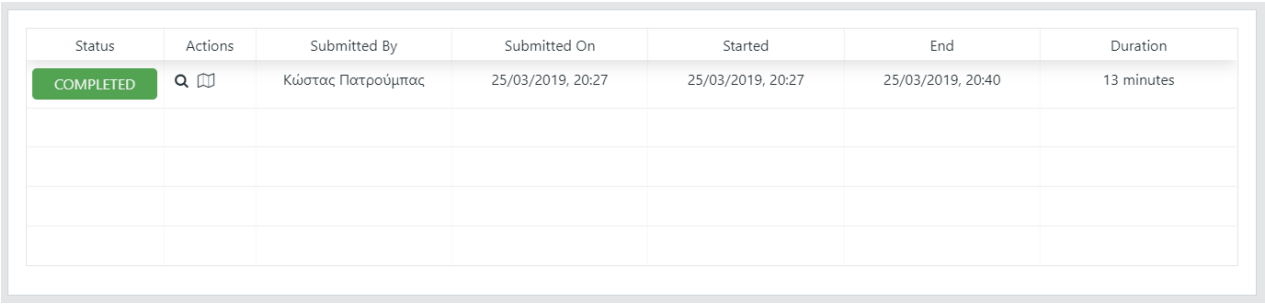


Figure 29: Workflow execution details

2.1.4.2. Workflow Executions

The workflow executions view presents historical data for all executions managed by the SLIPO Service. Users can filter records by workflow name, execution status and task type. Each record refers to a specific workflow version and supports editing the workflow, viewing execution details and browsing data on a map. An example of the view is illustrated in Figure 30.

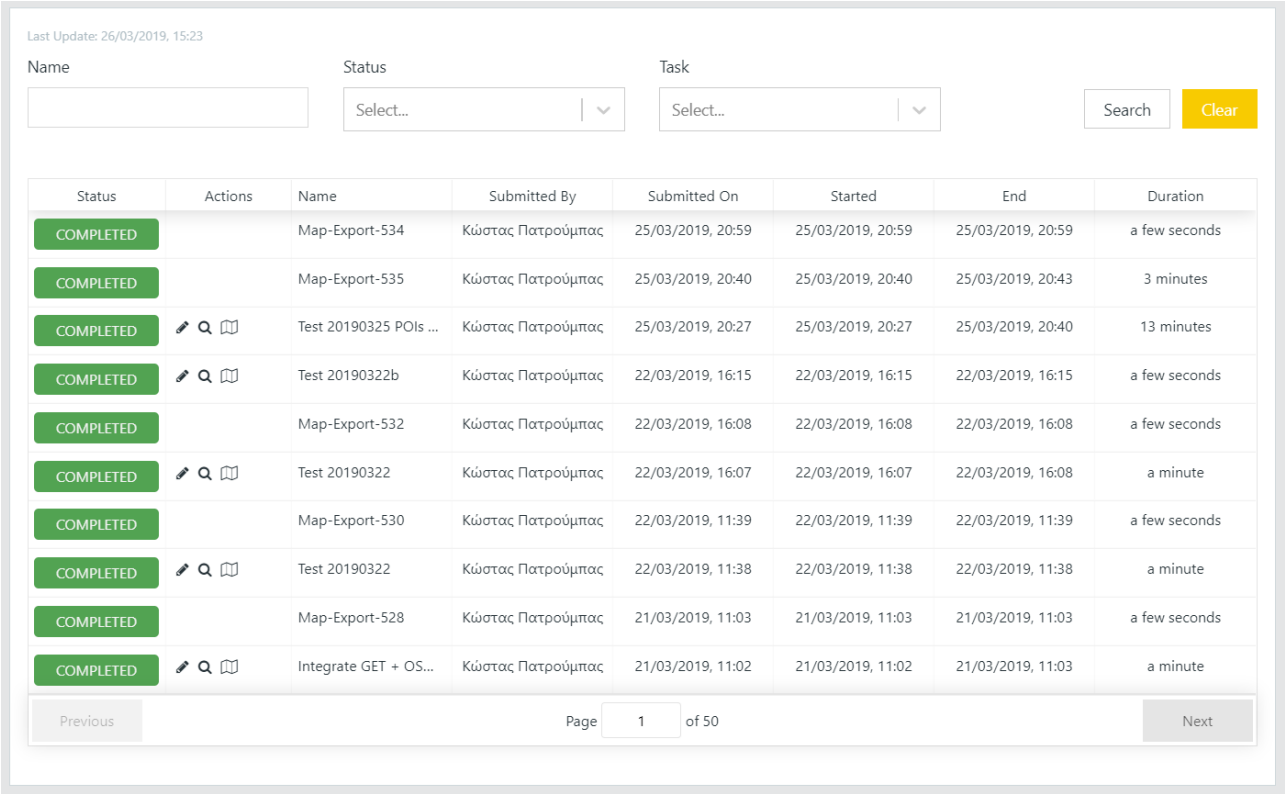


Figure 30: Workflow execution browser

2.1.4.3. Workflow Designer

The workflow designer is the central view of the Workbench application as it implements the functionality required for designing data integration workflows. The view consists of four main sections as show in Figure 31.

On the top, there is a list of all supported operations used for designing a workflow. Each operation belongs to a group based on its functionality. The main groups are:

- **Data Sources**: Used for defining the input to TripleGeo transformation operations.
- **SLIPO Toolkit**: Contains all the operations provided by the SLIPO Toolkit components, i.e., transformation (TripleGeo), interlinking (LIMES), fusion (FAGI) and enrichment (DEER).
- **Miscellaneous**: This group contains any operations that cannot be categorized to any of the other groups. Currently, it contains only a single item, namely "Register Resource" operation, that is used for registering workflow results to the catalog.

Below the operations section, there is a list of actions applicable to the current a workflow definition:

- **Discard**: Discards any unsaved user changes and redirects user to the workflow browser.
- **Clear**: Resets the workflow designer by removing all operations.
- **Undo**: Reverts the most recent user action.
- **Redo**: It is the opposite to undo and allows the user to restore an action previously undone.
- **Save**: Saves the current workflow and redirects the user to the workflow browser. Saving the workflow always creates a new version.

- **Save As Recipe**: Allows users to save a workflow as a recipe. This action is only available for new workflows. Existing workflows cannot be saved as recipes.
- **Save and Execute**: Saves the current workflow, schedules it for immediate execution, and redirects the user to the workflow browser.
- **Save, Execute and Create Map**: Saves the current workflow, schedules it for immediate execution, and on successful completion initializes a second workflow for importing input and output data to PostGIS.

At the center of the view, there is the workflow design area. The design area is separated into vertical lanes. Users can drag and drop operations into lanes to design a workflow. The first lane accepts only transformation operations. Initially, there are only two lanes available. As users add new operations, additional lanes are automatically added.

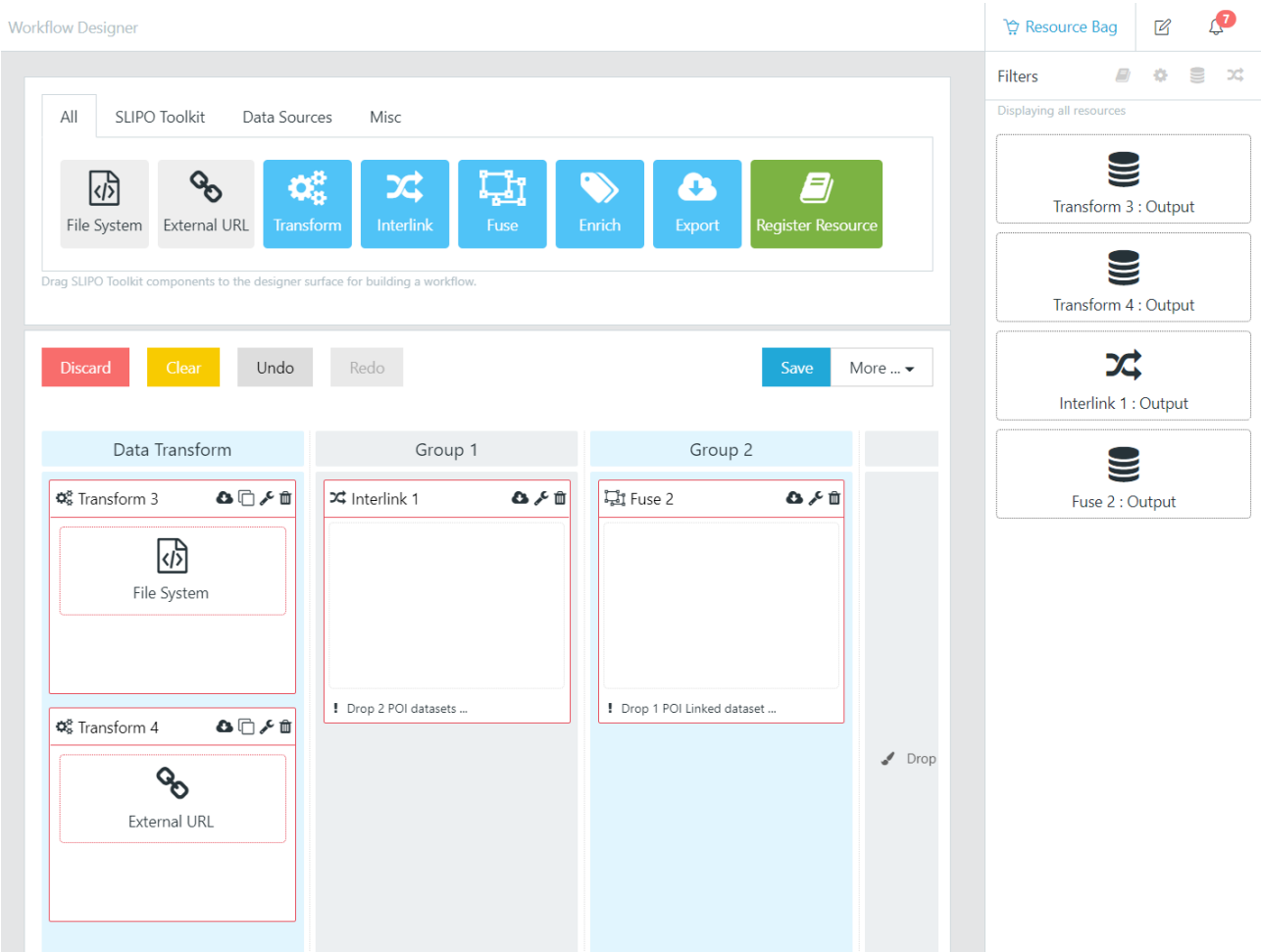


Figure 31: Workflow designer

Finally, on the right sidebar, additional information for the workflow is presented. The information is grouped into three sections, namely, resources, properties and messages, displayed in Figure 32, Figure 33 and Figure 34 respectively.

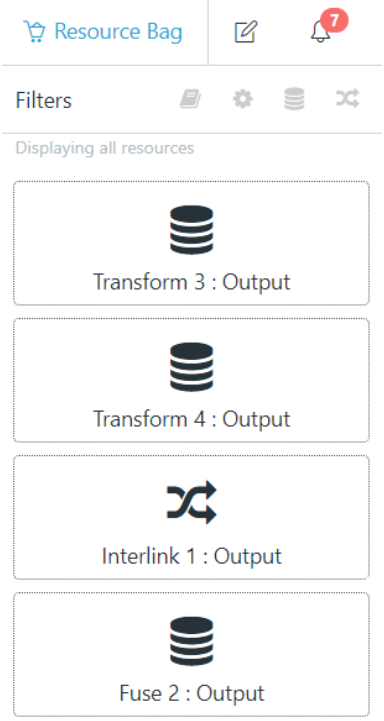


Figure 32: Resources

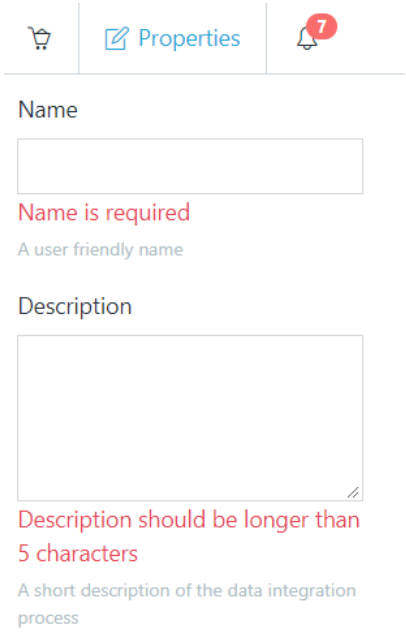


Figure 33: Properties

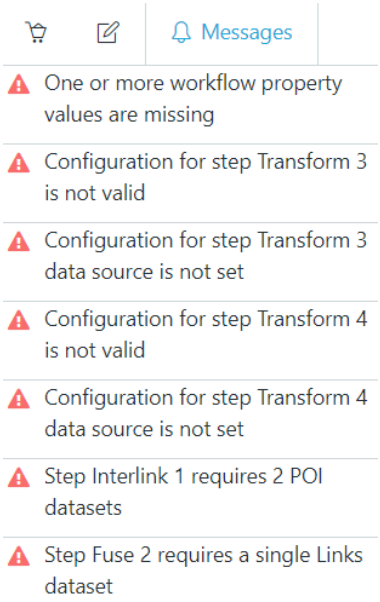


Figure 34: Messages

The resources section contains all the resources that can be used by operations in the current workflow. These include both the resources selected from the resource catalog, and the output of any operation that can produce one or more RDF datasets. The resources can be filtered by type (i.e., POI or Link data) and source (i.e., catalog or operation output).

The properties section contains a simple form for entering a user-friendly workflow name along with a short description. The workflow name must be unique and can only be set during the workflow creation. After creating a workflow, the name property becomes read-only.

Finally, the messages section contains validation messages for the current state of the workflow. The list of messages is dynamically updated after any change to the workflow occurs. Users cannot save a workflow unless there are no validation errors. The only exception is when the user selects to save a workflow as a recipe. In this case, several validation messages are suppressed, thus allowing to save partial workflows which can be used as templates later.

Moreover, every operation in a workflow requires one or more inputs. The designer supports three types of inputs, namely, POI datasets, Links datasets and external data sources. Users can assign inputs to an operation by either dragging a resource or an output dataset from the sidebar or selecting a data source component from the toolbar. If a SLIPO Toolkit component generates more than one output dataset, the designer supports the selection of a specific output to be used as input. An example of input selection is displayed in Figure 35.

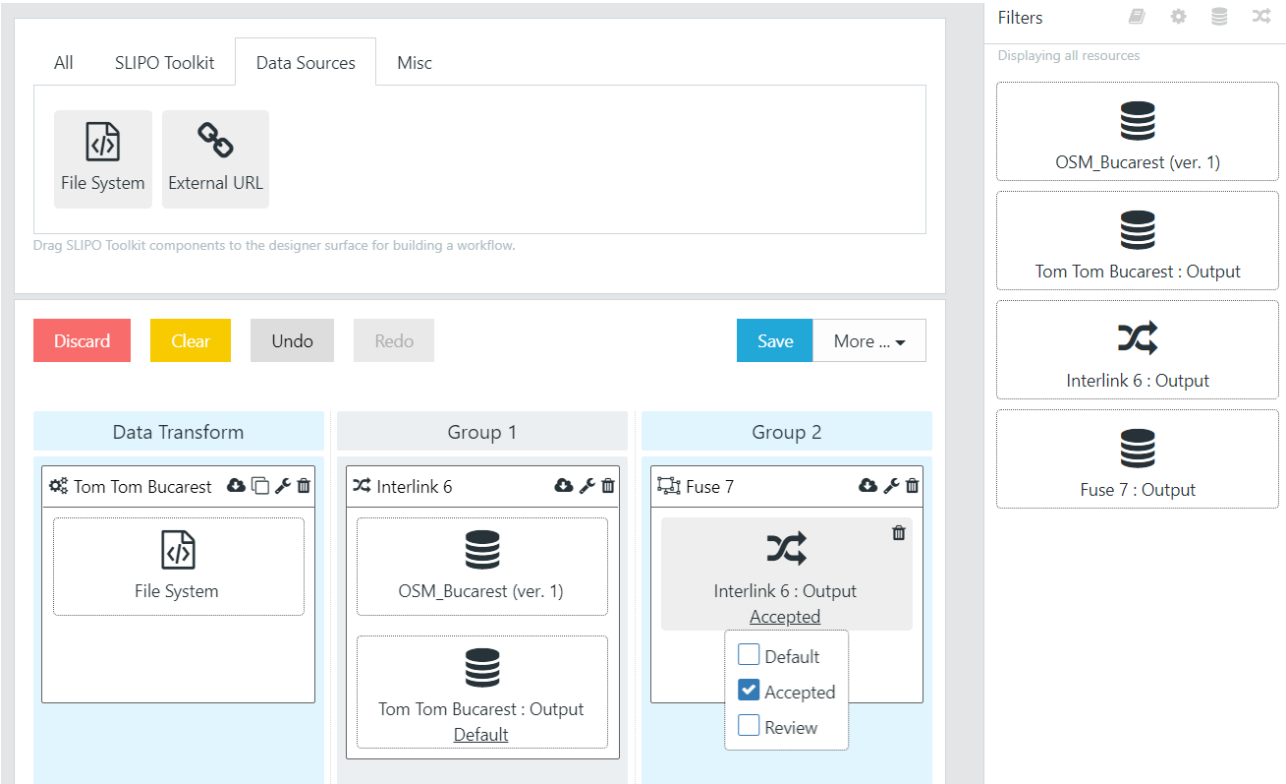


Figure 35: Input selection

Finally, for every operation and data source one or more configuration options must be provided. In the next sections we present an overview of the configuration options of each SLIPO Toolkit component and supported data source type.

2.1.4.4. Data Source Configuration

The workflow designer supports two types of data sources, namely, the **File System** and **External URL**. Data sources are supported only by the TripleGeo component and facilitate the usage of external data files in a POI data integration workflow. The **File System** data source requires users to select a single file from the

user's virtual file system stored at the sever, while the **External URL** allows the downloading of an external resource at the server using an accessible URI. Examples of the configuration options for each data source type are displayed in Figure 36 and Figure 37 respectively.

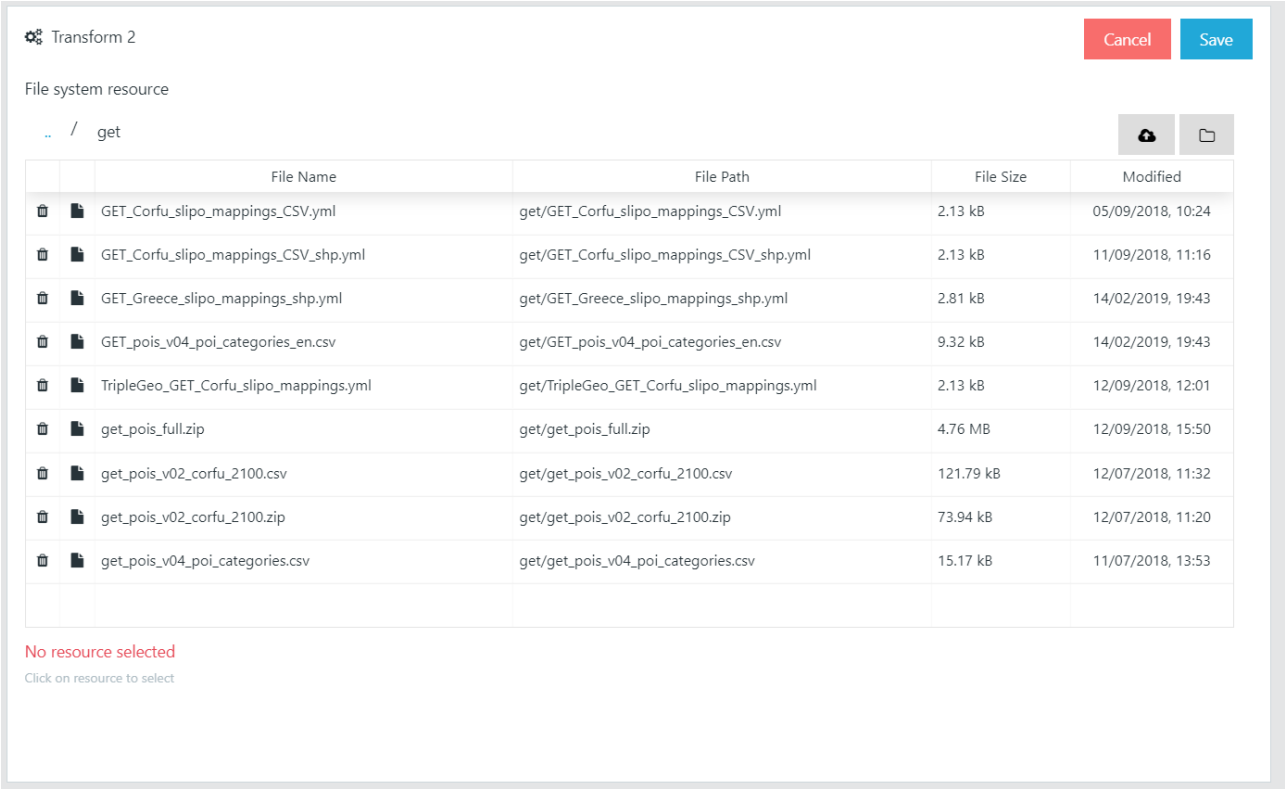


Figure 36: File System data source configuration

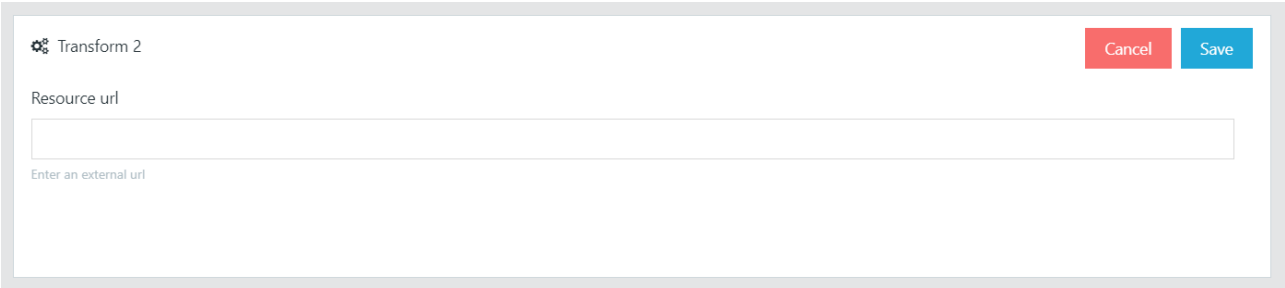


Figure 37: External URL data source configuration

2.1.4.5. Transform Configuration

The designer provides three configuration modes for TripleGeo, namely, **Auto**, **Simple** and **Advanced**. The simplest configuration mode is **Auto**, with **Simple** and **Advanced** modes adding additional configuration options. An example of the **Advanced** mode is displayed in Figure 38. In this mode the user can edit all

TripleGeo configuration options and upload custom mapping and classification files. The designer allows the user to initialize options by selecting an existing profile or set all options manually.

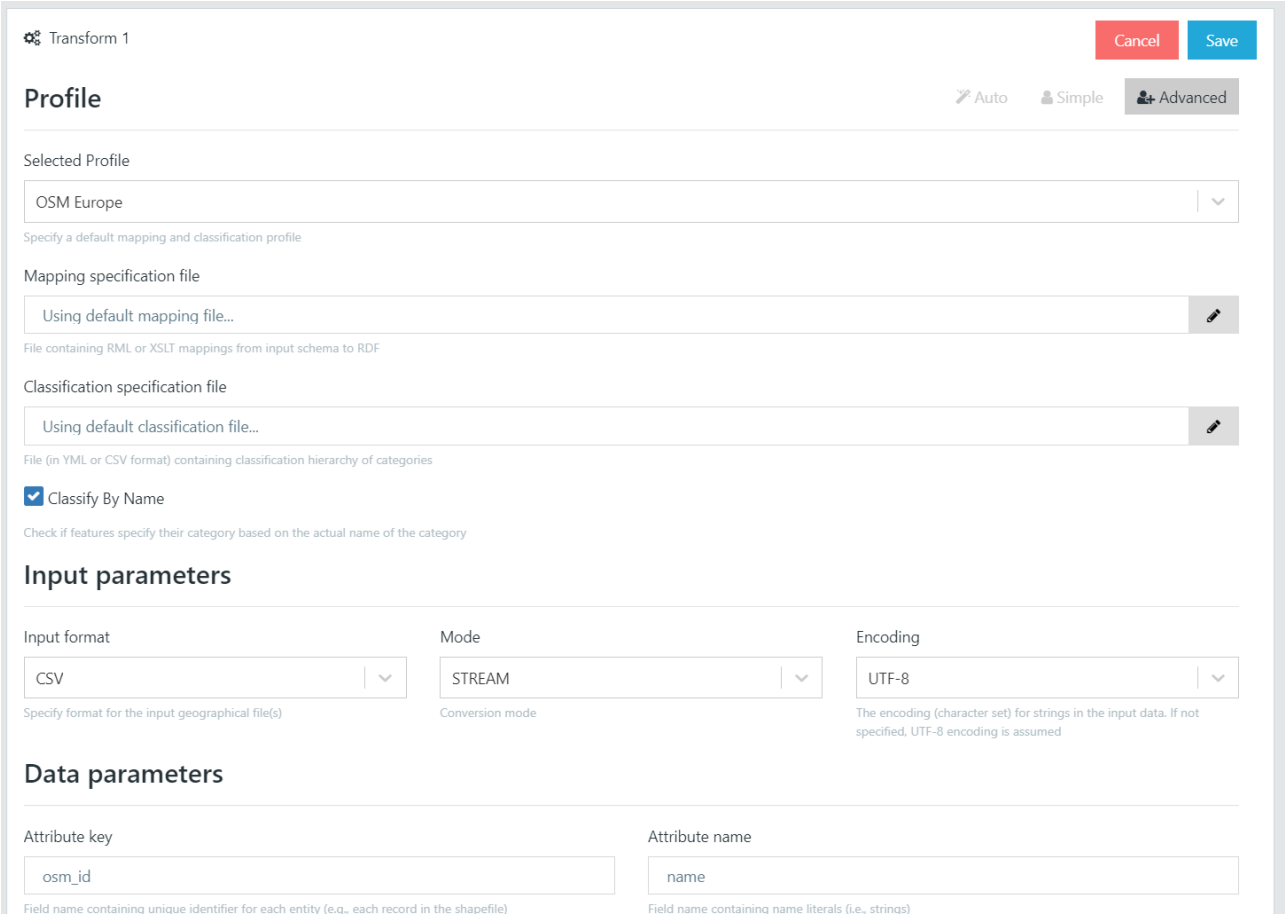


Figure 38: TripleGeo advanced configuration

In **Simple** mode, shown in Figure 39, the user is required to select a pre-configured profile which automatically sets mapping and classification files and initializes several other options. The user may update specific options but cannot change the mapping or classification files.

Auto mode, an example of which is depicted in Figure 40, is supported only for UTF-8 encoded CSV files and automatically creates a mapping file using a machine learning generated model. By pressing the **Mappings** button, the user is presented with the form illustrated in Figure 41. The predicate drop-down lists are initialized with the values suggested by the model. For every value a score is also displayed. Initially the predicate with the highest score is selected, but the user can change any of the predicate values. Finally, the user can preview and even edit, if she has the appropriate permissions, the generated mapping file as shown in Figure 42.

Transform 1

Cancel Save

Profile

Auto Simple Advanced

Select Profile ...

Required

Specify a default mapping and classification profile

Input parameters

Input format

Select...

Required

Specify format for the input geographical file(s)

Encoding

UTF-8

The encoding (character set) for strings in the input data. If not specified, UTF-8 encoding is assumed

Data parameters

Attribute key

Required

Field name containing unique identifier for each entity (e.g., each record in the shapefile)

Attribute name

Field name containing name literals (i.e., strings)

Figure 39: TripleGeo simple configuration

Transform 1

Cancel Save

Mappings

! Mappings Auto Simple Advanced

Classification specification file

Select classification file...

Required

File (in YML or CSV format) containing classification hierarchy of categories

Input parameters

Encoding

UTF-8

The encoding (character set) for strings in the input data. If not specified, UTF-8 encoding is assumed

Spatial Reference parameters

Source CRS

4326

Specify the EPSG numeric code for the source CRS

Target CRS

4326

Specify the EPSG numeric code for the target CRS

Other parameters

Default language

English

Default lang for the labels created in the output RDF. By default, the value will be English-en

Figure 40: TripleGeo auto configuration

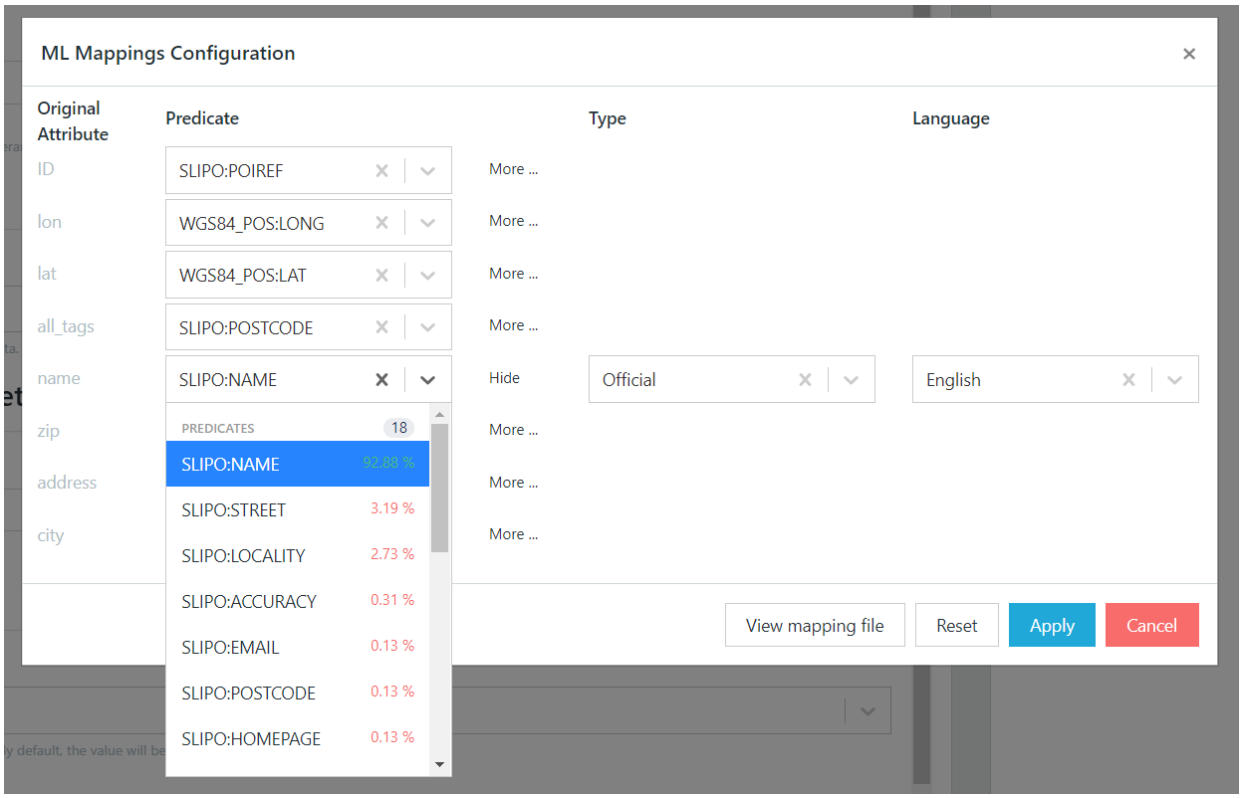


Figure 41: TripleGeo auto mappings configuration

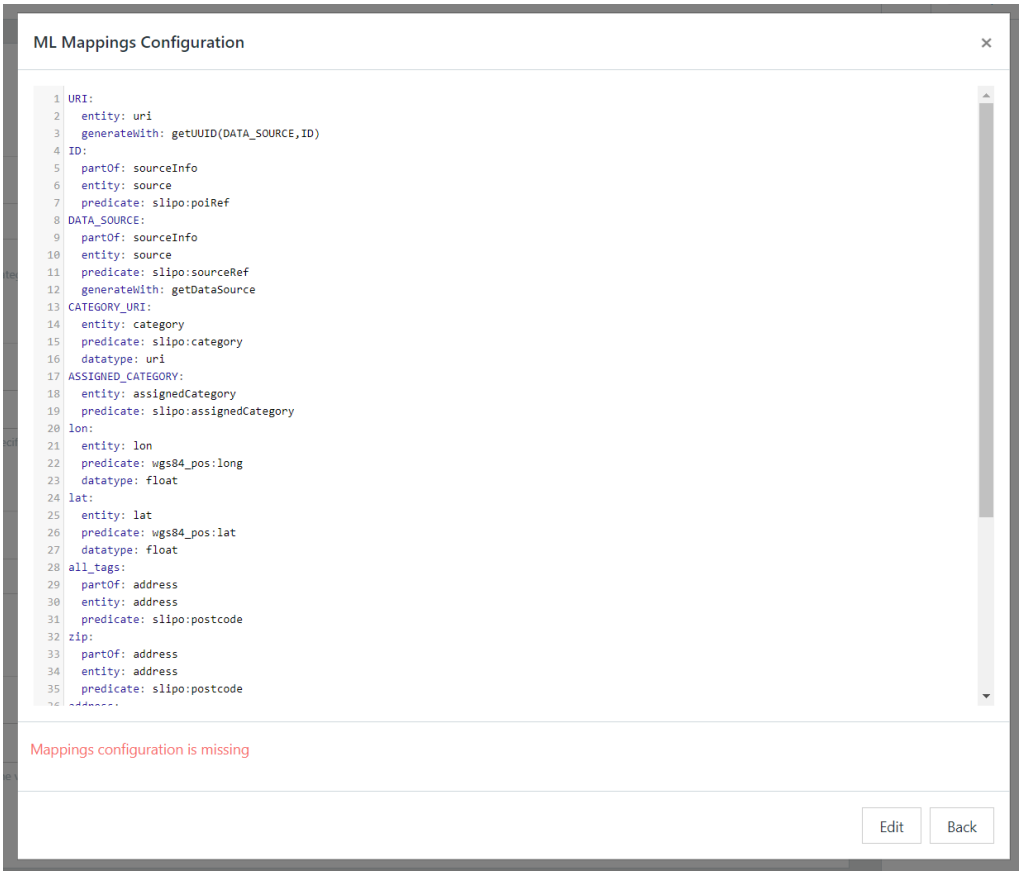


Figure 42: TripleGeo auto mappings editor

2.1.4.6. Interlink Configuration

The workflow designer supports two configuration modes for LIMES, specifically, **Auto** and **Advanced**, shown in Figure 43 and Figure 44 respectively. In **Auto** mode, a pre-defined profile is used for populating LIMES configuration options, while in **Advanced** mode the user is required to select a profile manually.

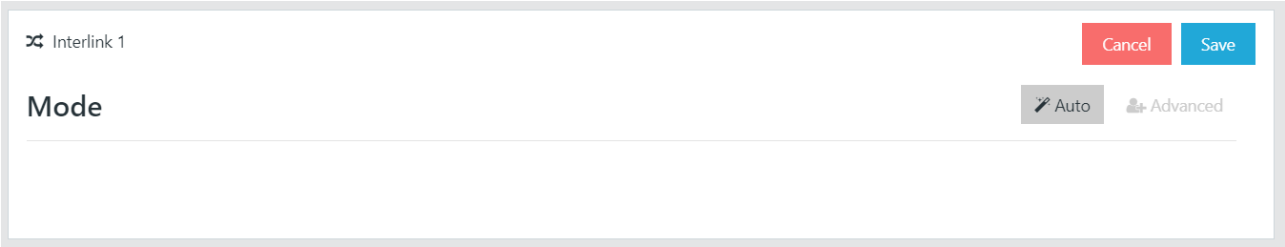


Figure 43: LIMES auto configuration

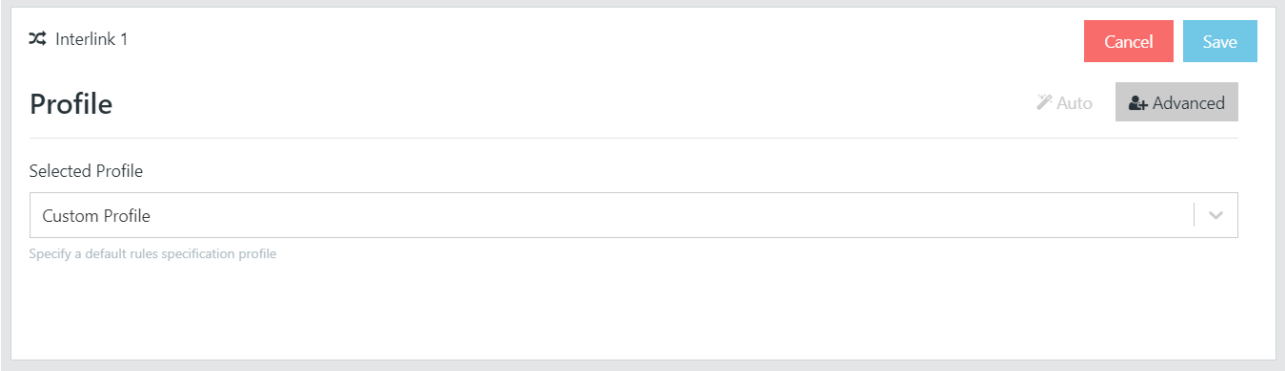


Figure 44: LIMES advanced configuration

2.1.4.7. Fusion Configuration

Like LIMES, the workflow designer supports two configuration modes for FAGI, the **Auto** and **Advanced** modes. An example of the options for each mode is depicted in Figure 45 and Figure 46 respectively. In **Auto** mode, a pre-defined profile is used for populating FAGI configuration options, while in **Advanced** mode the user is required to select a profile manually. Moreover, the designer allows the user to upload a custom rule specification file.

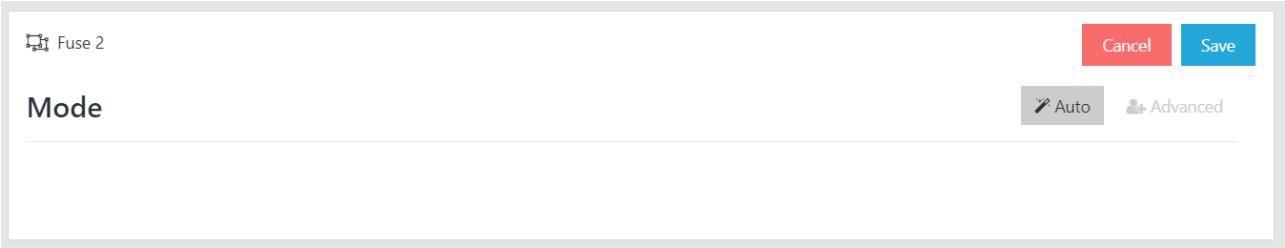


Figure 45: FAGI auto configuration

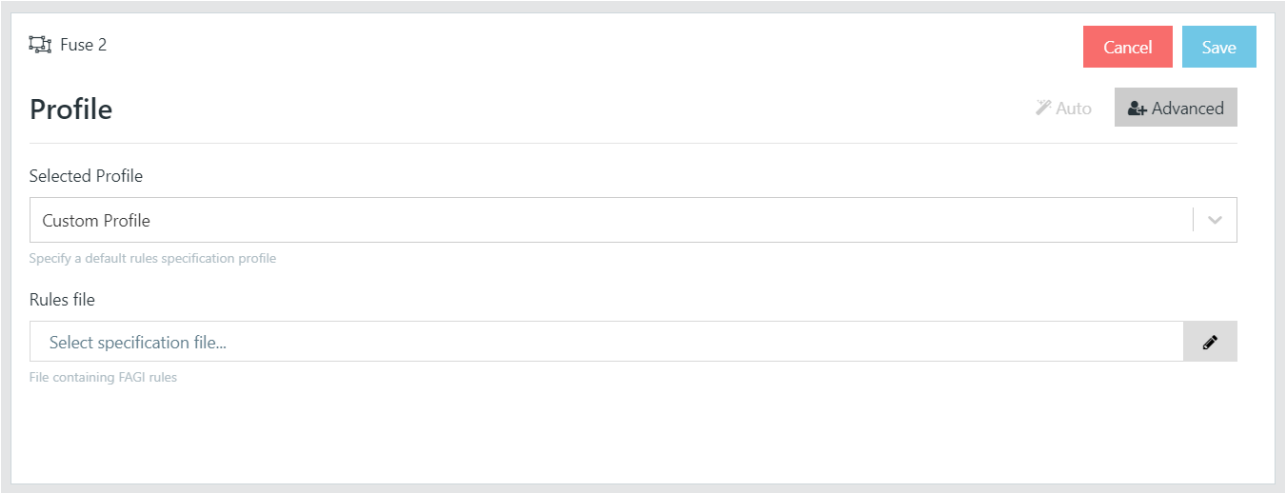


Figure 46: FAGI advanced configuration

2.1.4.8. Enrichment Configuration

DEER has two configuration modes. In **Auto** mode, a pre-defined profile is used for initializing all DEER configuration options, while in **Advanced** mode, the user is required to select a profile manually. In the latter case, the user may also select to override the profile specification file. Examples of the options for both modes are illustrated in Figure 47 and Figure 48.

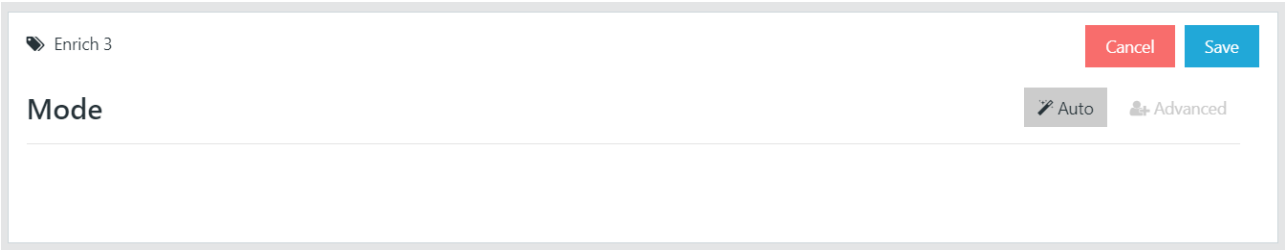


Figure 47: DEER auto configuration

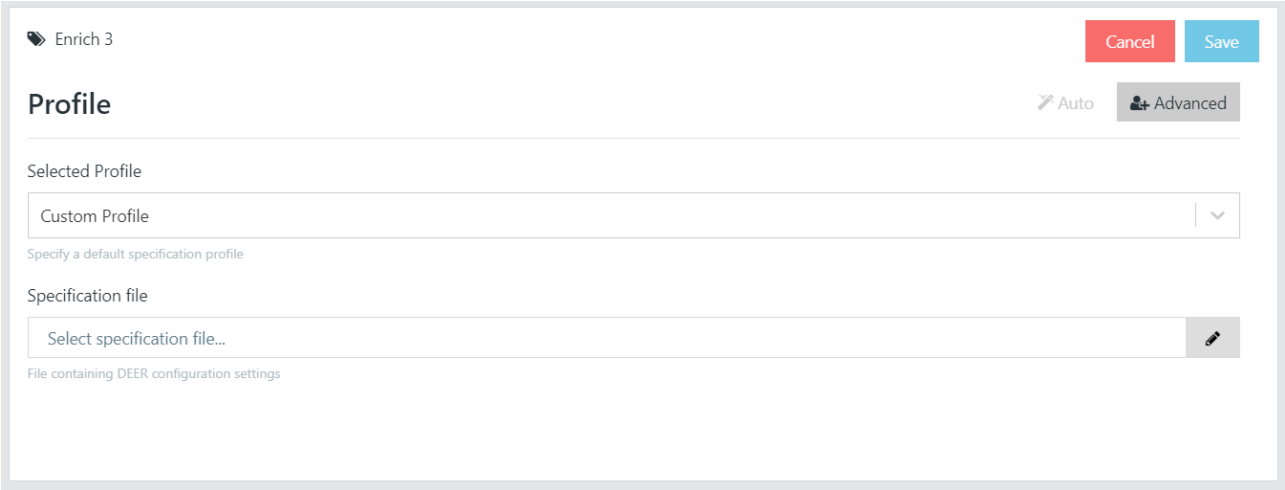


Figure 48: DEER advanced configuration

2.1.4.9. Export Configuration

Export operation refers to the TripleGeo reverse transformation and allows users to export RDF data to other formats such as CSV or Shapefile. The configuration options are displayed in Figure 49. The user can initialize the form by selecting a custom profile or set every option manually.

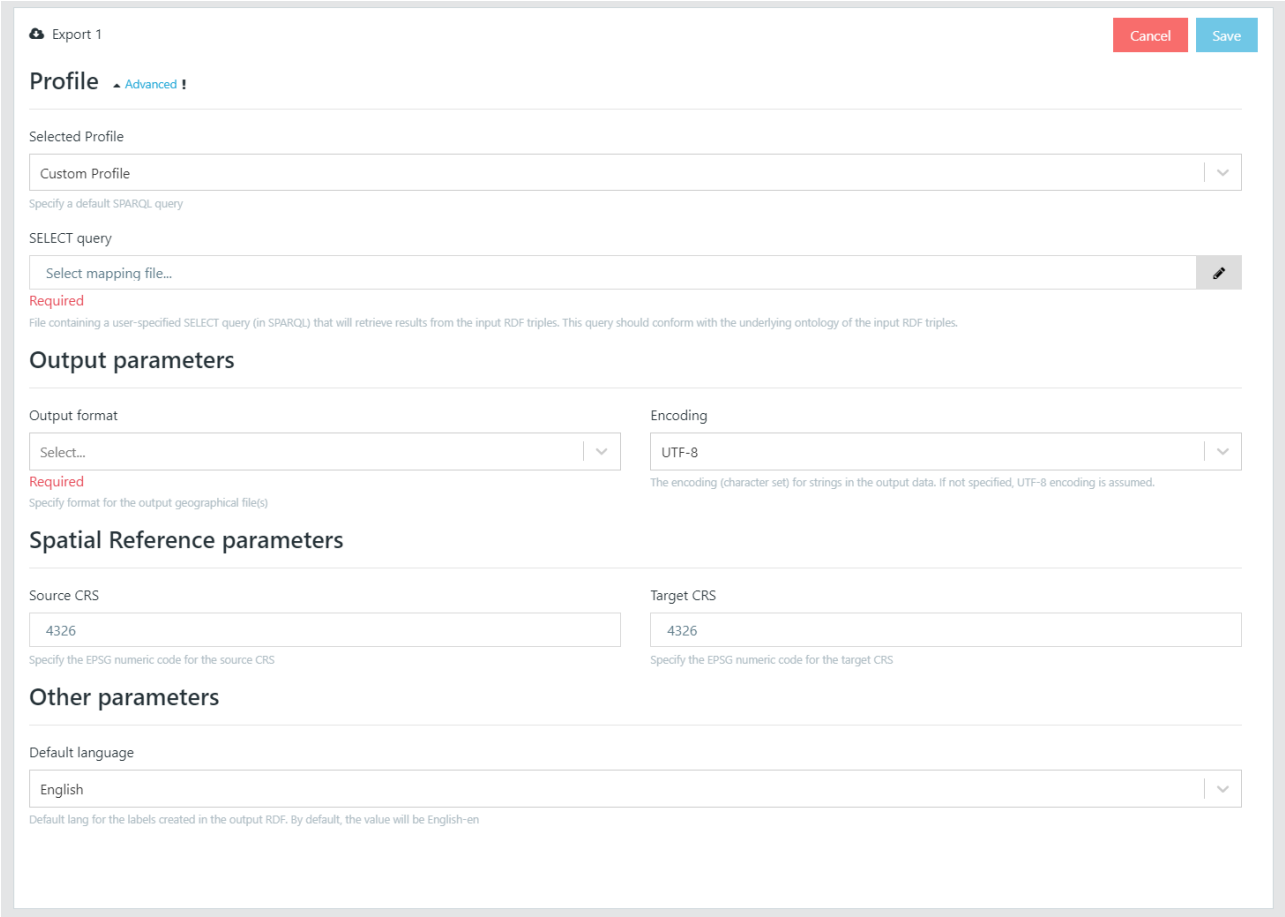


Figure 49: Reverse TripleGeo configuration

2.1.4.10. Registration Configuration

Finally, the workflow designer supports the registration of output POI datasets to the resource catalog. The configuration options are displayed in Figure 50 and are the same as the ones in the **Resource Metadata** step in the resource registration wizard described in section 2.1.2.

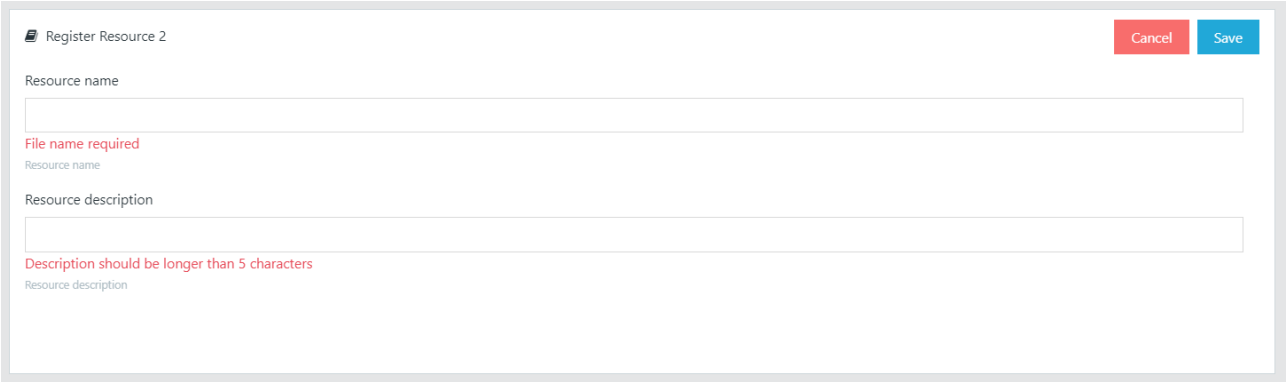


Figure 50: Register configuration

2.1.4.11. Execution Viewer

During the execution of a workflow, the SLIPO Service stores detailed data for every executed operation including all input, output and configuration files. Users can browse this data using the execution viewer. The data of an execution is updated incrementally at runtime; thus, users can even browse data about running and incomplete executions.

The execution viewer is based on a read-only version of the workflow designer. On the top of the view, there is aggregated data about the whole execution. Each operation is extended to include status information as shown in Figure 51. Selecting a single operation displays details about the selected step. For each operation the user can browse all files used from or generated by it.

The file browser for a single operation is illustrated in Figure 52 and allows users to download any of the files. Depending on the operation and type of the file, additional actions may be available. For KPI files, the user can browse data using a table component as depicted in Figure 53. For TripleGeo operations, additional information is also available, including the data bounding box and attribute transformation statistics as shown in Figure 53 and Figure 54 respectively.

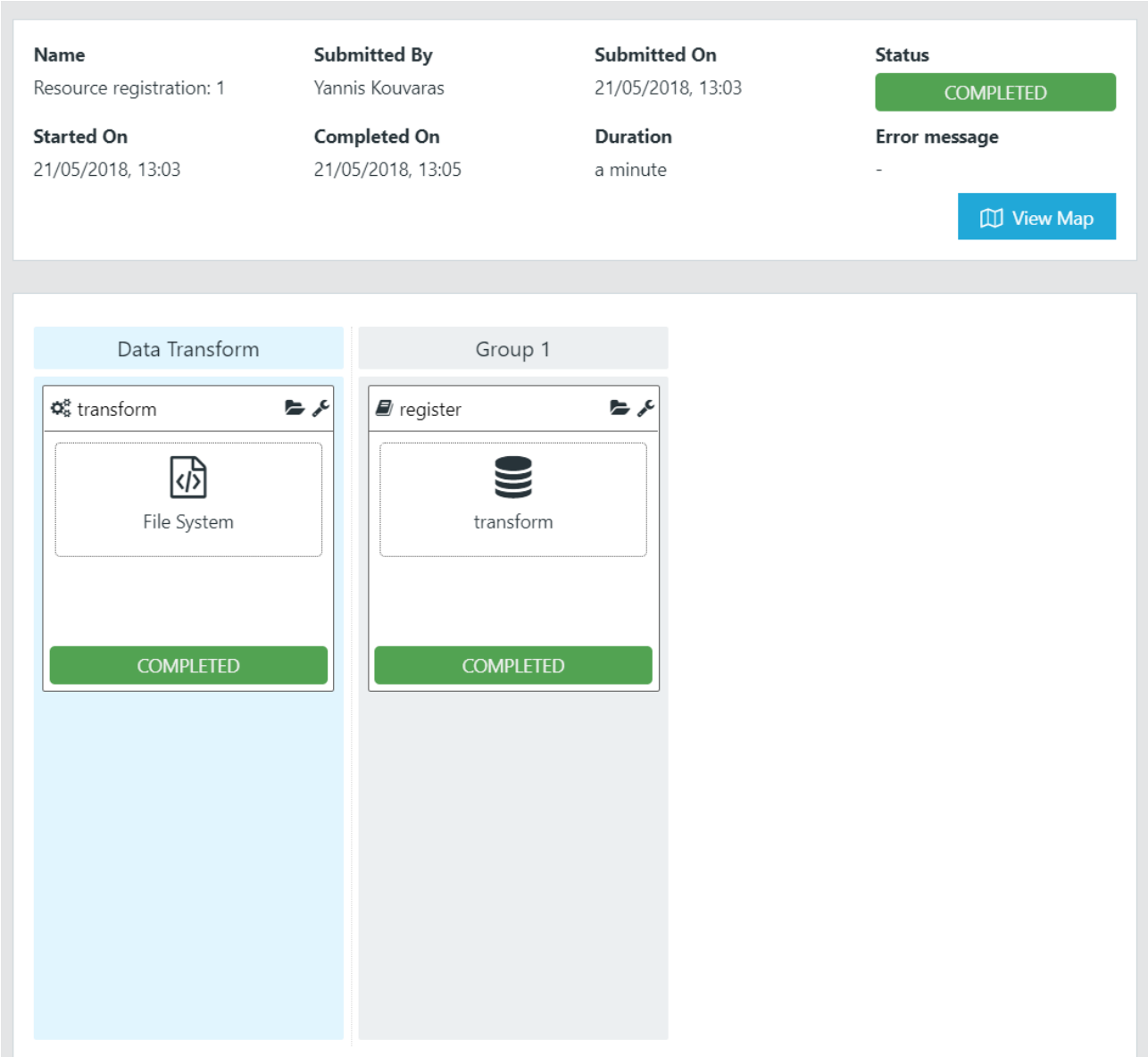


Figure 51: Workflow execution viewer

transform - Files

Actions	Type	Name	Size
	Configuration	classification.csv	36.69 kB
	Configuration	mappings.yml	2.08 kB
	Configuration	options.conf	978 bytes
	Input	TomTom_MultiNet_Austria_1.4.csv	56.71 MB
	Output	TomTom_MultiNet_Austria_1.4.nt	1847.07 MB
	Output	classification.nt	335.00 kB
	KPI	TomTom_MultiNet_Austria_1.4_metadata.json	613 bytes
	KPI	classification_metadata.json	299 bytes

Figure 52: Workflow step files

KPI File : osm_pois_v1.2_Fuel_Berlin_metadata.json

Name	Value
Attribute Statistics.osm_id	242
Attribute Statistics.country	133
Attribute Statistics.website	76
Attribute Statistics.city	161
Attribute Statistics.postcode	162
Attribute Statistics.DATA_SOURCE	242
Attribute Statistics.lon	242
Attribute Statistics.CATEGORY_URI	242
Attribute Statistics.housenumber	157
Attribute Statistics.phone	89

Previous Page 1 of 3 Next

Bounding Box

Figure 53: TripleGeo KPI data and bounding box

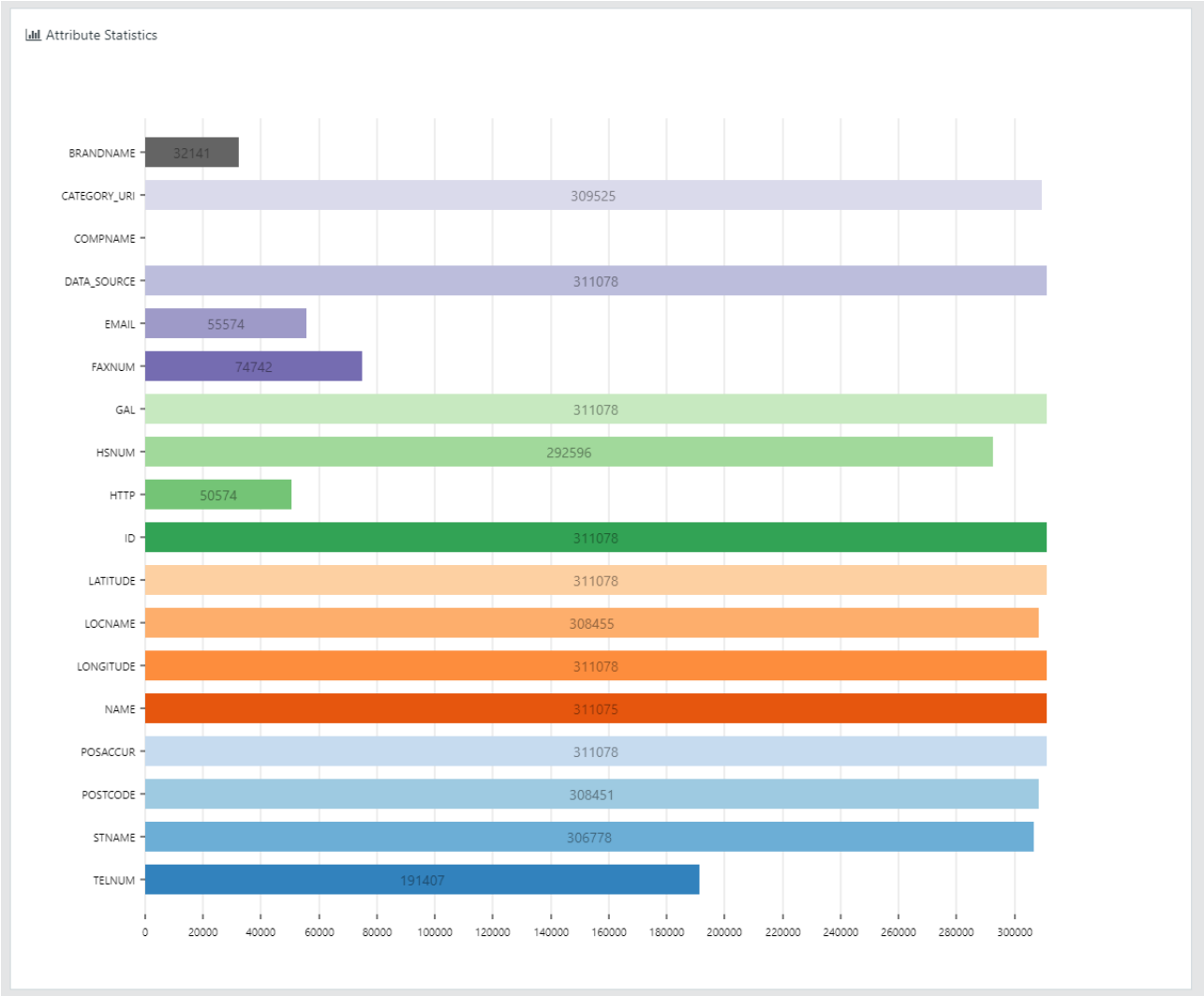


Figure 54: TripleGeo attribute statistics

2.1.4.12. Map Viewer

After a workflow execution is completed, the user may request the import of all input and output data to PostGIS. Every file is imported in a separate database table which in turn is published as a WFS layer. These layers are used by the map viewer component to render a thematic map specific to a single execution as show in Figure 55.

The map viewer supports the following actions:

- Selecting one or more features and viewing their properties in a floating window.
- Selecting the map base layer from a drop-down list as shown in Figure 56.
- Disabling a layer as shown in Figure 57.
- Changing the style of a layer as shown in Figure 58.

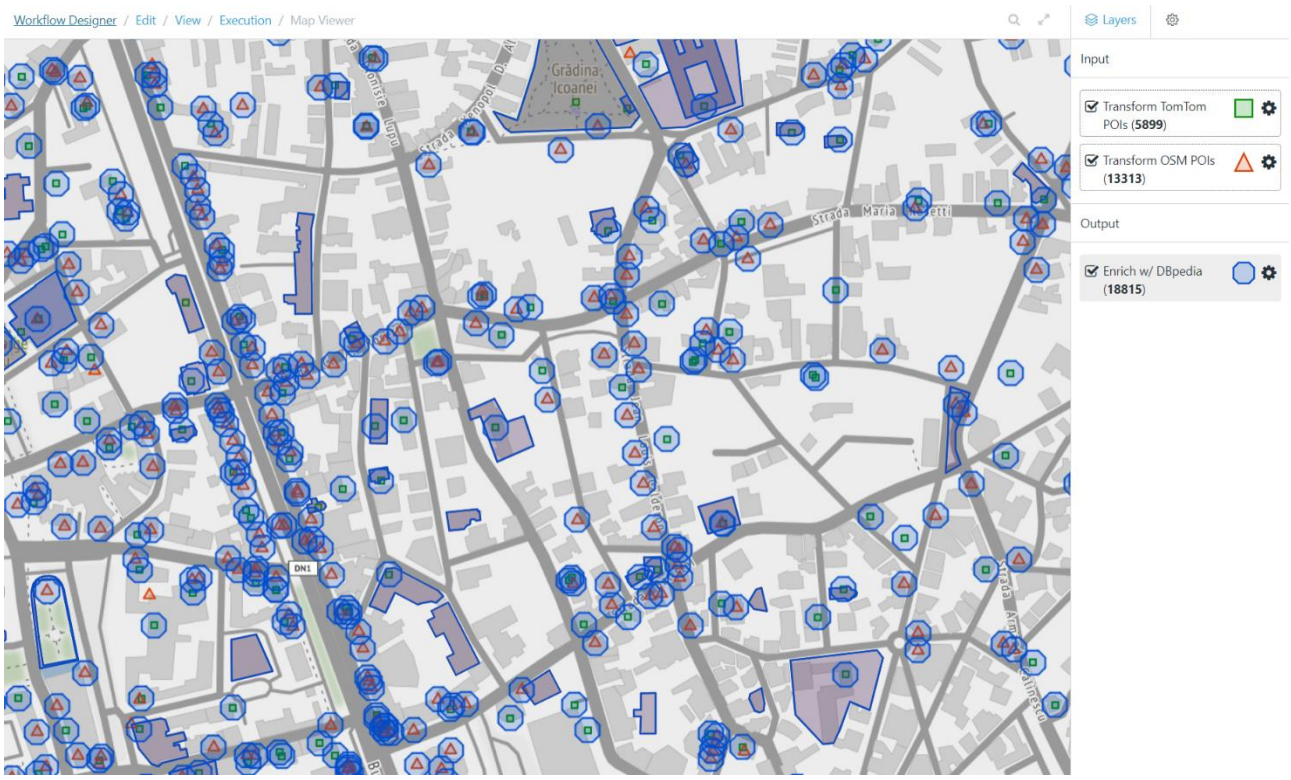


Figure 55: Workflow execution map viewer

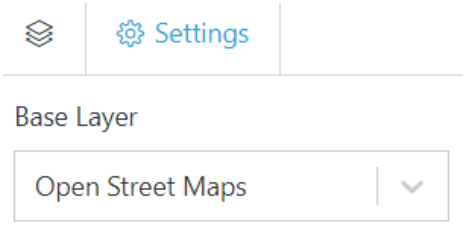


Figure 56: Base layer selection

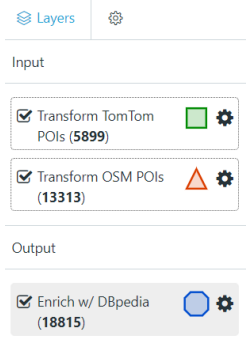


Figure 57: Layer list

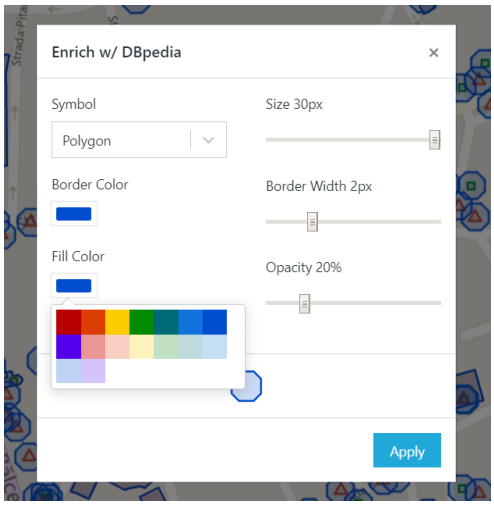


Figure 58: Layer configuration

2.1.4.13. Recipes

Recipes are special workflow instances that are partially saved and can be used as templates for creating new workflows. They are managed like common workflows, support versioning, and are created using the workflow designer. A recipe, however, cannot be scheduled for execution by the SLIPO Service, and is not displayed in the workflow browser. Instead, a separate view has been implemented for managing recipes. Users can either edit a recipe or clone it to create a new workflow.

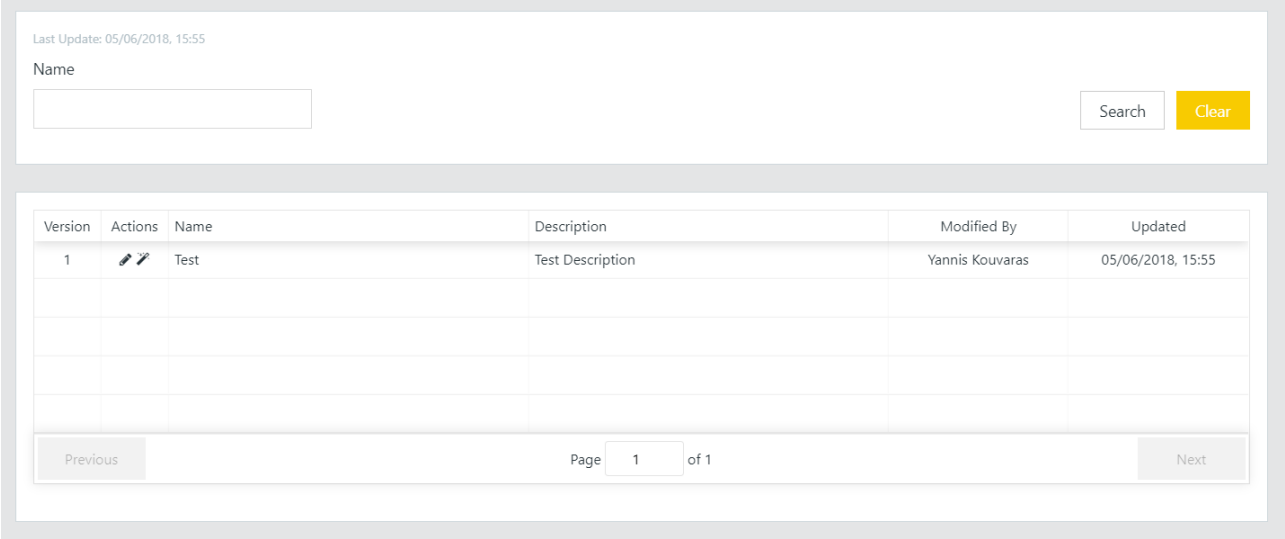


Figure 59: Workflow recipes

2.1.5. Provenance, Evolution and Q&A

In this section we present the Workbench User Interface (UI) for querying the provenance and evolution data and rendering query results. Workbench provides the following features for exploring POI provenance and evolution data:

- Building thematic maps where each input and output file is represented as a separate layer.
- Selecting one or more POIs from the map.
- Displaying provenance data for a single POI.
- Displaying evolution data for a single POI across multiple workflow version executions.
- Editing POI data and viewing POI update history.

Consider the POI data integration workflow illustrated in Figure 60. This workflow receives two POI datasets as input, which are interlinked and fused. Finally, an enrichment operation is executed on the fused result.

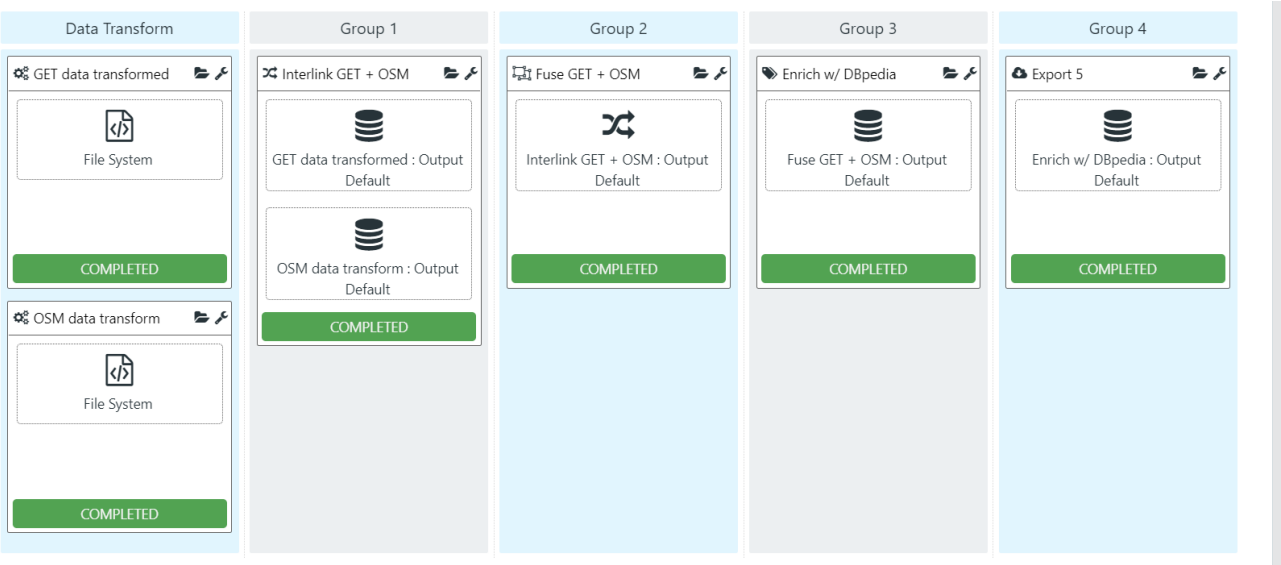


Figure 60: A POI data integration workflow.

The user can visualize the involved POIs on a map, with different layers corresponding to different input and output datasets as shown in Figure 61.

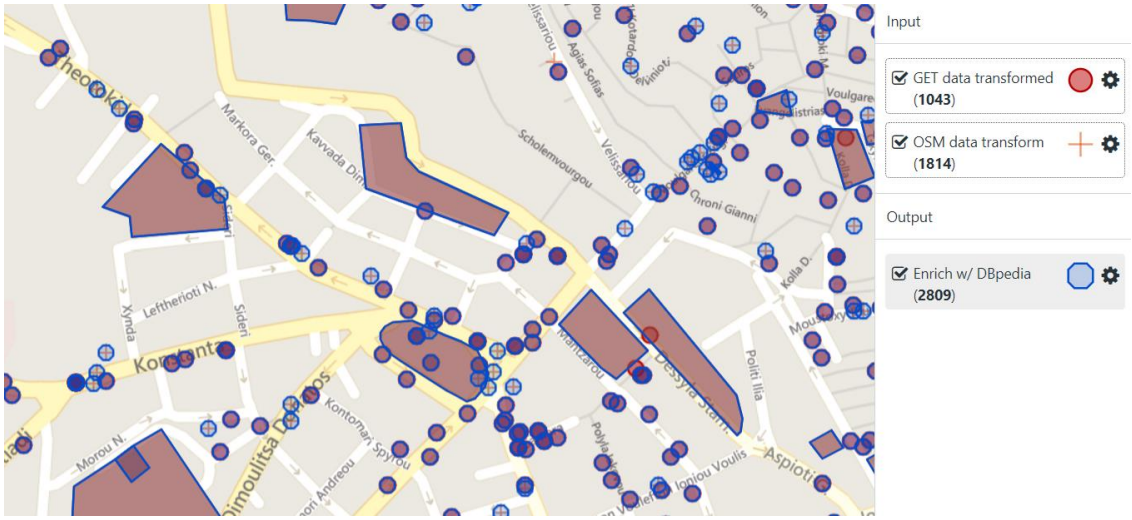


Figure 61: Map visualization of the workflow's input and output datasets.

By clicking on the map, the user can select a desired POI. In particular, given that often there may exist a large number of POIs located very close to each other (or even with the exact same coordinates), a popup window is displayed that allows the user to select among those POIs located close to the point that was clicked. This is shown in Figure 62. The popup window contains a table where each record corresponds to a POI, including the respective layer that it comes from, and basic POI attributes (id, name, category, etc.).

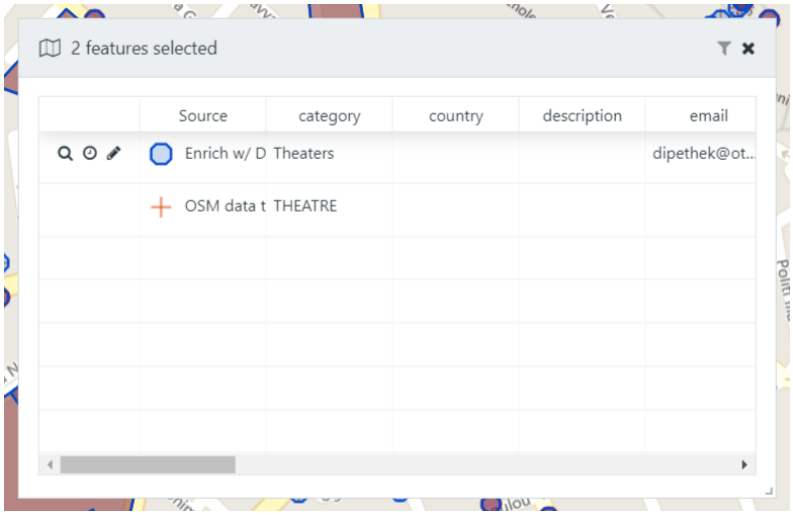
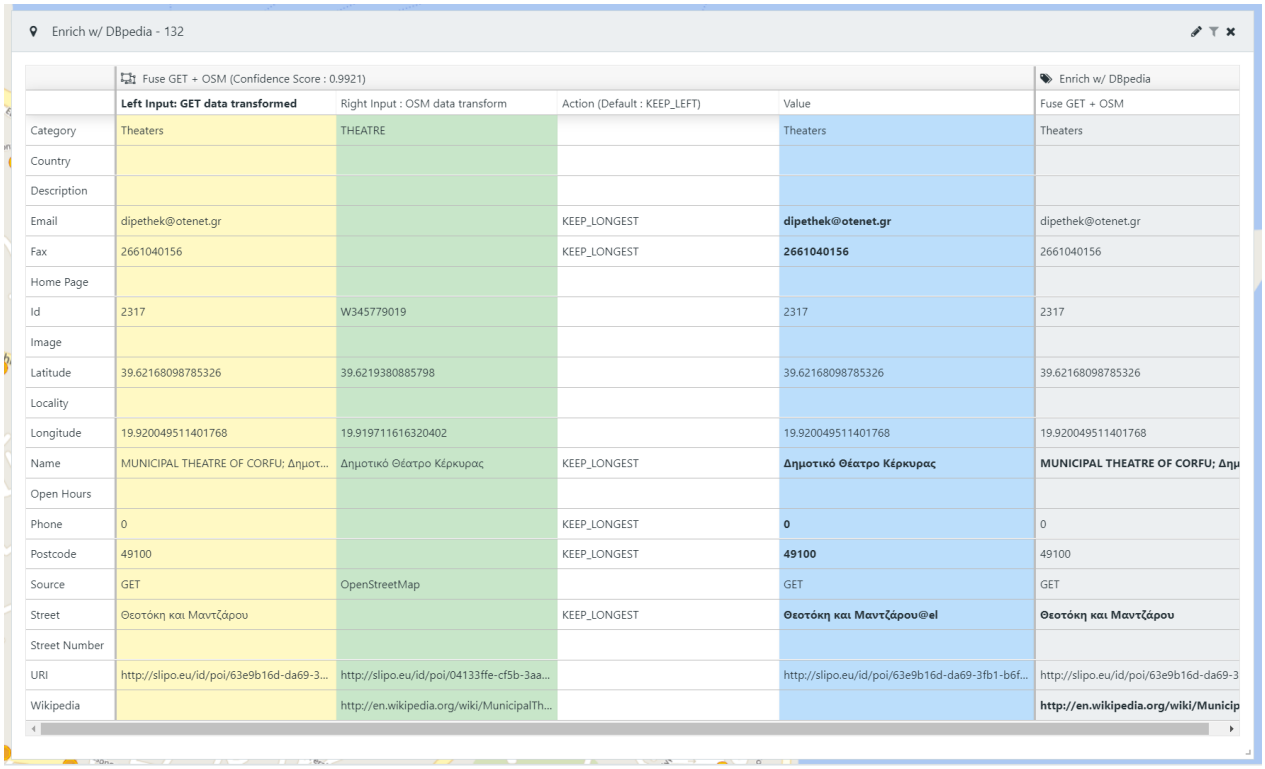


Figure 62: Selecting a POI on the map.

When the user selects an output POI, a new popup window is displayed containing the complete provenance information for this POI. This is shown in the following Figure 63.



Enrich w/ DBpedia - 132

	Fuse GET + OSM (Confidence Score : 0.9921)				Enrich w/ DBpedia
	Left Input: GET data transformed	Right Input : OSM data transform	Action (Default : KEEP_LEFT)	Value	Fuse GET + OSM
Category	Theaters	THEATRE		Theaters	Theaters
Country					
Description					
Email	dipethek@otenet.gr		KEEP_LONGEST	**dipethek@otenet.gr**	dipethek@otenet.gr
Fax	2661040156		KEEP_LONGEST	**2661040156**	2661040156
Home Page					
Id	2317	W345779019		2317	2317
Image					
Latitude	39.62168098785326	39.6219380885798		39.62168098785326	39.62168098785326
Locality					
Longitude	19.920049511401768	19.919711616320402		19.920049511401768	19.920049511401768
Name	MUNICIPAL THEATRE OF CORFU; Δημοτ...	Δημοτικό Θέατρο Κέρκυρας	KEEP_LONGEST	**Δημοτικό Θέατρο Κέρκυρας**	**MUNICIPAL THEATRE OF CORFU; Δημ**
Open Hours					
Phone	0		KEEP_LONGEST	**0**	0
Postcode	49100		KEEP_LONGEST	**49100**	49100
Source	GET	OpenStreetMap		GET	GET
Street	Θεοτόκη και Μαντζάρου		KEEP_LONGEST	**Θεοτόκη και Μαντζάρου@el**	**Θεοτόκη και Μαντζάρου**
Street Number					
URI	http://slipo.eu/id/poi/63e9b16d-da69-3...	http://slipo.eu/id/poi/04133ffe-cf5b-3aa...		http://slipo.eu/id/poi/63e9b16d-da69-3fb1-b6f...	http://slipo.eu/id/poi/63e9b16d-da69-3
Wikipedia		http://en.wikipedia.org/wiki/MunicipalTh...			**http://en.wikipedia.org/wiki/Municip**

Figure 63: Provenance information for a selected POI

This table provides detailed provenance information for each individual attribute of the POI. Specifically, each record in the table corresponds to an individual attribute (e.g., name, category). The columns of the table correspond to the individual operations taking place during this workflow. By following the values in

each row from left to right, we can easily track and inspect the value of the corresponding property throughout the workflow execution. Thus, we can see what the **original** value was in the input, and **if, when, and how** this value changed throughout the integration process.

In the example above, the UI highlights the following updates:

- During the fuse operation for GET and OSM datasets, the POI from the GET dataset was selected. A confidence score is also displayed next to the step tile. This value is computed by the FAGI component.
- Fusion actions have been applied to fields email, fax, name, postal code and street. The fused values and actions are also displayed. Moreover, the default fusion action applied when no rule is applicable is shown next to the action header title. In this example, the KEEP LEFT default action is used.
- Finally, the enrichment operation has updated the missing value for the Wikipedia attribute of the GET POI.

When the provenance details window is displayed, the user can **edit** the selected feature as part of the Q&A process. The UI allows the user to update the feature properties and geometry as shown in Figure 64. Selected properties such as the initial entity id and Uri are read-only.

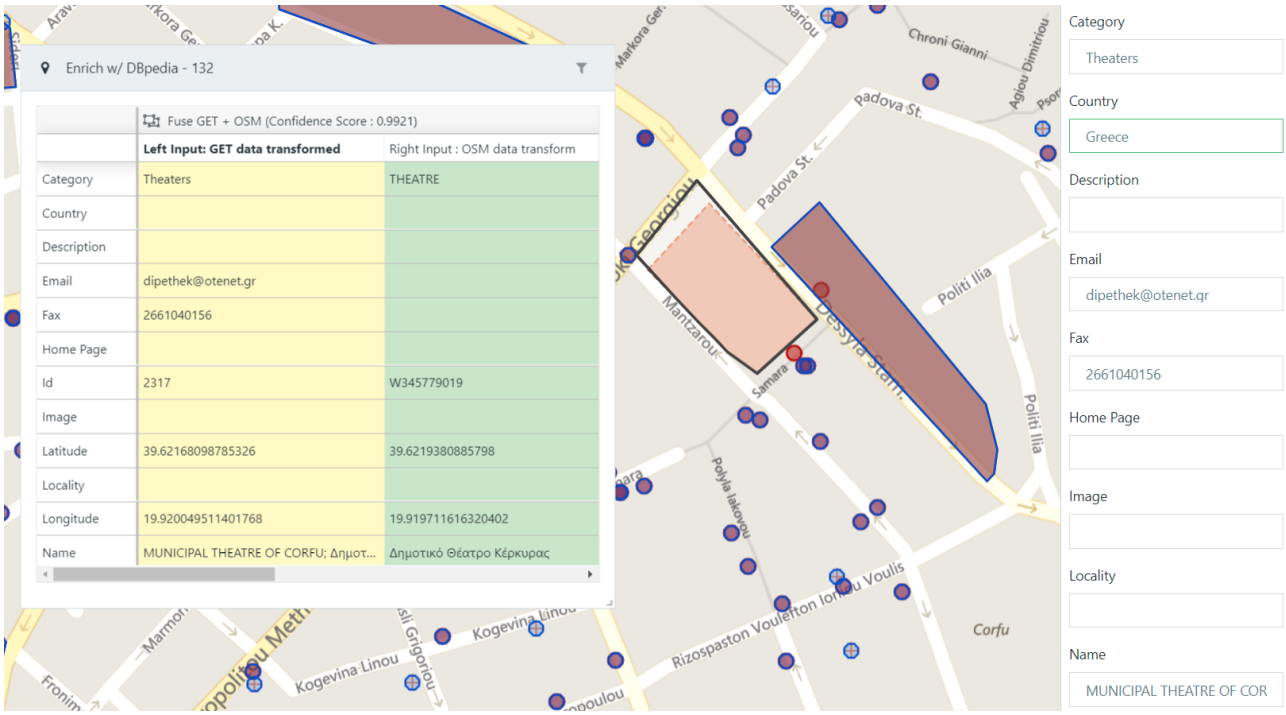


Figure 64: Editing POI properties and geometry

All the updates applied to a specific feature are logged by the system and the user can preview the update history from the provenance window. For every update, an additional column is added after the last workflow operation. The new columns provide information about *who* changed the entity, *when* the update was committed, and *which* attributes have been modified. An example of an update is displayed in Figure 65. Moreover, the geometry for every update can be viewed individually on the map.

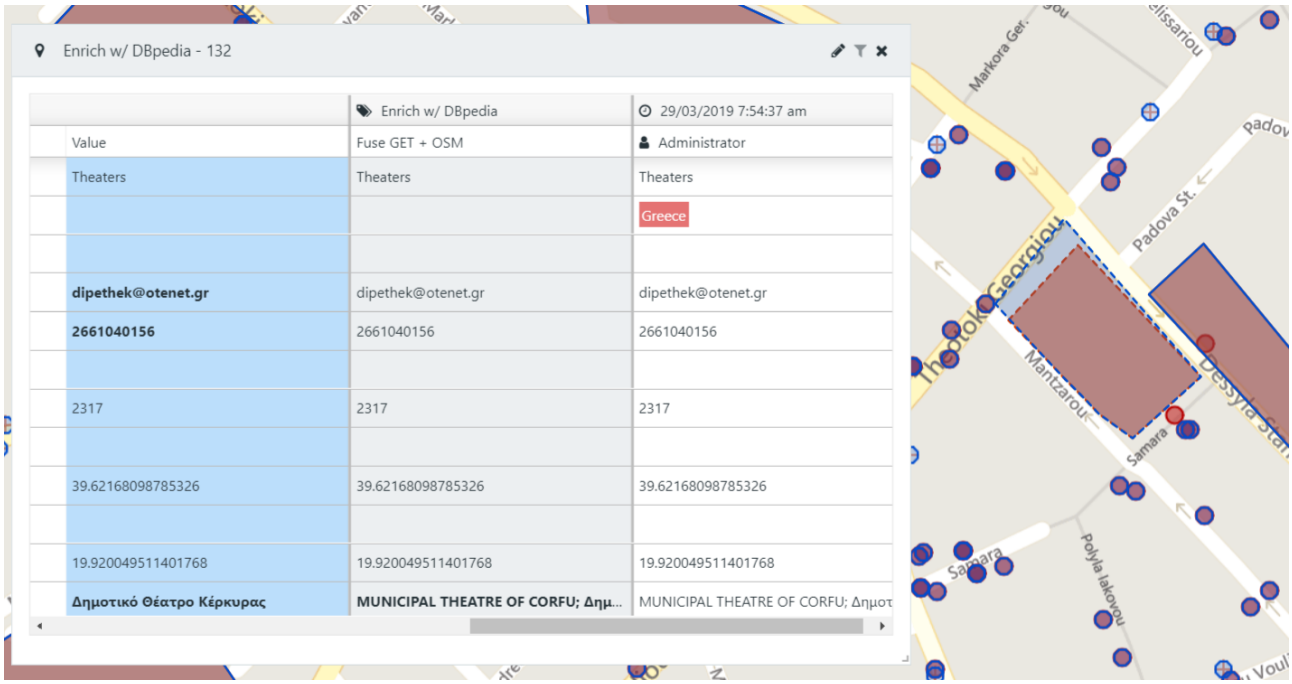


Figure 65: POI update history

If the workflow of the selected execution has multiple versions and the corresponding executions have also been processed by the RDF Import Service, the user can view the evolution of the POI across **multiple executions**. An example of the evolution viewer is depicted in Figure 66.

The evolution viewer displays a row for every POI attribute and a column for every execution. The columns are ordered by the most recent execution first. POI attribute changes between adjacent executions are also highlighted. The evolution viewer also takes into consideration any updates applied by the provenance editor for each individual execution. Users can hide or show the user-specific updates by toggling the appropriate button on each column. Finally, like the provenance viewer, the user can view the geometry of the POI for each execution.

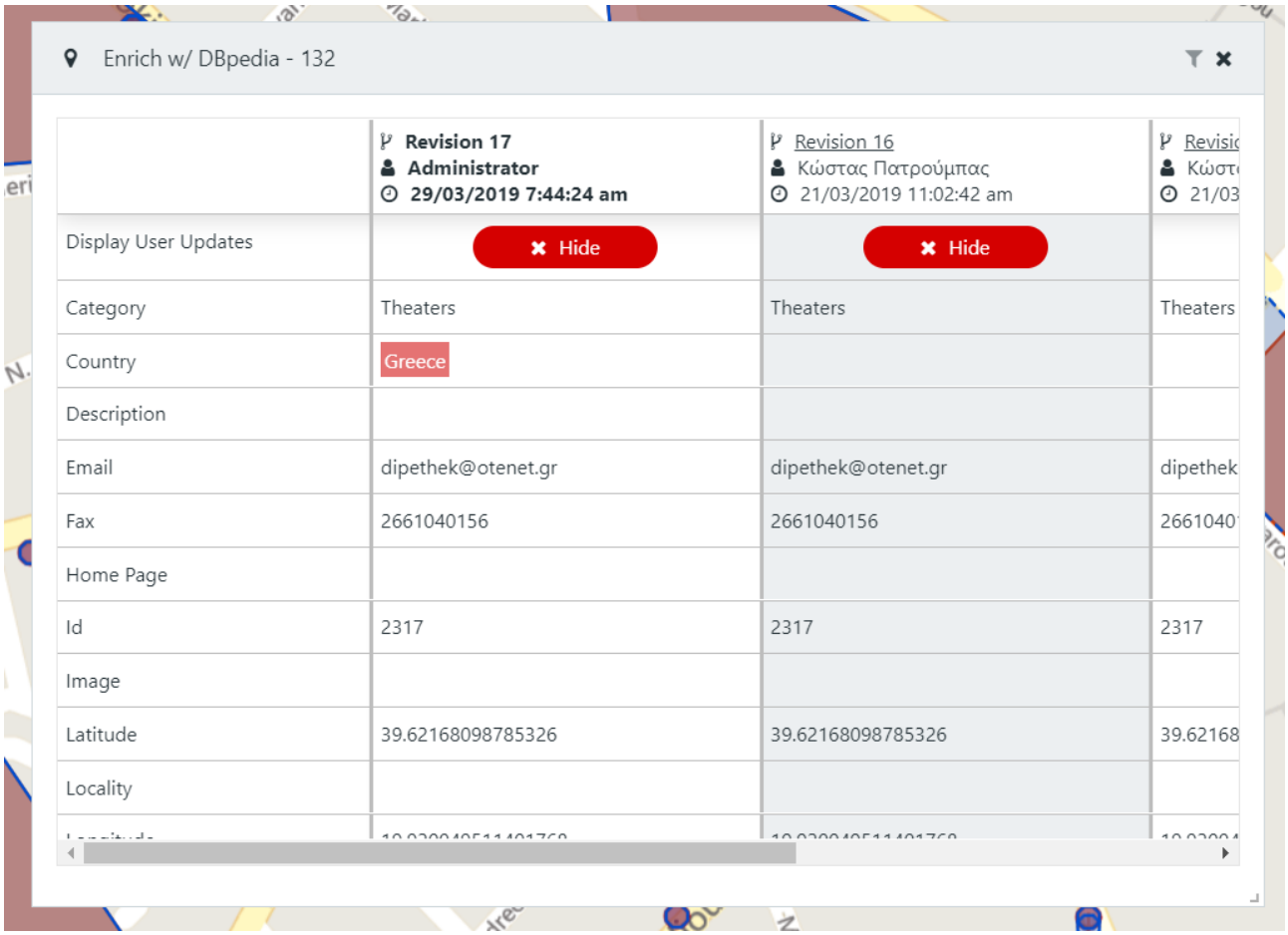


Figure 66: POI evolution data

2.1.6. Administration

In the previous sections we have enumerated the UI components that implement the functional requirements of the SLIPO workbench application. In this section, we provide an overview of the application views used for implementing non-functional requirements of the application such as user administration and application key management.

2.1.6.1. User Management

The user management view, depicted in Figure 67, allows the system administrators to easily modify user roles and data. After selecting a user record from the table, the administrator can modify the name of the user and the roles assigned to her.

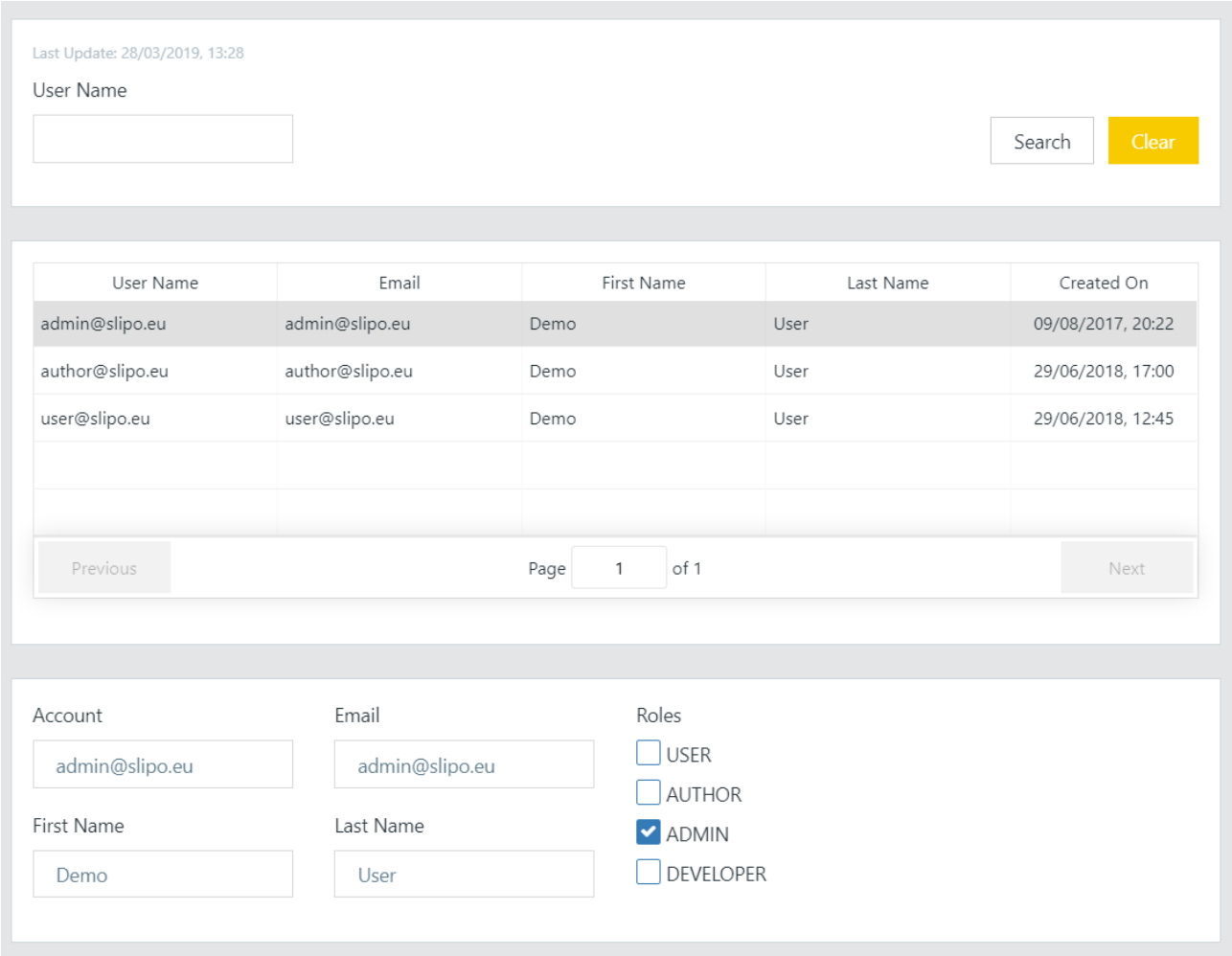


Figure 67: User management

2.1.6.2. System Events

The system events view, shown in Figure 68, supports browsing and filtering all the system events. The user can filter events by severity level and user account. The user account information is available only when a system event is generated by the immediate interaction of a user.

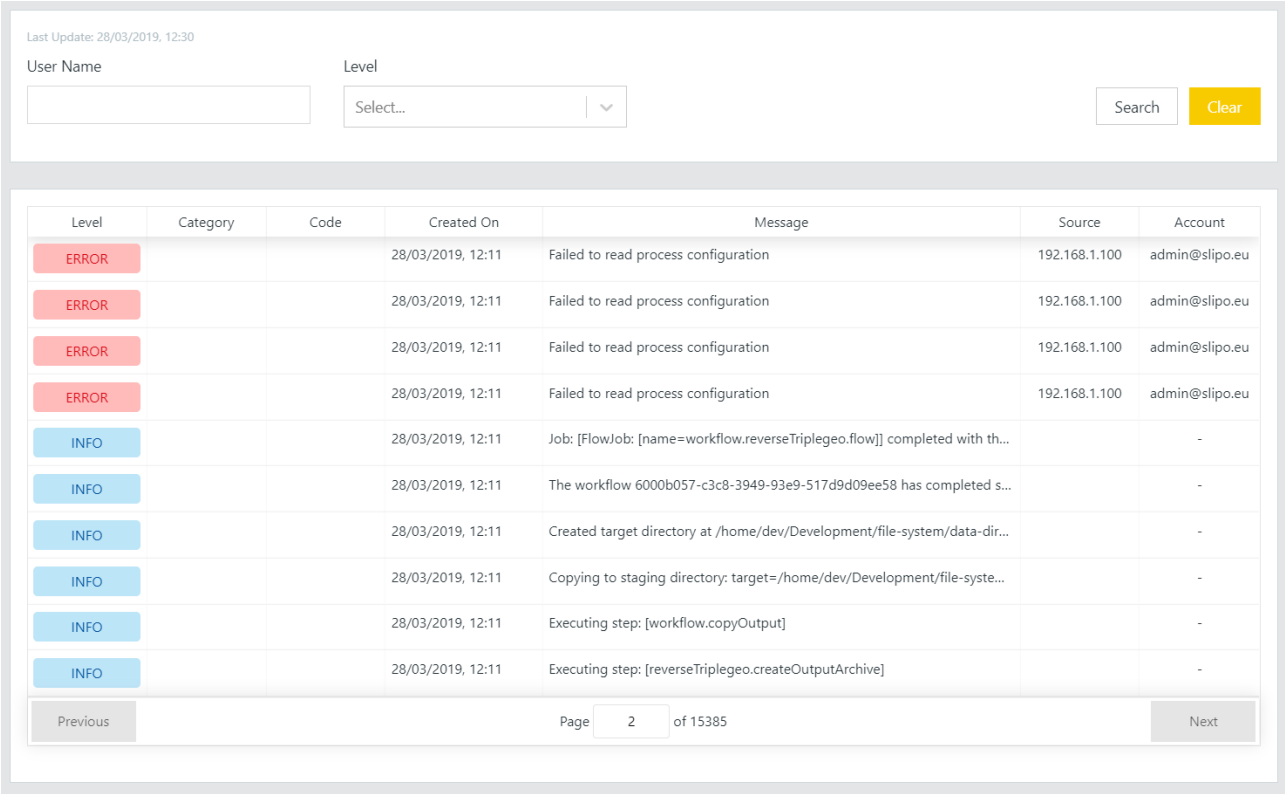


Figure 68: System events

2.1.6.3. Application Key Management

SLIPO API is using application keys for authenticating requests. Every application key is mapped to an existing user account. Any SLIPO API request is executed in the security context of the mapped user. The application key management view is used for managing SLIPO API application keys and provides the following features:

- Browse and query application keys. Keys can be filtered by application name, mapped account and status as illustrated in Figure 69.
- View the value of an application key and copy it to the clipboard.
- Revoke an existing key.
- Create a new key as show in Figure 70.

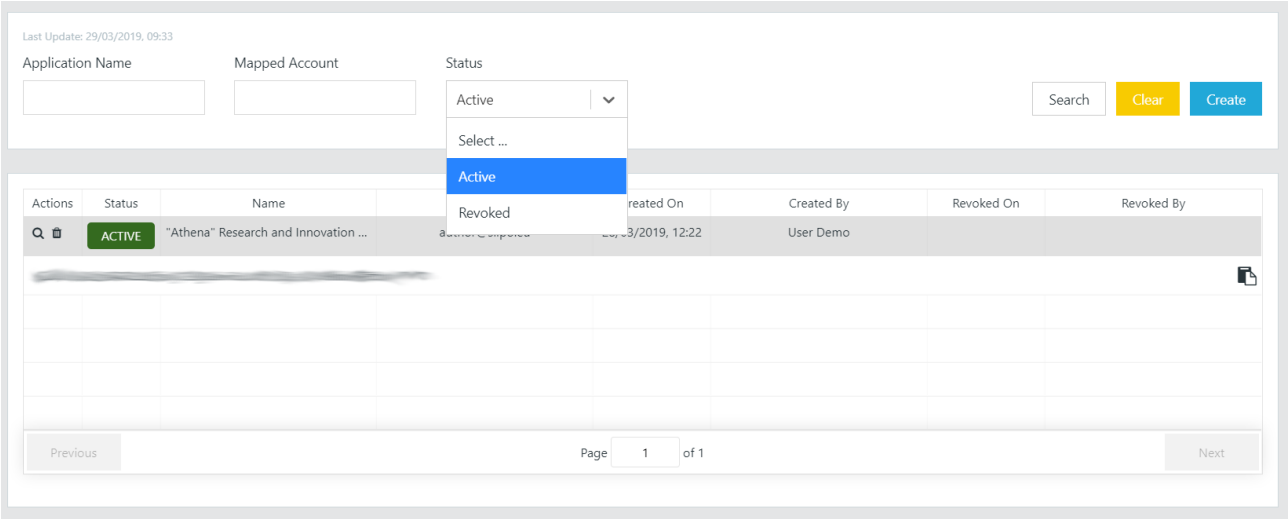


Figure 69: Application Key list

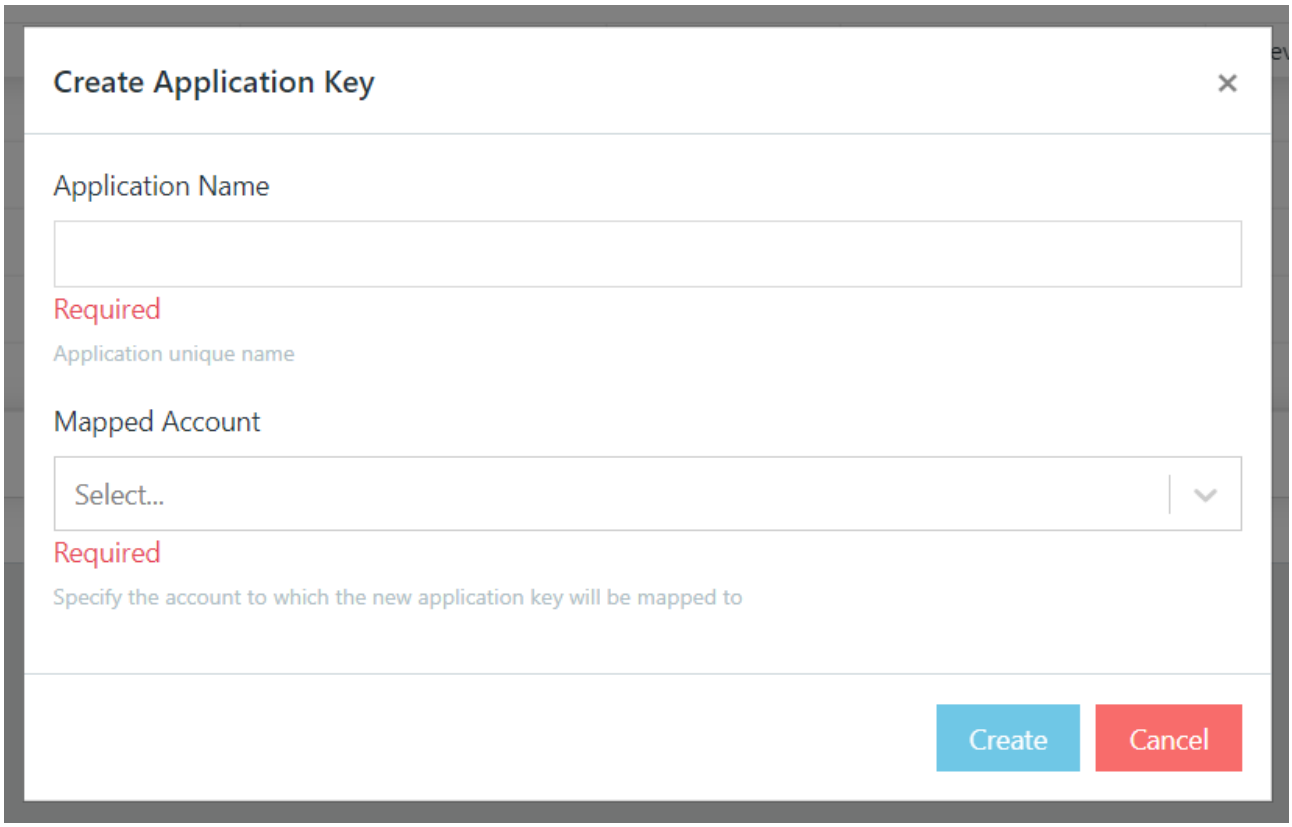


Figure 70: Application Key creation form

2.1.6.4. API Usage

The workflow executions view, described in section 2.1.4.2, is used for displaying historical data for all workflow executions initiated by the workbench web application. The API Usage view, shown in Figure 71,

is used for browsing and filtering SLIPO API method calls. Data can be filtered by the application name, the requested operation and the execution status.

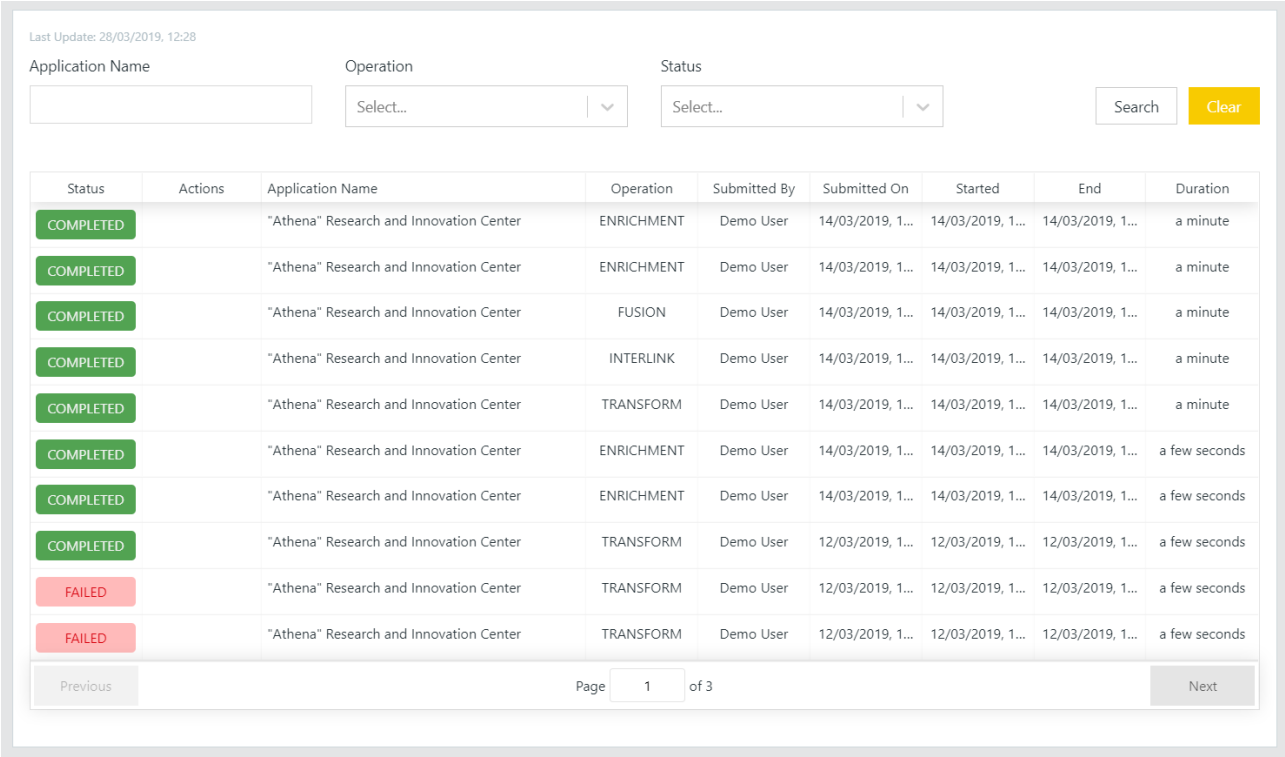


Figure 71: SLIPO API usage

2.1.7. User Roles

The SLIPO Workbench instantiates the following integrated user roles, corresponding to distinct operational requirements of the POI data integration workflows in a real-world setting. The creation of new users and role assignment to users is performed on an organizational-level by the corresponding administrators. New roles can also be created ad hoc, to better suit the internal processes and workflows of each organization.

- *SLIPO Administrator.* This role is responsible for the administration and management of the entire SLIPO system, ensuring for its optimal and efficient operation. In a production setting, these users are personnel of the entity offering SLIPO as a commercial service.
- *Organization Administrator.* This role is responsible for all administration activities related to a single organization having access to the SLIPO system. The Organization Administrator is responsible of organizational-level user management, the provision and management of data assets, as well as the management of workflow jobs.

  The requirements for users assigned to this role is limited to a basic understanding of user roles and rights within the SLIPO system, as well as the process for adding organization resources.
- *Workflow editor.* This role is responsible for authoring and managing data integration workflows and workflow recipes with the available assets for the specific organization. Editors have full control

over the authoring process, are responsible for the configuration and tuning of the individual components of data integration workflows, as well as the change management of data integration workflows during their lifecycle.

The requirements for users assigned to this role are *flexible*, ranging from basic knowledge of standard geospatial manipulation and data integration (*e.g., CRS, data formats, SQL*), to advanced expertise in data integration issues (*e.g., linking, fusion*). This is reflected in the workflow editing process, with all components supporting three modes: automated (*user provides basic information, such as the CRS*), simple (*user can edit a subset of configuration parameters*), advanced (*user can edit all configuration parameters*).

- *Data integrator*. This role is responsible solely for executing already available data integration workflows, as well as retrieving and evaluating their output.

  The requirements for users assigned to this role are limited to basic knowledge of geospatial data integration concepts as they are assumed to receive an optimized data integration workflow, with any changes needed performed by the Workflow editor.

- *QA user*. This role is responsible for performing the QA established per organization on the output of the data integration workflows, using the QA services provided by SLIPO Workbench. Any QA methodology can be supported by the system, including the manual editing of the output of a data integration workflow.

  The requirements for users assigned to this role are *negligible* and limited to a basic understanding of geospatial concepts. QA users typically operate under specific guidelines (*i.e., asset to examine, geographical coverage, sampling, steps to follow*) and use standard GIS software. SLIPO provides similar integrated facilities for QA (see Section 2.1.5), ensuring that the same individuals and processes can be applied.

2.2. Usage examples

In the following sub-sections, we demonstrate the functionality available in the Workbench through a typical data integration example that involves:

- *Transformation* of POI datasets to RDF according to user-specified mappings and parameters.
- *Interlinking* the POI datasets in order to identify matching POI entities.
- *Fusion* of the datasets into a new one according to several fusion strategies for fusing spatial and thematic properties of linked geospatial RDF entities.
- *Enrichment* of the fused dataset with information extracted from the OSM database.
- *Resource registration* concerning RDF data created in any of the aforementioned stages. These resources will be permanently stored in the file system and can readily invoked in workflows.
- *Export* of the integrated data into a conventional file format (e.g., shapefile, CSV) in order to be used by other applications or services.

In the example, we follow a data integration workflow applied on two different datasets containing POIs for fuel stations in Berlin, Germany. The two data sources have different schema, content, and quality, with one being crowdsourced (OSM) and the other offered by a mobility service provider (DKV). Through SLIPO, the user can define in a few minutes a simple data integration workflow that delivers a single dataset with more POIs (e.g., missing POIs from OSM), richer geometries (e.g., polygons from OSM), and more accurate information (e.g., updated telephone numbers from OSM).

Once users log in the Workbench, they can choose "***Design***" from the ***Dashboard*** in order to start composing their data integration workflow, as illustrated in Figure 72:

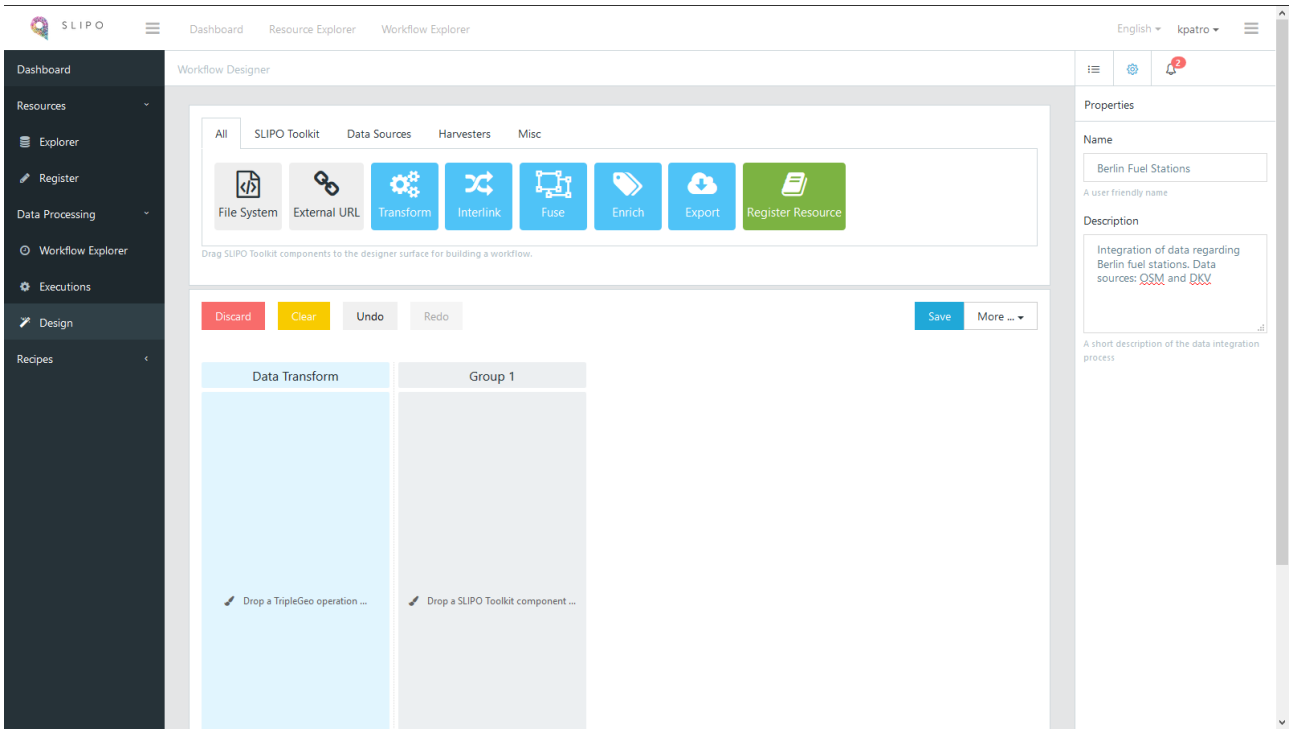


Figure 72: Design interface for a data integration workflow

Depending on the actual data integration task they want to implement, the users may choose instantiations of the available tools (e.g., TripleGeo for *transformation*, LIMES for *interlinking*, FAGI for *fusion*) and simply drag them into the design pane. At each user action, the Workbench application validates it and accordingly issues warnings or errors on the right side of the screen. In the example shown in Figure 72, these warnings concern the identification of the workflow, and the user is prompted to specify a name and a short description for the data integration process that is about to design. Note that tasks in a workflow can be defined in successive *groups*, always starting with a data transformation task. Tasks within the same group can run in parallel, while a task in a subsequent group must wait to receive the output from a previous group, if necessary.

2.2.1. Example Datasets

The POI datasets involved in these examples concern fuel stations in Berlin, Germany. More specifically:

1) *DKV dataset*[43], within lists 245 fuel stations with their point lon/lat coordinates in WGS1984, brand name, and full address (i.e., street, house number, postal code).

2) *OSM dataset,* which is extracted from the OSM database[44] and contains 242 fuel stations. Each record includes *full geometry* georeferenced in WGS1984, so position of POIs is not only based on lon/lat point coordinates, but it is more detailed for many of them (i.e., the geometry of the station as a polygon or multipolygon). In addition, many more *thematic attributes* are available per record: phone and fax numbers, email address, website, opening hours, full address (i.e., street, house number, postal code, town, country), as well as international name and classification of the POI.

2.2.2. Transformation

Typically, a data integration task starts with specification of one or more input POI datasets, which should be *transformed* into RDF in order to be processed within the Workbench. **TripleGeo** is used for transforming spatial features from several POI sources into RDF triples. Employing ontology mapping techniques, TripleGeo can transform POI data from various types of sources to their RDF representations and vice versa. The software provides support for several file formats including GML, KML, CSV and ESRI shapefiles, as well as geospatial databases such as Oracle Spatial, PostGIS and MySQL. Moreover, several output formats are supported, including the most common ones like RDF/XML, N-TRIPLES and TURTLE.

As illustrated in Figure 73, the user drags an instance of the **Transformation** tool (transformation to RDF) into the design, and subsequently drags a **File System** inside the Transformation container. This specifies that the transformation process will take as input the data contained in this file and will transform it into RDF. The user can give a name in this transformation process, e.g., "*DKV Transform*" in this case, so as to distinguish it from other such tasks that may be included in the same design.

[43] https://www.dkv-euroservice.com/gb/services/value-added-services/online-services/poi-navigation-data/

[44] https://www.openstreetmap.org

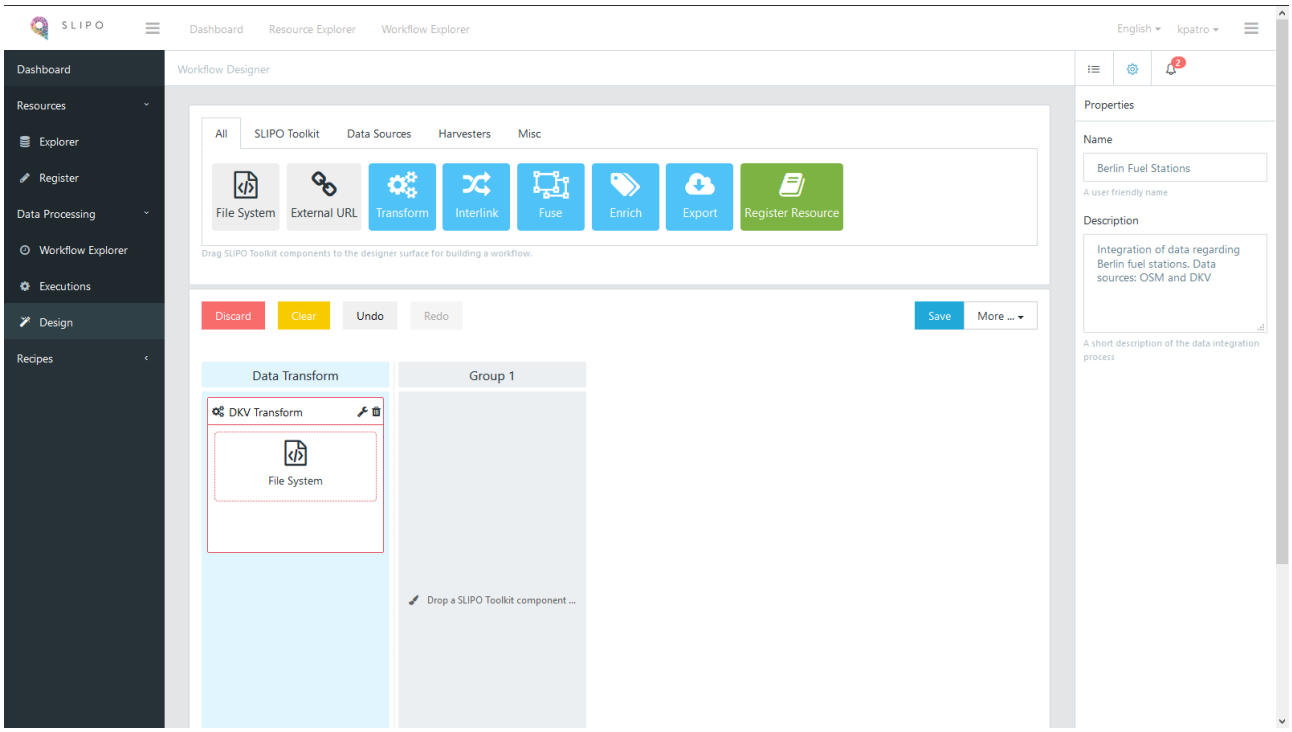


Figure 73: Adding a transformation task to the workflow

Once a *File System* instance is added, the user must select the input dataset. She can either upload a dataset or specify a file already available in the file system of the server, as shown in Figure 74.

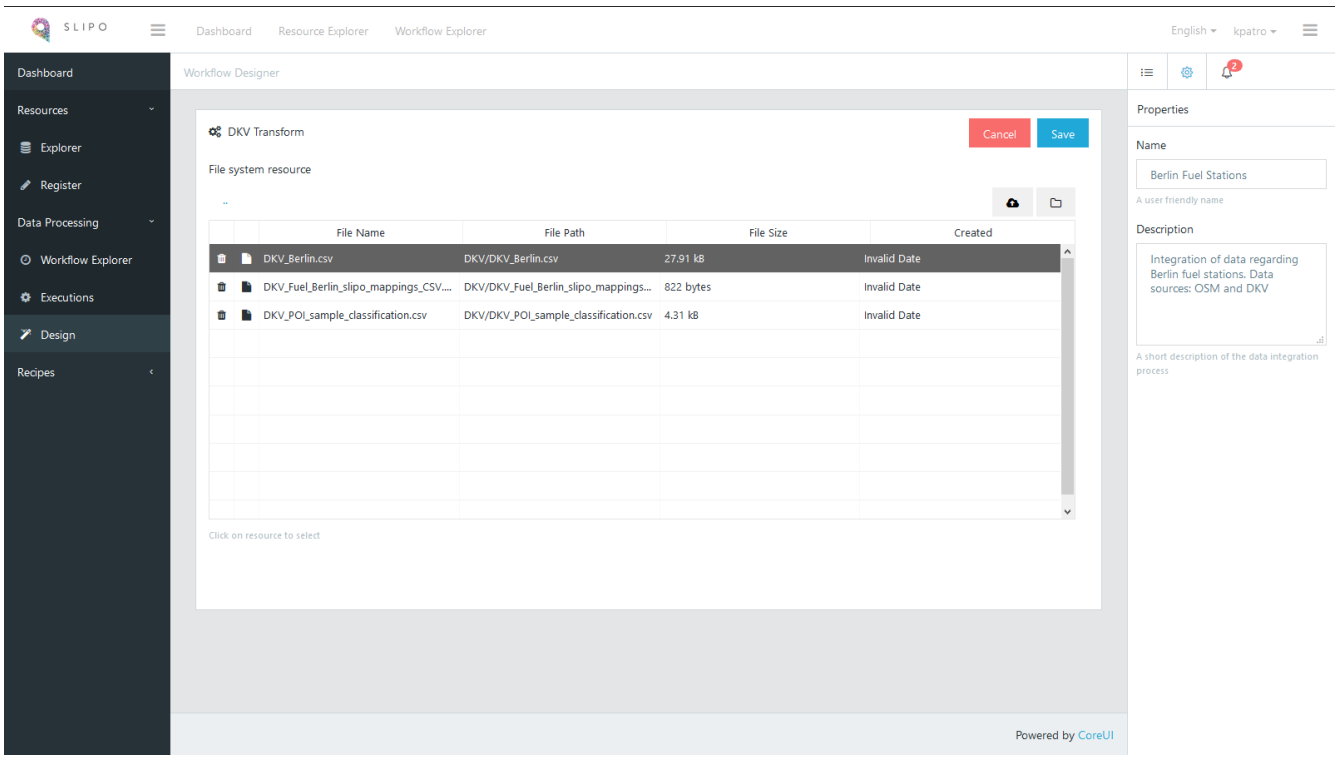


Figure 74: Specifying the POI dataset that will be transformed to RDF

DKV Transform

Cancel Save

Profile

Auto Simple Advanced

Selected Profile

Custom Profile

Specify a default mapping and classification profile

Mapping specification file

dkv/DKV_Fuel_Berlin_slipo_mappings_CSV.yml

File containing YAML mappings from input schema to RDF. Example mappings for TripleGeo can be found here.

Classification specification file

dkv/DKV_POI_sample_classification.csv

File (in YML or CSV format) containing classification hierarchy of categories. Example classification for TripleGeo can be found here.

☑ Classify By Name

Check if features specify their category based on the actual name of the category

Input parameters

Input format

CSV

Specify format for the input geographical file(s)

Mode

STREAM

Conversion mode

Encoding

UTF-8

The encoding (character set) for strings in the input data. If not specified, UTF-8 encoding is assumed

Data parameters

Attribute key

ID

Field name containing unique identifier for each entity (e.g., each record in the shapefile)

Attribute name

name

Field name containing name literals (i.e., strings)

Attribute category

type

Field name containing literals regarding classification into categories (e.g., type of points, road classes etc.) for each feature

Attribute geometry

Parameter that specifies the name of the geometry column in the input dataset

Delimiter

;

Specify the character delimiting attribute values

Quote

"

Specify quote character for string values

X-attribute

lon

Specify attribute holding X-coordinates of point locations

Y-attribute

lat

Specify attribute holding Y-coordinates of point locations

Namespace parameters

Feature source

DKV

Specifies the data source provider of the input features

Spatial Reference parameters

Source CRS

4326

Specify the EPSG numeric code for the source CRS

Target CRS

4326

Specify the EPSG numeric code for the target CRS

Other parameters

Default language

English

Default lang for the labels created in the output RDF. By default, the value will be English-en

Figure 75: Custom, user-specified profile concerning transformation of a POI dataset

The user must specify the properties of this transformation task. By clicking on the wrench icon next to caption of the Transformation task, all required parameters can be defined. The Workbench is equipped with predefined *profiles* for most common transformation tasks (e.g., OSM data), but the users can always edit them to reflect their input datasets. Otherwise, a use can simply specify *custom profiles* as the one shown in Figure 75. In this form, the user can specify all details concerning the transformation of a given dataset, including the mappings and a classification scheme as required for the proper conversion of input POI data into RDF triples according to the SLIPO ontology for POIs. Most importantly, an **Auto** mode is available specifically for the *attribute mappings*, which assists users in defining how each attribute will be mapped into RDF properties. This utility analyzes the contents of each attribute in a new POI dataset, based on its data type (string, numeric, etc.), formatting (e.g., phone numbers, postal codes), as well as the presence of special characters. Thanks to an ML *classifier*, it recommends a suitable mapping for each attribute available in the input based on a corpus of previously specified mappings utilized for the various use cases of POI data. Users can then verify or modify these recommended mappings through a graphical interface (Figure 41) before applying them for transforming their POI data into RDF.

At this stage, this workflow consists of a single transformation task. However, it can be validated and executed, once the user hits the "Save & Execute" button in the design. Once validation is triggered, the user is notified through the **Workflow Explorer** as depicted in Figure 76. Note that this list includes the history of all workflows specified by the same user in reverse chronological order, so the currently specified task is on the top. The red square indicates that this task is currently running. This can be also verified through the **Executions** menu on the left, which shows the status of all executions submitted by the user. As shown in Figure 77, the currently submitted task is still running. Once its execution is completed, note that its status has changed (Figure 78), notifying the user of its success (or failure).

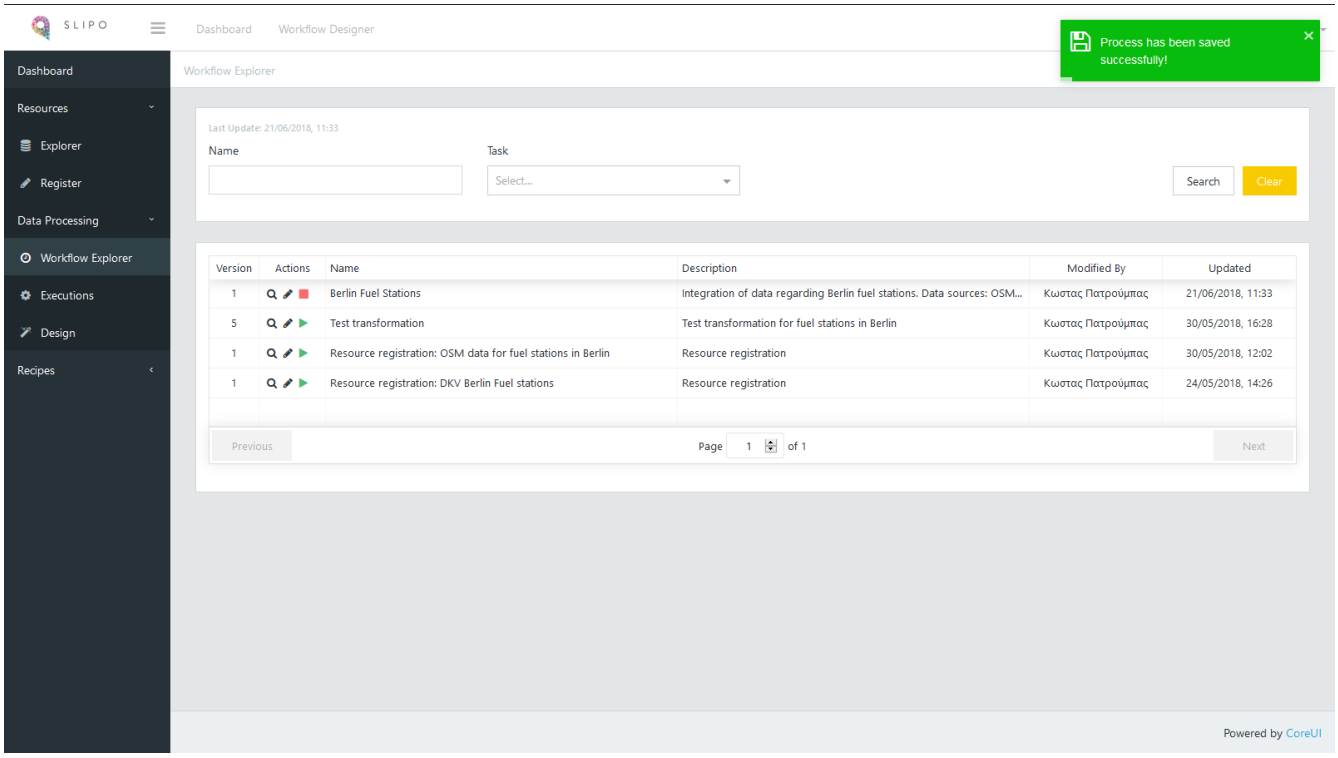


Figure 76: Validation of a user-specified workflow

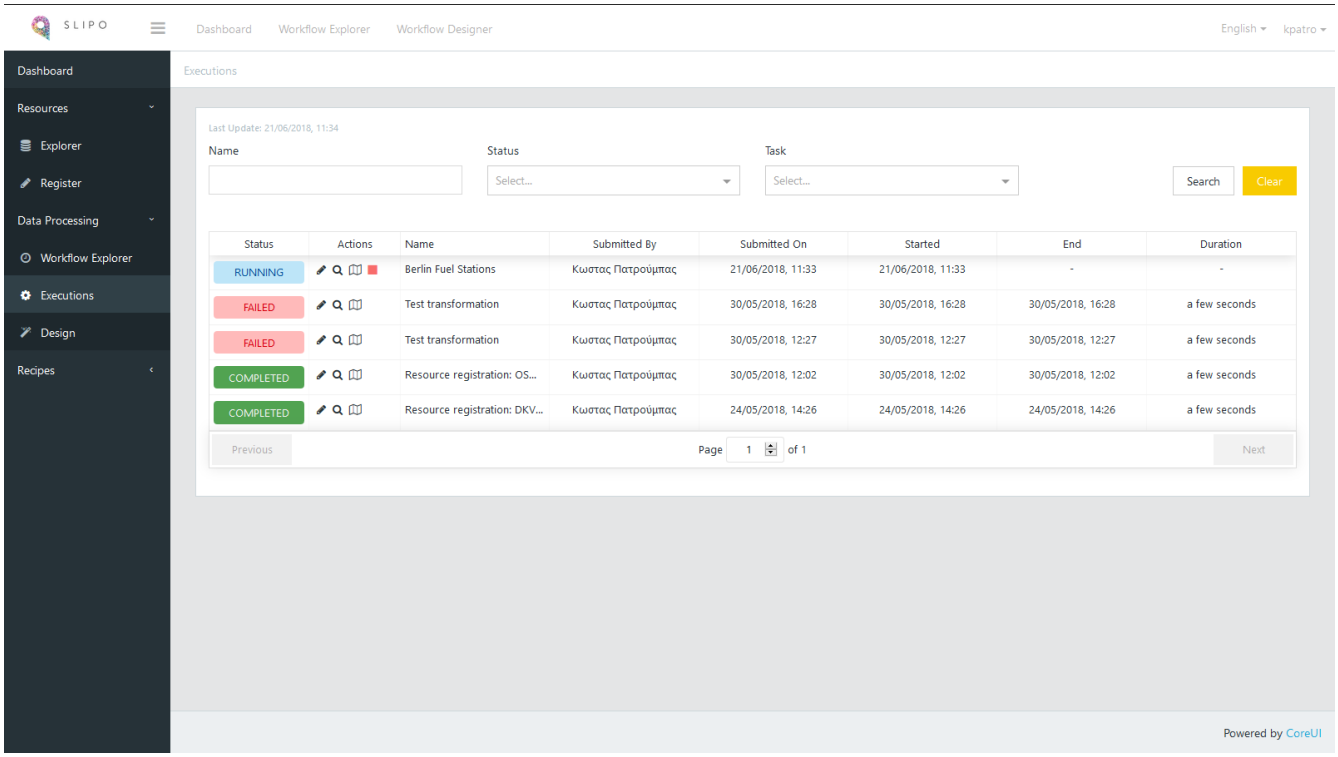


Figure 77: List of workflows specified by a user

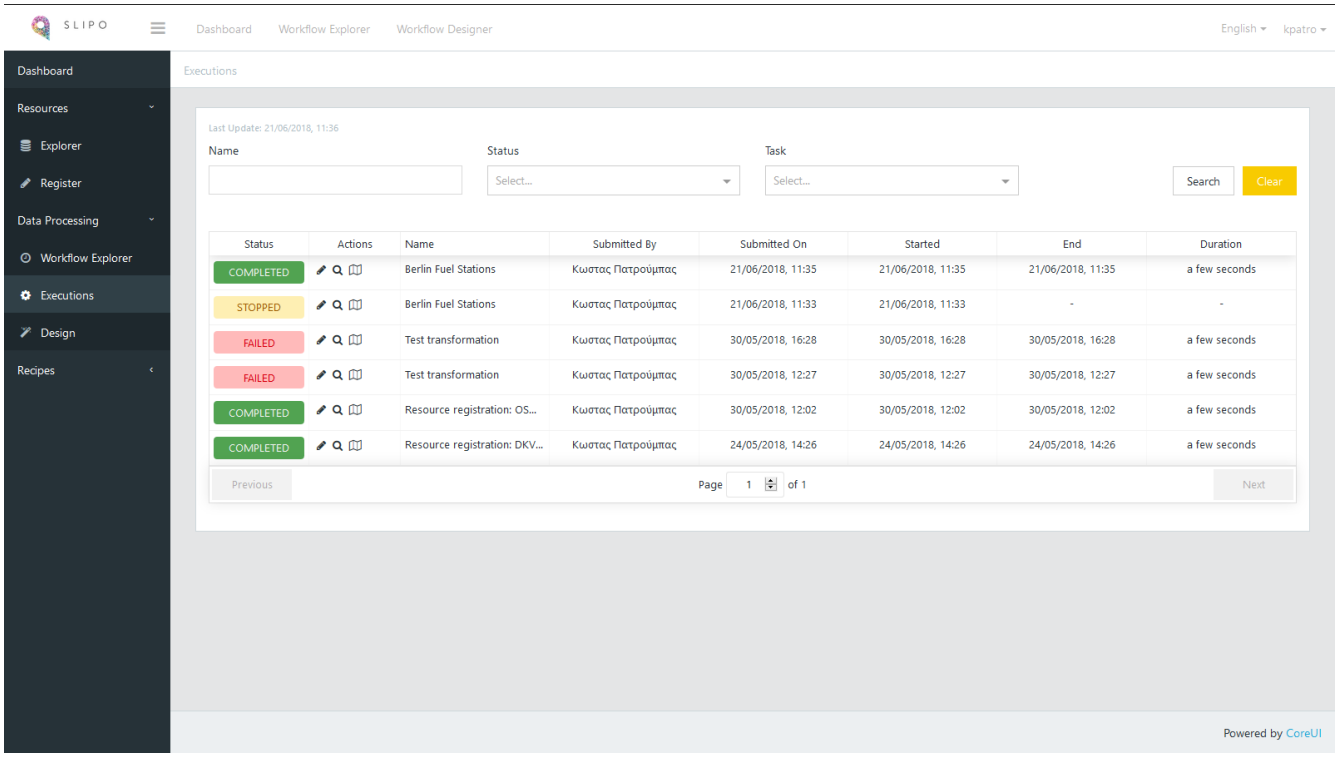


Figure 78: Changes in the status of executed workflows

Under the **Executions** mode, the user can *filter* the executions according to several criteria, such as their name, their status or the specific task. The example depicted in Figure 79 shows the list of successfully completed executions of tasks specified by the user.

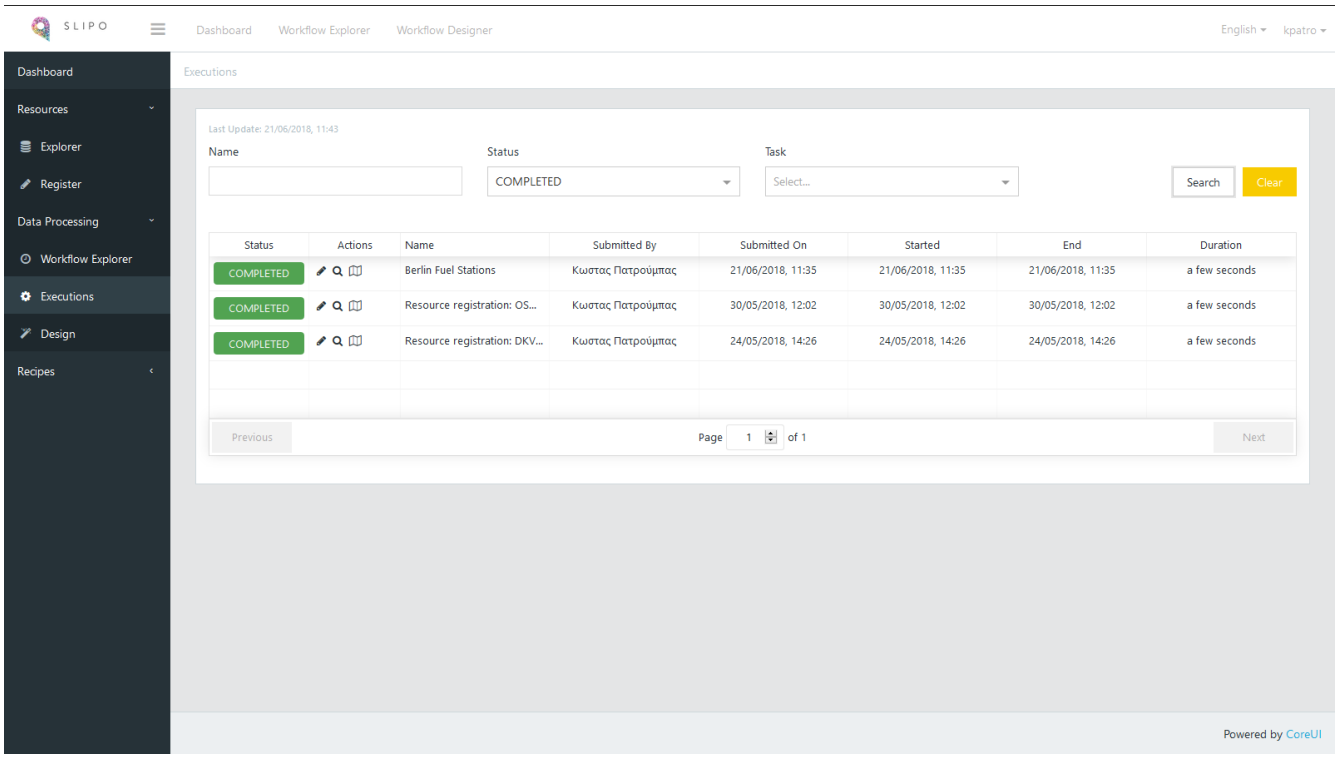


Figure 79: Filter applied on executed workflows

Once the user switches back to the Design mode, the workflow is depicted with all its components (currently only one transformation task has been defined) and its current status as shown in Figure 80.

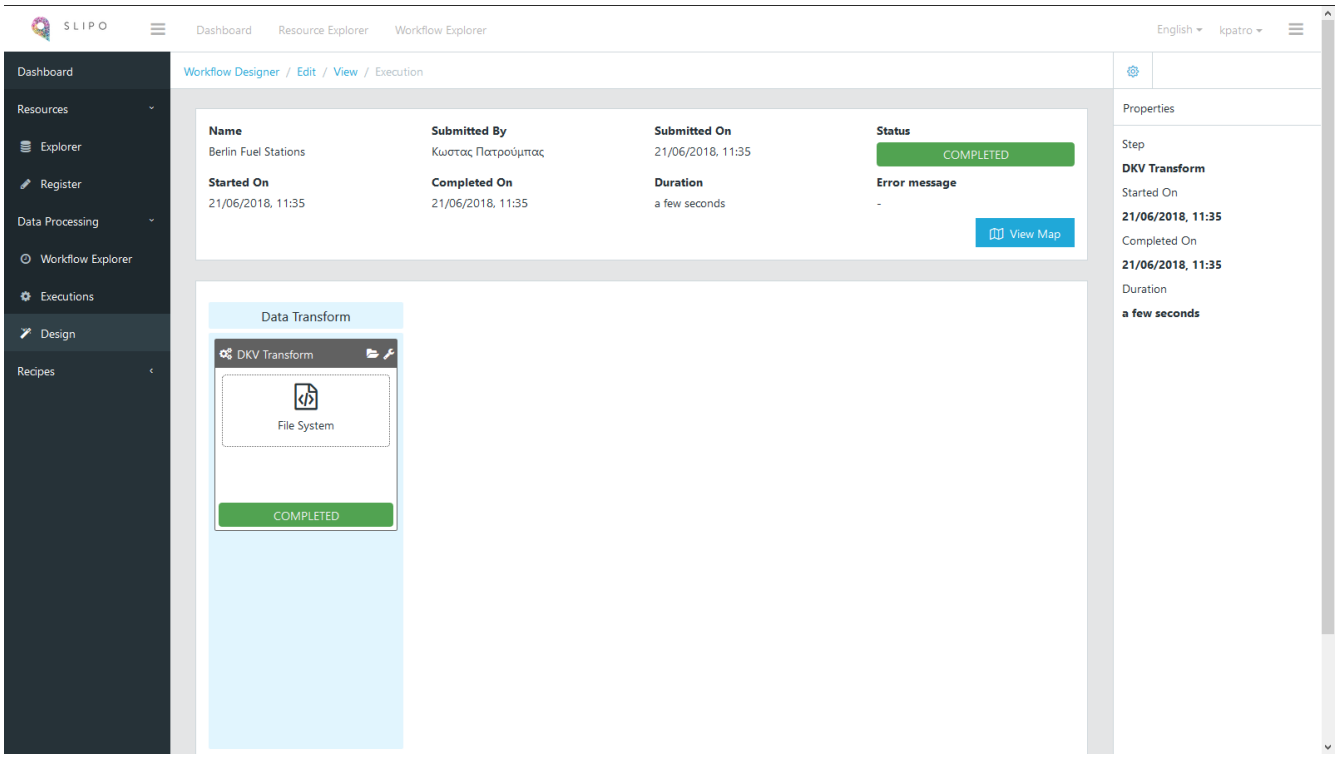


Figure 80: Design and status of a specific workflow

The user can click upon the *folder* icon next to the caption of the transformation task and inspect or download any *file* (input, configuration, output) related to that task. In the example shown in Figure 81, the user has selected the KPI file containing metadata and statistics of the transformation process. This file contains all statistics in JSON format, which can be also visualized with a map displaying the spatial extent of the area covered by the transformed POIs, as well as a chart depicting graphically the amount of values transformed per attribute.

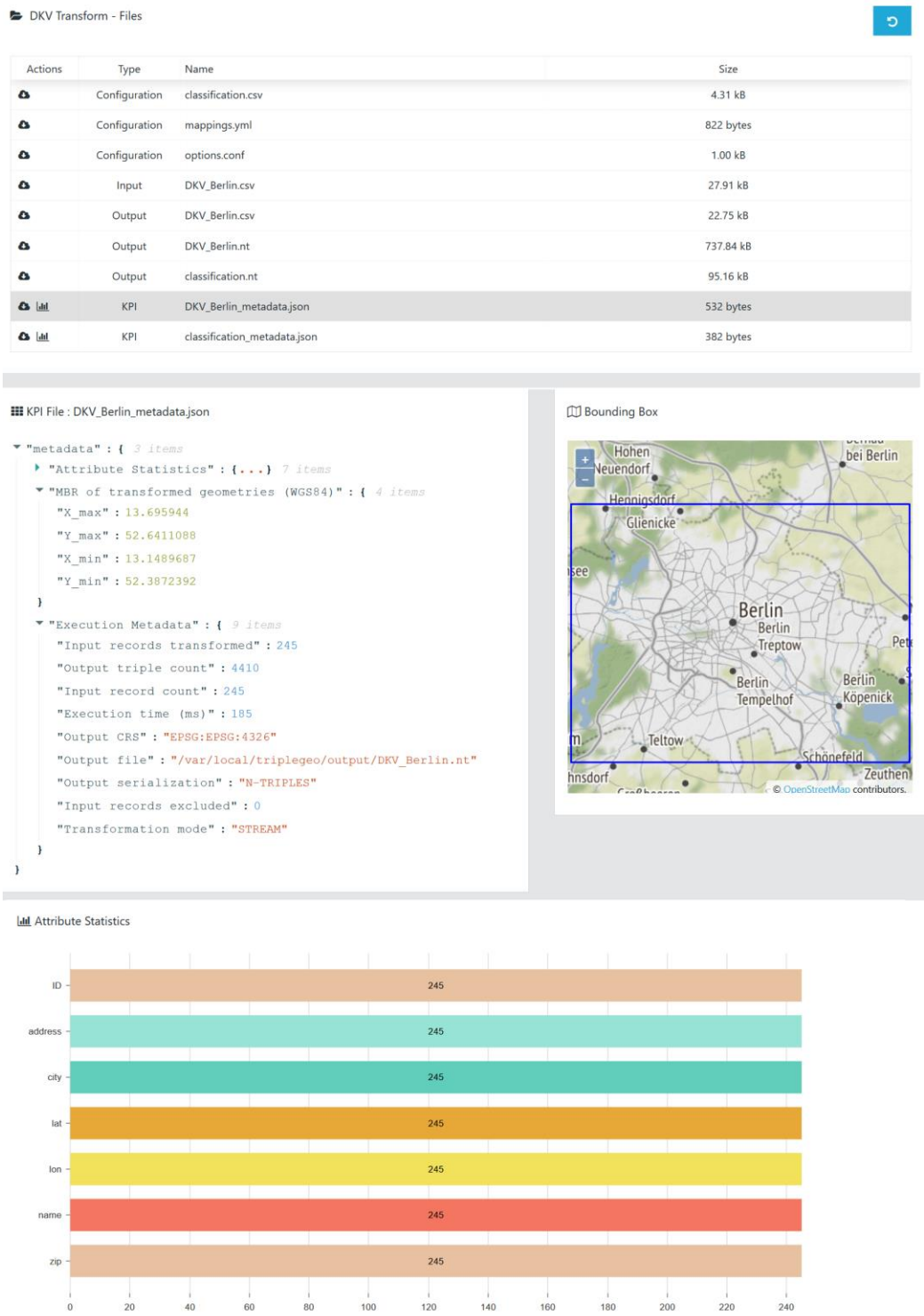


Figure 81: Visualization of KPIs concerning a transformation task

At this point, a single transformation task has been defined concerning the DKV dataset. So, a similar process is followed concerning the other dataset employed in this data integration task. In this case, we simply drag a second instance of the Transform component into the design (just under the first one) and we rename it to "*OSM Transform*", as shown in Figure 82. The steps concerning specification of the input dataset and the configuration of the transformation process are followed exactly like before.

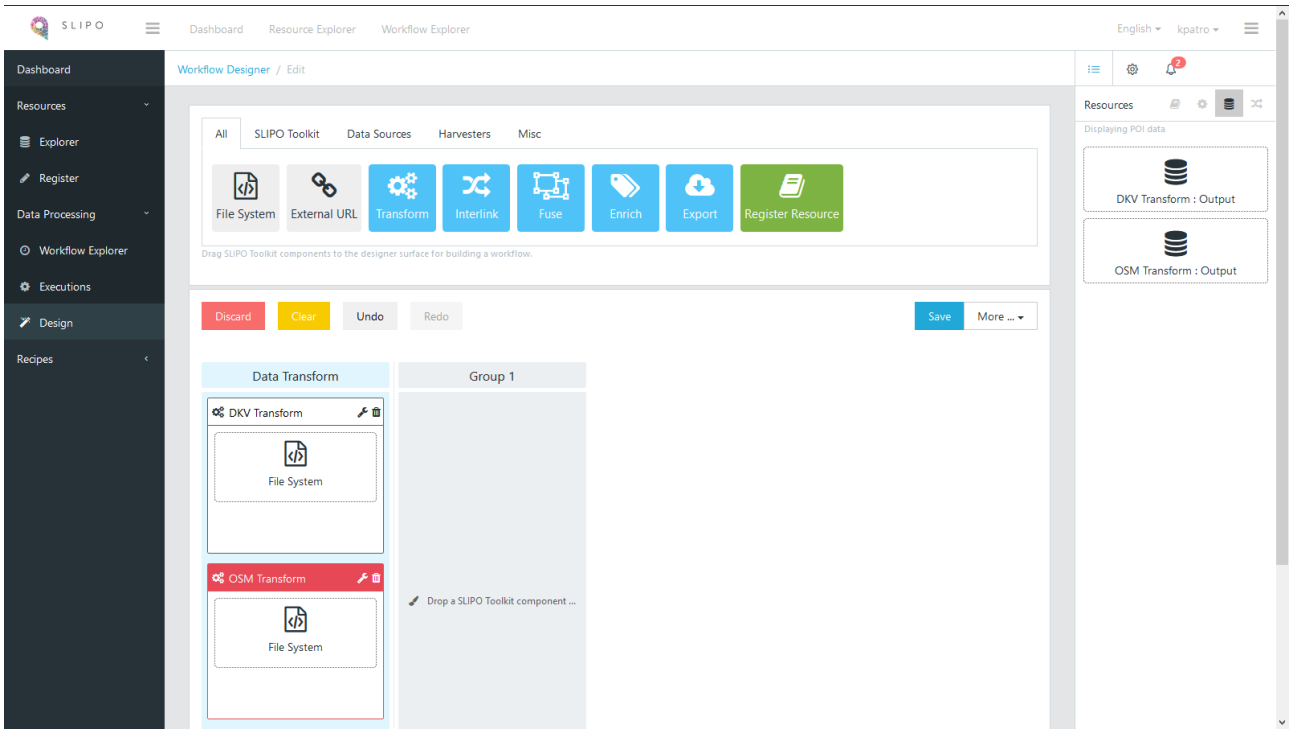


Figure 82: Insertion of a second transformation task in the workflow

2.2.3. Resource Registration

As the result of transformation is files with RDF triples that may be used in several executions, the Workbench provides the ability to declare them as **Resources**. As such, they are stored on the server and they can be invoked at any subsequent workflow, thus eliminating the need to repeat transformation of the same datasets in every workflow that uses them. The same applies for results derived from other stages as well, i.e., after interlinking, fusion, or enrichment.

Note that the output files are listed on the right side of the screen, so that users can utilize them in subsequent stages of the workflow. Defining an output file as a resource can be performed by dragging the **Register Resource** icon into the design and associating its instance with the respective output, as depicted in Figure 83, with the two resources defined for each transformed output. Note that each resource must be associated with a name a short description as shown in Figure 84, in order to be identifiable and reusable in the future.

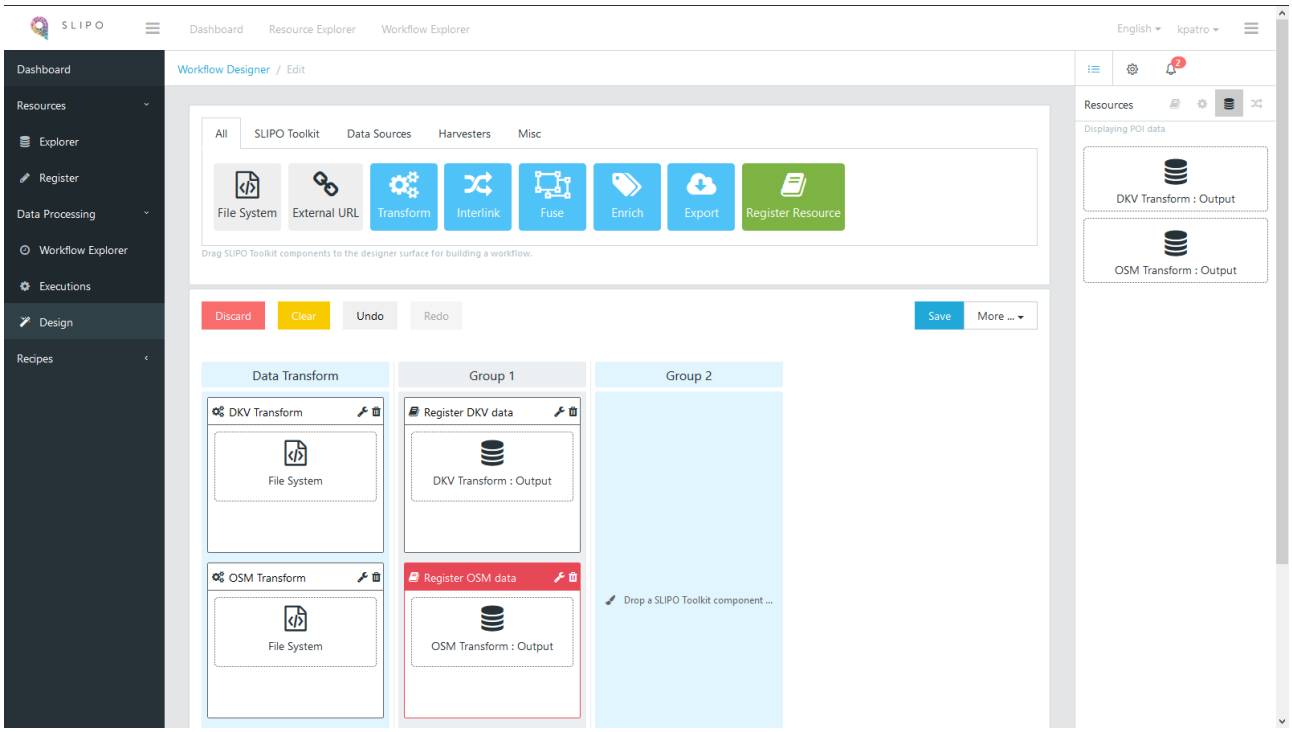


Figure 83: Defining resources on output files in a workflow

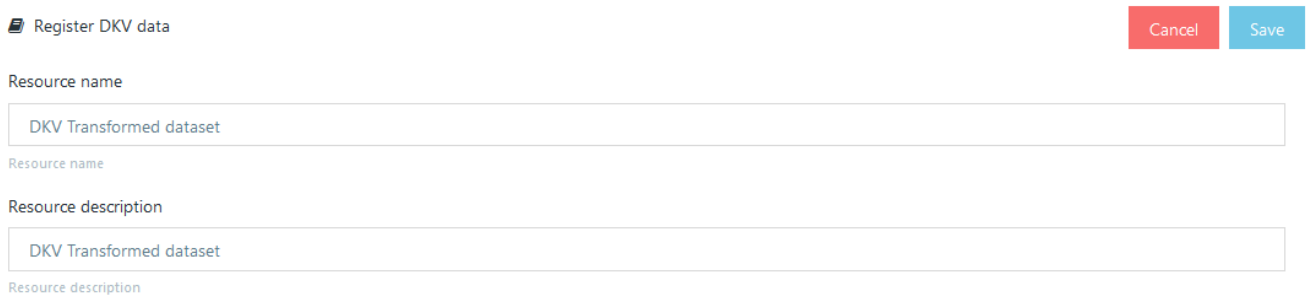


Figure 84: Registering a resource over the transformed DKV dataset

Especially for data transformation, users have an alternative option for a *step-by-step registration* of data as recourses using a wizard. Indeed, by choosing option **Register** available under menu **Resources** in the dashboard, they can be guided to insert POI datasets, transform them into RDF using *TripleGeo*, and also declare them as resources in the Workbench. Once the user invokes the **Register** option, she chooses a proper resource configuration in the following steps:

- Choose the input mode of the data, e.g., upload from her file system, through an external url (Figure 85).
- Specify the path to the file containing the raw POI data (Figure 86).
- Assign a name and description to the resource in order to make it identifiable (Figure 87).
- Configure a data transformation with TripleGeo that will generate the RDF data that will constitute the actual resource or request to be provided with automatically recommended attribute mappings as discussed (Figure 88).
- Inspect a summary of the resource definition to verify its correctness (Figure 89).

- Finally, execute the resource registration task, which also involves data transformation with TripleGeo. The resulting resource will appear in the list of available resources (Figure 90).

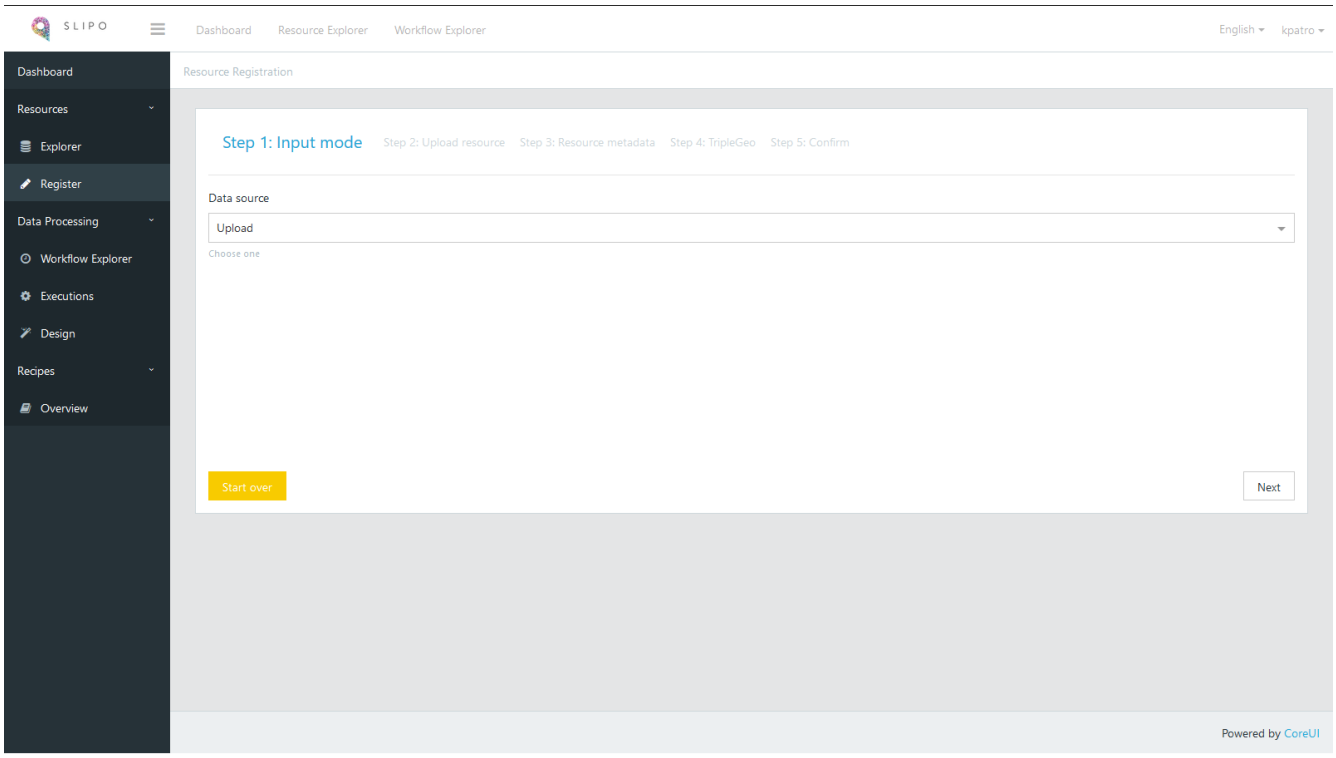


Figure 85: Input mode for defining a resource

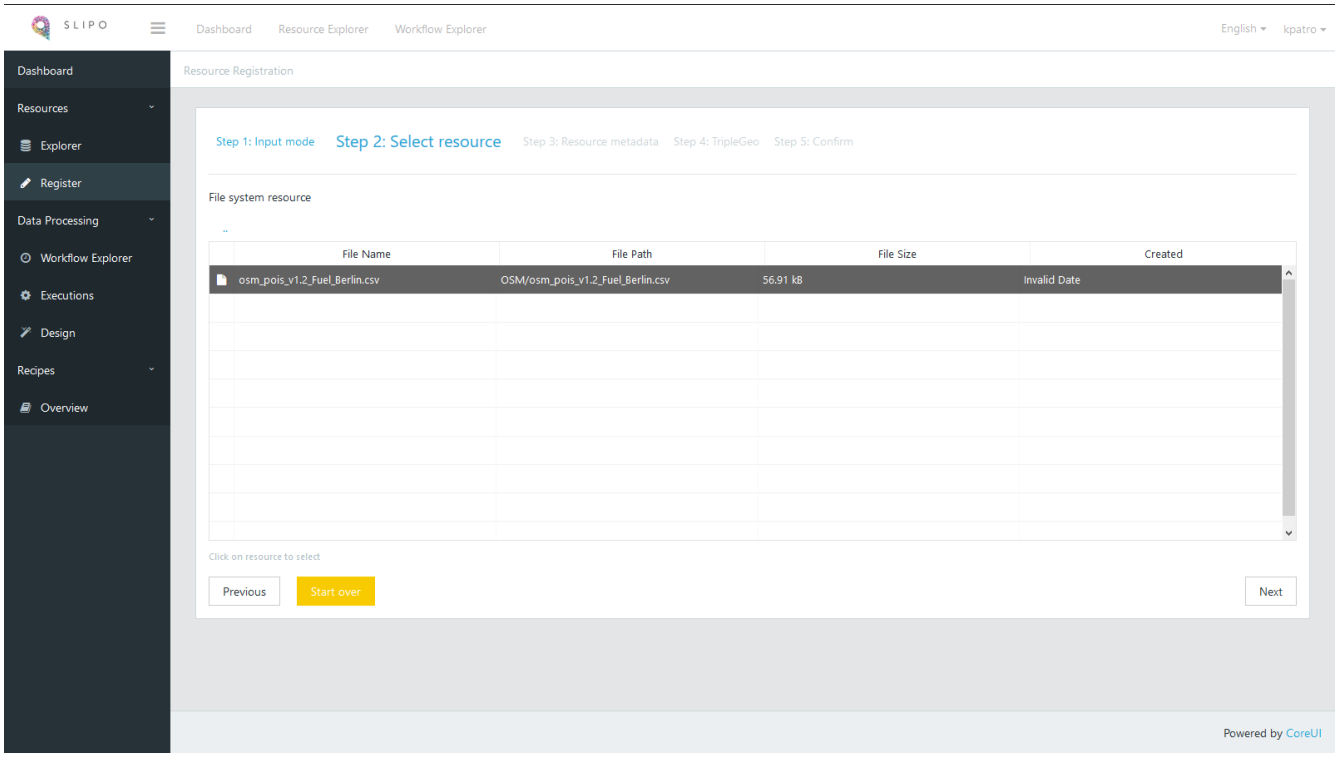


Figure 86: Select dataset for a resource

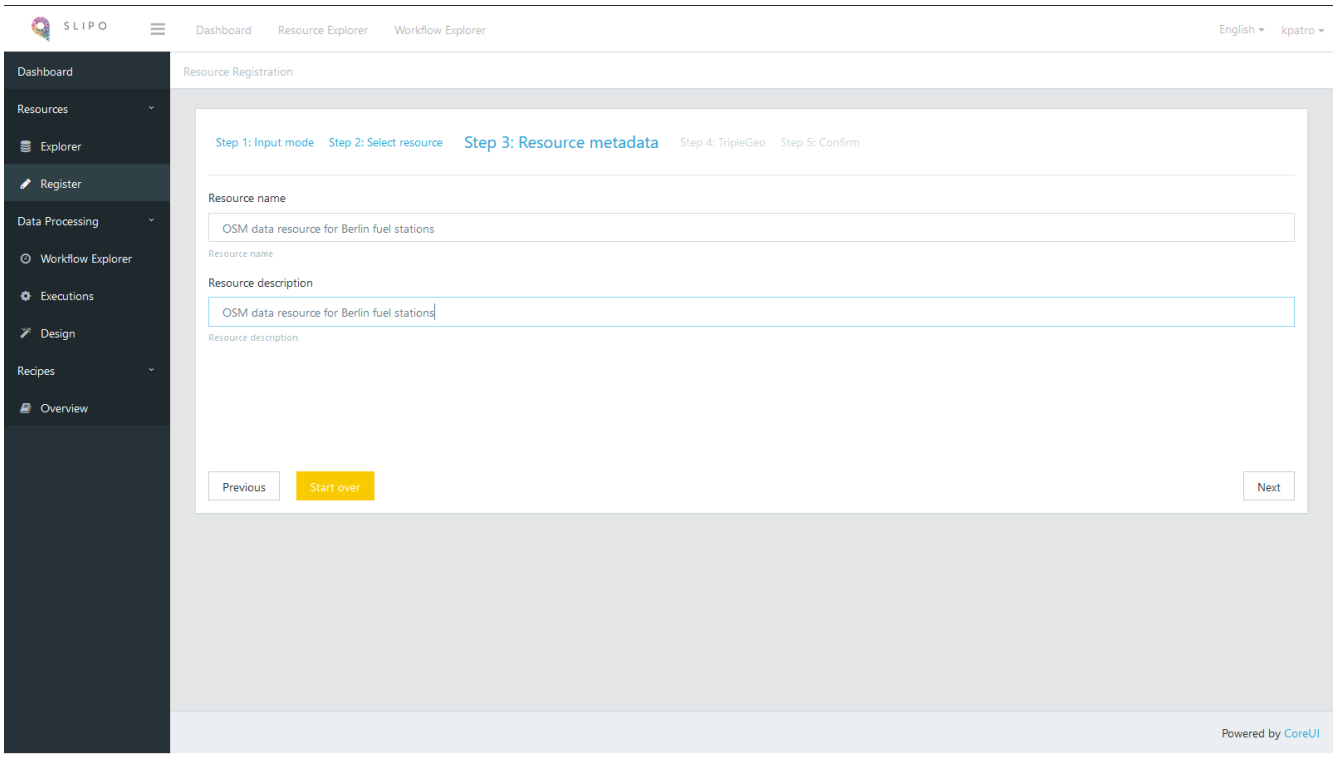


Figure 87: Assigning name and description to a resource

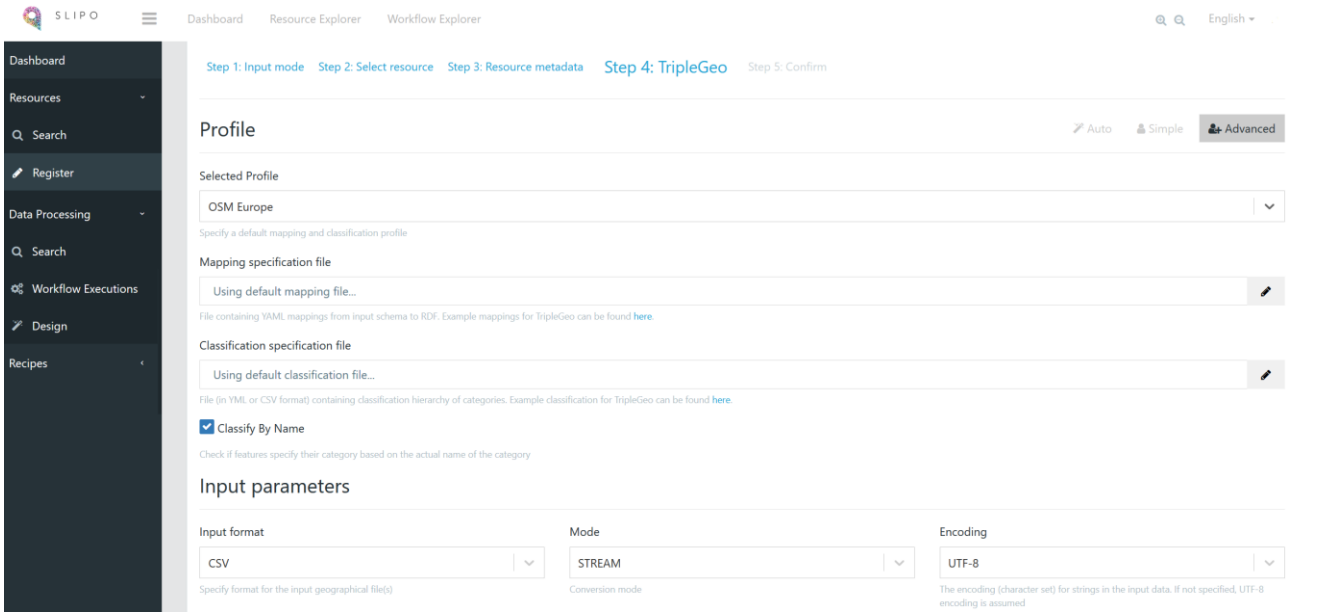


Figure 88: Configuring transformation properties for the data resource

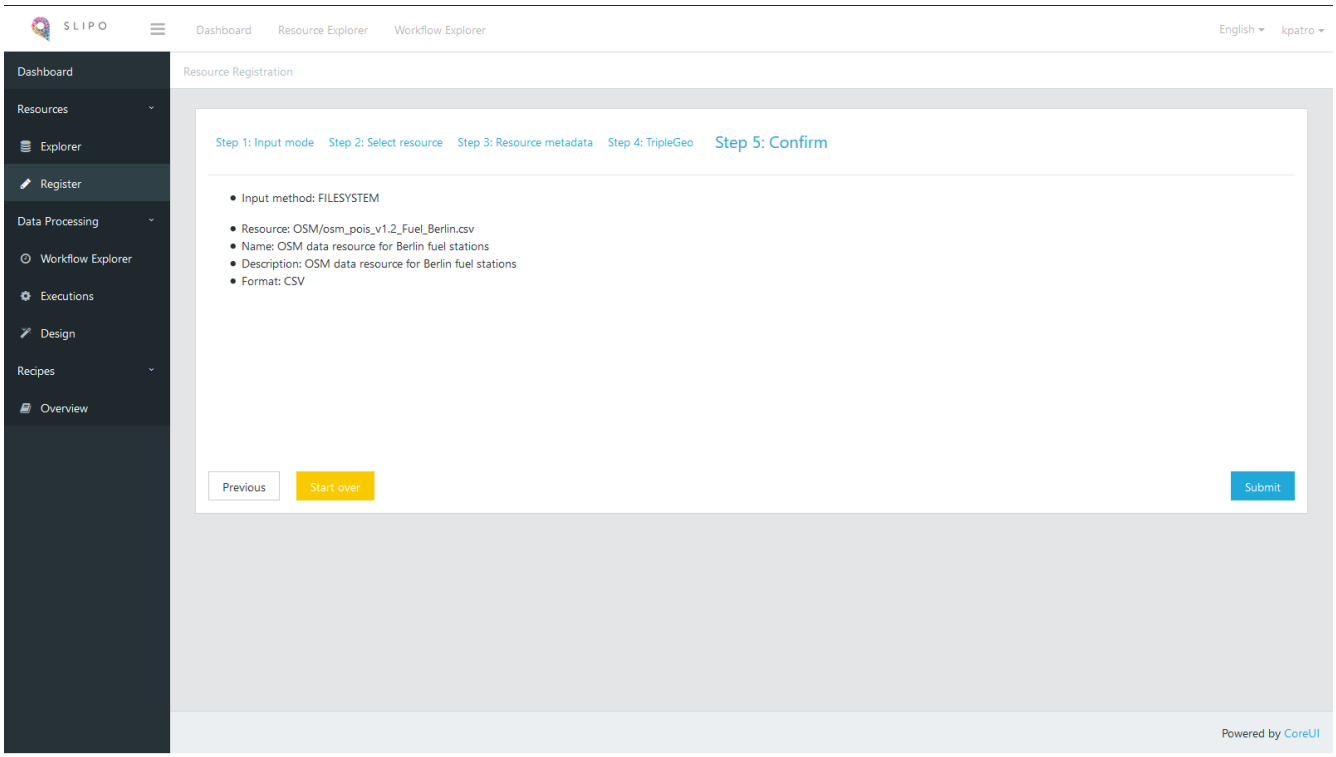


Figure 89: Summary of the resource definition

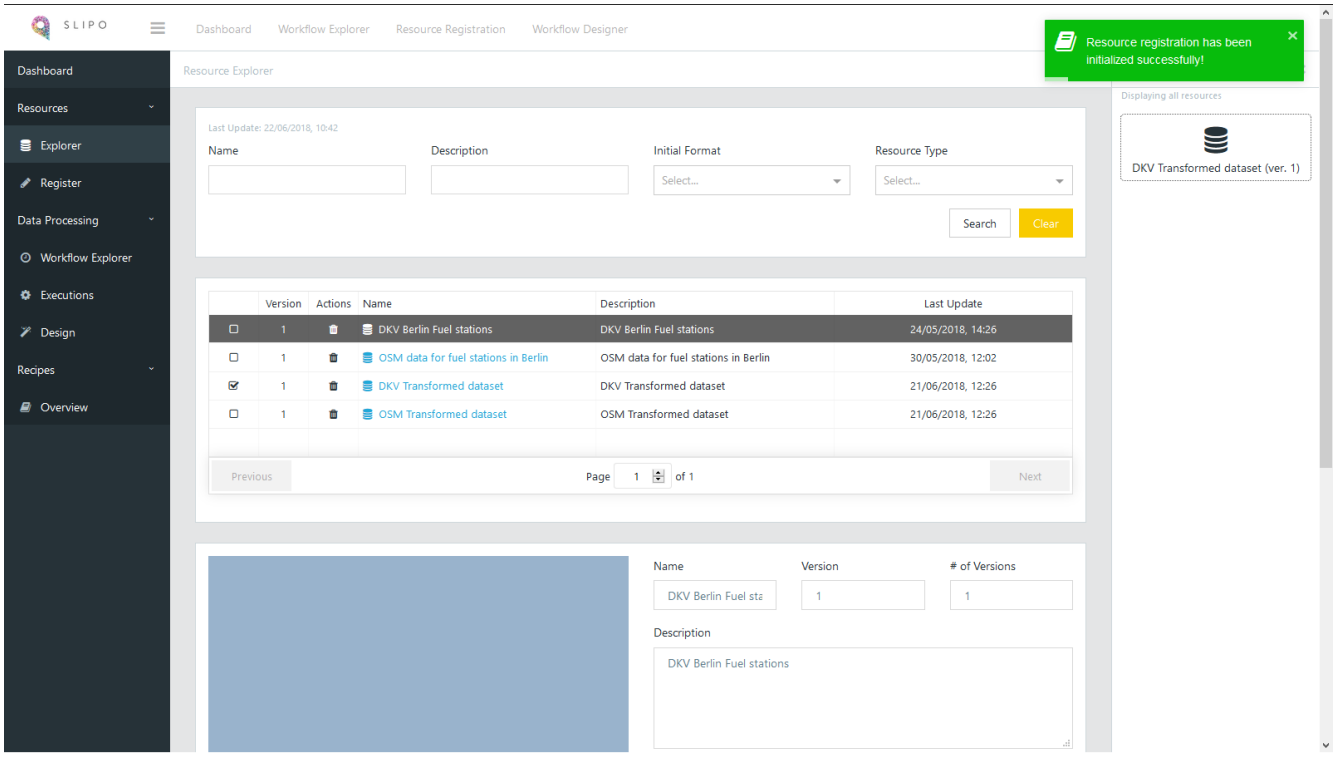


Figure 90: Submitting this resource registration task to be executed in the Workbench

2.2.4. Interlinking

Interlinking is the process of link discovery between entities contained in Linked Data sources. This process addresses the lack of common identifiers, by linking together different representations and instances of the same real-world entities found in different sources, thus tackling spatial, temporal, and semantic ambiguity in POI data, as well as the inherent multi-linguality of location-related information. **LIMES**, the *Link Discovery Framework for Metric Spaces*, is the software used in the SLIPO Workbench for discovering such links. It takes as input two RDF data sources and based on the mathematical characteristics of metric spaces as well prefix-, suffix- and position filtering, it can assess the similarity between entities appearing in both sources. For instance, it can indicate that two POIs (each coming from a different data source) written with slightly different name and located nearby are most probably the same POI. Interlinked results are issued with a similarity measure; values close to 1 indicate almost perfect matching.

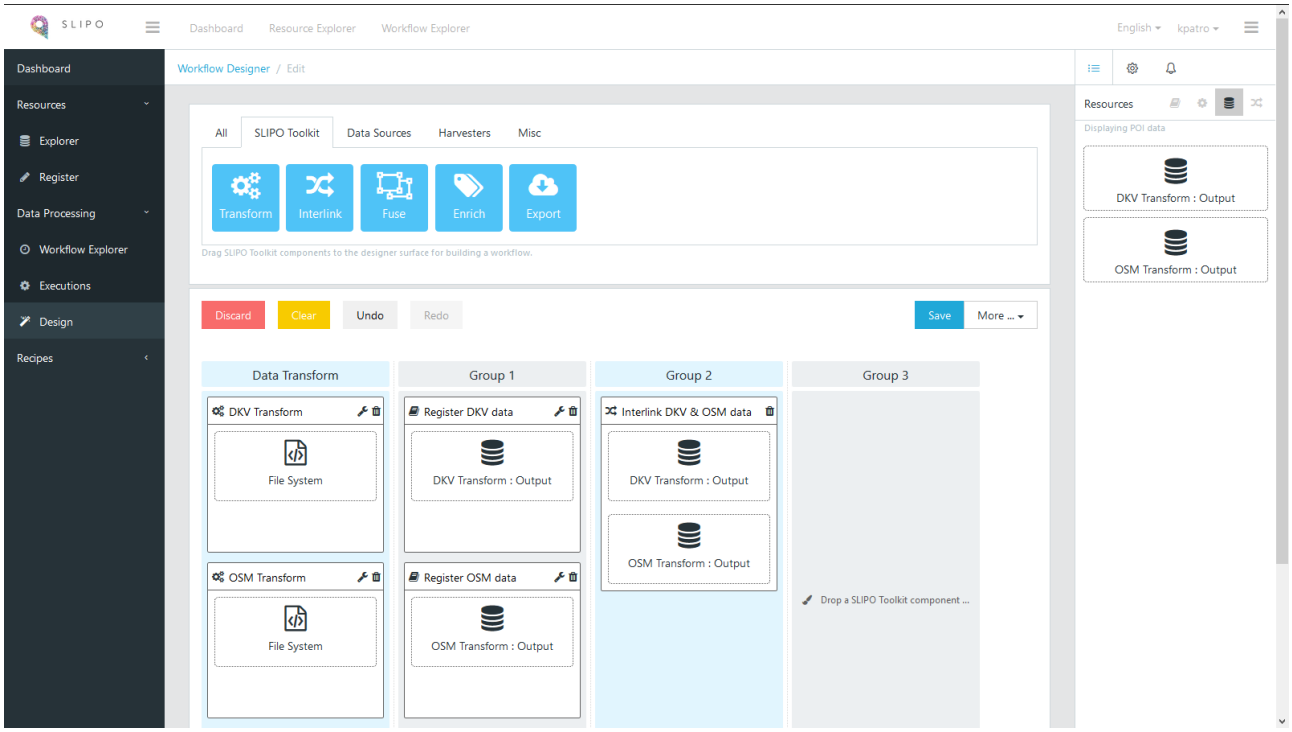


Figure 91: Adding an interlinking task to the workflow

To specify an interlinking task in the Workbench, the user simply drags an instance of the Interlink component to the design pane. Further, she needs to associate two RDF data sources as input, so that LIMES, our interlinking software, can discover links between them. In this example, those datasets have been produced after transformation in a previous step and in fact are already available as resources in the Workbench. As shown in Figure 91, these resources are listed on the right side, so the user can simply drag them inside the Interlink container. As both datasets comply with the SLIPO ontology for POIs, LIMES can take advantage of preconfigured settings in order to apply its link discovery methods. Similarly to the transformation step, interlinking with LIMES also provides an **Auto** mode (Figure 43). Once the process is properly defined and validated, the output of this interlinking task will be available as a *resource* in the Workbench.

2.2.5. Fusion

Following the interlinking of different representations of the same POI in multiple sources, *fusion* addresses the problem of assembling partial and incomplete POI profiles, as well as handling and resolving conflicting information, to derive a more complete and consolidated profile for each POI. In the SLIPO Workbench, **FAGI** is the platform that allows fusion of geospatial Linked Data, supporting several *thematic* and *spatial* fusion actions. Once invoked, FAGI receives two input RDF datasets and a list of linked entities between them (those identified earlier with LIMES), and produces a unified RDF dataset with the fused entities.

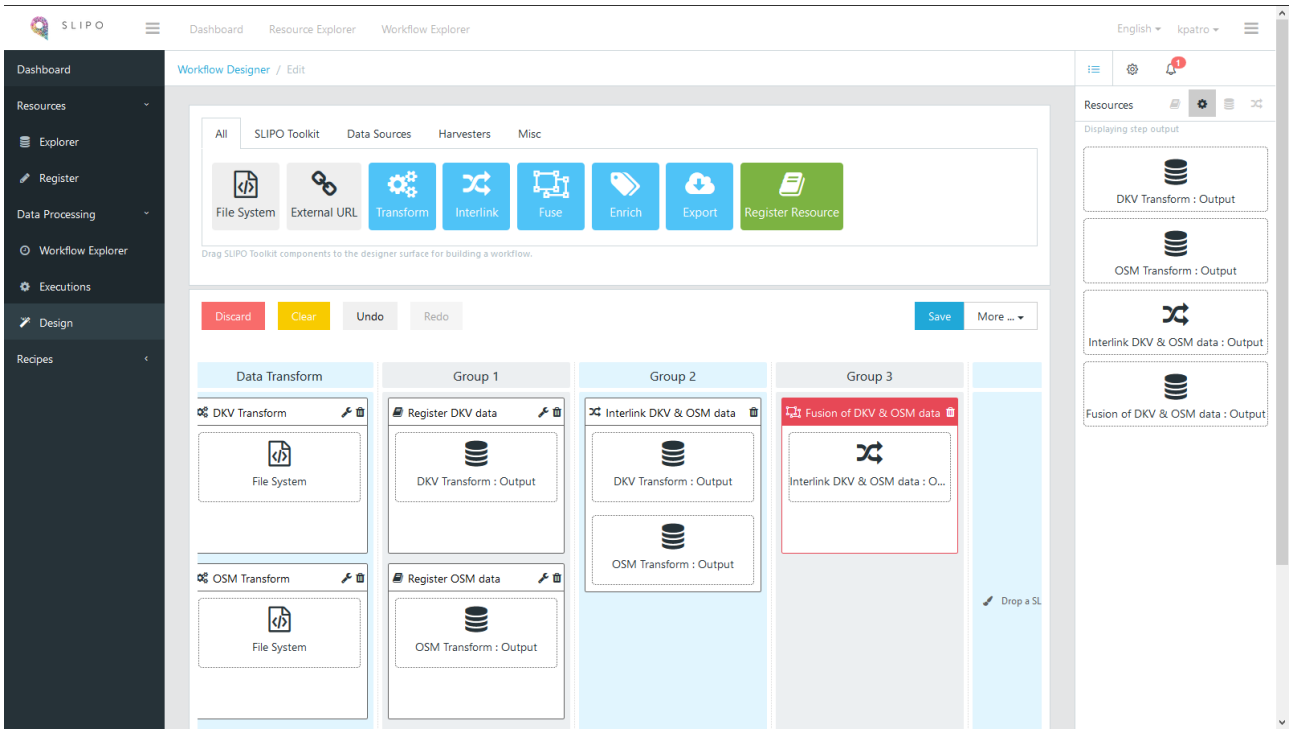


Figure 92: Adding a fusion task to the workflow

To specify a fusion task in the Workbench, the user simply drags an instance of a Fusion component to the design pane, as illustrated in Figure 92. Then, she must specify the list of interlinked entities, as derived from LIMES in the previous step. Note that this list is portrayed as a *resource* with the interlink symbol on the right side of the screen, so the user only needs to drag it inside the Fusion container and thus instruct the system to use these links in fusion. As those links were generated from two data sources, these are also implicitly associated with the fusion process. A fusion task also requires configuration, namely specification of the fusion mode and optionally rules that guide the fusion for each particular POI property. In *Auto* mode, the software takes advantage of an ML model (Figure 45), whereas in *Advanced* mode, the user can specify all details regarding the fusion task. Once the configuration is complete and the workflow is validated, the fusion output will also appear as a resource (on the right side of the design).

2.2.6. Enrichment

Exploiting the established links to other relevant data sources, each POI profile can be further enriched with extra knowledge related to its spatial, temporal and thematic context. In the SLIPO Workbench, **DEER** is the

data enrichment framework that applies enrichment functions and operators to discover implicit or explicit references of entities to external datasets.

As illustrated in Figure 93, the user can simply drag an instance of the Enrich component to the design pane and subsequently drag the result of the previous step (in this case, the fused dataset derived from FAGI) inside its container. This action will instruct DEER to enrich this fused dataset with a third data source. In this example, this third data concerns the OSM database, which will be accessed and extra information can be extracted, e.g., multi-lingual names, available services, etc. for POIs, which are not available in the OSM dataset of fuel stations for Berlin that was originally given as input to the workflow (Section 2.2.1). The user may specify a custom configuration setting for the enrichment component DEER, but an *Auto* mode is also available (Figure 47) with a pre-defined profile for initializing all configuration options that basically dictate retrieval of extra properties from DBpedia.

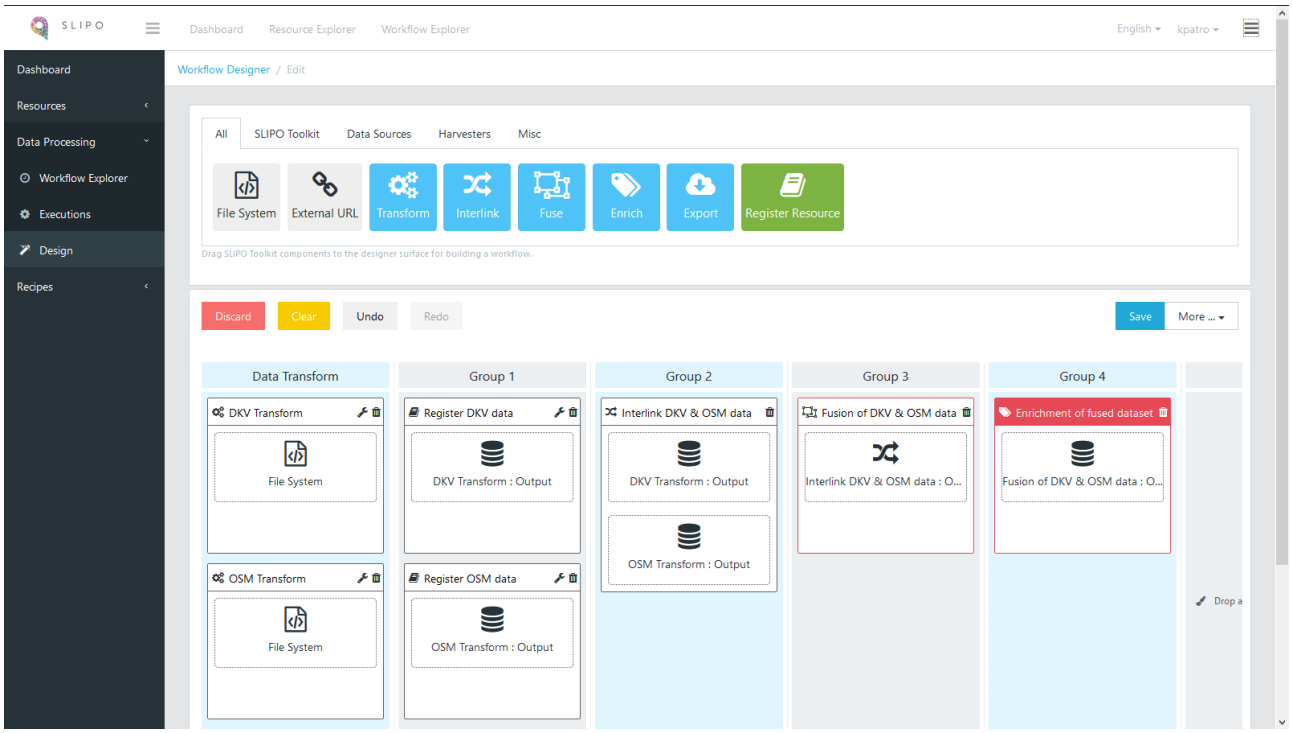


Figure 93: Adding an enrichment task to the workflow

The result of this enrichment is a single dataset that actually integrates all possible information available in the data. This dataset can be also declared as a resource (Figure 94), so that it can be readily utilized in further processing and/or other workflows in the SLIPO Workbench.

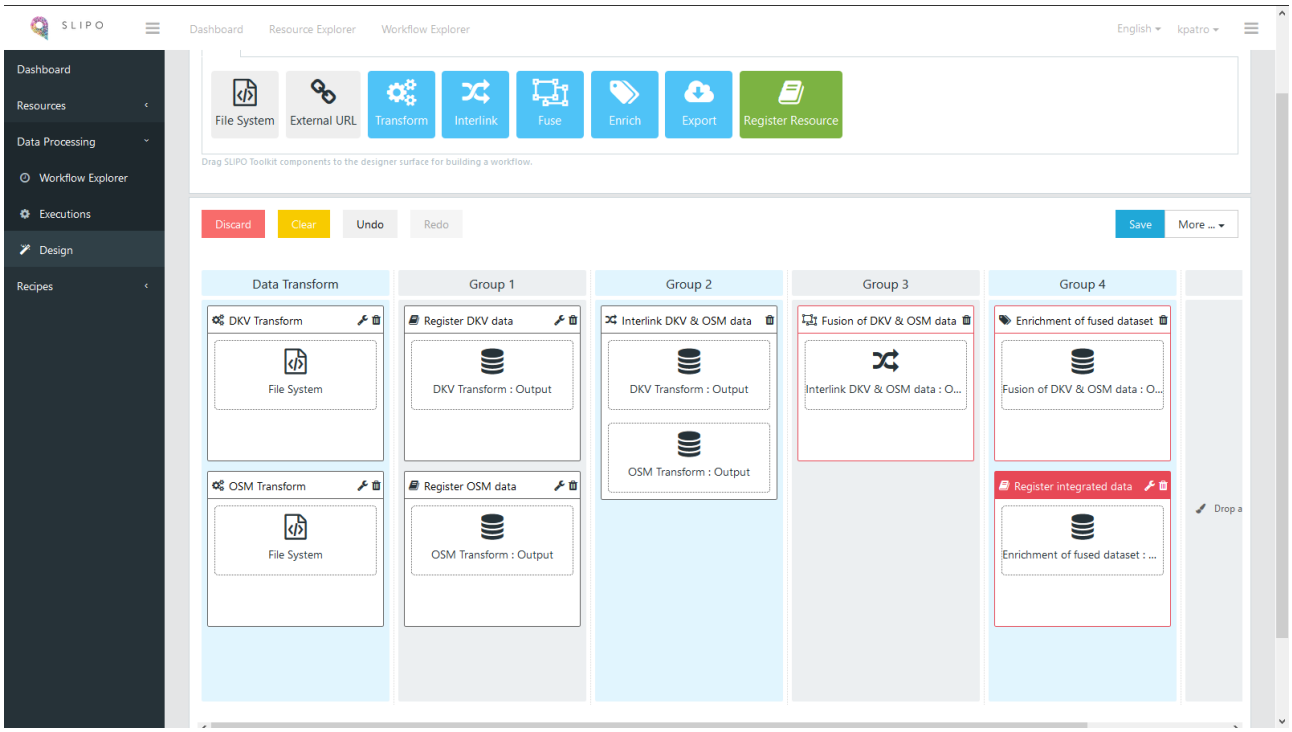


Figure 94: Creating a SLIPO resource from the enriched dataset

2.2.7. Export

After enrichment, the integrated POI data can be exported into a de facto geographical format in order to be used by other applications (e.g., GIS, DBMS) and third-party services. In this example, the user chooses to export the integrated POI dataset in a CSV file (comma separated values) making use of the *reverse transformation* utility included in TripleGeo. As illustrated in Figure 95, the user can simply drag an instance of the Export tool into the design and subsequently drag the result of the previous step (in this case, the enriched dataset derived from DEER) inside its container. It is possible to use a readily available *default* configuration settings (essential a SPARQL query that extracts records from the enriched RDF graph) or specify a *custom* configuration suitable for the data features at hand.

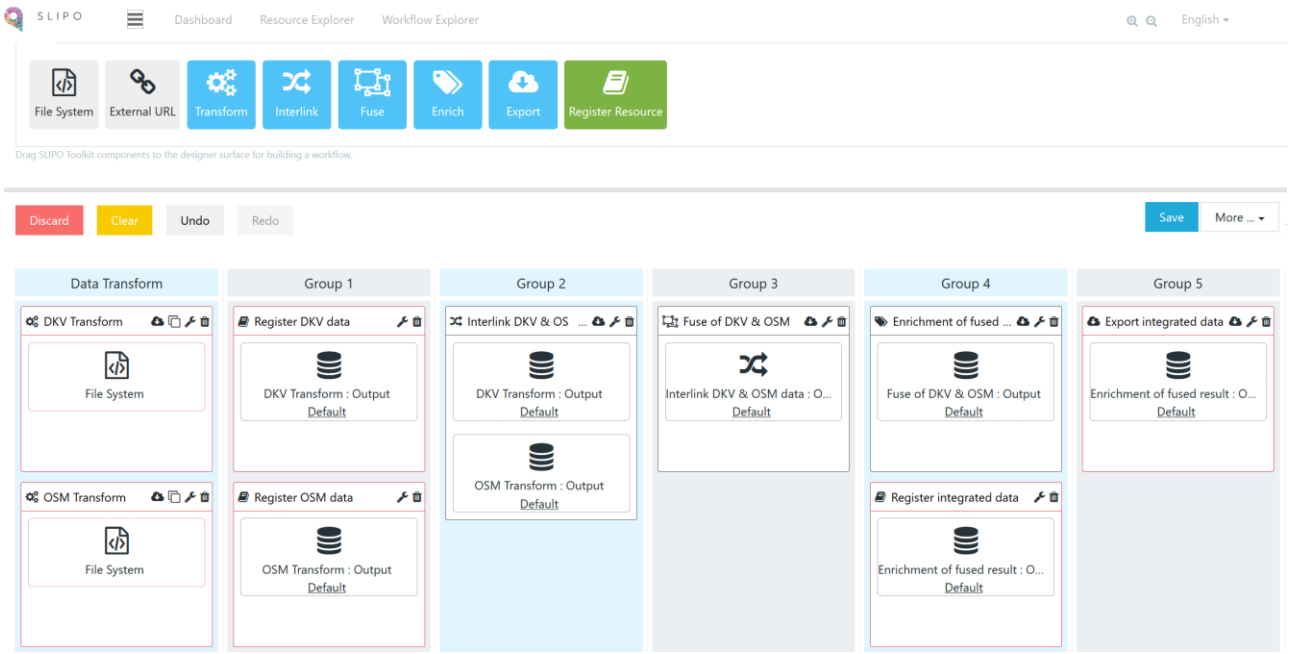


Figure 95: Adding an export (reverse transformation) task to the workflow

Once the user has finished designing the entire workflow, he/she can save it and execute it in order to invoke all successive stages and produce the results. The status of each execution is visible through the Executions option in the dashboard, as shown in Figure 96.

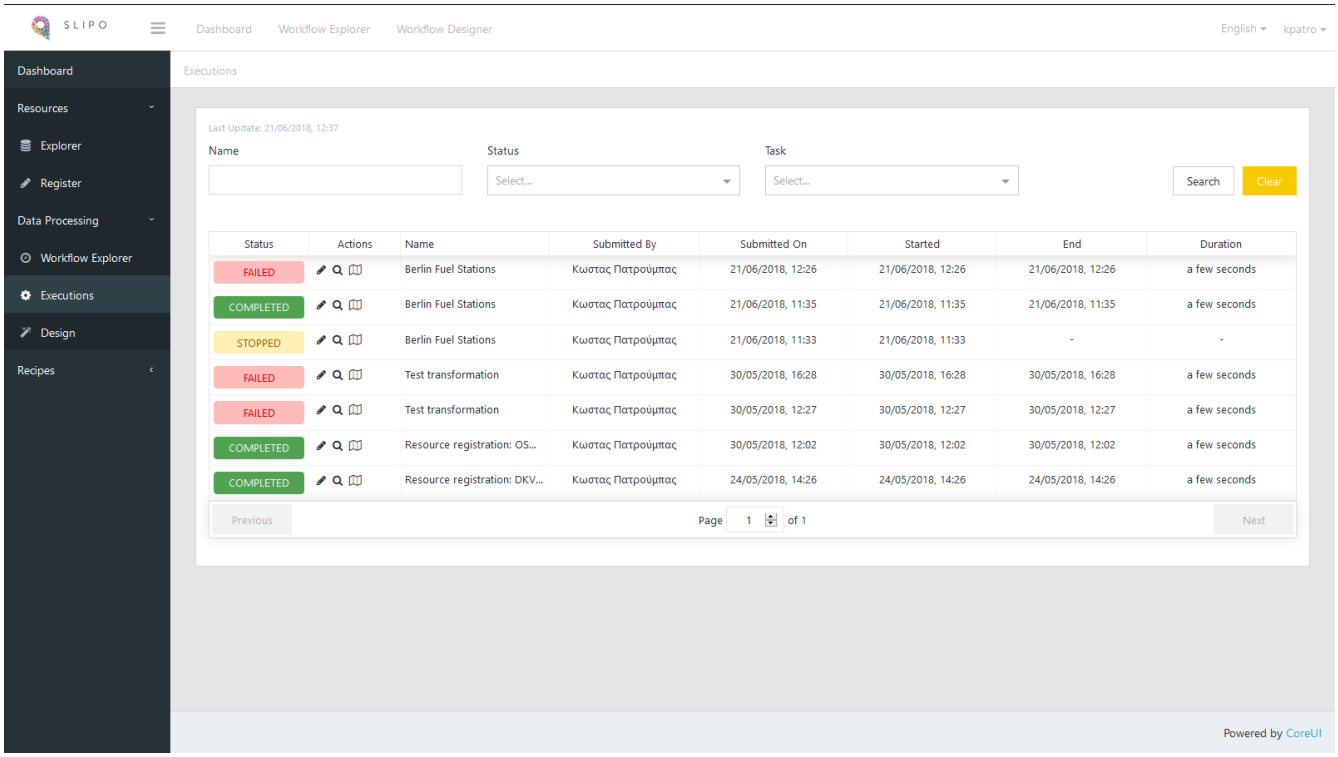


Figure 96: Execution status for workflows submitted by a user

2.2.8. Jupyter notebooks

The architecture of the SLIPO system, as well as its deployment over the HELIX computing infrastructure (*see Section 1.3*), enable the provision of its data integration services to external systems and users via simple, lightweight and interoperable end-points. These support integration in existing third-party systems and data integration workflows/pipelines (*e.g., invoke fusion operation or a data integration workflow as a service*), the development and provision of alternate UIs (*i.e., without requiring the direct use of the Workbench*), as well as interactive computing via Jupyter notebooks.

In the following, we present a walkthrough of how the SLIPO services support exploratory data interaction, manipulation, and integration within a Jupyter notebook. The notebook source file and data assets are available in our GitHub repository[45]. In this specific example:

- POI data assets are available after completing the Resource Registration process (*see 2.1.2 and 2.2.3*) or after being manually added by the user in the Jupyter environment; the assets are available in RDF (*n-triples format*) and can be manipulated through RDFLib.
- The kernel of the notebook is instantiated with SLIPOFrames[46], a lightweight library we developed exposing the SLIPO API services to the Python environment (CLI and Jupyter, see Section 1.2.3).
- The user holds a valid API key associated with her account, authorizing use of her resources, data integration workflows, and SLIPO toolkit components.
- The user can invoke an already available data integration workflow, create new data integration workflows by invoking the available SLIPO Toolkit components, or author custom data processing pipelines arbitrarily mixing ad hoc data processing tasks (*expressed in Python*) with SLIPO Toolkit services.

[45] https://github.com/SLIPO-EU/slipo-frames/blob/master/notebooks/Demo_04_POI_Integration_Workflow_Analytics.ipynb

[46] https://pypi.org/project/slipoframes/

Setup

To run the example, a file named `secret.py` must be created in the notebooks folder with the following content:

```
# Configuration settings

# SLIPO workbench installation
BASE_URL = 'https://app.dev.slipo.eu'

# SLIPO API key
API_KEY = ''
```

The `API_KEY` value must be set to a valid SLIPO Application Key. The file must be imported before creating a new context:

```
from secret import BASE_URL, API_KEY
```

In [5]:
```
# Create new context
from slipoframes.context import SlipoContext

from secret import BASE_URL, API_KEY

ctx = SlipoContext(
    base_url = BASE_URL,
    requires_ssl = False,
    api_key = API_KEY
)
```

```
Application key is valid!
```

Transform operation

Next we are going to:

- Upload the files `DKV_Fuel_Berlin.csv` and `OSM_Fuel_Berlin.csv` from the local folder `datasets` to the remote folder `notebooks/datasets`. The remote folder will be created automatically if not already exists. The option `overwrite` is also set to `True` to overwrite any existing files.
- Upload the contents of folder `config` from the local file system to the remote folder `notebooks/config`.
- Execute two transform operations to convert the `CSV` data to `N-Triples`.
- Check the status of each operation

In [2]:
```
# Upload file DKV_Berlin.csv
ctx.file_upload('./datasets/DKV_Fuel_Berlin.csv', 'notebooks/datasets/DKV_Fuel_Berlin.csv', overwrite=True)
```

```
File [./datasets/DKV_Fuel_Berlin.csv] uploaded successfully
```

In [3]:
```
# Upload file DKV_Berlin.csv
ctx.file_upload('./datasets/OSM_Fuel_Berlin.csv', 'notebooks/datasets/OSM_Fuel_Berlin.csv', overwrite=True)
```

```
File [./datasets/OSM_Fuel_Berlin.csv] uploaded successfully
```

In [4]:
```
# Upload all files in the config folder
ctx.file_upload('./config', 'notebooks/config', overwrite=True)
```

```
File [notebooks/config/DKV_POI_sample_classification.csv] uploaded successfully
File [notebooks/config/OSM_POI_sample_classification.csv] uploaded successfully
File [notebooks/config/DKV_Fuel_Berlin_slipo_mappings.yml] uploaded successfully
File [notebooks/config/OSM_Fuel_Berlin_slipo_mappings.yml] uploaded successfully
```

In [5]:
```
# Browse remote user file system
df_files = ctx.file_browse(sort_col='size', format_size=True, sort_asc=False)

df_files[df_files['path'].str.startswith("notebooks")]
```

Out[5]:

	modified	name	path	size
28	2019-11-12 13:12:07	OSM_Fuel_Berlin.csv	notebooks/datasets/OSM_Fuel_Berlin.csv	56.9 kB
27	2019-11-12 13:12:04	DKV_Fuel_Berlin.csv	notebooks/datasets/DKV_Fuel_Berlin.csv	28.0 kB
26	2019-11-12 13:12:09	OSM_POI_sample_classification.csv	notebooks/config/OSM_POI_sample_classification...	4.3 kB
24	2019-11-12 13:12:09	DKV_POI_sample_classification.csv	notebooks/config/DKV_POI_sample_classification...	4.3 kB
25	2019-11-12 13:12:09	OSM_Fuel_Berlin_slipo_mappings.yml	notebooks/config/OSM_Fuel_Berlin_slipo_mapping...	1.8 kB
23	2019-11-12 13:12:09	DKV_Fuel_Berlin_slipo_mappings.yml	notebooks/config/DKV_Fuel_Berlin_slipo_mapping...	822.0 B

In [6]:
```
# Convert DKV_Fuel_Berlin.csv file to N-Triples
transform1 = ctx.transform_csv(
    'notebooks/datasets/DKV_Fuel_Berlin.csv',
    attrCategory='all_tags',
    attrKey='ID',
    attrName='name',
    attrX='lon',
    attrY='lat',
    delimiter=';',
    featureSource='DKV',
    quote='',
    mappingSpec='notebooks/config/DKV_Fuel_Berlin_slipo_mappings.yml',
    classificationSpec='notebooks/config/DKV_POI_sample_classification.csv'
)
```

```
Process (380, 1) status is RUNNING
```

In [7]:
```
# Convert OSM_Fuel_Berlin.csv file to N-Triples
transform2 = ctx.transform_csv(
    'notebooks/datasets/OSM_Fuel_Berlin.csv',
    attrCategory='type',
    attrGeometry='wkt',
    attrKey='osm_id',
    attrName='name',
    attrX='lon',
    attrY='lat',
    delimiter='|',
    featureSource='OpenStreetMap',
    profile='OSM_Europe',
    quote='',
    mappingSpec='notebooks/config/OSM_Fuel_Berlin_slipo_mappings.yml',
    classificationSpec='notebooks/config/OSM_POI_sample_classification.csv'
)
```

```
Process (381, 1) status is RUNNING
```

In [10]:
```
# Check process status for transform operations
transform1 = ctx.process_status(transform1)
transform2 = ctx.process_status(transform2)
```

```
Process (380, 1) status is COMPLETED
Process (381, 1) status is COMPLETED
```

In [10]:
```
# Check process status for transform operations
transform1 = ctx.process_status(transform1)
transform2 = ctx.process_status(transform2)
```

```
Process (380, 1) status is COMPLETED
Process (381, 1) status is COMPLETED
```

Interlink operation

Execute an interlink operation on the RDF datasets generated by the previous two transformation operations

In [11]:
```
interlink1 = ctx.interlink(
    'SLIPO_equiMatchByNameAndDistance',
    left=transform1.output(),
    right=transform2.output()
)
```

```
Process (382, 1) status is RUNNING
```

In [12]:
```
# Check process status for interlink operation
interlink1 = ctx.process_status(interlink1)
```

```
Process (382, 1) status is COMPLETED
```

Fuse Operation

Fuse the two RDF datasets generated by operations `transform1` and `transform2` using the links from operation `interlink`

In [13]:
```
fuse1 = ctx.fuse(
    'SLIPO_default_abMode',
    left=transform1.output(),
    right=transform2.output(),
    links=interlink1.output()
)
```

```
Process (383, 1) status is RUNNING
```

In [16]:
```
# Check process status for fuse operation
fuse1 = ctx.process_status(fuse1)
```

```
Process (383, 1) status is COMPLETED
```

Enrich Operation

Enrich the fused RDF dataset from operation `fuse1`

In [17]:
```
enrich1 = ctx.enrich(
    'SLIPO_TomTom_Bucharest',
    source=fuse1.output()
)
```

```
Process (384, 1) status is RUNNING
```

In [21]:
```
# Check process status for enrich operation
enrich1 = ctx.process_status(enrich1)
```

```
Process (384, 1) status is COMPLETED
```

Export Operation

Export the enriched RDF dataset to a CSV file

In [22]:
```
export1 = ctx.export_csv(
    'SLIPO_default',
    enrich1.output(),
    delimiter='|',
    quote=""
)
```

```
Process (385, 1) status is RUNNING
```

In [24]:
```
# Check process status for export operation
export1 = ctx.process_status(export1)
```

```
Process (385, 1) status is COMPLETED
```

In [25]:
```
# Copy output file to local file system
ctx.process_file_download(export1.output(), target='./output/exported-data.zip', overwrite=True)
```

```
Process file (385, 1, 25451) copied to ./output/exported-data.zip successfully
```

Execute an existing workflow

Run a prespecified data integration workflow that involves all stages (transformation, interlinking, fusion, enrichment, export).

Identify the workflow, its various versions and their executions:

In [6]:
```
processes = ctx.process_query(
    'Integrate OSM & DKV data in Berlin (updated)',
    0,
    10
)

processes[['Id','Name','Executed On','Version']]
```

Out[6]:

	Id	Name	Executed On	Version
0	352	Integrate OSM & DKV data in Berlin (updated)	2019-11-12 10:25:58.958	1
1	352	Integrate OSM & DKV data in Berlin (updated)	2019-11-11 15:50:31.288	2
2	352	Integrate OSM & DKV data in Berlin (updated)	2019-11-12 10:29:35.733	3
3	352	Integrate OSM & DKV data in Berlin (updated)	2019-11-12 10:36:46.186	4
4	352	Integrate OSM & DKV data in Berlin (updated)	2019-11-12 10:46:31.034	5
5	352	Integrate OSM & DKV data in Berlin (updated)	2019-11-12 12:39:24.626	6
6	352	Integrate OSM & DKV data in Berlin (updated)	2019-11-12 13:13:58.796	7
7	352	Integrate OSM & DKV data in Berlin (updated)	2019-11-12 13:14:57.297	8

Execute a new version of this workflow:

In [30]:
```
ctx.process_start(352, 8)
```

```
Process (352, 8) started successfully
```

Inspect the status of this workflow execution:

In [3]:
```
workflow1 = ctx.process_status(352, 8)
```

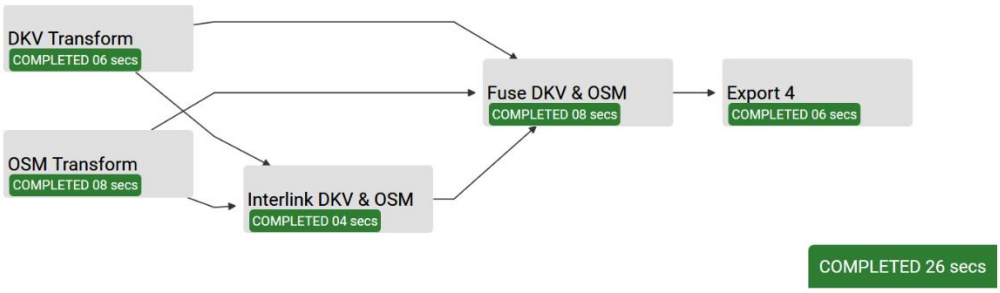
```
Process (352, 8) status is COMPLETED
```

Render this workflow as a graph with all its components:

Render this workflow as a graph with all its components:

In [4]:
```
ctx.process_render(workflow1)
```

Out[4]:



POI Data Analytics

Once integrated POI data has been saved locally, analysis can be perfomed using tools like pandas DataFrames, geopanadas GeoDataFrames or other libraries.

In [33]:
```
# Unzip exported CSV file with the results of data integration
import os
import zipfile

with zipfile.ZipFile('./output/exported-data.zip','r') as zip_ref:
    zip_ref.extractall("./output/")

os.rename('./output/points.csv', './output/Fuel_Berlin.csv')
```

In [34]:
```
# Load CSV data in a DataFrame
import pandas as pd

pois = pd.read_csv('./output/Fuel_Berlin.csv', delimiter='|', error_bad_lines=False)

# Geometries in the exported CSV file are listed in Extended Well-Known Text (EWKT)
# Since shapely does not support EWKT, update the geometry by removing the SRID value from EWKT
pois['the_geom'] = pois['the_geom'].apply(lambda x: x.split(';')[1])

pois.head()
```

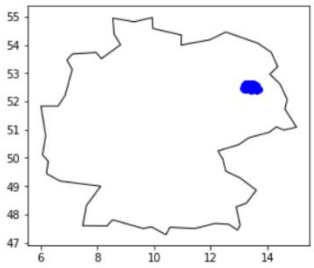
Out[34]:

	uri	id	source	lon	lat	the_geom	category	name	description	phone	...	email	homepage	
0	http://slipo.eu/id/poi/4b2fbd07-55e1-4941-9115...	N432779736	NaN	13.466368	52.596739	POINT (13.4663679 52.5967386)	FUEL	Shell	NaN	NaN	...	NaN	NaN	
1	http://slipo.eu/id/poi/7edc53fb-d632-44ce-b59f...	44	NaN	13.449799	52.496865	POINT (13.4497993 52.4968646)	NaN	ARAL	NaN	NaN	...	NaN	NaN	Schle
2	http://slipo.eu/id/poi/3782eb56-2264-4f52-8719...	152	NaN	13.432808	52.558474	MULTIPOLYGON (((13.4328366 52.5583578, 13.4326...	NaN	Elan	NaN	NaN	...	NaN	NaN	Schv
3	http://slipo.eu/id/poi/e916f77f-6fb5-4806-8848...	N4583828565	NaN	13.421895	52.458758	POINT (13.4218947 52.4587583)	FUEL	Hans Engelke Energie	NaN	NaN	...	NaN	http://hans-engelke.de/	Tem
4	http://slipo.eu/id/poi/c17b79aa-d273-4fdf-b788...	156	NaN	13.429671	52.571729	POINT (13.4296708 52.5717292)	NaN	Shell	NaN	NaN	...	NaN	NaN	Rother

5 rows × 21 columns

In [35]:
```
# Create a GeoDataFrame
import geopandas
from shapely import wkt

pois['the_geom'] = pois['the_geom'].apply(wkt.loads)

gdf = geopandas.GeoDataFrame(pois, geometry='the_geom')
```

In [36]:
```
# Display the location of the exported POIs on a simple plot using matplotlib
%matplotlib inline

import matplotlib.pyplot as plt

world = geopandas.read_file(geopandas.datasets.get_path('naturalearth_lowres'))

# Restrict focus to Germany:
ax = world[world.name == 'Germany'].plot(
    color='white', edgecolor='black')

# Plot the contents of the GeoDataFrame in blue dots:
gdf.plot(ax=ax, color='blue')

plt.show()
```



POI Data Analytics using LOCI

Perform spatial analytics over the integrated POI data.

In [38]:
```
# LOCI dependencies:
import loci as lc
from loci import io
from loci import analytics
from loci import plots
```

In [39]:
```
# Create a dataframe from the integrated POI dataset:
pois = lc.io.read_poi_csv(input_file='./output/Fuel_Berlin.csv',
                          col_id='id',
                          col_name='uri',
                          col_lon='lon',
                          col_lat='lat',
                          col_kwds='name',
                          col_sep='|',
                          kwds_sep=',',
                          source_crs='EPSG:4326',
                          target_crs='EPSG:4326',
                          keep_other_cols=False)

# Turn all names in uppercase characters to facilitate comparison:
pois['name'] = pois['name'].apply(lambda x: [element.upper() for element in x])

pois.head(10)
```

```
Loaded 373 POIs.
```

Out[39]:

	uri	id	name	geometry
0	http://slipo.eu/id/poi/4b2fbd07-55e1-4941-9115...	N432779736	[SHELL]	POINT (13.4663679 52.59673860000001)
1	http://slipo.eu/id/poi/7edc53fb-d632-44ce-b59f...	44	[ARAL]	POINT (13.4497993 52.49686460000001)
2	http://slipo.eu/id/poi/3782eb56-2264-4f52-8719...	152	[ELAN]	POINT (13.4328085 52.55847360000001)
3	http://slipo.eu/id/poi/e916f77f-6fb5-4806-8848...	N4583828565	[HANS ENGELKE ENERGIE]	POINT (13.4218947 52.4587583)
4	http://slipo.eu/id/poi/c17b79aa-d273-4fdf-b788...	156	[SHELL]	POINT (13.4296708 52.5717292)
5	http://slipo.eu/id/poi/16e8e79c-a156-4a55-8df9...	N1406258332	[SHELL]	POINT (13.4324401 52.60098570000001)
6	http://slipo.eu/id/poi/318f9611-81e6-4455-81f2...	185	[AGIP]	POINT (13.3467029 52.5915272)
7	http://slipo.eu/id/poi/00e9a9cb-1468-4e5f-af0f...	187	[ESSO]	POINT (13.312269 52.62256600000001)
8	http://slipo.eu/id/poi/8e44f29c-744a-4ad1-a853...	W123644874	[GO]	POINT (13.4293174674407 52.4845719560997)
9	http://slipo.eu/id/poi/14a83e18-9fc9-42f6-a1af...	132	[ARAL]	POINT (13.5386157 52.5101421)

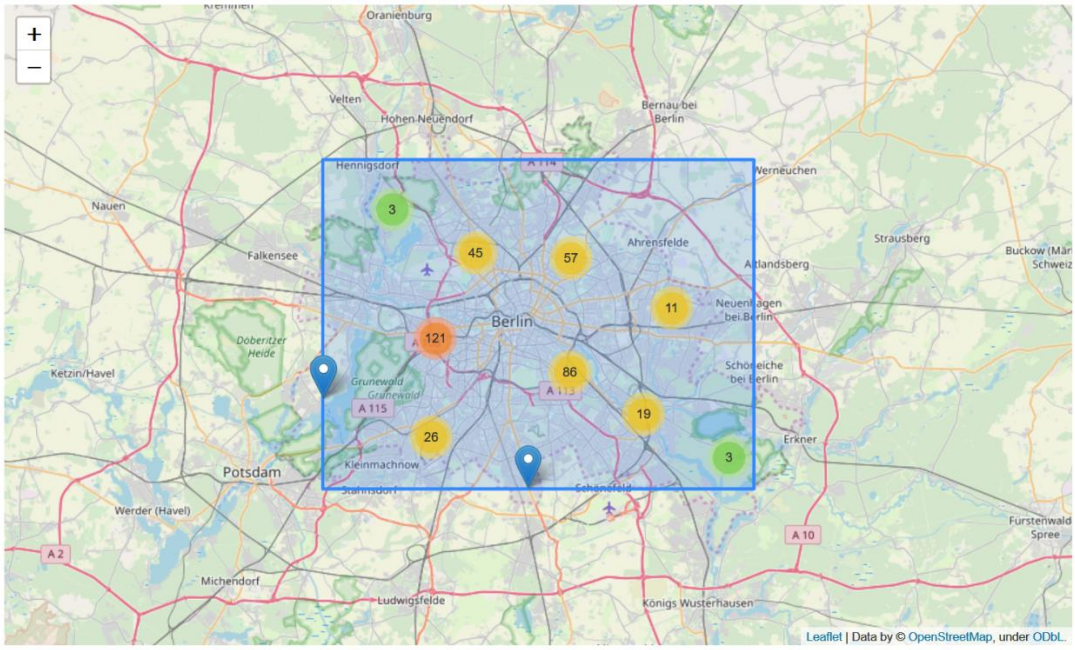
Utilize the name of the various brands as keywords for spatial analytics:

In [40]:
```
pois.rename(columns={'name': 'kwds'}, inplace=True)
pois.rename(columns={'uri': 'name'}, inplace=True)
```

Draw locations on map:

```
In [41]: m = lc.plots.map_points(pois, show_bbox=True)
         m
```

Out[41]:



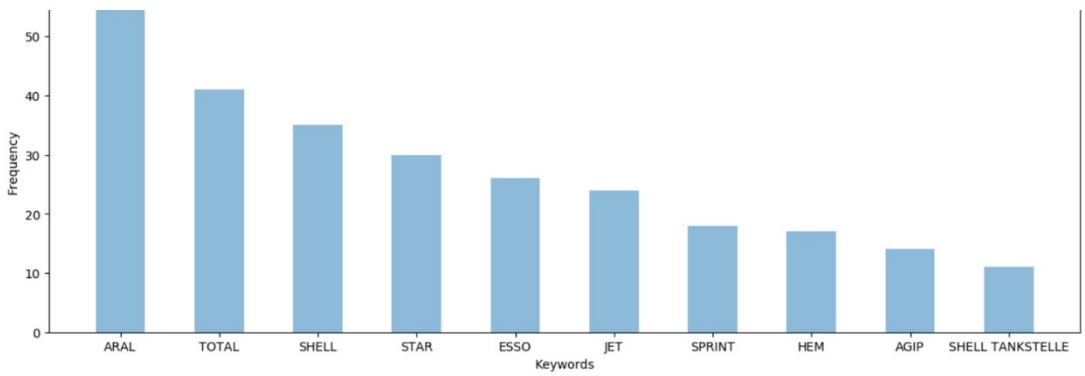
Statistics on the number of fuel stations per brand name

```
In [42]: kf = lc.analytics.kwds_freq(pois)
         kf
```

```
Out[42]: {'SHELL': 35,
          'ARAL': 55,
          'ELAN': 6,
          'HANS ENGELKE ENERGIE': 1,
          'AGIP': 14,
          'ESSO': 26,
          'GO': 3,
          'HEM': 17,
          'SPRINT': 18,
          'ARAL TANKSTELLE': 10,
          'TOTAL': 41,
          'ESSO TANKSTELLE': 4,
          'STAR': 30,
          'JET': 24,
          'HOYER': 2,
          'BEHALA': 1,
          'CLEAN CAR': 2,
          'MR. WASH': 2,
          'TAMOIL': 2,
          'TP24AUTOMAT': 1,
          'TANKSTELLE KRUMME LANKE': 1,
          'KAUFLAND': 1,
          'SUN': 7,
```

Chart showing fuel stations per brand name:

```
In [43]: lc.plots.barchart(kf, plot_title='Top Keywords', x_axis_label='Keywords', y_axis_label='Frequency')
Out[43]: <module 'matplotlib.pyplot' from '/home/kpatro/anaconda3/lib/python3.7/site-packages/matplotlib/pyplot.py'>
```



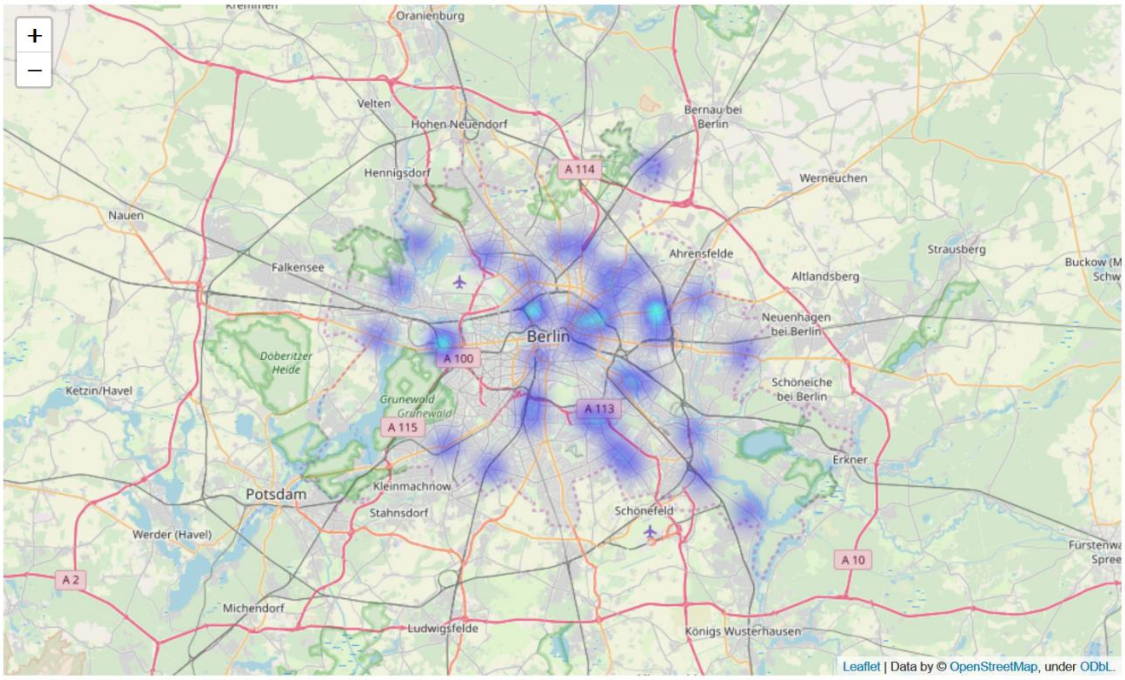
Word cloud of the various brands in the dataset:

```
In [44]: lc.plots.plot_wordcloud(pois)
```

Heatmap of the fuel stations belonging to a particular brand:

```
In [45]: pois_filtered = lc.analytics.filter_by_kwd(pois, 'TOTAL')
         lc.plots.heatmap(pois_filtered, radius=12)
```

Out[45]:



3. World-scale data integration

One of our original goals during the inception of the SLIPO system was to efficiently support *world-scale* POI data integration scenarios, i.e., data integration workflows involving POIs assets in the order of **100M POIs**. This ambitious goal has been framed based on the *current* available sizes of proprietary Big POI data assets available world-wide. Therefore, it consists a *pragmatic upper limit* regarding the maximum size of assets a POI data integration system would be required to tackle. Besides demonstrating the technical merits of our work, achieving this goal also has a clear business motivation as it enables the transfer of existing POI-related products and services in a cross-border manner. Specifically:

- **Big POI assets**. Our market assessment of *open and proprietary POI data assets* at a world-scale sets their maximum number at roughly 100M POIs. The **Google Places API**, one of the largest POI datasets and services worldwide, provides information about roughly **140M** POIs[47]. **HERE's Places (Search) API** offers information about over **61M** named and categorized places in **238** countries[48]. TomTom's POI database includes roughly **45M** POIs world-wide. Finally, from OSM, the largest open geospatial knowledge base, we have managed to extract **~18.5M** POIs. Despite the disparity of the various vendors, which is attributed to a certain extent to their varying levels of *quality* (*e.g., TomTom manually vets all POIs, while Google also uses user-contributed information*) and the semantics of what a POI is (*e.g., an office building vs. a post box, a place-name vs. an actual POI*), it is safe to consider that a size of 100M POIs is a pragmatic *upper bound* on the data sizes involved in POI data integration scenarios.
- ***Cross-border service provision***. The geospatial value chain is characterized by a strong locality in terms of coverage and market penetration. With the exception of the few international geospatial asset and service providers (*e.g., Google, TomTom, HERE*) the majority of businesses (*and especially EU SMEs*) offering products and value-added services built on POI assets are active on a very narrow geospatial setting. For example, a company specialized in geo-marketing (*such as our partner WiGeoGIS*) *could* apply its analyses methodology, workflows, and expertise (*easily transferable and proven intangible assets*), beyond DACH countries and across the EU and USA. However, as we have discussed in Deliverable D5.1 "Pilot Specifications", the actual challenge prohibiting the company from making this leap, is the availability of high-quality and fit-for-use POI data assets. Even for assets that the company knows *intimately*, extensive resource-intensive and time-consuming POI data integration workflows are required. For markets in other geographical areas, in which data sources are unknown, addressing these data integration challenges is even more critical, to the point of being materialized as a virtual market barrier for growth[49]. Hence, a system supporting POI data integration at a *world-scale* could lower the entry barrier to other markets and introduce additional revenue streams for geospatial value chain stakeholders.

[47] https://developers.google.com/places/

[48] https://developer.here.com/documentation/places/topics/coverage.html

[49] See D5.2 "Pilot Evaluation" (Confidential) in which a geo-marketing use case was successfully adapted for the Asia market.

In the following sub-sections, we present our experimental methodology for assessing the scalability of the SLIPO system in world-scale POI data integration scenarios. First, we present how we produced the synthetic POI data used, as well as a suitable data integration workflow that ensured a realistic workload for the components of the SLIPO Toolkit. Next, we briefly present our experimental infrastructure and the SLIPO toolkit components applied in this setting. Finally, we present our experimental results and discuss our findings.

3.1. Synthetic data

Since TripleGeo can natively access and transform geospatial data from the OpenStreetMap (OSM) database, we used it in order to extract POI data across the entire planet from OSM. This data includes the detailed geometry of POIs (i.e., not just long/lat point locations, but also polygons, linestrings, etc.) and all their tags as a list of key-value pairs. Based on the tags available for each OSM element, we categorized the extracted records according to a two-tier classification scheme[50] with 15 categories and 167 subcategories. OSM elements not qualifying as POIs to any term in this classification scheme were ignored (e.g., road segments). The resulting POI dataset contains 18,369,118 records[51] spread over the entire planet and was stored in a PostGIS database. We specified extra filters with SQL queries over the set of OSM tags in order to isolate some particular attributes concerning POIs that would be used in the data integration task as listed in Table 1. Note that many more attributes can be extracted (e.g., complete addresses, opening hours, payment methods, access to services) from the available OSM tags, but these values are missing from most POIs. Hence, in these tests we focused on a limited number of indicative attributes (Table 1), where the problem of missing values is the less pronounced. This base real dataset occupies about 5.6GB on disk in CSV format and has been used as the *seed* in the generation of the synthetic data.

Basic attributes	Contact-related attributes
location	postal code
OSM identifier	phone number
name	email address
type	webpage

Table 1: POI attributes extracted from OpenStreetMap and used in the experiments

Note that geometric information in the collected POIs concerns not only *Points* (in longitude/latitude coordinates) according to the OGC geometry types, but also *LineStrings*, *MultiLineStrings*, *MultiPolygons*, as well as Geometry Collections. In our benchmark we have only used the *lon/lat coordinates* of their centroids, thus all POI geometries are reduced to point locations.

We enhanced TripleGeo with an extra *generator utility*[52], which was used to create the synthetic datasets. This utility was first applied to *inflate* the seed OSM data and yield a synthetic dataset ***A*** with a substantially increased number of POIs. Further, this generator was also used to *alter* dataset A by modifying locations and slightly changing other attribute values in order to yield a second synthetic dataset ***B***. Our objective was

[50] Introduced in https://github.com/MorbZ/OsmPoisPbf for extracting POIs from OSM into CSV format.

[51] Data exported from OSM in CSV format (monthly updates) available at http://download.slipo.eu/results/osm-to-csv/

[52] https://github.com/SLIPO-EU/TripleGeo/blob/master/src/eu/slipo/athenarc/triplegeo/extra/SyntheticDataGenerator.java

to obtain two POI collections that have a similar number of POIs; however, each POI should not be in the same location in both datasets, nor should it have exactly the same value in its thematic attributes. Thus, POI data integration should be able to find POIs matching between the datasets *A* and *B*.

More specifically, generation of synthetic POI dataset ***A*** involved the following steps:

- *Displacement of locations*. Given parameters *dx*, *dy* that denote the requested displacement of each POI location, respectively along the *x*, *y* axes, the generator performs an affine transformation (*translation*) and relocates every input geometry by a fixed distance in *x*- and *y*-directions. Parameter values *dx*, *dy* are specified in the same units as the locations, i.e., in decimal degrees in this OSM dataset. We specified four such pairs of displacement values in decimal degrees, namely (1,0), (0,1), (0,-1), (-1,0), hence a similar number of replicated datasets were generated after each such translation. Effectively, these displacements create a replica of each point to the West, North, East, and South, always at a distance of 1 decimal degree from the original POI location. This was considered as a safe distance in order to avoid getting potentially matching POIs in nearby locations after the displacement. Note that such displacement may yield invalid geometries (e.g., POIs near the poles may get a latitude less than -90 degrees or greater than 90 degrees), hence these POIs with *out-of-bound* locations should be skipped and not included in the generated data. Those four replicated datasets obtained after applying such displacements were merged into a single dataset together with the seed OSM data. Hence, the number of the POIs in this merged synthetic dataset is practically *quintupled* (**5x**) compared to the original OSM dataset.
- *POI name transliteration*. POI names in the OSM database may be written in different languages, alphabets, phonetic representations, etc. In our case, we extracted the main OSM tag concerning names, but this also results in having names written in different alphabets, as the data contains POIs from all countries and names are mostly written in the native language. To overcome this issue, we performed transliteration of name strings from any language into Latin, making use of a Unicode-compliant transliterator library. This step was important not for the data generation process, but for enabling computation of the string similarity measures involved in data integration.
- *Altered POI names*. To avoid replicating POIs with exactly the same names, our generator randomly modifies them by eliminating a character from the string value. More specifically, specifying a *probability of* 50% for changing the name value of each POI, the algorithm removes a random character from the name value, provided that the original string is at least 5 characters long. Due to this constraint, effectively less than 50% of the POIs would get their names altered in the resulting synthetic dataset.
- *Eliminated thematic attributes*. The generator randomly picks one of the thematic attributes having a NOT NULL value and erases it. From a data integration point of view, this makes it possible to complement those eliminated values with those of a matching POI coming from another data source.

Generation of synthetic POI dataset ***B*** was based on the previously created synthetic dataset ***A***, according to the following steps:

- *Displacement of locations*. The previously modified geometries of dataset A were randomly *translated* by parameters *dx*=0.001, *dy*=0.001 decimal degrees along the *x*, *y* axes. Thus, the

geometry of each POI in the resulting dataset ***B*** is found in small distance from the corresponding POI in dataset ***A***. This small displacement makes it possible to identify such matching POIs in a data integration task. Note that this step may also incur a small number of *out-of-bound* locations, which must be excluded from synthetic dataset ***B***.

- *Altered POI names.* The transliterated names obtained from dataset ***A*** are changed, again by erasing a random character with a probability of 50% from string names having more than 5 characters. This creates POI names in dataset ***B*** having strong similarity with those in dataset ***A***, hence facilitating their matching.
- *Eliminated thematic attributes.* Again, the generator randomly picks one of the thematic attributes having a NOT NULL value and eliminates this value. Since this step is carried out against the contents of dataset ***A***, it is clear that at least one of the original values in each POI attribute survives after such elimination and can thus be restored through data integration.

The following Table 2 provides statistics regarding generation of both synthetic datasets:

Statistics	Dataset A	Dataset B
POIs removed due to out-of-bound geometries	2,745	3,236
POI names altered	43,008,036	42,228,772
Attribute values eliminated	5,045,101	4,807,617
TOTAL number of POIs	91,842,845	91,842,354

Table 2: Statistics regarding the synthetically generated POI datasets in CSV format

3.2. Data integration workflow

The data integration workflow for this indicative scenario can be designed in the SLIPO Workbench, as illustrated in Figure 97, and consists of the following steps:

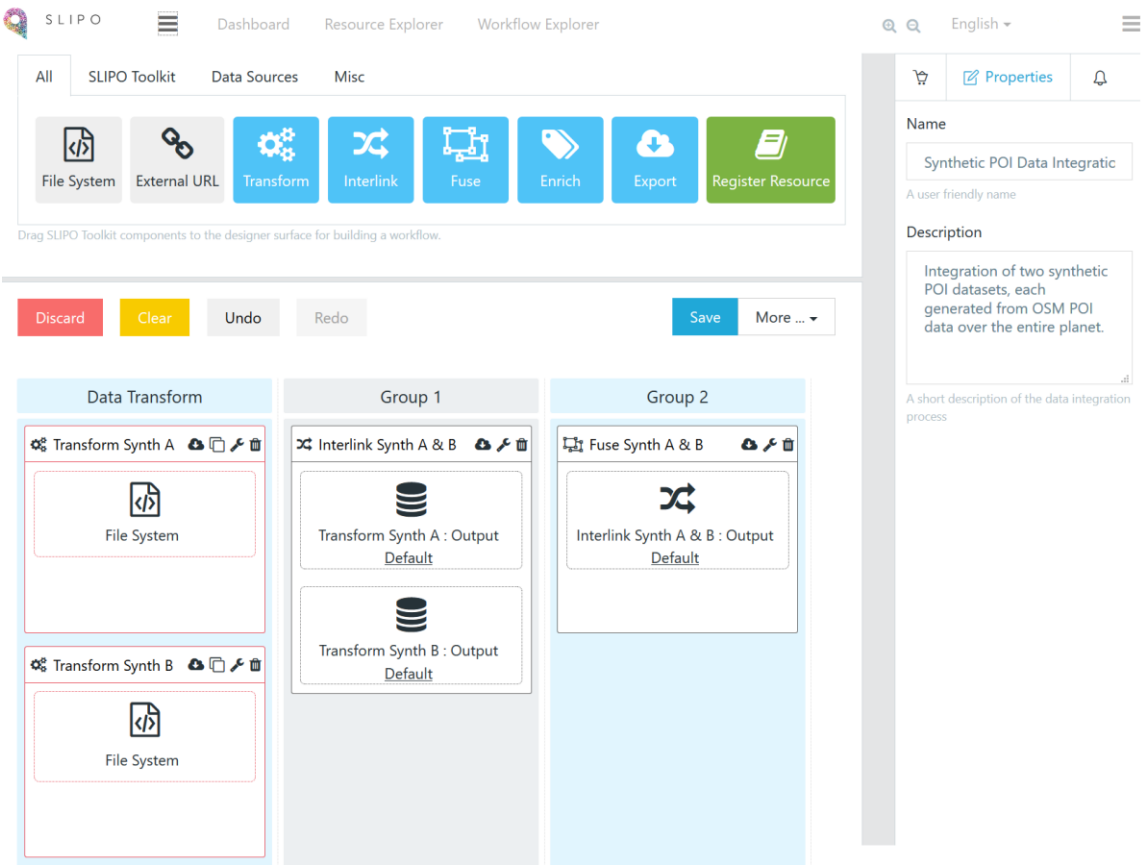


Figure 97: Integration workflow for the synthetic POI data in the SLIPO Workbench

- *Transformation.* Two instances of TripleGeo are employed, respectively for transforming synthetic datasets A and B into RDF. Mappings from the original OSM attributes to the SLIPO ontology were already available from DINUC A.2 and B.1, i.e., the use cases involving OSM data as described in Deliverable D5.2.
- *Interlinking.* An instance of LIMES is invoked to identify matching POIs between the two transformed RDF datasets *A* and *B*. The threshold for accepting links was set to 0.9996, which is quite restrictive in order to avoid receiving too many invalid links. Moreover, this effectively results in establishing links for about 25% of the input POIs, which is comparable to the real-world use cases examined in Deliverable D5.2.
- *Fusion.* An instance of FAGI takes the previously detected links as well as the two transformed RDF datasets and produces a unified output dataset that contains all POIs from both inputs. A predefined fusion configuration is used that prescribes a set of rules for deciding how the values for each property of the linked POIs will be fused. The user selects the proper fusion output mode, in this case *SLIPO_default_ab_mode* to produce an output that includes all possible information from both input datasets in terms of number of POIs (*coverage*) and number of attribute values available per POI (*completeness*).

3.3. Benchmarking infrastructure

Our experiments were performed in the cloud IaaS infrastructure of HELIX hosting the production version of SLIPO as presented in Section 1.3, with the following adaptations:

- *SLIPO Toolkit versions*. The SLIPO Toolkit components handle scaling independently and according to their respective operation parameters and workload characteristics. Further, certain components (*TripleGeo, LIMES*) are available in different flavors with support for different approaches for scaling. Specifically:
    - *TripleGeo.* As detailed in Deliverable D2.4, TripleGeo is available in two flavors, *parallel* and *distributed*. The former scales up via parallel execution in multi-threaded environments, while the latter scales out on top of Apache Spark. Both flavors provide the same features and can be used interchangeably depending on the available underlying computing infrastructures and workload. As reported in Deliverable D2.4, the distributed flavor running on top of Spark is more advantageous over very large datasets, as it can optimally exploit the available system resources (HDFS, CPU cores). Hence, this distributed edition of TripleGeo (v1.7) over Spark is the one tested in these experiments.
    - *LIMES*. LIMES is available in two flavors, *standard* and *distributed*. The standard version scales up within the prescribed execution environment, while the distributed flavor scales out on top of Apache Spark. In this test, we deployed the distributed edition of LIMES (v1.3.0), as this is the most suitable for coping with the two very large input RDF datasets (~1.5 billion triples each).
    - *FAGI*. FAGI is available in a single flavor, inherently scaling out by partitioning the available data as well as the links between them and executing in parallel over multiple nodes. FAGI v3.0 automatically selects the optimal number of nodes/partitions according to the required workload. At the end of the fusion process, the output of all FAGI instances are combined to a single or multiple output results (depending on the fusion mode).
    - *DEER*. DEER v2.0.1 is available in a single flavor, scaling up within the prescribed execution environment. The runtime execution of DEER is dominated by the I/O and constraints imposed by the querying of a remote SPARQL endpoint (i.e., DBPedia in our setting). Overall, 90% of DEER's runtime for moderate data assets (i.e., tens of thousands of triples) concerns the *response* time of DBpedia, which further increases for larger data assets. As such, we have excluded DEER from the benchmark.
- *Cloud* resources. We have allocated (*over-reserved*) additional resources from HELIX to ensure that (a) all installed versions of the deployed SLIPO Toolkit components can scale as needed to accommodate the data integration workloads, and (b) the operation of the production version of SLIPO remains unaffected. We have reserved roughly *twice* the size of resources compared to those presented in Section 1.3, and released them following the end of our experimental evaluation. In particular, tests were conducted on top of a YARN/Hadoop cluster consisting of 8 processing nodes using an HDFS cluster for storage (with a disk capacity of 1.2TB); by default, the HDFS block size is 64MBs. Each node has 7 vcores and 15GB RAM for running YARN tasks. All nodes in the cluster are

connected on a rack-local network with a Gigabit Ethernet switch and run Linux Ubuntu 16.04 LTS with Hadoop 2.9 and Spark 2.2.3.

- *Synthetic data generation*. We have allocated a single VM (16 vcores, 32GB RAM, 800GB free disk storage) to produce the synthetic data and make them available to the SLIPO Workbench via HDFS.
- *Workflow management*. The data integration workflow was instantiated, executed, managed, and monitored by SLIPO's workload management system in *debug* mode, allowing us to inspect the output results, statistics & indicator component logs, system run-time logs independently for each Toolkit component involved in the data integration workflow. This operation mode does not divert from the standard workflow management of the SLIPO system, in which each component runs in sequence (*i.e.., transformation/TripleGeo, followed by linking/LIMES, and fusion/FAGI*) and does not affect the runtime results.

3.4. Experimental Results

Using the aforementioned benchmarking infrastructure, we executed the POI data integration workflow specified in Section 3.2. We collected only performance measures, concerning the execution cost of each stage (and execution time of sub-stages, if supported by each component), as well as the number of features (either POI records or RDF triples) involved in each one.

Regarding the transformation of each synthetic POI dataset to RDF with TripleGeo, Table 3 shows that there is some significant partitioning overhead since it takes some time to split and distribute to the Spark workers such large volumes of input datasets. But afterwards, each transformation task can run in parallel for each partition at their assigned worker node. Each data partition has the same size in number of input POIs, although the number of attributes per POI may vary, hence each worker takes a slightly different amount of time to conclude its assigned transformation task. In Table 3, we report as transformation time the maximum duration of transformation tasks observed in all worker nodes. Thanks to this parallelization, the entire transformation of each synthetic dataset is carried out in **about 10 minutes**, which is quite remarkable considering the world-scale size of the input POI datasets and underscores the scalability, robustness, and efficiency of TripleGeo.

Dataset (records)	Partitioning time	Transformation time*	Total time
A: 91,842,845 POIs	216.938 sec	396.548 sec	613.486 sec
B: 91,842,354 POIs	211.710 sec	420.453 sec	632.163 sec

Table 3: Statistics for the transformation steps with TripleGeo

Subsequently, the RDF triples of the transformed POIs from either dataset were given to LIMES in order to identify links between them. Statistics from this stage are listed in Table 4, and indicate that LIMES managed to discover links in little more than **90 minutes**, with each worker running in parallel against disjoint subsets of the input RDF graphs. It should be noted that only geometries were considered in establishing matching POIs, whereas the similarity threshold was deliberately restrictive in order to minimize the invalid links between POIs that refer to different entities.

RDF triples A	RDF triples B	Linking time*
1,537,766,754	**1,533,742,530**	5880 sec

Table 4: Statistics for the linking step using LIMES

The links discovered with LIMES, along with the two RDF datasets transformed by TripleGeo were next given to FAGI for fusion. Table 5 lists the measured execution time of each subtask of the fusion process, as well as the total time for fusing these two RDF graphs into a single RDF dataset. Not surprisingly, most of the time is spent in splitting the initial RDF datasets into separate *partitions* to be parallelly processed by multiple FAGI instances. But afterwards, *loading* pairs of corresponding partitions (and the links involving them) into RDF graphs for processing is done efficiently by each FAGI instance, as it runs in parallel and involves reduced data volumes. However, there is some overhead from data transfer between worker nodes, as they may need to exchange RDF triples and links in order to establish the fusion results. However, the actual cost for applying the *fusion actions* on corresponding POI features and their attributes is less than half an hour (1678.9 sec). This subtask is carried out in parallel for each partition and performs in a very robust and efficient fashion, although it involves parsing of rule specifications, transformation of the data to the proper internal formats, and also writing the produced fused RDF triples into the output files. In total, the fusion stage (involving all previously mentioned subtasks) takes about **4.1 hours** to produce a single, unified RDF dataset that contains all available information from the input data (1,596,445,628 triples).

Links	Partitioning time**	Loading time*	Data transfer time	Fusion time*	Total time
20,196,049	12569.2 sec	112.4 sec	410.6 sec	1678.9 sec	14771.1 sec

Table 5: Statistics for the fusion step using FAGI v3.0

* *Max value among all worker nodes running in parallel, which represents the slowest processing node in this stage.*

** *Partitioning time also include the runtime for the inverse process, i.e. merging of the individual, fused files.*

* *Max value among all worker nodes running in parallel, which represents the slowest processing node in this stage.*

4. Annex

4.1. Database Schema

In SLIPO, the PostgreSQL relational database is used for storing application data, like user accounts, web session data, data integration workflows, etc. SLIPO stores relational data tables under four schemas, namely:

- **public**: Consists of tables for storing user accounts, resource metadata and workflow definitions used by SLIPO Web Application and Service.
- **rpc**: A set of tables for storing Spring Batch execution data used by the SLIPO Service.
- **web**: Contains tables for storing web session data for the SLIPO Web Application.
- **spatial**: This schema contains tables with spatial data imported to PostGIS using TripleGEO reverse transformation of RDF data.
- **fagi**: This schema contains tables for storing fusion logs generated by FAGI.

In the next sections, we describe the most important tables for the public schema. SQL Data Definition Language (DDL) scripts for all tables are available at the SLIPO repository[53]. Tables that are generated automatically by 3rd party libraries (i.e., Flyway schema migration tool, Spring Batch and Spring Session), are not included. Additional information about these tables can be found at Flyway, Spring Batch and Spring Session GitHub repositories[54,55,56], respectively.

For the spatial schema, there are no specific table definition scripts since the tables are created dynamically during the import of RDF data. The only convention used is that all tables contain spatial information in a single field named **the_geom** using **EPSG:4326**[57] spatial reference system.

4.1.1. account

Stores user account data.

Field	Name
id	Unique Id
username	User name (by convention the user name is the user email address)

[53] https://github.com/SLIPO-EU/workbench/tree/master/command/src/main/resources/db/migration

[54] https://github.com/flyway/flyway/tree/master/flyway-core/src/main/resources/org/flywaydb/core/internal/database

[55] https://github.com/spring-projects/spring-batch/tree/master/spring-batch-core/src/main/resources/org/springframework/batch/core

[56] https://github.com/spring-projects/spring-session/tree/master/spring-session-jdbc/src/main/resources/org/springframework/session/jdbc

[57] https://epsg.io/4326

active	-
blocked	-
email	-
family_name	-
given_name	-
lang	-
password	Password encoded using the Spring BCryptPasswordEncoder[58]
registered_at	-

4.1.2. account_role

Stores roles assigned to users. Currently the application supports two roles, namely, User and Administrator.

Field	Name
id	Unique Id
role	Role name as declared by the enumeration EnumRole[59]
account	Reference to account who is assigned the role
granted_at	-
granted_by	Reference to account who granted the role

4.1.3. application_key

Stores application keys used for authenticating SLIPO API requests.

Field	Name
id	Unique Id
application_name	Unique application name
application_key	-
created_on	-
created_by	-
revoked_on	-
revoked_by	-

[58] https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/crypto/bcrypt/BCryptPasswordEncoder.html

[59] https://github.com/SLIPO-EU/workbench/blob/master/common/src/main/java/eu/slipo/workbench/common/model/EnumRole.java

mapped_account	The account to which the application key is mapped to. A SLIPO API request is executed in the security context of the mapped user.
max_daily_request_limit	Maximum number of daily requests.
max_concurrent_request_limit	Maximum number of concurrent requests.

4.1.4. log4j_message

Stores application log messages.

Field	Name
id	Unique Id
application	Application module that generated the log message e.g. SLIPO Web Application or SLIPO Service.
generated	-
level	-
message	-
throwable	-
logger	-
client_address	Client remote IP address if available
username	Authenticated user name. Applicable only to SLIPO Web Application.

4.1.5. process

Table used for storing data integration workflow definitions. See Annex 4.2 for details on process definitions.

Field	Name
id	Unique Id
version	-
row_version	Used by Hibernate ORM for optimistic locking
name	User friendly name for the data integration workflow
description	-
created_by	-
updated_by	-

created_on	-
updated_on	-
definition	A process definition serialized in JSON format.
is_template	A template is a recipe for a process whose configuration is only partially saved
task_type	Task type as defined in EnumProcessTaskType[60]

4.1.6. process_revision

Stores different version for a process definition. Whenever a user updates a process, a new version is created.

Field	Name
id	Unique Id
parent	Id of the initial process
version	Current version for this process definition
name	-
description	-
updated_by	-
updated_on	-
definition	-

4.1.7. process_execution

Stores data for the execution of a specific version of a workflow definition.

Field	Name
id	Unique Id
process	Reference to a record in process_revision table
submitted_by	Reference to account table or null if the execution has been initialized by the SLIPO Service process
submitted_on	-
started_on	-
completed_on	-

[60] https://github.com/SLIPO-EU/workbench/blob/master/common/src/main/java/eu/slipo/workbench/common/model/process/EnumProcessTaskType.java

status	The overall status; one of "RUNNING", "STOPPED", "FAILED", "COMPLETED"
error_message	Error that has caused the execution to fail

4.1.8. process_execution_step

Stores data for a single logical step in a workflow execution e.g. LIMES execution. A logical step in a workflow is implemented as a Spring Batch flow which in turn may consist of one or more Spring Batch steps. A record in this table aggregates the data for the whole flow.

Field	Name
id	Unique Id
job_execution	The unique id of the underlying job execution
process_execution	Reference to the owner process execution
step_key	Unique step key as provided by the process designer and stored in process definition
step_name	-
tool_name	Name of the SLIPO Toolkit Component e.g. FAGI
operation	Name of the operation being executed e.g. FUSION
started_on	-
completed_on	-
status	The status of underlying job execution, e.g. RUNNING, STOPPED, FAILED, COMPLETED etc
error_message	Error that has caused this step to fail
node_name	The workflow-friendly name of a step

4.1.9. process_execution_step_file

This table stores metadata for all files used or created by a single process execution step including configuration files, input files, output files etc.

Field	Name
id	Unique Id
process_execution_step	Reference to the owner process execution step

type	File type as declared in enumeration EnumStepFile[61]
file_path	File path relative to workflow data directory
file_size	File size in bytes
resource	Resource id, if the file refers to a registered resource (e.g. an input file)
data_format	-
bbox	-
table_name	Table name in spatial schema if RDF data has been imported to PostGIS
verified	A flag to indicate if an expected output file is verified
primary_output	A flag that marks an output file as the primary output of the processing step it belongs
output_part	The key of the output part this file corresponds to
layer_style	Layer style if RDF data has been imported to PostGIS
row_count	Total rows imported to PostGIS

4.1.10. resource

In general, a workflow can use any file that can be transformed using TripleGEO as input. In the case that a specific file is used too often in workflow definitions, a user can register this file to be transformed once and used multiple times. Metadata for registered files are stored in resource table.

Field	Name
id	Unique Id
version	-
row_version	Used by Hibernate ORM for optimistic locking
type	Data type as declared in enumeration EnumResourceType[62]
source_type	Data source type as declared in EnumDataSourceType[63]
input_format	Initial data format as declared in EnumDataFormat[64]
format	RDF data format. Always equal to N-TRIPLES

[61] https://github.com/SLIPO-EU/workbench/blob/master/common/src/main/java/eu/slipo/workbench/common/model/process/EnumStepFile.java
[62] https://github.com/SLIPO-EU/workbench/blob/master/common/src/main/java/eu/slipo/workbench/common/model/poi/EnumResourceType.java
[63] https://github.com/SLIPO-EU/workbench/blob/master/common/src/main/java/eu/slipo/workbench/common/model/resource/EnumDataSourceType.java
[64] https://github.com/SLIPO-EU/workbench/blob/master/common/src/main/java/eu/slipo/workbench/common/model/poi/EnumDataFormat.java

process_execution	Reference to process execution that created this catalog entry
name	-
description	-
created_on	-
created_by	-
updated_on	-
updated_by	-
bbox	-
number_of_entities	-
file_path	-
file_size	-
table_name	Table name in spatial schema if RDF data has been imported to PostGIS

4.1.11. resource_revision

Stores different versions for a record in resource table.

Field	Name
id	Unique Id
parent	Reference to parent resource
version	-
type	-
source_type	-
input_format	-
format	-
process_execution	Reference to process execution that created this catalog entry
name	-
description	-
updated_on	-
updated_by	-
bbox	-

number_of_entities	-
file_path	-
file_size	-
table_name	Table name in spatial schema if RDF data has been imported to PostGIS
layer_style	Layer style if RDF data has been imported to PostGIS
row_count	Total rows imported to PostGIS

4.2. Process Definition

In SLIPO, data integration workflows are modeled using instances of the ProcessDefinition[65] class. A process definition contains all the metadata required by the workflow execution engine to compose a graph of Spring Batch flows implementing the described workflow.

Process definitions can be easily serialized in JSON format which facilitates the unobstructed communication between the Workbench, SLIPO Web Application and SLIPO Service modules and simplifies workflow persistence.

A process definition consists of three building blocks:

- **Metadata**: Contains a unique name and a description of the workflow
- **Resources**: Consists of one or more input resources. An intermediate step output can also be used as an input. A resource is modeled using a concrete implementation of the abstract ProcessInput[66] class.
- **Steps**: A list of one or more SLIPO Toolkit component operations or resource registration ones. All operations are modeled using the Step[67] class. Each step has a component specific configuration object assigned to it. Configuration object classes implement the ToolConfiguration[68] interface.

A simple process definition including two steps, namely, a transformation operation using TripleGeo and a resource registration operation, is depicted in Figure 98. The configuration of the TripleGeo operation has been truncated for brevity.

[65] https://github.com/SLIPO-EU/workbench/blob/master/common/src/main/java/eu/slipo/workbench/common/model/process/ProcessDefinition.java
[66] https://github.com/SLIPO-EU/workbench/blob/master/common/src/main/java/eu/slipo/workbench/common/model/process/ProcessInput.java
[67] https://github.com/SLIPO-EU/workbench/blob/master/common/src/main/java/eu/slipo/workbench/common/model/process/Step.java
[68] https://github.com/SLIPO-EU/workbench/blob/master/common/src/main/java/eu/slipo/workbench/common/model/tool/ToolConfiguration.java

```
{
  "name": "Test TripleGeo",
  "description": "Test TripleGeo",
  "resources": [
    {
      "key": 1,
      "inputType": "OUTPUT",
      "resourceType": "POI_DATA",
      "name": "TripleGeo : Output",
      "tool": "TRIPLEGEO",
      "stepKey": 0
    }
  ],
  "steps": [
    {
      "key": 0,
      "group": 0,
      "name": "TripleGeo",
      "nodeName": "triplegeo",
      "operation": "TRANSFORM",
      "tool": "TRIPLEGEO",
      "inputKeys": [ ],
      "sources": [
        {
          "type": "FILESYSTEM",
          "path": "csv/data.csv"
        }
      ],
      "outputKey": 1,
      "outputFormat": "N_TRIPLES",
      "configuration": {
        "inputFormat": "CSV",
        "mode": "STREAM",
        "targetGeoOntology": "GeoSPARQL",
        "attrKey": "ID",
        "attrName": "NAME",
        "attrCategory": "CATEGORY",

        // ...
      }
    },
    {
      "key": 1,
      "group": 2,
      "name": "Register Resource",
      "nodeName": "register-resource",
      "operation": "REGISTER",
      "tool": "REGISTER",
      "inputKeys": [
        1
      ],
      "sources": [

      ],
      "configuration": {
        "metadata": {
          "name": "Register Resource",
          "description": "Register Resource"
        },
      }
    }
  ]
}
```

Figure 98: Process definition example

4.3. Software versions

In this section, we enumerate the current versions for all major software components used for the SLIPO system.

Ubuntu:

- **Distributor ID**: Ubuntu
- **Description**: Ubuntu 16.04.4 LTS
- **Release**: 16.04
- **Codename**: xenial

PostgreSQL:

- PostgreSQL 9.5.12 on x86_64-pc-linux-gnu, compiled by gcc (Ubuntu 5.4.0-6ubuntu1~16.04.9) 5.4.0 20160609, 64-bit.

PostGIS:

- POSTGIS="2.2.1 r14555" GEOS="3.5.0-CAPI-1.9.0 r4084" PROJ="Rel. 4.9.2, 08 September 2015" GDAL="GDAL 1.11.3, released 2015/09/16" LIBXML="2.9.3" LIBJSON="0.11.99" TOPOLOGY RASTER

Java:

- OpenJDK Runtime Environment (build 1.8.0_171-8u171-b11-0ubuntu0.16.04.1-b11)
- OpenJDK 64-Bit Server VM (build 25.171-b11, mixed mode)

Maven:

- Apache Maven 3.3.9
- Java version: 1.8.0_171, vendor: Oracle Corporation

Node.js:

- node version: 8.11.1
- npm version: 5.6.0

Tomcat:

- Server version: Apache Tomcat/8.5.31
- Server built: Apr 27, 2018 20:24:25 UTC
- Server number: 8.5.31.0
- OS Name: Linux

- OS Version: 4.13.0-43-generic
- Architecture: amd64
- JVM Version: 1.8.0_171-8u171-b11-0ubuntu0.16.04.1-b11
- JVM Vendor: Oracle Corporation

Docker Server Engine:

- Version: 18.03.1-ce
- API version: 1.37 (minimum version 1.12)
- Go version: go1.9.5
- Git commit: 9ee9f40
- Built: Thu Apr 26 07:15:30 2018
- OS/Arch: linux/amd64
- Experimental: false

TripleGEO

- Version: 2.0
- Docker Recipe: https://github.com/SLIPO-EU/docker-recipes/tree/master/triplegeo
- Docker Image: https://hub.docker.com/r/athenarc/triplegeo/tags

FAGI

- Version: 3.0
- Docker Recipe: https://github.com/SLIPO-EU/docker-recipes/tree/master/fagi
- Docker Image: https://hub.docker.com/r/athenarc/fagi/tags

LIMES

- Version: 1.7
- Docker Recipe: https://github.com/SLIPO-EU/docker-recipes/tree/master/limes
- Docker Image: https://hub.docker.com/r/athenarc/limes/tags

DEER

- Version: 2.0
- Docker Recipe: https://github.com/SLIPO-EU/docker-recipes/tree/master/deer
- Docker Image: https://hub.docker.com/r/athenarc/deer/tags