DELIVERABLE D1.2

Architecture

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 8, 31/08/2017
Actual Submission Date	09/08/2017
Work Package	WP1 - Requirements, Architecture and Integration
Tasks	1.2 Architecture
Type	Report
Approval Status	Submitted for approval
Version	1.0
Number of Pages	59
Filename	D1.2_Architecture.pdf

Abstract

This report presents the architecture and design for the SLIPO system and its software components. First, an overview of the system architecture is presented and its main modules are identified. Next, the description, dependencies and development roadmap for each component of the SLIPO Toolkit are provided. Finally, the external third-party tools and libraries that will be used during the development of the SLIPO system are enumerated.

History

version	date	reason	revised by
0.1	25/06/2017	First draft	Yannis Kouvaras
0.2	27/06/2017	Added SLIPO Toolkit Components section	Nikos Karagiannakis, Yannis Kouvaras
0.3	04/07/2017	Added SLIPO Toolkit Components content	Mohamed Sherif, Axel Ngonga
0.4	06/07/2017	Added content to various Sections	Yannis Kouvaras
0.5	20/7/2017	Added content to various Sections	Yannis Kouvaras
0.6	07/08/2017	Added SLIPO Toolkit Components content	Claus Stadler, Jens Lehmann, Alexander Kister, Giorgos Giannopoulos
0.7	08/08/2017	Multiple revisions and improvements	Yannis Kouvaras, Giorgos Giannopoulos
1.0	09/08/2017	Final version	Spiros Athanasiou

Author list

organization	name	contact information
ATHENA RC	Spiros Athanasiou	spathan@imis.athena-innovation.gr
ATHENA RC	Giorgos Giannopoulos	giann@imis.athena-innovation.gr
ATHENA RC	Yannis Kouvaras	jkouvar@imis.athena-innovation.gr
ATHENA RC	Nikos Karagiannakis	nkaragiannakis@imis.athena-innovation.gr
InfAI	Axel Ngonga	ngonga@informatik.uni-leipzig.de
InfAI	Mohamed Sherif	sherif@informatik.uni-leipzig.de
InfAI	Claus Stadler	cstadler@informatik.uni-leipzig.de
IAIS	Jens Lehmann	lehmann@informatik.uni-leipzig.de
IAIS	Alexander Kister	Alexander.Kister@iais.fraunhofer.de

Executive Summary

This document presents the architecture and an initial design of the SLIPO platform and its individual components. SLIPO develops Linked Data technologies for the scalable and quality assured integration of Big POI Data assets; software, models and processes for transforming conventional POI data into RDF data; interlinking POI entities from different datasets; enriching POI entities with additional metadata; fusing Linked POI Data; assessing the quality of the integrated POI data; offering value added services based on spatial aggregation, association extraction and spatiotemporal prediction.

During the SLIPO architecture presentation, we will explore solutions to the challenges enacted by the user requirements recorded in deliverable D1.1 "Use Cases and Requirements". Since requirement elicitation is not a static one-time process, and requirements may change through the course of the project, the design of the SLIPO system may be updated accordingly to ensure it addresses the needs of a commercial POI integration software.

The layout of document is the following.

In Section 1, we present the SLIPO platform architecture and suggest solutions to the challenges enacted by the user requirements. Initially, we present an overview of the SLIPO logical architecture, identify the main SLIPO software modules, and establish the dependencies and data flow paths between them. Next, we provide a more detailed view of each module and the subsystems that comprise them. Finally, we describe the software integration and testing practices that will be applied during software development, as well as the deployment process of the SLIPO system.

In Section 2, we introduce the individual software components that comprise the SLIPO Toolkit and will be extended by the project to handle the scalable integration of Linked POI Data. For every component, we provide a short description, its internal architecture, its dependencies to external libraries, its role in the SLIPO data integration lifecycle, and the main contributions of the project to it. Moreover, a development roadmap of the features that will be developed in the course of the project is laid out.

Finally, in Section 3, we present the third-party libraries, frameworks and tools that will be used for implementing the software components of the SLIPO system, and we introduce the software systems and applications that will be deployed and combined in the SLIPO architecture. First, we provide a description of the systems and processing frameworks for storing and analyzing POI data. Next, we present the frameworks and libraries required for server-side development. Finally, we enumerate the libraries and frameworks which will be used in the development of SLIPO's front-end.

Abbreviations and Acronyms

AJAX	Asynchronous JavaScript and XML
AOI	Area of Interest
API	Application Programming Interface
CI	Continuous integration
CRS	Coordinate Reference System
CSRF	Cross-Site Request Forgery
CSS	Cascading Style Sheets
CSW	Catalog Service for the Web
DOM	Document Object Model
ETL	Extract Transform Load
GIS	Geographic Information Systems
GML	Geography Markup Language
HDFS	Hadoop Distributed File System
JSON	JavaScript Object Notation
KML	Keyhole Markup Language
MVC	Model View Controller
NFS	Network File System
OGC	Open Geospatial Consortium
OWL	Web Ontology Language
POI	Point of Interest
REST	Representational State Transfer
RDF	Resource Description Framework
SFS	Simple Feature Specification
SPA	Single Page Application
SSH	Secure Shell
UI	User Interface
VM	Virtual Machine

Table of Contents

1. Architecture

In this section, we present the SLIPO platform architecture and explore solutions to the challenges enacted by the user requirements. First, we present an overview of the SLIPO logical architecture, identify the main SLIPO software modules, and establish the dependencies and data flow paths between them. Next, we provide a more detailed view of each module and the subsystems that comprise them. Finally, we describe the software integration and testing practices that will be applied during software development, as well as the deployment process of the SLIPO system.

1.1. System Logical Architecture

In this section, we introduce the main parts of the SLIPO system, present a high-level overview of its logical architecture and identify the main data flow paths.

The SLIPO system consists of the following main modules as shown in Figure 1.

- **SLIPO Toolkit**: a collection of individual software components for quality-assured POI data integration, including transformation, interlinking, fusion, enrichment and analytics.
- **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.
- **SLIPO APIs**: a collection of RESTful HTTP programmatic interfaces for invoking SLIPO system functionality and integrating it into third-party systems.

In the following, the SLIPO Toolkit components are enumerated. A detailed description is provided in Section 2.

- **TripleGEO** is generic purpose, open-source software that is used for integrating spatial features from several sources into RDF triples. It provides support for several file formats including GML, KML 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.
- **Sparqlify** is a scalable SPARQL-SQL rewriter that enables the definition of RDF views on relational databases for querying data using SPARQL. The views are created using a novel syntax, namely Sparqlification Mapping Language (SML). The result of the query rewriter is a single SQL command that can be further optimized by the underlying database system.
- **LIMES** is a Link Discovery (interlinking) framework for the Web of Data. It implements time-efficient approaches for large-scale Link Discovery based on the characteristics of metric spaces. LIMES can be easily configured either via a configuration file or through a graphical user interface. It can also be executed as an HTTP service.

- **OSMRec** is a framework for the semantic enrichment and classification of geospatial entities. It provides the ability for assigning already existing categories/tags on newly created spatial entities. OSMRec is available as a standalone Java library, and as a plugin for JSOM[1].
- **DEER** is a data enrichment framework that applies enrichment functions and operators to discover implicit or explicit references of entities to external datasets. It relies on a simple pipeline system that consists of two main components: modules and operators. Modules implement functionality for processing the content of a single dataset while operators work at a higher level of granularity and combine datasets.
- **FAGI** is a platform that allows fusion of geospatial Linked Data, supporting several thematic and spatial fusion actions. The supported fusion actions are individually applied on each property of a pair of Linked geospatial entities and range from plain concatenation of values, to manipulation of the geometries of the respective entities. It can be invoked as a Java library and it receives two source datasets and a list of Linked entities between them, either in file format or through an available SPARQL endpoint.
- **SANSA** is a distributed *in-memory* framework for RDF that provides scalable inference and analytics capabilities for Linked Data. It can run on top of either the Apache Spark or Apache Flink distributed processing frameworks. Although SANSA is not part of the core SLIPO data integration lifecycle (Transformation, Interlinking, Enrichment, Fusion), it will be utilized for performing scalable analytics on integrated, RDF POI datasets.

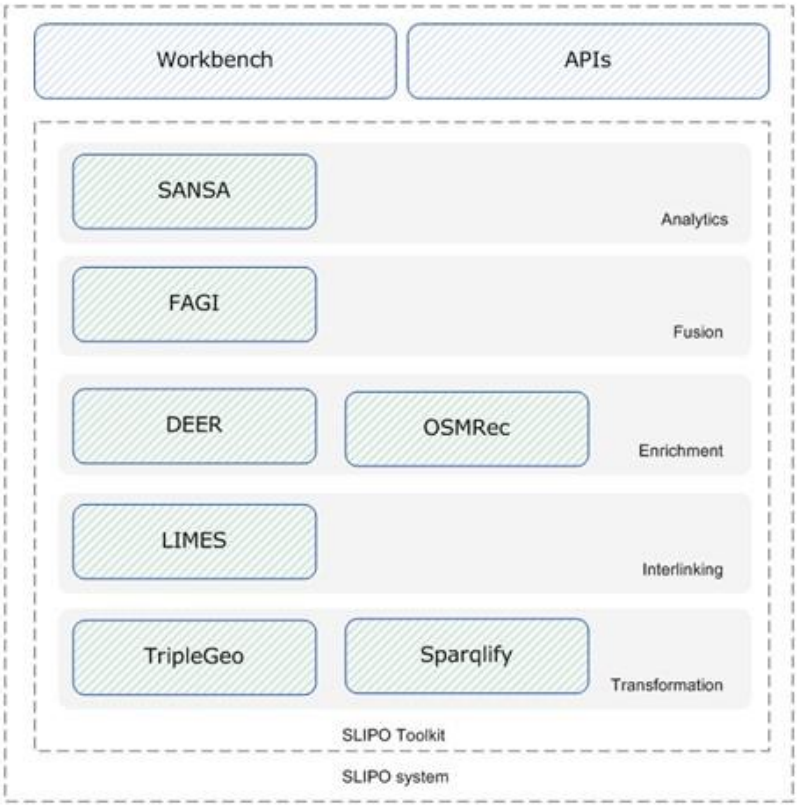

Figure 1: The SLIPO system

[1] https://josm.openstreetmap.de/

The SLIPO system implements a *workflow engine* that executes data integration workflows and a *scheduler* for initializing workflow execution. A workflow consists of several loosely coupled tasks. Each task invokes an operation implemented by a component in the SLIPO Toolkit. The *workflow engine* and the SLIPO Toolkit components are deployed over a cloud infrastructure. Workbench and APIs exchange messages with the *scheduler* to execute workflows. The *scheduler* propagates requests to the *workflow engine* which subsequently initiates the execution locally, or on a remote server. The latter has no control over task execution optimization, resource management and data partitioning. These issues are deferred to the individual SLIPO Toolkit component invoked by a task. Every SLIPO Toolkit component is responsible for providing a *scalable* implementation for the requested operation.

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 layer lower in the stack unless stated otherwise. Next, we provide a brief description of each module starting from the top layer of the stack and moving downwards.

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** and (c) scheduling and monitoring the execution of the integration workflows.
- **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. Operations that are computationally expensive are only exposed by the **Workbench** application.

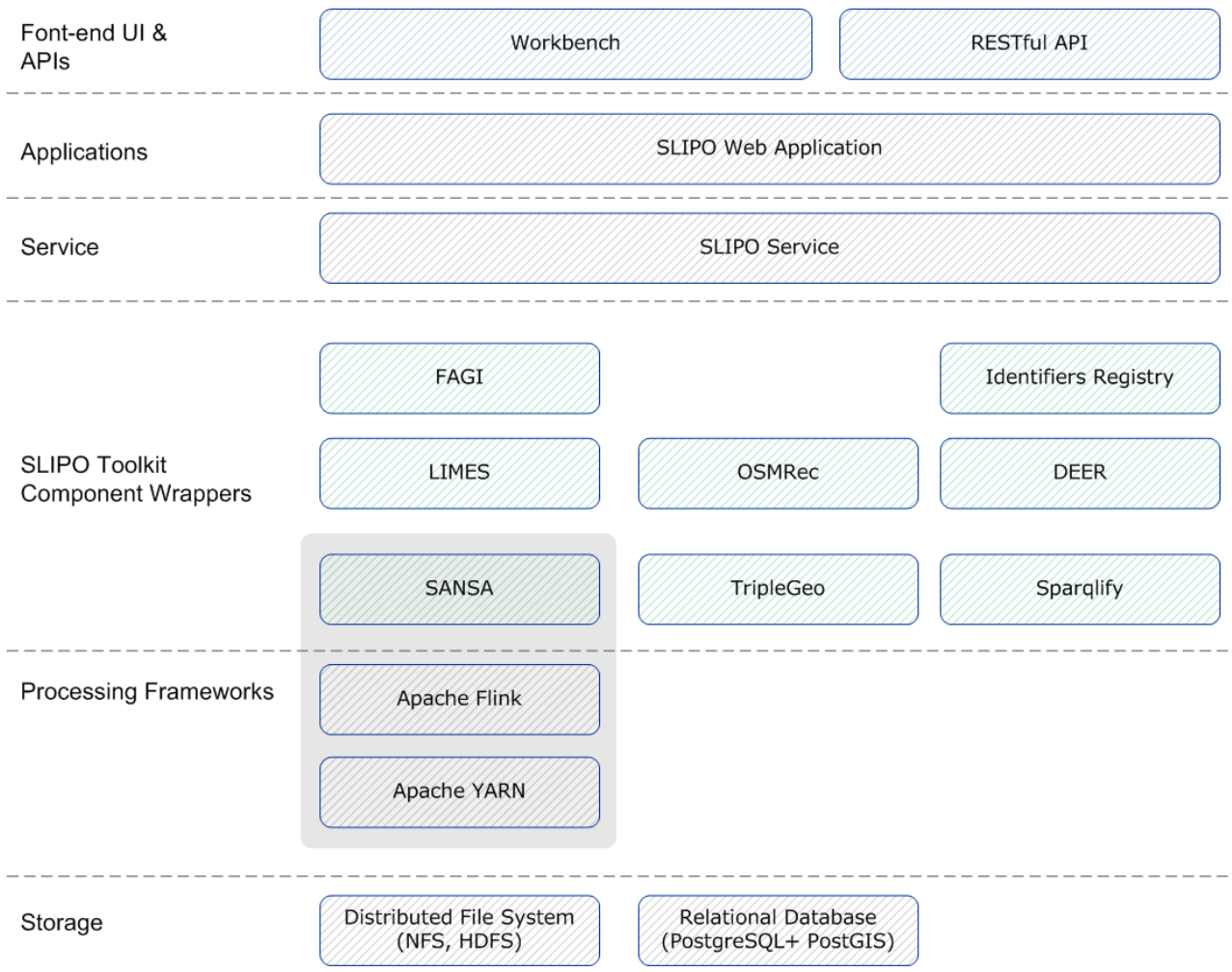

Figure 2: SLIPO system Architecture

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** web 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 quality assurance metrics for the whole workflow.

Next are the **SLIPO Toolkit Component Wrappers** and the **Identifiers Registry**.

- 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 **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.

The **Processing Frameworks** layer contains external data management and processing frameworks used by the SLIPO system.

- **Apache Flink** is a data processing system for analyzing large datasets. Flink implements its own job execution runtime and can be used as a standalone processing system. However, when used with Apache Hadoop, Flink can access data stored in HDFS and request cluster resources from the YARN resource manager. Flink extends the MapReduce programming model with additional operators, also called transformations. Moreover, the data model used by Flink operators is record based instead of the key-value pair model used by MapReduce. All operators will start working in memory and gracefully fallback to external memory algorithms once memory resources become low. The new operators represent many common data analysis tasks more naturally and efficiently.
- **YARN** is a resource manager that is agnostic to the applications that request resources, thus, allowing the implementation of arbitrary computational models, including Flink, such as graph processing, machine learning, etc. YARN uses a single ResourceManager for managing collectively the resources of a cluster and one NodeManager per server for managing each server's local resources. The resources on each server are organized in leases (containers) that represent CPU, memory, disk space, network bandwidth etc.

The diagram contains one final layer with all the storage types utilized by the system. The SLIPO Toolkit comprises several tools, with each one supporting several input sources and data formats. Since our goal is to *enhance* existing software with POI processing capabilities *without affecting* the way they are invoked, we decided to use a distributed file system as the common denominator of data storage. Moreover, a relational database will be used for storing and indexing small POI datasets (e.g., sample data generated by SLIPO Toolkit components). Therefore, the **SLIPO Service** will abstract the configuration and invocation of every **SLIPO Toolkit** component, and all components will have access to all data through a shared distributed file system. Security and/or performance constraints regarding data access at the component-level will be implemented inside the SLIPO Service. All data will be physically located inside a private cloud infrastructure (IaaS) and will not be publicly available to external users.

- **Distributed File System**. We can use either NFS or HDFS as the distributed file system, with our final decision reached after examining both options. NFS is a distributed file system service that can be used to share resources from one server (or optionally more, using additional configuration) with other systems across the network. If additional resilience and redundancy is required, HDFS can be used as the shared file system. HDFS provides replication but it also requires additional configuration to be accessible using the NFS protocol, since most of the tools do not support it out of the box.
- **PostGIS** is an extension to the **PostgreSQL** object-relational database system which allows GIS objects to be stored in the database. PostGIS includes support for GiST-based R-Tree spatial

indexes, and functions for analysis and processing of GIS objects. Using **PostGIS** and **Sparqlify** we can also expose POI datasets as SPARQL endpoints.

1.2. Components

In this section, we present a more detailed view of all SLIPO modules. Each module is collapsed to its basic components and a description is given for their functionality, their relationships and dependencies with other components, their external interfaces, the associated design decisions, and their rationale. In the followings, we often make references to several frameworks and libraries we will be using during the development process. Detailed information for these software components is provided in Section 3.

1.2.1. Workbench

To offer increased UI usability that is in par with desktop applications, the Workbench web application will be developed as a Single Page Application (SPA). In contrast to common web applications, a SPA application is initialized by loading a single web page and further needed resources, content and scripts are loaded dynamically on demand using AJAX requests. A diagram with the most important components of the Workbench application is shown in Figure 3.

At the top of the diagram are the user interface components. Instead of implementing all the UI components manually, we are going to use the **React** library. 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.

At the next level, there are the components that implement the core of the application logic. React components manage the rendering of the application state to views for the user to interact with. However, the state management must still be handled by the application. Instead of adopting the commonly used MVC (Model View Controller) design, we opt to use a different pattern, namely, **Redux**. Redux strictly dictates how the state is updated and attempts to make state mutation predictable by managing state mutation and asynchronicity. In contrast to the MVC approach, Redux removes two-way interactions between the controller and the application state (or model in MVC terms). 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. More details on React and Redux can be found in Sections 3.4.1 and 3.4.2 respectively.

Another important aspect of a SPA application is handling user navigation. Instead of loading a new page by the browser whenever a navigation request is submitted, a SPA application intercepts the request and renders the appropriate view. The **Router** component is responsible for handling navigation in the Workbench application and integrates with the Redux state management.

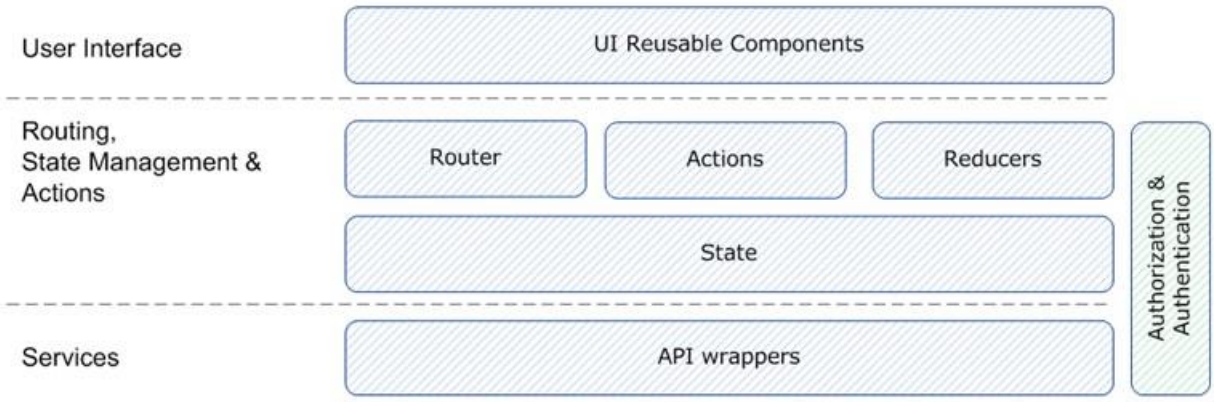

Figure 3: Workbench components

At the last level, there are the API wrappers. API wrappers are classes that expose API endpoints as a collection of simple functions that can be easily invoked by Redux actions.

All API endpoint requests can be categorized into a few distinct types, namely, multipart form POST submission and POST, PUT, GET and DELETE requests with JSON payloads. Thus, all API wrappers can be derived from a single class that makes the implementation of more advanced concepts, such as function call interception, simpler. For instance, when using OAuth 2.0, we can intercept calls that failed with status code 403, request a new access token transparently, and then replay the initial request. Moreover, isolating API calls in a separate layer allows handling security concerns, error handling and logging in a consistent manner across all requests.

Finally, the authentication and authorization component handles security concerns in the application. It manages the access and refreshes tokens for OAuth 2.0 authentication, updates the CSRF token after an AJAX request, sets CSRF header before submitting a new request, and restricts view rendering using role based authentication.

1.2.2. RESTful API

The RESTful API exposes 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 will be fully documented using the apiDoc[2] documentation tool and will be published at the SLIPO project development web site.

The API will expose a select subset of the functionality provided by the SLIPO Service, such as allowing users to browse POI dataset metadata, invoking single operations instead of complex workflows, and monitoring the status of an operation execution.

In addition to OAuth 2.0 authentication, the API will also require the use of application keys. Using application keys, quota constraints will be applied to the owner of a given key. The quota constraints will control API aspects such as number of invocations per day, number of active operations, etc.

[2] http://apidocjs.com

1.2.3. SLIPO Web Application

The SLIPO Web Application is an MVC web application that exposes the operations of the SLIPO Service to the Workbench application and implements the RESTful API. The main components of the SLIPO Application Server are depicted in Figure 4.

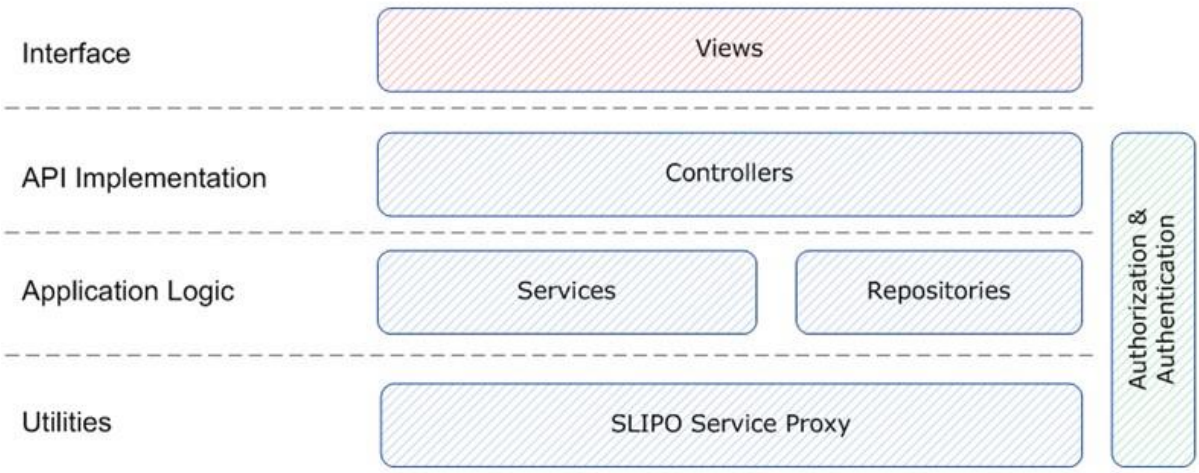

Figure 4: SLIPO Web Application components

The application has only a single anemic 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.

The controllers are lightweight components that implement the HTTP Action API and the RESTful API endpoints. The former is used exclusively by the Workbench application while the latter is available to third-party consumers. The controllers are responsible for parsing incoming messages (usually JSON messages), invoking services, serializing responses, and applying authentication and authorization rules.

Most of the time, a controller method just invokes a method from the SLIPO Service Proxy. If more complex operations are required, they are grouped and implemented as a single method in a separate service component. Services may access metadata about data sources and workflow executions stored in a relational database using Repositories.

Requests for browsing data either for datasets, or workflow executions, can be serviced by the SLIPO Web Application. If the user requests to process a POI dataset, then the request is submitted to the SLIPO service using the SLIPO Service Proxy component.

Finally, the Authorization and Authentication component handles all security aspects of the application including:

- User authentication using OAuth 2.0
- Role-based user authorization
- CSRF token validation for the HTTP Action API
- Application key validation and quota enforcement for the RESTful API

1.2.4. 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.

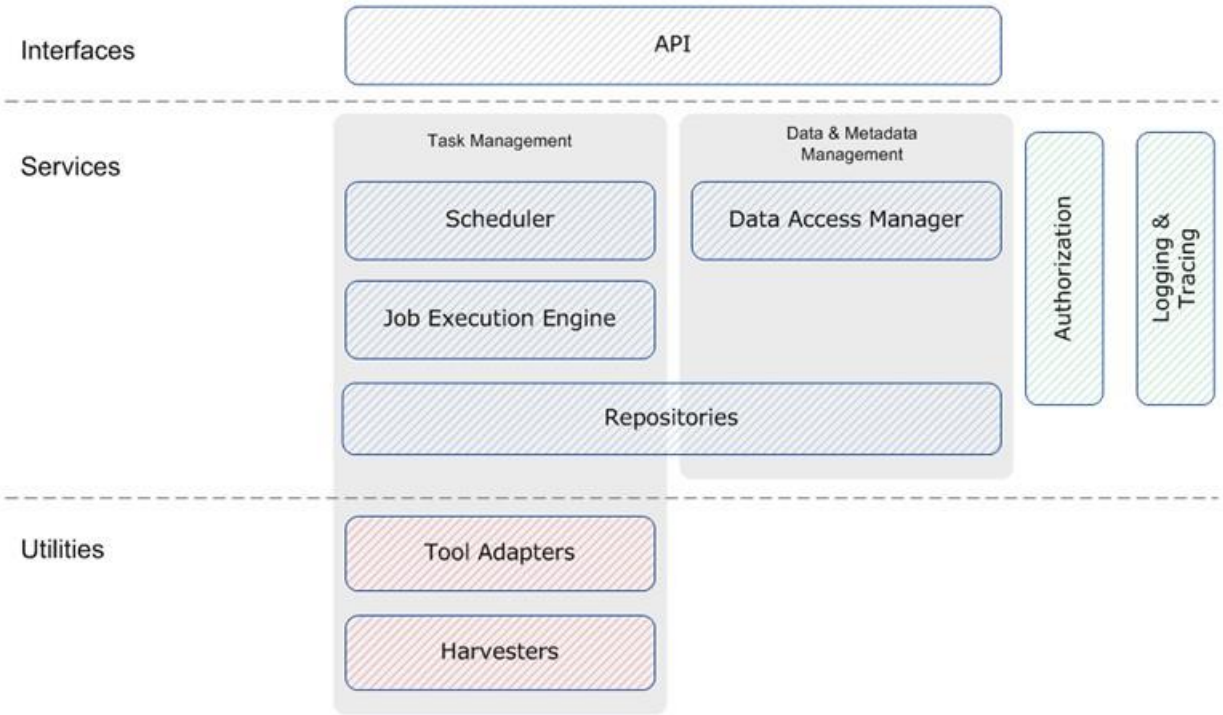

Figure 5: SLIPO Service components

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

- The **Scheduler** component initializes the execution of jobs at specific points in time. A job represents any processing task that must be executed asynchronously and (optionally) remotely. The most common types of jobs include the execution of data integration tasks, data analysis of Linked POI data, and data harvesting from external sources. Initially, scheduling will only depend on time constraints. We also plan to examine the possibility of making the scheduler aware of the available cluster resources thus preventing the execution of too many jobs when the cluster utilization is high. Finally, the execution time of a job is controlled using CRON[3] expressions.
- The **Job Execution Engine** orchestrates the execution of a job. 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.

[3] https://en.wikipedia.org/wiki/Cron

- The **Data Access Manager** is responsible of managing and publishing all application data to the Workbench application and RESTful API. Such data include POI dataset metadata, scheduler information, job execution status maintained by the Job Execution Engine, external tool configuration options, remote data sources and data harvester configuration options.
- **Repositories** are 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. The Data Manager may combine and aggregate data from multiple repositories to respond to Workbench application and RESTful API queries.
- **Tool Adapters** are 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** are 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.
- The **Authentication** module 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.
- The **Logging & Tracing** module 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 **API** allows external applications to submit requests to the SLIPO Service. It is implemented as a RESTful API that exchanges JSON messages. Many of the message types are shared with the Action API of the SLIPO Web Application and the SLIPO RESTful API since the operations of the latter two APIs are implemented by the SLIPO Service.

1.2.5. SLIPO Toolkit

The SLIPO Toolkit consists of several software components that support scalable and quality-assured POI data integration, including transformation, interlinking, fusion, enrichment and analytics. Integrating SLIPO Toolkit components in the SLIPO software stack requires us to efficiently address the following issues:

- **Software Integration**: How to integrate the execution of an external software component in the SLIPO Service and manage configuration, input and output files.
- **Workflow Orchestration**: How to orchestrate a data integration workflow that involves multiple operations.
- **Scalability**: How to encapsulate and invoke each software component in a manner that ensures its scalable execution without incurring dependencies from other components

In the next sections, we present the solutions we will adopt for the development of the SLIPO Toolkit.

1.2.5.1. Software Integration

In SLIPO we will be using several existing software components to implement different phases in the POI data integration lifecycle, i.e., transformation, interlinking, enrichment and fusion. In this section, we are discussing issues about integrating these components in the SLIPO architecture.

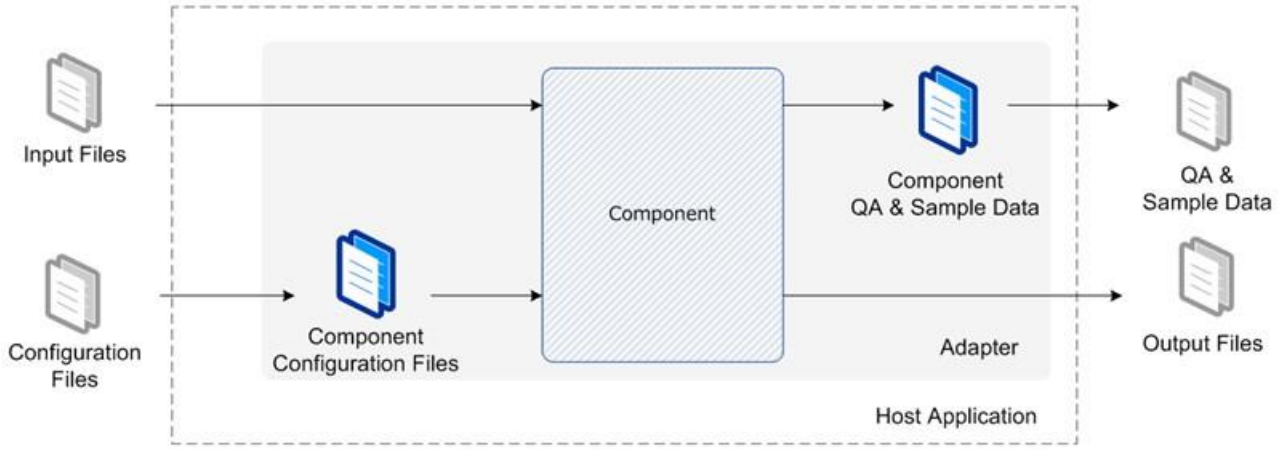

Figure 6: Software integration

All software components can be parameterized using external configuration files and either executed as a standalone service or integrated as an external library. Each individual component supports diverse configuration formats (e.g., plain text or turtle[4] files), accepts one or more input files, and generates one or more output files. Optionally, the output may include quality assurance metrics and sample data. To decouple implementation details and abstract disparate configuration file formats, we are going to wrap each library or service with an adapter component. The latter will be responsible for transforming a component-specific configuration format to a model suitable for the SLIPO Service and the SLIPO components higher in the software stack. The model should be easily serialized to JSON to be exposed by the SLIPO Web Application and consumed by the Workbench application. For every task in a data integration workflow that involves the execution of an external tool, an instance of the appropriate model will be created. The model will support the grouping of configuration options in two groups, namely, generic tool options and step specific options. Users will be able to create, persist and reuse named instances of the model with generic options only set, thus, creating reusable configuration templates.

Finally, if a component supports the generation of quality assurance metrics or sample data, the adapter component should be able to parse this data and export it to a format that can be easily aggregated by the host application. The diagram in Figure 6 displays the design described above.

1.2.5.2. Workflow Orchestration

A data integration workflow consists of one or more operations on POI datasets such as transformation, interlinking and fusion. Operations may be executed *sequentially*, in *parallel* or *conditionally*. In the SLIPO Service, we will use the Spring Batch[5] framework to model workflows and operations as Spring Batch jobs and tasks respectively. A simple workflow is illustrated in Figure 7. In this example, two input files are

[4] https://www.w3.org/TR/turtle/

[5] http://projects.spring.io/spring-batch/

transformed by TripleGeo, with subsequent interlinking performed by LIMES. The transformation of the input files executes in parallel. An optional set of parameters is used to initialize the job. All steps read/write data from/to the shared distributed file system. During the job execution, Spring Batch runtime logs information about the job and every execution step in a persistent store.

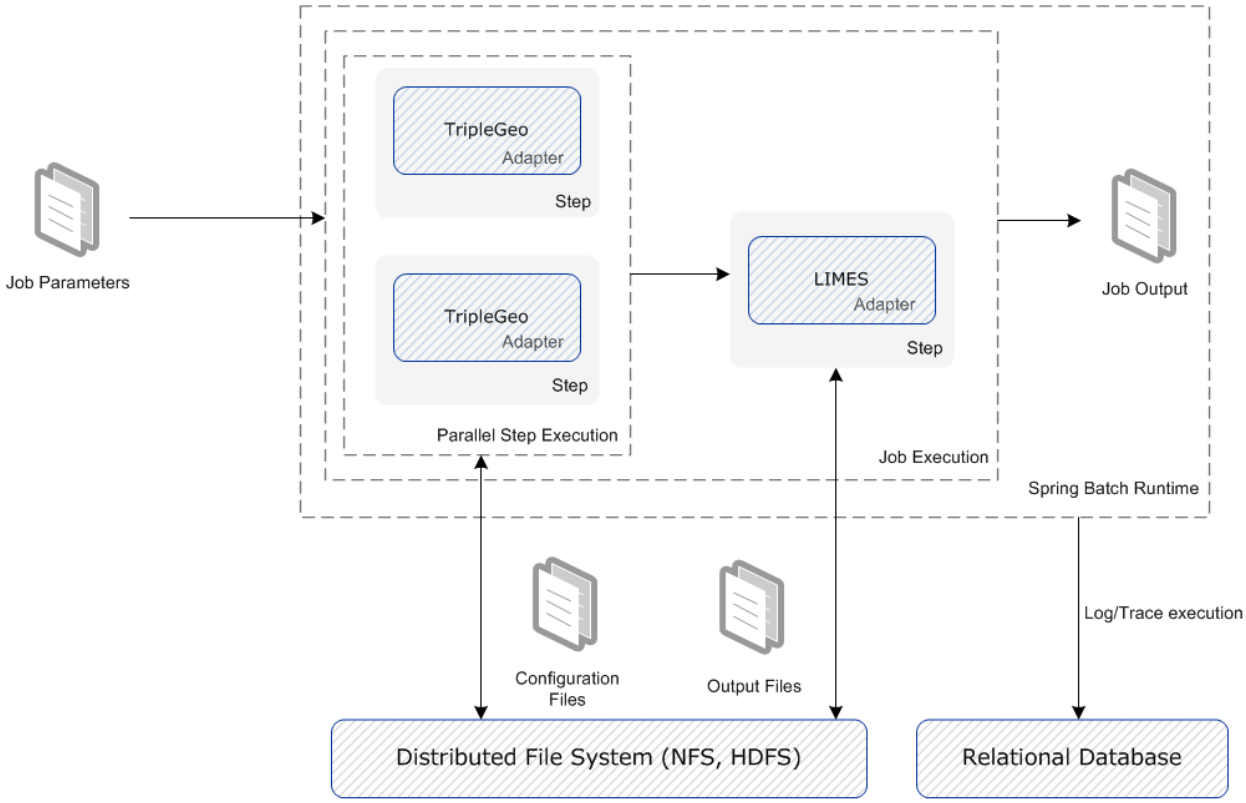

Figure 7: Simple Spring Batch Job

Every SLIPO Toolkit component is responsible for providing a scalable implementation for the requested operation. Spring Batch can initiate the execution of an operation locally, or on a remote server, but has no control over execution optimization, resource management and data partitioning. These issues are deferred to the tool been invoked. The SLIPO Toolkit components and SLIPO Service are deployed over a cloud infrastructure and communicate either by exchanging messages or sharing files stored on a distributed file system. The SLIPO Service is completely ignorant of the SLIPO Toolkit components implementation details.

1.3. Integration

In this section, we present the software development principles adopted in the project and the methodologies applied for integrating and testing all developed software.

1.3.1. Software Development principles

In the following, we provide a brief overview of the principles and methodologies followed in the project for software development, integration, testing, and release management. In summary, we employ an Agile

approach and the RERO (Release Early, Release Often) paradigm. Our methodological framework has been shaped both from our experiences in software development for open source projects (e.g., GeoKnow, DAIAD), as well as the requirements of SLIPO in terms of complexity, effort, and time constraints.

1.3.1.1. Agile

In the following we highlight specific Agile[6] practices employed in the project and, when required, the appropriate context for each one.

- **Pair-programming**. Software developers and researchers (PhD candidates, Research Associates) spend 20-50% of their time (depending on the actual task/functionality and importance) in pair-programming. This approach is also applied during internal testing (unit/integration) to speed-up the discovery and redress of bugs/issues.
- **Software that works**. Our emphasis is placed on delivering *working* software, deployed and tested on a production setting. This is sound both from a methodological perspective, but also absolutely needed to address the complexity of the complete SLIPO system.
- **Continuous change and development**. New or changed user requirements will be continuously collected from stakeholders and transferred in the development cycle, ensuring quick response to change.

1.3.1.2. Release Early, Release Often (RERO)

The RERO paradigm has its foundations deeply rooted in open source projects[7] (e.g., Linux kernel), focusing on maintaining *momentum* in development, as well as accelerating feedback received from end-users and testers. Its opposite is a strict and feature-based release schedule (i.e., release only if a new functionality is complete). Once again, the characteristics of the SLIPO system and its development time-frame, favors rapid vs. infrequent user feedback (e.g., weekly vs. quarterly). The implementation of RERO in the project will focus on short (1-3 weeks) development and testing cycles (i.e., sprints) across all various system components (e.g., APIs, libraries, UI elements).

1.3.1.3. Benevolent dictatorship and tight commit control

SLIPO is an open source project, creating, applying, and extending several existing mature open source projects to deliver its output. The governance and commit/attribution models in open source projects[8,9] vary greatly depending on their foundations, age, and popularity. The individual open source software components comprising the SLIPO Toolkit (see Section 2) will continue to follow their already established governance and commit control guidelines.

The remaining SLIPO components developed for the needs of the project (i.e., Workbench, API, Web Application, Service) consist a *new* open source project, focused on an innovation-centric agenda, with zero

[6] http://www.agilemanifesto.org/

[7] http://firstmonday.org/article/view/578/499

[8] http://oss-watch.ac.uk/resources/governancemodels

[9] http://producingoss.com/

external contributions, as well as a very tight timeframe for developing, testing, and validating its output. Towards this, we will adopt the following governance and collaboration directions.

- Benevolent dictator. Software development is steered by a single person (Yannis Kouvaras) with exclusive control over the directions and course of the software. With no external contributors (i.e., outside the project partners), this is the preferred governance model for young open source projects (especially during their incubation phase).
- Commit Control. The roles of code authors vs. committers are typically used interchangeably in open source development, but they are not actually the same. Commit roles/rights are a matter governance (commit access), policy (public attribution as incentives for participation) and technology (centralized vs distributed repos). In SLIPO, a Software Lead will be assigned to each software component (i.e., Workbench, API, Web Application, Service) in the project GitHub[10] organization repository. Code authors will submit contributions for each project to the assigned Software Lead. Each lead is responsible for receiving, reviewing, improving himself, or requesting further improvements from the authors. If the Leads are satisfied, the source code is committed by them in the corresponding public repository. This approach is proposed due to the strict timeframe of the project for delivering its beta prototype by M18, the high number of developers involved and the diverse software developed.

1.3.2. Integration and Testing

In the following, we summarize our methodology and practices for integrating and testing all software developed in the project. The integration and testing methodologies are as follows.

- Continuous integration (CI). Source code contributions are integrated and tested at very frequent intervals (from twice a day, to every couple of days). Due to the API-based and loosely-coupled nature of the SLIPO system, CI is often performed interdependently across its various components. For example, CI for the SLIPO Toolkit applications can be performed as often as needed, since each application is a standalone application and communication with the SLIPO Service is decoupled using wrapper components which can be easily replaced by testing mock components.
- Integration testing. We have adopted the following two methodologies for integration testing to accelerate development and ensure scalability.
    - Big Bang. This testing process is applied for the entire SLIPO system, following specific usage workloads that cover all usage aspects of the system, testing infrastructure, tester groups, and data. The workloads used for testing will be defined during the project development.
    - Risky-hardest. System APIs, which by definition are slowly changing and affect the entire system operation, are additionally tested first using the Risky-hardest methodology. When a new API version is available that provides a solution to a previously identified problem or new functionality, integration testing begins by focusing on the API itself.

[10] https://github.com/SLIPO-EU

- Staging and Production testing. For development, integration, and testing purposes we will deploy several computing infrastructures presented in detail in the next section.

1.4. Deployment

For testing the SLIPO system, we will deploy the SLIPO applications and services on top of Athena RC's private IaaS cloud implemented via the Synnefo cloud stack[11]. Synnefo is a complete open source cloud stack written in Python that provides Compute, Network, Image, Volume and Storage services, similar to those offered by AWS[12]. On a hardware level, the production system will be hosted on Athena RC's own server infrastructure allocated for SLIPO (120 CPU cores, 92 GB main memory, 20TB storage), which will be scaled horizontally (scale-out) depending on system performance and utilization. Figure 8 presents the data center where the physical hardware is hosted. Automation of deployment and updating will be handled by Ansible scripts. In the next sections, the SLIPO deployment scheme, the Synnefo cloud stack and the deployment orchestration method are described.

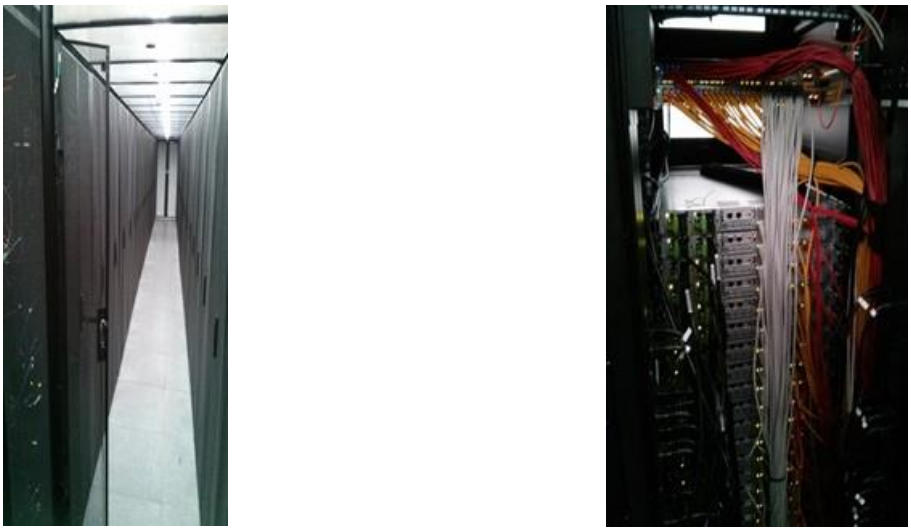

Figure 8: SLIPO production servers hosted at the data center of Greek Ministry of Education

1.4.1. Overview

An overview of the SLIPO deployment scheme is illustrated in Figure 9. The installation will comprise the following virtual machines:

- A single administration server that acts: (a) as a gateway for accessing the cluster private network and administrating hosts remotely using SSH, and (b) as a proxy for the HDFS, YARN and Flink administration sites.
- An application server for hosting the SLIPO Web Application, servicing the Workbench web application and exposing the SLIPO RESTful API.

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

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

- A build server for managing software integration tasks, Ansible scripts and application installation to other servers in the cluster.
- An application server for hosting the SLIPO Service and a PostgreSQL instance.
- An Apache Flink cluster for performing data analysis of Linked POI data using the SANSA framework. Optionally, Flink may be deployed over YARN for fine-grained resource management.
- An HDFS cluster for storing POI datasets.

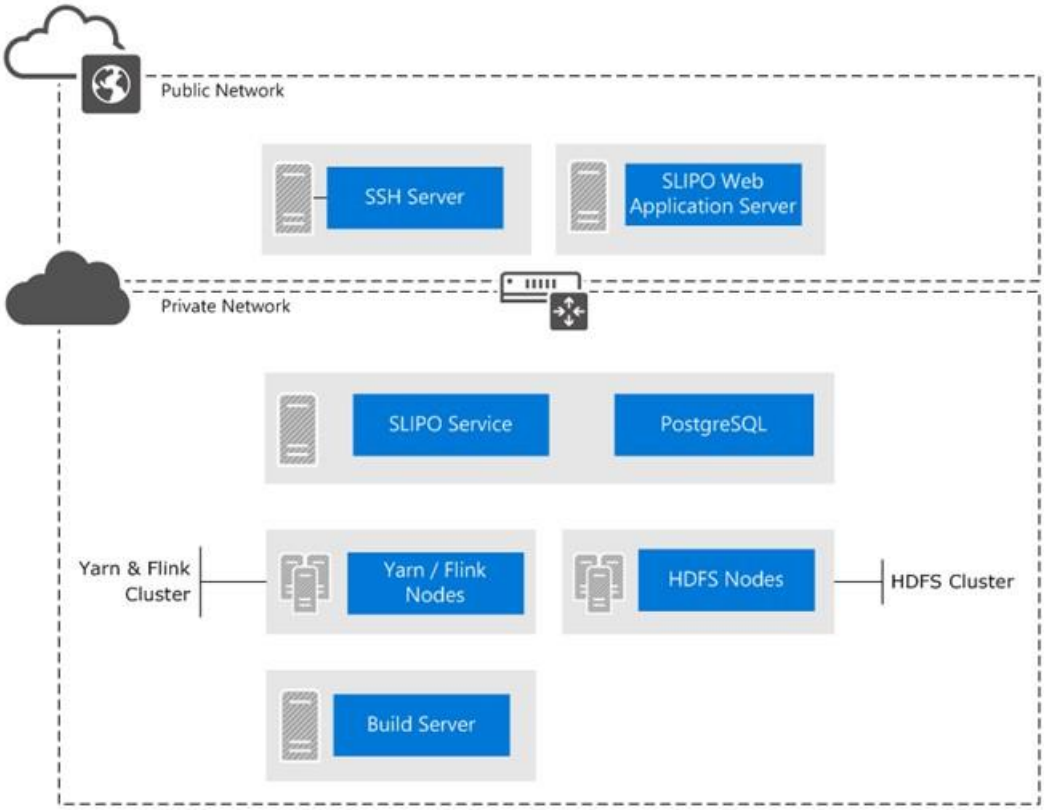

Figure 9: SLIPO system deployment overview

1.4.2. Synnefo

Synnefo manages multiple Ganeti[13] clusters at the backend for handling of low-level VM operations and uses Archipelago[14] to unify cloud storage. To boost third-party compatibility, Synnefo exposes the OpenStack APIs[15] to users. The Cyclades (Synnefo UI) allows administrators to maintain and manage the cloud resources of the system (restart, shut down, start up, delete and create a virtual machine). In Figure 10, an overview of the Synnefo services is presented.

[13] http://www.ganeti.org/

[14] https://www.synnefo.org/docs/archipelago/latest/

[15] https://developer.openstack.org/api-guide/quick-start/index.html

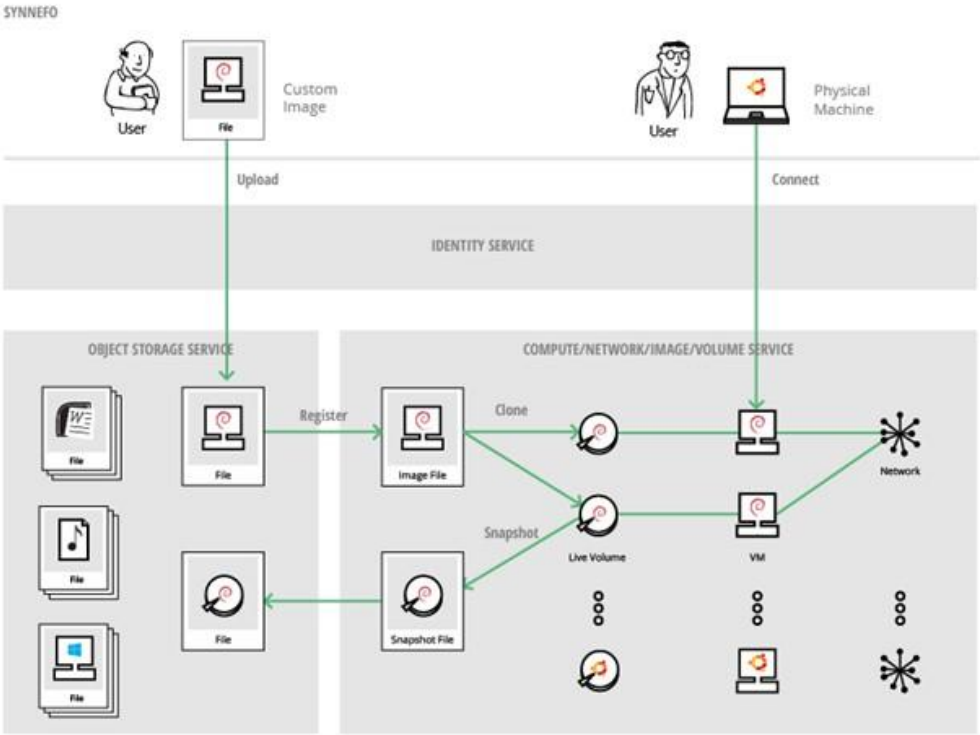

Figure 10: An overview of the Synnefo services

Synnefo keeps a clear separation between the traditional cluster management layer and the cloud layer. This unique design approach leads to a completely layered architecture boosting production readiness, maintainability and upgradability. The modular design allows for linear scalability, gradual extensibility and ease of operations. In Figure 11 an overview of the Synnefo architecture is presented, showing the major components on each layer.

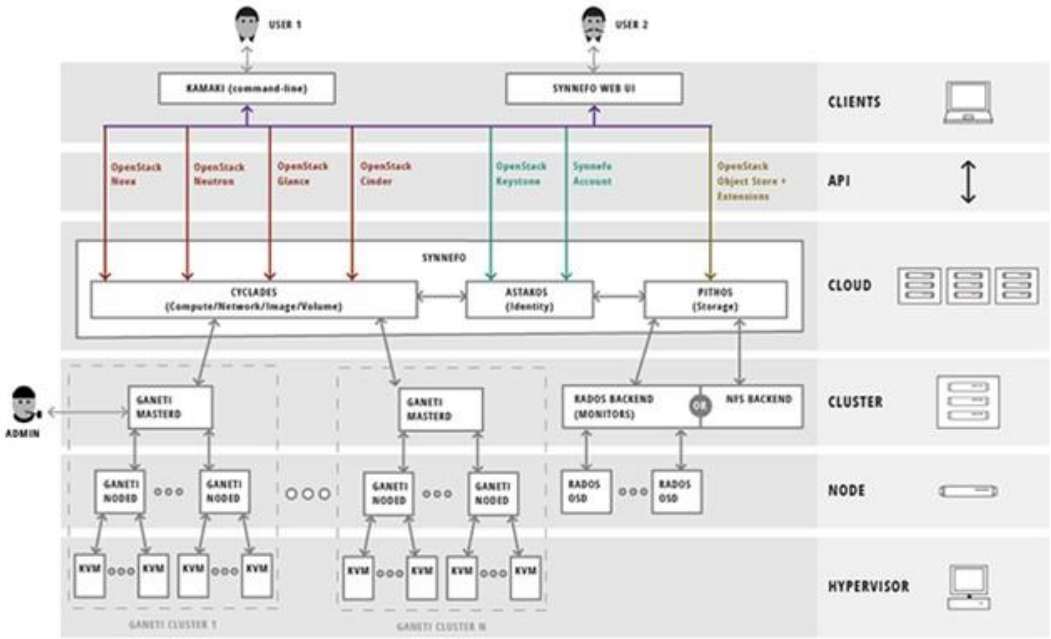

Figure 11: The system architecture of Synnefo

For all VMs inside the cluster, basic system-wide monitoring is available from the Synnefo layer. The collected statistics can be exported to a table-like format via Synnefo API, or (part of them) can be directly visualized from Synnefo UI.

1.4.3. Deployment Orchestration

For facilitating the deployment of SLIPO, we will create a GitHub repository for storing the scripts and configuration files needed for the several node groups. The tool used to automate the deployment and update process is Ansible.

Ansible is a radically simple IT automation engine that automates cloud provisioning, configuration management, application deployment, intra-service orchestration, and many other IT needs. Being designed for multi-tier deployments since day one, Ansible models IT infrastructure by describing how all systems inter-relate, rather than just managing one system at a time. It uses no agents and no additional custom security infrastructure, so it's easy to deploy – and most importantly, it uses a very simple language (YAML, in the form of Ansible Playbooks) that allows administrators to describe their automation jobs in a way that approaches plain English.

Ansible works by connecting to the system nodes and pushing out small programs, called "Ansible Modules" to them. These programs are written to be resource models of the desired state of the system. Ansible then executes these modules (over SSH by default), and removes them when finished. The library of modules can reside on any machine, and there are no servers, daemons, or databases required. Typically, the administrator works with a terminal program, a text editor, and probably a version control system to keep track of changes (e.g., git[16]). Ansible represents what machines it manages using a very simple INI file that puts all the managed machines in groups. Playbooks can finely orchestrate multiple slices of the infrastructure topology, with very detailed control over how many machines to tackle at a time.

The work of parameterizing and automating the deployment process will evolve as new software components of SLIPO are implemented.

[16] https://git-scm.com/

2. SLIPO Toolkit Components

The SLIPO Toolkit comprises the individual software components that will be applied and extended by the project to handle the scalable integration of Linked POI Data. For every component, we provide a short description, its internal architecture, its dependencies to external libraries, its role in the SLIPO data integration lifecycle, and a development roadmap of the features that will be developed during the project. This section focuses specifically on the six components that comprise the core SLIPO Toolkit, addressing the major tasks of the SLIPO data integration lifecycle: *Transformation (TripleGeo, Sparqlify), Interlinking (LIMES), Enrichment (DEER, OSMRec), Fusion (FAGI).*

2.1. TripleGeo

In this section, we outline the plan for developing and extending the TripleGeo software in the context of the project. First, we summarize the features already available in the current version (v.1.1) of the suite. Then, we present the planned additions to its functionality, as inferred by the SLIPO requirements, and discuss some specific issues and targeted actions that will guide the implementation of the next version (v.2.0) of TripleGeo as part of the SLIPO Toolkit.

TripleGeo is open source software and it can be redistributed and/or modified under the terms of the GNU General Public License[17] as published by the Free Software Foundation; either version 3.0 of the License, or (optionally) any later version.

2.1.1. Version 1.1 Features

Several ETL tools have been available for converting between geospatial formats, but only a few specifically addressed the emerging needs of geospatially-enabled RDF stores. In 2013, in the context of the GeoKnow[18] project, TripleGeo[19] was developed, an open-source ETL utility that can extract geospatial features from various sources and transform them into triples for subsequent loading into RDF stores. TripleGeo is the first tool that enabled conversion of geospatial features from several sources and formats into GeoSPARQL-compliant serializations according to the OGC GeoSPARQL[20] standard.

More specifically, the aim was to bridge the gap between typical geographic representations from a variety of proprietary files, DBMSs, and georeferenced systems with the demands of geospatially-enabled RDF stores. Development was initially based on the open-source geometry2rdf[21] library, but with notable modifications and substantial enhancements to meet interoperability needs in RDF stores. In fact, TripleGeo is designed as a spatial ETL tool, enabling users to:

[17] http://www.gnu.org/licenses/gpl-3.0.html

[18] http://geoknow.eu/

[19] https://github.com/GeoKnow/TripleGeo

[20] https://portal.opengeospatial.org/files/?artifact_id=47664

[21] https://github.com/boricles/geometry2rdf

- Extract spatial data from a source
- Transform this data into a triple format and geometry vocabulary prescribed by the target RDF store
- Load resulting triples into the target RDF store

Therefore, TripleGeo always preserves data integrity and provides consistent, well-defined geospatial information to end users.

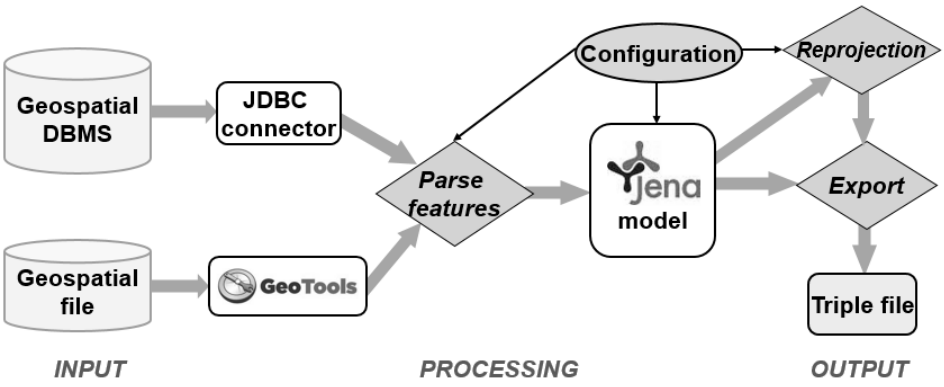

Figure 12: Processing flow diagram for ETL utility TripleGeo 1.1

Figure 12 illustrates the processing flow used for converting geospatial features into RDF triples. Among its distinctive features, we point out that the current version 1.1 of TripleGeo can:

- Directly access de facto geographic formats (e.g., ESRI shapefiles, GML, KML) or DBMSs (IBM DB2, MySQL, Oracle Spatial, PostgreSQL/PostGIS).
- Recognize many geometric data types, i.e., not only points, but (multi-)linestrings and (multi-)polygons as well.
- Extract a limited number of thematic attributes (e.g., identifiers, names, or characterizations), associated with each feature.
- Allow on-the-fly reprojection between any established Coordinate Reference Systems (CRS), e.g., transform geometries from GreekGrid87 to WGS84.
- Provide integrated transformation of INSPIRE-aligned spatial data and metadata into RDF using XSL stylesheets for selected INSPIRE data themes[22]. This allows geospatial data (standards-compliant or not) to be transformed to RDF and exposed through GeoSPARQL with limited effort.
- Export triples into various serializations (RDF/XML, NTRIPLES, TTL, etc.) and geometry vocabularies for swift loading into RDF stores.

From a user's perspective, this command-line utility is entirely automated and based on preconfigured settings. A configuration file declares user preferences concerning all stages of the conversion: how input source will be accessed, which data is involved, what geometric representation should be used, whether geometries must be re-projected into another CRS, as well as the output triple notation.

[22] http://inspire.ec.europa.eu/Themes/Data-Specifications/2892

2.1.2. Towards TripleGeo 2.0

Thanks to its modular implementation, TripleGeo can be enhanced with more utilities without affecting existing functionality. During the project, we will further extend TripleGeo with several novel features, and most importantly, specific functionalities that can support the scalable transformation of large POI datasets. Next, we outline the key points in developing the next major release (v.2.0) of TripleGeo. We also indicate a time schedule for their implementation in two stages: an initial functional release to be integrated early enough within the SLIPO Service and Workbench, as well as the final full-fledged version that will cover all transformation requirements for SLIPO. Next, we present the new features that will be implemented, the usability options in the data integration lifecycle, and performance improvements to satisfy our scalability goals.

2.1.2.1. New Features

The new functionality provided by TripleGeo v.2.0 will include:

- Ability to apply mappings and vocabularies and export both geometric and thematic attributes of the original dataset under a given OWL ontology. This will enable incorporating and deploying mappings between identified concepts and properties of the source data and the target ontology into the transformation mechanism. For compatibility, such mappings should ideally be expressed in a widely-used mapping language (e.g., R2RML[23] or D2RQ[24]) and we will apply best practices in the way original features are mapped into RDF concepts and properties. We will also examine whether TripleGeo should be extended with semi-automatic workflows for guiding the user into creating new mappings (e.g., in transforming datasets whose schema is not mapped to an existing POI ontology) or this functionality should preferably become an autonomous tool in the SLIPO architecture. [Initial release: M12, Final release: M36]
- Integration with URI identifier creation. The methodology that will be developed in SLIPO for creating persistent, unique, vendor and technology independent POI URIs will be integrated in the transformation process performed in TripleGeo. Furthermore, this functionality will become available for any other source dataset (i.e., even non-POI data) and replace the existing hard-coded creation of URIs by user-specified namespaces. This will allow users to customize the creation of linked identifiers and to choose a specific naming strategy for URIs (e.g., as a combination of metadata of the original feature). [Initial release: M12, Final release: M36]
- Interaction with other geographic data sources (e.g., GPX, CSV), de facto POI formats (like TomTom Overlay[25] files, OziExplorer[26] Waypoints etc.) and DBMS platforms (e.g., MS SQL Server spatial, SpatialLite). [Initial release: M12, Final release: M24]
- Support for more complex geometric types (e.g., geometry collections) is equally important for acquiring the most detailed representation of spatial entities instead of point coordinates of centroids. [Final release: M12]

[23] https://www.w3.org/TR/r2rml/

[24] http://d2rq.org/d2rq-language

[25] http://www.tomtom.com/lib/doc/ttnavsdk3_manual.pdf

[26] http://www.oziexplorer3.com/eng/help/fileformats.html

The usability options of TripleGeo in the SLIPO POI data integration lifecycle involve:

- Specialization to POI transformation is of major importance for SLIPO, so TripleGeo will be extended and optimized to effectively support POIs, with specific vocabularies and operations. TripleGeo will be aligned with the global POI ontology to be developed in SLIPO, thus allowing representation of more complex POI metadata and relations to facilitate the POI integration lifecycle. This way, TripleGeo should be able to provide support for vocabularies and mappings specifically for POI data handled in SLIPO (e.g., TomTom). Furthermore, we will examine whether TripleGeo should also need to handle vocabularies and mappings to existing custom POI schemata (e.g., OSM, Wikimapia) for effective manipulation of large-scale open geodatasets. [Initial release: M12, Final release: M36]
- Integration with the SLIPO Workbench, will provide users with a GUI or a RESTful API that facilitates customization of the transformation process. Instead of the current command-line interface, this GUI may expose the full functionality of TripleGeo for disk-based files, tables in a DBMS or web-accessible data, and offer a unified web interface to extract large POI datasets, convert them and produce their RDF representations. TripleGeo can be thus incorporated into the SLIPO toolkit and POI data integration workflows in a coherent, user-friendly, and flexible manner under the entire SLIPO data lifecycle (i.e., transformation, interlinking, fusion, and enrichment of POIs). [Initial release: M12, Final release: M36]
- Last, but not least, reverse transformation from RDF into de facto geospatial formats is required. TripleGeo will support reverse transformation of RDF POI data (potentially interlinked or fused in later stages) into de facto POI formats (e.g., shapefiles, GPX, CSV, etc.). Of course, there exists an impedance mismatch in this direction, given that the POI ontology is semantically more expressive than the conventional POI schemata, thus POI attributes, relations and metadata will be richer than what can be supported by conventional formats. To address this issue, we will define and implement the optimal reverse transformations that will allow the incorporation of the maximum amount of semantic (linked, enriched, fused) POI information and metadata into the available properties of conventional POI formats. [Initial release: M12, Final release: M24]

2.1.2.2. Scalability

Scalability with increasing data volumes is most challenging. Hence, a parallelization framework in transforming input features and generating RDF triples would be advantageous. Assuming that *n* processing nodes are available, we will opt for solutions that employ data partitioning of the input into *n* disjoint batches, so that TripleGeo can be invoked in *n* separate instantiations, each one executed in a processing node in isolation from the rest. Such a partitioning may be based on several alternative schemes. For example, the original file may be split into *n* smaller batches with equal number of records in each one. Besides, splitting can be based on spatial criteria, e.g., employing a subdivision of the space into *n* disjoint regions (e.g., a tessellation into cells), and creating a batch for each region with all entities contained therein. Of course, each original spatial entity with all its attributes will be included in only one such subset. Such partitioning has absolutely no impact on transformation to RDF, as each entity is transformed independently from the rest. By employing *n* concurrently running instantiations of TripleGeo to transform each dataset, and emitting triples into flexible RDF storage schemes (e.g., HDFS), we expect that scalability of transformation will be greatly improved.

For the current edition of TripleGeo, we had conducted a performance study[27] in the context of GeoKnow project. Indicatively, we noticed that transformation took less than 3 minutes for an OpenStreetMap layer with around 590,000 point geometries (including cost of writing the RDF triples to disk files). In contrast, 2,600,000 linestring geometries required much more time, about two and a half hours to conclude the transformation. Such delays should be mostly attributed to memory shortage, as the entire RDF model was retained in main memory and grew proportionally to the amount of statements generated per initial record, so it had to spill on disk in case of excessive load. This case signifies that triple extraction for large datasets with millions of features should better be performed in several smaller batches of the original data, as in many modern processing paradigms and exactly as we intend to do in SLIPO. According to our planned framework, our goal is to support RDF transformation of 100 million POIs in less than a minute, thus aiming to achieve orders of magnitude performance gains compared to the previous edition of the software. [Final release: M36]

2.1.3. Dependencies

TripleGeo has dependencies to several open-source tools and libraries:

- Apache Jena[28]: A Java framework for building Semantic Web applications.
- GeoTools[29]: A Java library that provides OGC standards compliant methods for geospatial data management.
- GDAL/OGR[30]: A Geospatial Data Abstraction Library [GDAL] with a variety of useful command-line utilities for data translation and processing of raster and vector datasets.
- Java Topology Suite (JTS)[31]: An API of 2D spatial predicates and functions conforming to the OGC Simple Features Specification for SQL.

2.2. Sparqlify

In this section, we outline the plan for developing and extending the Sparqlify software in the context of the project. First, we summarize the features already available in the current version of the software. Then, we present the planned additions to its functionality, as inferred by the SLIPO requirements, and discuss some specific issues and targeted actions that will guide the implementation of the next version of Sparqlify as part of the SLIPO Toolkit.

Sparqlify is provided as free software and its current version (including the Java source code and sample data) is available from Github[32]. It can be redistributed and/or modified under the terms of the Apache License, Version 2.0[33].

---

[27] http://svn.aksw.org/projects/GeoKnow/Public/D2.2.1_Integration_of_Geospatial_Databases.pdf

[28] https://jena.apache.org/

[29] http://www.geotools.org/

[30] http://www.gdal.org/

[31] http://docs.geotools.org/latest/userguide/library/jts/

[32] https://github.com/AKSW/Sparqlify

[33] Apache License, Version 2.0

2.2.1. Version 1.0 Features

Sparqlify is a SPARQL-to-SQL rewriter system that enables leveraging relational data on-the-fly as RDF. The system achieves this by rewriting SPARQL queries to corresponding SQL-based query plans with regard to a given set of declarative mappings. Sparqlify is optimized for SPARQL 1.0 and supports spatial datatypes and functions. In fact, the system allows any SQL function to be exposed in a corresponding SPARQL syntax. For convenience, SPARQL 1.1 (especially property paths) is supported, however this comes with performance limitations, as a Jena wrapper is used to compile such queries to plans that only make use of SPARQL 1.0 features.

Sparqlify supports two mapping formants: the standard R2RML and the more human friendly Sparqlification Mapping Language[34] (SML). Sparqlify is most prominently deployed at the LinkedGeoData community project where a public SPARQL endpoint is made available, backed by a replicated OpenStreetMap Postgres database.

The architecture of the current version is depicted in Figure 13. The main components are core, server and platform. The core provides the core SPARQL-to-SQL rewriting functionality, which is based on algebraic transformations, in regard to models of the SQL and SPARQL type systems and functions. The server module exposes the SPARQL query capabilities as an HTTP SPARQL endpoint together with a basic HTML web interface. In addition, the platform features a Linked Data interface via Pubby[35], a Linked Data frontend for SPARQL endpoints. The platform supports auto-configuring the mapping of dataset IRIs to the host URLs under which they are published based on the RDB2RDF mapping specifications.

Conceptually, upon SPARQL query execution, the Sparqlify engine will: (i) for each triple pattern in the query, find all predicate tables that can yield appropriate triples, and (ii) for each graph pattern, construct a UNION of JOINs.

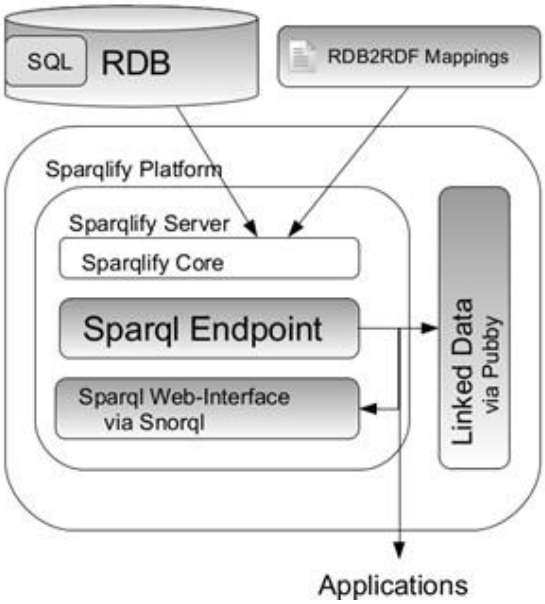

Figure 13: Sparqlify architecture

[34] http://sml.aksw.org/

[35] http://wifo5-03.informatik.uni-mannheim.de/pubby/

2.2.2. Towards Sparqlify 2.0

The major goal of Sparqlify in the frame of SLIPO will be to allow the transformation and serving of massive volumes of POI datasets, stored in database tables, as RDF data. To this end, our future development focuses on facilitating and evaluating SPARQL query processing on the Big Data frameworks Apache Spark and Flink by means of SPARQL-to-SQL rewriting. The following efforts are required to reach these goals:

- Partitioning of RDF data into the tabular data structures supported by Apache Spark and Flink.
- Configuration of the SPARQL-to-SQL rewriter according to the partitions.
- An RDF/SPARQL API tailored to Big Data frameworks.
- Optional optimization by precomputation of joins.

As Spark and Flink do not ship with any native RDF support, we intend to develop our solution in the context of the SANSA[36] library, which is also part of the SLIPO toolkit. SANSA features a layered architecture, where each layer defines a set of core interfaces aimed at bridging gaps between Semantic Web and Big Data technologies. In the remainder of this section we outline the components to be built in more detail.

2.2.2.1. New Features

The new functionality provided by Sparqlify v.2.0 will include:

- Data Partitioning and RDF Mapping. We will build a new data partitioner implementation which partitions an RDF graph into fine-grained predicate tables: For every predicate in an RDF dataset, partition tables with any of the following schemas are created whenever demanded by the data. The subject column, denoted by s, holds the full IRI of a triple's subject. The predicate is always implicit. The different schemas reflect the different types of RDF literals:
    - Standard Numeric Literals
    - Spatial Literals
    - Literals without language tag
    - Literals with (non-empty) language tag
    - Literals with custom datatypes
    - IRI Object
- Based on the partition information, the respective RDB-to-RDF mappings are then constructed.
- In the future, we plan to experiment with referencing subjects indirectly via a dictionary that maps integer IDs to IRIs. This may improve performance due to the smaller data volumes that need to be sent over the network.

2.2.2.2. Scalability

Regarding scalability, our goal will be to provide a Big Data SPARQL Query Processing API. Big Data frameworks generally abstract data as virtual collections that represent the result of the execution of an underlying workflow. Hence, such collections at the core provide primitive operators (e.g., filter or map) which extend a workflow specification rather than immediately performing the requested operation. This

[36] http://sansa-stack.net

gives the workflow processor the opportunity to optimize or prune certain operations. This is also different from conventional query API, such as JDBC or Jena, where the result of a query execution is an iterator over the result items rather than an abstract collection backed by a declarative workflow. Consequently, we envision to devise Big Data paradigm compliant abstractions for RDF graphs and SPARQL result sets. Note, that these are the only collections involved across all SPARQL query forms: *SELECT => result set; CONSTRUCT => RDF graph; DESCRIBE => RDF graph; ASK => boolean literal;* The main development effort for the API will be to abstract away the fact that an RDF graph is no longer directly a native Spark/Flink collection, but a union of such native collections according to the partitions with corresponding RDF mapping information.

2.2.3. Dependencies

Sparqlify has dependencies to several open-source tools and libraries:

- Apache Jena: A Java framework for building Semantic Web applications.
- Jena-sparql-api: An extension framework to Apache Jena for transparent fine-grained control over aspects of SPARQL processing, such as pagination and caching. Includes a base implementation of a SPARQL web server.
- Google Guava: A set of libraries that includes collection types (such as multimap and multiset), immutable collections, a graph library, functional types, an in-memory cache, and APIs/utilities for concurrency, I/O, hashing, primitives, reflection and string processing.
- JUnit: A simple framework to write repeatable tests.
- SLF4J: The Simple Logging Facade for Java (SLF4J) serves as a simple facade or abstraction for various logging frameworks.
- Apache Flink: An open source platform for distributed stream and batch data processing
- Apache Spark: A fast and general engine for large-scale data processing.

2.3. LIMES

In this section, we outline the plan for developing and extending the LIMES software in the context of the project. First, we summarize the features already available in the current version (v.1.0) of the software. Then, we present the planned additions to its functionality, as inferred by the SLIPO requirements, and discuss some specific issues and targeted actions that will guide the implementation of the next version (v.2.0) of LIMES as part of the SLIPO Toolkit, providing an indicative timeline for the implementation of the above.

LIMES is available under a dual license. For any non-commercial uses, the terms of the GPL3.0 license (version of June 29, 2007) hold[37]. Each commercial use is subject to fees, which are to be regulated contractually with the Semantic Data Management Research Unit of the Institute for Applied Informatics e.V.

[37] https://github.com/AKSW/LIMES-dev/blob/master/LICENSE

2.3.1. Version 1.0 Features

LIMES, the **Li**nk Discovery Framework for **Me**tric **S**paces, is a framework for discovering links between entities contained in Linked Data sources. LIMES is a hybrid framework that combines the mathematical characteristics of metric spaces as well prefix-, suffix- and position filtering to compute pessimistic approximations of the similarity of instances. These approximations are then used to filter out a large amount of those instance pairs that do not suffice the mapping conditions. By these means, LIMES can reduce the number of comparisons needed during the mapping process by several orders of magnitude and complexity without losing a single link.

The LIMES framework consists of eight main modules of which each can be extended to accommodate new or improved functionality. The central module of LIMES is the **controller** module (Figure 14 (left)), which coordinates the matching process, via the coordination of all the modules depicted in Figure 14 (right). The matching process is carried out as follows: First, the **controller** calls the **configuration** module, which reads the configuration file and extracts all the information necessary to carry out the comparison of instances, including the URL of the SPARQL-endpoints of source (S) and the target (T) knowledge bases, the restrictions on the instances to map (e.g., their type), the expression of the metric to be used and the threshold to be used.

Given that the configuration file is valid w.r.t. the LIMES Specification Language (LSL), the **query** module is called. This module uses the configuration for the target and source knowledge bases to retrieve instances and properties from the SPARQL-endpoints of the source and target knowledge bases that adhere to the restrictions specified in the configuration file. The query module writes its output into a file by invoking the **cache** module. Once all instances have been stored in the cache, the controller chooses between performing Link Discovery or Machine Learning. For Link Discovery, LIMES will re-write, plan and execute the Link Specification (LS) included in the configuration file, by calling the **rewriter**, **planner** and **engine** modules resp. The main goal of Link Discovery is to identify the set of links (mapping) that satisfy the conditions opposed by the input LS. For Machine Learning, LIMES calls the **machine learning** algorithm included in the configuration file, to identify an appropriate LS to link S and T. Then it proceeds in executing the LS. For both tasks, the mapping will be stored in the output file chosen by the user in the configuration file. The results are finally stored into an RDF or an XML file. A detailed overview of LIMES flow of execution is illustrated in Figure 14.

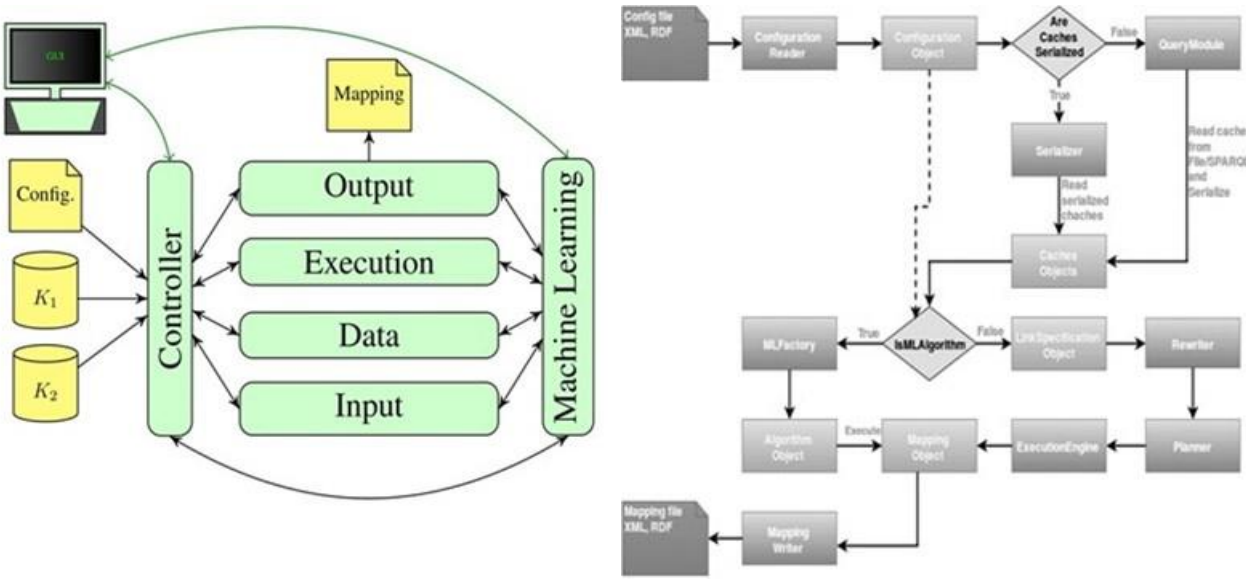

Figure 14: Overview of LIMES flow of execution

The advantages of LIMES are manifold. First, it implements highly time-optimized mappers, making it a complexity class faster than other Link Discovery Frameworks. Thus, the larger the problem, the faster LIMES is, w.r.t. other Link Discovery Frameworks. Secondly, LIMES supports a large set of string, numeric, topological and temporal similarity metrics, that provide the user with the opportunity to perform various comparisons between resources. In addition, LIMES is guaranteed to lead to exactly the same matching as a brute force approach, while at the same time reducing significantly the number of comparisons. Finally, LIMES supports a large number of input and output formats and can be extended very easily to fit new algorithms, new datatypes, new preprocessing functions and others thank to its modular architecture.

In general, LIMES can be used to set links between two data sources, e.g., a novel data source created by a data publisher and existing data source such as DBpedia. This functionality can also be used to detect duplicates within one data source for knowledge curation. The only requirement to carry out these tasks is a simple XML-based or TURTLE-based configuration file.

2.3.2. Towards LIMES 2.0

Similar to TripleGeo, LIMES also is based on a modular implementation. During SLIPO, we will further extend LIMES with several novel features and, most importantly, specific functionalities that can support the scalable interlinking of large POI datasets. Here, we outline the main points in developing the next major release (v. 2.0) of LIMES. We also indicate a time schedule for their implementation in two stages: an initial functional release to be integrated early enough with the SLIPO Service and Workbench, as well as the final full-fledged version that will cover all the POI interlinking requirements for SLIPO. Next, we present the new features that will be implemented, the usability options in the data integration lifecycle and performance improvements to satisfy our scalability goals.

2.3.2.1. New Features

The new functionality provided by LIMES v.2.0 will include:

- Linking of 5D POI resources. Using LIMES framework, we will harness the novel paradigm of 5D modelling for POIs by dealing with 2D geospatial coordinates, time and scale (also known as Level of Detail, resolution or granularity). The fifth dimension, granularity, will be employed for the optimization of the spatial interlinking approaches implemented in the LIMES framework. [Initial release: M18, Final release: M36]
- Learning of class-expression-specific specifications. In this task, we aim at discovering POI specific relations. In particular, the learning of class-expression-specific specifications will allow for the tuning of geospatial proximity thresholds based on POI types/categories. [Initial release: M18, Final release: M36]
- Discovery of temporal relations according to Allen's interval algebra. In particular, we will extend the scalable approaches for Euclidean spaces implemented in LIMES to the discovery of temporal relations (according to Allen's interval algebra and extensions thereof) between POIs such as for example temporal overlaps between POIs. [Initial release: M18, Final release: M36]
- Combining the new techniques with the ones already in LIMES. In particular, the novel optimized functions will be combined with the other mappers in LIMES using set theory operators to allow for hybrid similarity functions and the configurable weighting of specific POI metadata. [Initial release: M18, Final release: M36]

2.3.2.2. Scalability

In LIMES, we aim to develop scalable approaches for de-duplicating and interlinking massive, heterogeneous, and incomplete POI data at a world-scale. These will be scalable to handling billions of RDF triples and optimized according to the developed schema and ontology for managing POI data. We will also tackle the challenges arising from the multiple and inherent sources of ambiguity in POI data (spatial, temporal, semantic), as well as the inherent multi-linguality of location-related information. Scalability in linking 5D POI resources will be achieved as follows: By computing the error engendered by the usage of low-resolution data, we will be able to discard a large number of similarity computations when running our interlinking approaches and therewith achieve significantly better scalability than the state of the art. We will aim at achieving interlinking of 10 Million POIs in less than 10 minutes.

2.3.3. Dependencies

LIMES has dependencies to the following open-source tools and libraries:

- JUnit[38]: A simple framework to write repeatable tests.
- Commons Collections[39]: A library that adds many powerful data structures that accelerate development of Java applications.
- Commons FileUpload[40]: A library that makes it easy to add robust, high-performance, file upload capability to web applications.

[38] http://junit.org/junit4/

[39] http://commons.apache.org/proper/commons-collections/

[40] http://commons.apache.org/proper/commons-fileupload/

- Commons Lang[41]: A library that provides extra methods for Java core classes.
- ICU - International Components for Unicode[42]: A collection of Java libraries providing Unicode and Globalization support for software applications.
- Java Topology Suite (JTS): An API of 2D spatial predicates and functions conforming to the OGC Simple Features Specification for SQL.
- Apache Jena: A Java framework for building Semantic Web applications.
- Apache Log4j[43]: An open-source Java logging library.
- Jansi[44]: A small java library that allows to use ANSI escape sequences to format console output.
- SLF4J[45]: The Simple Logging Facade for Java (SLF4J) serves as a simple facade or abstraction for various logging frameworks.
- lanterna[46]: Easy console text GUI library for Java.
- Mime Detection Utility[47]: An open source java utility library that can detect MIME types from files, input streams, URL's and byte arrays.
- Ehcache[48]: An open source, standards-based cache used to boost performance, offload the database and simplify scalability.
- JGAP[49]: A Genetic Algorithms and Genetic Programming component provided as a Java framework.

2.4. DEER

In this section, we outline the plan for developing and extending the DEER software in the context of the project. First, we summarize the features already available in the current version (v.0.5). Then, we present the planned additions to its functionality, as inferred by the SLIPO requirements, and discuss some specific issues and targeted actions that will guide the implementation of the next version (v.1.0) of DEER as part of the SLIPO Toolkit, providing an indicative timeline for the implementation of the above.

DEER is available under a dual license. For any non-commercial uses, the terms of the GPL3.0 license (version of June 29, 2007) hold[50]. Each commercial use is subject to fees, which are to be regulated contractually with the Semantic Data Management Research Unit of the Institute for Applied Informatics e.V.

---

[41] http://commons.apache.org/proper/commons-lang/

[42] http://site.icu-project.org/

[43] https://logging.apache.org/log4j/2.x/

[44] https://github.com/fusesource/jansi

[45] https://www.slf4j.org/

[46] https://code.google.com/archive/p/lanterna/

[47] https://mvnrepository.com/artifact/eu.medsea.mimeutil/mime-util/2.1.3

[48] http://www.ehcache.org/

[49] http://jgap.sourceforge.net/

[50] https://github.com/SLIPO-EU/DEER/blob/master/LICENSE

2.4.1. Version 0.5 Features

DEER is a data enrichment framework that applies enrichment functions and operators to discover implicit or explicit references of entities to external datasets. This way, DEER allows the enrichment of a dataset's entities from several other data sources. DEER provides facilities for manual and automatic enrichment.

DEER was designed to be a modular tool which can be easily extended and re-purposed. In its current version, DEER provides two main types of artifacts:

- DEER Modules: These artifacts generate enrichment data based on RDF data. The input for such a module is an RDF dataset (in Java, a Jena Model). The output is also an RDF dataset enriched with additional information (in Java, an enriched Jena Model).
- DEER Operators: The idea behind operators is to enable users to define a workflow for processing their input dataset. Thus, in case a user knows the type of enrichment that is to be carried out (using linking and then links for example), he can define the sequence of modules that must be used to process his dataset. Note that the format of the input and output of modules is identical. Thus, the user is empowered to create workflows of arbitrary complexity by simply connecting modules.

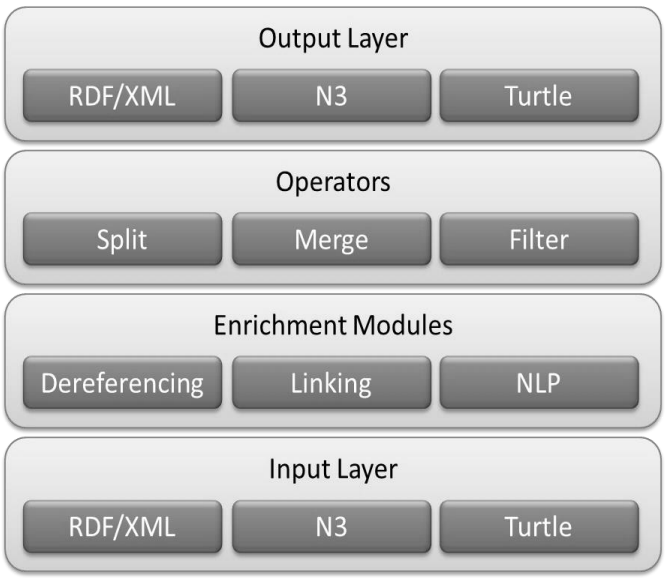

Figure 15: DEER architecture

The DEER architecture is shown in Figure 15. The input layer allows reading RDF in different serializations. The enrichment modules are in the second layer and allow adding information to RDF datasets by different means. The operators (which will be implemented in the frame of the SLIPO project) will combine the enrichment modules and allow defining a workflow for processing information. The output layer serializes the results in different format.

DEER implements several enrichment facilities that include linking, NLP, dereferencing and filtering techniques to identify and extract either new POIs that were previously implicitly represented, or new metadata for existing POIs. Further, DEER is able to learn on training examples and automatically detect and apply enrichment pipelines (workflows) on new POI sets. We will extend the DEER framework by proposing a set of novel POI-related enrichment functions (e.g., geo-locator of POIs) and operators (e.g., POI

aggregation). We will also enhance and fine-tune its automatic self-configuration procedures for POI-related metadata discovery and extraction. Since enrichment is by nature a task that focuses more on quality rather than quantity, we will emphasize on the effectiveness of the implemented, POI-specific functions and the quality assessment of the produced results.

2.4.2. Towards DEER 1.0

In this section, we describe the main contributions to DEER and provide a timeline estimation for each of them.

2.4.2.1. New Features

The new functionality provided by DEER v.1.0 will include:

- Providing non-linear enrichment pipelines. In DEER 1.0, we will deal with the enrichment of POI data sets. State-of-the-art frameworks for data enrichment such as DEER provide linear enrichment pipelines for enriching datasets. However, given the complexity of POI datasets, non-linear enrichment pipelines will have to be learned from existing data to ensure that high-quality datasets can be created by combining information from many heterogeneous sources. [Initial release: M18, Final release: M36]
- Introducing POI-specific enrichment functions. We will extend upon the kernel of DEER to integrate POI-specific enrichment functions that will allow discovering attributes of POIs from sources of various structures ranging from textual, to structured. [Initial release: M18, Final release: M36]
- Extending the automatic enrichment paradigm of DEER. EEER v.1.0 will support multiple data sets as input and output. Pro-active enrichment strategies as well as active learning will complete the extensions of enrichment towards POI-specific enrichment. [Initial release: M18, Final release: M36]
- Implementing a parallel execution engine to enable scalable execution of the complex non-linear enrichment pipelines. [Initial release: M18, Final release: M36]

2.4.2.2. Scalability

Given the increasing size of POI datasets, in DEER v.1.0, we will enhance the execution of complex non-linear enrichment pipelines by providing automatic parallelization approaches. To this end, approximation functions for the runtime and output size of atomic enrichment functions will be devised and alternative plans will be provided and integrated into a dynamic planning strategy.

2.4.3. Dependencies

DEER has dependencies to the following open-source tools and libraries:

- Google Collections[51]: A set of core libraries that includes new collection types. The library has been replaced by Google Guava[52].

[51] https://code.google.com/archive/p/google-collections/

[52] https://github.com/google/guava

- FOX[53]: A framework that integrates the Linked Data Cloud and makes use of the diversity of NLP algorithms to extract RDF triples of high accuracy out of NL.
- JUnit: A simple framework to write repeatable tests.
- LIMES-core[54]: LIMES – Link Discovery Framework for Metric Spaces.
- Apache HttpComponents[55]: A toolset of low level Java components focused on HTTP and associated protocols.
- Apache Log4j: An open-source Java logging library.
- DL-Learner[56]: A tool for supervised Machine Learning in OWL and Description Logics.
- JSON-java[57]: A reference implementation of a JSON package in Java.

2.5. OSMRec

In this section, we present OSMRec, a service for producing category recommendations for geospatial entities. First, a brief description of the current version (v.1.0) is given. Then, we present the planned additions to its functionality, as inferred by the SLIPO requirements, and discuss some specific issues and targeted actions that will guide the implementation of the next version (v.2.0) of OSMRec as part of the SLIPO Toolkit.

OSMRec is an open source software and it can be redistributed and/or modified under the terms of the GNU General Public License as published by the Free Software Foundation; either version 3.0 of the License, or (optionally) any later version.

2.5.1. Version 1.1 Features

OSMRec is a tool that trains on a set of spatial entities annotated with categories (classes) and provides category recommendations for new geospatial entities. OSMRec's goal is to exploit the richness of available geospatial datasets than contain geospatial entities already annotated with several categories (e.g., OpenStreetMap), in order to enrich new geospatial entities with category annotations.

OSMRec supports two modes of deployment: (a) a generic command line utility that allows the training and recommendation of arbitrary data, as long as they are formatted on the required input format (OSM/XML), and (b) a JOSM[58] plugin which allows the real-time recommendation of OSM categories for geospatial entities created within the JOSM user interface. JOSM, one of the most prominent authoring tools of OSM, is a graphical application that allows users to: (a) download all the geospatial entities (e.g., roads, buildings, stations, areas) of a geographic area from OSM, (b) visualize these features on a map overlay, (c) add, change or delete geospatial features, and (d) upload the changed dataset into the publicly accessible OSM map database.

[53] https://github.com/AKSW/FOX
[54] https://github.com/AKSW/LIMES-dev/tree/master/limes-core
[55] https://hc.apache.org/
[56] https://github.com/AKSW/DL-Learner
[57] https://github.com/stleary/JSON-java
[58] https://josm.openstreetmap.de/

OSMRec is implemented in Java and consists of two core modules: the Training Module and the Recommendations Module. The former runs an offline process that takes as training input an OSM dataset and produces as output a classification model. The latter component takes as input a set of geospatial entities, formatted according to the OSM XML format[59], as well as the trained model, and produces a set of categories that are the model's recommendation for annotating each geospatial entity from the input set.

To train the recommendation model, OSMRec applies multilabel SVM classification, using LIBLINEAR[60], and considering as training items the geospatial entities themselves and as labels the categories that characterize them. The method maps the training entities into a multi-dimensional feature space and aims at finding the optimal hyperplanes that discriminate the entities belonging to different categories. The training features that are used to represent each geospatial entity are the following: *Geometry type*; *Number of geometry points*; *Area of geometry*; *Mean edge length*; *Variance of edge lengths*; *Textual features*; *Category features*;

User preferences, context and perception of things play a crucial role in the process of annotating geospatial entities. For example, if a user A lives in a rural area whereas a user B lives in a city center, they are expected to insert into OSM and annotate geospatial entities of different categories. Moreover, a road that is considered as "street" in one city, might be considered as "avenue" in another. Thus, user context should be taken into account and the classification of entities and category recommendation processes should be adapted/personalized according to it.

In order to introduce the context based personalization in OSMRec, we follow a hybrid solution, where the user is able to train multiple recommendation models and combine them respectively, according to her preferences and/or setting. Specifically, OSMRec offers the following model training options:

- **Training on area**: The user is able to train a recommendation model on the geospatial entities of an area that is downloaded from OSM.
- **Training on user's editing history**: The user is able to download her editing history in OSM and train a personalized model that takes into account only the geospatial entities she has inserted or edited in OSM.
- **Training on user's editing history in an area**: This option essentially combines the two aforementioned options. It trains a model only on the user's geospatial entities that are contained in the downloaded OSM area.

[59] http://wiki.openstreetmap.org/wiki/OSM_XML
[60] http://www.csie.ntu.edu.tw/~cjlin/liblinear/

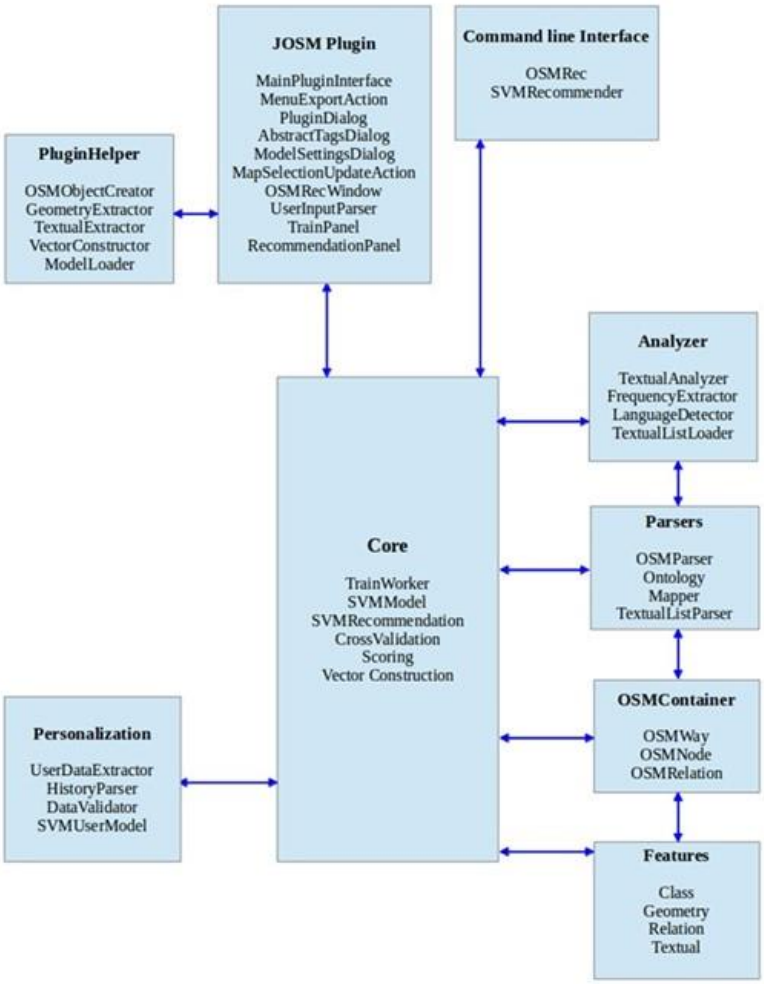

Figure 16: Architecture of OSMRec

Figure 16 depicts the architecture of the JOSM plugin version of OSMRec. It consists of eight basic modules as described next:

- **Command-line Interface Module**. It takes as input parameters specifying the type of execution of the program. Possible options are the training of a model from an OSM file, testing an already trained model, or executing both aforementioned operations.
- **Plugin Interface Module.** This module comprises the graphical user interface and is implemented as a plugin in JOSM. It provides all necessary functionality for interacting with the map and prepares the OSM data to be used by the core of OSMRec.
- **OSMContainer Module**. This module is responsible for keeping all properties and metadata (IDs, timestamps, user ids, geometries, etc.) about all the instances parsed from the OSM file.
- **Parsers Module**. The main objective of this module is to parse the provided OSM file and extract all the necessary data, concerning nodes and ways.
- **Core Module**. This module implements the construction of all the necessary input formats for the classification process. It prepares the data by constructing appropriate vectors for each instance of

the OSM file. The produced data structures are used within LIBLINEAR to carry out the training and testing of the model.

- **Features Module.** This module is responsible for the feature construction functionality of the tool. It is implemented in a modular way to allow feature selection for the vectors that will be constructed.
- **Scoring Module**. The scoring module implements a mechanism for computing scores for different configurations of the execution in order to decide the optimal parameters for the training process.
- **Analyzer Module.** The analyzer is a module that helps the configuration of the textual features construction.

2.5.2. Towards OSMRec 2.0

OSMRec is currently implemented in a flexible way that allows the deployment of the tool both as a command line utility and as JOSM plugin. It allows training of personalized recommendation models, based on specific geographic areas and/or user annotation history. Currently OSMRec is tied to OSM's data representation format (OSM – XML) and optimized/evaluated on geospatial entities annotated with the OSM categories hierarchy. However, it implements generic geospatial training features that can be extracted by geospatial entities (POIs) in most existing datasets. Further, it utilizes a robust learning library that can efficiently train classification models and recommend categories for new POI entities. The major functionality of OSMRec within SLIPO will be the automatic enrichment of POI entities with categories from several supported category ontologies/vocabularies. Next, we present the new features that will be implemented, as well as performance improvements to satisfy our scalability goals.

2.5.2.1. New Features

The new functionality provided by OSMRec v.2.0 will include:

- Generalization of the tool's application domain. One of our major goals will be to uncouple OSMRec from its dependence to OSM dataset and category hierarchy. We will generalize the data input modules and the training mechanisms so that they can take as input several POI datasets, as long as they comply with at least some basic concepts and properties of the SLIPO POI ontology. Further, we will extend the tool so that it is able to support several POI category ontologies/vocabularies, and use them to annotate POI entities. [Initial release: M18, Final release: M36]
- Specialization on POI data. The current version of OSMRec comprises a utility for recommending categories for *generic* geospatial entities (buildings, places, areas, roads, etc.). Within SLIPO, we will utilize the available POI datasets in order to fine tune the underlying machine learning models, specifically for POI entities. To this end, we will: (i) examine the definition of additional training features that improve the accuracy of the models when applied on POI entities) and (ii) parameterize and fine tune the models by training and evaluating them on a variety of available POI datasets. [Initial release: M18, Final release: M36]
- Utilization in POI quality assessment. OSMRec's functionality will be further adopted in order to be utilized in the quality assessment processes implemented within SLIPO. Specifically, the category recommendation functionality of OSMRec will be included into a category cross-checking

mechanism, for producing quality scores regarding the confidence of the categories POI entities are already annotated with. The mechanism will function as an auxiliary measure for identifying erroneous categories (quality assessment) or resolving conflicting category values (fusion). [Final release: M36]

2.5.2.2. Scalability

As mentioned above, OSMRec comprises two main modules: the Training and the Recommendation module. Regarding the former, scalability improvements are not applicable nor required: training an accurate recommendation model mostly requires selecting a proper, representative training dataset, rather than training on massive volumes of data. That is, in our setting, the accuracy of the trained models depends mainly on locality and context: if a stakeholder needs to annotate POIs of a specific city, and use these annotations for a specific business case, then they need to select a limited dataset, corresponding to the same city and containing POIs that are related-useful to the specific business case, in order to train their category recommendation model. Selecting, instead, for example, the whole Europe as training dataset, would only hurt the accuracy of category recommendation.

For the latter functionality, i.e., for the recommendation of categories for POI entities, parallelization of the category recommendation process is straightforward: each trained recommendation model occupies marginal space, so it can be distributed without significant cost to arbitrary processing nodes. Further, category recommendations are produced for each POI entity based solely on information of the said entity. Thus, POI entities may be partitioned in an arbitrary way to distributed processing nodes for parallelized processing. This allows us the agility to select among several approaches and frameworks: from adopting well-established frameworks (Hadoop, Flink, SANSA) to implementing a lightweight and resource-efficient, ad hoc partitioning mechanism. Our final choice will be dictated considering the targets of maximizing scalability and facilitating integration with the SLIPO Workbench.

2.5.3. Dependencies

TripleGeo has dependencies to several open-source tools and libraries:

- LIBLINEAR (Java version) is a linear classifier for data with millions of instances and features and a library for solving large-scale regularized linear classification and regression.
- GeoTools. GeoTools is an open source (LGPL) Java library, which provides standards compliant methods for geospatial data management comparable to those implemented in Geographical Information Systems (GIS).
- Java Topology Suite (JTS). JTS is an API of 2D geospatial predicates and functions, conforming to the OGC Simple Features Specification for SQL.
- Apache Lucene[61]. Apache Lucene is a high-performance, full-featured text search engine library written entirely in Java.
- Language-detection. It is a language detection library implemented in plain Java that offers over 99% detection precision for 53 languages.

[61] http://lucene.apache.org/

2.6. FAGI

In this section, we present FAGI, a platform that implements fusion of geospatial Linked Data. First, a brief description of the current version of FAGI (v1.1.0-beta) is given and then the roadmap of development and extension of FAGI in the frame of project (v.2.0) is outlined. We discuss the specific tasks that FAGI aims to handle, the features and functionality the tool will offer until the end of SLIPO project, and an indicative timeline for the implementation of the above.

FAGI is an open source software and is available from in GitHub[62]. It can be redistributed and/or modified under the terms of the GNU General Public License as published by the Free Software Foundation; either version 3.0 of the License, or (optionally) any later version.

2.6.1. Version 1.1.0-beta Features

FAGI is a tool that allows the fusion of interlinked geospatial RDF entities. It is designed to retrieve data through SPARQL endpoints, thus, it takes RDF graphs stored in an RDF store as input. This allows for FAGI to operate on several RDF datasets, as long as they are served by a SPARQL endpoint. It allows the fusion of both thematic and spatial properties between pairs of linked geospatial entities, so that a single geospatial entity is produced for each pair, that carries the most complete, correct and timely properties of the two initial entities.

FAGI works as follows: The user provides the tool with two source datasets and a list of Linked entities between them, either as files or through available SPARQL endpoints. The tool analyzes the datasets, discovering how geometric information is stored along with their accompanied metadata. The interface allows the user to identify mappings between properties of the Linked entities. Given the above mappings, a series of fusion actions are supported, per pair of mapped properties, regarding both spatial and thematic properties of the Linked entities. The available fusion actions are enumerated in Table 1.

Action	Type	Description
Concatenation	Geometric	Create a GEOMETRYCOLLECTION Well-Known-Text from the two geometries
ShiftAtoB/BtoA	Geometric	Shift one geometry in the direction of the other' s center
Keep A/B	Geometric	Keep only the geometry of one of the datasets
Keep most points	Geometric	Keep most complicated geometry
Keep Both	Geometric	Keep both geometries separately
Keep Concatenated A/B	Metadata	Keep a concatenation of the values corresponding to each property in one single property
Concatenation	Metadata	Perform the above action on both properties and keep a concatenation of the results
Keep Concatenated Both	Metadata	Perform the above action on both properties and keep both results

[62] https://github.com/SLIPO-EU/FAGI

Keep Flattened A/B	Metadata	With properties that are part of a long RDF triple chain this action allows for creating triples that move the values of the selected property one depth level less in the chain.
Keep Flattened Both	Metadata	Perform the above action on both properties and keep both results
Keep A/B	Metadata	Keep only the value from one of the datasets
Keep Both	Metadata	Keep both separate values

Table 1: FAGI available fusion actions

FAGI supports a series of advanced features, including batch fusion actions functionality, Linked entities clustering and link identification within neighborhoods of geospatial entities.

FAGI consists of two components, namely, FAGI-cli and FAGI-service. The former offers a command line interface for basic fusion functionality, while the latter provides the full-fledged interactive user interface and functionality for advanced previewing and fusion actions on geospatial entities and their metadata.

2.6.2. Towards FAGI 2.0

In this section, we describe how FAGI will be extended in the context of the SLIPO project to efficiently and effectively handle the fusion of interlinked POI entities. The extensions will be towards the following major directions: specification of the vocabularies, metrics and actions and rules for the fusion of POI entities; Incorporation of quality metrics, functions and indicators within the fusion process; semi-automation of the fusion process; parallelization of processing to satisfy the scalability requirement of fusing millions of Linked POIs. Next, the planned extensions and new functionality of FAGI v2.0 are presented in detail.

2.6.2.1. New Features

The new functionality provided by FAGI v.2.0 will include:

- Specialization on POI data. We will explore the available datasets and schemas, to extract patterns relating to naming of POIs, frequent property types, existence of potential temporal or quality metadata, etc. In parallel, we will consider the work done in WP2 regarding the SLIPO ontology, vocabularies and mappings to individual, existing POI schemas. Our main goal will be to identify peculiarities and specific characteristics of POI data, to be used in the development of (manual and automatic) fusion action functionality. Finally, POI-specific quality indicators and quality-based rules will be identified and implemented. [Initial release: M18, Final release: M36]
- Quality and evolution based fusion. We will develop rules and fusion actions criteria based on quality characteristics and provenance information of the datasets, as well as based on the evolution of the Linked POI entities. These will include metrics used to compare the similarity of the metadata between two Linked POIs and to measure and compare the quality of those metadata. Further, quality assessment and dataset comparison metrics will be integrated, not only to validate the fusion quality, but also to assist and guide the fusion process, e.g., by considering dataset quality indicators in the fusion action recommendation. Finally, we will incorporate processes for including provenance and evolution of POI entities and metadata during the fusion process. The provenance will be used as one of the indicators for deciding which metadata to maintain and which to discard. On the other hand, we will implement new, evolution-specific fusion actions for handling metadata

that change over time and through different, interlinked datasets. [Initial release: M18, Final release: M36]

- Integration with the SLIPO Workbench. Two basic deployment scenarios of FAGI will be supported: (a) (almost fully) automated deployment where human interaction is limited to the specification of a deployment configuration; (b) a more interactive version, where the user can perform manual fusion actions, train the offline learning mechanism, provide input in the active learning process and view/validate/rectify the produced fused POIs. For the latter, we will extend and adapt the existing map-based user interface, so that it supports several facilities, such as: visualization of Linked POI entities on the map; full demonstration of the properties of a selected POI in an elegant way; selecting/validating/rectifying fusion actions per properties. [Initial release: M18, Final release: M36]
- Development of learning mechanisms for the automation of fusion action selection. There will be three learning scenarios that we will consider: (a) offline training of fusion actions based on an initial set of fused POIs; (b) active learning of fusion actions where the system progressively trains, suggests fusion actions and re-trains based on user validation; (c) offline re-training where the system has produced a set of fused results, the user examines and validates/rejects each of them and then the system uses this feedback to "correct" its model. [Final release: M36]

2.6.2.2. Scalability

- Until the current version of FAGI, we have not implemented any parallel processing techniques. However, parallelization of the fusion process is straightforward, since each fusion action is performed individually in very small, autonomous fragments of the data (pairs of Linked entities along with all properties attached to them). Each pair of Linked entities, with all their properties, is included in only one processing node, with no impact on fusion, as each pair of entities is handled independently from the rest. Thus, we can apply the desired fusion actions separately in each node and combine the final results in a complete fused graph/RDF file. This allows us the agility to select among several approaches and frameworks for parallelized processing: from adopting well-established frameworks (Hadoop, Flink, SANSA) to implementing a lightweight and resource-efficient, ad hoc partitioning mechanism. Our final choice will be dictated considering the targets of maximizing scalability and facilitating integration with the SLIPO Workbench.
- In parallel, several performance-oriented enhancements will be implemented, that will further contribute to the scalability increase of FAGI v2.0: (i) We will restructure the backend of FAGI, so that it gathers all basic fusion functionality, providing a common API to both the automatic, batch version and the user interactive, map-based version of the software; (ii) We will minimize spatial processing and storage dependencies, uncoupling the current version of FAGI from unnecessary frameworks and libraries, as well as removing intermediate processing steps and maximizing the in-memory processing functionality.

2.6.3. Dependencies

FAGI has dependencies to several open-source tools, libraries and services:

- Virtuoso[63]: An RDF data store and SPARQL endpoint for remote graphs. This dependency may be removed in the future. Instead, FAGI will be extended to handle files as input/output, instead of graphs.
- Postgres/PostGIS [64] : For efficiently handling geospatial queries and transformations. This dependency may be replaced by a more lightweight library.
- Apache Jena: A Java framework for building Semantic Web applications.
- Google Guava: A set of libraries that includes collection types (such as multimap and multiset), immutable collections, a graph library, functional types, an in-memory cache, and APIs/utilities for concurrency, I/O, hashing, primitives, reflection and string processing.
- Java WordNet Library: A Java library for interfacing with Wordnet[65].
- WordNet: A large lexical database of English in which nouns, verbs, adjectives and adverbs are grouped into sets of cognitive synonyms, each expressing a distinct concept.
- LIBLINEAR: A linear classifier for data with millions of instances and features.
- GeoTools: A Java library that provides OGC standards compliant methods for geospatial data management.
- Java Topology Suite: An API of 2D spatial predicates and functions conforming to the OGC Simple Features Specification for SQL.
- Weka[66]: A Java library that provides a collection of machine learning algorithms for data mining tasks.

[63] https://virtuoso.openlinksw.com/

[64] http://postgis.net/

[65] http://wordnet.princeton.edu/

[66] http://www.cs.waikato.ac.nz/ml/weka/

3. Tool Stack

3.1. Overview

In this section, we present the third-party libraries, frameworks and tools that will be used for implementing the software components of the SLIPO system and we introduce the software systems and applications that will be deployed and combined in the SLIPO architecture. First, we provide a description of the systems and processing frameworks for storing and analyzing POI data. Next, we present the frameworks and libraries required for server-side development. Finally, we enumerate the libraries and frameworks which will be used in the development of SLIPO's front-end.

3.2. Systems and Applications

In Section 1.1, we have presented the SLIPO Toolkit, a collection of independent and scalable software components for POI data integration. The SLIPO system orchestrates these components to create data integration workflows and stores input and output files to a distributed file system. Moreover, a spatially enabled relational database is deployed for indexing and querying small POI datasets including sample data generated by the SLIPO Toolkit components. Finally, the Apache Flink distributed processing framework is installed since it is required by SANSA distributed processing framework. In the next sections, we provide a brief introduction to these systems.

3.2.1. Hadoop Distributed File System

Hadoop Distributed File System[67] (HDFS) is a highly available and scalable distributed file system, designed to run on commodity hardware and is an integral component of the Hadoop ecosystem. HDFS splits files in blocks which are replicated across a set of servers designated as DataNodes. A single server in the cluster, namely the NameNode, is responsible for managing the file system namespace (directory structure), coordinating file replication and maintaining metadata about the replicated blocks. Every time a modification is made (e.g., a file or directory is created or updated), the NameNode creates a log entry and updates metadata. Clients contact the NameNode only for accessing file metadata and perform I/O operations directly on the DataNodes. A high-level overview of the HDFS architecture is depicted in Figure 17.

[67] https://hadoop.apache.org/docs/current/

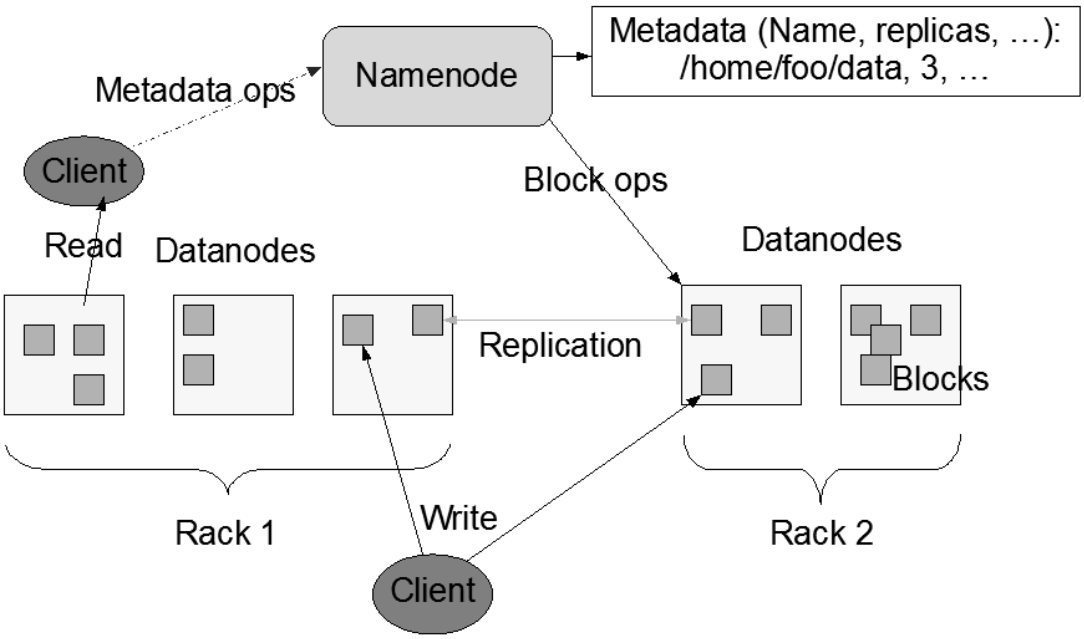

Figure 17: Overview of the HDFS architecture

To increase availability, the NameNode maintains multiple copies of the metadata and log files. Moreover, an optional secondary NameNode can be deployed for creating checkpoints for the metadata and log files. Creating checkpoints allows for faster recovery times of the NameNode after a failure. In addition, HDFS can be configured to use multiple independent NameNodes, thus implementing many autonomous file system namespaces. The latter feature increases I/O operation throughput and offers isolation between different applications. Finally, HDFS is optimized for managing very large files, delivering a high throughput of data using a write-once, read-many-times pattern. Hence, it is inefficient for handling random reads over numerous small files or for applications that require low latency.

In early versions of HDFS, the NameNode used to be a single point of failure in a HDFS cluster. Newer versions support deploying two NameNodes in an Active/Passive configuration. At any given time, a single node is active, while the second node is kept synchronized by watching over any changes applied by the active node. Thus, the file system can be recovered even faster after a failure.

3.2.2. PostgreSQL and PostGIS

PostGIS[68] is an extension to the PostgreSQL object-relational database system which allows GIS objects to be stored in the database. The PostGIS 2 series provides a rich feature set for storing and processing both vector and raster data. In the scope of the SLIPO project we are primarily interested in vector data. Next, we enumerate several of the most important and most commonly used PostGIS features.

- Processing and analytic functions for both vector and raster data for splicing, dicing, morphing, reclassifying with the power of SQL.
- Spatial re-projection SQL callable functions for both vector and raster data.

[68] http://postgis.net/

- Support for importing/exporting ESRI shapefile vector data via both command line and GUI packaged tools and support for more formats via other third-party Open Source tools.
- Packaged command-line for importing raster data from many standard formats including GeoTIFF, NetCDF, PNG and JPG.
- Rendering and importing vector data support functions for standard textual formats such as KML, GML, GeoJSON, GeoHash and WKT using SQL
- Rendering raster data in various standard formats including GeoTIFF, PNG, JPG and NetCDF.
- Seamless raster/vector SQL callable functions for extrusion of pixel values by geometric region, running stats by region, clipping raster by a geometry, and vectorizing raster.
- 3D object support, spatial index, and functions.
- Network Topology support.

PostGIS follows the Open Geospatial Consortium's "Simple Features for SQL Specification" and has been certified as compliant with the "Types and Functions" profile. PostGIS is open source software, released under the GNU General Public License.

3.2.3. Apache YARN

Apache YARN[69] is a resource management framework that allows the separation of job scheduling and resource management from the underlying programming model implementation. YARN is agnostic to the applications that request resources and allows Hadoop to be used for implementing other computational models apart from MapReduce such as graph processing, machine learning etc.

YARN uses a single ResourceManager for managing collectively the resources of the whole cluster and one NodeManager per server for managing the server's local resources. The resources on each server are organized in leases (containers) that represent CPU, memory, disk space, network bandwidth etc. The ResourceManager has two main components, namely, the Scheduler and the ApplicationsManager.

The ApplicationsManager is responsible for accepting job requests, initializing job execution and monitoring their status in case a restart is required when a failure occurs. The Scheduler is responsible for allocating resource containers to running applications (jobs) and enforcing constraints on the resources utilization. The policy for sharing resources among various applications is pluggable and extensible. An example of such a pluggable scheduler is the FairScheduler that assigns resources to applications such that all applications get, on average, an equal share of resources over time.

The ResourceManager is ignorant of the semantics of each resource allocation requested by an application. For each application (job) running on the cluster, an ApplicationMaster is assigned. ApplicationMaster is a framework specific library that implements a programming model (e.g., Hadoop MapReduce, Apache Flink or Spring Batch) and is responsible for negotiating appropriate resource containers from the Scheduler, generating a physical execution plan, monitoring and tracking execution progress and handling execution errors. Before starting a new job, a container must be acquired from the ResourceManager for launching the ApplicationMaster itself. Afterwards, the ApplicationMaster negotiates resources from the

[69] https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

ResourceManager and once sufficient resources are acquired, it works with the one or more instances of NodeManager for executing its jobs.

3.2.4. Apache Flink

Apache Flink[70] is a distributed data processing framework capable of analyzing large datasets. Instead of building its functionality on top of the existing Apache Hadoop MapReduce framework, Flink implements its own job execution runtime. Therefore, it can be used as an alternative to Hadoop MapReduce component for processing big data. When used with Hadoop, Flink can access data stored in HDFS and request cluster resources from the YARN resource manager.

Flink extends the MapReduce programming model with additional operators, also called transformations. An operator consists of two components, a user-defined function (UDF) and a parallel operator function. The operator function parallelizes the execution of the user-defined function and applies the UDF on its input data. The data model used by Flink operators is record based instead of the key-value pair model used by MapReduce. Still, key-value pairs can be mapped to records. A collection of records is referred to as dataset. All operators will start working in memory and gracefully fallback to external memory algorithms once memory resources become low. The new operators represent many common data analysis tasks more naturally and efficiently.

A short description of the available operators as presented in the Flink documentation follows.

- **Map**: The Map transformation applies a user-defined function on each element of the input dataset. It implements a one-to-one mapping, that is, exactly one element must be returned by the function. This behavior differs from that of the classic Map operator.
- **FlatMap**: The FlatMap transformation applies a user-defined function on each element of the input dataset. This variant of a map function can return arbitrary many result elements (including none) for each input element. The behavior of this operator matches that of the classic Map operator.
- **Filter**: The Filter transformation applies a user-defined function on each element of the input dataset and retains only those elements for which the function returns true.
- **Project**: The Project operator allows the modification of the fields of a tuple. A tuple is an ordered list of fields. Users can shuffle, add or remove fields. Project operator does not need the definition of a user-defined function.
- **Reduce** on grouped dataset: A Reduce operator that, when applied on a grouped dataset, reduces each group to a single element. For each group of input elements, the user-defined function successively combines pairs of elements into one element until only a single element for each group remains. There are different variations for this operator. For example, users may define an additional function for extracting the key used for grouping from each element.
- **GroupReduce** on grouped dataset: A transformation that, when applied on a grouped dataset, calls a user-defined function for each group. The difference between GroupReduce and Reduce is that the user defined function gets the whole group at once instead of a pair of elements at a time. The

[70] https://flink.apache.org/

function is invoked with an iterator over all elements of a group and can return an arbitrary number of result elements. This operator resembles the classic Reduce operator.

- **Reduce** on full dataset: Applies a user-defined function to all elements of the dataset. Pairs of elements are subsequently combined into one element until only a single element remains.
- **GroupReduce** on full dataset: Applies a user defined function to a dataset by iterating over all the elements of the dataset. An arbitrary number of result elements is returned.
- **Aggregate** on grouped tuple dataset: Supports min, max and sum aggregation operations. The aggregate transformation can only be applied on a dataset of tuples.
- **Join**: Joins two datasets into one dataset. The elements of both datasets are joined on one or more keys which can be specified either by a user defined function or by field indexes, if elements are tuples.
- **Cross**: The Cross transformation combines two datasets into one dataset by building a Cartesian product. The Cross transformation either calls a user defined function on each pair of elements or applies a projection
- **CoGroup**: The CoGroup transformation jointly processes groups of two datasets. Both datasets are grouped on a defined key and groups of both datasets that share the same key are passed together to a user-defined function which iterates over the elements of both groups. If for a specific key of any dataset there are no matching elements from the other dataset, an empty group is passed to the user function.
- **Union**: Produces the union of two datasets, which must be of the same type.

Flink allows to model job processing as directed acyclic graphs (DAGs) of operations, which is a more flexible model than MapReduce, in which Map operations are strictly followed by Reduce operations. The combination of various operations allows for data pipelining and in-memory data transfer optimizations, which increase performance by drastically reducing disk access and network traffic. Moreover, Flink supports highly efficient iterative algorithms, which are very important for Data Mining, Machine Learning and Graph processing, since such jobs often require looping over the working data multiple times. Implementing such jobs with frameworks like MapReduce is quite expensive since data is transferred between iterations by using the distributed storage. In contrast, Flink supports iterative algorithms in its core.

Flink offers powerful APIs in Java and Scala. The Flink optimizer compiles the user programs into efficient, parallel data flows which are executed on a cluster or a local server. The optimizer is independent of the actual programming interface and supports cost-based optimization for selecting operator algorithms and data transfer strategies, in-memory pipelining of operators, data storage access reduction and sampling for determining cardinalities.

3.3. Server Development

In this section, we enumerate various server-side development frameworks and libraries. The evaluation of a framework usually depends on multiple criteria such as productivity and ease of use, framework complexity and learning curve slope, available documentation and support, security, testability, performance, easiness of maintenance, framework ecosystem, etc. While all frameworks aim at delivering

robust applications, each of them favors different qualities like testing, rapid application development, or simplicity over others. Moreover, applications usually prioritize features like security and scalability higher than others. For SLIPO applications and services, scalability, security and testability are the primary required qualities.

3.3.1. Spring

Spring[71] is one of the most popular and mature application development frameworks for Java, featuring a vast ecosystem of projects for developing applications from the mobile to the enterprise and cloud. Spring's vertical tool stack handles almost any programmatic task like security, data access, transaction management, social service provider integration, etc. Yet, its modular architecture allows using only the features required by the solution being developed. Despite its complex architecture, Spring's extensive documentation and supreme extensibility features allow developers to easily configure Spring to their needs. In the next sections, we enumerate the most important Spring modules that will be used during the project development.

3.3.1.1. Spring Framework

The Spring Framework consists of several core modules that offer the basic building blocks for developing any kind of application. The features provided by the Spring Framework include:

- Inversion of Control (IoC) and Dependency Injection (DI) features for configuring the creation and managing the lifecycle of objects.
- Aspect-Oriented Programming (AOP) features for cleanly decoupling shared functionality such as transaction management and logging.
- Data Access features, including integration with object relation mapping APIs such as Java Persistence API (JPA) or Hibernate. Moreover, Spring Framework allows the combination of these APIs with features such as declarative transaction management using AOP as mentioned earlier.
- Spring MVC web application and RESTful Web Service framework for easily building web application and services.

3.3.1.2. Spring Batch

Spring Batch provides methods for configuring and executing jobs for processing large volumes of data. Jobs are organized in one or more processing units named steps. Data processing can be row based, e.g., repeatedly processing chunks of rows of a database table, or task based, e.g., copying a set of files. Step execution flow can be controlled declaratively or programmatically. Steps can be executed sequentially, in parallel or even conditionally, which may result in specific steps not being processed at all. Spring Batch provides the infrastructure for launching and monitoring job execution and offers many features including logging/tracing, transaction management, job processing statistics, job restart, skip, and resource management.

[71] https://spring.io/

3.3.1.3. Spring Security

Spring Security provides the tools for implementing authentication and authorization in Java applications. For authentication, Spring Security supports several methods such as HTTP Basic Authentication, Form Based Authentication and OAuth 2.0 authentication for establishing and managing the application principal, i.e., the user who can use the application. Once the application principal is set, role based security is used for controlling authorization. In addition to authentication and authorization, Spring Security offers many advanced features for web applications such as protection against attacks like session fixation, clickjacking, cross site request forgery, etc.

3.3.1.4. Spring Boot

When developing an application using the Spring Framework many configuration options must be set either using external configuration files or programmatically. Spring Boot takes an opinionated view of the Spring platform by promoting convention over configuration and selecting sensible default values for most configuration settings. Thus, an application requires minimum configuration. At the same time, whenever the default values are not appropriate to the requirements of the application, they can easily be replaced with custom configuration options.

3.3.1.5. Spring Hadoop

Spring Hadoop simplifies the development of Hadoop solutions by providing a unified configuration model and simple APIs for accessing applications and services from the Hadoop ecosystem such as the HDFS file system, YARN resource manager and HBase NoSQL database. Moreover, it supports integration of other Spring modules, such as Spring Batch, with Hadoop allowing the development of applications that have access to all Spring features and can be executed either as standalone applications or easily deployed to a YARN cluster.

3.3.2. Apache Log4j2

Every application requires some sort of logging to support tracing and debugging. Apache Log4j2 is a feature-rich and extensible logging framework that allows fine-grained message logging to several destinations e.g., console output, file system, remote servers and relational databases. Apache Log4j2 can be configured programmatically or by using external configuration files. In the latter case, there is the ability to automatically detect changes and reconfigure logging policies during runtime. Moreover, during the reconfiguration process, no logging events are missed.

3.3.3. Hibernate

Hibernate is one of the most popular Java Object/Relational Mapping (ORM) frameworks that allows users to easily store and query data in relational databases. In addition to its own proprietary Java API, it also conforms to the Java Persistence API (JPA) specification, thus allowing seamless integration to Java EE application servers. Hibernate uses several techniques such as lazy loading and optimistic locking to achieve high performance. Moreover, when combined with Spring and Spring Boot, it allows the dynamic implementation of repositories by simply declaring repository interfaces.

3.3.4. JTS Topology Suite

JTS Topology Suite implements a set of spatial operations over two-dimensional spatial predicates such as intersection, overlapping, distance etc. The goal of the project is to be used for developing applications that manipulate and query spatial datasets. JTS attempts to implement as accurately as possible the OpenGIS Simple Feature Specification (SFS). Wherever SFS specification is unclear, JTS implementation attempts to use a reasonable and consistent alternative. Details about differences against SFS can be found at the official documentation.

3.3.5. GeoTools

GeoTools is an open source Java library that provides tools for managing geospatial data. Internally it uses JTS Topology Suite for handling geometry instances. Moreover, it provides APIs for accessing spatial data in several file formats and spatial databases, transforming data between different coordinate reference systems and filtering data using spatial and non-spatial attributes. GeoTools is implemented in accordance to the Open Geospatial Consortium (OGC) standards.

3.3.6. JSch – Java Secure Channel

JSch is a Java implementation of SSH2. It provides support for secure remote login, secure file transfer, and secure TCP/IP and X11 forwarding. It can automatically encrypt, authenticate, and compress transmitted data.

3.3.7. Flyway

Flyway is a database schema migration tool that favors convention over configuration. Database schema migration is implemented either by using SQL scripts written in the target database specific syntax, or using Java code. It supports multiple database vendors and can be invoked either as a standalone tool from the command-line or using the Java API from inside the application.

3.4. Web Development

In this section, we present JavaScript and CSS libraries and frameworks that accelerate the development of SPA web applications by streamlining the management of the application state, data flow and view rendering.

3.4.1. React

React is a JavaScript framework for building interactive User Interfaces. React can be thought as the View in the MVC pattern that allows building reusable UI components and promotes composition of existing ones. Each component maintains its internal state which controls the rendering process. It is also possible to create stateless components that inherit state information from parent components using properties, resulting in pure presentational components. Whenever state (or a property) changes, only the parts of the DOM that are affected are updated. This is achieved by using a virtual representation of the DOM that efficiently detects changes to the actual DOM. The latter feature makes React interoperability with other UI

libraries more challenging. React adopts a declarative model for creating views with the help of JSX, an XML-like syntax with a smooth learning curve for developers who are familiar with HTML5 and CSS3.

3.4.2. Redux

Redux is a predictable state container for JavaScript applications that is very popular for handling the increased application logic complexity that arises in Single Page Applications. In such applications, the state management becomes increasingly harder since various user interactions – which quite often involve asynchronous requests – result in state changes. Redux attempts to manage state in a predictable way by imposing specific restrictions on how and when state updates can occur. Redux makes a perfect match to React by deferring component state management to Redux. It was based on the principle ideas of Flux for making the flow of an application unidirectional. The main difference Redux introduced is the core idea of keeping the state in a single store (following a Single source of Truth principle), instead of multiple stores. The single application state maps at any moment to its view representation via React UI components. User actions such as clicks may dispatch actions that change the state in a predefined way with the help of reducers that dictate how a specific action modifies the application state. In this manner, a one-way flow is achieved, making the application easy to reason about, debug and scale. The Redux abstract application flow in comparison to the one of MVC is shown in Figure 18.

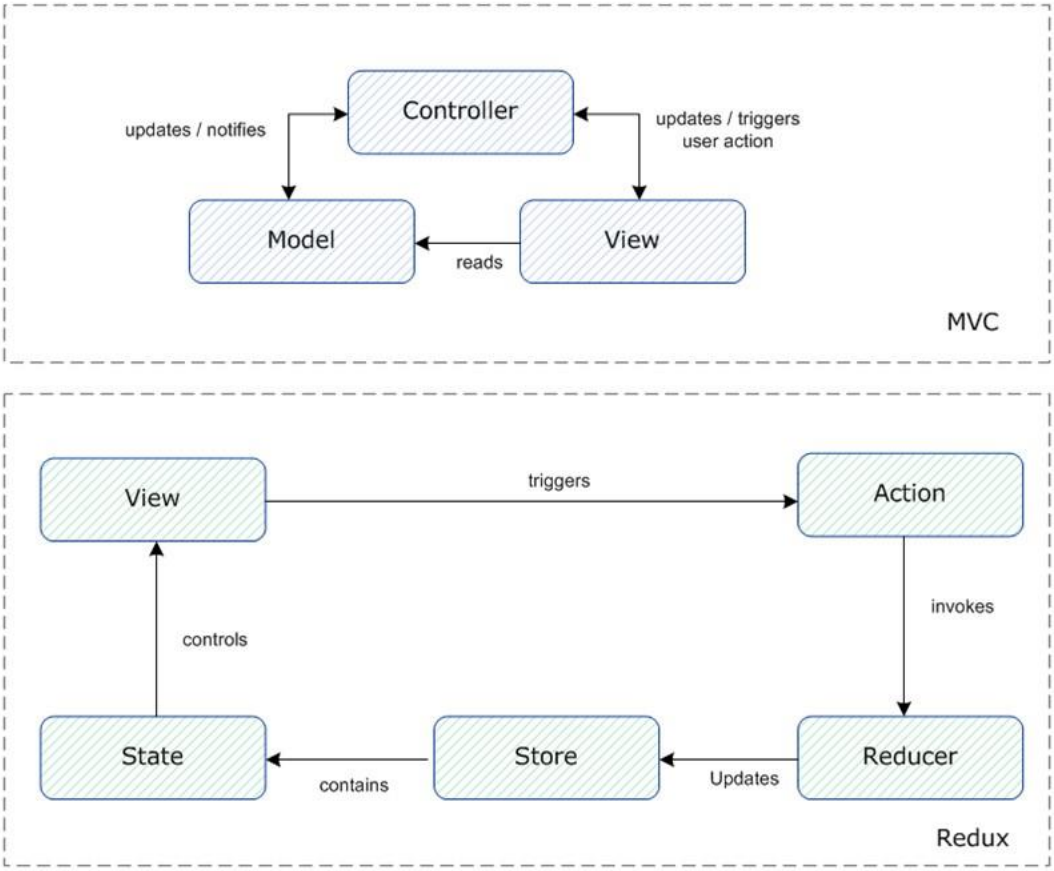

Figure 18: MVC and Redux abstract application flow

The main components of the Redux pattern are enumerated next.

- **Store**: In Redux the store holds the entire application state, which is the representation of the application at any given time. It is an object containing any number of valid JS data types, such as numbers, strings, boolean values, arrays, or other objects. A key concept is that the state object cannot be mutated directly, but only by emitting actions.
- **Actions**: Actions can be dispatched by user interactions or other actions and cause the state to change. There are two types of actions, simple and complex actions that execute with the help of a special middleware. Simple actions are plain objects containing the unique action type and any data that needs to be passed to the store. Complex actions are functions that get access to the state and can perform asynchronous operations (such as fetching data from a remote service) and/or orchestrate multiple simple action dispatches.
- **Reducers**: Reducers are pure functions that determine how an action modifies the state. Multiple reducers can be combined, each responsible for mutating a specific part of the state. Reducers do not mutate the state, but instead return a new state object, which allows easy recognition of any changes so that the view can be updated.
- **Views**: Redux combines very well with React as its view layer with the help of react-redux[72] library. In react-redux terminology, components are divided into two types: smart components or containers and pure or presentational components. Containers are aware of redux and map parts of the state and action callback functions to react component properties, and are responsible for causing the components to re-render any time a mapped property has changed. On the other hand, presentational components are just pure functions of their input properties, completely ignorant of redux, allowing successful separation of logic and templating.

3.4.3. React-Router

React Router is a JavaScript library that allows an application implemented using React and Redux to keep the application state in sync with routing information. This feature is achieved by automatically storing additional data about the current URL inside the state. This information is then propagated to React which can in turn suitably change the component tree rendering process. If there is no need for syncing routing information and application state, a simpler implementation can be obtained by using the React Router library. The latter provides support for keeping only the UI in sync with the URL.

3.4.4. React-Bootstrap

React-Bootstrap is a library of reusable UI components for the React framework. It offers the look-and-feel of the Twitter Bootstrap library using the React syntax, but has no dependencies on any third-party libraries like jQuery. React-Bootstrap offers a comprehensive list of UI components such as buttons, menus, form input controls, modal dialog, to name a few. All components can be used as provided or customized using CSS.

[72] https://github.com/reactjs/react-redux