LimeDS and the TraPIST Project: A Case Study
An OSGi-based Ontology-enabled Framework Targeted at Developers in Need of an Agile Solution for Building REST/JSON-based Server Applications

Stijn Verstichel, Wannes Kerckhove, Thomas Dupont, Bruno Volckaert, Femke Ongenae, Filip De Turck and Piet Demeester

Department of Information Technology (INTEC), Ghent University – iMinds, G. Crommenlaan 8/201, 9050, Gent, Belgium

Keywords: LimeDS, TraPIST, REST/JSON, OSGi, Semantics, Reasoning, Transportation.

Abstract: Real-Time Travel Information (RTTI) for rail commuters is still used inefficiently today and is rarely combined with other knowledge to come to a truly personalised and situation-aware multimodal travelling assistance. It is up to the travellers themselves to look for important info about their trip through static schedules or dedicated non-personalised applications. In a highly dynamic context such as that of public transportation, it would make life easier if one was able to consult the right information at the right time (removing superfluous information), for a variety of multimodal public transportation options, taking into account the context of the person travelling. In this paper we present the LimeDS framework, allowing application developers to rapidly define data workflows from a variety of data sources, deploy these workflows in a scalable and resilient manner and expose results to client applications as REST endpoints. A Proof-of-Concept (PoC) shows how our proposed framework can be used to tie together different open transportation data sources in order to create highly dynamic multimodal travel assistance applications by semantically enriching the data into knowledge, checking for ontological consistency and reason over the resulting knowledge.

1 INTRODUCTION

The current generation of applications based on Real-Time Train Information (RTTI) is still not fully using (open) data to facilitate rail travel. They are often based on only one source, even though other sources can be relevant for the passenger's journey (e.g. transportation vehicle load sensor data, passenger data, weather forecast, tourist events and promotions, etc.).

The high-level goal of the *Train Passenger Interfaces for Smart Travel (TraPIST)* (iMinds VZW, 2014) project is to offer public transportation travellers relevant information pro-actively, at the right time and through the most appropriate channel. The development of a framework that collects, analyses, classifies and filters data from various sources, based on a dynamic traveller profile, is at the heart of the project. Based on this info, specific traveller applications are developed giving passengers direct access to the information they need. The moment when the information is presented is triggered by the situation, such as a specific time, a location, an event or the travellers own circumstances (e.g. present company, destination, activities).

It quickly became apparent that the development of these situation-aware knowledge consolidation applications introduced a lot of common overhead (consulting multiple open data sources, processing their results, dealing with sudden unavailability of sources, etc.) and as such, the decision was made to develop a generic framework for building REST/JSON-based ontology-enabled applications, named a Lightweight modular environment for Data-oriented Services (LimeDS) (IBCN, 2014). Some of the main properties of LimeDS are:

- **Rapid & User-friendly HTTP Web Exposure:** through the built-in support to expose Representational State Transfer (REST) endpoints
- **Agile Dependency Injection**: each component can be injected and called in the context of other components without the need for boiler plate code
- **Flexible Storage System:** the actual storage implementation is abstracted from the specification and can easily be replaced according to the requirements at hand

Verstichel, S., Kerckhove, W., Dupont, T., Volckaert, B., Ongenae, F., Turck, F. and Demeester, P.
LimeDS and the TraPIST Project: A Case Study - An OSGi-based Ontology-enabled Framework Targeted at Developers in Need of an Agile Solution for Building REST/JSON-based Server Applications.
In *Proceedings of the 7th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K 2015) - Volume 2: KEOD*, pages 501-508
ISBN: 978-989-758-158-8
Copyright © 2015 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

- **Scalability and Profiling Toolkit:** support for load-balancing services and data caching
- **Support for Processing of JSON-LD:** formatted data and reasoning on that data
- **Polyglot Environment:** support for multiple programming languages: Java, JavaScript and Python
- **Powerful Management User Interface (UI).**

The remainder of this paper is structured as follows: Section 2 introduces the PoC application while LimeDS, the foundation for the TraPIST developments is presented in Section 3. In Section 4 a number of important modules are detailed. A selection of related work is enlisted in Section 5. Finally, we conclude this paper in Section 6.

2 PoC DEMONSTRATOR

To demonstrate the strength of the presented LimeDS framework, a PoCs has been defined and is being implemented. This PoC concerns the personalisation of feasible connections at interchange stations. The steps needed to create such an application using the LimeDS framework are presented in this section.

In general, we will employ an ontology to describe public transport related concepts, and to classify using a combination of OWL DL and SWRL based reasoning. The LimeDS framework can then be used to compose the data flows for specific applications, allowing to connect a variety of data producers (e.g. railway timetable information providers) with reasoning modules drawing conclusions based on the currently available information and context, and exposing these reasoning results as REST endpoints for visualisation by (mobile) client applications.

2.1 Personalised Connections

An important aspect of a satisfactory travel experience for all travellers is the fact that transportation information should be correct and accurate at all times, and preferably tuned for the specific situation or context of the person at hand. Let us clarify this with an example: a connecting public transportation option may not be feasible to catch for every type of person if only five minutes are scheduled between arriving at a station and boarding that scheduled connection leaving from a different platform, as there is a need to disembark, orient yourself and find/make your way to the other platform and finally board the connecting transport. In other words, the time a traveller requires to transfer between public transport platforms can be dependent on the context (e.g. a first time visit versus daily use/knowledge of the station layout) and physical limitations of that person (e.g. people with physical disabilities or in wheelchairs, elderly, people using a child carrier or heavy travelling luggage). In order to come up with a true personalised travel guidance system, this contextual information needs to be captured and taken into account before presenting travelling guidance to the user.

2.1.1 Ontology Engineering

In support of this scenario an ontology has been created extending two existing ontologies:

- *Transit* (Davis, 2011): A vocabulary for describing transit systems and routes, and
- *Weather* (Gajderowicz, 2011): Based on the ontology created by Aaron Elkiss (Elkiss, 2011).

Starting from the *Transit* and *Weather* ontologies an extension has been modelled, representing the railway timetable from Nationale Maatschappij Belgische Spoorwegen (Belgian National Railways) (NMBS)/SNCB (SNCB/NMBS, 2015) at a number of Belgian railway stations. This represents the railway schedule as it should be if everything runs according to plan. The main new Web Ontology Language (OWL) class has been named *Connection*. Individuals of these *Connection*s represent the actual running of that service on a specific date and are used at-runtime to determine which of the *Connection*s at any given railway station can be caught for the given traveller's profile. To perform the classification, a combination of OWL DL as well as SWRL reasoning is adopted. A number of illustrative axioms are presented in the following paragraphs.

For a *Connection* to be classified as suitable for travellers with a mobility impairment, the OWL definition is given below, based on the terms available in the *Transit* vocabulary. It specifies that the *Connection* should be on a route which has a pre-arrangement for mobility impaired passenger, this arrangement should have been confirmed by the operator and the *Connection* should depart from a platform at ground level:

```
MobilityImpairmentSuitableConnection
<=>
(route some owl:Thing)
 and (hasAccessArrangement value prearranged)
 and (platform some xsd:int[> "-1"^^xsd:int])
 and (isArrangementOK value true)
```

It should be clear that for other situations or other transport modes, such as bus, taxi or cycle hire, other definitions can be specified in the domain ontology, used for that specific operator deployment. However, thanks to the generic notion of a *MobilityIm-*

pairmentSuitableConnection, the UI visualising this information does not need to be aware of the specific definitions in place for that specific situation. The reasoner performs the job of filtering and classification. In first instance, the use of as much DL-based axioms as possible has been pursued, for the reason of genericness and to support the ability to exchange multiple OWL DL enabled reasoner implementations. However, sometimes it might not be possible to purely rely on OWL DL. To facilitate those more complex situations, support for SWRL rules has been included as well. An example of using SWRL to support complex rules and to take the real-time running information of the *Conection*s into account, is given below:

```
MobilityImpairmentSuitableConnection
  (?connection),
currentTime(Now, ?ct),
departureTime(?connection, ?departuretime),
hasDelay(?connection, ?delay),
add(?earliestdeparturetime, ?ct, 900000),
add(?realdeparturetime,
  ?departuretime, ?delay),
greaterThan(?realdeparturetime,
  ?earliestdeparturetime)
->
MobilityImpairmentTimedSuitableConnection
  (?connection)
```

This SWRL rule specifies that, for a *Connection* to be classified as a *MobilityImpairment**Timed**SuitableConnection*, it should already have been classified as a *MobilityImpairmentSuitableConnectionthe* and in addition its actual departure time should be at least 15 minutes in the future.

Other definitions have been included for other categories of passengers as well, such as *HearingImpairmentSuitableConnection* or *VisualImpairmentSuitableConnection* in case of *Connection*s with specific requirements for an on-board Passenger Information System (PIS) or taking into account the earlier introduced *Weather* ontology. Examples include a *FrostSuitableConnection*, *SnowSuitableConnection* or *BicycleSuitableConnection*.

2.1.2 LimeDS *Data Flow*

Once the ontology and its defining axioms has been agreed upon, the implementation within the LimeDS framework can start. For this, the LimeDS *Flow Builder* can be used. For the scenario presented in this section, the *Data Flow* as modelled using the LimeDS *Flow Builder* is illustrated in Figure 1.

The main *Flow Function* is illustrated in the middle of the figure. This represents the configuration of the reasoner. In addition, the configuration also allows to specify the endpoint which should

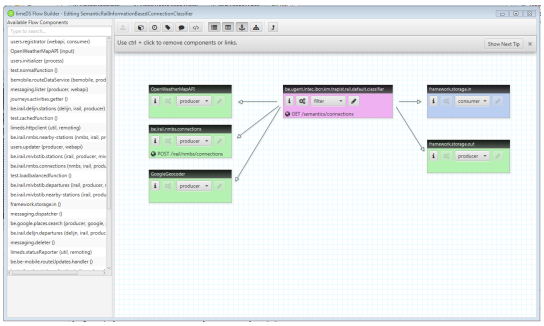

Figure 1: *Data Flow* to classify the personalised connections.

be exposed and used by client applications to trigger the *Data Flow*, as well as extra-functional properties for caching, load-balancing and authentication/authorisation characteristics, and this in a user friendly manner without the need to write any boiler plate code.

Finally, on the left hand side in Figure 1, several input components (data producers) can be seen. These ensure that the correct information is *(i)* fetched from the online data sources (e.g. for retrieving the at-runtime information of the railway operations, the weather at a certain location and optionally geocoding that location), and *(ii)* converted into JSON-LD ready for processing by the reasoner component, illustrated in the middle of the figure. On the right hand side, components are visualised and linked to the reasoner for persistency functionality. As indicated in Section 3, built-in support for persistency by means of MongoDB (Chodorow, 2013) is provided by the framework.

3 LimeDS

The TraPIST applications are built on top of LimeDS, which facilitates development by leveraging on the *Data Flow* abstraction. In this section we highlight how LimeDS has been engineered and how it supports the processing of semantically enriched knowledge to support data-oriented situation-aware applications.

3.1 Data Flows

The *Data Flow* abstraction adds structure to the way in which data travels through the system by specifying a number of generic components that each represent a specific role in the way the data can be consolidated. By leveraging LimeDS, the TraPIST applications are able to respond in an agile manner to the dynamic environment in which they will be deployed, e.g.:

- A public transport operator provides a new API which facilitates access to information that was not readily available before, making some of the TraPIST components obsolete.
- A new algorithm is published that would greatly enhance the way in which new information can be derived from historic passenger records, but it requires additional data that is already available via another framework component currently not involved in the process.

The *Data Flow* system revolves around two conceptual component types (Figure 2):

- **Flow Functions:** components that take in an optional JSON argument and can optionally produce processed JSON data.
- **Flow Processes:** components that execute a specific set of actions when triggered.

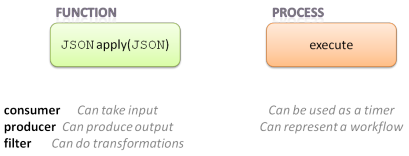

Figure 2: The *Data Flow* paradigm.

Using these two conceptual types, complex data-oriented services can be built by combining instances of *Flow Functions* and *Flow Processes* components to create a modular and dynamic implementation of the required functionality. In LimeDS, each such component can be:

- implemented in the language best-suited for the developer (a.o. JavaScript, Java, Python),
- exposed as a REST endpoint,
- protected against unwanted behaviour by automated dependency management,
- guarded by means of authentication and authorisation mechanisms, using a user-friendly configurable strategy,
- automatically load balanced and cached,
- profiled to allow for performance tracking,
- (graphically) connected into a *Data Flow*, and
- reused from existing *Data Flows* and changed to other *Data Flow* components introducing novel *Data Flows*.

An important requirement for the LimeDS framework is that it must interact with all sorts of external systems, such as mobile applications, web applications, operator management software, public transport time table sources, geographical databases,

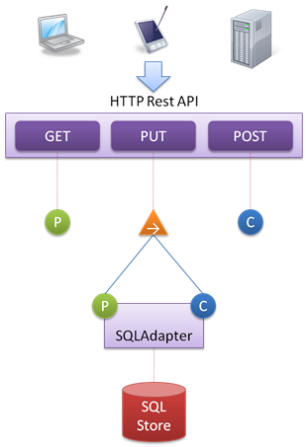

Figure 3: *Data Flow* involving external systems.

weather information services, etc. As the primary mechanism for communication within the framework, the *Data Flow* model must hence facilitate this.

Figure 3 demonstrates how the *Data Flow* model can support external systems by means of two examples: *(i)* clients connecting to a public HTTP REST API and *(ii)* a SQL database system. Retrieval requests (e.g. HTTP GET) from public API can be redirected to a relevant *Flow Function* that can answer the requests, while other *Flow Functions* are well suited to handle requests that provide new content (e.g. HTTP POST). External system calls could also trigger a *Flow Function*, for example by sending update requests (e.g. HTTP PUT) that change the context of a field of an object. In the diagram this results in an operation on a SQL database, using producers and consumers, i.e. specific *Flow Function* implementations provided by a module that is able to communicate with the external SQL store.

3.2 Dynamic Module System

The dynamic module system for LimeDS has been based on an existing standard specification called Open Services Gateway initiative (OSGi) (OSGi Alliance, 2003). A module in OSGi is called a bundle. Bundles are represented as plain JAR-files that contain additional meta-data in the manifest of the JAR. This meta-data is what defines the bundles and includes two important concepts: *(i)* a bundle version identifier and *(ii)* an explicit specification of imported and exported packages.

The OSGi framework hides everything that is inside the bundle, unless a package is explicitly exported (as defined in the manifest). Other bundles can only use the classes that are in the exported package by explicitly importing the package. This concept of code hiding and explicit sharing enforces modularity

and all the benefits that come with it.

It is considered good practice to use code sharing through exports/imports only to share definitions (interfaces) and model classes between bundles and employ the OSGi service model to do the actual collaboration.

3.3 Service Layer

The OSGi specification also consists of a service model that provides a very good match with the LimeDS service layer that was envisioned as illustrated in Figure 4. The service layer is one of the most important concepts of the framework as it allows modules to communicate while still retaining a loose coupling, enabling functionality to be extended or replaced at-runtime. It also provides the foundation for the *Data Flow* system that is the basis of the presented approach and further explained in Section 4.

Figure 5 shows the Application Programming Interface (API) of the OSGi service model. Via the `BundleContext`, which can be retrieved for each bundle or module, developers can access the three main operations that are required for the LimeDS framework: registering a service, retrieving a service and Listening to a service.

3.4 TraPIST API

The system is self-documenting for all APIs that are implemented using LimeDS *Data Flows*. Input and output formats for the API can reference LimeDS JSON schema documents that are locally stored or through an accessible URL. The schema documents can describe JSON objects of a specific type and can be used to automatically validate *Data Flow* messaging. The input and output JSON formats of *Flow Functions* can be documented using the `@Documentation` annotation in Java or through the configuration window of the *Flow Builder* client for Javascript or Python-based implementations. Once the input and output formats are defined, these meta JSON-objects can be used as a mechanism of validating the actual input and output of the *Flow Functions* at-runtime.

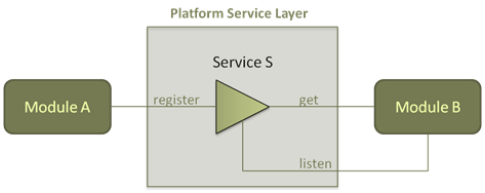

Figure 4: TraPIST service layer.

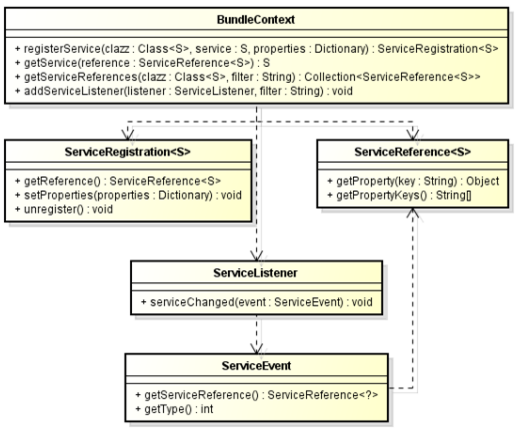

Figure 5: OSGi service model.

In contrast to more elaborate JSON description formats such as JSON schema (JSON–schema.org, 2015), our goal is to keep the format simple and representative of how an actual instance of a JSON item that matches the format looks, while still retaining a valid JSON structure. Four basic concepts are introduced, namely *(i)* Primitive types, *(ii)* JSON Objects, *(iii)* JSON arrays and *(iv)* References.

The first two concepts are straightforward. A JSON array is represented in the same way as it would occur in an actual JSON instance, but instead of multiple values of a certain type, it will only contain a single item that represents the type of the potential contents of the array. In reality, this item will either be a primitive type description, an object description or another array description. When the item that is contained in the array is marked as optional, this applies to the field that holds the array. The format allows references to other JSON documentation objects through the `@typeRef` field, e.g.

```
{
"@typeRef" :
"http://example.org/list/example/Address.json"
}
```

The validator is compatible with these references and will recursively evaluate the links to process the actual field descriptors.

4 FRAMEWORK MODULES OVERVIEW

For ease of reference, the framework modules are split into three categories: *(i)* TraPIST specific modules (`TraPIST Modules`), *(ii)* generic LimeDS modules that are required when developing any TraPIST module (`LimeDS Developer Modules`) and *(iii)* modules that are part of the LimeDS framework's internal details (`LimeDS Internal Modules`).

4.1 Reasoner Configuration Module

The `tools.reasoning.configurator` bundle can be used instantiating the specified reasoner configurations into fully functional *Flow Function* components, integrating semantic reasoning into the *Data Flows*. To instantiate a new reasoner, one needs to create a configuration file with four different properties *(i)* name, *(ii)* tags, *(ii)* url (refering to the ontology that is used as the base model for this reasoner instance) and *(iv)* reasoner (currently supported reasoners are Pellet (Sirin et al., 2007) and the Jena OWL DL reasoner (Carroll et al., 2004)). Each new configuration will result in a *Flow Function* becoming available with the specified name. This *Flow Function* can then be used as any other normal *Data Flow* component. The following request schema applies to this *Flow Function*:

```
{    "type" : "String [add, get, dump]
    (The type of the request.)",
    "body" : {
        "@typeInfo" : "Object *
        (A JSON-LD object when the request
        is of the type add.)"
    },
    "query" : "String *
    (A SPARQL query when the request
    is of the type get.)"}
```

Each reasoner instance supports three operations:

- **add:** Adds the JSON-LD object encapsulated in the body of the request to the ontology.
- **get:** Retrieves information from the ontology by executing the SPARQL query encapsulated in the query of the request.
- **dump:** Dumps current state as a RDF/XML file.

4.2 Communication Subsystem

The LimeDS framework supports the standard specification JAX-RS (Burke, 2009), allowing the automated generation of REST Web Services. The principle is very similar to how *Flow Functions* can be scheduled (see Subsection 4.4): JAX-RS annotated services are added to the service registry. The annotations contain information about the REST service path and the available HTTP methods. The *JAX-RS Bridge module* connects with the service registry and is continually scanning for new annotated services.

As shown in Figure 6, the *JAX-RS Bridge* then calls Apache Wink (Apache Software Foundation, 2015) for each discovered service. Wink is a library that translates the annotated resources to HTTP servlets, which can be hosted by a servlet container. To facilitate calling the services provided in this way

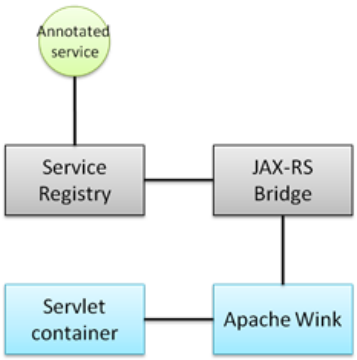

Figure 6: Service to HTTP bridge.

(or external REST services), a client utility library has been developed to easily call REST services with a minimal amount of code. The *REST Client API* allows expressing REST calls in a rather natural way, e.g. in order to post a JSON object to an example service, one can write the following statement:

```
JsonNode jsonObject =
 Json.from("\"{ \"example\" :
  \"Hello world!\" }");
client
 .target("http://example.com/API/example")
 .post(jsonObject).returnNoResult();
```

With both a way to easily expose REST services (server-side) and a way to fluently call and make use of these services (client-side), the framework provides all the features that are required of a basic communication system.

4.3 Load Manager Subsystem

The framework provides a *Load Manager System* which facilitates *Data Flow* component developers in adding robustness and scalability to their services. Figure 7 shows the basic principle that is used to implement this system. For each registered service, the *Load Manager* will generate a proxy service that transparently captures all requests sent to the original service and redirects it to the service instance (running locally or remote) that is currently best suited to answer the request. This decision can be based on the number of requests for each instance, the mean response times, etc.

4.4 API and Scheduling Subsystem

This bundle includes important constants and values, the complete *Data Flow* API and a definition of the storage system. *Flow Functions* can be activated periodically to perform certain tasks, e.g. a *Flow Function* that enriches all stored places of interest with up-to-date weather information can be activated hourly.

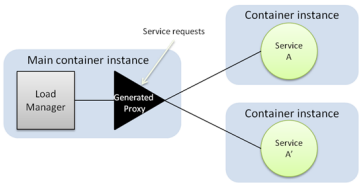

Figure 7: Load manager system.

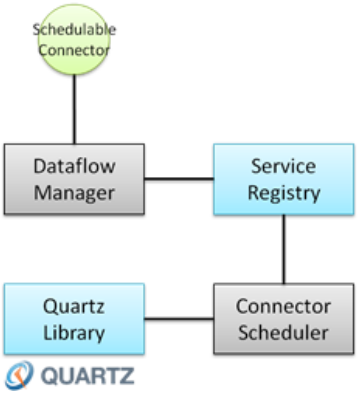

Figure 8: Scheduling functionality.

To support this functionality, the framework must integrate a scheduler. Figure 8 shows the high level modules which are involved to enable this specific feature. We use the Quartz library (Cavaness, 2006) to actually trigger the recurring tasks while a *Connector Scheduler* is responsible for the synchronisation of schedulable *Flow Functions* available from the service registry with the triggers that are enabled within the Quartz scheduler.

Finally, the architecture of the framework adds support for dynamic languages (e.g. JavaScript). As the framework does not rely on a fixed data representation, static typed languages (such as Java) could introduce unnecessary development overhead for certain types of use-cases. Dynamic languages are not hindered by fixed definitions and could prove to be more elegant for simple cases.

5 RELATED WORK

5.1 JAX-RS

Java API for Restful Web Services (JAX-RS) (Burke, 2009) is the Java standard specification for a Java API that allows implementing REST-based applications. It relies on annotations added as meta-data on classes to transform these into REST resources that are exposed as HTTP-endpoints. Various implementations of this standard exist ranging from Jersey (Oracle Corporation, 2015), to Resteasy (RedHat, 2015) and Apache Wink (Apache Software Foundation, 2015).LimeDS uses Apache Wink as a base layer and thus offers support for JAX-RS resources. On top of this, the *Data Flow* mechanism allows developers to build Web API with integrated support for reliability, scalability and security with a minimal amount of code that can easily be modified at-runtime.

5.2 Dropwizard

Dropwizard (Dallas, 2014) is an open-source Java project that integrates a number of mature libraries into a light-weight ecosystem targeted at developing REST-based applications. This is done by allowing JAX-RS resources to be wired with various support services, ranging from logging, to storage and authentication / authorisation. The goal of Dropwizard is similar to that of LimeDS, but the focus and scope differs. While Dropwizard is a very powerful framework to build high-performance REST-based applications that are easy to monitor and maintain, LimeDS provides a more dynamic environment where changes can be made on-the-fly while the application is running. Dropwizard also focuses on a single-node setup, while LimeDS can be configured to run as a cluster with load balancing.

5.3 OSGi

OSGi (OSGi Alliance, 2003) is a proven standard that specifies a modular middleware framework for Java. OSGi implementations such as Apache Felix or Eclipse Equinox (The Eclipse Foundation, 2015) provide an execution environment where software modules (bundles) can be installed at-runtime and communicate safely in highly dynamic circumstances. While a lot of applications could benefit from the (at-runtime) modularity and loose coupling of the OSGi framework, a lot of developers struggle to unlock its true potential because of the steep learning curve. LimeDS is built on top of OSGi and can offer developers OSGi features such as dynamic reconfiguration of dependencies and at-runtime addition of new functionalities in a much easier to use package – albeit for a more restricted set of REST-based applications.

5.4 Vert.x

Vert.x (Clement Escoffier, 2015) is a Java-based framework that allows developers to build 'reactive' applications by setting up modular components called *Verticles* and by routing data between these components using a distributed event bus. The 'reactive' keyword refers to the framework using event-driven,

non-blocking calls and the fact that it is designed from the ground up to support high availability. Some of the additional services provided by Vert.x are: *(i)* the ability to use multiple programming languages (Java, JavaScript, Groovy, Ruby), *(ii)* integration of storage solutions (MongoDB, SQL, Redis), *(iii)* support for clustering, advanced capturing of metrics on a per-component basis and *(iv)* built-in support for authentication and authorisation. Vert.x and LimeDS overlap in some features, but the latter focuses on HTTP/JSON-based applications with a clear choice to build upon the services model of OSGi, while Vert.x offers a more generic approach. When developing those type of applications, LimeDS has an edge as there is no need for boiler plate code in setting up the HTTP endpoints. Additionally, LimeDS has built-in support for semantic reasoning.

6 CONCLUSIONS AND OUTLOOK

In this paper we presented LimeDS, an OSGi-based framework allowing for agile development of data processing applications which rely on a multitude of heterogeneous open data and knowledge sources. LimeDS has been employed in the TraPIST research project to develop multimodal public transportation applications with a focus on offering personalised and context-filtered information to the end-user, allowing that end-user to be presented with information tailored to his or her needs and (physical) disabilities. LimeDS is able to deal with sudden unavailability of select data sources and offers a developer-friendly way of reasoning over this data. Furthermore it can (visually) aid with the construction of data processing/reasoning workflows and provides inherent and configurable support for scaling, resilience and fallback scenarios. Current plans are to release LimeDS in Open Source format to the community at the end of the TraPIST project (i.e. January 2016).

ACKNOWLEDGEMENTS

The iMinds TraPIST project is co-funded by iMinds (Interdisciplinary Institute for Technology), a research institute founded by the Flemish Government. Companies and organisations involved in the project are Televic Rail, Be-Mobile, Digitopia, NMBS/SNCB and TreinTramBus, with project support of IWT.

REFERENCES

Apache Software Foundation (2015). Apache Wink – a simple yet solid framework for building RESTful Web Services. Online. https://wink.apache.org/.

Burke, B. (2009). *RESTful Java with JaX-RS*. O'Reilly Media, Inc.

Carroll, J. J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., and Wilkinson, K. (2004). Jena: implementing the semantic web recommendations. In *Proceedings of the 13th international conference on World Wide Web, Alternate track papers & posters (WWW Alt. 2004)*, pages 74–83, New York, NY, USA. ACM.

Cavaness, C. (2006). *Quartz Job Scheduling Framework: Building Open Source Enterprise Applications*. Pearson Education.

Chodorow, K. (2013). *MongoDB: the definitive guide*. O'Reilly Media, Inc.

Clement Escoffier, M. K. (2015). Vert.x – a toolkit for building reactive applications on the JVM. Online. http://vertx.io/.

Dallas, A. (2014). *RESTful Web Services with Dropwizard*. Packt Publishing Ltd.

Davis, I. (2011). TRANSIT: A vocabulary for describing transit systems and routes. Online. http://vocab.org/transit/terms/.html.

Elkiss, A. (2011). A weather ontology. Online. http://www.csd.abdn.ac.uk/∼ggrimnes/AgentCities/WeatherAgent/weather-ont.daml.

Gajderowicz, B. (2011). *Using Decision Trees for Inductively Driven Semantic Integration and Ontology Matching*.

IBCN (2014). LimeDS – An OSGi-based Java framework targeted at developers that need an agile solution for building REST/JSON-based server applications in rapidly changing environments (a.k.a. when the customer does not know what she/he needs). Online. http://bit.ly/1Qy0qWO.

iMinds VZW (2014). TraPIST – Information for train passengers on a silver platter. Online. https://www.iminds.be/en/projects/2014/03/20/trapist.

JSON–schema.org (2015). JSON Schema – JSON Schema describes your JSON data format. Online. http://json-schema.org/.

Oracle Corporation (2015). Jersey – RESTful Web Services in Java. Online. https://jersey.java.net/.

OSGi Alliance (2003). *OSGi Service Platform, Release 3*. IOS Press, Inc.

RedHat (2015). RestEASY – Distributed peace of mind. Online. http://resteasy.jboss.org/.

Sirin, E., Parsia, B., Grau, B. C., Kalyanpur, A., and Katz, Y. (2007). Pellet: A practical OWL-DL reasoner. *Journal of Web Semantics*, 5(2):51–53.

SNCB/NMBS (2015). SNCB/NMBS – Belgian national railways. Online. http://www.belgianrail.be/en/Default.aspx.

The Eclipse Foundation (2015). Equinox – an implementation of the OSGi core framework specification. Online. http://www.eclipse.org/equinox/.