



simpli-city

The Road User Information System Of The Future

WP3 – Architecture, Functional & Technical Specification, Security & Privacy Concept, Integration

D3.2.2: Technical Specification

Deliverable Lead: TU Vienna

Contributing Partners: TUV, ASC, TIE, TUDA, IBM, FGM, TALK, TEMP

Delivery Date: 01/2014

Dissemination Level: Public

Version 1.00

This document provides an in-depth technical definition of all SIMPLI-CITY software components and their subcomponents, including the technical specification of the provided interfaces. Furthermore, it documents the defined app and service data models as well as the technologies chosen as foundation for the implementations of the SIMPLI-CITY software components.



Document Status	
Deliverable Lead	Stefan Schulte, TU Vienna
Internal Reviewer 1	Vadim Petrenko, TIE
Internal Reviewer 2	Freddy Lecue, Robert Tucker, IBM
Type	Deliverable
Work Package	WP3: Architecture, Functional & Technical Specification, Security & Privacy Concept, Integration
ID	D3.2.2: Technical Specification
Due Date	30.09.2013
Delivery Date	28.01.2014
Status	Approved

Document History	
Contributions	V0.01, TUV, 30.07.2013 V0.02, ASC, TUV, 07.08.2013 V0.03, TUV, 14.08.2013 V0.04, TUV, 20.09.2013 V0.10, ASC, TUV, 14.11.2013 V0.11, TALK, TUV, 15.11.2013 V0.12, IBM, TUV, 18.11.2013 V0.13, TUDA, TUV, 19.11.2013 V0.14, TALK, TUV, ASC, 20.11.2013 V0.15, TUV, ASC, 21.11.2013 V0.16, TUV, TIE, ASC, 26.11.2013 V0.17, TUV, ASC, 02.12.2013 V0.18, TUV, 04.12.2013 V0.19, TUV, ASC, 05.12.2013 V0.20, TUV, ASC, 06.12.2013 V0.21, TUV, ASC, TIE, 09.12.2013 V0.22, TUV, TUDA, TIE, 10.12.2013 V0.23, TUV, ASC, TEMP, 11.12.2013 V0.24, TUV, ASC, TEMP, 12.12.2013 V0.25, TUV, TALK, 13.12.2013 V0.26, TUV, TEMP, ASC, TIE, 17.12.2013

	V0.50, TUV (First Review Version), 19.12.2013 V0.60, TUV (Reviews for Sections 1-4 have been addressed), 20.12.2013 V0.61, TUV, ASC, IBM (Reviews for Section 5 have been addressed), 15.01.2014 V0.62, TUV (Reviews for Sections 6-10 partially included), 17.01.2014 V0.63, TUV, (Reviews for Subsections 6.X further regarded), 21.01.2014 V0.64, TUV, ASC, TUDA, TALK, TIE (Reviews for Section 6 fully incorporated), 22.01.2014 V0.65, TUV, IBM, TIE, ASC, TALK, TUDA (Started to incorporate reviews for Section 7), 22.01.2014 V0.66, TUV (Reviews for Section 9 and 10 fully incorporated), 23.01.2014 V0.67, TUV, ASC, TUDA (Reviews for Section 7 fully incorporated), 23.01.2014 V0.70, TUV, IBM, TUDA, ASC, TIE (Very last comments solved), 27.01.2014
Final Version	V1.00, TUV (Checked layout, final minor changes), 28.01.2014

Disclaimer

The views represented in this document only reflect the views of the authors and not the views of the European Union. The European Union is not liable for any use that may be made of the information contained in this document.

Furthermore, the information is provided “as is” and no guarantee or warranty is given that the information is fit for any particular purpose. The user of the information uses it at its sole risk and liability.

Project Partners



TECHNISCHE
UNIVERSITÄT
WIEN
Vienna University of Technology

Vienna University of Technology (Coordinator),
Austria



Ascora GmbH, Germany



TIE Nederland B.V., The Netherlands



Technische Universität Darmstadt, Germany



IBM Research – Ireland
Smarter Cities Technology Centre



Forschungsgesellschaft Mobilität, Austria



Talkamatic AB, Sweden



Atos Worldline, Spain



Centro Ricerche FIAT, Italy



SRM – Reti e Mobilità, Italy

Executive Summary

Within this deliverable D3.2.2, the technical specification of the SIMPLI-CITY software components is documented. Together with the already existing Requirements Analysis Report (D2.3), the initial State of the Art Review (D2.4.1), the Global Architecture Definition (D3.1), the Functional Specification (D3.2.1), and the Holistic Security and Privacy Concept (D3.3), this deliverable provides the foundation for the Research, Technology, and Development work to be conducted within the SIMPLI-CITY work packages WP4-WP6.

The goals of this deliverable are twofold: first and foremost, it documents the advancements in the definition of the SIMPLI-CITY software engineering process and is therefore an important artefact for the task-internal coordination between the project partners; second, it establishes the definition of the interfaces between the major SIMPLI-CITY software components and is therefore also providing important information for the whole project as well as external stakeholders. The outcome of this Technical Specification is an in-depth technical description of all SIMPLI-CITY software components.

From a technical perspective, the most important choices defined within this deliverable are the selection of the OSGi framework as a basic platform for the SIMPLI-CITY Mobility Services Framework, and of the mobile operating system Android for the SIMPLI-CITY Personal Mobility Assistant (more precisely: the Application Runtime Environment). In detail, Apache Felix has been chosen as the actual OSGi framework, while Apache Karaf has been chosen as OSGi runtime. For the Cloud-based Information Infrastructure, the most important technical choice is the Bucket concept introduced in D3.1 and its selection of Amazon S3 for binary storage, MongoDB for semi-structured data, and Sesame for semantic data. For the SIMPLI-CITY Data Processing component, Pellet and IBM InfoSphere Stream have been selected in order to achieve scalable, real-time stream processing and reasoning.

In order to make this deliverable self-contained, it starts with a recapitulation of the SIMPLI-CITY software system (from deliverable D3.2.1). Afterwards, the technical specification of the SIMPLI-CITY Data Integration, Mobility Services Framework, Personal Mobility Assistant, and Developer Support software components are presented: An in-depth discussion of the technical aspects of each component is provided by discussing major design decisions, comparing different software frameworks and technologies that could be selected as a foundation for the respective SIMPLI-CITY software component, and the actual selection of a particular technology. Afterwards, missing elements and implementation needs as well as an updated component structure are presented. In order to explain how to interact with the respective software component, the RESTful and Java interfaces provided are discussed in detail.

Afterwards, the service and app data models applied in SIMPLI-CITY are introduced. The document concludes with a brief summarisation of the main outcomes.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 5 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

Table of Contents

1	Introduction	7
1.1	SIMPLI-CITY Project Overview	7
1.2	Deliverable Purpose, Scope and Context	8
1.3	Document Status and Target Audience	8
1.4	Abbreviations and Glossary	9
1.5	Document Structure	9
2	System Overview	10
2.1	SIMPLI-CITY Architecture Summary	10
2.2	End User Perspective	14
2.3	Software Developer Perspective	15
2.4	Data Integration Perspective	16
3	General Description of Technical Specification	18
3.1	Technical Specification	18
3.2	Generic Comparison Criteria	20
4	Major Design Principles	22
4.1	Data Storage	22
4.2	Security and Privacy	22
4.3	Java-based and RESTful Interfaces	23
4.4	JavaScript Object Notation	24
4.5	Resource Description Framework	25
5	Technical Specification: Data Integration	26
5.1	Data Processing	26
5.2	Cloud-based Information Infrastructure	50
5.3	Sensor Abstraction and Interoperability Interfaces	116
5.4	Media Data Streams and Data Prefetching Logic	147
6	Technical Specification: Mobility Services Framework	204
6.1	Service Runtime Environment	204
6.2	Monitoring	219
6.3	Context-based Service Personalisation	243
6.4	Service Registry	267
6.5	Service and App Marketplaces	314
7	Technical Specification: Personal Mobility Assistant	345
7.1	Application Runtime Environment	345
7.2	Multimodal Dialogue Interface	379
7.3	PMA-based Sensor Abstraction	394
8	Technical Specification: Developer Support	407
8.1	Application Design Studio	407
8.2	Service Development API	415
9	Manifest Data Models	423
9.1	Service Manifest Model	423
9.2	App Manifest Model	429
10	Conclusion	434
	References	435

1 Introduction

SIMPLI-CITY – The Road User Information System of the Future – is a project funded by the Seventh Framework Programme of the European Commission under Grant Agreement No. 318201. It provides the technological foundation for bringing the “App Revolution” to road users by facilitating data integration, service development, and end user interaction.

This document provides an in-depth technical definition of all SIMPLI-CITY software components, including the selection of software and technologies that will provide the foundation for the individual components, interface definitions, and a declaration of missing elements and implementation needs.

1.1 SIMPLI-CITY Project Overview

Analogously to the “App Revolution”, SIMPLI-CITY adds a “software layer” to the hardware-driven “product” mobility. SIMPLI-CITY will take advantage of the great success of mobile apps that are currently being provided for systems such as Android, iOS, or Windows Phone. These apps have created new opportunities and even business models by making it possible for developers to produce new apps on top of the mobile device infrastructure. Many of the most advanced and innovative apps have been developed by players formerly not involved in the mobile software market. Hence, SIMPLI-CITY will support third party developers to efficiently realise and sell their mobility-related service and app ideas by a range of methods and tools, including the Mobility Services and App Marketplaces.

In order to foster the wide usage of those services, an holistic framework is needed which structures and bundles potential services that could deliver data from various sources to road user information systems. SIMPLI-CITY will provide such a framework by facilitating the following main project results:

- **Mobility Services Framework:** A next-generation European Wide Service Platform (EWSP) allowing the creation of mobility-related services as well as the creation of corresponding apps. This will enable third party providers to produce a wide range of interoperable, value-added services, and apps for drivers and other road users.
- **Mobility-related Data as a Service:** The integration of various, heterogeneous data sources like sensors, cooperative systems, telematics, open data repositories, people-centric sensing, and media data streams, which can be modeled, accessed, and integrated in a unified way.
- **Personal Mobility Assistant:** An end user assistant that allows road users to make use of the information provided by apps and to interact with them in a non-distracting way – based on a speech recognition approach. New apps can be integrated into the Personal Mobility Assistant in order to extend its functionalities for individual needs.

To achieve its goals, SIMPLI-CITY conducts original research and applies technologies from the fields of Ubiquitous Computing, Big Data, Media Streaming, the Semantic Web, the Internet of Things, the Internet of Services, and Human-Computer Interaction. For more information, please refer to the project website at <http://www.simpli-city.eu>.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 7 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

1.2 Deliverable Purpose, Scope and Context

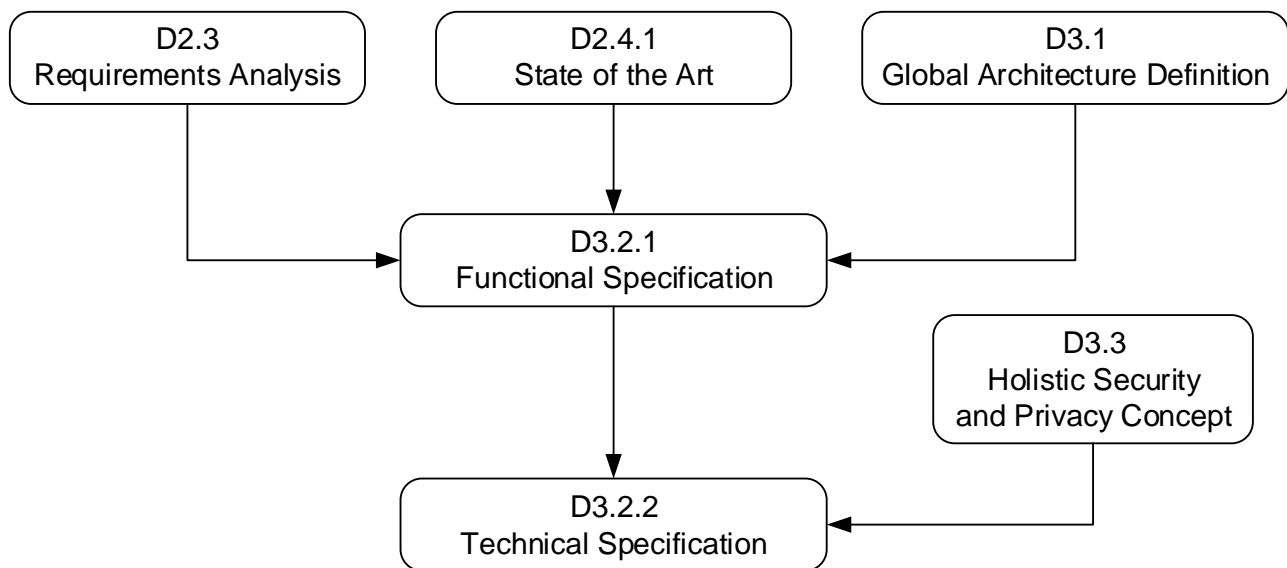


Figure 1: Context of Deliverable D3.2.2

The purpose of this document is to provide the detailed technical specification of all SIMPLI-CITY software components. As depicted in Figure 1, this Technical Specification is based on the software engineering activities of SIMPLI-CITY, which were previously presented within the Requirements Analysis (deliverable D2.3), the Global Architecture Definition (deliverable D3.1), and – most importantly – the Functional Specification (deliverable D3.2.1). Furthermore, it takes into account the current state of the art as identified in deliverable D2.4.1. The Holistic Security and Privacy Concept (deliverable D3.3) complements the document at hand by defining (amongst other things) security and privacy aspects to be regarded during the Research, Technology, and Development work in work packages WP4-6.

For this purpose, this document defines the individual subcomponents on a deep technical level by discussing major design decisions, a comparison of possible technologies to make use of and a subsequent selection of the technological foundation for the implementation efforts, a definition of missing elements and implementation needs, and an updated component structure. Very importantly, the interfaces offered to other software components are specified.

1.3 Document Status and Target Audience

This document is listed in the Description of Work (DoW) as “public”, since it provides the technical specifications of the SIMPLI-CITY software components and can therefore be used by external parties in order to get according insights into the project activities.

While the document primarily aims the project partners, this public deliverable can also be useful for the wider scientific and industrial community. This includes other publicly funded projects, which may be interested in collaboration activities.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 8 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

1.4 Abbreviations and Glossary

A definition of common terms and roles related to the realization of SIMPLI-CITY as well as a list of abbreviations is available in the supplementary document “Supplement: Abbreviations and Glossary”, which is provided in addition to this deliverable.

Further information can be found at <http://www.simpli-city.eu>.

1.5 Document Structure

This deliverable is broken down into the following sections:

Section 1 provides an introduction for this deliverable including a general overview of the project, and outlines the purpose, scope, context, status, and target audience of this deliverable.

Section 2 gives an overview of the overall SIMPLI-CITY software architecture. Furthermore, a system level analysis of the SIMPLI-CITY components from an end user, software developer, and data integration perspective is provided.

Section 3 introduces the structure of the component descriptions in Sections 5-8.

Section 4 provides an overview of major design principles which have to be regarded by all SIMPLI-CITY software components.

Section 5 presents the technical specification of the SIMPLI-CITY Data Integration components, recapitulating briefly their functional specification in terms of major design decisions, a comparison and selection of technologies and software frameworks which may be used as foundation for the SIMPLI-CITY software components, an analysis of missing features and subsequent implementation needs, an update of the component's structure, and the interfaces provided by the respective component to other (SIMPLI-CITY-internal) software components.

Section 6 does the same for the SIMPLI-CITY Mobility Services Framework.

Section 7 does the same for the SIMPLI-CITY Personal Mobility Assistant.

Section 8 does the same for the SIMPLI-CITY Developer Support software components.

Section 9 defines the data models for services, apps, and message formats as applied in SIMPLI-CITY.

Section 10 concludes the deliverable with a short summarisation of its findings.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 9 / 435
http://www.simpli-city.eu/	Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201			

2 System Overview

Within this section, a brief overview of SIMPLI-CITY is given from a technical perspective. In Section 2.1, the general architecture is summarized as a recapitulation of the outcomes of deliverable D3.1 (Global Architecture Definition). Afterwards, a system level analysis of SIMPLI-CITY is provided. This analysis gives an introductory overview of the different SIMPLI-CITY components. It is based on a functional point of view and illustrates how the different components are employed to achieve the functionality targeted within SIMPLI-CITY. Thus, the context for the detailed technical component specifications within Sections 5-8 is provided.

Within the following functional analysis, three perspectives are covered:

- **User Perspective:** Elaborates on how the components work together to provide means for intuitive and user-friendly interaction (Section 2.2).
- **Software Developer Perspective:** Shows which means are provided within SIMPLI-CITY to support software developers to provide (new) apps and services in order to create an innovative, new road user information system (Section 2.3).
- **Data Integration Perspective:** Highlights the employed means to integrate the different data sources available within SIMPLI-CITY (Section 2.4).

Note: The following subsections are reiterated from the Functional Specification (deliverable D3.2.1) in order to make this deliverable self-contained.

2.1 SIMPLI-CITY Architecture Summary

SIMPLI-CITY is intended to be used by a large number of users from different countries, which employ SIMPLI-CITY-enabled services and apps for a range of different purposes, e.g., to get a bit greener, to raise their comfort or to request road-specific information. In this environment, SIMPLI-CITY targets different transport modes, with a focus on car drivers.

For achieving its goals, the architecture of SIMPLI-CITY needs to be flexible and scalable. The concept for this is to split the architecture into four basic building blocks as depicted in Figure 2:

- **Vehicle & PMA:** Elements that are running on the mobile device and – more generically – inside the vehicle. This covers apps which are executed within the Personal Mobility Assistant (PMA) and it also covers local sensors from the vehicle (e.g., data from the car sensors about the current speed).
- **Elements that are running on the server side:** This mainly covers services running inside the Service Runtime Environment, but it also includes components for the local storage of data and data management in general.
- **External data sources,** which deliver data from external sources into SIMPLI-CITY: This can be sensor information, but it may also be personal data sources such as social networks.
- **Components for supporting developers:** Unlike all other areas, this contains components that are usually used during the software design time and not during runtime. More precisely, developers will be equipped with an Application Design Studio and with a set of Application Programming Interfaces (APIs) and HowTo documents.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 10 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

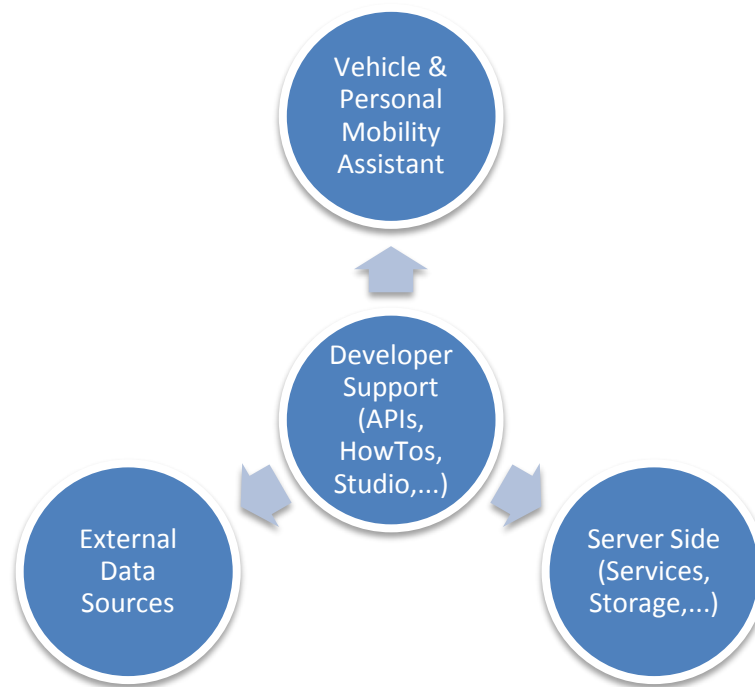


Figure 2: Core Areas of the SIMPLI-CITY Architecture

The “Vehicle & PMA” area focuses on the mobile device that will be the most visible element of SIMPLI-CITY from an end user perspective. Components located in this area contain the Application Runtime Environment as well as all mobile apps. Access to car sensors is also covered via a reduced implementation of the Sensor Abstraction and Interoperability Interfaces, and a Local Storage is created. User interaction is performed via the Multimodal Dialogue Interface. The area has three relationships to other SIMPLI-CITY components: The first one is the interaction between apps and services, which is completely handled by the Application Runtime Environment that communicates with the Service Runtime Environment. The second one is the data prefetching, which will be realized based on a stream connection to the data prefetching server side. Finally, the third relationship is with the App Marketplace which allows users to access apps from the market using a User Interface (UI) inside the device. This UI communicates with the market backend on the server side, where all app market information is stored and provided.

The second area of the SIMPLI-CITY architecture is covering components that will jointly realize the European Wide Service Platform (EWSP) provided by SIMPLI-CITY. Those components will be located at the server side of SIMPLI-CITY, meaning that they do *not* run inside the mobile device. The main component of the server side area is the Service Runtime Environment, hosting and controlling all deployed services. It also coordinates the communication with the PMA and contains components for the Context-based Service Personalization and the Data Prefetching Logic. The server side also covers the data storage facilities, which will be realized by the Cloud-based Information Infrastructure. Additionally, the data access of SIMPLI-CITY to external data sources will be handled by the server side components. This contains the communication with external sensors, but also with user centric and Open Data information sources. Finally, a set of web consoles, which targets developers, will be realized. Through these web consoles, software developers will be able to access information from SIMPLI-CITY including service

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 11 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

monitoring data. Also, software developers will be able to submit their own backend services to the Service Marketplace.

The third area of the architecture represents external data sources (sensors, information sets, calendars, etc.). This area cannot be directly influenced by the project since those data sources are usually provided by third parties. However, SIMPLI-CITY will provide the means to integrate such data sources in an easy, unified way. Examples for data sources are public data sets as well as personalized data sources, such as the personal calendar of a user. SIMPLI-CITY provides the Sensor Abstraction and Interoperability Interfaces component for accessing sensor data from this area. The User Centric & Open Data Access component of SIMPLI-CITY will act as an enabler for making those sources available in SIMPLI-CITY. Media Data Streams are also supported through this component. Finally, the Data Processing component offers sophisticated functionalities for data contextualisation and data analysis.

The final area covers those components that support developers in the creation, deployment, and updating of apps and services. According developer support will be provided in an Integrated Development Environment (Application Design Studio) and APIs, which will bundle most of the resources. In addition, developers will be provided with documentation, guidelines, and examples. This will ease the development of SIMPLI-CITY-based services and apps and therefore increase the impact of the project.

The following Figure 3 shows an overview of the components and their interconnection as introduced in deliverable D3.1.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 12 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

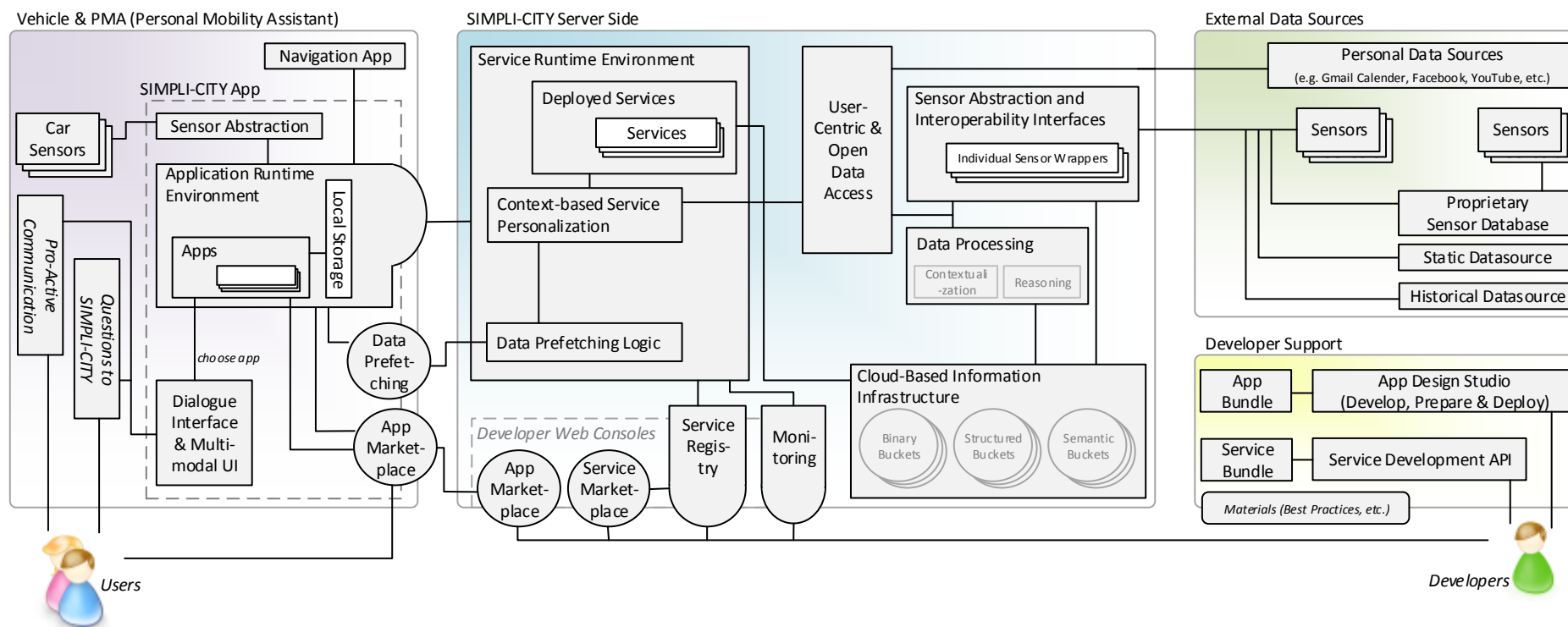


Figure 3: SIMPLI-CITY Architecture

2.2 End User Perspective

The End User Perspective elaborates on the PMA. The PMA constitutes the central interface provided to the SIMPLI-CITY user. It is consequently employed for corresponding user interaction within SIMPLI-CITY. The PMA exploits current smartphone technology and will make SIMPLI-CITY's solutions and functionalities accessible to the user. In this context, a common framework will be provided, which incorporates the SIMPLI-CITY components locally required on a mobile device. Specifically, these are:

- Application Runtime Environment
- PMA-based Sensor Abstraction
- App Marketplace
- Multimodal Dialogue Interface

The Application Runtime Environment constitutes the core of SIMPLI-CITY on the PMA. It provides the means to integrate different apps for supporting the road user and a sub-component to realize a Local Storage. Furthermore, it constitutes the central point for interaction between the PMA components as well as for interaction between PMA components and the SIMPLI-CITY server components.

The PMA-based Sensor Abstraction provides the means for accessing car-based sensors as well as PMA-internal sensors (e.g., GPS). The correspondingly accessed data is forwarded to the Application Runtime Environment, where it can be used locally on the PMA, stored in the Local Storage of the PMA, or further forwarded to the SIMPLI-CITY system's server side component.

The App Marketplace constitutes a component with which the user directly interacts on the PMA, opposed to the other two components mentioned above (Application Runtime Environment and PMA-based Sensor Abstraction). The App Marketplace can be accessed by the user via the PMA to enhance his or her PMA's capabilities with new functionalities provided within SIMPLI-CITY in the form of apps. Thus, the App Marketplace provides the user with the capability to search for new apps, e.g., by simple browsing or by explicitly searching for a specific app. Afterwards, apps can be purchased and installed on the user's PMA.

The Multimodal Dialogue Interface is the second component with which the users directly interact. Thus, these two components explicitly constitute user interfaces on the PMA. The Dialogue Interface realizes the basic user interface layer within the SIMPLI-CITY PMA. The first basic task is to receive user inputs in form of voice utterances, process them and distribute them appropriately to the according software components, and respectively initiate appropriate reactions to the utterances. The second basic task of the Multimodal Dialogue Interface is to realize the other way round by offering the ability to "talk" to the user, which means that instructions to the user are uttered. Two different ways of basic interaction with the user are envisioned to be realized with the PMA within SIMPLI-CITY: One way of interaction is based on user queries, e.g., formulated as "questions to SIMPLI-CITY" and the second way will be constituted by allowing means of pro-active communication with the user, e.g., in order to actively call the user's attention to a changed and perhaps critical situation, like a traffic jam.

2.3 Software Developer Perspective

The Software Developer Perspective outlines the support provided by SIMPLI-CITY to software developers in order to allow them to easily develop, publish, and distribute mobility-related apps and services supporting the road user, respectively the road user information system, via the PMA. Three major SIMPLI-CITY components are explicitly targeting at software developer support. Specifically, these components are:

- Application Design Studio
- Service Development API
- Service Marketplace

The Application Design Studio constitutes an Integrated Development Environment (IDE) for app developers and provides them with step-by-step support during the whole app development process, finally allowing developers to deploy their developed apps. For this purpose, different tools are offered within the Application Design Studio. These tools comprise guideline documents as well as best practice and HowTo documents. Therefore, with the Application Design Studio an overall integrated tool for app development is offered to app developers, which explicitly describes and supports a development process for SIMPLI-CITY apps. This helps, for example, to keep a consistent look and feel as well as compliance to the user interface employed on SIMPLI-CITY's PMA. The documents are supported by code examples in the form of code snippets or comprehensive SIMPLI-CITY apps, which can be used by app developers as basis to realize their ideas for new SIMPLI-CITY apps. Furthermore, a supporting tool to create App Manifest files is provided. This tool helps developers to create a Manifest file, which specifically describes runtime environment requirements and other specifics for their app. Finally, compiling an app bundle on the basis of the Manifest file is supported by the Application Design Studio as well. The compiled app bundle contains all files needed by the developed app and thus constitutes the basis of a deployable version of the developed app, which can be provided to the marketplace.

The Service Development API focuses on developers, who want to create their own services in the context of SIMPLI-CITY, as opposed to the Application Design Studio, which aims at the development of SIMPLI-CITY apps. However, services constitute basic foundation blocks for SIMPLI-CITY apps, since they are used within the apps, e.g., to access external data sources or to provide means for data aggregation. Thus, services have to be seen as deeply interconnected with SIMPLI-CITY apps. For ease of creation and management of services, the Service Development API will be provided as a programmatic API. It will be accompanied by tutorials, guides, and examples. To keep developed services up to date and allowing them to find useful services, service developers are provided with means to modify and update information of developed services and configuration settings as well as to update or even delete their developed and published services themselves. Additionally, they will be provided with means for searching existing services. Furthermore, by realizing the approach of Mobility-related Data as a Service within SIMPLI-CITY, service developers get a simple access possibility for different data sources, e.g., provided by external data providers, and can thus easily make use of these data sources within their services.

The Service Marketplace allows service developers to publish and offer their developed services for free or for a defined price. Thus, an easy way to find services and use them,

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 15 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

for example in the context of developing new SIMPLI-CITY apps by app developers, is provided. However, service developers can themselves make use of the Service Marketplace to find appropriate services, which they can reuse for developing (composite) services. For convenient access, the Service Marketplace will be accessible via a web-based user interface. In order to guarantee the quality and usability of SIMPLI-CITY's Service Marketplace, services provided to the marketplace will not be immediately made accessible to the public, but rather have to pass through a review process first, which includes the execution of some test routines on the submitted services. Besides this review process, service consumers are also provided with means to rate services they used.

2.4 Data Integration Perspective

Within the app context of SIMPLI-CITY, data from different sources are provided and exploited in order to reach the goal of realizing the PMA-based "Road User Information System of the Future". Consequently, means for accessing and more specifically, for integrating, this data within the SIMPLI-CITY context are required. These means are highlighted within the Data Integration Perspective described in this section. The components explicitly taking care of this data integration are specifically:

- Sensor Abstraction and Interoperability Interfaces
- PMA-based Sensor Abstraction
- Data Processing
- Media Data Streams and Data Prefetching Logic
- Cloud-based Information Infrastructure

The Sensor Abstraction and Interoperability Interfaces provide a way to access sensor data originating from different, technologically heterogeneous sources. Within SIMPLI-CITY, it is expected to exploit different types of data sources. In particular, external sensing systems providing continuous measurements are differentiated into event-based sensing systems, which transmit data only in case a certain event occurred, as well as proprietary databases, which provide actual or historic sensor data. The Sensor Abstraction and Interoperability Interfaces will realize the possibility to access these heterogeneous data sources in a homogeneous way. Specifically, this means that data format translations between the Unified Data Model applied within SIMPLI-CITY and the heterogeneous data formats of the different sensor data sources will be provided. Furthermore, SIMPLI-CITY will provide means for querying these sensor data sources and simultaneously means for pushing received data from these sensor data sources in an event-based manner to the SIMPLI-CITY system.

The PMA-based Sensor Abstraction constitutes the mobile complement to the server-based Sensor Abstraction and Interoperability Interfaces. It also provides homogeneous access possibilities to heterogeneous sensor data. However, the focus of the PMA-based Sensor Abstraction is on access to sensor data sources available in the user's car. Thus, it is realized on the PMA, which provides a communication channel with the car-based sensors. In addition to the car-based sensors, sensors integrated in the PMA can be accessed via the PMA-based Sensor Abstraction, too. For this purpose, data format translations between the data formats employed in the different sensor sources and the common data format used within SIMPLI-CITY are provided within the PMA-based Sensor

Abstraction, similar to the data translation means of the Sensor Abstraction and Interoperability Interfaces.

The Data Processing component provides the basic access possibility to user centric data and Open Data/government data repositories. Furthermore, it offers the means for interpreting and enriching data received from different sources. These data sources are constituted within the SIMPLI-CITY context of the just mentioned user centric data, Open Data/government data, as well as sensor data and historical data. For further interpretation and corresponding data enrichment, the Data Processing component employs means for contextualization, which permits building a combined semantic structure of incoming data. Based on this semantic structure, the Data Processing component employs reasoning techniques, which allow conducting diagnosis and prediction on the received data pool. With these means, data can be cleaned, e.g., by filtering, enriched, e.g., by fusing, aggregating, correlating data, or condensed, e.g., by summarizing data.

The Media Data Streams and Data Prefetching Logic component provides the means to handle media streaming data within SIMPLI-CITY and according possibilities for prefetching data. In order to enhance the road user experience, the playback of media streams, e.g., music or video, might be necessary in the context of SIMPLI-CITY apps. In this context, the avoidance of interruptions of such playbacks is a basic necessity for a decent user experience. Thus, media buffering capabilities are provided which rely on prefetching the data which is most probably required in the actual context to allow an uninterrupted media playback even in case the user will be offline for a certain amount of time in the near future. Similar means are provided as well for prefetching services, which the user might need, respectively wants to use, in the near future. Such prefetching can, for example, be realized based on the analysis of actual context data of the user, like location, driving direction, etc.

The Cloud-based Information Infrastructure constitutes the central data storage component within SIMPLI-CITY. In the context of SIMPLI-CITY apps, these apps will exploit diverse available data. This data can either be provided dynamically, as, e.g., directly accessed by services from the corresponding data sources, or on the basis of data persisted in SIMPLI-CITY's data storage. This data storage is realized as the Cloud-based Information Infrastructure, which offers different storage possibilities for various data types, reflecting the heterogeneous data sources within SIMPLI-CITY. The access for storing data in this data storage as well as retrieving data from this data storage is realized as a service, consistent to SIMPLI-CITY's approach of Mobility-related Data as a Service. The data stored within the data storage may originate from SIMPLI-CITY apps, services, or external data sources, like sensors. Since sometimes, and for some apps or services, data has to be stored on the PMA as well, a correspondingly adapted Local Storage is additionally offered on the PMA.

3 General Description of Technical Specification

In Sections 5-8, different aspects of the technical specification for all SIMPLI-CITY software components are defined. As an introduction, the single aspects are explained in the following subsections. In Section 3.1, the actual aspects of the technical specification are covered, while Section 3.2 introduces the “Generic Comparison Criteria” which are applied as part of the technology comparison and selection for all SIMPLI-CITY software components.

3.1 Technical Specification

In the following subsections, a “blueprint” for the Technical Specifications of all SIMPLI-CITY software components will be introduced. All aspects from this template will be discussed in a corresponding subsection for the individual software components in Sections 5-8.

3.1.1 Major Design Decisions

The goal of this subsection is to briefly name the functionality and scope for the particular component as defined within the Functional Specification (deliverable D3.2.1). Second, this subsection discusses the major design decisions which provide the foundation for this component. These major design decisions are technology-independent, i.e., not determined by particular attributes of the technology to be selected. Quite to the contrary, these design decisions considerably influence the later technology comparison and selection.

Notably, the major design *decisions* as defined within a separate subsection for each SIMPLI-CITY software component should not be confused with the overall major design *principles*, which are applied for all SIMPLI-CITY software components (see Section 4).

3.1.2 Technology Comparison

First, this subsection recapitulates the technology comparison criteria as defined in deliverable D3.2.1 and extends the criteria specification by giving further insights on the meaning and importance of a particular parameter for a particular component. Importantly, only the “Specific Comparison Criteria” are discussed, as the “Generic Comparison Criteria” hold for all SIMPLI-CITY software components and are therefore explained separately in Section 3.2.

Second, this section presents the technologies / software frameworks to come into consideration for the actual technology selection. For this, the technologies are briefly named and introduced, including benefits and downfalls of a particular technology with respect to its intended usage within SIMPLI-CITY.

Third, the actual comparison of technologies for the software component at hand is presented. Hence, the discussed technologies are rated with regard to the defined comparison criteria. Please note that the actual technology selection is part of the next subsection.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 18 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

3.1.3 Technology Selection

This section has two different goals: First, the identification of the selected technologies based on the comparison conducted within the last subsection. For this, the reasons why a particular technology has been selected are explained. Second, this section includes a discussion of missing elements, i.e., subcomponents and functionalities needed within SIMPLI-CITY, but not provided by the chosen technologies. Third, implications of the technology selection with regard to the original functional specification are discussed.

3.1.4 Component Structure

While the structure for each SIMPLI-CITY software component has been drafted with the Functional Specification (deliverable D3.2.1), this structure has been technology-independent. Due to the selection of a particular technology, changes in the component structure arise that need to be depicted. In addition, in this subsection, the particular subcomponents of each software component are explained in more detail.

3.1.5 Interfaces

In a collaborative RTD project like SIMPLI-CITY, the introduction of well-defined interfaces between the major software components is of primary importance as means to align the work between the partners working on single components and make sure that the correct functionalities are provided by all components.

Hence, in this subsection, the focus is on the interfaces of the SIMPLI-CITY software component, i.e., which methods are provided to other components. For this, all interactions which have been identified during the definition of the Global Architecture (deliverable D3.1) and the Functional Specification (deliverable D3.2.1) are covered. In addition, new interactions that have been identified during the technical specification are also covered and according interfaces are provided.

In SIMPLI-CITY, software components either provide Java- or REpresentational State Transfer (REST)-based interfaces (see Section 4.3), depending on intended interactions. Therefore, for each software component, Java- or REST-based interfaces or a combination thereof are defined in-depth. This includes a definition of the parameters, return values, error handling, source code (class stubs), and message examples. A unified style is used to describe the interfaces.

3.1.6 Content Format

This subsection defines the data content formats necessary to make use of the component at hand. For this, according class signatures and data schemas are provided. Since for most of the SIMPLI-CITY software components JSON has been chosen as messaging format, in most cases this subsection states the defined JSON schemas along with some further explanations, e.g., design decisions.

3.1.7 Summary

The summary briefly recapitulates the most important aspects of the technical specification and planned implementation for the considered software component.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 19 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

3.2 Generic Comparison Criteria

As described in Section 3.1, for each SIMPLI-CITY software component, a technology comparison and selection is conducted. As foundation for this, Generic and Specific Comparison Criteria are used. As the terms imply, the Generic Comparison Criteria are the same for all SIMPLI-CITY software components, but their importance (and therefore weight during technology selection) may differ. Table 1 shows the Generic Comparison Criteria; the importance ratings in this overview are only illustrative examples, but the descriptions are valid for all SIMPLI-CITY software components.

Table 1: Generic Comparison Criteria

Parameter	Importance	Description
Up-to-Datedness	--+	In general, it is the aim of the consortium to apply and extend cutting-edge technologies. This will allow the consortium to create more innovative solutions which will subsequently attract attention during the project dissemination and exploitation
Stability	+	The technologies to be chosen should be stable and mature, i.e., preferably, not in beta status. This is especially crucial if a product is chosen which is not regularly updated.
Extensibility & Open Source/Standards	+	<p>Even if an existing technology can be used as a foundation for a particular component, it is very likely that the technology needs to be extended in order to meet all the project requirements. As such, open source solutions are preferable. However, as defined in the Consortium Agreement, software provided under an infecting license such as GNU General Public License (GPL) should be avoided by all means. As an alternative to a good quality open source solution, a technology with well-defined extensibility, e.g., in terms of plugin mechanisms, may be considered.</p> <p>Open standards are supported by both open source and proprietary solutions. Furthermore, especially in the Internet of Services and Semantic Web domains, a number of open standards exist which could ease the implementation efforts in SIMPLI-CITY.</p>

Familiarity	+	Project partners may be familiar with particular technologies, either because they have been developed in another (EU) research project or as a commercial solution by a partner, or have been applied by the partner in the past. In any case, familiarity with an existing technology substantially decreases the implementation efforts of the partners.
Performance	++	Performance may have different meanings regarding the different SIMPLI-CITY components, e.g., runtime performance, provided Quality of Service (QoS), provided Quality of Experience (QoE), data/result quality, scalability, or accuracy.
Interoperability	+	If possible, technologies should be preferred which provide a certain degree of interoperability, e.g., by providing open interfaces, a well-defined API, or standardized data formats.

4 Major Design Principles

In the following subsections, major design principles for SIMPLI-CITY are presented. These are design decisions that concern several software components. This includes the usage of the Cloud-based Information Infrastructure as foundation for all data storages within the SIMPLI-CITY system (Section 4.1), the general security and privacy concept which needs to be taken into account by all developers (Section 4.2), the provision of Java-based and/or REST-based interfaces for communication between the single software components of the SIMPLI-CITY system (Section 4.3), the usage of the JavaScript Object Notation (JSON) as format for messaging between data sources and service as well as services and apps (Section 4.4), and finally the usage of the Resource Description Framework (RDF) for data analysis and processing purposes (Section 4.5). In addition, the choice of Android as underlying mobile Operating System (OS) for the PMA will be briefly discussed in Section 7.1.

4.1 Data Storage

Since SIMPLI-CITY provides the Cloud-based Information Infrastructure, there is no need to use any further databases apart from very special reasons, e.g., the usage of an in-memory database for testing purposes or for performance reasons.

The SIMPLI-CITY Cloud-based Information Infrastructure allows the storage of different data types; four types of data are to be distinguished:

- Private data or small key-value pairs, which will be stored locally with the help of a Local Key Storage.
- Semantic data, which will be stored with the help of the Cloud Storage in a semantic store database.
- Binary data, which will be stored with the help of the Cloud Storage in a file storage.
- Semi-structured data, which will be stored with the help of the Cloud Storage in a NoSQL (Not only SQL) database.

While the Local Key Storage allows the storage of data within the PMA on the mobile device, the Cloud Storage is offered as a Cloud-based online storage or database, respectively. In these cases the data is sent to the Cloud Storage with the help of a RESTful Web Service. The data will be saved in compatible Buckets based on the type of data to be stored. A Bucket is an isolated storage space managing data. A Bucket may contain multiple data entries, which may be created, read, updated or deleted. In SIMPLI-CITY, each component may create multiple Buckets for data storage, which are fully separated and not influenced by activities of other Buckets.

The owner of a Bucket is able to grant and restrict access to his Buckets to other users of the Cloud Storage. Users who are allowed to at least read the data of a Bucket can be notified when changes to a Bucket occur. The management of the Buckets and databases is administered transparent to the users.

4.2 Security and Privacy

As a distributed platform, SIMPLI-CITY deals with user data within different physical locations. User interaction takes place at the PMA Side and may in many cases involve

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 22 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

the transfer of user-specific data to the Server Side in order to make use of services. As such, security and privacy aspects are an important aspect of the project. In order to cope with them, the SIMPLI-CITY consortium has created a set of concepts within deliverable D3.3 (Holistic Security and Privacy Concept). It describes security, privacy and also ethical aspects and lists the key action points for WP4-8.

Among a large amount of smaller decisions, deliverable D3.3 defines the following key concepts for the project:

- Whenever possible, users will be able to make use of SIMPLI-CITY without the need to transfer any personal data outside the PMA.
- If a user has to be identified, then SIMPLI-CITY will use a unique ID for a user, i.e., a Universally Unique Identifier (UUID) (pseudonymization). This allows apps and services to recognize a user without being able to identify his or her real identity. A single unique UUID could lead to a security problem in case that the UUID was used to correlate data from different services and then subvert (to some degree) the pseudonym. In order to avoid this, SIMPLI-CITY will return a separate UUID *per app* so that each app of the PMA will have its own UUID.
- The OpenID¹ standard will be supported for any web UI created by the project. This includes OpenID support for developers when accessing the Services and App Marketplaces.
- Messages exchanged between different parts of SIMPLI-CITY may be signed by making use of the RSA algorithm in combination with SHA-256. This is especially useful when considering the communication between apps and services.
- End-to-end encryption is supported by SIMPLI-CITY by allowing the transfer of encrypted information, which will be embedded in the payload of a service call.
- The Advanced Encryption Standard (AES) will be supported for symmetric encryption of data in combination with the aforementioned UUID.

4.3 Java-based and RESTful Interfaces

SIMPLI-CITY provides a distributed system with *high cohesion*, i.e., single software components encapsulate a particular, well-defined functionality and the degree of redundancies is limited, and *loose coupling*, i.e., the single software components are independent from each other and could also be used in a different environment or be exchanged within SIMPLI-CITY. As a result, the single software components need to communicate through interfaces.

Within SIMPLI-CITY, this is achieved through Java- and REST-based interfaces. The usage of Java-based interfaces is an obvious choice, since most of the SIMPLI-CITY software components are implemented using Java and the necessary interfaces are actually an outcome of these implementations. Hence, whenever possible, Java interfaces will be used, because it provides a direct, lightweight interaction between Java-based software components running in the same JVM (Java Virtual Machine).

However, there are different cases where direct interaction using Java interfaces is not possible. It is self-evident that this applies if a different programming language is used for the implementation of a software component, as it is the case for, e.g., the Cloud-based Information Infrastructure (see Section 5.2). Furthermore, because of the distributed

¹ <http://openid.net>

nature of SIMPLI-CITY, software components will be spread amongst different machines, including small sensor nodes, the SIMPLI-CITY Service Runtime Environment (see Section 6.1), and naturally the Personal Mobility Assistant (see Section 7.1). In such cases, direct interaction through Java interfaces is not possible and a different solution is necessary, which facilitates interaction between remote software components and machines.

Within SIMPLI-CITY, interaction between remote software components will be done using RESTful web services. RESTful services provide remote access to functionalities using common HTTP methods (e.g., GET, PUT, POST, or DELETE). An alternative would be the usage of the SOAP protocol, which also allows exchanging messages (structured information) between software components in a distributed manner. SOAP makes use of the so-called WS-* stack for service implementations and is therefore based on a well-defined set of standardized protocols and technologies. However, SOAP provides a verbose messaging format and RESTful web services currently have much more momentum and have gained high popularity in practice.

Hence, in SIMPLI-CITY, RESTful web services will provide the interface to software components whenever it is not possible to make use of Java interfaces and/or functionalities may be invoked remotely. Notably, RESTful web services do not feature a standardized description and messaging format, which makes it necessary to strictly define the message data formats. In SIMPLI-CITY, JSON will be used for this (see next section) and the messaging format will be part of the component specifications in Sections 5-8.

Software components may offer both Java-based interfaces and RESTful web services. This is especially useful if a software component will be used both locally by other software components as well as remotely.

4.4 JavaScript Object Notation

Due to the nature of a distributed system like SIMPLI-CITY, a lot of data has to be exchanged between different systems and software components. In a first step, data has to be serialized to enable the transmission between two systems, e.g., the PMA and the Service Runtime Environment. Furthermore, both sides need to know how to handle the object that contains the information. Following this, a common markup for object serialization has to be defined. A possible data format that is suitable for this purpose is JSON. This data format is very simple, lightweight and in addition it is easily readable for humans. Alternatives would be XML (Extensible Markup Language) or YAML (YAML Ain't Markup Language), which, however, are quite heavyweight and therefore require relatively large efforts for parsing and serializing (XML), or need specific libraries to be installed on both the PMA and the server (YAML). Through the use of JSON this is not required. Particular benefits of JSON are:

- Human readable structure
- Easy-structured and resource-efficient
- Directly available for many programming languages
- Possibility of direct translation to Java objects and vice versa

Although JSON can be directly interpreted by Java, a custom parser should be used to restrict the input parameters of the JSON objects and prevent potential security breaches.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 24 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

As mentioned above, JSON is more efficient in the use of memory in comparison to XML that has a larger overhead when it describes attributes of objects. In a system that includes battery-powered devices, e.g., the PMA, which also has to deal with network connections with limited bandwidth, e.g., 3G, the reduction of the amount of data that has to be transmitted is worthwhile. Deliverable D4.1.1 gives an overview how JSON will be used for data exchange within SIMPLI-CITY.

4.5 Resource Description Framework

In order to fulfil the project and use case requirements of combining the different data sources for contextualization, diagnosis and prediction, SIMPLI-CITY has to provide a robust, well-formed and scalable approach in providing these solutions.

In order to achieve this, SIMPLI-CITY will make use of Semantic Web technologies, i.e., semantic vocabularies and ontologies in the Data Processing component. This allows a common semantic schema to be generated into which all of the different datasets can be loaded. This schema makes use of RDF, which is a standard in semantic data structuring and processing. The benefit of the RDF structure is that it allows data elements from different datasets to be related together (contextualization) and consequently reasoned over. Specifically in relation to SIMPLI-CITY, RDF facilitates contextualizing data both spatially and temporally. RDF enables the ability to query the structured data in order to do diagnosis and prediction in a way which would be very difficult to achieve across heterogeneous datasets. In this way the SIMPLI-CITY Unified Data Model is also fulfilled.

The implications of using RDF are:

- A deeper level of data manipulation and reasoning is possible.
- Structured data can be returned to service requests to allow developers to do further analysis.
- Services which do not want to return or process RDF require a data mapping from RDF to JSON. For this, a generic data mapping will be provided to service developers.

In conclusion using RDF enables some major requirements of SIMPLI-CITY to be fulfilled with the consequence that a transformation of RDF into JSON will be facilitated in the Service Runtime Environment for services which do not want to directly use RDF.

Notably, both RDF and JSON (see Section 4.4) are applied as *data* formats in SIMPLI-CITY. While the Data Processing component offers RDF-based data, other components including the Sensor Abstraction and Interoperability Interfaces exclusively offer JSON-based data. Data Processing will also transform sensor and user data into RDF to allow contextualization etc. The selection of one of these data formats is based on the respective intended use of the data from a particular data source. For instance, in case of direct sensor data access (i.e., not through the Data Processing component), there is no need for (heavyweight) semantics, and therefore processing overhead. In any case, JSON is the only applied *messaging* format within SIMPLI-CITY. Even RDF data will be serialized in JSON for messaging purposes. However, this will maintain the semantic structure of the data.

5 Technical Specification: Data Integration

5.1 Data Processing

5.1.1 Major Design Decisions

The Data Processing component is a key component in the SIMPLI-CITY system as (except for media data streams) it acts on and can combine all the distinct data source types available to the SIMPLI-CITY project. These are the user personal and Open Data sources as well as the sensor data available through the Sensor Abstraction and Interoperability Interfaces from fixed sensors (see Section 5.3) and the PMA (see Section 7.3). This data can be static in some cases and streaming data in other cases. Therefore, a key decision is how to process and combine the input from all of these sources which can be all updated at different frequencies and are provided via different protocols and formats (heterogeneous datasets).

An essential feature of the Data Processing component is its facility to contextualize and reason over these heterogeneous datasets. This means the ability to link and cross-reference data from different datasets, e.g., traffic and weather information over time and space (contextualization) and the ability to infer information, e.g., if weather is bad at point A then traffic is bad at point B (Reasoning). This is a key requirement of SIMPLI-CITY. From this perspective of the Data Processing component, it must be able to combine all of these datasets in a meaningful way. The Data Processing component also interacts with the Cloud Storage Information Infrastructure (see Section 5.2) to both store and retrieve historical data.

Semantic Structure:

The first major design decision for handling this combining of datasets is to make use of semantic vocabularies and ontologies so that a common semantic schema is available into which the datasets can be loaded and consequently allowing contextualization of data. This in turn allows reasoning to happen over the combined datasets. This mechanism fulfils the SIMPLI-CITY requirement for a Unified Data Model. This decision allows us to make use of a robust structured semantic graph in the form of the Resource Description Framework (RDF). In this way, it is possible to provide diagnosis and/or prediction as required by the SIMPLI-CITY use cases and services.

Data Retrieval:

The second major design decision is to have a separate subcomponent called Data Retrieval which handles the data access and fetching for the data sources. As Open Data sources are fetched within the Data Processing component, a further decision is the initial transformation of fetched data into a Comma-Separated Values (CSV) format.

All of the fetched datasets are loaded into the Contextualization subcomponent where stream-handling software operates on the datasets to transform them into the semantic schemas.

REST-Based Requests:

As the Data Processing component will be hosted on separate servers (both physically and geographically), the local method invocation of Data Processing facilities is not

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 26 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

possible. Keeping in line with other SIMPLI-CITY components, the third major design decision in conjunction with the other components is to provide a REST-based service to other components so that they can make remote requests to the Data Processing component.

5.1.2 Technology Comparison

5.1.2.1 Comparison Criteria

The below table is a summary of criteria for the Data Processing component, however not all criteria apply to each technology comparison within the component.

Table 2: Specific Criteria for Technical Specification

Parameter	Importance	Description
Scalability	+	<p>The technology selected should have the ability to scale as the volume of user requests increases, as datasets are added so that the system is flexible enough to cater for this in the Data Processing subcomponents.</p> <p>This criterion is used for the comparison of stream-handling software and the needed Web application server.</p>
Asynchronous Request/Data Handling	++	<p>The solution should be able to process requests, dataset retrieval and further processing in parallel or asynchronously to help performance so that the Data Processing component is not sequentially doing data-fetches and processing.</p> <p>This criterion is used for the comparison of stream-handling software and the needed Web application server.</p>
Querying Mechanism over Combined Structured Data	++	<p>The selected format for the datasets transformed into the SIMPLI-CITY Unified Data Model (as defined in deliverable D4.1.1) should allow standard ways to exercise queries over this combined structured data.</p> <p>This criterion is used for the comparison of Reasoning software.</p>
Expressivity	+	<p>The Reasoning software should be able to make use of the expressivity of the underlying semantic data. This means that the available richness of the semantic data content is utilised in the reasoning software allowing finer grained results.</p>
Usage of Rules	++	<p>The Reasoning software should make use of rules and regeneration of rules in synch with data collection over time which are then applied to the decision making process for requests to the Reasoning software.</p>

5.1.2.2 Possible Technologies and Comparison

In the following, some technologies such as Semantic Web, Stream software, contextualization and reasoning software which will be used in the Data Processing component are briefly introduced. Afterwards, alternatives in each category are compared and one particular technology is selected as foundation within SIMPLI-CITY.

Semantic reasoning over *heterogeneous* data sets makes use of Semantic Web technologies which are well defined in the W3C standards and make use of, e.g., the OWL 2.0² (Web Ontology Language) standard. OWL 2.0 provides a means to define the vocabulary and ontologies for heterogeneous datasets, e.g., Open Data such as traffic and weather information, as well as for sensor and user personal data.

The Semantic Web technologies help to enable automatic understanding and response to complex requests over varied information sources, i.e., relating elements of different datasets to each other (*Contextualization*), e.g., relating a weather and traffic event to each other and also inferring additional information from the combined datasets (*Reasoning*), e.g., when location A is flooded, location B has blocked traffic. This requires that the information sources (data) are semantically structured or transformed into such a structure. To this end, OWL 2.0 ontologies are used to describe the underlying semantic of the SIMPLI-CITY data sources. OWL 2.0 offers description languages which are expressed in RDF. With RDF, dataset merging is possible even if the dataset schemas are different.

Stream-handling software has become a big topic in data processing due to ever increasing numbers of available data source and increases in data update frequencies. In addition to this, there is a need to classify and extract relevant information from the data in real-time or near real-time. Tailored solutions for specific data-feed types have been implemented dealing with, e.g., financial data feeds. In the SIMPLI-CITY project, data from several sources is exploited, e.g., Open Data, user personal data, sensor data etc. The required datasets can have several sources in each category and can be static or streaming. To facilitate all of this incoming data, a robust, mature and scalable stream-handling software will be used. This software will facilitate the transformation into the SIMPLI-CITY Unified Data Model, which is described in more detail in deliverable D4.1.1.

5.1.2.3 Stream-Handling Software Comparison

A brief introduction to some stream-handling solutions is given and a comparison table follows.

IBM InfoSphere Streams:

IBM's InfoSphere Streams³ (IIS) is part of IBM's big data platform. IBM InfoSphere Streams is a high-performance computing platform able to ingest, analyse and correlate large amounts of streaming data. IIS allows plugin operators in Java and C++ to handle particular data streams.

² <http://www.w3.org/TR/owl2-overview/>

³ <http://www-03.ibm.com/software/products/us/en/infosphere-streams/>

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 28 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

Storm:

Storm⁴ is an open source stream handler developed by Twitter. The three main features of Storm are stream processing, Remote Procedure Calls (RPC) for distributed processing and continuous computation. Storm is not able to run algorithms which require historical knowledge of the data. Storm does not operate on static data.

Apache S4:

Apache S4⁵ describes itself as a general-purpose, distributed, scalable, fault-tolerant, pluggable platform that allows programmers to easily develop applications for processing continuous unbounded streams of data. It is a recent offering at version 0.6.0 based on what was formerly Yahoo S4 and therefore does look like a complete and stable enough solution.

Table 3: Comparison of Technologies for Stream Handling Software

Parameter	Importance	Infosphere Streams	Storm	Apache S4
Generic Criteria				
Up-to-Datedness	++	10	10	10
Stability	++	10	7	6
Extensibility & Open Source/Standards	-	8	6	8
Familiarity	++	10	4	4
Performance	++	9	6	4
Interoperability	++	10	8	4
License		Commercial	EPL 1.0	Apache 2.0
Specific Criteria				
Scalability	+	10	10	4
Asynchronous Request/Data Handling	++	10	10	4

5.1.2.4 Reasoning Software Comparison

A brief introduction to some reasoning software solutions is given and a comparison table follows.

Pellet:

Pellet⁶ is open source reasoning software for OWL 2 DL. It provides both standard and up-to-date reasoning services. It is fully W3C conformant and allows rule generation which is a particular requirement for the reasoning software.

TrOWL:

TrOWL⁷ is a 'tractable' reasoning infrastructure for OWL 2. TrOWL supports standard TBox and ABox reasoning and also conjunctive query answering in SPARQL. TrOWL is a good candidate however currently TrOWL does not support rule generation. Rule generation is planned to be added in the future but there is no timeline available currently for this.

⁴ <http://storm-project.net/>

⁵ www.stanford.edu/class/cs347/reading/S4PaperV2.pdf

⁶ <http://pellet.owldl.com/papers/sirin05pellet.pdf>

⁷ <http://trowl.eu/about/>

CEL:

CEL⁸ is an open source for OWL 2 EL. It uses polynomial-time classifiers. It has worked well for biomedical ontologies. It does not support nominals and datatypes/values and is quite domain-specific for the biomedical domain.

Apache Jena:

Apache Jena⁹ is a Java developer framework for developing Semantic Web applications. It uses a rule-based inference engine for reasoning with RDF and OWL data sources. In some initial tests with this software, some bugs in the rule generation aspect were encountered which means it is not currently a good candidate for Reasoning software.

Elk:

Elk¹⁰ is an open source reasoner for OWL EL that has shown good performance for life science ontologies. It does not support keys, datatype reasoning and has limited support for nominals/data.

FaCT++:

FaCT++¹¹ is an open source OWL 2 DL reasoner implemented in C++. It has good performance on expressive ontologies. One major drawback is that it currently only partially supports OWL 2 DL standard.

Table 4: Comparison of Technologies for Reasoning Software

Parameter	Importance	Trowl	Pellet	CEL	ELK	Jena	Fact++
Generic Criteria							
Up-to-Datedness	++	10	10	7	7	10	10
Stability	++	10	10	8	4	10	8
Extensibility & Open Source/Standards	-	8	10	8	8	10	6
Familiarity	++	10	10	8	4	10	6
Performance	++	10	10	8	10	10	10
Interoperability	++	10	10	6	6	10	6
License		AGPL3	GPL3	Apache 2.0	Apache 2.0	Apache 2.0	GPL3
Specific Criteria							
Querying Mechanism over Combined Structured Data	++	10	10	8	8	10	8
Expressivity	+	10	10	8	8	6	10
Rule Generation	++	0	10	6	0	6	0

5.1.2.5 Web Application Server Comparison

There are many solutions available for web application servers and a few examples are listed here.

IBM WAS:

IBM's WebSphere Application Server¹² (IBM WAS) is a stable robust scalable, multi-threaded and up-to-date and familiar offering. It is a standard and familiar tool for exposing REST APIs which will be needed to facilitate the SIMPLI-CITY server requests made to the Data Processing component hosted remotely at IBM.

⁸ <http://lat.inf.tu-dresden.de/systems/cel/>

⁹ <http://jena.apache.org/>

¹⁰ <http://www.cs.ox.ac.uk/isg/tools/ELK/>

¹¹ <http://owl.man.ac.uk/factplusplus/>

¹² <http://www-03.ibm.com/software/products/us/en/appserv-was/>

Apache TomEE Server:

Apache TomEE Server¹³ is the Java enterprise edition of Apache Tomcat. It combines several Java EE projects. In particular it now includes support for REST services, however this is a recently added feature and therefore the stability and familiarity are potential issues.

Oracle GlassFish Server:

GlassFish¹⁴ was an open source project initiated under Sun Microsystems. Oracle have since released a version 4 in June 2013 with Java EE 7 support. Since this is a recent release of updated software, the stability and lack of familiarity are again potential issues in the development phase.

JBoss:

The current stable version of JBoss¹⁵ AS 7.1 was released in February 2012 and therefore should be a reasonably stable offering, however the Java EE profile is not fully implemented which is a drawback for the offering.

Table 5: Comparison of Technologies for Web Application Server

Parameter	Importance	WebSphere	TomEE	Glasfish	JBOSS
Generic Criteria					
Up-to-Datedness	++	10	10	10	8
Stability	++	10	7	6	8
Extensibility & Open Source/Standards	-	8	6	8	4
Familiarity	++	10	6	4	4
Performance	++	9	6	4	8
Interoperability	++	10	8	4	8
License		Proprietary	Apache 2.0	CCDL/GPL	LGPL
Specific Criteria					
Scalability	+	10	6	4	6
Asynchronous Request/Data Handling	++	10	8	4	6

5.1.3 Technology Selection

The selection of the technologies compared in the previous section is explained below.

5.1.3.1 Selection for Stream-Handling Software

The selected software for stream handling is IBM InfoSphere Streams. It is very well suited to applications which need to process high volumes of data, to compute high complexity operation and with low latency. It is a stable robust solution which facilitates Java and C++ plugins as stream operators. It is scalable and automatically manages distributing stream processing jobs to the available processing nodes. From the Data Processing component implementation perspective using IBM InfoSphere Streams will give a strong and familiar platform to ingest and transform SIMPLI-CITY's heterogeneous datasets.

¹³ <http://tomee.apache.org/enterprise-tomcat.html>

¹⁴ <https://glassfish.java.net/>

¹⁵ <http://www.jboss.org/overview/>

5.1.3.2 Selection for Reasoning Software

The Pellet reasoning solution is the main candidate for reasoning software due to its ability to generate and handle rules. Its query engine can be accessed through a command line interface which fits well with the Data Processing model. It also plugs into the Jena framework which is a preferred development environment for this component.

TrOWL is a close second candidate due to having better scalability. However, TrOWL does not currently handle rules. Rule handling is planned to be added to TrOWL and if this happens in a relevant timeframe with a stable implementation, TrOWL would become the preferred reasoning software.

5.1.3.3 Selection for Web Service Application Server

IBM's WebSphere Application Server is the selected software here for handling the REST interfaces associated with the Data Processing component. It is a stable robust scalable, multi-threaded and up-to-date offering with which the developers of the Data Processing component are very familiar. For handling the incoming service requests and REST communication between SIMPLI-CITY components interacting with the Data Processing component, this is a complete solution. Additionally, there is no incompatibility with the web service application chosen by other SIMPLI-CITY components and as this component is hosted on IBM servers, there is no licensing issue.

5.1.3.4 Missing Elements and Implementation Needs

Implementation requirements for the Data Processing component are outlined below:

Data Retrieval:

The modules and frequency handlers need to be put in place which cover all of the protocols and formats for retrieving data from the necessary SIMPLI-CITY data sources (Open Data, sensor data, user personal data, semantic historical data) to fulfill the SIMPLI-CITY use cases. Further, it is necessary to implement the interaction with the SIMPLI-CITY Cloud-based Information Infrastructure to fetch and store historical data.

Contextualization:

The necessary mapping and operators for each required dataset to the semantic structure need to be implemented including the interaction with the SIMPLI-CITY Cloud-based Information Infrastructure for storing transformed data.

Reasoning:

The reasoning tool implementation needs to be on-going to uncover and tune rule generation and inferences over the transformed datasets.

Service Request Handler:

As the Data Processing component is hosted at a different physical location to the SIMPLI-CITY server, it is necessary to provide REST call interfaces and interactions with the other SIMPLI-CITY components. In particular all request handling between the SIMPLI-CITY server and the Data Processing component needs to be implemented.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 32 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

Result Handler:

This subcomponent needs to fully implement the interaction between the outcomes of requests to the Data Processing component and the modules which handle storing results in the Cloud-based Information Infrastructure.

Subcomponent Interactions:

All Data Processing subcomponent interactions need to be implemented, tested, tuned and re-assessed for acceptable performance and response times.

Security and Privacy:

All Security and privacy recommendations defined in the SIMPLI-CITY deliverable D3.3 (Holistic Security and Privacy Concept) need to be implemented, including, e.g., format of user IDs, secure transfer of data with credentials over https.

Encryption:

Encryption can be handled on a basic level by using standard Linux (which has been chosen as the server operating system for the SIMPLI-CITY server) encryption libraries and results can be returned over secure http. The details of this have been defined in the Holistic Security and Privacy Concept (deliverable D3.3). Any other encryption techniques required by a particular service would need to be detailed and implemented.

Static Data ID Mappings:

Some static data in the project may need to be catalogued, i.e., given identifiers or acronyms to be used in message exchange between components. For example, when exchanging a list of Open Data sources, it is necessary to be able to map the requesting data source ID, name or acronym to an actual data source.

5.1.3.5 Further Information and Conclusion from Technology Comparison

The Data Processing component makes use of familiar mature technology for the standard infrastructure of the component, e.g., data retrieval using Perl and bash in Linux, WebSphere Application Server for the REST API and request handling mechanism.

For the key feature of Unified Data Model and the Contextualization and Reasoning facilities, the Data Processing component makes use of the latest Semantic Web technologies. Ontologies and vocabularies combine static and stream data into a structured semantic tree (RDF) and rule generation techniques facilitate reasoning.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 33 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

5.1.4 Component Structure

There follows a diagram showing the component structure for the Data Processing component and then a description of each subcomponents shown in the diagram.

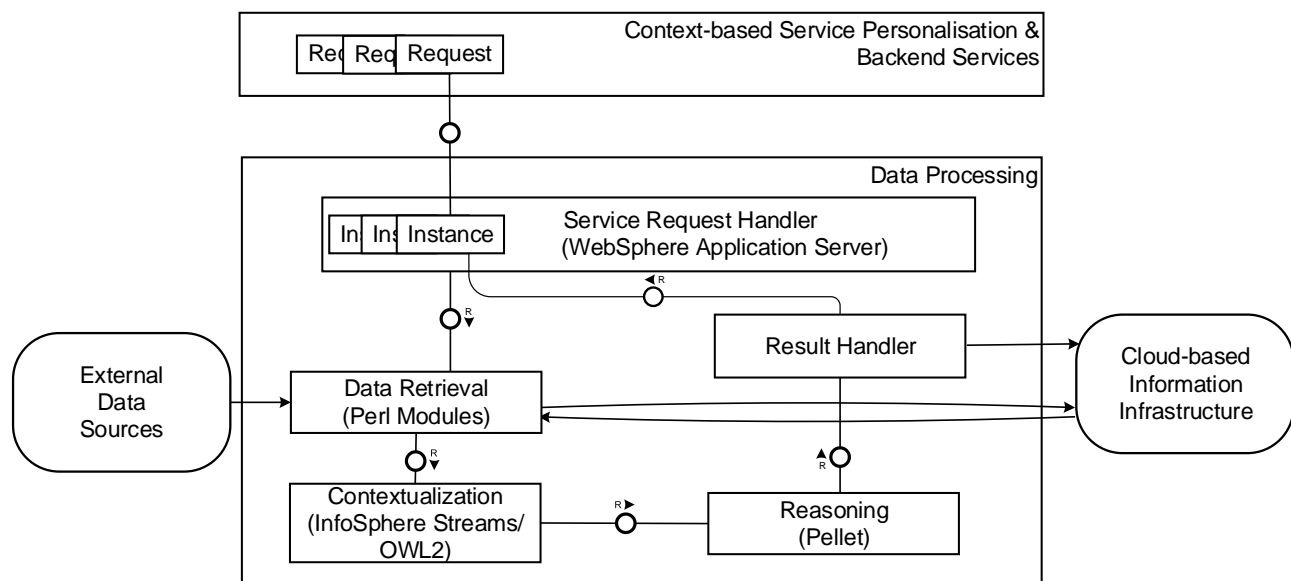


Figure 4: Component Structure of Data Processing

Service Request Handler:

The Service Request Handler subcomponent manages incoming requests from the SIMPLI-CITY backend services and the server-side Context-based Service Personalisation component. The requests are made to the Web Application Server which will process RESTful requests. As selected above, the IBM WebSphere Application Server solution (IBM WAS) will be used. This will be configured to automatically process incoming requests and pass them to the relevant handling software which is in this case a Web Application archive (WAR) file deployed into the IBM WAS server.

The deployed WAR file application will parse the incoming parameters in the REST query and react accordingly. It will pass these parameters such as data sources to be used and frequency of update to the Data Retrieval sub-component to retrieve the relevant SIMPLI-CITY data-sources. This behaviour is the same as envisioned in the SIMPLI-CITY Functional Specification (deliverable D3.2.1).

Data Retrieval:

The Data Retrieval subcomponent is responsible for collection of data from Open Data sources at frequencies specified by the SIMPLI-CITY backend and data services. The SIMPLI-CITY Open Data sources are heterogeneous, have different frequency of updates and can be static or streaming data. The access to these data sources requires using different protocols such as SFTP, SOAP, HTTP and processing different data formats such as WSDL, CSV, HTML and PDF.

This Open Data collection is an on-going process and will be implemented using a combination of Perl scripting and bash script wrappers running on Linux. The Perl scripting language and bash shell scripting are utilities available on the Linux operating system and use is made of these facilities for data collection. The Perl scripts will use modules to assist

access and processing for, e.g., SOAP, HTTP, WSDL. All of these datasets are transformed by the Perl modules into CSV format for consumption by the stream-handling software. The bash scripts are used to facilitate the frequency at which particular datasets are called and to set common attributes of a particular set of data fetches such as, e.g., timestamp. The Open Data model is described in more detail in the SIMPLI-CITY D4.1.1 deliverable.

The Data Retrieval subcomponent also interfaces with other SIMPLI-CITY components such as the Sensor Abstraction and Interoperability Interfaces for sensor data including user-personal data and with the Cloud Storage Infrastructure for semantic historical data fetch and store.

The interaction with the above components is specified in their respective descriptions to be via REST API calls. The Data Retrieval subcomponent will process calls from these components using the Perl module structure based on the parameters received in the service request and as defined by the RESTful interface descriptions given in Section 5.1.5.

Contextualization:

The Contextualization subcomponent comprises the stream-handling software, the vocabularies and ontologies defined for the data sources and the operators defined to transform those datasources. The operators process the fetched raw data and by using the defined ontologies transform the data into RDF format for contextualization.

The *IBM InfoSphere Streams* (IIS) software which was chosen for this subcomponent combines all of these required functionalities into one solution. IIS provides the facility to be aware of new incoming data for the defined sources and operating on them to transform them from raw data into the relevant schema in RDF format.

The ontologies should be general enough to cater for adding new datasources. However, the operators will likely need to be updated or a new operator written to cover any new schema mapping requirements.

Reasoning:

The Reasoning subcomponent will be invoked subsequent to the contextualization phase of the Data Processing component if the particular service request has specified that it wants to use reasoning facilities such as diagnosis or prediction. The Reasoning software will act on the combined set of data required by the service request. This can include any of the SIMPLI-CITY datasets (user personal, vehicle-sensor, Open Data, historical data) which at this stage of the request have been transformed using the Unified Data Model into a common format (RDF) which can be reasoned over.

In the technology selection the Pellet solution was selected with TrOWL coming in a close second. Pellet will learn and apply rules to the unified data and regenerate rules based on new data.

Result Handler:

The Result Handler subcomponent provides different functions. The first is to notify the Service Request Handler subcomponent of a result of a request and the second is to call the Cloud Storage API of the Cloud-based Information Infrastructure in order to store any

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 35 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

updated semantic historical information. The third function is to execute any encryption which may be required on the result data.

This is achieved in the first case by executing the required notification to the Service Request Handler, e.g., by passing *RDF* output. In the second case, this is achieved by making a call to execute the *Perl* module for interacting with the Cloud-based Information Infrastructure. Data encryption will be achieved based on the service request specification and implemented encryption mechanisms.

5.1.5 Interfaces

The following sections describe the offered interfaces to the SIMPLI-CITY Data Processing component. As described previously, the Data Processing component is exposed to other SIMPLI-CITY components via RESTful interfaces; Java-based interfaces will not be provided. The interfaces use the REST context combined with a JSON object specific to that kind of request. The REST context is also shown in the JSON object as the “requestType” entry. The full JSON schema for the Data Processing component is given in Section 5.1.6 (Content Format). Specifically, the API is invoked by using *HTTP Post* with the JSON object to the Data Processing base URL (still to be specified) with the particular context, i.e., with example base URL <http://www.simpli-city.eu/data-processing/>.

5.1.5.1 Data Processing – Get Open Data Source Current Data

This API call returns the latest fetched values from a particular data source. The raw data is transformed to CSV format for later consumption by the contextualizing component. The data in CSV format is pushed back to the requesting service using HTTP PUT. The location and name of the returned CSV data file is passed back to the service via a JSON object as shown in the listing below. A serialization of the csv data into a JSON object is also possible, however it should be noted that the serialization would be specific to the particular data source. The serialization would convert the lines of CSV data into a JSON array. In this case the filename passed back in the requestResponse JSON object would have the extension “.json” and the format type would be JSON.

The call to the data Processing component is made to the base URL with context *getOpenDataCurrentData*. The table below shows the corresponding RESTful interface description.

Table 6: RESTful Interface Description – Get Open Data Source Current Data

Method	POST	URL	\$API_ROOT/simplicity/dataProcessing/getOpenDataCurrentData				
Description	Request the most current data for an open data source - post data source id as part of JSON object						
Parameter	none	Required	no	Possible Values	Description no parameters		
JSON Object	\$API_ROOT/simplicity/dataProcessing/JSON-Schema/requestOpenData						
JSON Attribute	requestType	Required	yes	Possible Values	getOpenDataCurrentDat	Description	context
JSON Attribute	requestId	Required	yes	Possible Values	set by SRE	Description	
JSON Attribute	serviceld	Required	yes	Possible Values	set by SRE	Description	
JSON Attribute	userId	Required	yes	Possible Values	UUID	Description	
JSON Attribute	dateSourceId	Required	yes	Possible Values	string	Description	
JSON Attribute	dataType	Required	yes	Possible Values	raw	Description	
Example URL	\$API_ROOT/simplicity/dataProcessing/getOpenDataCurrentData						
Response	HTTP PUT of output CSV datafile followed by HTTP status code response + JSON object with data filename/location or Cloud Storage location						
HTTP Status Code		Required	yes	Possible Values	200 400 401 500 501	Description	200: OK 400: Bad Request: Invalid elements of JSON object 401: Unauthorised. 500: Server Error 501: Not Implemented.
JSON Object	\$API_ROOT/simplicity/dataProcessing/JSON-Schema/requestResponse						
Example Response	HTTP status + JSON object requestResponse - see examples in the following listings						

The JSON object which is sent as part of the RESTful request is constructed as per the example message below.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 36 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

Listing 1: Message-Example for Get Open Data Source Current Data – Request

```
{
  "requestOpenData": {
    "requestType": "GetOpenDataCurrentData",
    "requestId": "xxxx",
    "serviceId": "yyyy",
    "userId": "zzzzzzz",
    "dataSourceId": "Data Source Id",
    "format": "raw"
  }
}
```

Listing 2: Message-Example for Get Open Data Source Current Data – Response

```
{
  "requestResponse": {
    "status": "success",
    "responseString": "Data fetch completed with no errors",
    "requestId": "xxxx",
    "data": {
      "filename": "data_source_id.csv",
      "location": "/requestId",
      "format": "csv",
      "cloudStorageReference": {
        "bucketId": "",
        "object": ""
      }
    }
  }
}
```

The above table and listings show the required interactions when the Service Request Handler subcomponent receives a request to fetch raw data for a SIMPLI-CITY Open Data source. The Data Retrieval subcomponent is requested to fetch the data for the Open Data source. It looks up the source configuration to determine what is the protocol and format, e.g., *protocol=SOAP*, *format=WSDL* by calling the Data Retrieval Protocol Handler. Next, the actual fetch of the data is executed. This is followed by conversion of the fetched data to CSV format. The completion of the conversion is notified to the Service Request Handler. The Service Request Handler then requests the Result Handler to execute any post-processing of the data such as storing the data in the Cloud-based Information Infrastructure or encrypting the data. (For this kind of request these post-processing actions are likely not requested.)

5.1.5.2 Data Processing – Get Open Data Source Transformed

This API call returns the latest fetched values from a particular data source transformed into the SIMPLI-CITY Unified Data Model. The data is transformed from csv format into the common structured semantic data (RDF). The data in RDF format is sent back to the requesting service via an HTTP PUT call. The location and name of the returned RDF data file is passed back to the service via a JSON object as shown in the listing below. A serialization of the RDF data into a JSON object is also possible, however it should be noted that the serialization would be specific to the particular data source. The serialization would convert the lines of RDF data into a JSON array. This conversion of data from RDF

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 37 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

to JSON format is currently proposed to take place on the SIMPLI-CITY Server Side in the Service Runtime Environment.

The call to the data Processing component is made to the base URL with context *getOpenDataTransformed*. The below table shows the corresponding RESTful interface description.

Table 7: RESTful Interface Description – Get Open Data Source Transformed

Method	POST	URL	SAPI_ROOT/simplicity/dataProcessing/getOpenDataTransformed			
Description	Request the most current data for an open data source, transform the data into the Unified Data Model and return as part of JSON object					
Parameter	none	Required	no	Possible values	Description no parameters	
JSON Object	SAPI_ROOT/simplicity/dataProcessing/JSON-Schema/requestOpenData					
JSON Attribute	requestType	Required	yes	Possible Values	getOpenDataTransforme	Description
JSON Attribute	requestId	Required	yes	Possible Values	set by SRE	Description
JSON Attribute	serviceId	Required	yes	Possible Values	set by SRE	Description
JSON Attribute	userId	Required	yes	Possible Values	UUID	Description
JSON Attribute	dataSourceId	Required	yes	Possible Values	string	Description
JSON Attribute	dataType		yes	Possible Values	transformed	Description
Example URL	SAPI_ROOT/simplicity/dataProcessing/getOpenDataTransformed					
Response	HTTP PUT of output RDF datafile followed by HTTP status code response + JSON object with data filename/location or Cloud Storage location					
HTTP Status Code	(HTTP status code)	Required	yes	Possible Values	200 400 401 500 501	Description 200: OK 400: Bad Request: Invalid elements of JSON object 401: Unauthorised. 500: Server Error 501: Not Implemented.
Example Response	HTTP status + JSON object requestResponse - see examples in the following listings					

The JSON object which is sent as part of the RESTful request is constructed as per the example message below.

Listing 3: Message-Example for Get Open Data Source Transformed – Request

```
{
  "requestOpenData": {
    "requestType": "GetOpenDataTransformedData",
    "requestId": "xxxx",
    "serviceId": "yyyy",
    "userId": "zzzzzzz",
    "dataSourceId": "Data Source Id",
    "format": "transformed"
  }
}
```

The table and listings in this section show the required interactions when the Service Request Handler subcomponent receives a request to get data for a SIMPLI-CITY Open Data source. It transforms it into the SIMPLI-CITY Unified Data Model. The Data Retrieval subcomponent is requested to fetch the data for the Open Data source. It looks up the source configuration to determine what is the protocol and format, e.g., *protocol=SOAP, format=WSDL* by calling the Data Retrieval Protocol Handler. Next, the actual fetch of the data is executed. This is followed by conversion of the fetched data to CSV format. The converted data is passed to the Stream handling software which has an operator to convert this dataset into the Unified Data Model. The resulting data is in RDF format. The completion of the transformation is notified to the Service Request Handler. The Service Request Handler then requests the Result Handler to execute any post-processing of the data such as storing the transformed data in the Cloud-based Information Infrastructure or encrypting the data.

Listing 4: Message-Example for Get Open Data Source Transformed – Response with Data to Requestor

```
{
  "requestResponse": {
    "status": "success",
    "responseString": "Data fetch completed with no errors",
    "requestId": "xxxx",
    "data": {
      "filename": "data_source_id.rdf",
      "location": "/requestId",
      "format": "rdf",
      "cloudStorageReference": {
        "bucketId": "",
        "object": ""
      }
    }
  }
}
```

Listing 5: Message-Example for Get Open Data Source Transformed – Response with Cloud Storage Location

```
{
  "requestResponse": {
    "status": "success",
    "responseString": "Data transform completed with no errors",
    "requestId": "xxxx",
    "data": {
      "filename": "data_source_id.rdf",
      "location": "/requestId",
      "format": "rdf",
      "cloudStorageReference": {
        "bucketId": "TestBucketId",
        "object": "PD94bWw[...]zcz4K"
      }
    }
  }
}
```

5.1.5.3 Data Processing – Do Diagnosis

The ‘Do Diagnosis’ REST call provides access to diagnosis over a set of SIMPLI-CITY data sources. The call is made to the base URL with context “*doDiagnosis*” and JSON object constructed as per the example message in the below listing. In this example the request is using two data sources which should be merged based on ‘space’ and ‘time’ attributes and then filtered based on ‘time’ and ‘road condition’ attributes. Diagnosis will then be run over the resulting dataset for the given time window and spatial area. Note that the merging and filtering attributes are dynamic in the sense that the requester specifies what attributes they want to use. The requester can also set the data to be sent back directly (ToRequestor) or to be stored in the cloud (ToCloudStorage).

The call to the Data Processing component is made to the base URL with context *doDiagnosis*. The next table shows the corresponding RESTful interface description.

The JSON object which is sent as part of the RESTful request is constructed as per the example message in the listing which follows the table below.

Table 8: RESTful Interface Description – Do Diagnosis

Method	POST	URL	\$API_ROOT/simplicity/dataProcessing/doDiagnosis				
Description	Request a diagnosis for a category using a set of parameters over a set of data sources						
Parameter	none	Required	no	Possible values	Description	no parameters	
JSON Object	\$API_ROOT/simplicity/dataProcessing/JSON-Schema/requestDataProcessing						
Example URL	\$API_ROOT/simplicity/dataProcessing/doDiagnosis						
Response	HTTP PUT of output RDF datafile followed by HTTP status code response + JSON object with data filename/location or Cloud Storage location						
HTTP Status Code		Required	yes	Possible values	200	Description	
					400	200: OK	
					401	400: Bad Request: Invalid elements of JSON object	
					500	401: Unauthorised.	
					501	500: Server Error	
						501: Not Implemented.	
JSON Object	\$API_ROOT/simplicity/dataProcessing/JSON-Schema/requestResponse						
Example Response	HTTP status + JSON object requestResponse - see examples in the following listings						

The above table and below listings show the required interactions for the case where the Service Request Handler receives a request to perform diagnosis over a combined set of data from the SIMPLI-CITY data sources. The data sources that can be requested are any Open Data, sensor data or user-centric data sources available to that service and user. In practice, these can be any combination or number of the SIMPLI-CITY data sources which are accessible to the requesting service. If no data source is specified, all relevant data sources to the diagnosis category will be used which the service and user are allowed to access. The diagnosis will then be done on the transformed data for the relevant datasets and the result returned to the requesting service or the Cloud Storage or both.

Listing 6: Message-Example for Do Diagnosis with Merging and Filtering – Request

```

{
  "requestDataProcessing": {
    "requestId": "xxxx",
    "requestType": "doDiagnosis",
    "serviceId": "yyyy",
    "userId": "zzzzzzz",
    "category": "Traffic Congestion",
    "dataSources": [
      "Data Source 1",
      "Data Source 2"
    ],
    "dataFrequencySeconds": 300,
    "dataOperations": [
      {
        "action": "merge",
        "attributes": [
          "space",
          "time"
        ]
      },
      {
        "action": "filter",
        "attributes": [
          "time",
          "road condition"
        ]
      }
    ],
    "timeWindow": [
      {
        "startTimeStamp": "2013-08-28T10:00:00.000Z",
        "endTimeStamp": "2013-08-29T12:00:00.000Z"
      }
    ],
    "spatialArea": {
      "lat1": 53.3478,
      "long1": 6.2597,
      "lat2": 53.3378,
      "long2": 6.2397
    },
    "roadNames": [
      "O'Connell St",
      "Pearse St "
    ],
    "sendResult": "ToRequestor",
    "encryptData": "none"
  }
}

```

The data is in RDF and the location and name of the returned RDF data file or storage location is passed back to the service via a JSON object as shown in the following two listings.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 41 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

Listing 7: Message-Example for Do Diagnosis – Respond with Data to Requestor

```
{
  "requestResponse": {
    "status": "success",
    "responseString": "Diagnosis completed with no errors",
    "requestId": "xxxx",
    "data": {
      "filename": "diagnosis_id.rdf",
      "location": "/requestId",
      "format": "rdf",
      "cloudStorageReference": {
        "bucketId": "",
        "object": ""
      }
    }
  }
}
```

Listing 8: Message-Example for Do Diagnosis –
Response with Cloud-based Information Infrastructure Location

```
{
  "requestResponse": {
    "status": "success",
    "messageString": "Diagnosis completed with no errors",
    "requestId": "xxxx",
    "data": {
      "filename": "data_source_id.rdf",
      "location": "/requestId",
      "format": "rdf",
      "cloudStorageReference": {
        "bucketId": "TestBucketId",
        "object": "PD94bWw[...]zcz4K"
      }
    }
  }
}
```

5.1.5.4 Data Processing – Do Prediction

The ‘Do Prediction’ REST call provides access to prediction in specific categories related to a selected set of SIMPLI-CITY data sources e.g. prediction of ‘Traffic Conditions’ for a particular spatial area and time window. The call is made to the base URL with context “doPrediction” and JSON object constructed as per the example message in the first listing below. In this example, the request is using two data sources which should be merged based on ‘space’ and ‘time’ attributes and then filtered based on ‘time’ and ‘road condition’ attributes. Prediction software will then be run over the resulting dataset for the given time window and spatial area. Note that the merging and filtering attributes are dynamic in the sense that the requester specifies what attributes they want to merge or filter on, however they must correspond directly to parameters contained in the datasets. The requester can set the data to be sent back directly (ToRequestor) or to be stored in the cloud (ToCloudStorage).

Table 9: RESTful Interface Description – Do Prediction

Method	POST	URL	SAPI_ROOT/simplicity/dataProcessing/doPrediction			
Description	Request a prediction using a set of parameters over a set of data sources					
Parameter	none	Required	no	Possible values	Description no parameters	
JSON Object	SAPI_ROOT/simplicity/dataProcessing/JSON-Schema/requestDataProcessing					
Example URL	SAPI_ROOT/simplicity/dataProcessing/doPrediction					
Response	HTTP PUT of output RDF datafile followed by HTTP status code response + JSON object with data filename/location or Cloud Storage location					
HTTP Status Code		Required	yes	Possible values	Description	
				200	200: OK	
				400	400: Bad Request: Invalid elements of JSON object	
				401	401: Unauthorised.	
				500	500: Server Error	
				501	501: Not Implemented.	
JSON Object	SAPI_ROOT/simplicity/dataProcessing/JSON-Schema/requestResponse					
Example Response	HTTP status + JSON object requestResponse - see examples in the following listings					

Listing 9: Message-Example for Do Prediction – Request

```

{
  "requestDataProcessing": {
    "requestId": "xxxx",
    "requestType": "doPrediction",
    "serviceId": "yyyy",
    "userId": "zzzzzzz",
    "category": "Traffic Congestion",
    "dataSources": [
      "Data Source 1",
      "Data Source 2"
    ],
    "dataFrequencySeconds": 300,
    "dataOperations": [
      {
        "action": "merge",
        "attributes": [
          "space",
          "time"
        ]
      },
      {
        "action": "filter",
        "attributes": [
          "time",
          "road condition"
        ]
      }
    ],
    "timeWindow": [
      {
        "startTimeStamp": "2013-08-28T10:00:00.000Z",
        "endTimeStamp": "2013-08-29T12:00:00.000Z"
      }
    ],
    "spatialArea": {
      "lat1": 53.3478,
      "long1": 6.2597,
      "lat2": 53.3378,
      "long2": 6.2397
    },
    "roadNames": [
      "O'Connell St",
      "Pearse St"
    ],
    "sendResult": "ToRequestor",
    "encryptData": "AES256"
  }
}

```

The data is in RDF format and the storage location is passed back to the service via a JSON object as shown in the listing below.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 44 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

Listing 10: Message-Example for Do Prediction – Response with Cloud-based Information Infrastructure Location

```
{
  "requestResponse": {
    "status": "success",
    "responseString": "Prediction completed with no errors",
    "requestId": "xxxx",
    "data": {
      "filename": "",
      "location": "",
      "format": "rdf"
    },
    "cloudStorageReference": {
      "bucketId": "TestBucketId",
      "object": "PD94bWw[...]zcz4K"
    }
  }
}
```

The above table and listings shows the Data Processing required interactions for the case where the Service Request Handler receives a request to perform prediction over a combined set of data from the SIMPLI-CITY data sources.

As for the diagnosis example, in the prediction case, the Data Retrieval subcomponent is requested to fetch the data for the particular data sources. If no data source is specified, all relevant data sources to the prediction category will be used which the service and user are allowed to access. The prediction software is then invoked. The Service Request Handler then requests the Result Handler to execute any post-processing of the data such as storing the transformed data in the Cloud Storage Infrastructure or encrypting the data.

5.1.6 Content Format

CSV Format:

The Comma Separated Values (CSV) format is used for representing raw data from the SIMPLI-CITY Open Data sources. This parsing of Open Data sources is done in the Data Retrieval sub-component. Since data is fetched using different protocols (e.g., SFTP, HTTP, SOAP) and different formats (e.g., WSDL, HTML, PDF) the data is initially transformed into CSV format for further processing. When a service requests the latest unprocessed data for a particular Open Data source, the data is returned in CSV format.

RDF Format:

The Resource Description Framework (RDF) is used as part of the SIMPLI-CITY Unified Data Model for representing all datasets. All data relating to a particular request (e.g., sensor data, user personal data, Open Data) is transformed into RDF format. This facilitates contextualization and reasoning over the combined datasets. When a service requests the Data Processing to retrieve the latest transformed data for an Open Data source or requests diagnosis or prediction facilities, the result is returned in RDF format.

JSON Format:

The JavaScript Object Notation (JSON) is used in communication between the Data Processing component and the SIMPLI-CITY backend services running in the Service

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 45 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

Runtime Environment and Context-based Service Personalisation components. The communication is done via the REST interfaces which are specified in the Section 5.1.5. JSON objects are used to wrap data and messages exchanged with the Data Processing component.

5.1.6.1 JSON Message Format Schema

Examples of the usage of the data formats and protocols are given in Section 5.1.5. The JSON schemas for Data Processing are as follows.

Listing 11: JSON Format Schema for Requesting Open Data

```
{
  "requestOpenData": {
    "title": "Request Open Data",
    "type": "object",
    "id": "http://simpli-city.eu/data_processing/json-
schema/requestOpenData",
    "required": true,
    "items": {
      "requestType": {
        "type": "string",
        "enum": [
          "getOpenDataCurrentData",
          "getOpenDataTransformedData"
        ]
      },
      "requestId": {
        "type": "String",
        "required": true
      },
      "serviceId": {
        "type": "String",
        "required": true
      },
      "userId": {
        "type": "String",
        "required": true
      },
      "datasourceId": {
        "type": "String",
        "required": true
      },
      "dataType": {
        "type": "String",
        "enum": [
          "raw",
          "transformed"
        ],
        "required": true
      }
    }
  }
}
```

Listing 12: JSON Format Schema for Response

```

{
  "requestResponse": {
    "title": "Request Response",
    "type": "object",
    "id": "http://simpli-city.eu/data_processing/json-schema/requestResponse",
    "required": true,
    "items": {
      "status": {
        "type": "string",
        "enum": [
          "success",
          "failed",
          "in progress"
        ],
        "required": true
      },
      "responseString": {
        "type": "string",
        "required": true
      },
      "requestId": {
        "type": "string",
        "required": true
      },
      "data": {
        "description": "description: location of file put to server or of data in
cloud or both if there was an error neither",
        "type": "object",
        "id": "http://simpli-city.eu/data_processing/json-schema/data",
        "required": true,
        "items": {
          "filename": {
            "type": "string",
            "required": false
          },
          "location": {
            "type": "string",
            "required": false
          },
          "format": {
            "type": "string",
            "enum": [
              "csv",
              "rdf"
            ],
            "required": true
          },
          "cloudStorageReference": {
            "type": "object",
            "items": {
              "bucketId": "string",
              "object": "string"
            },
            "required": false
          }
        }
      }
    }
  }
}

```

Listing 13: JSON Format Schema for Request Data Processing – Part 1

```

{
  "requestDataProcessing": {
    "requestType": {
      "type": "string",
      "enum": [
        "doDiagnosis",
        "doPrediction"
      ],
      "required": true
    },
    "requestId": {
      "type": "string",
      "required": true
    },
    "serviceId": {
      "type": "string",
      "required": true
    },
    "userId": {
      "type": "string",
      "required": true
    },
    "category": {
      "type": "string",
      "enum": [
        "Traffic Condition",
        "Traffic Congestion",
        "Bus Delay",
        "Road Incident",
        "Travel Time"
      ],
      "required": true
    },
    "dataSourceIds": {
      "type": "array",
      "items": {
        "type": "string"
      },
      "minItems": 2,
      "required": false
    },
    "dataFrequencySeconds": {
      "type": "number",
      "required": false
    },
    "dataOperations": {
      "type": "array",
      "items": {
        "type": "object",
        "items": {
          "action": {
            "type": "string",
            "enum": [
              "merge",
              "filter",
              "correlate",
              "summarize",
              "aggregate",
              "split"
            ]
          }
        }
      ]
    }
  }
}

```


Listing 14: JSON Format Schema for Request Data Processing – Part 2

```

        "minItems": 1,
        "required": false
    },
    "attributes": {
        "type": "array",
        "items": {
            "type": "string"
        },
        "minItems": 0,
        "required": false
    }
},
"timeWindow": {
    "type": "object",
    "items": {
        "startTimeStamp": "DateUTC",
        "endTimeStamp": "DateUTC"
    },
    "required": false
},
"spatialArea": {
    "type": "object",
    "items": {
        "lat1": "number",
        "long1": "number",
        "lat2": "number",
        "long2": "number"
    },
    "required": false
},
"roadNames": {
    "type": "array",
    "items": {
        "type": "string"
    },
    "required": false
},
"sendResult": {
    "type": "string",
    "enum": [
        "ToRequestor",
        "ToCloudStorage",
        "ToRequestorAndcloud"
    ],
    "required": true
},
"encryptData": {
    "type": "string",
    "enum": [
        "none",
        "AES256"
    ],
    "required": false
}
}

```

5.1.7 Summary

The Data Processing component will make use of Semantic Web technologies to fulfil the SIMPLI-CITY Unified Data Model for its heterogeneous datasets and contextualization and reasoning requirements over those datasets. In particular OWL 2.0 will be used for its expressivity and ability to handle static and streaming data.

IBM InfoSphere Streams is used for data combining and contextualization due to the maturity, scalability and operator plugin capability. For reasoning software, Pellet was selected for its rule generation capability with TrOWL potentially being used later if the rule feature becomes available and is stable. The Data Retrieval subcomponent will utilize Perl modules and bash scripts on Linux to cover collection of heterogeneous datasets over different protocols and in different formats.

For Data Retrieval, the modules and frequency handlers which cover all of the protocols and formats for retrieving data from the necessary SIMPLI-CITY data sources need to be implemented. For Contextualization, required mapping and operators for each required dataset to the semantic structure need to be implemented. The reasoning work involves rule generation and inferences over the transformed datasets. The Service Request Handler will implement the REST call handling and interactions with the other sub-components in the Data Processing component. Finally, the Result Handler needs to fully implement the interaction between the reasoning outcomes and the modules which handle storing results in the Cloud-based Information Infrastructure and also with the Service Request Handler and any post-processing actions such as result-data encryption.

5.2 Cloud-based Information Infrastructure

5.2.1 Major Design Decisions

A central element of SIMPLI-CITY is the provision of a scalable data storage facility. The Cloud-based Information Infrastructure has to manage data (binary, document-oriented or semantic) provided by other SIMPLI-CITY components, services or apps (indirectly via services). The Cloud-based Information Infrastructure will offer simple CRUD (create, read, update, delete) operations for the before mentioned types of data, e.g., returning values of a data set or updating properties of a data entry. Additionally, the Cloud-based Information Infrastructure will provide more advanced interfaces for structured and semantic data, supporting complex queries.

These different data types have different requirements. For example, storing large amounts of binary data files requires optimal data throughput while a semantic database needs to enable graph-based queries, but data throughput is a neglectable factor. Consequentially, several technologies for the different data types will be used as a base by the Cloud-based Information Infrastructure component.

During the discussion of the Global Architecture (deliverable D3.1) and Functional Specification (deliverable D3.2.1) discussions the focus for design decision have been made for Scalability, Separation of Concerns and Data Privacy. These three aspects will be further discussed in this chapter, which takes the current discussions to the next level and slightly adopts the component structure to fit to the technology selection.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 50 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

For further references, the Cloud-based Information Infrastructure is the whole component including the interfaces to access, manipulate and manage data as well as all databases, where the actual data will be stored, whereby the core of the component will be referred as Cloud Storage.

Scalability:

In theory, SIMPLI-CITY could be used by a very large number of users and apps concurrently, with backend services and data services serving the information needs of users and apps. As it is the goal of SIMPLI-CITY to provide a “European-Wide Service Platform”, the number of data sources and data services which need to buffer and store data in the data storage make it necessary to provide a highly scalable, yet reliable data storage.

Separation of Concerns:

Since the Cloud Storage has to store data from several components, all data will be stored in so-called Buckets. A Bucket in SIMPLI-CITY is an isolated storage space managing data. A Bucket may contain multiple data entries of the same type, which may be added, received, updated or deleted. It can be compared to a folder in a file system or a collection in a document-oriented data management system.

In SIMPLI-CITY, each component may create multiple Buckets for storing data, which are fully separated and not influenced by activities of other Buckets. The Bucket concept allows the usage of different storage backend points. By defining a Bucket Type during the creation of a Bucket, it is possible to support the data storage of different data types. In this context, a Bucket Type defines the nature of a Bucket. In this way the components can select the optimal Bucket Types depending on the best fit for their data.

Within SIMPLI-CITY, a set of three different Bucket Types will be implemented:

- **Semi-structured Bucket Type:** This Bucket Type will be used to store typical data in a document-oriented way without a fixed data schema. It is usually referred to as “NoSQL” (Not only Structured Query Language) storage and may be realized by technologies such as CouchDB or Apache Cassandra. A typical data entry in this storage will have an ID and a set of key-value entries, sometimes being hierarchical.
- **Binary Data Bucket Type:** This Bucket Type will allow a file-centric storage for binary data. It may be used to, e.g., store videos, PDFs or any other type of binary data. Queries will be based on the file name or ID, e.g., by requesting the content of document “brochure.pdf”. Potential base technologies include Amazon S3 or (distributed) file systems.
- **Semantic Data Bucket Type:** This Bucket Type will allow the storage of semantic information. Queries can be based on a semantic query language such as SPARQL. Possible base technologies include Jena or Sesame.

Data Privacy:

Each Bucket in SIMPLI-CITY is protected with its own Access Control List-feature (ACL). In this way, it is possible for each Bucket owner to specify exactly which other user or user groups can access their Bucket. One component may use several different Buckets and may specify their access rights (Denied, Read, Write, Super) to share data with other components. As such, the Cloud-based Information Infrastructure will provide a simplified

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 51 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

ACL-feature, which may be used by other SIMPLI-CITY components for controlling data access.

5.2.2 Technology Comparison

5.2.2.1 Comparison Criteria

Table 10: Criteria for Technical Specification

Parameter	Importance	Description
Scalability	++	Since SIMPLI-CITY may need to manage a large amount of service data that will grow continuously, it is necessary to store the increasing amount of data in an efficient way.
CRUD Operation Support	++	CRUD stands for create, read, update and delete and forms the base operations of any Database Management System (DBMS). It should be supported by all base technologies.
Response Time	++	Speed of writing or reading data from the data storage varies per data storage system used. So it is a very important decision criterion about choosing the underlying software/technology for the data storage.
Open Format Compatibility	+	The used data storage systems must be able to be used in different environments and by different components. As such, an Open Data format has to be chosen to transfer the queries and data between the data storage and the SIMPLI-CITY components.
Costs for Data Storage	++	From an economic point of view, the costs for the data storage should be included in the selection decision, especially when considering cloud-based storages as some are fully managed and the usage of them is not free of costs.
Cloud Compatibility	+	Not all of the existing data storage systems are absolutely suitable to work within a cloud-based system. Therefore it is pivotal whether the chosen systems support it or not.

5.2.2.2 Possible Technologies and Comparison

For the realization of the SIMPLI-CITY Cloud-based Information Infrastructure component, it is necessary to identify adequate technologies to store binary, semi-structured and semantic data. The assessment of these data storages will be carried-out independently. Additionally, a base technology with an Open Data format has to be selected to transfer the queries and data between the Cloud-based Information Infrastructure and other SIMPLI-CITY components, services or apps (indirectly).

Note: The selection in the following subsections is based on the different Bucket Types (semi-structured, binary and semantic). For managing those data types, a lot of proven technologies exist. For example, there are a lot of different approaches for storing

structured information in relational databases. As such, the following selection will compare only the 4-6 most promising technologies for each area in order to keep the document sharp and short.

5.2.2.3 Semi-Structured Data – Comparison

Amazon DynamoDB and SimpleDB:

Amazon DynamoDB¹⁶ and SimpleDB¹⁷ are NoSQL database services. They are easy to use and have a very good scalability. In comparison to SimpleDB, DynamoDB is faster, more flexible and there are no restrictions to the capacity. For this reason, DynamoDB will be used for further comparisons. The core idea is based on distributed hash tables to ensure a high availability as well as incremental scalability. As an externally hosted solution, the use of DynamoDB will lead to additional cost.

Microsoft Azure Table Storage Service:

Microsoft Azure Table Storage Service¹⁸ is a NoSQL database service. It serves access over a RESTful interface or libraries for several programming languages such as .Net, Java and PHP. It allows having multiple values for an attribute; the attributes are differentiated by a timestamp, allowing the storage of temporal data. As an externally hosted solution, the use of the Azure Table Storage will lead to additional cost.

MongoDB:

MongoDB¹⁹ is schema-less open source database that is based on documents. The documents are represented internally with the help of the BSON specification, which leads to a good performance. MongoDB is written in C++ and is available for Windows, Linux, OS X and Solaris. It serves libraries for many programming languages such as .Net, C++, Java, PHP. It contains a master-slave replication mechanism and can be run on Microsoft Azure or own hosted servers and is a cost-free solution.

Apache CouchDB:

Apache CouchDB²⁰ is a document-based database that can be accessed by a RESTful interface. It is written in Erlang and has a plug-in-architecture, which allows adding extensions. It contains a master-master merge replication mechanism. It can be hosted on own servers to avoid usage costs.

Apache Cassandra:

Apache Cassandra²¹ is an open source distributed DBMS. It is a structured key/value store and is used to store a big amount of data on different systems. It is written in Java. It runs on Linux servers and so can be used for free on the SIMPLI-CITY server.

In Table 11, the comparison of the before listed semi-structured data storages is made.

¹⁶ <http://aws.amazon.com/de/dynamodb/>

¹⁷ <http://aws.amazon.com/de/simpledb/>

¹⁸ <http://www.windowsazure.com/en-us/>

¹⁹ <http://www.mongodb.org/>

²⁰ <http://couchdb.apache.org/>

²¹ <http://cassandra.apache.org/>

Table 11: Comparison of Technologies for Semi-Structured Data Storage

Parameter	Importance	Amazon DynamoDB	Microsoft Azure Table	MongoDB	Apache CouchDB	Apache Cassandra
Generic Criteria						
Up-to-Datedness	++	10	10	10	10	10
Stability	+	8	8	8	6	6
Extensibility & Open Source/Standards	+	6	6	10	8	8
Familiarity	-	0	0	10	8	10
Performance	++	10	10	10	10	10
Interoperability	++	10	10	10	8	4
License		Proprietary	Proprietary	GNU AGPL v3.0 (drivers: Apache 2.0)	Apache 2.0	Apache 2.0
Specific Criteria						
Scalability	++	10	10	10	10	10
CRUD Operation Support	++	10	10	10	10	10
Response Time	++	8	8	10	10	10
Open Format Compatibility	+	8	8	10	10	8
Costs for Data Storage	++	4	4	8	8	6
Cloud Compatibility	+	Yes	Yes	Yes	Yes	Yes

5.2.2.4 Semantic Data – Comparison

Sesame:

Sesame²² is an open source de facto standard framework for RDF data. It supports a variety of different storage device due to a generic architecture, which abstracts the storage device layer, called SAIL (Storage And Interface Layer). It runs in a Java virtual machine and supports the following relational DBMS to store its data: PostgreSQL, MySQL, Microsoft SQL Server or Oracle. Sesame supports two query languages, SeRQL and SPARQL. As an open source product, Sesame is free of charge and can be hosted on any server running Java.

Virtuoso:

Virtuoso²³ is a multi-model cross-platform data server. Noteworthy of Virtuoso is that it stores its data in tuples. This allows also RDF data management with SPARQL support. Successfully used in various projects and products, this data server can store different types of data. As it is cross-platform and a free community edition is provided, it can be hosted on an own server.

AllegroGraph:

AllegroGraph²⁴ is a commercial RDF database. It exists as a free version with a limitation of 5 million triples. It runs on Windows, Linux and OS X and serves libraries for several programming languages including C# and Java. So it can be hosted on local server to reduce the cost of the storage system.

²² <http://www.openrdf.org/>

²³ <http://virtuoso.openlinksw.com/>

²⁴ <http://www.franz.com/agraph/>

Jena:

Jena²⁵ is an open source RDF framework. It supports SPARQL as query language. It is very similar to Sesame with the exception that it supports a better support for reasoning, but for this reason the scalability suffers, so that Sesame is the better choice in case of a large amount of data. For data storage, it can use various storage solutions like SQL databases. As an open source product, it can be used free of charge and installed on local servers.

BrightstarDB:

BrightstarDB²⁶ is a schema-free RDF store for the .Net platform. It supports SPARQL 1.1 and OData and it is ACID (Atomicity, Consistency, Isolation, Durability) compliant. It can be hosted on a local Windows server and is downloadable for free.

In Table 12, the comparison of the before listed semantic data storages is made.

Table 12: Comparison of Technologies for Semantic Data

Parameter	Importance	Sesame	Virtuoso	AllegroGraph	Jena	BrightstarDB
Generic Criteria						
Up-to-Datedness	++	10	10	10	10	6
Stability	+	8	8	8	8	8
Extensibility & Open Source/Standards	+	8	6	0	8	8
Familiarity	-	10	10	10	10	10
Performance	++	10	6	10	8	6
Interoperability	++	10	8	10	10	2
License		BSD 3- Clause License	GPLv2 and proprietary	Proprietary	Apache 2.0	MIT License
Specific Criteria						
Scalability	++	8	8	8	6	10
CRUD Operation Support	++	10	10	10	10	10
Response Time	++	10	10	10	10	10
Open Format Compatibility	+	10	10	8	10	10
Costs for Data Storage	++	8	8	8	8	8
Cloud Compatibility	+	No	No	No	No	No

5.2.2.5 Binary Data – Comparison

Binary data contains all data stored in files such as documents, images or configuration files of applications. For that reason, a technology is needed which offers an easy to use and scalable storage. For storing binary data, the following technologies are compared:

Amazon Simple Storage Service (S3):

Amazon S3²⁷ is a key-value storage which is accessible through web service interfaces like REST and SOAP. Amazon provides scalability, high availability and promises an availability of 99.99%. It is a fully managed and externally hosted service and so the usage would lead to additional cost for the project.

²⁵ <http://jena.apache.org/>

²⁶ <https://brightstardb.com/>

²⁷ <http://aws.amazon.com/en/s3/>

Microsoft Azure Blob Storage Service:

Microsoft Azure Blob Storage Service²⁸ offers a service to store large binary files in the Cloud. All stored files are replicated automatically. The storage is available via web service interfaces. The usage is not free of charge since this is a fully managed and hosted solution.

File Server:

A simple file server is a computer which provides a shared access to storage resources. It is attached to a network and accessed by attached workstations. It could be installed in a dedicated or in a non-dedicated manner. In the first case, the dedicated server is specially configured as file server and does not offer any other functionality. In the second case, the server is used in parallel for other tasks. This can lead to worse performance. There are several ways and protocols to access the data of a file server regardless of the manner it is installed, e.g., FTP, SFTP, or HTTP. As a technology and not a product, this is free of charge and can be hosted externally or internally.

Apache Subversion (SVN):

Apache SVN²⁹ (as representative for the wide range of versioning systems) is an open source versioning system which is distributed under an Apache License. It allows maintaining different versions of source code or documents. A SVN repository can also be installed on an Amazon EC2 instance, so it could be used in local and Cloud environments. Apache SVN is open source software and so can be used for free on the SIMPLI-CITY server.

In Table 13, the comparison of the above listed binary data storages is made.

Table 13: Comparison of Technologies for Binary Data Storage

Parameter	Importance	Amazon S3	Microsoft Azure Blob Storage Service	Simple File Server
Generic Criteria				
Up-to-Datedness	++	10	10	10
Stability	+	10	10	10
Extensibility & Open Source/Standards	+	10	10	6
Familiarity	-	10	8	8
Performance	++	10	10	4
Interoperability	++	10	8	6
License		Proprietary	Proprietary	Technology-specific
Specific Criteria				
Scalability	++	10	10	4
CRUD Operation Support	++	10	10	10
Response Time	++	10	8	8
Open Format Compatibility	+	8	8	10
Costs for Data Storage	++	6	6	4
Cloud Compatibility	+	Yes	Yes	Technology-specific

²⁸ <http://www.windowsazure.com/en-us/>

²⁹ <http://subversion.apache.org/>

5.2.3 Technology Selection

After comparing the technologies in the previous section, the selected technology for each data backend is explained in this section.

5.2.3.1 Selection for Semi-Structured Data Storage

MongoDB was selected as semi-structured data storage, because it is very stable and has a good runtime performance. It is internally in use by at least one project partner that is involved in the development of the Cloud-based Information Infrastructure. Thus, it is well-tested and the necessary knowhow is already available. MongoDB is an open source solution and uses the GNU Affero General Public License (AGPL), a non-infecting license. It is available for Windows, Linux, OS X and Solaris and drivers exist for many programming languages. The number of drivers continues to grow, because the community develops new ones for further programming languages. It can be run on its own server, in its own private cloud or on Microsoft Azure as a public cloud. The database supports replications by itself, so that the synchronization is fully integrated.

5.2.3.2 Selection for Semantic Data Storage

Sesame was selected to store semantic data, because it is a de-facto standard to store RDF-based data. It works with several relational DBMS and is platform independent. It has a better performance than Jena for large amounts of data and a better stability than BrightstarDB. It is open source and uses the GNU Lesser General Public License (LGPL), a non-infecting license.

5.2.3.3 Selection for Binary Data Storage

Amazon S3 was selected as binary storage, because it offers a good interoperability through standardized interfaces like REST and SOAP. It offers high availability and scalability. The account on the services is performed in a pay-as-you-use manner and thus there are no up-front costs for initial hardware purchase. Also, through the managed service, costs for storage extensions and administrative work can be avoided, which is a big advantage over a self-hosted “free” network storage system. Furthermore, it is a broadly established solution for binary data storage and is well documented and features a large community.

5.2.3.4 Missing Elements and Implementation Needs

As SIMPLI-CITY is a highly innovative research project with specific needs for data storage, the selected technologies only provide the foundation for storage and a part of the data transmission. The whole logic around this is missing and has yet to be implemented. In particular, the following functionalities need to be added in order to provide SIMPLI-CITY with an holistic data storage.

Streaming Support:

Streaming support may be a feature which has to be supported to allow components, apps and services to deliver (media) data from the Cloud-based Information Infrastructure via data streams. To support a streaming to mobile devices, it is important to adapt the data size to the actual bandwidth. For media data, this can be done by media transcoding. For this requirement an extra component is responsible, (see Section 5.4.4.2.2). It reduces the

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 57 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

data size by sampling down the quality. To adjust the streaming data size, a monitoring of the current bandwidth for all data streams must be provided. This should be transparent to components, apps and services.

Bucket Creation and Bucket Management:

As the Cloud Storage should abstract from datasets of the specific storage backend systems, it will introduce buckets as the SIMPLI-CITY Cloud Storage dataset. This abstraction is missing for all data storage types. For each abstraction creation and management for a bucket is to be implemented. The translation from and to the message protocol has to be implemented for NoSQL, binary and semantic queries. This step is needed as the Cloud Storage will provide a generic bucket storage type to components, apps and services. As a result, developers do not have to care about the specific storage systems. They can just use the Cloud Storage like it is a storage system which provides NoSQL, binary and semantic storage natively.

Access Control List:

As a generic bucket storage type will be introduced, the access control list has to be provided. It gives the opportunity for developers to control which of their application data can be accessed by whom. The access control of specific storage backends will be abstracted so that developers do not have to care about system peculiarities. With this system, it will be possible to provide an access control even for storage types which do not provide such functionality on the required granularity. For instance, based on this list, the bucket-based internal access control and user-based access control for binary storage has to be implemented.

Version Control for Binary Data:

To give developers the possibility to have access to older versions of binary data, a version control for binary data has to be created and integrated in Amazon S3. This will provide the possibility to revert changes or to recover a file when the content is damaged unintentionally. It increases the data security as information is accessible even after deletion in a bucket.

Historical Data Engine:

The possibility to store and manage historic data in a long-lasting data archive has to be implemented and integrated into the selected storage types. It enables components, apps and services to access the whole bundle of information, e.g., for statistical analysis. Another aspect is that old data does not have to be handled together with more current data, which increases the processing speed of the live system.

Subscribe Model for Buckets:

A subscribe model for buckets has to be provided in a way that components with read rights can monitor a bucket and get notified by a callback from the Cloud Storage system as soon as a change in the bucket has occurred (for further information see Sections 5.2.5.1 and 5.2.5.2). It strengthens the collaboration of components, apps and services, as they can react to data changes immediately. With this approach, components, apps and services do not have to handle the notification of others about new or changed data themselves.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 58 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

5.2.3.5 Further Information and Conclusion from Technology Comparison

The data within the different storage types need to be queried and retrieved. For this purpose, an easy to use API will be provided by the Cloud-based Information Infrastructure component for basic CRUD operations, allowing users to receive an element by its ID, to store new elements or to update/delete elements. Additionally, more advanced query facilities will be provided for some Bucket Types:

- **Semi-Structured Data Bucket Types:** Those Bucket Types will support basic CRUD operations. Additionally, queries may be performed using the self-defined JSON structure. This JSON structure will be used to abstract from the database and database query language. This reduces the dependence of the Cloud-based Information Infrastructure on a specific database or database language.
- **Semantic Data Bucket Types:** Semantic Data Bucket Types will support SPARQL queries in addition to the CRUD operations, thus allowing creation of complex queries.
- **Binary Data Bucket Types:** Binary Data Bucket Types will support only CRUD operations.

5.2.4 Component Structure

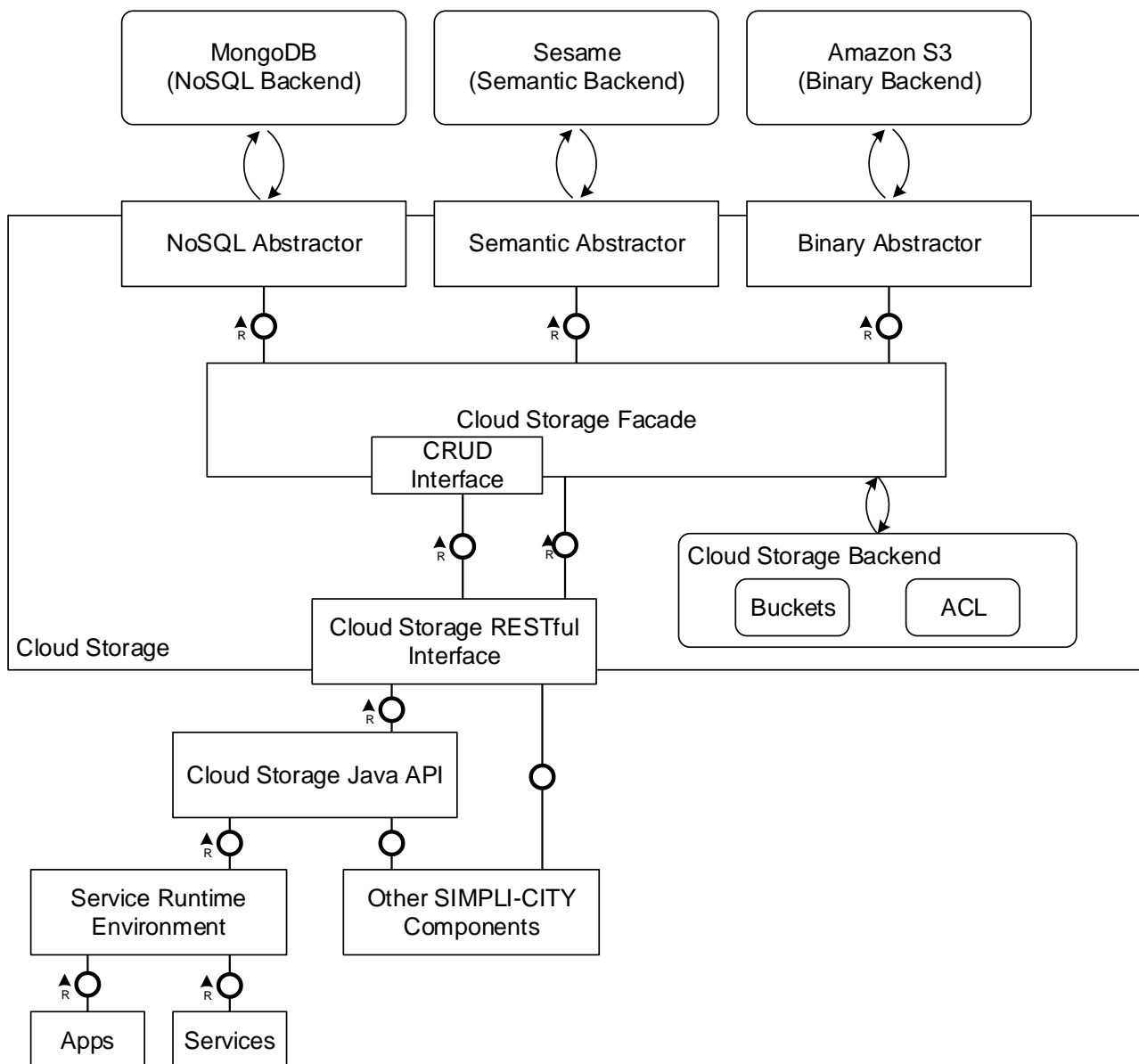


Figure 5: Component Structure of Cloud-based Information Infrastructure

The component structure (see Figure 5) has been updated to reflect the technology selection in comparison with D3.2.1. In comparison to the Functional Specification, the subcomponents have slightly changed in order to reflect the technology selection and in order to better cope with the implementation planning described in Section 5.2.5:

- Generally, the Cloud-based Information Infrastructure provides two storage locations: The *Cloud Storage*, which stores elements in the Cloud and the *Local Key Storage* (not depicted in Figure 5) which acts as a lightweight storage at the PMA. This separation into the two different areas of data storage allows other deliverables – such as deliverable D3.3 – to better distinguish between the location (Cloud or local) of the data that is to be managed by the Cloud-based Information Infrastructure.

- The Cloud Storage API has been split into a RESTful part and a Java API for quickly accessing the functionality.
- The access to the Abstractors is now handled via a façade in combination with a subcomponent for the backend, managing the Buckets and an Access Control List (ACL).

To achieve the functionalities described in the last subsection, the Cloud-based Information Infrastructure provides the following subcomponents as depicted in the figure above.

Storage Backends:

Those are external storage systems offering a concrete storage facility for a storage type. SIMPLI-CITY will support NoSQL (MongoDB), Binary (Amazon S3) and Semantic (Sesame) Storages.

Storage Abstractors:

Abstractors provide an abstraction layer around concrete storage implementations. This will allow SIMPLI-CITY to replace one storage engine with another one if necessary.

CRUD Interface:

The CRUD Interface will provide standard functionality for Create, Read, Update and Delete (CRUD), which will be available for all storage types. The CRUD Interface acts as a interface for the Cloud Storage RESTful Interface. It decouples the CRUD operations from management operations or advanced queries.

Cloud Storage Façade:

The Cloud Storage Façade will host the CRUD Interface as well as more advanced query facilities to send generic queries to the backends and provide the possibility to manipulate the Access Control List. This component is responsible for the bucket management and handles the configuration and connections to the specific storage backend systems.

Cloud Storage Backend:

The Cloud Storage Backend will be used by the Cloud Storage Façade, which will delegate the management of buckets and ACLs to this component. The Cloud Storage Backend holds the bucket definitions and ACLs.

Cloud Storage RESTful Interface:

The Cloud Storage RESTful Interface will provide the interface between the Cloud Storage and the external components like the Service Runtime Environment. This includes the access for all query and CRUD operations, the streaming access and the definition of simplified ACL functionalities.

Cloud Storage Java API:

The Cloud Storage Java API will provide the functionalities of the Cloud Storage RESTful Interface as an easy to use Java component. A component using the Cloud Storage Java API could be deployed on a different server or device than the Cloud Storage itself. To provide a flexible usage, the Cloud Storage Java API abstracts the RESTful interface and so can be used on a different physical machine.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 61 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

Local Key Storage:

While not a subcomponent of the Cloud-based Information Infrastructure, the Local Key Storage (not depicted in Figure 5) is nevertheless necessary to fulfil the requirements towards data storage within SIMPLI-CITY. The Local Key Storage provides simplistic and monolithic key-value storage for the PMA. It will be located only inside the PMA and will only be used for allowing apps to store simplistic data such as settings.

The following (see Listing 15) Java interface will be implemented for the PMA to provide the Local Key Storage. It will be offered as a Java library to the Application Runtime Environment.

Listing 15: Java Interface for the Local Key Storage

```
public interface ILocalStorage
{
    /**
     * A string will be stored under the specific key. If the key already
     * exists the old value will be overridden
     * @param key, the key under which the value will be saved
     * @param value, the value to save under the key
     * @return the Old value that was stored under the key, or null
     * @throws on connect or store failures, an IOException is thrown
     */
    String put(String key, String value) throws IOException;

    /**
     * A string will be retrieved, which was stored under the specific key.
     * If no entry was found under the key null will be returned
     * @param key, the key under which the string is stored to be retrieved
     * @return the value that was stored under the key, or null
     * @throws on connect or store failures, an IOException is thrown
     */
    String get(String key) throws IOException;

    /**
     * Removes the value stored under the specific key
     * @param key, the key under which the data should be deleted
     * @return the value that was stored under the key, or null
     * @throws on connect or store failures, an IOException is thrown
     */
    String remove(String key) throws IOException;
}
```

In order to use the Local Key Storage, a concrete implementation of the Java interface ILocalStorage (see Listing 15) must be instantiated. After that, the Local Key Storage provides the simplistic access shown in Listing 16.

Listing 16: Example on How to Use the Local Key Storage

```

ILocalStorage localKeyStorage;

...

try {
    //Save new data under a new key, null must be returned
    localKeyStorage.put(TEST_ID1, TEST_VALUE1);
    //Retrieve the data and check if it has not changed
    localKeyStorage.get(TEST_ID1).equals(TEST_VALUE1);
    //Overrides the data under key one, the old value is retrieved
    localKeyStorage.put(TEST_ID1, TEST_VALUE2).equals(TEST_VALUE1);
    //Save the same data under the second key, null must be returned
    localKeyStorage.put(TEST_ID2, TEST_VALUE2);
    //retrieves the data under both keys, the must match
    localKeyStorage.get(TEST_ID1).equals(localKeyStorage.get(TEST_ID2));
    //remove the data under key 1, data must be returned
    localKeyStorage.remove(TEST_ID1);
    //data under both keys are not the same anymore
    localKeyStorage.get(TEST_ID2).equals(localKeyStorage.get(TEST_ID1));
} catch (IOException ex) {
    // ...
}

```

5.2.5 Interfaces

In order to access the feature of the Cloud-based Information Infrastructure the component provides RESTful interfaces (described in Section 5.2.5.1) as well as a Java API (described in Section 5.2.5.2).

5.2.5.1 RESTful Interfaces

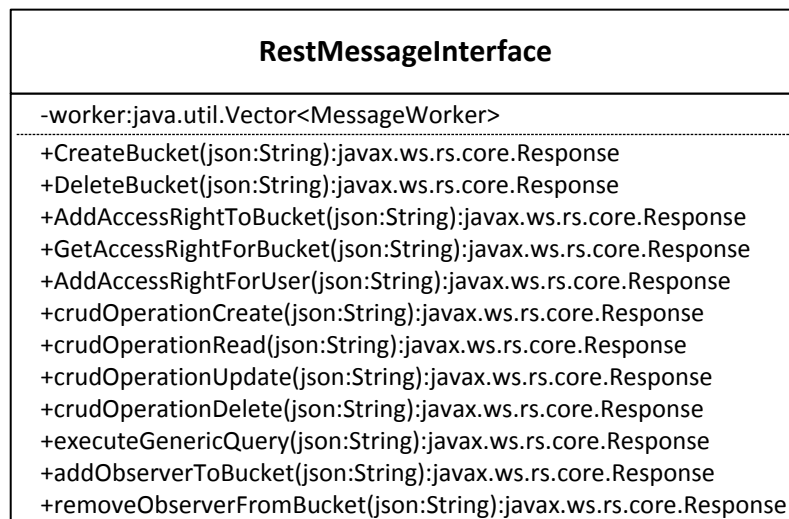


Figure 6: UML Class Diagram – RESTful Interfaces of the Cloud-based Information Infrastructure

To describe the RESTful interface methods, each supported call of the interface is described in a separate table. This table is followed with a listing showing an example for

the JSON parameter (if required for the call) as well as the JSON return value (if provided by the call). For the RESTful interfaces, it is assumed that an authentication of the user is assured and the user ID is provided additionally for each API call. Figure 6 shows the UML class diagram an overview for this Java interface. In the CRUD operations and executeGenericQuery, the JSON Command schema (Section 5.2.6.1.2) is used to encapsulate the JSON data objects. This allows communicating Commands to the Cloud-based Information Infrastructure.

5.2.5.1.1 Create Bucket

The RESTful interface Create Bucket (Table 14), creates a new bucket in the Cloud Storage, a JSON instance object with the JSON Bucket Schema (see Listing 47) is a required argument. A user context unique ID for the bucket must be provided as well as the bucket type (see Listing 17). Henceforth the Bucket can be used to store data matching the bucket type.

Table 14: RESTful Interface Description – Create Bucket

Method	POST	URL	\$API_ROOT/bucket				
Description	Creates a new Bucket for the user						
Parameter	none	Required		Possible Values		Description	
JSON Object	http://simpli-city.eu/CloudStorage/JSON-Schema/Bucket						
JSON Attribute	bucketId	Required	yes	Possible Values	any string	Description	
JSON Attribute	bucketType	Required	yes	Possible Values	SemiStructured	Description	The Bucket Type for the Bucket. This determines the backend in which the data of the Bucket will be stored.
				Binary			
				Semantic			
Example URL	\$API_ROOT/bucket						
Response	HTTP status code only						
HTTP Status Code		Required	yes	Possible Values	201	Description	Bucket created
					409		Exist already
					502		Database error
Example Response	HTTP/1.1 201 OK						

Listing 17 is an example for a valid JSON object which is required for the RESTful interface described in Table 14. The bucketId attribute, which is used to identify the bucket during further operations, is a “testId”. The bucketType attribute is set as “binary”, thereby only binary data can be stored in the bucket identified as “testId”. The identifier for the bucket is unique in the context of the owner. Every user can have a bucket with the same bucketId, but is responsible that the ID is unique in his context. This operation does not provide any data, so it does not have a response message.

Listing 17: JSON Example: Argument for Creating a Bucket

```
{
  "bucketId": "testId",
  "bucketType": "binary"
}
```

5.2.5.1.2 Delete Bucket

The RESTful Interface Delete (see Table 15), deletes an existing bucket in the Cloud Storage, a JSON instance object with the JSON Bucket Schema (see Listing 47) is a required argument. Only the ID of the bucket must be provided.

Table 15: RESTful Interface Description – Delete Bucket

Method	DELETE	URL	\$API_ROOT/bucket/:bucketId				
Description	Deletes the specific Bucket						
Parameter	bucketId	Required	yes	Possible Values	any string	Description	The ID of the Bucket
Example URL	\$API_ROOT/bucket/testId						
Response	HTTP status code only						
HTTP Status Code		Required	yes	Possible Values	200	Description	Bucket deleted
					401		Insufficient rights
					404		Bucket does not exist
Example Response	HTTP/1.1 200 OK						

The bucketId, which is used to identify the bucket to be deleted, was defined by the owner of the bucket during the creation of the bucket (see Section 5.2.5.1.1). All data saved in the bucket will be deleted with the bucket. For this reason, only the owner and user with the access right “Super” (see also Section 5.2.5.1.3) can delete a bucket. This operation does not provide any data, so it does not have a response message.

5.2.5.1.3 Add Access Right to Bucket

The RESTful interface for Add Access Right (see Table 16), adds an access right to an existing bucket in the Cloud Storage, a JSON instance object with the JSON AccessRight Schema (see Listing 49) is a required argument. The access right consists of an ID identifying the user or user group and the actual right (see Listing 18):

- *NotSet* will be used to delete an AccessRight.
- *Denied* will be used to deny a user or group the any access to a bucket.
- *Read* will be used to allow a user or group to read the contents of a bucket.
- *Write* will be used to allow a user or group to read and write the contents of a bucket.
- *Super* will be used to allow a user or group the same rights as the owner, this includes the right to grant or restrict access rights to other user or groups as well as to delete the bucket. It is only restricted in the way, that it will not be possible for a user or group with this AccessRight to remove the “Super” access right of the owner of the bucket (This right is granted to the owner without a specific access right, hence a restriction will not be possible).

As this operation does not return any data, only a HTTP status code will be returned.

Table 16: RESTful Interface Description – Add Access Right to Bucket

Method	PUT	URL	\$API_ROOT/bucket/:bucketId/access				
Description	Adds Access Rights to a bucket						
Parameter	bucketId	Required	yes	Possible Values	any string	Description	The ID of the Bucket
JSON Object	http://simpli-city.eu/CloudStorage/JSON-Schema/AccessRight						
JSON Attribute	id	Required	yes	Possible Values	any string	Description	User or group ID
JSON Attribute	right	Required	yes	Possible Values	NotSet	Description	The right to be granted
				Denied			
				Read			
				Write			
				Super			
Example URL	\$API_ROOT/bucket/testId/access						
Response	HTTP status code only						
HTTP Status Code		Required	yes	Possible Values	200	Description	Right updated
				201	Right created		
				401	Insufficient rights		
				404	Bucket does not exist		
Example Response	HTTP/1.1 200 OK						

Listing 18 is an example for a valid JSON object which is required for RESTful interface described in Table 16. The id attribute is set to “testUser”, this string identifies a user. The right attribute is set to “Read”, which allows the “testUser” to read the data stored in the Bucket identified by the RESTful parameter bucketId (see Table 16).

Listing 18: JSON Example: Argument for Add Access Right to Bucket

```
{
  "id": "testUser",
  "right": "Read"
}
```

5.2.5.1.4 Get Access Right for Bucket

The RESTful Interface Get Access Right for Bucket (see Table 17), get the access rights linked to an existing Bucket in the Cloud Storage, a list JSON instance objects with the JSON AccessRight Schema (see Listing 49) will be returned (see).

Table 17: RESTful Interface Description – Get Access Right for Bucket

Method	GET	URL	\$API_ROOT/user/:bucketId/access				
Description	Retrieves a list with access rights for the Bucket						
Parameter	bucketId	Required	yes	Possible Values	any string	Description	The ID of the Bucket
Example URL	\$API_ROOT/user/testId/access						
Response	HTTP status code and JSON object, a List of AccessRight objects						
HTTP Status Code		Required	yes	Possible Values	200	Description	OK
					404		Bucket does not exist
JSON Object	http://simpli-city.eu/CloudStorage/JSON-Schema/AccessRight						
JSON Attribute	id	Required	yes	Possible Values	any string	Description	The ID of the user or group the access right is granted to
JSON Attribute	right	Required	yes	Possible Values	Denied		Description
					Read		
					Write		
					Super		
Example Response	HTTP/1.1 200 OK						

is an example for a valid JSON object which is returned by the RESTful Interface described in Table 17. In the list attribute, a list of access right JSON objects is encapsulated. The list has only one entry. The Access Right for the user identified by the id “testUser” and the right “Read”, which allows the user to read all data entries in the bucket identified by the RESTful parameter bucketId (see Table 17).

Listing 19: JSON Example: Return Value for Get Access Right for Bucket

```
{
  "list": [
    {
      "id": "testUser",
      "right": "Read"
    }
  ]
}
```

5.2.5.1.5 CRUD-Operation Create

The RESTful interface CRUD-Operation Create (see Table 18), executes a create operation on an existing bucket in the Cloud Storage. A JSON instance object to the JSON DTO Schema (see Listing 42) is a required argument as is for all commands compliant with the JSON Schema for Command (see Listing 52).

Table 18: RESTful Interface Description – CRUD-Operation Create

Method	POST	URL	\$API_ROOT/bucket/:bucketId/crud				
Description	CRUD-Operation Create: Creates a data object in the specific Bucket						
Parameter	bucketId	Required	yes	Possible Values	any string	Description	The ID of the Bucket
JSON Object	http://simpli-city.eu/CloudStorage/JSON-Schema/DTO						
JSON Attribute	description	Required	no	Possible Values	any string	Description	Meta description of the object
JSON Attribute	link	Required	no	Possible Values	any string	Description	Meta link for more data depending on object type
JSON Attribute	GENERIC	Required	yes	Possible Values	anything	Description	The DTO is a Supertype
Example URL	\$API_ROOT/bucket/testId/crud						
Response	HTTP status code only						
HTTP Status Code		Required	yes	Possible Values	201	Description	Data object created
					401		Insufficient rights
					404		Bucket does not exist
Example Response	HTTP/1.1 201 OK						

An example of a valid instance of such a JSON DTO Schema (see Listing 42) the KeyValueCollection JSON Schema (see Listing 50) is used in Listing 20. This schema can be utilized to manage data in all three bucket types as long as the developer assures the compatibility of stored data by a proper encoding for the value, e.g., Base64. The key attribute is used to identify the data object.

Listing 20: JSON Example: Argument for CRUD-Operation Create

```
{
  "description": "Testobject",
  "GENERIC": {
    "key": "myKey",
    "value": "TestValue"
  }
}
```

5.2.5.1.6 CRUD-Operation Read

The RESTful interface CRUD-Operation Read (see Table 50), executes a read operation on an existing bucket in the Cloud Storage. A JSON instance object with the JSON DTO Schema (see Listing 42) is a required argument as is for all commands compliant with the JSON Schema for Commands (see Listing 52).

Table 19: RESTful Interface Description – CRUD-Operation Read

Method	GET	URL	\$API_ROOT/user/:bucketId/crud				
Description	CRUD-Operation Read: Retrieves as list of data objects from the specific Bucket						
Parameter	bucketId	Required	yes	Possible Values	any string	Description	The ID of the Bucket
JSON Object	http://simpli-city.eu/CloudStorage/JSON-Schema/DTO						
JSON Attribute	description	Required	no	Possible Values	any string	Description	Meta description of the object
JSON Attribute	link	Required	no	Possible Values	any string	Description	Meta link for more data, depending on object type
JSON Attribute	GENERIC	Required	yes	Possible Values	anything	Description	The DTO is a Supertype
Example URL	\$API_ROOT/user/testId/access						
Response	HTTP status code and JSON object, a List of DTO						
HTTP Status Code		Required	yes	Possible Values	200 401 404	Description	OK Insufficient rights Bucket does not exist
JSON Object	http://simpli-city.eu/CloudStorage/JSON-Schema/DTO						
JSON Attribute	description	Required	no	Possible Values	any string	Description	Meta description of the object
JSON Attribute	link	Required	no	Possible Values	any string	Description	Meta link for more data, depending on object type
JSON Attribute	GENERIC	Required	yes	Possible Values	anything	Description	The DTO is a Supertype
Example Response	HTTP/1.1 200 OK						

Listing 21 and Listing 22 are examples for a valid JSON objects which are required (respectively provided) by the RESTful interface described in Table 19. It is conforming to the JSON Schema for the DTO (see Listing 42) KeyValueType (see Listing 50). This schema can be utilized to manage data in all three bucket types as long as the developer assures the compatibility of stored data by a proper encoding for the value, e.g., Base64. The key attribute is used to identify the data object. In Listing 21, the attribute for key is set to “myKey”. This data entry will be looked up in the bucket identified by the RESTful parameter bucketId (see Table 50).

Listing 21: JSON Example: Argument for CRUD-Operation Read

```
{
  "GENERIC": {
    "key": "myKey"
  }
}
```

In Listing 22, an example JSON object for the response of the query is shown. The data entry with the key attribute “myKey” was found in the bucket identified by the RESTful parameter bucketId (see Table 50).

Listing 22: JSON Example: Return Value for CRUD-Operation Read

```
{
  "GENERIC": {
    "key": "myKey",
    "value": "TestValue"
  }
}
```

5.2.5.1.7 CRUD-Operation Update

The RESTful interface CRUD-Operation Update (see Table 20), executes an update operation on an existing bucket in the Cloud Storage. Two JSON instance objects (as list) with the JSON DTO Schema (see Listing 42) are required arguments. As this operation does not return any data, only a HTTP status code will be returned.

Table 20: RESTful Interface Description – CRUD-Operation Update

Method	PUT	URL	\$API_ROOT/bucket/:bucketId/crud				
Description	CRUD-Operation Update: Updates a data object in the specific Bucket						
Parameter	bucketId	Required	yes	Possible Values	any string	Description	The ID of the Bucket
JSON Object	List of JSON objects with two entries of type: http://simpli-city.eu/CloudStorage/JSON-Schema/DTO						
JSON Attribute	description	Required	no	Possible Values	any string	Description	Meta description of the
JSON Attribute	link	Required	no	Possible Values	any string	Description	Meta link for more data,
JSON Attribute	GENERIC	Required	yes	Possible Values	anything	Description	The DTO is a Supertype
Example URL	\$API_ROOT/bucket/testId/crud						
Response	HTTP status code only						
HTTP Status Code		Required	yes	Possible Values	200	Description	Data objects updated
					401		Insufficient rights
					404		Bucket does not exist
Example Response	HTTP/1.1 200 OK						

In Listing 23, examples for the two JSON objects (as list) which are required as parameter by the RESTful interface described in Table 20 are shown. They are conforming to the JSON Schema for the DTO (see Listing 42) KeyValueType (see Listing 50). This schema can be utilized to manage data in all three bucket types as long as the developer assures the compatibility of stored data by a proper encoding for the value, e.g., Base64. The key attribute is used to identify the data object, which shall be updated. The second object holds the value, which shall be updated.

Listing 23: JSON Example: Argument for CRUD-Operation Update

```
{
  "GENERIC": {
    "list": [
      {
        "key": "myKey"
      },
      {
        "value": "SecondTestValue"
      }
    ]
  }
}
```

5.2.5.1.8 CRUD-Operation Delete

The RESTful interface CRUD-Operation Delete (see Table 21), executes a delete operation on an existing bucket in the Cloud Storage. A JSON instance object with the JSON DTO Schema (see Listing 42) is a required argument as is for all commands compliant with the JSON Schema for Command (see Listing 52). As this operation does not return any data, only a HTTP status code will be returned.

Table 21: RESTful Interface Description – CRUD-Operation Delete

Method	DELETE	URL	\$API_ROOT/bucket/:bucketId/crud				
Description	CRUD-Operation Delete: Deletes a data object in the specific Bucket						
Parameter	bucketId	Required	yes	Possible Values	any string	Description	The ID of the Bucket
JSON Object	http://simpli-city.eu/CloudStorage/JSON-Schema/DTO						
JSON Attribute	description	Required	no	Possible Values	any string	Description	Meta description of the
JSON Attribute	link	Required	no	Possible Values	any string	Description	Meta link for more data,
JSON Attribute	GENERIC	Required	yes	Possible Values	anything	Description	The DTO is a Supertype
Example URL	\$API_ROOT/bucket/testId/crud						
Response	HTTP status code only						
HTTP Status Code		Required	yes	Possible Values	200	Description	Data objects deleted
					401		Insufficient rights
					404		Bucket does not exist
Example Response	HTTP/1.1 200 OK						

In Listing 24, an example for a valid JSON object which is required as parameter by the RESTful Interface described in Table 21 is shown. It is conforming to the JSON Schema for the DTO (see Listing 42) KeyValueObject (see Listing 50). The key attribute is used to identify the data object, which shall be deleted. This data entry will be looked up in the bucket identified by the RESTful parameter bucketId (see Table 21).

Listing 24: JSON Example: Argument for CRUD-Operation Delete

```
{
  "GENERIC": {
    "key": "Test"
  }
}
```

5.2.5.1.9 Execute Generic Query

The RESTful Interface Generic Query (see Table 22), executes a generic query on an existing bucket in the Cloud Storage. A JSON instance object with the JSON DTO Schema (see Listing 42) is a required argument as is for all commands compliant with the JSON Schema for Command (see Listing 52).

Table 22: RESTful Interface Description – Generic Query

Method	PUT	URL	\$API_ROOT/user/:bucketId/query				
Description	Executes a generic query on the specific bucket, the query is encapsualted in a DTO						
Parameter	bucketId	Required	yes	Possible Values	any string	Description	The ID of the Bucket
JSON Object	http://simpli-city.eu/CloudStorage/JSON-Schema/DTO						
JSON Attribute	description	Required	no	Possible Values	any string	Description	Meta description of the
JSON Attribute	link	Required	no	Possible Values	any string	Description	Meta link for more data,
JSON Attribute	GENERIC	Required	yes	Possible Values	anything	Description	The DTO is a Supertype
Example URL	\$API_ROOT/user/testId/access						
Response	HTTP status code and JSON object, a List of DTO						
HTTP Status Code		Required	yes	Possible Values	200 401 404	Description	OK Insufficient rights Bucket does not exist
JSON Object	http://simpli-city.eu/CloudStorage/JSON-Schema/DTO						
JSON Attribute	description	Required	no	Possible Values	any string	Description	Meta description of the
JSON Attribute	link	Required	no	Possible Values	any string	Description	Meta link for more data,
JSON Attribute	GENERIC	Required	yes	Possible Values	anything	Description	The DTO is a Supertype
Example Response	HTTP/1.1 200 OK						

Listing 25 and Listing 26 are examples for valid JSON objects required as parameter (respectively the return value) of the RESTful interface described in Table 22. It is conforming to the JSON Schema for the DTO (see Listing 42) KeyValueType (see Listing 50). This schema can be utilized to manage data in all three bucket types as long as the developer assures the compatibility by a proper encoding for the value, e.g., Base64. The key attribute is used to identify the data object on which the query will be executed. If no key is provided the query will be executed on the whole bucket identified by the RESTful parameter bucketId (see Table 22).

Listing 25: JSON Example: Argument for Generic Query

```
{
  "GENERIC": {
    "key": "myKey",
    "value": "QueryString"
  }
}
```

The value attribute of the JSON object in Listing 26 contains the result of the query.

Listing 26: JSON Example: Return Value for Generic Query

```
{
  "GENERIC": {
    "key": "myKey",
    "value": "ResultQueryString"
  }
}
```


5.2.5.1.10 Add Observer to Bucket

The RESTful interface Add Observer to Bucket (see Table 23), adds an observer to an existing bucket in the Cloud Storage. A JSON instance object with the JSON DTO Schema (see Listing 42) is a required argument as is for all commands compliant with the JSON Schema for Command (see Listing 52). The link attribute of the DTO object is used to transfer the URL which shall be used as the callback to the SIMPLI-CITY service, which will act as the observer. On notification, this URL will be called to notify the service.

Table 23: RESTful Interface Description – Add Observer to Bucket

Method	PUT	URL	\$API_ROOT/user/:bucketId/observer				
Description	Adds an observer to the specific Bucket						
Parameter	bucketId	Required	yes	Possible Values	any string	Description	The ID of the Bucket
JSON Object	http://simpli-city.eu/CloudStorage/JSON-Schema/DTO						
JSON Attribute	description	Required	no	Possible Values	any string	Description	Meta description of the
JSON Attribute	link	Required	no	Possible Values	any string	Description	Link to the web service as
Example URL	\$API_ROOT/user/testId/observer						
Response	HTTP status code						
HTTP Status Code		Required	yes	Possible Values	200	Description	OK
					404		Bucket does not exist
Example Response	HTTP/1.1 200 OK						

Listing 27 shows an example for a valid JSON object which is required as parameter by the RESTful interface described in Table 23. It is conforming to the JSON Schema for the DTO (see Listing 42). The attribute link is utilized to pass the callback URI identifying the observer.

Listing 27: JSON Example: Argument to Add an Observer to a Bucket

```
{
  "link": "http://www.somewebservice.url"
}
```

5.2.5.1.11 Remove Observer from Bucket

The RESTful interface Remove Observer from Bucket (see Table 24), removes a observer from an existing bucket in the Cloud Storage. A JSON instance object with the JSON DTO Schema (see Listing 42) is a required argument as is for all commands compliant with the JSON Schema for Command (see Listing 52). The observer is identified by the URL, which was used as a callback to the WebService of the observer.

Table 24: RESTful Interface Description – Remove Observer from Bucket

Method	DELETE	URL	\$API_ROOT/user/:bucketId/observer				
Description	Adds an observer to the specific Bucket						
Parameter	bucketId	Required	yes	Possible Values	any string	Description	The ID of the Bucket
JSON Object	http://simpli-city.eu/CloudStorage/JSON-Schema/DTO						
JSON Attribute	description	Required	no	Possible Values	any string	Description	Meta description of the
JSON Attribute	link	Required	no	Possible Values	any string	Description	Link to the web service as
Example URL	\$API_ROOT/user/testId/observer						
Response	HTTP status code						
HTTP Status Code		Required	yes	Possible Values	200	Description	OK
					404		Bucket or observer does not
Example Response	HTTP/1.1 200 OK						

Listing 28 shows an example for a valid JSON object which is required as parameter by the RESTful Interface described in Table 24. It is conforming to the JSON Schema for the DTO (see Listing 42). The attribute link is utilized to identify the observer which shall be removed.

Listing 28: JSON Example: Argument to Remove an Observer from a Bucket

```
{
  "link": "http://www.somewebservice.url"
}
```

5.2.5.2 Java Interfaces

The Java interfaces act as wrappers for the RESTful interfaces (see Section 5.2.5.1). They provide a convenient way to access the Cloud Storage for Java developers. Figure 7 shows the UML class diagram as overview for the Java interfaces. To ease the implementation of the data transfer objects, JAXB³⁰ will be used to generate Java class files from a XML description of these objects.

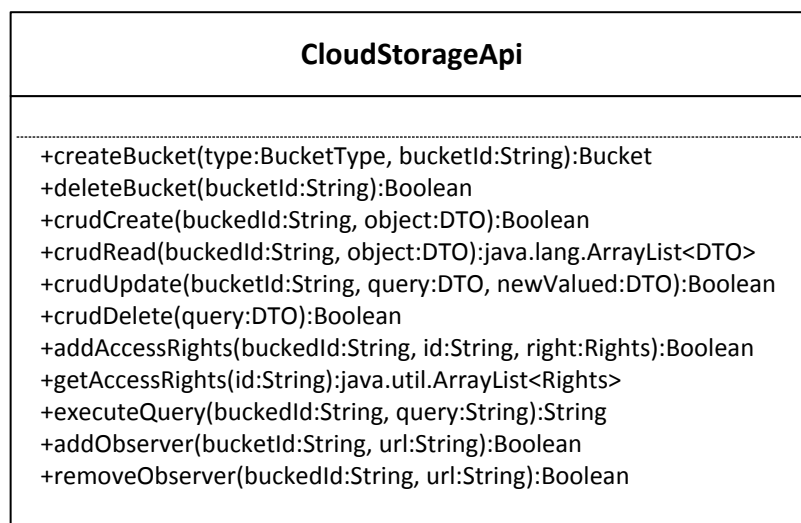


Figure 7: UML Class Diagram – Java Wrapper for the RESTful Interfaces

5.2.5.2.1 Create Bucket

For the creation of buckets, the API provides the method `createBucket`. The method has two parameters. The first defines the type of the bucket which shall be created, the second is a user defined ID for the bucket. The ID must be unique in the context of the user.

Parameters:

- `type`: `BucketType` (see Listing 59) of the Bucket which shall be created
- `bucketId`: A user defined ID for the bucket. The ID must be unique in the context of the user.

Return Value:

A bucket object representing the created bucket (see Listing 64).

Error Handling:

In case of an error, a null-value is returned.

³⁰ <https://jaxb.java.net>

Remarks:

The access rights for the bucket will be set automatically by the Cloud Storage component depending on the user executing the underlying RESTful Interface.

In Listing 29, the signature of the method for create bucket is shown as well as an example for the usage of this method.

The exemplary usage of the method at the bottom of Listing 29 will create a new bucket for semi-structured data with the ID string of the constant BUCKET_TEST_ID. If the API call is successful a bucket object will be returned, otherwise null.

Listing 29: Source Code Example – API Method Signature for
Creating a Bucket and the Usage of this Method

```
/**
 * @param type, bucket type of the bucket which shall be created
 * @param bucketId, a user defined ID for the bucket. The ID must be unique in the
 * context of the user
 * @return on success a Bucket object, otherwise null
 */
public Bucket createBucket(BucketType type, String bucketId)

...

//Creates a semi-structured bucket under with the id BUCKET_TEST_ID
Bucket bucket = api.createBucket(BucketType.SEMI_STRUCTURED, BUCKET_TEST_ID);
```

5.2.5.2.2 Delete Bucket

For the deletion of buckets, the API provides the method deleteBucket. The method has one parameter, the ID of the bucket which shall be deleted.

Parameters:

- bucketId: The ID of the bucket which shall be deleted.

Return Value:

True on success

Error Handling:

In case of an error false is returned

Remarks:

All data saved in the bucket will be deleted with the bucket. For this reason, only the owner and user with the AccessRight Super (see also Section 5.2.5.1.3) can delete a bucket.

In Listing 30, the signature of the method to delete a bucket is shown as well as an example for the usage of this method.

Listing 30: Source Code Example – API Method Signature for Deleting a Bucket and the Usage of the Method

```
/**
 * @param bucketId, the ID of the bucket which shall be deleted
 * @return true on success
 */
public boolean deleteBucket(String bucketId)

...

//Deleted the bucket with the ID BUCKET_TEST_ID
api.deleteBucket(BUCKET_TEST_ID);
```

The exemplary usage of the method at the bottom of Listing 30 deletes the bucket with the provided ID. If the API call is successful true is returned, otherwise false.

5.2.5.2.3 CRUD-Operation Create

To create a new data object in a bucket, the API provides the method createBucket. As example for a data object for this and all following examples, which require a data object Listing 31 shows a factory method. This method creates an instance of the test class MyTestClass. This class has valid JAXB annotations as well as a corresponding XSD file (see Listing 32).

Listing 31: Source Code Example – Factory Method to Create a Test Data Object

```
//Creates an instance of MyTestClass. A class with valid JAXB-Annotations and an apt
XSD file
private MyTestClass getTestClass()
{
    MyTestClass test = new MyTestClass();
    test.setTestAttributeBoolean(true);
    test.setTestAttributeInt(INT_BEFORE);
    test.setTestAttributeString(STRING_BEFORE);
    return test;
}
```

Listing 32: Source Code Example – XSD Description for a Test Class

```

<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified">
  <xs:element name="MyTestClass"
    xmlns="http://www.ascora.de/xmlSchema/CloudStorage/CloudStorage.xsd">
    <xs:complexType >
      <xs:sequence>
        <xs:element name="testAttributeString"
          type="xs:string"
          minOccurs="0" />
        <xs:element name="testAttributeInt"
          type="xs:int"
          minOccurs="0" />
        <xs:element name="testAttributeBoolean"
          type="xs:boolean"
          minOccurs="0" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>

```

The crudCreate method of the API has two parameters. The first identifies the bucket in which the data object shall be stored. The second is the data object to be stored in the bucket. The data object must be a subclass of the DTO class (see Listing 62).

Parameters:

- bucketId: The ID of the bucket in which the data object shall be created.
- object: A generic (Java) object (see Listing 66) which shall be stored in the bucket.

Return Value:

True on success

Error Handling

In case of an error, false is returned

Remarks:

The Java interface manipulates with the CRUD-operations generic Java objects.

In Listing 33, the signature of the method for the CRUD-operation crudCreate is shown as well as an example for the usage of this method. The DTO class will be further discussed in Listing 62.

Listing 33: Source Code Example – API Method Signature
for the CRUD-Operation crudCreate and the Usage of this Method

```

/**
 * @param bucketId, the ID of the bucket in which the data object shall be created
 * @param object, the data object which shall be stored in the bucket
 * @return true on success
 */
public boolean crudCreate(String bucketId, DTO object)

...

//Creates test data
MyTestClass testData = getTestClass();

//Call of a Helper Method. The Helper method is a factory method which creates
//a DTO object. The Following arguments are required:
//    -the URI to a valid XSD file describing the class of the object
//    -the object instance itself
//    -the package name of the class of the object
DTO test = Helper.GenerateGenericObject("res/TestSchema.xsd", testData,
    "junit.test.vo");

//Creates the data object "test" in the (with the ID) specific bucket
boolean success = api.crudCreate(BUCKET_TEST_ID, test);

```

The exemplary usage of the method at the bottom of Listing 33 will store the created test data object in the bucket for semi-structured data with the ID BUCKET_TEST_ID. If the API call is successful, the variable success is set to true.

5.2.5.2.4 CRUD-Operation Read

To retrieve existing data object from the Cloud Storage component, the API provides the method crudRead. For this, a query object is necessary. This object must be of the same type as the data objects that should be retrieved. As mentioned before, for this the factory method shown in Listing 31 will be used. This method creates an instance of the test class MyTestClass. This class has valid JAXB annotations as well as a corresponding XSD file (see Listing 32).

The crudRead method of the API has three parameters. The first identifies the bucket from which the data objects shall be retrieved. The second is the query data object used to find the data objects in question. The last parameter is for the Java object class of the objects that shall be retrieved. The query object must be a subclass of the DTO class (see Listing 42).

Parameters:

- bucketId: The ID of the bucket from which the data objects shall be retrieved.
- object: A generic (Java) object (see Listing 66) as query with attributes set to search for in the bucket.

Return Value:

An ArrayList of objects which have matched the attributes of the query.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 77 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

Error Handling:

If no matching data object is found, the returned list is empty.

Remarks:

The Java interface manipulates with the CRUD-operations generic Java objects.

In Listing 34, the signature of the method for the CRUD-operation crudRead is shown as well as an example for the usage of this method.

Listing 34: Source Code Example – API Method Signature
for the CRUD-Operation crudRead and the Usage of this Method

```

/**
 * @param bucketId, the ID of the bucket from which the data objects shall be
 * retrieved
 * @param object, query object with attributes set to search for in the bucket
 * @return a ArrayList of object which matches the attributes of the query object
 */
public ArrayList<DTO> crudRead(String bucketId, DTO object)

...

//Creates test data to query for data objects in the cloud storage
MyTestClass testQuery = new MyTestClass();
//Changes the integer attribute of the test data object
testQuery.setTestAttributeInt(INT_AFTER);

//Call of a Helper Method. The Helper method is a factory method which creates
//a DTO object. The Following arguments are required:
//    -the URI to a valid XSD file describing the class of the object
//    -the object instance itself
//    -the package name of the class of the object
DTO test = Helper.GenerateGenericObject("res/TestSchema.xsd", testQuery,
"junit.test.vo");

//Creates a list for the results of the CRUD operation
ArrayList<MyTestClass> list = new ArrayList<MyTestClass>();
//Retrieve the data objects from a specific bucket matching the query object
ArrayList<Object> resultList = api.crudRead(BUCKET_TEST_ID, test ,
testQuery.getClass());
//Cast the generic result list of objects to the test class
for (Object object : resultList)
    list.add((MyTestClass) object);

```

The exemplary usage of the method at the bottom of Listing 34 creates a query object, which is used for the API call. The result list of objects is cast to the expected type. If no matching objects could be found in the specific bucket the result list is empty.

5.2.5.2.5 CRUD-Operation Update

To update attributes of existing data object in the Cloud Storage component, the API provides the method `crudUpdate`. For this, a query object is necessary. This object must be of the same type as the data objects that should be updated as well as a data object with new values for the objects in question. As mentioned before, for this the factory method shown in Listing 31 will be used. This method creates an instance of the test class `MyTestClass`. This class has valid JAXB annotations as well as a corresponding XSD file (see Listing 32).

The `crudUpdate` method of the API has three parameters. The first identifies the bucket in which the data objects shall be updated. The second is the query data object used to find the data objects in question. The last parameter is the data object with the new values for the found data objects in the Cloud Storage component. The query object as well as the object encapsulating the new values must be a subclass of the DTO class (see Listing 42).

Parameters:

- `bucketId`: The ID of the bucket from which the data objects shall be retrieved.
- `query`: A DTO object, e.g., a generic (Java) object (see Listing 66), with attributes set to identify the data objects to be updated.
- `newValues`: A DTO object, e.g., a generic (Java) object (see Listing 66), with attributes set to be updated.

Return Value:

True on success

Error Handling:

In case of an error, false is returned

Remarks:

The Java interface manipulates generic Java objects. In Listing 35, the signature of the method for the CRUD-operation `crudUpdate` is shown as well as an example for the usage of this method.

Listing 35: Source Code Example – API Method Signature for the CRUD-Operation crudUpdate and the Usage of this Method

```

/**
 * @param bucketId, the id of the bucket in which the data objects shall be updated
 * @param object, query object with attributes set to identify the data objects to *
 * updated
 * @param newValues, a data object with new values
 * @return true on success
 */
public boolean crudUpdate(String bucketId, DTO query, DTO newValues)

...

//Creates test data to query for data objects in the cloud storage
MyTestClass testQuery = getTestClass();
testQuery.setTestAttributeString(null);

//Creates test data with updated values
MyTestClass testData = getTestClass();
testData.setTestAttributeInt(INT_AFTER);
testData.setTestAttributeString(STRING_AFTER);

//Double call of a Helper Method. The Helper method is a factory method which
creates
//a DTO object. The Following arguments are required:
// -the URI to a valid XSD file describing the class of the object
// -the object instance itself
// -the package name of the class of the object
DTO query = Helper.GenerateGenericObject("res/TestSchema.xsd", testQuery,
"junit.test.vo");
DTO newValues = Helper.GenerateGenericObject("res/TestSchema.xsd", testData,
"junit.test.vo");

//Updates all attributes of objects found with the query object with the
//attributes of the newValues object
boolean success = api.crudUpdate(BUCKET_TEST_ID, query, newValues);

```

The exemplary usage of the method at the bottom of Listing 35 creates a query object and an object with new values for the data objects which shall be updated in the Cloud Storage component. If the API call is successful, true is returned, otherwise false.

5.2.5.2.6 CRUD-Operation Delete

To delete data objects in the Cloud Storage component, the API provides the method crudDelete. For this, a query object is necessary. This object must be of the same type as the data objects that should be. As mentioned before, for this the factory method shown in Listing 31 will be used. This method creates an instance of the test class MyTestClass. This class has valid JAXB annotations as well as a corresponding XSD file (see Listing 32).

The crudDelete method of the API has two parameters. The first identifies the bucket from which the data objects shall be deleted. The second is the query data object used to find the data objects in question. The query object must be a subclass of the DTO class (see Listing 42).

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 80 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

Parameters:

- bucketId: The ID of the bucket in which the data object shall be deleted.
- Query: A generic (Java) object (see Listing 66) with attributes set to identify the data objects to be updated.

Return Value:

True on success

Error Handling:

In case of an error, false is returned

Remarks:

The Java interface manipulates generic Java objects. In Listing 36, the signature of the method for the CRUD-operation crudDelete is shown as well as an example for the usage of this method.

Listing 36: Source Code Example – API Method Signature
for the CRUD-Operation crudDelete and the Usage of this Method

```

/**
 * @param bucketId, the ID of the bucket in which the data object shall be deleted
 * @param object, query object with attributes set to identify the data objects to *
 * be deleted
 * @return true on success
 */
public boolean crudDelete(String bucketId, DTO object)

...

//Creates test data to query for data objects which shall be deleted
MyTestClass testQuery = new MyTestClass();

//Call of a Helper Method. The Helper method is a factory method which creates
//a DTO object. The Following arguments are required:
//    -the URI to a valid XSD file describing the class of the object
//    -the object instance itself
//    -the package name of the class of the object
DTO query = Helper.GenerateGenericObject("res/TestSchema.xsd", testQuery,
"junit.test.vo");
boolean success = api.crudDelete(BUCKET_TEST_ID, query);

```

The exemplary usage of the method at the bottom of Listing 36 creates first a data object with attributes to identify the data objects in the Cloud Storage component, which shall be deleted. If the API call is successful, true is returned, otherwise false.

5.2.5.2.7 Add Access Rights for User

To grant users (or user groups, henceforth the word user is used synonym for users and user groups in the context of access rights) access rights for a specific bucket the API provide the method `addAccessRight`. The method has three parameters, the ID of the bucket to which the access rights shall be granted, an ID, which identifies the user or a group of users, and third the right, which shall be granted.

Parameters:

- `bucketId`: The ID of the bucket to which the right will be granted.
- `id`: The user or group ID to whom the access right should be granted. The value `NotSet` will delete the access right.
- `right`: The access right which will be granted to the user.

Return Value:

True on success

Error Handling:

In case of an error, false is returned

Remarks:

Only the owner of the bucket or a user with the “Super” access right can add access rights to a bucket. In Listing 37, the signature of the method to add access rights for a user is shown as well as an example for the usage of this method.

Listing 37: Source Code Example – API Method Signature to Grant Access Rights to a User for a Bucket and the Usage of this Method

```
/**
 * @param bucketId, the ID of the bucket to which the right will be granted
 * @param id, the user or group ID to whom the access right should be granted
 *         the value NotSet will delete the access right
 * @param right, the right to grant to the user or group for the bucket
 * @return true on success
 */
public boolean addAccessRight(String bucketId, String id, Rights right)

...

//adds the Read-AccessRight to the specific bucket for the specific user
boolean success = api.addAccessRight(BUCKET_TEST_ID, SECOND_USER_NAME, Rights.READ);
```

The exemplary usage of the method at the bottom of Listing 37 grants the access right “READ” to the specific user for the bucket with the provided ID. If the API call is successful, true is returned, otherwise false.

5.2.5.2.8 Get Access Rights for User

In order to retrieve the access rights for a specific user, the API provides the method `getAccessRights`. The method has one parameter, the ID of the user of whom the access rights shall be queried.

Parameters

- `id`: The user or group ID of whom the access right should be queried.

Return Value:

An ArrayList of Right values (see Listing 61)

Error Handling:

In case of an error, the list is empty

Remarks:

None

In Listing 38, the signature of this method is shown as well as an example of its usage.

Listing 38: Source Code Example – API Method Signature to Get Access Rights for a User and the Usage of this Method

```
/**
 * @param id, the user or group ID of whom the access right should be queried
 * @return a list of Rights for the user or group ID
 */
public ArrayList<Rights> getAccessRights(String id)

...

//Retrieves the access rights for the specific user
ArrayList<Rights> rights = api.getAccessRights(SECOND_USER_NAME);
```

The exemplary usage of the method at the bottom of Listing 38 retrieves the access rights for the user with the provided ID. All found access rights are stored in the list.

5.2.5.2.9 Execute Query on Bucket

In order to execute a generic query to utilize specific features of one the database backends, the API provides the method `executeQuery`. The method has two parameters, the `bucketId` identifying the bucket on which the query should be executed and the query itself as a string.

Parameters:

- `bucketId`: The ID of the bucket on which the query will be executed.
- `query`: The query as string.

Return Value:

The result of the query as string.

Error Handling:

In case of an error, a null value is returned

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 83 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

Remarks:

The queries will not be parsed nor evaluated before executing, allowing the possible destruction of all data in a database.

In Listing 39, the signature this method is shown as well as an example of its usage.

Listing 39: Source Code Example – API Method Signature to Execute a Generic Query on a Specific Bucket and the Usage of this Method

```
/**
 * @param bucketId, the ID of the bucket on which the query will be executed
 * @param query, the query as string
 * @return the result of the query as string
 */
public String executeQuery (String bucketId, String query)

...

//Retrieves the result of a generic query for the specific bucket
String result = api.executeQuery(BUCKET_TEST_ID, QUERY_STRING);
```

The exemplary usage of the method at the bottom of Listing 39 retrieves the access rights for the user with the provided ID. All found access rights are stored in the list.

5.2.5.2.10 Add Observer to Bucket

In order to get notified when the content of a bucket changed, the API provides the method `addObserver`. The method has two parameters, the `bucketId` identifying the bucket which shall be monitored and the URL as callback to a web service in case of a change in the bucket.

Parameters:

- `bucketId`: The ID of the bucket which shall be observed.
- `url`: The callback URL to the observer.

Return Value:

True on success

Error Handling:

In case of an error, false is returned

Remarks:

None

In Listing 40, the signature this method is shown as well as an example of its usage.

Listing 40: Source Code Example – API Method Signature to Add an Observer to a Specific Bucket and the Usage of this Method

```
/**
 *
 * @param bucketId, the ID of the bucket which shall be observed
 * @param url, the callback URL to the observer
 * @return true on success
 */
public boolean addObserver(String bucketId, String url)
...

//Adds an observer to a specific bucket
boolean success = api.addObserver(BUCKET_TEST_ID, OBSERVER_CALLBACK);
```

The exemplary usage of the method at the bottom of Listing 40 adds an observer to the provided ID.

5.2.5.2.11 Remove Observer from Bucket

In order to get not notified anymore when the content of a bucket changed, the API provides the method `removeObserver`. The method has two parameters, the `bucketId` identifying the bucket which is monitored and the URL as callback to a web service. The second parameter is used to distinct the correct observer to be removed.

Parameters

- `bucketId`: The ID of the bucket from which the observer shall be removed.
- `url`: The callback URL to the observer (used to identify the correct observer).

Return Value:

True on success

Error Handling:

In case of an error, false is returned

Remarks:

None

In Listing 41, the signature of this method is shown as well as an example of its usage.

Listing 41: Source Code Example – API Method Signature
to Remove an Observer from a Specific Bucket and the Usage of this Method

```
/**
 *
 * @param bucketId, the ID of the bucket from which the observer
 * shall be removed
 * @param url, the callback URL to the observer
 * (used to identify the correct observer)
 * @return true on success
 */
public boolean removeObserver(String bucketId, String url)

...

//Removes an observer from a specific bucket
boolean success = api.removeObserver(BUCKET_TEST_ID, OBSERVER_CALLBACK);
```

The exemplary usage of the method at the bottom of Listing 41 removes the observer from the specific bucket.

5.2.6 Content Format

5.2.6.1 JSON Schema

This section lists the JSON schemas which will be used by the Cloud-based Information Infrastructure internally as well as for the communication with other components via the RESTful and Java interfaces.

5.2.6.1.1 Data Transfer Objects

All Data Transfer Objects (DTO) will be used for the transfer of structured data for the Cloud-based Information Infrastructure (internally and externally). The list of objects described in this section is not complete and subject to changes to accommodate the special needs of external components which will use the Cloud-based Information Infrastructure.

The DTO Schema (see Listing 42) acts as the superclass for all objects which will be used to transfer data. In the context of Java, this will be utilized to support generic objects without losing all advantages of typed objects.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 86 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

Listing 42: JSON Schema – DTO Object Description

```
{
  "type": "object",
  "id": "http://simpli-city.eu/CloudStorage/JSON-Schema/DTO",
  "properties": {
    "description": {
      "type": "string",
      "id": "http://simpli-city.eu/CloudStorage/JSON-Schema/description",
      "required": false
    },
    "link": {
      "type": "string",
      "id": "http://simpli-city.eu/CloudStorage/JSON-Schema/link",
      "required": false
    }
  }
}
```

The Generic Object (see Listing 43) can be used in conjunction with the Class Object (see Listing 44) and the Class Instance (see Listing 45) to transfer generic objects described and instantiated with XSD and XML. In conjunction with JAXB, this provides the possibility to easily transfer Java objects.

Listing 43: JSON Schema – Generic Object Object Description

```
{
  "type": "object",
  "id": "http://simpli-city.eu/CloudStorage/JSON-Schema/GenericObject",
  "properties": {
    "classDescription": {
      "type": "object",
      "id": "http://simpli-city.eu/CloudStorage/JSON-Schema/classDescription",
      "required": false,
      "properties": {
        "classBinary": {
          "type": "object",
          "required": true
        },
        "namespace": {
          "type": "string",
          "required": true
        }
      }
    },
    "instance": {
      "type": "object",
      "id": "http://simpli-city.eu/CloudStorage/JSON-Schema/ClassInstance",
      "required": true
    }
  }
}
```

Listing 44: JSON Schema – Class Object Object Description

```
{
  "type": "object",
  "id": "http://simpli-city.eu/CloudStorage/JSON-Schema/ClassObject",
  "properties": {
    "classBinary": {
      "type": "object",
      "required": true
    },
    "namespace": {
      "type": "string",
      "required": true
    }
  }
}
```

Listing 45: JSON Schema – Class Instance Object Description

```
{
  "type": "object",
  "id": "http://simpli-city.eu/CloudStorage/JSON-Schema/ClassInstance",
  "properties": {
    "instance": {
      "type": "object",
      "required": true
    }
  }
}
```

Listing 46 shows an instance of a Generic Object of the JSON Schema shown in Listing 43. The binary as well as the instance are encoded as a base64 string.

Listing 46: JSON Example Instance – Generic Object

```
{
  "classDescription": {
    "classBinary": [
      "HFJD[...]6JDZS"
    ],
    "namespace": "eu.simplicity.test"
  },
  "instance": {
    "instance": [
      "LADYB[...]DASI3"
    ]
  }
}
```

The JSON Schema for a bucket object (see Listing 47) can be utilized as representation of a bucket. The different values for the bucket type are congruent with the bucket types described in Section 5.2.1.

Listing 47: JSON Schema – Bucket Object Description

```

{
  "type": "object",
  "id": "http://simpli-city.eu/CloudStorage/JSON-Schema/Bucket",
  "properties": {
    "accessRights": {
      "type": "array",
      "required": false,
      "items": {
        "type": "object",
        "id": "http://simpli-city.eu/CloudStorage/JSON-Schema/accessRights",
        "required": false,
        "properties": {
          "id": {
            "type": "string",
            "required": true
          },
          "right": {
            "type": {
              "enum": [
                "NotSet",
                "Denied",
                "Read",
                "Write",
                "Super"
              ]
            },
            "required": true
          }
        }
      }
    },
    "bucketId": {
      "type": "string",
      "required": true
    },
    "bucketType": {
      "type": {
        "enum": [
          "Structured",
          "SemiStructured",
          "Semantic"
        ]
      },
      "required": true
    }
  }
}

```

Listing 48 shows an instance for a bucket instance as described in the JSON Schema in Listing 47. The bucket representation has an ID, a bucket type and three access rights are associated with this bucket.

Listing 48: JSON Example Instance – Bucket

```
{
  "bucketId": "BucketTestId",
  "bucketType": "SemiStructured",
  "accessRights": [
    {
      "id": "user1",
      "right": "Denied"
    },
    {
      "id": "user2",
      "right": "Write"
    },
    {
      "id": "public",
      "right": "Read"
    }
  ]
}
```

The AccessRight Schema (see Listing 49) will be utilized to describe the access rights for a user or group for a bucket. The ID identifies a user (or group) the right defines the possible rights granted to the user (or group).

The different values for the right are defined as follows:

- *NotSet* will be used to delete an access right.
- *Denied* will be used to deny a user or group the any access to a bucket.
- *Read* will be used to allow a user or group to read the contents of a bucket.
- *Write* will be used to allow a user or group to read and write the contents of a bucket.
- *Super* will be used to allow a user or group the same rights as the owner. This includes the right to grand or restrict access rights to other user or groups as well as to delete the bucket. It is only restricted in the way, that it will not be possible for a user or group with this access right to remove the “Super” access right of the owner of the bucket (This right is granted to the owner without a specific access right, hence a restriction will not be possible).

For an example of an AccessRight JSON instance, see the encapsulated instance in the example for a bucket JSON instance in Listing 48.

Listing 49: JSON Schema – AccessRight Object Description

```

{
  "type": "object",
  "id": "http://simpli-city.eu/CloudStorage/JSON-Schema/AccessRight",
  "properties": {
    "id": {
      "type": "string",
      "required": true
    },
    "right": {
      "type": {
        "enum": [
          "NotSet",
          "Denied",
          "Read",
          "Write",
          "Super"
        ]
      },
      "required": true
    }
  }
}

```

The Key Value Object Schema (see Listing 50) will be utilized to save simple and complex key/value pairs. The values can be a simple string or a complex JSON object.

Listing 50: JSON Schema – Key Value Object Object Description

```

{
  "type": "object",
  "id": "http://simpli-city.eu/CloudStorage/JSON-Schema/KeyValueObject",
  "required": false,
  "properties": {
    "key": {
      "type": "string",
      "required": true
    },
    "value": {
      "type": "string",
      "required": true
    }
  }
}

```

Listing 51 shows a simple example instance of the JSON Schema for a Key Value Object as described in Listing 50.

Listing 51: JSON Example Instance – Key Value Object

```

{
  "key": "myKey",
  "value": "TestValue"
}

```

5.2.6.1.2 Commands

The Command JSON schemes described in this section are utilized to encapsulated structured data in order to communicate commands to the Cloud-based Information Infrastructure. They encapsulate JSON instance objects described with JSON Schema of Section 5.2.6.1.1.

The Command JSON Schema described in Listing 52 is the superclass for all Commands. It defines that all command objects require a DTO instance as parameter.

Listing 52: JSON Schema – Command Description

```
{
  "type": "object",
  "id": "http://simpli-city.eu/CloudStorage/JSON-Schema/Command",
  "properties": {
    "parameter": {
      "type": "object",
      "id": "http://simpli-city.eu/CloudStorage/JSON-Schema/DTO",
      "required": true,
      "properties": {
        "description": {
          "type": "string",
          "required": false
        },
        "link": {
          "type": "string",
          "required": false
        }
      }
    }
  }
}
```

The Create Bucket command JSON Schema described in Listing 53 can be utilized in the creation of a new bucket in the Cloud-based Information Infrastructure. This command only requires a bucket JSON instance with a bucketId.

Listing 53: JSON Schema – Create Bucket Command Description

```

{
  "type": "object",
  "id": "http://simpli-city.eu/CloudStorage/JSON-Schema/CreateBucket",
  "properties": {
    "bucket": {
      "type": "object",
      "id": "http://simpli-city.eu/CloudStorage/JSON-Schema/bucket",
      "required": true,
      "properties": {
        "accessRights": {
          "type": "array",
          "required": false,
          "items": {
            "type": "object",
            "id": "http://simpli-city.eu/CloudStorage/JSON-
Schema/bucket/accessRights",
            "properties": {
              "id": {
                "type": "string",
                "required": true
              },
              "right": {
                "type": {
                  "enum": [
                    "NotSet",
                    "Denied",
                    "Read",
                    "Write",
                    "Super"
                  ]
                },
                "required": true
              }
            }
          }
        },
        "bucketId": {
          "type": "string",
          "required": true
        },
        "bucketType": {
          "type": {
            "enum": [
              "Structured",
              "SemiStructured",
              "Semantic"
            ]
          },
          "required": true
        }
      }
    }
  }
}

```

The Delete Bucket command JSON schema described in Listing 54 can be utilized for the deletion of an existent bucket in the Cloud-based Information Infrastructure. This command only requires a bucket JSON instance with a valid bucketId.

Listing 54: JSON Schema – Delete Bucket Command Description

```
{
  "type": "object",
  "id": "http://simpli-city.eu/CloudStorage/JSON-Schema/DeleteBucket",
  "properties": {
    "bucket": {
      "type": "object",
      "id": "http://simpli-city.eu/CloudStorage/JSON-Schema/bucket",
      "required": true,
      "properties": {
        "bucketId": {
          "required": true
        }
      }
    }
  }
}
```

The CRUD operation command JSON Schema described in Listing 55 can be utilized to execute a CRUD operation in a bucket existing already in the Cloud-based Information Infrastructure. This command requires a bucket JSON instance with a bucketId, the operation type and for the Update operation additionally a query object.

Listing 55: JSON Schema – CRUD Operation Command Description

```
{
  "type": "object",
  "id": "http://simpli-city.eu/CloudStorage/JSON-Schema/CrudOperation",
  "properties": {
    "bucket": {
      "type": "object",
      "id": "http://simpli-city.eu/CloudStorage/JSON-Schema/bucket",
      "required": true,
      "properties": {
        "bucketId": {
          "type": "string",
          "required": true
        }
      }
    },
    "operation": {
      "type": {
        "enum": [
          "Create",
          "Read",
          "Write",
          "Delete"
        ]
      },
      "required": true
    },
    "query": {
      "type": "object",
      "id": "http://simpli-city.eu/CloudStorage/JSON-Schema/DT0",
      "required": false,
      "properties": {
        "description": {
          "type": "string",
          "required": false
        },
        "link": {
          "type": "string",
          "required": false
        }
      }
    }
  }
}
```

The Add Access Right Command JSON Schema in Listing 56 can be utilized to execute the Add Access Right for User or Group operation in a bucket existing already in the Cloud-based Information Infrastructure. This command requires a JSON bucket instance with a valid bucketId, as well as AccessRight instance with an identifying string for the user or group as well as the actual right, which shall be granted.

Listing 56: JSON Schema – Add Access Right Command Description

```
{
  "type": "object",
  "id": "http://simpli-city.eu/CloudStorage/JSON-Schema/AddAccessRight",
  "properties": {
    "bucket": {
      "type": "object",
      "id": "http://simpli-city.eu/CloudStorage/JSON-Schema/bucket",
      "required": false,
      "properties": {
        "bucketId": {
          "type": "string",
          "id": "http://simpli-city.eu/CloudStorage/JSON-
Schema/bucket/bucketId",
          "required": false
        }
      }
    },
    "right": {
      "type": "object",
      "id": "http://simpli-city.eu/CloudStorage/JSON-Schema/AccessRight",
      "required": true,
      "properties": {
        "id": {
          "type": "string",
          "required": true
        },
        "right": {
          "type": {
            "enum": [
              "NotSet",
              "Denied",
              "Read",
              "Write",
              "Super"
            ]
          }
        },
        "required": true
      }
    }
  }
}
```


The Get Access Right Command JSON schema in Listing 57 can be utilized to execute the Get Access Right operation for a bucket existing already in the Cloud-based Information Infrastructure. This command requires a JSON bucket instance with a valid bucketId to identify the bucket of which the information should be queried.

Listing 57: JSON Schema – Get Access Rights Command Description

```
{
  "type": "object",
  "id": "http://simpli-city.eu/CloudStorage/JSON-Schema/GetAccessRights",
  "properties": {
    "bucket": {
      "type": "object",
      "id": "http://simpli-city.eu/CloudStorage/JSON-Schema/bucket",
      "required": true,
      "properties": {
        "bucketId": {
          "required": true
        }
      }
    }
  }
}
```

The Query Operation Command JSON schema in Listing 58 can be utilized to execute the query operation for a bucket existing already in the Cloud-based Information Infrastructure. This command requires a JSON bucket instance with a valid bucketId to identify the bucket in which the query will be executed as well as the query as a string.

Listing 58: JSON Schema – Query Operation Command Description

```
{
  "type": "object",
  "id": "http://simpli-city.eu/CloudStorage/JSON-Schema/QueryOperation",
  "properties": {
    "bucket": {
      "type": "object",
      "id": "http://simpli-city.eu/CloudStorage/JSON-Schema/bucket",
      "required": true,
      "properties": {},
      "bucketId": {
        "type": "string",
        "id": "http://simpli-city.eu/CloudStorage/JSON-Schema/bucket/bucketId",
        "required": false
      }
    },
    "query": {
      "type": "string",
      "required": true
    }
  }
}
```

5.2.6.2 Java Classes

The Java classes described in this section are corresponding to the JSON Schema described in the previous section (see Section 5.2.6.1). The Java classes have JAXB annotations in order to allow an easier transformation between different representations of the object instances.

Figure 8 shows exemplary the UML class diagram for two mandatory DTO with appended enumeration description.

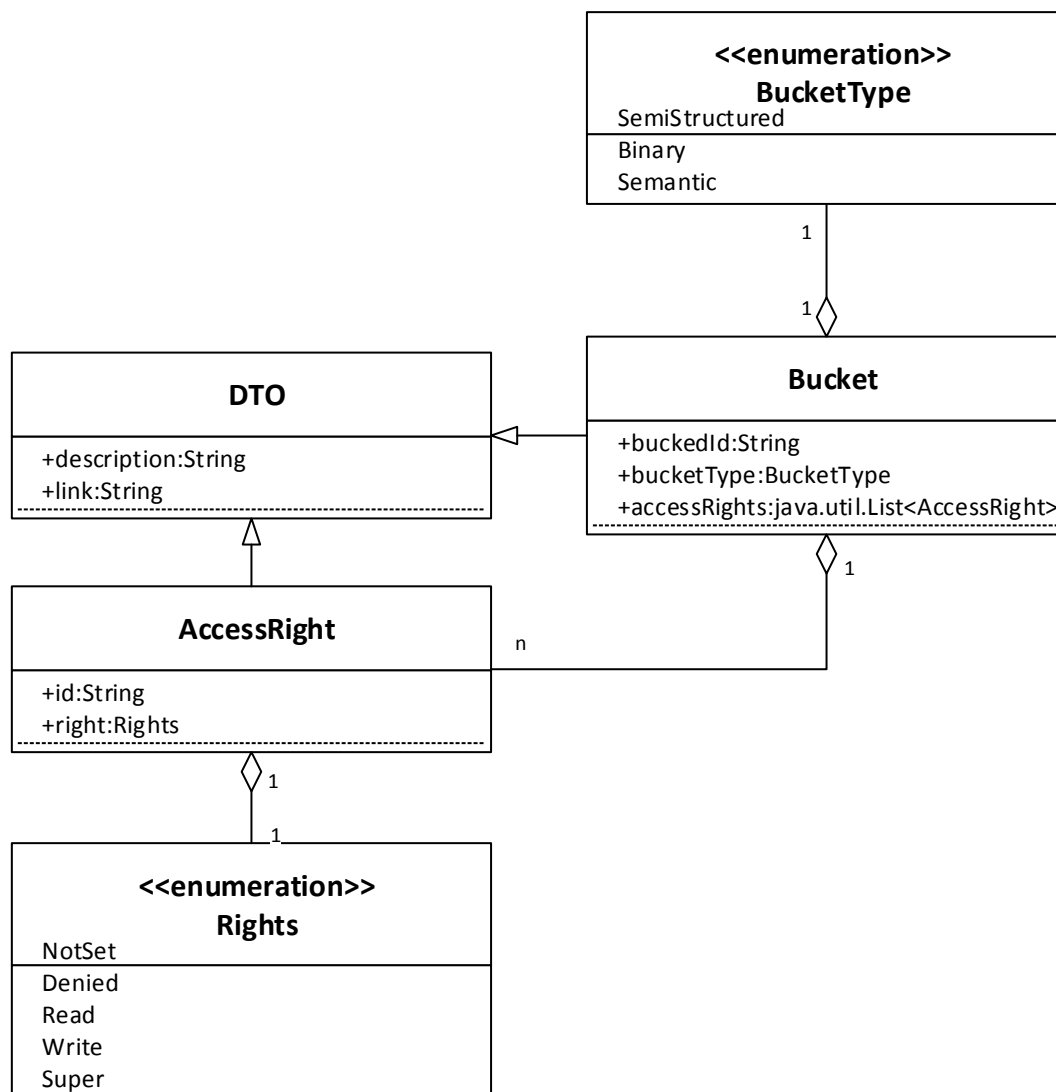


Figure 8: UML Class Diagram – DTO Bucket and AccessRight with Appended Enumeration

5.2.6.2.1 Enumeration

The BucketType enum (see Listing 59) contains the three different enumerations for the Bucket Type.

Listing 59: Java Class – Bucket Type Enumeration

```
@XmlType(name = "BucketType")
@XmlEnum
public enum BucketType {

    @XmlEnumValue("SemiStructured")
    SEMI_STRUCTURED("SemiStructured"),
    @XmlEnumValue("Binary")
    BINARY("Binary"),
    @XmlEnumValue("Semantic")
    SEMANTIC("Semantic");
    private final String value;

    BucketType(String v) {
        value = v;
    }
    public String value() {
        return value;
    }

    public static BucketType fromValue(String v) {
        for (BucketType c: BucketType.values()) {
            if (c.value.equals(v)) {
                return c;
            }
        }
        throw new IllegalArgumentException(v);
    }
}
```

The CrudType enumeration (see Listing 60) contains the four different enumerations for the CRUD operation.

Listing 60: Java Class – CRUD Type Enumeration

```
@XmlType(name = "CrudType")
@XmlEnum
public enum CrudType {

    @XmlEnumValue("Create")
    CREATE("Create"),
    @XmlEnumValue("Read")
    READ("Read"),
    @XmlEnumValue("Update")
    UPDATE("Update"),
    @XmlEnumValue("Delete")
    DELETE("Delete");
    private final String value;

    CrudType(String v) {
        value = v;
    }
    public String value() {
        return value;
    }
    public static CrudType fromValue(String v) {
        for (CrudType c: CrudType.values()) {
            if (c.value.equals(v)) {
                return c;
            }
        }
        throw new IllegalArgumentException(v);
    }
}
```

The Rights enumeration (see Listing 61) contains the five different values for the Access Rights as defined in Listing 49.

Listing 61: Java Class – Rights Enumeration

```
@XmlType(name = "Rights")
@XmlEnum
public enum Rights {

    @XmlEnumValue("NotSet")
    NOT_SET("NotSet"),
    @XmlEnumValue("Denied")
    DENIED("Denied"),
    @XmlEnumValue("Read")
    READ("Read"),
    @XmlEnumValue("Write")
    WRITE("Write"),
    @XmlEnumValue("Super")
    SUPER("Super");
    private final String value;

    Rights(String v) {
        value = v;
    }
    public String value() {
        return value;
    }
    public static Rights fromValue(String v) {
        for (Rights c: Rights.values()) {
            if (c.value.equals(v)) {
                return c;
            }
        }
        throw new IllegalArgumentException(v);
    }
}
```

5.2.6.2.2 Data Transfer Objects

All Data Transfer Objects (DTO) will be used for the transfer of structured data for the Cloud-based Information Infrastructure (internally and externally). The list of objects described in this section is not complete and subject to changes to accommodate the special needs of the components.

The Java class DTO (see Listing 62) is the superclass for all objects which will be used to transfer data. In the context of Java, this will be utilized to support generic objects without losing all advantages of typed objects.

Listing 62: Java Class – DTO Superclass

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "DTO", propOrder = { "description", "link" })
@XmlSeeAlso({
    ClassObject.class,
    Bucket.class,
    AccessRight.class,
    KeyValueObject.class,
    ClassInstance.class
})
public class DTO {
    protected String description;
    protected String link;
    /**
     * Gets the value of the description property.
     * @return the description
     */
    public String getDescription() {
        return description;
    }
    /**
     * Sets the value of the description property.
     * @param value, description to be set
     */
    public void setDescription(String value) {
        this.description = value;
    }
    /**
     * Gets the value of the link property.
     * @return the link
     */
    public String getLink() {
        return link;
    }
    /**
     * Sets the value of the link property.
     * @param value, link to be set
     */
    public void setLink(String value) {
        this.link = value;
    }
}
```

The Java class AccessRight (see Listing 63) is used to transfer information regarding the access rights of a user or group for a bucket.

Listing 63: Java Class – DTO for Access Right

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "AccessRight", propOrder = { "id", "right" })
public class AccessRight extends DTO
{
    @XmlElement(name = "Id", required = true, defaultValue = "public")
    protected String id;
    @XmlElement(required = true, defaultValue = "Denied")
    protected Rights right;

    /**
     * Gets the value of the id property.
     * @return the id
     */
    public String getId() {
        return id;
    }
    /**
     * Sets the value of the id property.
     * @param value, id to be set
     */
    public void setId(String value) {
        this.id = value;
    }
    /**
     * Gets the value of the right property.
     * @return the right
     */
    public Rights getRight() {
        return right;
    }
    /**
     * Sets the value of the right property.
     * @param value, the right to be set
     */
    public void setRight(Rights value) {
        this.right = value;
    }
}
```

The Java class Bucket (see Listing 64) is used to transfer information of a bucket.

Listing 64: Java Class – DTO for Bucket

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "Bucket", propOrder = { "bucketId", "bucketType", "accessRights"})
public class Bucket extends DTO
{

    @XmlElement(required = true)
    protected String bucketId;
    protected BucketType bucketType;
    protected List<AccessRight> accessRights;

    /**
     * Gets the value of the bucketId property.
     * @return the bucketId
     */
    public String getBucketId() {
        return bucketId;
    }
    /**
     * Sets the value of the bucketId property.
     * @param value, bucketId to be set
     */
    public void setBucketId(String value) {
        this.bucketId = value;
    }
    /**
     * Gets the value of the bucketType property.
     * @return the bucketType
     */
    public BucketType getBucketType() {
        return bucketType;
    }
    /**
     * Sets the value of the bucketType property.
     * @param value, bucketType to be set
     */
    public void setBucketType(BucketType value) {
        this.bucketType = value;
    }
    /**
     * Gets the value of the accessRights property.
     * For example, to add a new item, do as follows:
     * getAccessRights().add(newItem);
     */
    public List<AccessRight> getAccessRights() {
        if (accessRights == null) {
            accessRights = new ArrayList<AccessRight>();
        }
        return this.accessRights;
    }
}
```


The Java class `KeyValueObject` (see Listing 65) is used to transfer Key Value information.

Listing 65: Java Class – DTO for Key-Value Object

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "KeyValueObject", propOrder = { "key", "value" })
public class KeyValueObject extends DTO
{
    @XmlElement(required = true)
    protected String key;
    @XmlElement(required = true)
    protected String value;

    /**
     * Gets the value of the key property.
     * @return the key
     */
    public String getKey() {
        return key;
    }

    /**
     * Sets the value of the key property.
     *
     * @param value, key to be set
     */
    public void setKey(String value) {
        this.key = value;
    }

    /**
     * Gets the value of the value property.
     * @return the value
     */
    public String getValue() {
        return value;
    }

    /**
     * Sets the value of the value property.
     *
     * @param value, value to be set
     */
    public void setValue(String value) {
        this.value = value;
    }
}
```

Listing 66, Listing 67 and Listing 68 are DTO used in conjunction with each other. They can be utilized to transfer Objects described in Java (or other XSD compatible Languages) to be transferred between a component and the Cloud Storage. Listing 66 is a wrapper for the class description and the instance object.

Listing 66: Java Class – DTO for Generic (Java) Objects

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "GenericObject", propOrder = { "classDescription", "instance"})
public class GenericObject extends DTO
{
    protected ClassObject classDescription;
    @XmlElement(required = true)
    protected byte[] instance;
    /**
     * Gets the value of the classDescription property.
     * @return the classDescription object
     */
    public ClassObject getClassDescription() {
        return classDescription;
    }
    /**
     * Sets the value of the classDescription property.
     *
     * @param value, ClassObject to be set
     */
    public void setClassDescription(ClassObject value) {
        this.classDescription = value;
    }
    /**
     * Gets the value of the instance property.
     * @return the instance object byte[]
     */
    public byte[] getInstance() {
        return instance;
    }
    /**
     * Sets the value of the instance property.
     * @param value, instance to be set
     */
    public void setInstance(byte[] value) {
        this.instance = value;
    }
}
```

Listing 67 is the description for the Class Object used for the generic (Java) object description.

Listing 67: Java Class – DTO for the Description
of the Class Object Required for Generic (Java) Objects

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "ClassObject", propOrder = { "classBinary", "namespace" })
public class ClassObject extends DTO
{
    @XmlElement(required = true)
    protected byte[] classBinary;
    @XmlElement(required = true)
    protected String namespace;
    /**
     * Gets the value of the classBinary property.
     *
     * @return the classBinary
     */
    public byte[] getClassBinary() {
        return classBinary;
    }
    /**
     * Sets the value of the classBinary property.
     * @param value, classBinary to be set
     */
    public void setClassBinary(byte[] value) {
        this.classBinary = value;
    }
    /**
     * Gets the value of the namespace property.
     * @return the namespace string
     */
    public String getNamespace() {
        return namespace;
    }
    /**
     * Sets the value of the namespace property.
     *
     * @param value, the namespace to be set
     */
    public void setNamespace(String value) {
        this.namespace = value;
    }
}
```

Listing 68 is the description for the Class Instance used for the generic (Java) object description.

Listing 68: Java Class – DTO for the Description
of the Class Instance Required for Generic (Java) Objects

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "ClassInstance", propOrder = { "classDescription", "instance" })
public class ClassInstance extends DTO
{
    protected ClassObject classDescription;
    @XmlElement(required = true)
    protected byte[] instance;
    /**
     * Gets the value of the classDescription property.
     * @return the classDescription object
     */
    public ClassObject getClassDescription() {
        return classDescription;
    }
    /**
     * Sets the value of the classDescription property.
     *
     * @param value, classDescription to be set
     */
    public void setClassDescription(ClassObject value) {
        this.classDescription = value;
    }
    /**
     * Gets the value of the instance property.
     * @return the instance object
     */
    public byte[] getInstance() {
        return instance;
    }
    /**
     * Sets the value of the instance property.
     * @param value, instance to be set
     */
    public void setInstance(byte[] value) {
        this.instance = value;
    }
}
```

5.2.6.2.3 Commands

The Command classes described in this chapter are utilized to encapsulate structured data in order to communicate commands to the Cloud-based Information Infrastructure.

The Java class Command (see Listing 69) is used the superclass for all Commands.

Listing 69: Java Class – The Superclass for Commands

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "Command", propOrder = { "parameter" })
@XmlSeeAlso({
    DeleteBucket.class,
    CrudOperation.class,
    CreateBucket.class,
    QueryOperation.class,
    AddAccessRight.class,
    AddOberserver.class,
    GetAccessRights.class,
    RemoveOberserver.class
})
public class Command {

    protected DTO parameter;
    /**
     * Gets the value of the parameter property.
     *
     * @return the parameter
     */
    public DTO getParameter() {
        return parameter;
    }
    /**
     * Sets the value of the parameter property.
     * @param value, parameter to be set
     */
    public void setParameter(DTO value) {
        this.parameter = value;
    }
}
```

The Java class CreateBucket (see Listing 70) is used to encapsulate the command to create a new Bucket in the Cloud Storage.

Listing 70: Java Class – Command to Create a Bucket

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "CreateBucket", propOrder = { "bucket" })
public class CreateBucket extends Command
{
    @XmlElement(required = true)
    protected Bucket bucket;
    /**
     * Gets the value of the bucket property.
     * @return the bucket
     */
    public Bucket getBucket() {
        return bucket;
    }
    /**
     * Sets the value of the bucket property.
     * @param value, bucket to be set
     */
    public void setBucket(Bucket value) {
        this.bucket = value;
    }
}
```

The Java class DeleteBucket (see Listing 71) is used to encapsulate the command to delete an existing bucket from the Cloud Storage.

Listing 71: Java Class – Command to Delete a Bucket

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "DeleteBucket", propOrder = { "bucket" })
public class DeleteBucket extends Command
{
    @XmlElement(required = true)
    protected Bucket bucket;
    /**
     * Gets the value of the bucket property.
     * @return the bucket
     */
    public Bucket getBucket() {
        return bucket;
    }
    /**
     * Sets the value of the bucket property.
     * @param value, bucket to be set
     */
    public void setBucket(Bucket value) {
        this.bucket = value;
    }
}
```

The Java class CrudOperation (see Listing 72) is used to encapsulate all CRUD operation commands, which will be executed on an existing bucket in the Cloud Storage.

Listing 72: Java Class – Command for CRUD Operations

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "CrudOperation", propOrder = { "bucket", "operation", "query" })
public class CrudOperation extends Command
{
    @XmlElement(required = true)
    protected Bucket bucket;
    @XmlElement(required = true)
    protected CrudType operation;
    protected DTO query;
    /**
     * Gets the value of the bucket property.
     * @return the bucket
     */
    public Bucket getBucket() {
        return bucket;
    }
    /**
     * Sets the value of the bucket property.
     * @param value, bucket to be set
     */
    public void setBucket(Bucket value) {
        this.bucket = value;
    }
    /**
     * Gets the value of the operation property.
     * @return the operation
     */
    public CrudType getOperation() {
        return operation;
    }
    /**
     * Sets the value of the operation property.
     * @param value, CrudType to be set
     */
    public void setOperation(CrudType value) {
        this.operation = value;
    }
    /**
     * Gets the value of the query property.
     * @return the query object
     */
    public DTO getQuery() {
        return query;
    }
    /**
     * Sets the value of the query property.
     * @param value, query object to be set
     */
    public void setQuery(DTO value) {
        this.query = value;
    }
}
```

The Java class `AddAccessRight` (see Listing 73) is used to encapsulate the command to add access rights to an existing bucket in the Cloud Storage.

Listing 73: Java Class – Command to Add Access Rights to a Bucket

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "AddAccessRight", propOrder = {
    "bucket",
    "right"
})
public class AddAccessRight
    extends Command
{

    @XmlElement(required = true)
    protected Bucket bucket;
    @XmlElement(required = true)
    protected AccessRight right;

    /**
     * Gets the value of the bucket property.
     * @return the bucket
     */
    public Bucket getBucket() {
        return bucket;
    }

    /**
     * Sets the value of the bucket property.
     * @param value, bucket to be set
     */
    public void setBucket(Bucket value) {
        this.bucket = value;
    }

    /**
     * Gets the value of the right property.
     * @return value, AccessRight to be set
     */
    public AccessRight getRight() {
        return right;
    }

    /**
     * Sets the value of the right property.
     * @param value, right to be set
     */
    public void setRight(AccessRight value) {
        this.right = value;
    }

}
```


The Java class `GetAccessRights` (see Listing 74) is used to encapsulate the command to get the access rights for an existing bucket in the Cloud Storage.

Listing 74: Java Class – Command to Get Access Rights

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "GetAccessRights", propOrder = { "id" })
public class GetAccessRights extends Command
{
    @XmlElement(required = true)
    protected String id;
    /**
     * Gets the value of the id property.
     * @return the id
     */
    public String getId() {
        return id;
    }
    /**
     * Sets the value of the id property.
     *
     * @param value, id to be set
     */
    public void setId(String value) {
        this.id = value;
    }
}
```

The Java class QueryOperation (see Listing 75) is used to encapsulate the command to execute a generic query on an existing bucket in the Cloud Storage.

Listing 75: Java Class – Command to Execute a Query on a Bucket

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "QueryOperation", propOrder = { "bucket", "query", "description" })
public class QueryOperation extends Command
{
    @XmlElement(required = true)
    protected Bucket bucket;
    @XmlElement(required = true)
    protected String query;
    protected String description;
    /**
     * Gets the value of the bucket property.
     * @return the bucket
     */
    public Bucket getBucket() {
        return bucket;
    }
    /**
     * Sets the value of the bucket property.
     * @param value, bucket to be set
     */
    public void setBucket(Bucket value) {
        this.bucket = value;
    }
    /**
     * Gets the value of the query property.
     * @return query string
     */
    public String getQuery() {
        return query;
    }
    /**
     * Sets the value of the query property.
     * @param value, query string to be set
     */
    public void setQuery(String value) {
        this.query = value;
    }
    /**
     * Gets the value of the description property.
     * @return the description
     */
    public String getDescription() {
        return description;
    }
    /**
     * Sets the value of the description property.
     * @param value, description to be set
     */
    public void setDescription(String value) {
        this.description = value;
    }
}
```

The Java class AddObserver (see Listing 76) is used to encapsulate the command to add an observer to an existing bucket in the Cloud Storage.

Listing 76: Java Class – Command to Add an Observer to a Bucket

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "AddObserver", propOrder = { "bucket" })
public class AddObserver extends Command
{
    @XmlElement(required = true)
    protected Bucket bucket;
    /**
     * Gets the value of the bucket property.
     * @return the bucket
     */
    public Bucket getBucket() {
        return bucket;
    }
    /**
     * Sets the value of the bucket property.
     * @param value, bucket to be set
     */
    public void setBucket(Bucket value) {
        this.bucket = value;
    }
}
```

The Java class RemoveObserver (see Listing 77) is used to encapsulate the command to remove an existing observer from an existing bucket in the Cloud Storage.

Listing 77: Java Class – Command to Remove an Observer from a Bucket

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "RemoveObserver", propOrder = { "bucket" })
public class RemoveObserver extends Command
{
    @XmlElement(required = true)
    protected Bucket bucket;
    /**
     * Gets the value of the bucket property.
     * @return the bucket
     */
    public Bucket getBucket() {
        return bucket;
    }
    /**
     * Sets the value of the bucket property.
     * @param value, bucket to be set
     */
    public void setBucket(Bucket value) {
        this.bucket = value;
    }
}
```

5.2.7 Summary

In this section, the Cloud-based Information Infrastructure was technically specified. During the technology selection (Section 5.2.3), MongoDB, Sesame and Amazon S3 were selected to store different types of data. An abstraction of these specific storage systems will enable each component and each backend service to create a bucket in one of these data storages to store its data without knowledge of the backend systems. The access to a bucket will be implemented, managed with an Access List Control feature to share data with several components. To control the Access Control List, an interface is given, where rights can be added, removed and queried. So each component can use the Access Control List to query the rights of a specific user. The management of buckets and the dispatching of the according data to the backend systems will be handled by the Cloud-based Information Infrastructure internally.

According to this Specification, the following components have to be implemented:

- RESTful Interface to expose the Cloud Storage Façade to other SIMPLI-CITY components
- Cloud Storage Façade to provide CRUD and query functionality
- Cloud Storage Backend with Access Control List feature
- Bucket Abstractors for the three mentioned Storage Backend Systems
- JAVA API for implementation ease, i.e., for the Application Runtime Environment

The technical specification of the Cloud-based Information Infrastructure had a big focus on scalability, data privacy and the separation of concerns as mentioned in Section 5.2.1. The Access Control Lists will give the opportunity to restrict access to data while the abstraction into buckets separates the data type concerns. To provide scalability, responsiveness and reliability, the Storage Backends were chosen carefully.

5.3 Sensor Abstraction and Interoperability Interfaces

5.3.1 Major Design Decisions

The Sensor Abstraction and Interoperability Interfaces are the core component within SIMPLI-CITY to integrate external sensor data sources. As these sensors and the corresponding sensor data usually are rather heterogeneous, for example ranging from different parking lot data to data from traffic lights, etc., a homogeneous access method has to be provided for this data to be of any use. Furthermore, an integrative access method eases the efforts for software/service developers who want to exploit these data sources. The function of the Sensor Abstraction and Interoperability Interfaces component is to provide this integrative access. This component will provide the seamless integration of heterogeneous sensor sources and sensor readings within SIMPLI-CITY, e.g., by providing corresponding wrappers.

The Sensor Abstraction and Interoperability Interfaces also provide access to sensors connected to the PMA and user-related data, accessible on the PMA, e.g. contacts and calendar data to other SIMPLI-CITY server components. This functionality is supported through the PMA-based Sensor Abstraction, which is in fact a subcomponent of the Sensor Abstraction and Interoperability Interfaces that is executed as a background service on the PMA and is responsible for allowing access from the server side Sensor

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 116 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

Abstraction and Interoperability Interfaces component to the local data sources on the PMA. The PMA-based Sensor Abstraction is separately discussed in Section 7.3.

During the discussion of the Global Architecture (deliverable D3.1) and the Functional Specification (deliverable D3.2.1), the following major design decisions have been made:

Extensible Data Model:

Due to the heterogeneity of possible sensor data sources, the structure and complexity of single sensor readings can differ. The used data model must be easily extensible with respect to new data sources. The structure of values, e.g., concatenated values, and the amount of single value entries must be adaptable, e.g., for complex sensor readings combined of several single values.

Distributed Approach:

From the top-level view, sensors can be grouped in two different categories, user centric sensors and sensors deployed in the environment. While server side components can integrate sensors and wrap respective data, the local sensors also have to be available if no network connection is available. This leads to the necessity of a distributed sensor access approach where a component on the PMA enables to interface local sensors and provide the respective data to locally running Apps.

Ease of Use:

The used solution should be intuitive and easy to use for application and service developer. The integration of external sensor data sources is a key functionality in SIMPLI-CITY. Developers should be able to integrate these sources without large effort. The used data formats should be human readable and easy to understand.

Scalability:

As the potential amount of external sensor data sources is extremely high, the system must be able to deal with a very large number of sensor sources and should allow a location-based pre-filtering for sensor requests to reduce the amount of data transfers.

5.3.2 Technology Comparison

5.3.2.1 Comparison Criteria

Table 25: Criteria for Technical Specification

Parameter	Importance	Description
Extensible Data Model	++	Different data sources will have different data formats with different data structures. To be able to include all available data sources, the SIMPLI-CITY Sensor Abstraction and Interoperability Interfaces must provide an extensible data model.
Ease of Use	++	To enable service and application developers to integrate external sensor data sources in an easy way, it is important to provide simple interfaces and easy to understand data structures.

Platform (Portability)	+	The main components should be platform independent to be portable to another server infrastructure.
Lightweight	--	The server side Sensor Abstraction and Interoperability Interfaces have in principle no restrictions for resource aware behaviour. The solution does not have to be lightweight to achieve higher prioritized comparison criteria parameters.
Fast Response Time	++	The integration of sensor data is a key functionality within many SIMPLI-CITY apps and services. Since these could depend on nearly real time data, the response time of the Sensor Abstraction and Interoperability Interfaces should be very fast.

5.3.2.2 Possible Technologies and Comparison

A unified approach to sensor-based data sources is currently not provided by commercial tools, which in fact offer mostly proprietary approaches restricted to particular technologies. To develop the server side Sensor Abstraction and Interoperability Interfaces, SIMPLI-CITY will therefore first analyse existing research-driven software prototypes and frameworks.

Sensor Abstraction Layer:

Sensor Abstraction Layer (SAL) [GA07] is a software architecture for heterogeneous sensor networks, constructed to run on the gateway of a sensing network. It hides the disparities of involved technologies behind a hardware-independent interface, that is easy to extend and can reduce the implementation effort.

Sensor Abstraction and Integration Layer:

The Sensor Abstraction and Integration Layer (SAIL) [GLF+08] is – encapsulated within the OSGi framework – a design for heterogeneous Wireless Sensor Networks (WSNs), with the purpose of context awareness. A WSN is considered as a context information source. Therefore SAIL provides a node centric and a data centric approach.

Cougar:

The approach of Cougar³¹ is to map a sensor network on a database, with an easy to use query system similar to SQL. It is usable with a large amount of sensors, but is based on global knowledge of the network and is therefore not very suitable for mobile applications. Also there is a lack of openness and support for heterogeneous hardware.

Mate:

Mate³² uses a VM based approach, supporting mobility via various ad hoc routing protocols. Its main focus is the energy efficient reprogramming of sensor nodes. Besides architecture there is only byte code available, so a higher-level language is needed for application development.

³¹ <http://www.cs.cornell.edu/bigreddata/cougar/index.php>

³² <http://www.cs.berkeley.edu/~pal/research/mate.html>

Impala:

Impala [LM03] makes use of state machines to choose the adequate protocol for communication. Through its modular approach, it reduces transmission overhead and increases energy efficiency. The downside of Impala is its special implementation, running only on a Hewlett-Packard/Compaq handheld.

Mires:

Mires [SGV+06] is a message-oriented, WSN applicable middleware, using asynchronous communication and a publish-subscribe mechanism. It is implemented for the sensor node operating system TinyOS and thus supports multiple hardware platforms.

Table 26: Comparison of Technologies for
(Server Side) Sensor Abstraction and Interoperability Interfaces

Parameter	Importance	SAL	SAIL	Cougar	Mate	Impala	Mires
Generic Criteria							
Up-to-Datedness	+	7	5	4	2	2	4
Stability	+	3	3	6	5	6	4
Extensibility & Open Source/Standards	++	4	6	5	4	2	2
Familiarity	++	4	2	5	2	2	3
Performance	+	7	4	4	4	5	4
Interoperability	++	6	6	4	2	2	3
License	(e.g., Apache 2.0)	Copyleft	OSGI specific license	Proprietary	Proprietary	free	Proprietary
Specific Criteria							
Lightweight	--	6	3	2	7	5	3
Ease of Use	++	5	3	7	2	2	5
Fast Response Time	++	6	5	6	4	5	4
Extensible Data Model	++	5	5	3	5	2	2
Platform (Portability)	+	5	4	2	2	1	7

5.3.3 Technology Selection

5.3.3.1 Selection of the Sensor Abstraction and Interoperability Interfaces

All discussed frameworks provide some degree of functionality useful for SIMPLI-CITY. However, none of these solutions fits to the specific requirements of the project. All of them are built for special scenarios, mainly pure sensor networks. Especially the integration of mobile clients and car sensors is a fundamental functionality within SIMPLI-CITY and is not covered by these products. This leads to the necessity of a custom solution that will be developed to fit all needs.

The Service Runtime Environment will make use of OSGi and some components of the Sensor Abstraction and Interoperability Interfaces will be services. Here the previously mentioned SAL and SAIL would fit, but cannot be used because the source code is not publicly available. Because of the requirements of the project that the Sensor Abstraction and Interoperability Interfaces have to be fast and especially extensible to a variety of

more sensor types, it is recommended to create a full custom solution rather than to combine different components which do not fit in the end that well. Especially the response time and the interaction with the PMA are constraints that are not fulfilled by any available solution.

5.3.3.2 Missing Elements and Implementation Needs

Since it was not possible to identify an appropriate available technology, the whole component will be implemented from scratch. The custom approach brings a lot of effort but also a lot of benefits. With a custom approach the integration of the built-in PMA sensors and attached car sensors can be integrated much more seamlessly. A multitude of sensor and data sources have to be accessible by a variety of requesters. That demands an access technology which supports parallelization. This characteristic is not only required by many sensor and data sources, also plenty of users and apps want to have access to the server. This means that the whole system needs to be scalable in a way that it can manage the workload that occurs by this possibly large number of requests. Not only current sensor values have to be accessible, likewise historical data shall be provided which leads to a huge amount of data. Because of this, sensor values need to be stored in a compact way and duplicates and redundancy must be avoided. The main task of the system is to provide not only access to simple sensor types, e.g., temperature sensors, but also to manage the access to user calendars and contact data as sensor values. This means that many different types of sensors need to be handled and exported in a common format by the system. This capability is important to integrate new sensor sources in the future that are not known now.

Furthermore, the connection between the PMA and the server side has to be implemented in an energy efficient way. A static connection between the PMA and the server side is not desired because this would drain off the battery of the PMA very fast. This leads to the problem that the server needs to find the PMA to build up a connection and does not know when it will receive an answer. Because of that this is mainly depending on whether the PMA is connected to the Internet or not and also the network address of the PMA might be unknown. Furthermore, the system needs to be able to use multiple communication channels that can change over time due to the fact that the PMA is a mobile device. Thus, an energy efficient and fast built-up connection between the server side and the PMA needs to be realized. In addition, the access to car sensors is something partly new and not standardized until now. All these challenges lead to the necessity to use multiple different interface technologies in such a way as to enable the different components to communicate amongst each other.

5.3.3.3 Further Information and Conclusion from Technology Comparison

As mentioned before, the Sensor Abstraction and Interoperability Interfaces will be a custom implementation. Nevertheless we will make use of concepts of the named technologies in Section 5.3.2 as far as possible and reasonable. But there is still a lot of work to be done. The main purpose of the Sensor Abstraction and Interoperability Interfaces is to build a bridge between sensor nodes, PMAs, the information storage for historical data, but also applications and other SIMPLI-CITY components which request for access to data as fast as possible. This leads to a complex infrastructure of different components that need to communicate between each other. Some of these problems have

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 120 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

been previously processed in part by other researchers but no common solution has been presented until now.

5.3.4 Component Structure

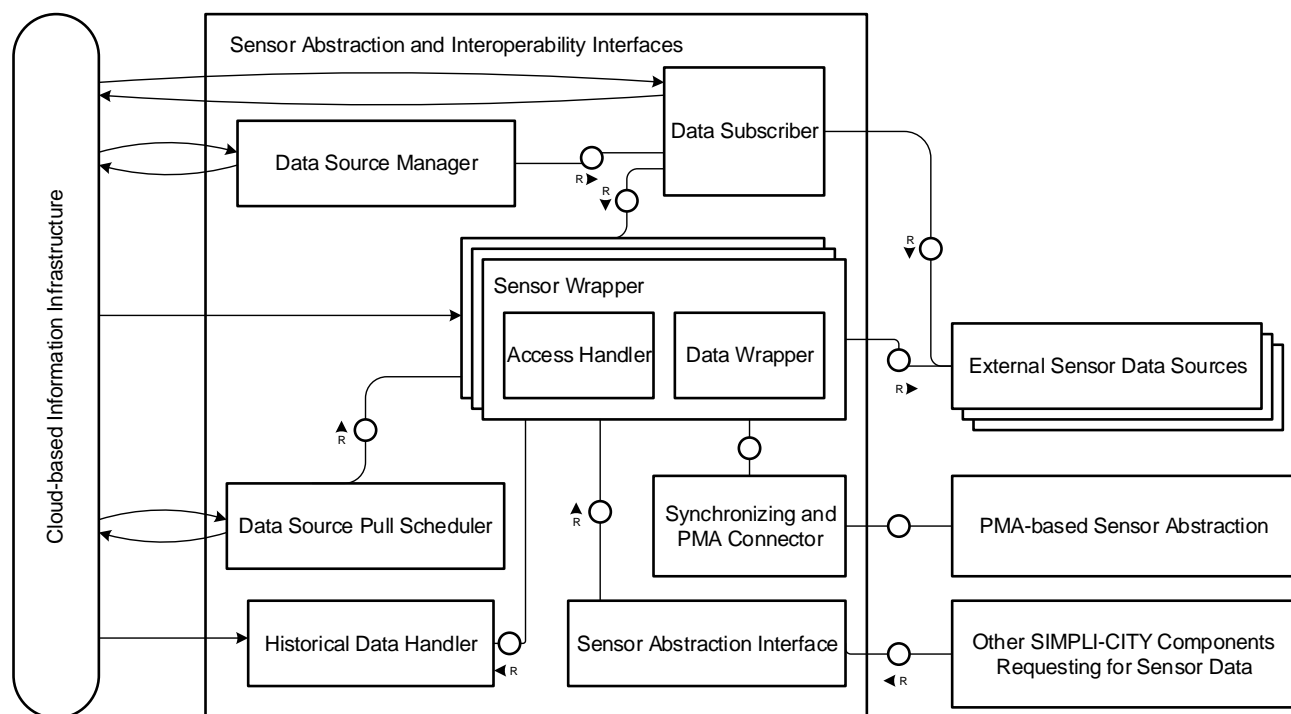


Figure 9: Component Structure of the
(Server Side) Sensor Abstraction and Interoperability Interfaces

The component structure has been updated in comparison with D3.2.1 and is depicted in Figure 9. Since a custom implementation is intended, there have been only minor changes.

The figure is simplified with respect to External Sensor Data Sources. There is only one box depicted as placeholder for all external data sources to increase the readability; these external data sources include sensors, static data sources, sources with historical data, the PMA-based Sensor Abstraction, etc. Also all Other SIMPLI-CITY Components Requesting for Sensor Data are depicted together in only one box; this includes the Service Runtime Environment and the Data Processing component. The new subcomponent Data Source Manager is introduced. This subcomponent of the Sensor Abstraction and Interoperability Interfaces will act as master control over all subcomponents that interact with external data sources.

The Sensor Abstraction and Interoperability Interfaces will run on the same physical machines as the Service Runtime Environment to achieve a seamless integration with all other SIMPLI-CITY components.

To achieve the functionalities, the Sensor Abstraction and Interoperability Interfaces are providing the following subcomponents as depicted in the figure above. The Cloud-based Information Infrastructure is used as central storage for all subcomponents.

5.3.4.1 Sensor Wrapper

The Sensor Wrapper is the basic component to access external sensor data sources and translate the external data format into the SIMPLI-CITY data format. It mainly consists of the two subcomponents Access Handler, which is responsible for accessing the external sensor data source, and the Data Wrapper that provides the wrapping facilities for transforming diverse proprietary sensor data to data (formats) usable by the other SIMPLI-CITY components, respectively SIMPLI-CITY apps and services.

5.3.4.2 Synchronizing and PMA Connector

The Synchronizing and PMA Connector component has two major functionalities. On the one hand, it is responsible to synchronize some of the PMA or user-related data with a local buffer on the PMA to make it available even if the PMA is offline. On the other hand, it is responsible for establishing a connection to the PMA.

5.3.4.3 Sensor Abstraction Interface

This component provides the API that is exposing the interfaces to other SIMPLI-CITY components, respectively apps and services which exploit sensors and sensor data respectively. These APIs are provided as Java interfaces for all server side SIMPLI-CITY components running on the same machine and RESTful interfaces to be available for services on other physical machines via the network connection.

5.3.4.4 Data Source Pull Scheduler

The Data Source Pull Scheduler takes care of pulling data from sensors according to a pre-defined time interval. Pulling for sensor data is optional and will be used for sensor data sources that are only available at certain times and for the case that historical data of a sensor data source should be available. The time intervals for pulling the external sensor data are configured individually for all sources. The retrieved data will be stored in the Cloud-based Information Infrastructure for later usage.

5.3.4.5 Data Subscriber

This component is responsible for receiving data triggered by an external event. It provides the functionality to subscribe to external event bus systems and allows receiving event triggered sensor readings.

5.3.4.6 Historical Data Handler

The Historical Data Handler is responsible for collecting historical sensor data from the Cloud-based Information Infrastructure. Data can be requested for a specific time interval and for a per-sensor granularity.

5.3.4.7 Data Source Manager

The Data Source Manager is responsible for configuring the Data Source Pull Scheduler, the Data Subscriber and the Historical Data Handler and to observe the sensor data stored in the Cloud-based Information Infrastructure to prevent storage of unnecessary data.

5.3.5 Interfaces

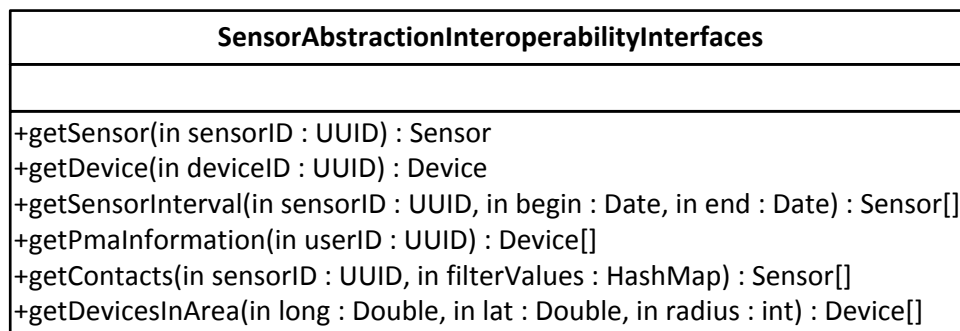


Figure 10: Class Diagram of the (Server Side)
Sensor Abstraction Interoperability Interfaces

The interfaces to access the sensor data will be, as mentioned before, available in Java for other SIMPLI-CITY components running on the same machine and as RESTful interfaces to access the Sensor Abstraction and Interoperability Interfaces via a network connection. For the interfaces, it is assumed that an authentication of the user is assured and the user ID is provided additionally for each API call. The RESTful interfaces are described in the following Section 5.3.5.1 and the Java interfaces in the subsequent Section 5.3.5.2.

5.3.5.1 RESTful Interfaces

To describe the RESTful Interface of the Sensor Abstraction and Interoperability Interfaces, each service call of the interface is in the following described in a separate table.

5.3.5.1.1 Get Sensor Data

The RESTful interface Get Sensor Data (see Table 27) returns the actual value of the specified sensor. The input parameter is the unique ID of the respective sensor.

Table 27: RESTful Interface Description – Get Sensor Data

Method	GET	URL	\$API_ROOT/sensor
Description	Returns the sensor information of the sensor with the respective ID		
Parameter	id	Required	yes Possible values 128 bit UUID
Example URL	\$API_ROOT/sensor?id=528739A0-F508-4551-A12A-04A9B51718D0		
Response	HTTP header + optional text/JSON		
Response value	(HTTP status code) + (JSON data)	Required	yes Possible values 200 204 404
		Description	200: Value returned 204: No Value available. 404: ID not found.
JSON Object	http://Simpli-City.eu/SensorAbstraction/JSON-Schema/Sensor		
Example Response	HTTP/1.1 200 OK		

Listing 78 is an example for a valid JSON object that is the return value of the previously mentioned request. The object describes the sensor type, the value with all necessary parameters and an optional accuracy. The value can also be an array of single values that describe a more complex concatenated value. For more details regarding the JSON data format see deliverable D4.1.1 and Section 5.3.6.

Listing 78: JSON Example – Return Value of Get Sensor Data

```
{
  "object": "sensor",
  "sensorType": {
    "type": "vehicle",
    "subtype": "engineSpeed"
  },
  "sensorValue": [
    {
      "name": "engineSpeedRPM",
      "value": 3254,
      "unit": "rpm",
      "accuracy": {
        "unit": "percentage",
        "value": 0.05
      }
    }
  ],
  "objectID": "528739A0-F508-4551-A12A-04A9B51718D0",
  "timeStamp": "2013-08-28T10:27:10.000Z",
  "parent": "ID-BBB"
}
```

5.3.5.1.2 Get Device Data

The RESTful interface Get Device Data (see Table 28) returns a description of the requested device. The input parameter is the unique ID of the respective device.

Table 28: RESTful Interface Description – Get Device Data

Method	GET	URL	\$API_ROOT/device				
Description	Returns the description of the device: Ids of connected devices and sensors						
Parameter	id	Required	yes	Possible values	128 bit UUID	Description	Unique ID of the requested device
Example URL	\$API_ROOT/device?id=528739A0-F508-4551-A12A-04A9B51718D1						
Response	HTTP header + optional text/JSON						
Response value	(HTTP status code) + (JSON data)	Required	yes	Possible values	200 204 404	Description	200: Value returned 204: No Value available. 404: ID not found.
JSON Object	http://Simpli-City.eu/SensorAbstraction/JSON-Schema/Device						
Example Response	HTTP/1.1 200 OK						

Listing 79 is an example for a valid JSON object that is the return value of the previously mentioned request. The object describes the device type with all connected devices and sensors. The hierarchy is that a sensor always belongs to a device and a device can have directly connected additional devices. Thus the list of sensors and devices is an array and can contain several entries. A parent is a pointer to a device this device belongs to. Thus the PMA has no value for this parameter but a connected device will have the PMA's ID as entry for the parameter parent. For more details regarding the JSON data format see deliverable D4.1.1 and Section 5.3.6.

Listing 79: JSON Example – Return Value of Get Device Data

```
{
  "object": "device",
  "deviceType": "PMA",
  "connectedSensors": [
    {
      "sensorType": {
        "type": "location",
        "subType": "coordinates"
      },
      "sensorID": "ID-SENSOR-CCC"
    },
    {
      "sensorType": {
        "type": "location",
        "subType": "address"
      },
      "sensorID": "ID-SENSOR-DDD"
    }
  ],
  "connectedDevices": [
    {
      "deviceType": "car",
      "deviceID": "528739A0-F508-4551-A12A-04A9B51718AA"
    }
  ],
  "timeStamp": "2013-08-28T10:27:10.000Z",
  "objectID": "528739A0-F508-4551-A12A-04A9B51718D0",
  "parent": null
}
```

5.3.5.1.3 Get Sensor Interval

The RESTful interface Get Sensor Interval (see Table 29) returns the values of the specified sensor within the given interval. The input parameters are the unique ID of the respective sensor and start and endpoint in time of the requested interval. The time format is ISO8601. The return value is an array of all available values within the given interval.

Table 29: RESTful Interface Description – Get Sensor Interval

Method	GET	URL	\$API_ROOT/sensorInterval				
Description	Returns all available sensor values within the given interval of the sensor with the respective ID						
Parameter	id	Required	yes	Possible values	128 bit UUID	Description	Unique ID of the requested sensor
Parameter	begin	Required	yes	Possible values	time in ISO_8601	Description	Start time of requested sensor data interval
Parameter	end	Required	yes	Possible values	time in ISO_8601	Description	End time of requested sensor data interval
Example URL	\$API_ROOT/sensorInterval?id=528739A0-F508-4551-A12A-04A9B51718D0&begin=2013-08-28T10:27:10.000Z&end=2013-08-28T10:37:10.000Z						
Response	HTTP header + optional text/JSON						
Response value	(HTTP status code) + (JSON data)	Required	yes	Possible values	200 204 404	Description	200: Value returned 204: No Value available. 404: ID not found.
JSON Object	array of http://Simpli-City.eu/SensorAbstraction/JSON-Schema/Sensor						
Example Response	HTTP/1.1 200 OK						

Listing 80 is an example for a valid JSON object that is the return value of the previously mentioned request. The object describes the sensor type, all available values within the given interval with all necessary parameters and an optional accuracy. Each Value can also be an array of single values that describe a more complex concatenated value. For more details regarding the JSON data format see deliverable D4.1.1 and Section 5.3.6.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 125 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

Listing 80: JSON Example – Return Value of Get Device Data

```
{
  "object": "sensor",
  "sensorType": {
    "type": "location",
    "subtype": "coordinates"
  },
  "sensorValue": [
    {
      "name": "latitude",
      "value": "50.00526032632832",
      "unit": "degree",
      "accuracy": {
        "unit": "meters",
        "value": 24
      }
    },
    {
      "name": "longitude",
      "value": "8.647010091683981",
      "unit": "degree",
      "accuracy": {
        "unit": "meters",
        "value": 24
      }
    }
  ],
  "objectID": "528739A0-F508-4551-A12A-04A9B51718D1",
  "timeStamp": "2013-08-28T10:27:10.000Z",
  "parent": "528739A0-F508-4551-A12A-04A9B51718D0"
}
```

5.3.5.1.4 Get PMA Information

The RESTful interface Get PMA Information (see Table 30) returns the device description of the user's PMA, i.e., the IDs of the sensors and devices connected to the user's PMA including the IDs of the users PMA itself. The input parameter is the unique ID of the user.

Table 30: RESTful Interface Description – Get Devices Connected to the PMA

Method	GET	URL	\$API_ROOT/pmaInformation				
Description	Returns the device description of the PMA of a user, i.e., the device IDs and sensor IDs of the users PMA						
Parameter	userID	Required	yes	Possible values	128 bit UUID	Description	Unique ID of the respective user
Example URL	\$API_ROOT/pmaInformation?userID=528739A0-F508-4551-A12A-04A9B51718D1						
Response	HTTP header + optional text/JSON						
Response value	(HTTP status code) + (JSON data)	Required	yes	Possible values	200 204 404	Description	200: Value returned 204: No Value available. 404: ID not found.
JSON Object	http://Simpli-City.eu/SensorAbstraction/JSON-Schema/Device						
Example Response	HTTP/1.1 200 OK						

Listing 81 is an example for a valid JSON object that is the return value of the previously mentioned request. The object describes the PMA device with all connected devices and sensors. The hierarchy is that a sensor always belongs to a device and a device can have directly connected additional devices. Thus the list of sensors and devices is an array and can contain several entries. A parent is a pointer to a device this device belongs to. Thus, the PMA has no value for this parameter but a connected device will have the PMA's ID as entry for the parameter parent. For more details regarding the JSON data format see deliverable D4.1.1 and Section 5.3.6.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 126 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

Listing 81: JSON Example – Return Value of Get PMA Information

```
{
  "object": "device",
  "deviceType": "PMA",
  "connectedSensors": [
    {
      "sensorType": {
        "type": "location",
        "subType": "coordinates"
      },
      "sensorID": "528739A0-F508-4551-A12A-04A9B51718D4"
    },
    {
      "sensorType": {
        "type": "location",
        "subType": "address"
      },
      "sensorID": "528739A0-F508-4551-A12A-04A9B51718D5"
    }
  ],
  "connectedDevices": [
    {
      "deviceType": "car",
      "deviceID": "528739A0-F508-4551-A12A-04A9B51718D6"
    }
  ],
  "timeStamp": "2013-08-28T10:27:10.000Z",
  "objectID": "528739A0-F508-4551-A12A-04A9B51718D7",
  "parent": null
}
```

5.3.5.1.5 Get Contact Data

The RESTful interface Get Contact Data (see Table 31) returns the contact data of a particular user. Data sources for personal data are handled as virtual sensors of the PMA and the corresponding IDs are returned amongst others by the method *getPmaInformation* (see Section 5.3.5.2.4). To allow requesting a particular data set, e.g., the contact data of one particular person, this method allows filtering the data source for particular values. Thus the input parameters are the unique ID of the user and a simple JSON object that contains the filter parameters as array of Key Value pairs.

Table 31: RESTful Interface Description – Get Contact Data

Method	POST	URL	\$API_ROOT/contactData			
Description	Returns the requested contact data					
Parameter	userID	Required	yes	Possible values	128 bit UUID	Description Unique ID of the respective user
JSON Object	array of http://Simpli-City.eu/SensorAbstraction/JSON-Schema/contactFilter					
Example URL	\$API_ROOT/contactData?userID=528739A0-F508-4551-A12A-04A9B51718D1					
Response	HTTP header + optional text/JSON					
Response value	(HTTP status code) + (JSON data)	Required	yes	Possible values	200 204 404	Description 200: Value returned 204: No Value available. 404: ID not found.
JSON Object	array of http://Simpli-City.eu/SensorAbstraction/JSON-Schema/Sensor					
Example Response	HTTP/1.1 200 OK					

Listing 82 shows an example JSON message necessary to request for a particular contact of the user.

Listing 82: JSON Example – Input Message as Key Value Pairs as
Filter of Get Contact Data

```
[
  {
    "Key": "Name",
    "Value": "Smith"
  },
  {
    "Key": "Street",
    "Value": "Main Street"
  }
]
```

Listing 83 is an example for a valid array of JSON objects that is the return value of the previously mentioned request. The array describes the contacts that match to the given filter. For more details regarding the JSON data format see deliverable D4.1.1 and Section 5.3.6.

Listing 83: JSON Example – Return Value of Get Contact with Filter

```
[
  {
    "object": "sensor",
    "sensorType": {
      "type": "userData",
      "subtype": "contact"
    },
    "sensorValue": [
      {
        "name": "name",
        "value": "Smith",
        "unit": null,
        "accuracy": null
      },
      {
        "name": "firstName",
        "value": "John",
        "unit": null,
        "accuracy": null
      },
      {
        "name": "mobilePhone",
        "value": "5553798",
        "unit": null,
        "accuracy": null
      },
      {
        "name": "street",
        "value": "Main Street",
        "unit": null,
        "accuracy": null
      },
      {
        "name": "houseNumber",
        "value": "10",
        "unit": null,
        "accuracy": null
      },
      {
        "name": "postalCode",
        "value": "64283",
        "unit": null,
        "accuracy": null
      },
      {
        "name": "country",
        "value": "Germany",
        "unit": null,
        "accuracy": null
      }
    ],
    "objectID": "762739A0-F772-3651-A1F2-04A9B5171E72",
    "timeStamp": "2013-08-28T10:27:10.000Z",
    "parent": "528739A0-F508-4551-A12A-04A9B51F528E"
  }
]
```

5.3.5.1.6 Get Devices in Area

The RESTful interface Get Devices in Area (see Table 32) returns the device description of all devices in a certain area. Mobile devices like the users PMA will not be included. The input parameters are the latitude and longitude of the centre of the requested area and a radius in meters around this centre point to determine the size of the requested area. Each device provides a location sensor to request the device', and respectively the sensors, position.

Table 32: RESTful Interface Description – Get Devices in Area

Method	GET	URL	\$API_ROOT/devicesInArea				
Description	Returns the device IDs and description within a certain area						
Parameter	lat	Required	yes	Possible values	latitude	Description	Latitude of the point of interest
Parameter	long	Required	yes	Possible values	longitude	Description	Longitude of the point of interest
Parameter	radius	Required	yes	Possible values	radius	Description	Radius in meters
Example URL	\$API_ROOT/devicesInArea?lat=50.00526032632832&long=8.647010091683981&radius=450						
Response	HTTP header + optional text/JSON						
Response value	(HTTP status code) + (JSON data)	Required	yes	Possible values	200 204 404	Description	200: Value returned 204: No Value available. 404: ID not found.
JSON Object	array of http://Simpli-City.eu/SensorAbstraction/JSON-Schema/Device						
Example Response	HTTP/1.1 200 OK						

Listing 84 is an example for a valid array of JSON objects that is the return value of the previously mentioned request. The array describes the devices within the requested area. For more details regarding the JSON data format see deliverable D4.1.1 and Section 5.3.6.

Listing 84: JSON Example – Return Value of Get Devices in Area

```
[
  {
    "object": "device",
    "deviceType": "parkingLot",
    "connectedSensors": [
      {
        "sensorType": {
          "type": "parking",
          "subtype": "parkingLot"
        },
        "sensorID": "528739A0-F508-4551-A12A-04A9B5171818"
      },
      {
        "sensorType": {
          "type": "location",
          "subType": "coordinates"
        },
        "sensorID": "528739A0-F508-4551-A12A-04A9B51718A1"
      }
    ],
    "connectedDevices": null,
    "timeStamp": "2013-08-28T10:27:10.000Z",
    "objectID": "528739A0-F508-4551-A12A-04A9B5171819",
    "parent": "528739A0-F508-4551-A12A-04A9B5171810"
  },
  {
    "object": "device",
    "deviceType": "parkingLot",
    "connectedSensors": [
      {
        "sensorType": {
          "type": "parking",
          "subtype": "parkingLot"
        },
        "sensorID": "528739A0-F508-4551-A12A-04A9B5171822"
      },
      {
        "sensorType": {
          "type": "location",
          "subType": "coordinates"
        },
        "sensorID": "528739A0-F508-4551-A12A-04A9B5171E31"
      }
    ],
    "connectedDevices": null,
    "timeStamp": "2013-08-28T10:27:10.000Z",
    "objectID": "528739A0-F508-4551-A12A-04A9B5171823",
    "parent": "528739A0-F508-4551-A12A-04A9B5171810"
  }
]
```

5.3.5.2 Java Interfaces

All interfaces of the Sensor Abstraction and Interoperability Interfaces are also available as Java interfaces. These interfaces are equivalent to the previously describes RESTful interfaces. Each Java interface is described in detail in the following by the description of the input and output parameters and a code example.

5.3.5.2.1 Get Sensor Data

The Java interface Get Sensor Data (see Listing 85) returns the actual data of a particular sensor.

Parameters:

- **sensorID:** UUID of the respective sensor

Return Value:

The sensor data as `eu.simpli-city.Sensor` object.

Error Handling:

A `SensorNotFoundException` is thrown in case the sensor ID is invalid.

Remarks:

None

Listing 85: Source Code Example – Get Sensor Data

```
/**
 * Requests for the actual data of a particular sensor
 *
 * @param sensorId of the sensor
 * @throws SensorNotFoundException in case the sensor ID is invalid
 */
public Sensor getSensor(UUID sensorID) throws SensorNotFoundException;

...

// Request sensor
try {
    Sensor mySensor = getSensor(sensorID);
} catch (SensorNotFoundException e) {
    ...
}
```

5.3.5.2.2 Get Device Data

The Java interface Get Device Data (see Listing 86) returns the actual data of a particular device.

Parameters:

- deviceId: UUID of the respective device

Return Value:

The sensor data as eu.simpli-city.Device object.

Error Handling:

A DeviceNotFoundException is thrown in case the device ID is invalid.

Remarks:

None

Listing 86: Source Code Example – Get Device Data

```
/**
 * Requests for the actual data of a particular device
 *
 * @param deviceId of the sensor
 * @throws DeviceNotFoundException in case the device ID is invalid
 */
public Device getDevice(UUID deviceId) throws DeviceNotFoundException;

...

// Request device
try {
    Device myDevice = getDevice(deviceID);
} catch (SensorNotFoundException e) {
    ...
}
```

5.3.5.2.3 Get Sensor Interval

The Java interface Get Sensor Interval (see Listing 87) returns the data of a particular sensor in a given interval.

Parameters:

- sensorID: UUID of the respective sensor
- begin: Start point in time of the requested Interval as java.util.Date
- end: End point in time of the requested Interval as java.util.Date

Return Value:

The sensor data as array of eu.simpli-city.Sensor objects.

Error Handling:

A SensorNotFoundException is thrown in case the sensor ID is invalid.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 133 / 435
http://www.simpli-city.eu/ Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201				

Remarks:

None

Listing 87: Source Code Example – Get Sensor Interval

```
/**
 * Requests for the actual data of a particular sensor
 *
 * @param sensorId of the sensor
 * @param begin of interval as java.util.Date
 * @param end of interval as java.util.Date
 * @throws SensorNotFoundException in case the sensor ID is invalid
 */
public Sensor[] getSensorInterval(UUID sensorID, Date begin, Date end)
                                throws SensorNotFoundException;

...

// Request sensor interval
try {
    Sensor[] mySensors = getSensorInterval(sensorID, Date begin, Date end);
} catch (SensorNotFoundException e) {
    ...
}
```

5.3.5.2.4 Get PMA Information

The Java interface Get PMA Information (see Listing 87) provides the IDs of the devices and sensors connected to the PMA of a particular user.

Parameters:

- userID: UUID of the respective device

Return Value:

The device data of the PMA as eu.simpli-city.Device object.

Error Handling:

A DeviceNotFoundException is thrown in case the user ID is invalid or the respective PMA device cannot be found.

Remarks:

None

Listing 88: Source Code Example – Get PMA Information

```

/**
 * Requests for the actual data of a particular device
 *
 * @param userId of the of the user whose PMA is requested
 * @throws DeviceNotFoundException in case the user ID is invalid or the
 * respective PMA device cannot be found
 *
 */
public Device getPmaInformation(UUID userID) throws DeviceNotFoundException;

...

// Request PMA device
try {
    Device myDevice = getPmaInformation (userID);
} catch (noDeviceNotFoundException e) {
    ...
}

```

5.3.5.2.5 Get Contact Data

Data sources for personal data are handled as virtual sensors of the PMA and the corresponding IDs are returned amongst others by the method *getPmaInformation* (see last subsection). To allow requesting a particular data set, e.g., the contact data of one particular person, this method allows filtering the data source for particular values.

Parameters:

- sensorID: UUID of the respective sensor, i.e., the virtual contacts sensor
- filterValues: (key,value) pairs to filter the data source as Strings in a HashMap, e.g., (street, Mainstreet) or (name, Smith)

Return Values:

The contact data as array eu.simpli-city.Sensor objects (one array object per matched contact). If no available contact matches the filterValues, the return value is null. This is a special case for the use of a virtual sensor. Here, each contact is treated as a virtual sensor.

Error Handling:

A sensorNotFoundException is thrown in case the sensor id is invalid.

Remarks:

None

Listing 89: Source Code Example – Get Contact Data

```

/**
 * Requests for the actual data of a particular sensor
 * Filtered by the input parameters
 *
 * @param filterValues HashMap<String,String> as Key Value pairs to filter
 * the data source
 * @throws SensorNotFoundException in case the sensor id is invalid
 *
 */
public Sensor[] getContacts(UUID sensorID, HashMap filterValues) throws
    SensorNotFoundException;

...

// Request sensor
try {
    Sensor[] s = getContacts (sensorID, filterValues);
} catch (SensorNotFoundException e) {
    ...
}

```

5.3.5.2.6 Get Devices in Area

The Java interface Get Devices in Area (see Listing 90) provides the devices in a particular area.

Parameters:

- lat: Latitude of the centre of the requested area
- long: Longitude of the centre of the requested area
- radius: Radius in meters around this centre point to determine the size of the requested area

Return Values:

The device data of the PMA as eu.simpli-city.Device object. If no device can be found, the return value is null.

Error Handling:

None

Remarks:

The input parameters are the latitude and longitude of the centre of the requested area and a radius in meters around this centre point to determine the size of the requested area.

Listing 90: Source Code Example – Get Devices in Area

```

/**
 * Requests for the devices in a particular area
 *
 * @param lat of the centre of the requested area
 * @param long of the centre of the requested area
 * @param radius in meters around this centre point to determine the size
 * of the requested area
 *
 */
public Device[] getDevicesInArea(Double lat, Double long, int radius);
...

// Request PMA device
Device[] d = getDevicesInArea(lat, Double, long, radius);

```

5.3.6 Content Format

The JSON content format used is described in this subsection. For a more detailed explanation see deliverable D4.1.1.

The structure of the JSON data format used for sensor data is depicted in Figure 11 that gives a hierarchical overview of the data structure. In the hierarchical view, on the top level there will be an object class that brings the basic properties for devices and sensors. Each object brings the following properties:

- Object: The respective class descriptor (sensor, device)
- Timestamp: The point in time of object creation
- ObjectID: A unique 128 bit identifier
- Parent: The object ID of the parent object if available

However, the type object will never be directly initiated. Instead, objects of the derived classes sensor or device will be used. Thus there is also no direct JSON description of the class object necessary. A device can be connected from 0 to n sensors and 0 to m other devices. That means there could be a cascade of devices, each in between device can have sensors but do not have to. Sensors are always connected to devices and have 1 to n values. That means a sensor has at minimum one corresponding value. All sensors have a field describing their type. Sensor values are provided as array; so, complex sensor values can be represented as an array of single values. Each sensor value can have, but does not have to have, an accuracy property that has a value describing the accuracy range and a corresponding unit. If a data object is not available the value *NULL* is provided.

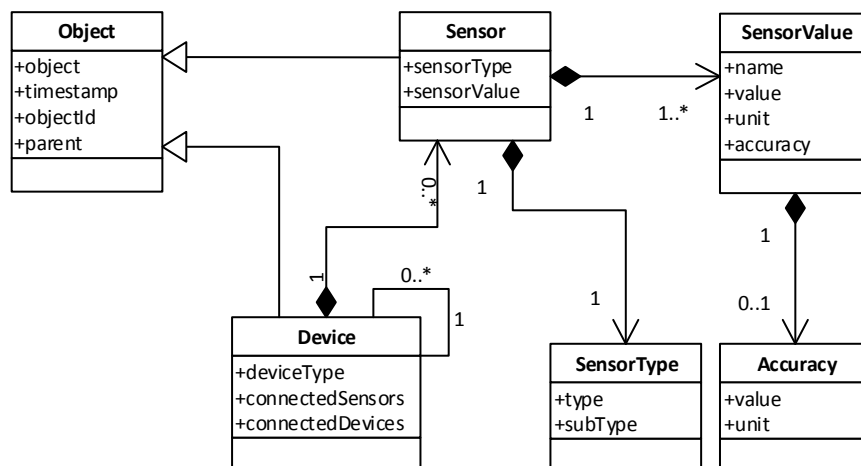


Figure 11: JSON Object Structure for Sensor-related Data

Following the previously introduced data structure, Listing 91 gives an example of the JSON data representation of a device. In this example the device is the PMA and has two different location sensors and one connected device.

Listing 91: JSON Example – JSON data of a device

```

{
  "object": "device",
  "deviceType": "PMA",
  "connectedSensors": [
    {
      "sensorType": {
        "type": "location",
        "subType": "coordinates"
      },
      "sensorID": "ID-SENSOR-CCC"
    },
    {
      "sensorType": {
        "type": "location",
        "subType": "address"
      },
      "sensorID": "ID-SENSOR-DDD"
    }
  ],
  "connectedDevices": [
    {
      "deviceType": "car",
      "deviceID": "528739A0-F508-4551-A12A-04A9B51718AA"
    }
  ],
  "timeStamp": "2013-08-28T10:27:10.000Z",
  "objectID": "528739A0-F508-4551-A12A-04A9B51718D0",
  "parent": null
}

```

Following the previously introduced data structure, Listing 92 gives an example of the JSON data representation of a sensor. In this example the sensor is an engine speed sensor of a car and has one single sensor value.

Listing 92: JSON Example – JSON data of a single sensor

```
{
  "object": "sensor",
  "sensorType": {
    "type": "vehicle",
    "subtype": "engineSpeed"
  },
  "sensorValue": [
    {
      "name": "engineSpeedRPM",
      "value": 3254,
      "unit": "rpm",
      "accuracy": {
        "unit": "percentage",
        "value": 0.05
      }
    }
  ],
  "objectID": "528739A0-F508-4551-A12A-04A9B51718D0",
  "timeStamp": "2013-08-28T10:27:10.000Z",
  "parent": "ID-BBB"
}
```

5.3.6.1 Device JSON Schema Description

According to the previously given explanation of the JSON data format that was example-driven, in the following, a formal JSON Schema description is specified. A device can be connected with 0 to n sensors and 0 to m other devices. That means there could be a cascade of devices, each in between device can have sensors but do not have to. A device is mainly described by its property *deviceType*. Furthermore it brings the arrays *connectedSensors* and *connectedDevices* that are both optional. Obviously the existence of minimal one of them would be necessary to make the device object become meaningful. Listing 93 gives the JSON schema description of a *Device* object.

Listing 93: Device JSON Schema

```

{
  "type": "object",
  "id": "http://Simpli-City.eu/SensorAbstraction/JSON-Schema/Device",
  "properties": {
    "deviceType": {
      "type": "string",
      "required": true
    },
    "connectedSensors": {
      "type": "array",
      "required": false,
      "items": {
        "type": "object",
        "id": "http://simpli-city.eu/SensorAbstraction/JSON-
Schema/ConnectedSensors",
        "required": true,
        "properties": {
          "sensorType": {
            "type": "object",
            "id": "http://Simpli-City.eu/SensorAbstraction/JSON-
Schema/SensorType",
            "required": true,
            "properties": {
              "type": {
                "type": "string",
                "required": true
              },
              "subType": {
                "type": "string",
                "required": true
              }
            }
          },
          "sensorID": {
            "type": "string",
            "required": true
          }
        }
      }
    },
    "connectedDevices": {
      "type": "array",
      "required": false,
      "items": {
        "type": "object",
        "id": "http://simpli-city.eu/SensorAbstraction/JSON-
Schema/ConnectedDevices",
        "required": true,
        "properties": {
          "deviceType": {
            "type": "string",
            "required": true
          },
          "deviceID": {
            "type": "string",
            "required": true
          }
        }
      }
    }
  }
}

```

For better readability of the *Device* JSON schema description (see Listing 93) the JSON schema of the subobject *ConnectedSensors* is given in Listing 94.

Listing 94: ConnectedSensors JSON Schema

```
{
  "type": "object",
  "id": "http://simpli-city.eu/SensorAbstraction/JSON-
Schema/ConnectedSensors",
  "required": true,
  "properties": {
    "sensorType": {
      "type": "object",
      "id": "http://Simpli-City.eu/SensorAbstraction/JSON-
Schema/SensorType",
      "required": true,
      "properties": {
        "type": {
          "type": "string",
          "required": true
        },
        "subType": {
          "type": "string",
          "required": true
        }
      }
    },
    "sensorID": {
      "type": "string",
      "required": true
    }
  }
}
```

For readability of the *ConnectedSensors* JSON schema description (see Listing 94) the JSON schema of the subobject *SensorType* is given in Listing 95.

Listing 95: SensorType JSON Schema

```
{
  "type": "object",
  "id": "http://simpli-city.eu/SensorAbstraction/JSON-
Schema/ConnectedSensors",
  "required": true,
  "properties": {
    "sensorType": {
      "type": "object",
      "id": "http://Simpli-City.eu/SensorAbstraction/JSON-
Schema/SensorType",
      "required": true,
      "properties": {
        "type": {
          "type": "string",
          "required": true
        },
        "subType": {
          "type": "string",
          "required": true
        }
      }
    },
    "sensorID": {
      "type": "string",
      "required": true
    }
  }
}
```

For better readability of the *Device* JSON schema description (see Listing 93) the JSON schema of the subobject *ConnectedDevices* is given in Listing 96.

Listing 96: ConnectedDevices JSON Schema

```
{
  "type": "object",
  "id": "http://simpli-city.eu/SensorAbstraction/JSON-
Schema/ConnectedSensors",
  "required": true,
  "properties": {
    "sensorType": {
      "type": "object",
      "id": "http://Simpli-City.eu/SensorAbstraction/JSON-
Schema/SensorType",
      "required": true,
      "properties": {
        "type": {
          "type": "string",
          "required": true
        },
        "subType": {
          "type": "string",
          "required": true
        }
      }
    },
    "sensorID": {
      "type": "string",
      "required": true
    }
  }
}
```

5.3.6.2 Sensor JSON Schema Description

Corresponding to the previously given explanation of the device JSON schema, the respective Sensor JSON Schema description is specified in the following listings. Sensors are always connected to devices and have 1 to n value entries. That means a sensor has at minimum one corresponding value. All sensors have an object describing their type, composed of the properties type and subtype. Sensor values are provided as array; so, complex sensor values can be represented as an array of single values. Each sensor value can have, but does not have to, an object for the accuracy property that contains a value describing the accuracy range and a corresponding unit. Listing 97 gives JSON schema description of a *Sensor* object.

Listing 97: Sensor JSON Schema

```

{
  "type": "object",
  "id": "http://Simpli-City.eu/SensorAbstraction/JSON-Schema/Sensor",
  "properties": {
    "sensorType": {
      "type": "object",
      "id": "http://Simpli-City.eu/SensorAbstraction/JSON-
Schema/SensorType",
      "required": true,
      "properties": {
        "type": {
          "type": "string",
          "required": true
        },
        "subType": {
          "type": "string",
          "required": true
        }
      }
    },
    "sensorValue": {
      "type": "object",
      "id": "http://Simpli-City.eu/SensorAbstraction/JSON-
Schema/SensorValue",
      "required": true,
      "properties": {
        "name": {
          "type": "string",
          "required": true
        },
        "value": {
          "type": "value",
          "required": true
        },
        "unit": {
          "type": "string",
          "required": false
        },
        "accuracy": {
          "type": "object",
          "id": "http://Simpli-City.eu/SensorAbstraction/JSON-
Schema/Accuracy",
          "required": false,
          "properties": {
            "unit": {
              "type": "string",
              "required": true
            },
            "value": {
              "type": "number",
              "required": true
            }
          }
        }
      }
    }
  }
}

```


For better readability of the *Sensor* JSON schema description (see Listing 97), the JSON schema of the sub object *SensorValue* is given in Listing 98. A detailed view for the subobject *SensorType* was already given in Listing 95.

Listing 98: SensorValue JSON Schema

```
{
  "type": "object",
  "id": "http://Simpli-City.eu/SensorAbstraction/JSON-Schema/SensorValue",
  "required": true,
  "properties": {
    "name": {
      "type": "string",
      "required": true
    },
    "value": {
      "type": "value",
      "required": true
    },
    "unit": {
      "type": "string",
      "required": false
    },
    "accuracy": {
      "type": "object",
      "id": "http://Simpli-City.eu/SensorAbstraction/JSON-
Schema/Accuracy",
      "required": false,
      "properties": {
        "unit": {
          "type": "string",
          "required": true
        },
        "value": {
          "type": "number",
          "required": true
        }
      }
    }
  }
}
```

Finally, Listing 99 gives a detailed view for the subobject Accuracy for better readability of the *Sensor* JSON schema description (see Listing 97) and *SensorValue* JSON schema description (see Listing 98).

Listing 99: Accuracy JSON Schema

```
{
  "type": "object",
  "id": "http://Simpli-City.eu/SensorAbstraction/JSON-Schema/Accuracy",
  "required": false,
  "properties": {
    "unit": {
      "type": "string",
      "required": true
    },
    "value": {
      "type": "number",
      "required": true
    }
  }
}
```

5.3.6.3 ContactFilter JSON Schema Description

A filter can be described as composition of Key Value pairs, to enable the request for particular contact information of the user. The object is an input parameter of Get Contact data (see Section 5.3.5.2.5). Listing 100 gives JSON schema description of a ContactFilter object.

Listing 100: ContactFilter JSON Schema

```
{
  "type": "object",
  "id": "http://Simpli-City.eu/SensorAbstraction/JSON-Schema/ContactFilter",
  "properties": {
    "contactFilter": {
      "type": "array",
      "required": false,
      "items": {
        "key": {
          "type": "string",
          "required": true
        },
        "value": {
          "type": "value",
          "required": true
        }
      }
    }
  }
}
```

5.3.7 Summary

The available technologies for sensor abstraction services have been investigated. Since no appropriate solution is available, the solution within this project will be an own implementation for the Sensor Abstraction and Interoperability Interfaces to realize an holistic concept for data integration of various sources and formats including user related

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 146 / 435
http://www.simpli-city.eu/ Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201				

data. In contrast to other available technologies, in this particular case, the Sensor Abstraction and Interoperability Interfaces will not only abstract sensor networks but also interact with sensor data of mobile devices and user-related data. To allow an easy access to the available sensor data sources a lightweight and simple JSON data structure has been introduced within this section.

5.4 Media Data Streams and Data Prefetching Logic

5.4.1 Major Design Decisions

In the mobile environment, and therefore in SIMPLI-CITY, content accessibility in the Internet is a big and unsolved problem. This problem is often caused by delayed responses due to low bandwidth or even missing responses due to no network connection at all. If a user of the Personal Mobility Assistant (PMA) has to deal with one of these problems, then the Quality of Experience (QoE) suffers. Either the user has to wait a long time for the requested data or in the worst case the requested data does not get to the mobile device at all. The chances to have a bad user experience like this increases if the mobile device is moving at a high velocity. After a look into the SIMPLI-CITY use cases, it is likely that this case is fairly common for users of the PMA.

This component tries to solve these problems using two approaches:

- Media data transcoding is applied with the Media Data Analysis and the Media Data Transcoding components to lower the amount of needed data to be transferred, in case the bandwidth is small. The Data Prefetching and Streaming Service component provides the possibility to stream this optimized data to a PMA to ensure uninterrupted media playback on the PMA.
- Prefetching mechanisms are applied to download the content onto the PMA before the user wants to consume it. This will improve the user experience of using the PMA.

The Data Prefetching Logic is Split into Client- and Server-side Parts:

To accomplish this goal it is necessary to split the Media Data Streams and Data Prefetching Logic into two parts. The first part is the client-side Data Prefetching API, which is part of the Data Prefetching Gateway, running on the PMA and the second part is the server-side Data Prefetching and Streaming Service. To understand prefetching correctly, it is necessary to see it as a combination of predicting what resources will be needed in the future and caching those resources beforehand. The Data Prefetching Logic is caching data for the users and the Data Prefetching and Streaming Service is trying to predict when which data should be cached.

On different network layers, there is a long tradition of improvements in caching, e.g., web browser caches or Domain Name System (DNS) caches in the operating system. The HTTP header and common practices of caching web content offer excellent and proven caching algorithms. The client side contains a component where all the outgoing and incoming HTTP traffic is passed through. This component applies the caching algorithms to the flowing data.

SIMPLI-CITY has the opportunity to collect user activities and build general user behaviour profiles as well as personalised user behaviour profiles. The Data Prefetching and Streaming Service will be informed of the upcoming activities of a PMA from the Data

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 147 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

Prefetching Gateway. In cooperation with the Context-based Service Personalisation, the server components will try to predict future content requests of the PMA based on the user behaviour profiles and historical service call data.

Focus on Location-based Services Based on Historical Data:

The result of investigating the differences of the mobile environment and other environments connected to the Internet shows that location-based services have a fundamentally higher priority than other applications the user may want to make use of in a mobile environment. The SIMPLI-CITY PMA will interact with a wide range of location-based services very frequently. As users in the same area tend to request the same type of location related data or services, it is possible to predict some of the future requests of the PMA if the upcoming locations can be predicted or derived. Hence, it makes sense to set the main focus of predictions on location-based services. The historical data is collected by the Data Prefetching Gateway in collaboration with the Data Prefetching and Streaming Service.

Consider Media Streams apart from Normal Data:

Media streams are another very important resource for the SIMPLI-CITY users. Generally, most media data providers expect that users have a good Internet connection with a reasonable amount of bandwidth when they provide media files or streams. Therefore, the most media data providers do not provide data for low bandwidth environments without special adjustments.

In addition, Auphonic³³ showed that most media data providers do not have the required technical know-how or the motivation to encode and deliver their content to serve all types of connections. Most audio and video content can be delivered in a better quality and with a lower file size by using more specific encoding mechanisms.

For SIMPLI-CITY, this means that there are many possibilities to improve media content delivery, especially for the envisioned use cases that involve media consumption. Hence, the Data Prefetching Logic will manage media streams differently than other content types.

³³ https://auphonic.com/about_us

5.4.2 Technology Comparison

5.4.2.1 Comparison Criteria

Table 33: Specific Criteria for Technical Specification

Parameter	Importance	Description
Zero Configuration	++	The technology will need to be as configuration-inexpensive as possible to keep the focus on the real functionality. However, the amount of prefetched data heavily depends on the available internal storage, which may differ from device to device and from user to user. Therefore, it should be possible for a user to specify the maximum amount prefetched data in MB and have smartly chosen defaults. Also time limits for media consumption can apply, when the maximum time can be extrapolated that the user can consume media.
Multi-Format Support	++	Prefetched data is dynamic and has to be handled without any information about the type of data. Hence the prefetching component must be able to handle different kinds of data and data schemes.
Data Expiration Support	++	Data should have an inner lifecycle inside the caching component, which can either be manually adjusted or is automatically determined by the age and popularity of data.
Streaming Capabilities	+/-	The chosen technology should be able to either directly stream data or at least be able to send chunks of data instead of complete data blocks.
Small Data and Memory Footprint	+	Data in memory should be as pure as possible, as there is no need for additional metadata. The memory should be used for the pure necessary data.

5.4.2.2 Possible Technologies and Comparison

As it has already been described above, the Media Data Streams and Prefetching Logic is made up from a client-side and a server-side part. For both parts, technologies have to be chosen: The server-side cache (see Section 5.4.2.3) is a subcomponent of the Data Prefetching and Streaming Service that is used to store data as long as it is relevant. Selection criteria (see above) and ratings can be found in Table 34. Second, the client-side Local Prefetching Store will store the prefetched HTTP responses, files and the necessary metadata. To do this, either the file system in combination with a database for the metadata or a full blown caching system can be used. For both possibilities, a number of technologies are considered in Section 5.4.2.4. Selection criteria (see above) and ratings can be found in Table 35.

5.4.2.3 Server-Side Cache

Memcached:

Memcached³⁴ is an in-memory key-value store for small chunks of arbitrary data (e.g., strings or objects) from results of database calls, API calls, or page rendering. The simple design of Memcached promotes quick deployment, ease of development, and solves many problems facing large data caches. Its API is available for most popular programming languages. It is configured on three layers: client, server and cluster. Memcached needs to be run on a separate server, which requires more hardware than other caching systems.

Groupcache:

Groupcache³⁵ is a caching and cache-filling library, intended as a replacement for Memcached (see above) in different contexts. Groupcache is in production use by its creator Google for the main Google download service dl.google.com³⁶, and is used in the blogging platform Blogger, the source code network Google Code, the content delivery in Google Fiber, the Google production monitoring system and other Google applications. It is currently only available for Google's programming language Go, which is not a problem in this selection as the Data Prefetching and Streaming Service will also be implemented in Go. Groupcache is an open source library and therefore open for extensions to support expiration of data or streaming capabilities.

MemoryCache:

MemoryCache³⁷ is a native framework element of the .NET Framework and therefore available for common .NET programming languages (C#, VB and F#). MemoryCache was introduced for C# and VB.NET in the .NET Framework Version 4.0. MemoryCache supports out-of-the-box cache-expiration and is able to store different kind of objects. The major downside of MemoryCache is the need for a .NET Runtime Environment, which is very heavyweight in an environment that builds on performance. It also needs the Microsoft Windows operating system and a proprietary Microsoft server, which produces additional costs, since there is no other component in this project which uses Microsoft software.

EhCache:

EhCache³⁸ is an open source, standards-based cache for boosting performance, offloading databases, and simplifying scalability. It is the most widely used Java-based cache since it is robust, proven, and full-featured. EhCache scales from in-process, with one or more nodes, to mixed in-process/out-of-process configurations with terabyte-sized caches. It has a multi-tier approach to data expiration scaling from fast-accessible in-heap data to data that is stored. EhCache is very configuration-intensive and therefore not suitable for the Media Data Streams and Data Prefetching Logic component.

³⁴ <http://memcached.org/>

³⁵ <https://github.com/golang/groupcache>

³⁶ Slides by Brad Fitzpatrick: <http://talks.golang.org/2013/oscon-dl.slide#1>

³⁷ [http://msdn.microsoft.com/de-de/library/system.runtime.caching.memorycache\(v=vs.110\).aspx](http://msdn.microsoft.com/de-de/library/system.runtime.caching.memorycache(v=vs.110).aspx)

³⁸ <http://ehcache.org/>

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 150 / 435
http://www.simpli-city.eu/ Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201				

NCache:

NCache³⁹ is an extremely fast and scalable in-memory distributed cache that caches application data and reduces time-expensive database calls. NCache also stores ASP.NET session state in web farms. NCache is available for the .NET Framework as well as for the Java Virtual Machine. It has a free and two paid versions, with the free version being very limited in its functionality.

Table 34: Comparison of Technologies for Server-Side Cache

Parameter	Importance	Groupcache	MemoryCache	Memcached	EhCache	NCache
Generic Criteria						
Up-to-Datedness	++	10	5	8	10	6
Stability	+	10	7	8	10	8
Extensibility & Open Source/Standards	+	10	2	10	9	3
Familiarity	-	9	7	7	4	8
Performance	++	10	4	10	8	7
Interoperability	++	10	3	10	6	5
License		Apache 2.0	BCL, Proprietary	BSD	Apache 2.0	Proprietary
Specific Criteria						
Zero Configuration	++	10	7	6	7	3
Multi-Format Support	++	10	8	10	8	5
Data Expiration Support	++	9	5	8	5	4
Streaming Capabilities	+/-	N/A	N/A	N/A	N/A	N/A
Small Data and Memory Footprint	+	10	5	8	4	4

5.4.2.4 Client-side Database and Cache**LevelDB:**

LevelDB⁴⁰ is a fast key-value storage library written at Google in C++ that provides an ordered mapping from string keys to string values. The store is file-based and the data is automatically compressed by using the Snappy compression algorithm, also created at Google. LevelDB is designed to work very well with 1 billion entries, which makes it the largest tested database⁴¹. Implementing our own wrapper around a server would include a more extensive development and research activity.

SQLite:

SQLite⁴² is a software library that implements a self-contained, lightweight, server-less, zero-configuration, high-performance and transactional SQL database engine. SQLite is the most widely deployed SQL database engine in the world. It is usable on all operating systems, including Android. The source code for SQLite is in the public domain.

³⁹ <http://www.alachisoft.com/ncache/>

⁴⁰ <https://code.google.com/p/leveldb/>

⁴¹ <http://highscalability.com/blog/2011/8/10/leveldb-fast-and-lightweight-keyvalue-database-from-the-auth.html>

⁴² <http://www.sqlite.org/>

CouchDB:

CouchDB⁴³ is a NoSQL open source database licensed under the Apache 2.0 license. Unlike in a relational database, CouchDB does not store data and relationships in tables. Instead, each database is a collection of independent documents. Each document maintains its own data and self-contained schema. CouchDB follows a client/server approach as well, making it too resource intensive for the PMA itself, where the resources are limited and should be used for more important tasks.

Smart Cache:

Smart Cache⁴⁴ is a Java HTTP proxy server with offline browsing support. It has URL blocking and URL rewriting. It also supports large caches. Smart Cache does not provide any storing functionality and is a Java implementation which takes a large amount of resources of the PMA.

LittleProxy:

LittleProxy⁴⁵ is a HTTP proxy written in Java. It is an open source project and under development since 2009. It still has open issues and does not support HTTP/1.1 fully yet.

Table 35: Comparison of Technologies for Client-side Database and Cache

Parameter	Importance	LevelDB	SQLite	CouchDB	Smart Cache	LittleProxy
Generic Criteria						
Up-to-Datedness	→	8	9	10	7	6
Stability	+	10	10	10	5	4
Extensibility & Open Source/Standards	+	5	5	5	6	6
Familiarity	-	8	10	7	4	5
Performance	++	10	10	8	5	5
Interoperability	++	10	10	10	7	7
License		BSD	Public Domain	Apache 2.0	GPL v2	Apache 2.0
Specific Criteria						
Zero Configuration	++	10	10	6	7	8
Multi-Format Support	++	5	5	5	5	5
Data Expiration Support	++	3	3	3	5	5
Streaming Capabilities	++	N/A	N/A	N/A	6	4
Small Data and Memory Footprint	++	10	10	7	1	1

5.4.3 Technology Selection

Groupcache was selected for the caching subcomponent in the Data Prefetching and Streaming Service due to the extensibility and speed of the component combined with a favourable programming language to ease development. The main selection criteria were up-to-datedness, performance and multi-format support. All of these criteria are perfectly supplied by Groupcache. Groupcache is used for the Google download server, which provides downloads for all Google programs and therefore needs to be highly reliable and stable in highly occurring parallel download requests. The Google download server itself is also developed using Go, a highly concurrent and well performing language.

⁴³ <http://couchdb.apache.org/>

⁴⁴ <http://scache.sourceforge.net/>

⁴⁵ <http://littleproxy.org/>

To keep the server-side in a consistent programming language, Go is also selected for the Data Prefetching and Streaming Service, the Media Transcoding and the Media Data Analysis. In addition, the Go concurrency model perfectly meets the requirements of these components.

SQLite (in combination with the file system) was selected for the Local Prefetching Store due to its stability and performance. Furthermore, complex search algorithms are needed to match an HTTP request to a possibly already cached HTTP request plus the according HTTP response. Regarding the missing case-sensitivity of HTTP parameters and other variations in HTTP requests, simple key-value storages do not meet these requirements, since they are either not fast enough, or do not offer the needed flexibility in terms of queries, or need additional server-side functionalities which would defeat the purpose of the whole component in minimizing necessary Internet communication.

As specified in deliverable D2.3 Requirements Analysis, Android compatibility is a necessary requirement for all client-side functionalities in SIMPLI-CITY. Hence, the Media Data Streams and Data Prefetching Logic component will provide their API for the app developers on the PMA in Java, as the main programming language on Android is Java.

The Data Prefetching Gateway covers a special position in the architecture of the Media Data Streams and Data Prefetching Logic component as described in Section 5.4.4.1. All HTTP requests will be passed through this gateway. Hence, the Data Prefetching Gateway could be a performance bottleneck for the PMA. After specific investigations on Smart Cache and LittleProxy and overall Java and JVM performance (see Section 5.4.3.1), the “C++11” standard of C++ is selected as the programming language for the Data Prefetching Gateway, as it is best suited for performance-critical operations and to prevent possible performance bottlenecks. Android’s Native Development Kit includes the gcc 4.6, which covers the most important feature of the last C++ standard “C++11”. Today, C++ provides the best performance by far, but it requires the most extensive language-specific tuning. C++11 combined with the “Boost.Asio” library provides fast and memory lightweight performance with a consistent asynchronous model, which meets the requirements of a proxy on a mobile device.

5.4.3.1 Missing Elements and Implementation Needs

This subsection introduces missing elements of the selected products for the Media Data Streams and Data Prefetching Logic component. It also introduces implementation approaches for each subcomponent.

Data Prefetching Gateway:

As described in Section 5.4.4.1, all HTTP requests and HTTP responses of the PMA will pass through this subcomponent. In Section 5.4.2.3, the possibility of the client-side Data Prefetching Gateway being a bottleneck was already described. Investigations during the technology selection disclose that neither Smart Cache nor LittleProxy are specifically designed for mobile environments like SIMPLI-CITY's PMA with slow CPUs and small amounts of memory. The conclusion is that it is not possible to select existing software for the Data Prefetching Gateway. Therefore this subcomponent will be written from scratch in the SIMPLI-CITY project.

The inadequacy of Smart Cache and LittleProxy for the project is based on peculiarities of the environment they would have to run in (Java in combination with Android). The open

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 153 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

performance issues on a mobile device like the PMA are based on the managed code concept of Java. A parallelism test on Android reveals that the garbage collector has to stop all threads for a moment to do its work, which does not fit well in a consistent asynchronous model for the handling of incoming and outgoing TCP connections without loss in QoS (Quality of Service).

The following features have to be implemented during development:

- This subcomponent must be able to handle HTTP GET methods of the HTTP 1.1 standard.
- This subcomponent must be able to handle HTTP POST methods of the HTTP 1.1 standard.
- This subcomponent must be able to handle HTTP PUT methods of the HTTP 1.1 standard.
- This subcomponent must be able to handle HTTP DELETE methods of the HTTP 1.1 standard.
- This subcomponent must be able to handle HTTP CONNECT methods of the HTTP 1.1 standard to support HTTPS.
- This subcomponent must be able to handle HTTP OPTIONS methods of the HTTP 1.1 standard.
- This subcomponent must be able to handle HTTP HEAD methods of the HTTP 1.1 standard.
- This subcomponent must be able to handle HTTP TRACE methods of the HTTP 1.1 standard.
- This subcomponent has to provide an opt-out functionality for the Local Prefetching Store mechanics, triggered by a HTTP parameter in the HTTP request.
- This subcomponent implements the API of the Local Prefetching Store to return prefetched data directly to requester. This will be done by each incoming HTTP request

Local Prefetching Store:

The Local Prefetching Store is based on SQLite, as described in Section 5.4.4.1. The Local Prefetching Store mainly interacts with the Data Prefetching Gateway and therefore it also uses C++ to provide the functionalities.

The following features have to be implemented during development:

- Capabilities to store a HTTP request combined with an HTTP response
- Capabilities to access a matching HTTP response of an HTTP request
- Intelligent storage volume capabilities to remove no longer needed data if not enough storage is available on the device

Data Prefetching API:

The Data Prefetching API is a small and simple Java library for app developers, described in Section 5.4.4.1.3.

The following features have to be implemented during development:

- The functionality to call the RESTful Interface described in Section 5.4.5.4 with the Java interface described in Section 5.4.5.2.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 154 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

Media Playback API:

The Media Playback API provides an abstract class to prefetch media streams, described in Section 5.4.5.2.2, and targets app developers. Thus, it is written in Java. This subcomponent is a wrapper for the functionalities of the Media Data Analysis and Media Data Transcoding components of the Data Prefetching and Media Streams component and server-side Data Prefetching and Streaming Service.

The following features have to be implemented during development:

- The asynchronous download interface, described in Section 5.4.5.2.2
- The asynchronous download mechanism to call the server-side services, especially the Media Data Analysis and the Media Data Transcoding subcomponents
- Capabilities to make announcements about the current bandwidth of the network connection

Media Data Analysis:

The Media Data Analysis is an internal subcomponent of the server-side Media Data Streams and Data Prefetching Logic component. It provides only one external function, described in Section 5.4.5.3.1, which analyses a multimedia resource. The Go Media Framework⁴⁶ is a wrapper for FFmpeg⁴⁷, which is a cross-platform solution to record, convert, and stream audio and video. FFmpeg provides functionalities to read detailed codec information's of a media file.

The following features have to be implemented during development:

- The interface, as described in Section 5.4.5.3.1
- Capabilities to read the beginning of an external media resource
- Capabilities to read the beginning of a media resource included in the SIMPLI-CITY infrastructure
- Capabilities to read the beginning of a media resource from the Data Access Relay
- Capabilities to identify the codec of the media resource
- Capabilities to discover which type of content is encoded in the media resource
- Capabilities to interpret the collected metainformation about the media resource and estimate if it makes sense to transcode the media resource in the current situation of the PMA. This estimation focuses on continuous playback of the media resource with the passed bandwidth.

Media Data Transcoding:

The Media Data Transcoding is an internal subcomponent of the Media Data Streams and Data Prefetching Logic component on the server-side components. It is intended to stream multimedia resources and transcode these if it is useful and recommended by the Media Data Analysis subcomponent. Overall, this will be done to improve the user experience of consuming multimedia in a mobile environment.

The following features have to be implemented during development:

- The interface, as described in Section 5.4.5.3.2
- Capabilities to read a media resource over the Data Access Relay in chunks

⁴⁶ <http://code.google.com/p/gmf/>

⁴⁷ <http://ffmpeg.org/>

- Interactions to give the Data Access Relay possibilities to assimilate the transcoded content
- Capabilities to transcode a media resource chunk by chunk
- Capabilities to stream the transcoded chunks immediately

Data Access Relay:

The Data Access Relay is a layer between the media streams parts of the Media Data Streams and Data Prefetching Logic component on the Server Side and the media resources stored in the Cloud-based Information Infrastructure or on an external server. Groupcache, described in Section 5.4.2.2, meets the requirements of this functionality perfectly. Another feature of this subcomponent is to have the transcoded multimedia resource as stored in the Groupcache of the Media Data Streams and Data Prefetching Logic component to grant fast access to the requested resources. This will reduce the load on the Media Data Transcoding subcomponent.

Data Prefetching and Streaming Service:

The Data Prefetching and Streaming Service does not use existing products to reach its goals, because it is running on the PMA and will make use of the interfaces provided by the server side of the Media Data Streams and Data Prefetching component, which is making use of a hybrid solution consisting of openly available solutions as well as customized components written from scratch. This subcomponent includes a RESTful interface, described in Section 5.4.5.4, and a data engine that stores historical data. The RESTful interface is a wrapper for the other subcomponents of the Server Side.

The historical data engine has to analyse the incoming historical data, integrate it in a location-driven database to collect user behaviour data in the Cloud-based Information Infrastructure. To decide which data can be prefetched, it is necessary to dismantle the incoming historical data, described in Section 5.4.5.4.1, and decide which parts of the request are user-related and how much it affects the response. That does not mean that user-related data cannot be prefetched, but perhaps some data can be prefetched only briefly before the user performs the request. To discover the possibilities of the historical data engine, research efforts are necessary during development.

The following features have to be implemented during development:

- The RESTful Interface, as described in Section 5.4.5.4.1
- Capabilities to manage user-related lists of items to prefetch and store in the Cloud-based Information Infrastructure
- Capabilities to build a location-driven database to collect service and app usage behaviour
- The engine to analyse historical data
- A system to send push-notifications to the PMA to inform the device about data that has to be cached

5.4.3.2 Further Information and Conclusion from Technology Comparison

During the technology comparison, the idea of moving some parts of the Media Data Streams and Data Prefetching Logic component into the HTTP layer was brought up. This implies the possibility to influence the whole mobile network computing state-of-the-art and not only the SIMPLI-CITY use cases using SIMPLI-CITY's infrastructure. Hence, it was determined that a local proxy-like component on the device is needed for maximum impact. Therefore the Data Prefetching Gateway was introduced and described in Section 5.4.4.1 in this document.

After the investigation of existing software for the Data Prefetching Gateway, it turned out that there is no technology that meets the performance requirements of this subcomponent. Creation of this subcomponent with a fast access to the Local Prefetching Store, described in Section 5.4.4.1 is a difficult task, but the invested time has been deemed reasonable for the possible impact.

How efficient the Media Data Streams and Data Prefetching Logic component works depends mostly from the research outcomes of the historical data engine (see Section 5.4.3.1) and provided predictions of the Context-based Service Personalisation (T5.2), delivered via the RESTful interface, defined in Section 6.3.

5.4.4 Component Structure

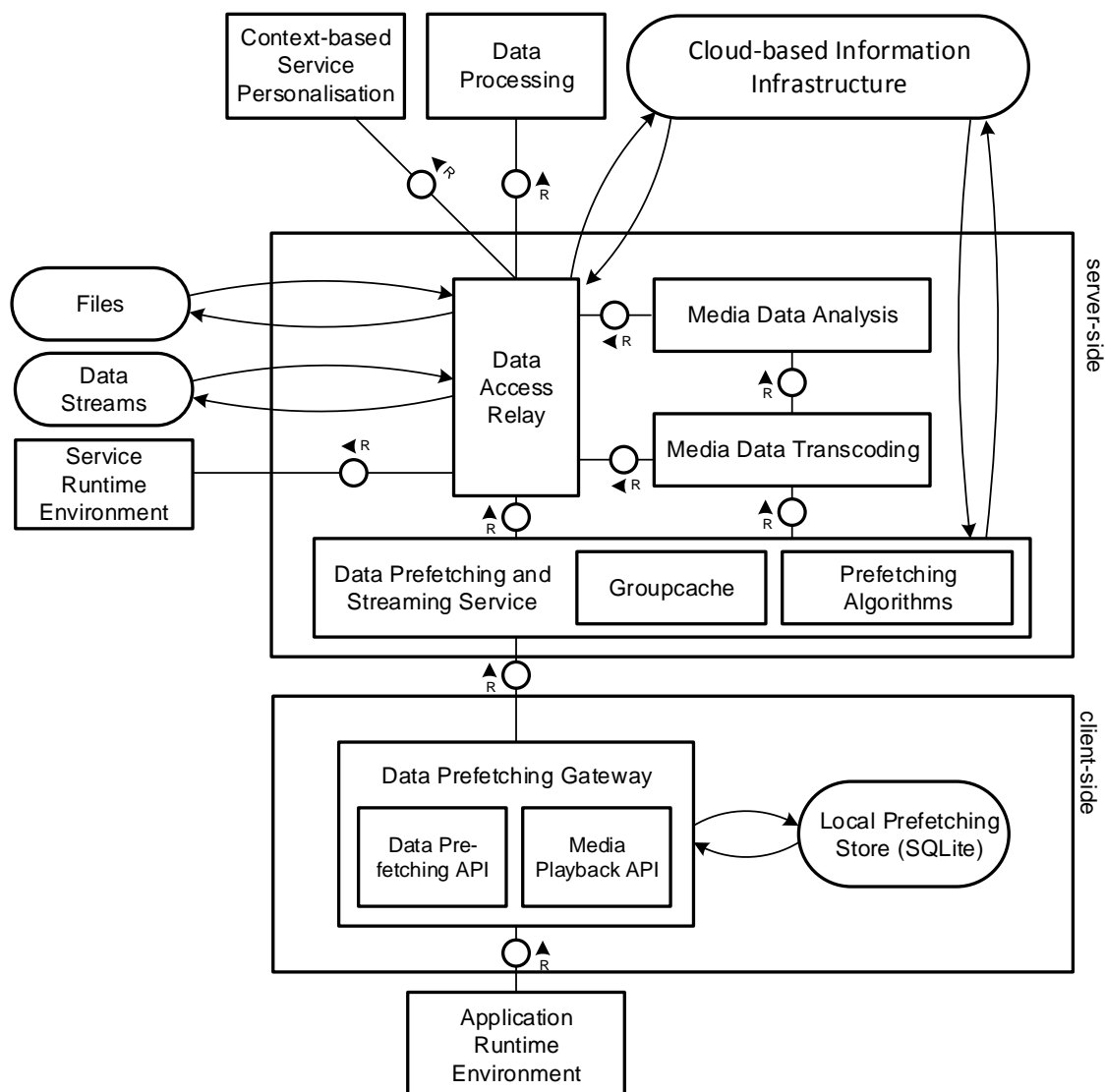


Figure 12: Block Diagram – Media Data Streams and Data Prefetching Logic

This subsection gives an overview about the component structure. As introduced in Section 5.4.4.1, the Media Data Streams and Data Prefetching Logic component is divided into two parts: The client-side and the server-side subcomponents.

Figure 12 shows the updated component structure of the SIMPLI-CITY Media Data Streams and Data Prefetching Logic. Compared to the Functional Specification (D3.2.1), the former Data Prefetching & Media Playback API has now been replaced by the Data Prefetching Gateway, so the content of the Local Prefetching Store is no longer separated into media and other data and the Smart Media Selection has been removed. This change was made due to the move of the component into the HTTP layer, as explained in Section 5.4.3.2.

The Data Prefetching and Streaming Service also gets separated from the used Prefetching Algorithms, as the algorithms are supposed to be exchangeable and as the necessary work is better separated when the technical implementation is not intermixed

with the complexities of the algorithm that gets used, as this is matter of continuous improvement.

In the next subsections, each subcomponent of the Media Data Streams and Data Prefetching Logic component gets a detailed overview.

5.4.4.1 Client-Side Components

The client-side components consist of four major subcomponents: The Local Prefetching Store, the Data Prefetching API, the Media Playback API and the Data Prefetching Gateway.

5.4.4.1.1 Local Prefetching Store

The Local Prefetching Store is the database to store every kind of HTTP request for a limited period of time and keep these requests accessible via a key. This store is scanned on almost every HTTP request of the user. Therefore the performance of the Local Prefetching Store is very important. Also, on mobile devices like the PMA, data storage is limited. To prevent overflow of the data storage, the Local Prefetching Store has to do cache invalidation, i.e., decide which cached data is still relevant and valid. It is important to state that this component does not decide which data is prefetched. The Local Prefetching Store gets this information from the server-side Data Prefetching and Streaming Service.

This component is based on a SQLite database. Therefore the data storage of this subcomponent is on the PMA's file system. In order to fulfil the performance requirements, the multi-process access policy of SQLite is disabled⁴⁸. This stability feature is not necessary for this subcomponent and would come along with performance issues. The Local Prefetching Store specific functionalities are deployed in a native library and written in modern C++. To guarantee a consistent state in the SQLite database file with a disabled synchronous feature, only one instance of that library will access the SQLite database file. Synchronization issues will be dealt with in the C++ library with lock-free programming techniques [ALE04]⁴⁹.

5.4.4.1.2 Data Prefetching Gateway

All HTTP requests of the device will pass through the Data Prefetching Gateway, which works technically similar to a proxy on the PMA. If this component is passed an HTTP request, it communicates with the Local Prefetching Store to figure out if this HTTP request was already expected. In this case, the response to that request is stored in the Local Prefetching Store. The Data Prefetching Gateway then reads the response directly from the Local Prefetching Store, so a network connection is not necessary, saving bandwidth and making locally cached services possible even with no cellular connection and even if the service was never directly accessed by the user before. When the response was not cached, the Data Prefetching Gateway will perform the HTTP request on the network and deliver the response using the PMA's wireless connection.

⁴⁸ defined with "PRAGMA synchronous = 0" in the environment configuration, and the rollback journal is stored in the memory, defined with "PRAGMA journal_mode = MEMORY".

As already noted, the servers-side components make the predictions of what data and service calls are prefetched. To validate and improve the outcome of this part of the Media Data Streams and Data Prefetching component, the Data Prefetching Gateway collects metadata about the performed HTTP requests and the prefetching success. The Data Prefetching Gateway uses background functionalities to deliver this data periodically to the server-side components.

The Data Prefetching Gateway will be implemented in modern C++, as decided in Section 5.4.3.1. To handle the incoming requests more efficiently, the consistent asynchronous I/O model of the Asio C++ library will be used.

5.4.4.1.3 Data Prefetching API

The Data Prefetching API provides a set of methods to inform the server-side components about upcoming events. The API targets the app developers and especially app developers of apps from the navigation domain. For the Prefetching Algorithms, it is important that navigation apps deliver information about the upcoming positions of the PMA to this API. The Data Prefetching API transmits the information to the server-side Data Prefetching and Streaming Service, so it can predict the upcoming HTTP requests.

The API targets app developers. Therefore, the API will be provided as a Java library on the PMA.

5.4.4.1.4 Media Playback API

The Media Playback API provides functionalities to stream media data in a mobile environment and targets app developers. The app developer can hand over a media data resource on the network as a target by an API call. The Media Playback API will manage the buffering and caching functionalities. Furthermore, this component will make use of the Media Data Analysis and the Media Data Transcoding to optimize the encoding of the media file. This will help to stream media data continuously in a mobile environment with small bandwidth. The way it works is described in Section 5.4.5.2.2.

The API is targeted towards app developers. Therefore, the methods will be provided as a Java library on the PMA.

5.4.4.2 Server-Side Components

The server-side parts of the component are composed of three subcomponents: The Media Data Analysis, the Media Data Transcoding and the Data Prefetching and Streaming Service. SIMPLI-CITY is supposed to scale to a very large number of users and apps. This means that the load of all subcomponents can be very high. To manage an unpredictable load on the subcomponents it is indispensable to have highly scalable and well-performing server-side subcomponents.

5.4.4.2.1 Media Data Analysis

The Media Data Analysis is part of the media streaming functionality. A media data stream invoked by the Media Data Transcoding can use this component to analyse the media data. The Media Data Analysis can identify music and speech from an audio data stream and handle them in different ways. The adaptive normalizer can detect the base quality of a media stream and create copies of the stream with different encoding options (e.g.,

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 160 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

lower bitrate to create smaller files). With this information, the component is able to make the best choice of encoding for the current bandwidth of the PMA. As decided in Section 5.4.3.1, this subcomponent will be implemented in the programming language Go and will use the Go Media Framework to analyse the multimedia resource.

5.4.4.2.2 Media Data Transcoding

As media streams have a very high data bandwidth needed to encode the media data, this component has to cope with a lot of data throughput. The Media Data Transcoding transforms the media stream to another encoding if the efforts of the Media Data Analysis imply that is helpful to guarantee a continuous playback on the device. To prevent performance issues, this subcomponent calls the Data Access Relay before it starts a transcoding and provides the transcoded multimedia data to the Data Access Relay subcomponent, which means that the Media Data Transcoding subcomponent is the source for the Groupcache in the Data Access Relay. Thus, if more than one user consumes the same media resource in similar network environments than the media resource is transcoded by the Media Data Transcoding only once. As decided in Section 5.4.3.1, this subcomponent is developed with the Go programming language and the Go Media Framework is used for the transcoding.

5.4.4.2.3 Data Access Relay

The Data Access Relay provides caching functionalities to the media-streaming infrastructure of the server-side Media Data Streams and Data Prefetching Logic component. In situations that contain consuming multimedia data, data throughput peaks on some multimedia resources are common. It is anticipated that several users use the same multimedia resources to reduce redundant work for the Data Access Relay. The Data Access Relay is applied to minimize the traffic between the Media Data Streams and Data Prefetching Logic component and external multimedia resources and reduce the load from the Media Data Transcoding subcomponent. Moreover, already transcoded media data is cached in the Data Access Relay to reduce the load on the Media Data Transcoding subcomponent.

5.4.4.2.4 Data Prefetching and Streaming Service

First of all, the server-side Data Prefetching and Streaming Service acts like an abstraction layer of the Media Data Transcoding and the Media Data Analysis so the Media Playback API uses it to invoke the Media Data Transcoding. But the main functionality is to receive and interpret all the metadata of the PMAs, or more specific the metadata of the Data Prefetching Gateways on the PMAs. This subcomponent will build a database in the Cloud-based Information Infrastructure of service and app usage behaviour as HTTP requests in combination with the current GPS positions (e.g., popular parking spots will be saved for later use). Aside from that, the Data Prefetching API on the client-side invokes the Data Prefetching and Streaming Service to hand over the prediction of events to occur and asks for a list of prefetchable resources at once. To provide this list, the Data Prefetching and Streaming Service invokes the Context-based Service Personalisation and its database in the Cloud-based Information Infrastructure to compute a list of possible targets of user request at the upcoming GPS positions of this user.

The Prefetching Algorithms subcomponent is the core structure for making prefetching suggestions. This could be a number of algorithms specially designed for different types of

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 161 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

traveling (e.g., by car, by foot, by train), can be very simple for developing the Data Prefetching and Streaming Service, or can be very advanced or even be provided by third parties. These specialized algorithms will be invoked by the Data Prefetching and Streaming Service depending on their metadata descriptions.

These algorithms represent a research topic, as the best ways of selecting data that suits the current context of the user cannot be predefined. Also, some algorithms may work better for users that travel by bike, since there are no parking spaces needed, while some might work better when a user travels by train, where there might be the need for a bus transmit to a certain Point of Interest (POI) in a city.

5.4.5 Interfaces

This section describes the interfaces of the Media Data Streams and Data Prefetching Logic component. Therefore this section is split into two subsections. The first subsection contains the client-side PMA while the second subsection contains the interfaces for the server-side part of the Application Runtime Environment (see Section 7.1).

Notably, the client-side components offer C++ (see Section 5.4.5.1) and Java interfaces (see Section 5.4.5.2.1), as all components offer external Java interfaces that will be used to maintain a common approach for other components of SIMPLI-CITY. In contrast, the server-side component interfaces offer both Java (see Section 5.4.5.2) and REST interfaces (see Section 5.4.5.4). Please note that the UML class diagrams for the C++, Go, and Java interfaces are provided in the respective subsections of Section 5.4.6.

5.4.5.1 Client-Side C++ Interfaces

5.4.5.1.1 Local Prefetching Store – Store

The Local Prefetching Store provides functionality on the PMA to store HTTP requests and the corresponding HTTP responses. The Data Prefetching Gateway mainly uses the Local Prefetching Store. Therefore the Local Prefetching Store API is written in C++.

The store function is used to store HTTP requests and responses.

Parameters:

- req: The HTTP request, described in Section 5.4.6.1. Since a HTTP request will inevitably contain parameters which are not needed for storing data in the Local Prefetching Store, these parameters will be omitted.
- resp: The HTTP response, described in Section 5.4.6.1.

Return Value:

None

Error Handling:

In case of an error, an `lps_store_exception` is thrown, described in Section 5.4.6.1.

Remarks:

This function stores the HTTP request and HTTP response jointly in the Local Prefetching Store.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 162 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

In Listing 101, the signature of the method for storing in the Local Prefetching Store is shown as well as an example for the usage of this method.

Listing 101: Source Code Example – Local Prefetching Store: Store

```
/**
 * @param req, the HTTP request. All not important or variable parameters should be
 * removed.
 * @param resp, the HTTP response.
 */
void store(const http_request& req, const http_response& resp);
//...

http_request req;
http_response resp;
//...
try
{
    store(req, resp);
}
catch(lps_store_exception& e)
{
    //...
}
```

5.4.5.1.2 Local Prefetching Store – Search

The search function is used to match an HTTP response matching an HTTP request.

Parameters:

- req: The HTTP request, described in Section 5.4.6.1. Since a HTTP request will inevitably contain parameters which are not needed for searching data in the Local Prefetching Store, these parameters will be omitted.

Return Value:

The HTTP response is returned as an http_response class, described in Section 5.4.6.1. If no response is found then the result of this function is an empty http_response.

Error Handling:

In case of an error, an lps_store_exception is thrown, described in Section 5.4.6.1.

Remarks:

This function loads an HTTP response from the Local Prefetching Store.

In Listing 102, the signature of the method for searching the Local Prefetching Store is shown as well as an example for the usage of this method.

Listing 102: Source Code Example – Local Prefetching Store: Search

```

/**
 * @param req, the HTTP request. All not important or variable parameters should be
 * removed.
 */
http_response search(const http_request& req);

//...

http_request req;
//...
try
{
    http_response resp = store(req);
}
catch(lps_search_exception& e)
{
    //...
}

```

5.4.5.1.3 Data Prefetching Gateway

The Data Prefetching Gateway is an HTTP and HTTPS proxy and runs locally on the PMA. All HTTP/HTTPS traffic is passed via this proxy. Therefore, the Data Prefetching Gateway accepts TCP connections from its own device and complies with the HTTP 1.1 standard.

Parameters:

- **Prefetching Algorithms:** This parameter enables/disables the Prefetching Algorithms. The default value is activated for HTTP. On HTTPS, the Prefetching Algorithms are always disabled. If the Prefetching Algorithms are activated, the Data Prefetching Gateway will look into the Local Prefetching Store for the response.

Return Value:

The result is the expected HTTP response.

Error Handling:

For this subcomponent no specific error exists. HTTP errors are directly forwarded to the requester.

Remarks:

The Data Prefetching Gateway expects incoming requests on a configurable TCP port and accepts only incoming TCP packets from the PMA itself.

Listing 103: Signature in the Data Prefetching Gateway – HTTP Request

```
[GET|POST|PUT|HEAD|DELETE|TRACE|CONNECT] [PATH] [HTTP/1.1|HTTP/1.0]
Host: [HOST]
Connection: [close|keep-alive]
...
SIMPLI-CITY_Prefetching_Logic: [off|on]
...
```

5.4.5.2 Client-Side Java Interfaces

5.4.5.2.1 Data Prefetching API – Upcoming Locations

The Data Prefetching API targets the app developers and is written in Java. The upcomingLocations method is used to transmit predictions about upcoming positions to the server side.

Parameters:

- **userId:** An ID to identify a SIMPLI-CITY user.
- **positions:** List of timestamps paired to positions. A timestamp is presented as a java.util.Date object.

Return Value:

The result is the expected HTTP response.

Error Handling:

There are no low level errors that could occur in this function. All possible errors will be handled by the managing component (the Application Runtime Environment) and do therefore not need to be caught at this stage of execution. If the method does not find any upcoming locations an empty resultset is returned.

Remarks:

App developers are expected to call this function when they can make predictions on upcoming positions, e.g., a navigation app starts to navigate.

In Listing 104, the signature of the method to get the upcoming location is shown as well as an example for the usage of this method.

Listing 104: Source Code Example – Method Signature for Upcoming Locations

```

/**
 * @param positions, list of time stamps paired to positions.
 * @param userId, an ID to identify a SIMPLI-CITY user.
 */
public void upcomingLocations(long userId, List<AbstractMap.SimpleEntry<Date,
android.location.Location>> positions) {

    //...

}
List<AbstractMap.SimpleEntry<Date, android.location.Location>> pos = new
ArrayList<AbstractMap.SimpleEntry<Date, android.location.Location>>();
pos.add(new AbstractMap.SimpleEntry<Date, android.location.Location>(date,
location));
//...
upcomingLocations(5134, pos);

```

5.4.5.2.2 Media Playback API – General Comments

The Media Playback API is for app developers and is a Java API. App developers have to extend the `MediaStream` class (see Listing 105). This is the one and only way to make use of the media streaming part of SIMPLI-CITY's prefetching logic. It guarantees that apps on the PMA handle downloads in chunks. Each method is described in detail in the following subsections.

Listing 105: Source Code Example – Method Signature for MediaStream Class

```

import java.net.URL;
/* abstract class to handle media data resources.*/
public abstract class MediaStream
{
/**
 * Starts the streaming process.
 * @param url, a java.net.Url representation a media resource.
 */
public void start(URL url) throws java.net.SocketException;
/**
 * Stops the streaming process.
 */
public void stop();
/**
 * Invoked on new data.
 * @param dataChunk, a chunk of data
 */
protected void onNewData(byte[] dataChunk);
/**
 * Invoked on finish.
 */
protected void onFinished();
/**
 * Invoked in the case of an error
 * @param e, the occurred error.
 * @return true, if a continue is desired. Otherwise, false.
 */
protected boolean onError(Exception e);
/**
 * Invoked on reestablished streaming connection.
 */
protected void onResume();
}

```

5.4.5.2.3 Media Playback API – Start

With the start method of the Media Playback API the data streaming can be started. After a call of this method the streaming and prefetching of the media resource begins.

Parameters:

The following parameters are expected:

- url: A java.net.Url representation of a media resource.

Return Value:

None

Error Handling:

In case of an exception, it is caught and forwarded to the onError method.

Remarks:

None

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 167 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

In Listing 106, the signature of the method start of the Media Playback API is shown as well as an example for the usage of this method.

Listing 106: Source Code Example – Media Playback API: Start

```
/**
 * Starts the streaming process.
 * @param url, a java.net.Url representation of a media resource.
 */
public void start(URL url) throws java.net.SocketException;

//...

class MyMediaStream extends MediaStream
{
    //...
}
//...
MyMediaStream stream = new MyMediaStream();
//...
stream.start(url);
```

5.4.5.2.4 Media Playback API – Stop

With the stop method of the Media Playback API, the streaming and prefetching process can be interrupted.

Parameters:

No parameters are expected.

Return Value:

None

Error Handling:

In case of an exception, it is caught and forwarded to the onError method.

Remarks:

None

In Listing 107, the signature of the method stop of the Media Playback API is shown as well as an example for the usage of this method.

Listing 107: Source Code Example – Media Playback API: Stop

```
/**
 * Stops the streaming process.
 */
public void stop();

//...

class MyMediaStream extends MediaStream
{
    //...
}
//...
MyMediaStream stream = new MyMediaStream();
//...
stream.stop();
```

5.4.5.2.5 Media Playback API – onNewData

The onNewData method of the Media Playback API is an event handler, which is called when new data in terms of a media stream arrives at the PMA.

Parameters:

- dataChunk: A chunk of the media stream

Return Value:

None

Error Handling:

In case of an exception, it is caught and forwarded to the onError method. If the method returns true (which means that the exception was handled), the function will try to call itself again.

Remarks:

None

In Listing 108, the signature of the method onNewData of the Media Playback API is shown as well as an example for the usage of this method.

Listing 108: Source Code Example – Media Playback API: OnNewData

```

/**
 * Invoked on new data.
 * @param dataChunk, a chunk of data
 */
protected void onNewData(byte[] dataChunk);

//...

class MyMediaStream extends MediaStream
{
    //...
    protected void onNewData(byte[] dataChunk)
    {
        try
        {
            // handle data
        }
        catch (Exception ex)
        {
            if (onError(ex)) // exception was successfully handled
            {
                onNewData(dataChunk); // try again
            }
        }
    }
}

```

5.4.5.2.6 Media Playback API – onFinish

The onFinish method of the Media Playback API is an event handler, which is called as soon as the data streaming is finished.

Parameters:

No parameters are expected.

Return Value:

None

Error Handling:

In case of an exception, it is caught and forwarded to the onError method.

Remarks:

None

In Listing 109, the signature of the method onFinish of the Media Playback API is shown as well as an example for the usage of this method.

Listing 109: Source Code Example – Media Playback API: OnFinished

```

/**
 * Invoked on finish.
 */
protected void onFinished();

//...

class MyMediaStream extends MediaStream
{
    //...
    protected void onFinished()
    {
        try
        {
            // it is done!
        }
        catch (Exception ex)
        {
            onError(ex);
        }
    }
}

```

5.4.5.2.7 Media Playback API – onError

The onError method of the Media Playback API is an event handler, which is called as soon as an error occurs.

Parameters:

No parameters are expected.

Return Value:

If this method returns true, the exception was found in a general approach (e.g., when the connection was closed unexpectedly, the component will try to re-establish the connection.). This has to be implemented by the app developer.

Error Handling:

This event is used by all functions of the Media Playback API. The onError method logs all error messages and tries to handle general error messages. In case of a general error, the onError method returns true to tell the calling function that something happened. If it returns false, the exception could not be generally handled and in most cases an error will be shown to the user through the Multimodal Dialogue Interface.

Remarks:

None

In Listing 110, the signature of the method onError of the Media Playback API is shown as well as an example for the usage of this method.

Listing 110: Source Code Example – Media Playback API: OnError

```

/**
 * Invoked in the case of an error
 * @param e, the occurred error.
 * @return true, if a continue is desired. Otherwise, false.
 */
protected boolean onError(Exception e);

//...

class MyMediaStream extends MediaStream
{
    //...
    protected boolean onError(Exception e)
    {
        // general exception: probably lost connection to the stream
        if (e instanceof SocketException)
        {
            // connection lost?
            if (!this.connected)
            {
                this.connect();
                return true;
            }
        }
        // general exception: file could not be found
        else if (e instanceof FileNotFoundException)
        {
            // the file was not found, send message to the user
            MultiModalDialogueInterface.showError("file could not be
found.");
            return false;
        }
        // general exception: output device could not be found or opened
        else if (e instanceof IOException)
        {
            // send message to the user that the device could not be
accessed.
            MultiModalDialogueInterface.showError("output device could not
be accessed.");
            return false;
        }
        return false;
    }
}

```

5.4.5.2.8 Media Playback API – onResume

The onResume method of the Media Playback API is an event handler, which is called as soon as a media streaming and prefetching has been resumed.

Parameters:

No parameters are expected.

Return Value:

None

Error Handling:

In case of an exception, it is caught and forwarded to the onError method. If the method returns true (which means that the exception was handled), the function will try to call itself again.

Remarks:

None

In Listing 111, the signature of the method onResume of the Media Playback API is shown as well as an example for the usage of this method.

Listing 111: Source Code Example – Media Playback API: OnResume

```
/**
 * Invoked on reestablished streaming connection.
 */
protected void onResume();

//...

class MyMediaStream extends MediaStream
{
    //...
    protected void onResume()
    {
        try
        {
            // handle data
        }
        catch (Exception ex)
        {
            if (onError(ex)) // exception was successfully handled
            {
                onResume(); // try again
            }
        }
    }
}
```

5.4.5.3 Server-Side Go Interfaces

The server-side Media Data Streams and Data Prefetching Logic is fully written in Go, as has been described in Section 5.4.2.3.

5.4.5.3.1 Media Data Analysis

The `AnalyseMediaResource` method analyses which encoding would be the best in the current situation of the PMA.

Parameters:

- `user`: representation of a SIMPLI-CITY user, described in Section 5.4.6.2.
- `bandwidth`: the current bandwidth of the PMA.
- `mediaResource`: A multimedia resource object (in most cases an audio file)

Return Value:

Returns parameters for the Media Data Transcoding, e.g., the bitrate, the length and the size of the file.

Error Handling:

In a case of an error, the method returns a `MediaAnalyseError` as second return value, described in Section 5.4.6.2.

Remarks:

In Listing 112, the signature of the method `AnalyseMediaResource` of the Media Data Analysis is shown as well as an example for the usage of this method.

Listing 112: Source Code Example – Media Data Analysis: AnalyseMediaResource

```

/*
Analyzes which encoding would be the best with this bandwidth
in:
    user - representation of a SIMPLI-CITY user
    bandwidth - the bandwidth in bytes
    mediaresource - multimedia resource
out:
    {0} - the transcoding parameters for the mediaresource with the bandwidth
    {1} - provides error information in a case of an error
*/
func AnalyseMediaResource(user User, bandwidth int32, mediaresource *url.URL)
(*TranscodingParameters, *MediaAnalyseError)
//...
u := new(User)
//...
bandwidth := 242483;
url, err := url.Parse("http://simpli-city.eu/content.mp3")
if err == nil {
    params, err := AnalyseMediaResource(u, bandwidth, url)
    if err == nil {
        // use the params
    }
}
}

```

5.4.5.3.2 Media Data Transcoding

The method TranscodeMediaResource transcodes media data.

Parameters:

- params: A transcoding parameters object, described in Section 5.4.6.2..
- mediaResource: A multimedia resource object, e.g., an audio or video file)

Return Value:

Returns channel for the transcoded media data. That channel delivers chunks, described in Section 5.4.6.2.

Error Handling:

In a case of an error on starting the transcoding the second values is returned as a TranscodingError, described in Section 5.4.6.2. In a case of an error during transcoding chunks member transError is not nil, as described in Section 5.4.6.2.

Remarks:

In Listing 113, the signature of the method TranscodeMediaResource of the Media Data Transcoding is shown as well as an example for the usage of this method.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 175 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

Listing 113: Source Code Example – Media Data Transcoding: TranscodeMediaResource

```

/*
Transcodes media data.
in:
    params - the transcoding parameters
    mediaresource - the multimedia resource
out:
    {0} - the channel of data
    {1} - provides error informations in a case of an error
*/
func TranscodeMediaResource(params *TranscodingParameters, mediaresource *url.URL)
(chan<- *Chunk, *TranscodingError)

//...
dataChannel, length, err := TranscodeMediaResource(params, url)
if err == nil {
    //use length and the dataChannel
}

```

5.4.5.4 RESTful Interfaces (Server Side)

To describe the RESTful Interface, each supported call of the interface is described in a separate table. These tables are followed by listings showing examples for the JSON parameter if one is required for the call, as well as the JSON return value if one is provided by the call. For the RESTful interface it is assumed that an authentication of the user is assured.

5.4.5.4.1 Data Prefetching and Streaming Service – Add to History

This method provides a way to add historical data of a user profile.

Table 36: REST Interface Description – Add to History

Method	POST	URL	\$API_ROOT/users/:userId/history/add				
Description	Provides metadata of the performed http requests						
Parameter	userId	Required	yes	Possible Values	string	Description	A SIMPLI-CITY user id
JSON Object	http://simpli-city.eu/data_prefetching/json-schema/requests						
JSON Attribute	requests	Required	yes	Possible Values	array	Description	An array of requests
Example URL	\$API_ROOT/users/history/123214/add						
Response	HTTP status code						
HTTP Status Code		Required	yes	Possible Values	200	Description	OK
					400		Invalid parameters
					404		User not found
Example Response	HTTP/1.1 200 OK						

Listing 114 is an example for a valid JSON object, which is required for the REST interface shown in Table 36. It contains a JSON object called requests, which is an array as described in Section 5.4.6.4. Each item of this array contains an HTTP request combined with metadata.

Listing 114: JSON Example: Add to History

```

{
  "requests": [
    {
      "method": "GET",
      "version": "HTTP/1.1",
      "url": "http://www.w3schools.com/json/json_syntax.asp?user=ida",
      "parameter": [
        {
          "Host": "www.w3schools.com"
        },
        {
          "Connection": "close"
        },
        {
          "User-Agent": "Web-sniffer/1.0.46"
        },
        {
          "Accept-Charset": "ISO-8859-1,UTF-8;q=0.7,*;q=0.7"
        },
        {
          "Cache-Control": "no-cache"
        },
        {
          "Accept-Language": "de,en;q=0.7,en-us;q=0.3"
        }
      ],
      "datetime": "1997-07-16T19:20:30.45+01:00",
      "position": [
        {
          "latitude": 21.3225
        },
        {
          "longitude": 25.325
        },
        {
          "altitude": 23.3242
        },
        {
          "velocity": 60.155
        },
        {
          "direction": 23.15
        },
        {
          "accuracy": 13.355
        }
      ]
    }
  ]
}

```

5.4.5.4.2 Data Prefetching and Streaming Service – Get Prefetch List

This method provides a list of items, to be stored in the Local Prefetching Store.

Table 37: REST Interface Description – Get Prefetch List

Method	GET	URL	\$API_ROOT/users/:userId/prefetchlist				
Description	Provides the prefetchlist of an user.						
Parameter	max_count	Required	no	Possible Values	uint64	Description	Maximal count of items
Parameter	prefetch_id_offset	Required	no	Possible Values	uint64	Description	Prefetch_id of the last received prefetch item
Parameter	userId	Required	yes	Possible Values	string	Description	A SIMPLI-CITY user id
Example URL	\$API_ROOT/users/341/prefetchlist						
Response	HTTP status code and JSON object						
HTTP Status Code		Required	yes	Possible Values	200	Description	OK
					400		Invalid parameters
					404		User not found
JSON Object	http://simpli-city.eu/data_prefetching/json-schema/prefetchlist						
JSON Attribute	prefetchlist	Required	yes	Possible Values	array	Description	an array of request and response pairs
Example Response	HTTP/1.1 200 OK						

Listing 115 is an example for a valid JSON object, which is returned by the REST interface described in Table 37. It contains a JSON object called prefetchlist, which is an array as described in Section 5.4.6.4. Each item of this array is a pair of a request and a response.

Listing 115: JSON Example: Get Prefetch List

```

{
  "prefetchlist": [
    {
      "prefetch_id": 1,
      "request": {
        "method": "GET",
        "version": "HTTP/1.1",
        "url": "http://www.w3schools.com/json/json_syntax.asp?user=ida",
        "parameter": [
          {
            "Host": "www.w3schools.com"
          },
          {
            "Connection": "close"
          },
          {
            "User-Agent": "Web-sniffer/1.0.46"
          },
          {
            "Accept-Charset": "ISO-8859-1,UTF-8;q=0.7,*;q=0.7"
          },
          {
            "Cache-Control": "no-cache"
          },
          {
            "Accept-Language": "de,en;q=0.7,en-us;q=0.3"
          }
        ],
        "data": ""
      },
      "response": {
        "version": "HTTP/1.1",
        "status_code": "200",
        "status_msg": "OK",
        "parameter": [
          {
            "Location": "http://google.de"
          },
          {
            "Content-Type": "text/html; charset=UTF-8"
          },
          {
            "Date": "Thu, 29 Aug 2013 13:59:46 GMT"
          },
          {
            "Expires": "Sat, 28 Sep 2013 13:59:46 GMT"
          },
          {
            "Cache-Control": "public, max-age=2592000"
          },
          {
            "Server": "gws"
          },
          {
            "Content-Length": "218"
          },
          {
            "X-XSS-Protection": "1; mode=block"
          },
          {
            "X-Frame-Options": "SAMEORIGIN"
          },
          {
            "Alternate-Protocol": "80:quic"
          },
          {
            "Connection": "close"
          }
        ],
        "data": "<html>...</html>"
      }
    }
  ]
}

```

5.4.5.4.3 Data Prefetching and Streaming Service – Add to Prefetch List

This method provides a possibility to add items to a user's prefetch list.

Table 38: RESTful Interface Description – Add to Prefetch List

Method	POST	URL	\$API_ROOT/users/:userid/prefetchlist/add				
Description	Adds an item to the prefetchlist of a user						
Parameter	userid	Required	yes	Possible Values	string	Description	A SIMPLI-CITY user id
JSON Object	http://simpli-city.eu/data_prefetching/json-schema/prefetchlist_add						
JSON Attribute	requests	Required	yes	Possible Values	array	Description	An array of requests
Example URL	\$API_ROOT/users/356/prefetchlist/add						
Response	HTTP status code						
HTTP Status Code		Required	yes	Possible Values	200	Description	OK
					400		Invalid parameters
					404		User not found
Example Response	HTTP/1.1 200 OK						

Listing 116 is an example for a valid JSON object, which is required by the REST interface described in Table 38. It contains a JSON object called requests, which is an array of requests as described in Section 5.4.6.4.

Listing 116: JSON Example: Add to Prefetch List

```

{
  "requests": [
    {
      "method": "GET",
      "version": "HTTP/1.1",
      "url": "http://www.w3schools.com/json/json_syntax.asp?user=ida",
      "parameter": [
        {
          "Host": "www.w3schools.com"
        },
        {
          "Connection": "close"
        },
        {
          "User-Agent": "Web-sniffer/1.0.46"
        },
        {
          "Accept-Charset": "ISO-8859-1,UTF-8;q=0.7,*;q=0.7"
        },
        {
          "Cache-Control": "no-cache"
        },
        {
          "Accept-Language": "de,en;q=0.7,en-us;q=0.3"
        }
      ],
      "datetime": "1997-07-16T19:20:30.45+01:00",
      "position": [
        {
          "latitude": 21.3225
        },
        {
          "longitude": 25.325
        },
        {
          "altitude": 23.3242
        },
        {
          "velocity": 60.155
        },
        {
          "direction": 23.15
        },
        {
          "accuracy": 13.355
        }
      ]
    }
  ]
}

```

5.4.5.4.4 Data Prefetching and Streaming Service – Add Predictions (Locations)

The REST interface described in Table 39 provides functionalities to add predictions of upcoming locations of a user.

Table 39: RESTful Interface Description – Add Predictions (Locations)

Method	POST	URL	\$API_ROOT/users/:userid/prediction/location/add				
Description	A method to provide predictions of upcoming locations of a user						
Parameter	userid	Required	yes	Possible Values	string	Description	a SIMPLI-CITY user id
JSON Object	http://simpli-city.eu/data_prefetching/json-schema/predictions						
JSON Attribute	predictions	Required	yes	Possible Values	array	Description	an array of predictions
Example URL	\$API_ROOT/users/26741/prediction/location/add						
Response	HTTP status code only						
HTTP Status Code		Required	yes	Possible Values	200	Description	OK
					400		Invalid parameters
					404		User not found
Example Response	HTTP/1.1 200 OK						

Listing 117 is an example for a valid JSON object, which is required by the RESTful interface described in Table 39. It contains a JSON object called requests, which is an array of requests with positions and timestamps as described in Section 5.4.6.4.

Listing 117: JSON Example: Add Predictions (Locations)

```

{
  "requests": [
    {
      "method": "GET",
      "version": "HTTP/1.1",
      "url": "http://www.w3schools.com/json/json_syntax.asp?user=ida",
      "parameter": [
        {
          "Host": "www.w3schools.com"
        },
        {
          "Connection": "close"
        },
        {
          "User-Agent": "Web-sniffer/1.0.46"
        },
        {
          "Accept-Charset": "ISO-8859-1,UTF-8;q=0.7,*;q=0.7"
        },
        {
          "Cache-Control": "no-cache"
        },
        {
          "Accept-Language": "de,en;q=0.7,en-us;q=0.3"
        }
      ],
      "datetime": "1997-07-16T19:20:30.45+01:00",
      "position": [
        {
          "latitude": 21.3225
        },
        {
          "longitude": 25.325
        },
        {
          "altitude": 23.3242
        },
        {
          "velocity": 60.155
        },
        {
          "direction": 23.15
        },
        {
          "accuracy": 13.355
        }
      ]
    }
  ]
}

```

5.4.6 Content Format

This subsection defines the data content formats necessary to make use of the described component.

5.4.6.1 C++ Definitions

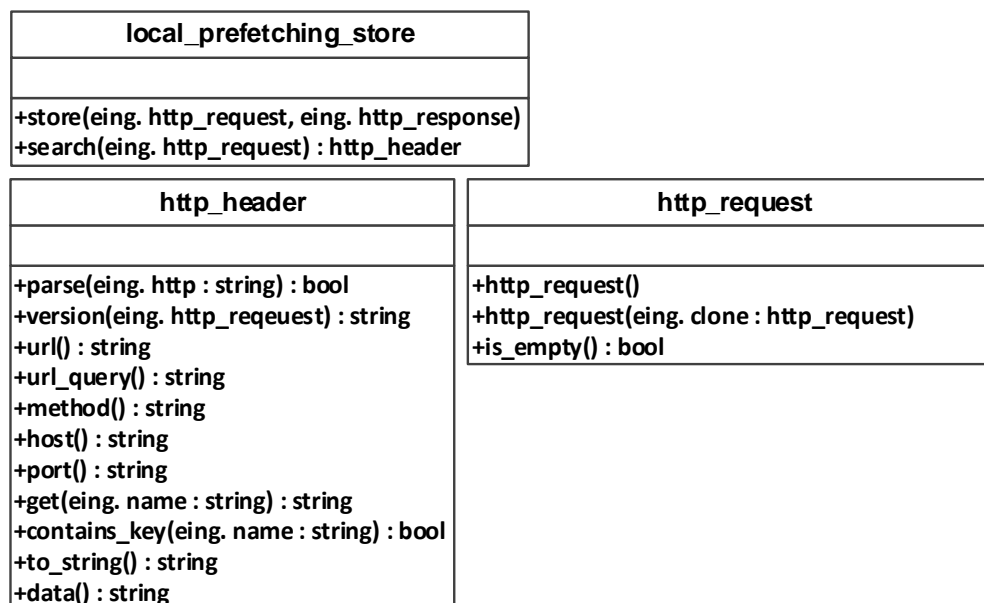


Figure 13: UML Class Diagram – C++ Components

The Local Prefetching Store is used by the Data Prefetching Gateway, which is written in C++. Therefore the Local Prefetching Store contains a C++ interface. To get a comprehensive understanding of interfaces defined in Section 5.4.5.1.1, additional information is needed.

Figure 13 depicts an overview of the provided functionalities of the Media Data Streams and Data Prefetching Logic component written in C++.

Listing 118 defines a C++ representation of a HTTP request. It is used in Listing 101 and Listing 102. After construction, the instance provides the bool parse(const std::string& http) method to set the represented HTTP request.

Listing 118: Class Signature – HTTP Request

```

class http_request
{
    public:
        http_request();
        http_request(const http_request&);
        ~http_request();
        bool parse(const std::string& http);
        std::string version() const;
        std::string url() const;
        std::string url_query() const;
        std::string method() const;
        std::string host() const;
        std::string port() const;
        get(const std::string& name) const;
        bool contains_key(const std::string& name) const;
        std::string to_string() const;
        std::string data() const;
};

```

Listing 119 defines a C++ representation of an HTTP request. It is used in Listing 101 and Listing 102. After construction, the instance provides the bool parse(const std::string& http) method to set the represented HTTP request.

Listing 119: Class Signature – HTTP Response

```

class http_response
{
    public:
        http_response();
        ~http_response();
        bool parse(const std::string& http);
        std::string version() const;
        std::string url() const;
        std::string url_query() const;
        std::string method() const;
        std::string host() const;
        std::string port() const;
        get(const std::string& name) const;
        bool contains_key(const std::string& name) const;
        std::string to_string() const;
        std::string data() const;
        bool is_empty() const;
};

```

Listing 120 defines the lpr_store_exception. It represents errors that can occur when calling the store function described in Listing 101.

Listing 120: Structure Signature – lps_store_exception

```

class lpr_store_exception : public std::logic_error
{
    public:
        lpr_store_exception(std::string message);
        lpr_store_exception(std::string message, std::exception&&
inner_exception);
        ~lpr_store_exception();
        bool has_inner_exception() const;
        std::exception inner_exception() const;
        virtual const char* what() const;
};

```

Listing 121 defines the lpr_store_exception. It represents errors which can occurs during calling the store function described in Listing 102.

Listing 121: Structure Signature – lps_search_exception

```

class lpr_search_exception : public std::logic_error
{
    public:
        lpr_search_exception(std::string message);
        lpr_search_exception(std::string message, std::exception&&
inner_exception);
        ~lpr_serach_exception();

        bool has_inner_exception() const;

        std::exception inner_exception() const;
        virtual const char* what() const;
};

```

5.4.6.2 Go Definitions

The Media Data Analysis and Media Data Transcoding are contained in the server-side Media Data Streams and Data Prefetching Logic. As described in Section 5.4.4.2, the whole server-side Data Prefetching and Streaming Service Logic component is written in the Go programming language. To get a comprehensive understanding of interfaces defined in Section 5.4.5.3.1 and Section 5.4.5.3.2, additional information is needed.

Figure 14 depicts an overview of the provided functionalities of the Media Data Streams and Data Prefetching Logic component written in Go.

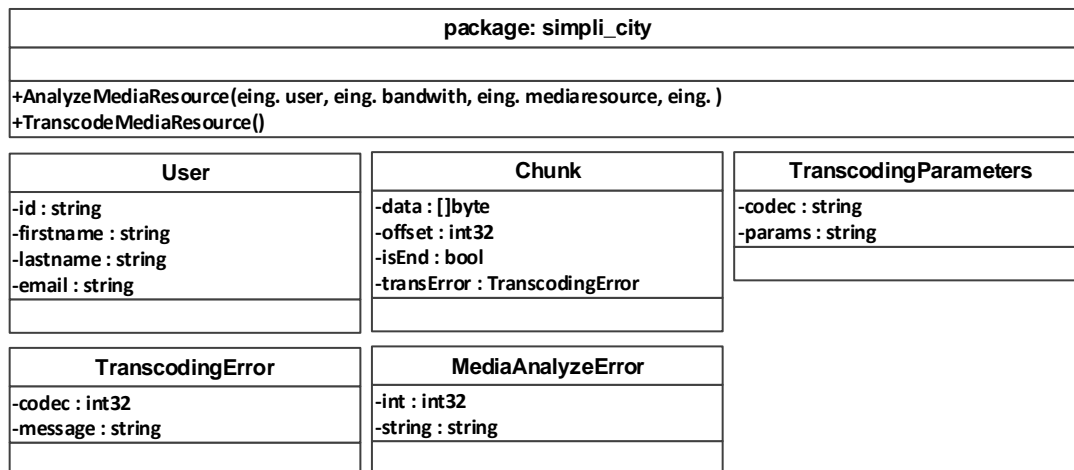


Figure 14: UML Class Diagram – Go Components

Listing 122 defines the structure TranscodingParameters, which stores detailed parameters about a transcoding process of multimedia files.

TranscodingParameters is returned by AnalyseMediaResource described in Listing 112. Furthermore TranscodingParameters is required by TranscodeMediaResource, described in Listing 113.

Listing 122: Structure Signature – Transcoding Parameters

```

type TranscodingParameters struct {
    codec string
    params string
}
  
```

Listing 123 defines the structure user, which represents a SIMPLI-CITY user. User is in use by AnalyseMediaResource, described in Listing 112.

Listing 123: Structure Signature – User

```

type User struct {
    id string
    firstname string
    lastname string
    email string
}
  
```

Listing 124 defines error types that can occur through calling `AnalyseMediaResource` or `TranscodeMediaResource`, described in Listing 112 and Listing 113. Each error defines a new type to identify the error domain and contains an error code to identify exactly what happened and a human readable error message.

Listing 124: Structure Signature – Error Types

```
const (
    HTTPError = iota
    NetworkError
    EncodingError
    InternalError
    UnknownError
)

type MediaAnalyseError struct {
    code    int
    message string
}

type TranscodingError struct {
    code    int
    message string
}
```

`TranscodeMediaResource` returns a channel of chunks, defined in Listing 125. It represents a chunk of the transcoded media resource combined with metadata.

The following elements are members of chunk:

- **data:** This element is an array of bytes and contains one chunk of the row data of the media resource.
- **offset:** This element is an integer and describes the current position in the stream or channel.
- **isEnd:** This element is a boolean and is true if this is the last chunk of data, otherwise it is false. It is also true if an error is occurred.
- **transError:** In a case of an error during transcoding, this element provides detailed information's of that error. If no error was occurred then this element is nil.

Listing 125: Structure Signature – Chunk

```
type Chunk struct {
    data      []byte
    offset    int32
    isEnd     bool
    transError *TranscodingError // this is not nil in a case of an error
}
```

5.4.6.3 Java Definitions

The client-side Data Prefetching API and Media Playback API are providing functionalities of the Media Data Streams and Data Prefetching Logic component to the app developers. As described in Section 5.4.4.1, the Data Prefetching API and Media Playback API are written in Java. To get a comprehensive understanding of interfaces defined in Section 5.4.5.2.1 and Section 5.4.5.2.2 additional information is needed.

Figure 15 depicts an overview of the provided functionalities of the Media Data Streams and Data Prefetching Logic component.

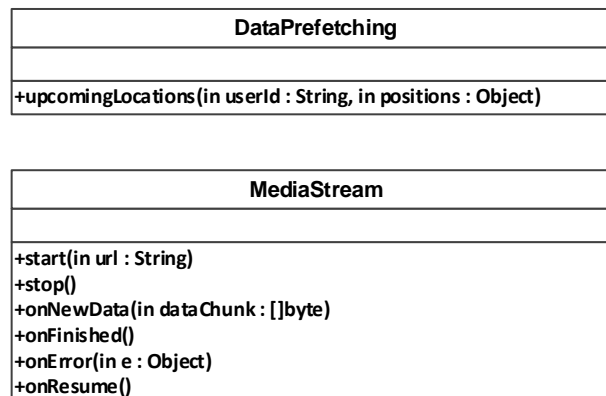


Figure 15: UML Class Diagram – Java Components

5.4.6.4 JSON Definitions

The server-side Data Prefetching and Streaming Service provides RESTful interfaces, described in Section 5.4.5.4, to interact with other components of SIMPLI-CITY. To get a comprehensive understanding of interfaces additional JSON schemas are needed. To minimize the variations of content formats in SIMPLI-CITY, this component selected the content format JSON. The same content format is selected for the Cloud-based Information Infrastructure. JSON was selected for the interactions with other components due to its lightweight, minimalistic approach, which keeps the data overhead low and provides fast transfers between the components.

Listing 126 to Listing 128 show the JSON Scheme for a stored HTTP Request and the according data format. This JSON document also contains some metadata about the PMA itself (e.g., current location). Listing 129 to Listing 131 show a list of prefetched HTTP requests with according responses prefetched by this component and stored in the Local Prefetching Store. Listing 132 to Listing 135 shows the JSON scheme is used to store a predicted response to the Local Prefetching Store for later use. Finally, Listing 136 to Listing 138 show the JSON scheme used to describe new predictions made by the PMA, which will then be stored in the Cloud-based Information Infrastructure.

Listing 126: JSON Schema – Requests Part 1

```

{
  "requests": {
    "type": "array",
    "id": "http://simpli-city.eu/data_prefetching/json-schema/requests",
    "required": true,
    "items": {
      "type": "object",
      "id": "http://simpli-city.eu/data_prefetching/json-schema/requests/0",
      "required": true,
      "properties": {
        "datetime": {
          "type": "string",
          "id": "http://simpli-city.eu/data_prefetching/json-schema/requests/0/datetime",
          "required": true
        },
        "method": {
          "type": "string",
          "id": "http://simpli-city.eu/data_prefetching/json-schema/requests/0/method",
          "required": true
        },
        "parameter": {
          "type": "array",
          "id": "http://simpli-city.eu/data_prefetching/json-schema/requests/0/parameter",
          "required": false,
          "items": {
            "type": "object",
            "id": "http://simpli-city.eu/data_prefetching/json-schema/requests/0/parameter/0",
            "required": false,
            "properties": {
              "PARAMETERNAME": {
                "type": "string",
                "id": "http://simpli-city.eu/data_prefetching/json-schema/requests/0/parameter/0/PARAMETERNAME",
                "required": false
              }
            }
          }
        },
        "url": {
          "type": "string",
          "id": "http://simpli-city.eu/data_prefetching/json-schema/requests/0/url",
          "required": true
        }
      }
    }
  }
}

```

Listing 127: JSON Schema – Requests Part 2

```

    "version": {
      "type": "string",
      "id": "http://simpli-city.eu/data_prefetching/json-
schema/requests/0/version",
      "required": true
    },
    "position": {
      "type": "array",
      "id": "http://simpli-city.eu/data_prefetching/json-
schema/requests/0/position",
      "required": true,
      "items": [
        {
          "type": "object",
          "id": "http://simpli-city.eu/data_prefetching/json-
schema/requests/0/position/0",
          "required": true,
          "properties": {
            "latitude": {
              "type": "number",
              "id": "http://simpli-city.eu/data_prefetching/json-
schema/requests/0/position/0/latitude",
              "required": true
            }
          }
        },
        {
          "type": "object",
          "id": "http://simpli-city.eu/data_prefetching/json-
schema/requests/0/position/1",
          "required": true,
          "properties": {
            "longitude": {
              "type": "number",
              "id": "http://simpli-city.eu/data_prefetching/json-
schema/requests/0/position/1/longitude",
              "required": true
            }
          }
        }
      ],
      {
        "type": "object",
        "id": "http://simpli-city.eu/data_prefetching/json-
schema/requests/0/position/2",
        "required": false,
        "properties": {
          "altitude": {
            "type": "number",
            "id": "http://simpli-city.eu/data_prefetching/json-
schema/requests/0/position/2/altitude",
            "required": false
          }
        }
      }
    ]
  }

```

Listing 128: JSON Schema – Requests Part 3

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 191 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

```

    },
    {
      "type": "object",
      "id": "http://simpli-city.eu/data_prefetching/json-  
schema/requests/0/position/3",
      "required": false,
      "properties": {
        "velocity": {
          "type": "number",
          "id": "http://simpli-city.eu/data_prefetching/json-  
schema/requests/0/position/3/velocity",
          "required": false
        }
      }
    },
    {
      "type": "object",
      "id": "http://simpli-city.eu/data_prefetching/json-  
schema/requests/0/position/4",
      "required": false,
      "properties": {
        "direction": {
          "type": "number",
          "id": "http://simpli-city.eu/data_prefetching/json-  
schema/requests/0/position/4/direction",
          "required": false
        }
      }
    },
    {
      "type": "object",
      "id": "http://simpli-city.eu/data_prefetching/json-  
schema/requests/0/position/5",
      "required": false,
      "properties": {
        "accuracy": {
          "type": "number",
          "id": "http://simpli-city.eu/data_prefetching/json-  
schema/requests/0/position/5/accuracy",
          "required": false
        }
      }
    }
  ]
}

```


Listing 129: JSON Schema – Prefetch List Part 1

```

{
  "prefetchlist": {
    "type": "array",
    "id": "http://simpli-city.eu/data_prefetching/json-schema/prefetchlist",
    "required": true,
    "items": {
      "type": "object",
      "id": "http://simpli-city.eu/data_prefetching/json-schema/prefetchlist/0",
      "required": true,
      "properties": {
        "prefetch_id": {
          "type": "number",
          "id": "http://simpli-city.eu/data_prefetching/json-schema/prefetchlist/0/prefetch_id",
          "required": true
        },
        "request": {
          "type": "object",
          "id": "http://simpli-city.eu/data_prefetching/json-schema/prefetchlist/0/request",
          "required": true,
          "properties": {
            "data": {
              "type": "string",
              "id": "http://simpli-city.eu/data_prefetching/json-schema/prefetchlist/0/request/data",
              "required": true
            },
            "method": {
              "type": "string",
              "id": "http://simpli-city.eu/data_prefetching/json-schema/prefetchlist/0/request/method",
              "required": true
            },
            "parameter": {
              "type": "array",
              "id": "http://simpli-city.eu/data_prefetching/json-schema/prefetchlist/0/request/parameter",
              "required": false,
              "items": {
                "type": "object",
                "id": "http://simpli-city.eu/data_prefetching/json-schema/prefetchlist/0/request/parameter/0",
                "required": false,
                "properties": {
                  "HttpParameter": {
                    "type": "string",
                    "id": "http://simpli-city.eu/data_prefetching/json-schema/prefetchlist/0/request/parameter/0/HttpParameter",
                    "required": false
                  }
                }
              }
            }
          }
        }
      }
    }
  }
}

```

Listing 130: JSON Schema – Prefetch List Part 2

```

    },
    "url": {
      "type": "string",
      "id": "http://simpli-city.eu/data_prefetching/json-
schema/prefetchlist/0/request/url",
      "required": true
    },
    "version": {
      "type": "string",
      "id": "http://simpli-city.eu/data_prefetching/json-
schema/prefetchlist/0/request/version",
      "required": true
    }
  }
},
"response": {
  "type": "object",
  "id": "http://simpli-city.eu/data_prefetching/json-
schema/prefetchlist/0/response",
  "required": true,
  "properties": {
    "data": {
      "type": "string",
      "id": "http://simpli-city.eu/data_prefetching/json-
schema/prefetchlist/0/response/data",
      "required": true
    },
    "parameter": {
      "type": "array",
      "id": "http://simpli-city.eu/data_prefetching/json-
schema/prefetchlist/0/response/parameter",
      "required": true,
      "items": {
        "type": "object",
        "id": "http://simpli-city.eu/data_prefetching/json-
schema/prefetchlist/0/response/parameter/0",
        "required": false,
        "properties": {
          "HttpParameter": {
            "type": "string",
            "id": "http://simpli-city.eu/data_prefetching/json-
schema/prefetchlist/0/response/parameter/0/HttpParameter",
            "required": false
          }
        }
      }
    }
  }
},

```

Listing 131: JSON Schema – Prefetch List Part 3

```
        "status_code": {
            "type": "string",
            "id": "http://simpli-city.eu/data_prefetching/json-
schema/prefetchlist/0/response/status_code",
            "required": true
        },
        "status_msg": {
            "type": "string",
            "id": "http://simpli-city.eu/data_prefetching/json-
schema/prefetchlist/0/response/status_msg",
            "required": true
        },
        "version": {
            "type": "string",
            "id": "http://simpli-city.eu/data_prefetching/json-
schema/prefetchlist/0/response/version",
            "required": true
        }
    }
}
}
```

Listing 132: JSON Schema – Add to Prefetch List Part 1

```

{
  "type": "object",
  "$schema": "http://simpli-city.eu/data_prefetching/json-schema/prefetchlist_add",
  "id": "http://jsonschema.net",
  "required": true,
  "properties": {
    "requests": {
      "type": "array",
      "id": "http://simpli-city.eu/data_prefetching/json-schema/requests",
      "required": true,
      "items": {
        "type": "object",
        "id": "http://simpli-city.eu/data_prefetching/json-schema/requests/0",
        "required": true,
        "properties": {
          "datetime": {
            "type": "string",
            "id": "http://simpli-city.eu/data_prefetching/json-schema/requests/0/datetime",
            "required": true
          },
          "method": {
            "type": "string",
            "id": "http://simpli-city.eu/data_prefetching/json-schema/requests/0/method",
            "required": true
          },
          "parameter": {
            "type": "array",
            "id": "http://simpli-city.eu/data_prefetching/json-schema/requests/0/parameter",
            "required": true,
            "items": {
              "type": "object",
              "id": "http://simpli-city.eu/data_prefetching/json-schema/requests/0/parameter/0",
              "required": false,
              "properties": {
                "HttpParameter": {
                  "type": "string",
                  "id": "http://simpli-city.eu/data_prefetching/json-schema/requests/0/parameter/0/HttpParameter",
                  "required": false
                }
              }
            }
          }
        }
      }
    }
  }
},

```

Listing 133: JSON Schema – Add to Prefetch List Part 2

```

    "position": {
      "type": "array",
      "id": "http://simpli-city.eu/data_prefetching/json-
schema/requests/0/position",
      "required": true,
      "items": [
        {
          "type": "object",
          "id": "http://simpli-city.eu/data_prefetching/json-
schema/requests/0/position/0",
          "required": true,
          "properties": {
            "latitude": {
              "type": "number",
              "id": "http://simpli-city.eu/data_prefetching/json-
schema/requests/0/position/0/latitude",
              "required": true
            }
          }
        },
        {
          "type": "object",
          "id": "http://simpli-city.eu/data_prefetching/json-
schema/requests/0/position/1",
          "required": true,
          "properties": {
            "longitude": {
              "type": "number",
              "id": "http://simpli-city.eu/data_prefetching/json-
schema/requests/0/position/1/longitude",
              "required": true
            }
          }
        },
        {
          "type": "object",
          "id": "http://simpli-city.eu/data_prefetching/json-
schema/requests/0/position/2",
          "required": false,
          "properties": {
            "altitude": {
              "type": "number",
              "id": "http://simpli-city.eu/data_prefetching/json-
schema/requests/0/position/2/altitude",
              "required": false
            }
          }
        }
      ]
    },

```

Listing 134: JSON Schema – Add to Prefetch List Part 3

```

    {
      "type": "object",
      "id": "http://simpli-city.eu/data_prefetching/json-
schema/requests/0/position/3",
      "required": false,
      "properties": {
        "velocity": {
          "type": "number",
          "id": "http://simpli-city.eu/data_prefetching/json-
schema/requests/0/position/3/velocity",
          "required": false
        }
      }
    },
    {
      "type": "object",
      "id": "http://simpli-city.eu/data_prefetching/json-
schema/requests/0/position/4",
      "required": false,
      "properties": {
        "direction": {
          "type": "number",
          "id": "http://simpli-city.eu/data_prefetching/json-
schema/requests/0/position/4/direction",
          "required": true
        }
      }
    },
    {
      "type": "object",
      "id": "http://simpli-city.eu/data_prefetching/json-
schema/requests/0/position/5",
      "required": true,
      "properties": {
        "accuracy": {
          "type": "number",
          "id": "http://simpli-city.eu/data_prefetching/json-
schema/requests/0/position/5/accuracy",
          "required": true
        }
      }
    }
  ]

```

Listing 135: JSON Schema – Add to Prefetch List Part 4

```
    },
    "url": {
      "type": "string",
      "id": "http://simpli-city.eu/data_prefetching/json-
schema/requests/0/url",
      "required": true
    },
    "version": {
      "type": "string",
      "id": "http://simpli-city.eu/data_prefetching/json-
schema/requests/0/version",
      "required": true
    }
  }
}
}
```

Listing 136: JSON Schema – Add Predictions Part 1

```

{
  "type": "object",
  "$schema": "http://simpli-city.eu/data_prefetching/json-schema/predictions",
  "required": true,
  "properties": {
    "requests": {
      "type": "array",
      "id": "http://simpli-city.eu/data_prefetching/json-schema/requests",
      "required": true,
      "items": {
        "type": "object",
        "id": "http://simpli-city.eu/data_prefetching/json-schema/requests/0",
        "required": true,
        "properties": {
          "datetime": {
            "type": "string",
            "id": "http://simpli-city.eu/data_prefetching/json-schema/requests/0/datetime",
            "required": true
          },
          "method": {
            "type": "string",
            "id": "http://simpli-city.eu/data_prefetching/json-schema/requests/0/method",
            "required": true
          },
          "parameter": {
            "type": "array",
            "id": "http://simpli-city.eu/data_prefetching/json-schema/requests/0/parameter",
            "required": true,
            "items": {
              "type": "object",
              "id": "http://simpli-city.eu/data_prefetching/json-schema/requests/0/parameter/0",
              "required": false,
              "properties": {
                "HttpParameter": {
                  "type": "string",
                  "id": "http://simpli-city.eu/data_prefetching/json-schema/requests/0/parameter/0/HttpParameter",
                  "required": false
                }
              }
            }
          },
          "position": {
            "type": "array",
            "id": "http://simpli-city.eu/data_prefetching/json-schema/requests/0/position",
            "required": true,

```


Listing 137: JSON Schema – Add Predictions Part 2

```

    "items": [
      {
        "type": "object",
        "id": "http://simpli-city.eu/data_prefetching/json-
schema/requests/0/position/0",
        "required": true,
        "properties": {
          "latitude": {
            "type": "number",
            "id": "http://simpli-city.eu/data_prefetching/json-
schema/requests/0/position/0/latitude",
            "required": true
          }
        }
      },
      {
        "type": "object",
        "id": "http://simpli-city.eu/data_prefetching/json-
schema/requests/0/position/1",
        "required": true,
        "properties": {
          "longitude": {
            "type": "number",
            "id": "http://simpli-city.eu/data_prefetching/json-
schema/requests/0/position/1/longitude",
            "required": true
          }
        }
      },
      {
        "type": "object",
        "id": "http://simpli-city.eu/data_prefetching/json-
schema/requests/0/position/2",
        "required": false,
        "properties": {
          "altitude": {
            "type": "number",
            "id": "http://simpli-city.eu/data_prefetching/json-
schema/requests/0/position/2/altitude",
            "required": false
          }
        }
      },
      {
        "type": "object",
        "id": "http://simpli-city.eu/data_prefetching/json-
schema/requests/0/position/3",
        "required": false,
        "properties": {
          "velocity": {
            "type": "number",

```

Listing 138: JSON Schema – Add Predictions Part 3

```

    "id": "http://simpli-city.eu/data_prefetching/json-  
schema/requests/0/position/3/velocity",  
    "required": false  
  },  
},  
{  
  "type": "object",  
  "id": "http://simpli-city.eu/data_prefetching/json-  
schema/requests/0/position/4",  
  "required": false,  
  "properties": {  
    "direction": {  
      "type": "number",  
      "id": "http://simpli-city.eu/data_prefetching/json-  
schema/requests/0/position/4/direction",  
      "required": false  
    }  
  }  
},  
{  
  "type": "object",  
  "id": "http://simpli-city.eu/data_prefetching/json-  
schema/requests/0/position/5",  
  "required": false,  
  "properties": {  
    "accuracy": {  
      "type": "number",  
      "id": "http://simpli-city.eu/data_prefetching/json-  
schema/requests/0/position/5/accuracy",  
      "required": false  
    }  
  }  
}  
]  
,  
"url": {  
  "type": "string",  
  "id": "http://simpli-city.eu/data_prefetching/json-  
schema/requests/0/url",  
  "required": true  
},  
"version": {  
  "type": "string",  
  "id": "http://simpli-city.eu/data_prefetching/json-  
schema/requests/0/version",  
  "required": true  
}  
}  
}  
}

```

5.4.7 Summary

Within this section, the Media Data Streams and Data Prefetching Logic component has been technically specified. Most importantly, this component is split into two main parts:

- The server-side subcomponents containing the main logic defining which elements to prefetch and describing them in a REST interface.
- Client-side subcomponents handling the feedback of the server-side subcomponents and performing the prefetching of elements by adding them to the Local Prefetching Store.

In addition to the prefetching functionality, this component also supports two additional functionalities: The client-side playback of media and the server-side transcoding and delivery of media to the PMA. This transcoding is performed to react to the mobile nature of the PMA, which has to cope with changing connection bandwidth and unstable connections.

It should be noted that this component works automatically without forcing app or service developers to actively care about the prefetching, as all HTTP requests are routed through a proxy in the Data Prefetching Gateway. The decision on what will be prefetched and the delivery of the prefetched data will be handled by the component itself and is based on historical and crowdsourced data.

6 Technical Specification: Mobility Services Framework

6.1 Service Runtime Environment

6.1.1 Major Design Decisions

The Service Runtime Environment is the basic framework for hosting and invoking of internal backend services (i.e., services running within the SIMPLI-CITY system). In addition, it offers functionalities for the invocation of data services and external services, i.e., services not running within the SIMPLI-CITY system but nevertheless used by mobile apps. For this, the Service Runtime Environment offers the functionalities to execute and bind services. Furthermore, it controls the monitoring of services; however, the actual monitoring is done through the Monitoring component as described in Section 6.2. The accounting part of Monitoring keeps track of services usage.

Usually, services are invoked by apps running in the PMA through the Application Runtime Environment (see Section 7.1). These requests will be processed in the Service Runtime Environment by the REST Proxy which is the single entry point for service consumers and which would call the actual service implementation, hosted within the Service Runtime Environment. The result will be then routed back to the app. For internal backend services, the Service Runtime Environment will also control the execution in terms of resource allocation and load balancing.

This subsection will compare existing technologies to implement service hosting functionality. These technologies will be used as a basis for the development of the SIMPLI-CITY Service Runtime Environment. In addition, the Service Management Server, a component exposing interfaces for controlling services' individual lifecycles will be briefly discussed (the full technical definition of the component can be found in Section 8.2) as well as the technical definition of the REST Proxy relaying calls to the real service implementations will be laid out.

During the discussion of the Global Architecture (deliverable D3.1) and Functional Specification (deliverable D3.2.1) discussions, the following major design decisions have been made:

Service Container Support:

SIMPLI-CITY needs a fully-fledged support for multiple simultaneously running decoupled services running in a container. These services have to be encapsulated but having the ability to lookup each other in a registry as well as expose information about themselves.

Service Lifecycle Management:

There should be the possibility to programmatically deploy, undeploy, start and stop services. Managing one service should be completely isolated and not affect other running services. These actions can be done via standard service container means (like command line or a web GUI) as well as programmatically as they can be invoked from the Service Development IDE plugin (see Section 8.2).

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 204 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

Service Monitoring Support Facilitation:

The overall approach should allow transparent and pluggable monitoring functionality. In other words, the service developer should be able to focus on developing their services and not taking care about monitoring. The actual monitoring will be provided and attached to running service instances at runtime by the hosting framework. Monitoring is discussed in more detail in Section 6.2.

Service SLA (Service Level Agreement) Fulfillment Checking:

The Service Runtime Environment should check the conformance to the Service Level Agreement (SLA) which is defined per service and possible per user and stored in the Service Registry. Measures like service response time, number of failed invocations and so on should be monitored transparently for the developer, similar to the monitoring functionality. In case of detected SLA violations, the means for taking countermeasures should be facilitated by the designed framework.

Pushing Functionality:

SIMPLI-CITY apps need to have the possibility to subscribe to certain types of events and then listen for event notifications asynchronously. This functionality will be implemented programmatically with the internal wrapping of the real push with periodic polling of the service. The Service Runtime Environment foundation does not need to provide any specific functionality for that, since pushing is provided through the Push Service of the Application Runtime Environment (see Section 7.1.4.7).

Scalability:

As the SRE will host a multitude of services, there should be ways for horizontal scalability with fault tolerance as added benefit. Some of the services will require a lot of CPU power and preferably, they have to have a little impact on other services hosted in the same container.

Standard Technologies:

In recent years, a number of standard technologies for Service-oriented Computing (SOC) have been introduced by industry alliances and initiatives as well as non-profit corporations. In order to facilitate technology exchange with fellow European projects as well as easy adaptation of SIMPLI-CITY technologies by third parties, the project should make use of a standard technology as foundation for the Service Runtime Environment. Furthermore, as will be seen in the upcoming technology comparison, most existing service frameworks provide an extensive set of functionalities. Implementing a new Service Runtime Environment from scratch would therefore mean to reinvent the wheel.

6.1.2 Technology Comparison

6.1.2.1 Comparison Criteria

Table 40: Criteria for Technical Specification

Parameter	Importance	Description
Hot Service Deployment	++	Hot deployment of services is essential for the functionality of SIMPLI-CITY. A newly developed service should have the possibility to be deployed into the Service Host container without affecting other already running services. The same applies to starting/stopping services as well as undeployment.
Supports embedded Tomcat	++	Whether an OSGi implementation supports embedding Tomcat for serving RESTful requests to the services.
Security Features	+-	There should be a possibility to restrict access to services of authorized users based on their individual access rights.
Fault Tolerance Mechanisms	+	As SIMPLI-CITY is a highly distributed system which allows a number of services to run within its Service Runtime Environment, it is very likely that faults will occur. However, such faults should not lead to a negative effect on the overall platform. In addition, especially if QoS faults occur, the Service Runtime Environment should be able to start according countermeasures.
Scalability	+	As SIMPLI-CITY will grow in number of services running, there should be a possibility to horizontally scale, that is, to seamlessly add new hardware to support more and more services as well as requests served.
Monitoring Features	++	SIMPLI-CITY needs the means to monitor the health of the entire system as well as individual services. The framework should support this from the beginning.
Included Service Registry	++	Services need to look up each other in a registry as well as expose information about themselves.

6.1.2.2 Possible Technologies and Comparison

Apart from few exceptions, the language of choice of the SIMPLI-CITY is Java. Therefore, technologies based on this platform will be considered. The SIMPLI-CITY Service Runtime Environment should provide a facility to simultaneously host multiple services, allow easy service lifecycle management, have a built-in service registry and in general be based on a mature technology. The Service Runtime Environment will run on a server or will respectively run on different servers in order to achieve fault tolerance and scalability. The following two options are a natural choice:

Apache Tomcat:

Apache Tomcat⁵⁰ or any other servlet container conformant to the Java servlet specification are well-known solutions to host web-based applications. Tomcat is responsible for managing the lifecycle of servlets, mapping a URL to a particular servlet and ensuring that the URL requester has the correct access rights. A servlet container implements the web component contract of the Java EE architecture, specifying a runtime environment for web components that includes security, concurrency, lifecycle management, transaction, deployment, and other services. While being an industry standard and incorporating many properties SIMPLI-CITY needs, Tomcat in general is aimed towards web-based applications, and SIMPLI-CITY services represent pure backend functionalities. For instance, the possibility to expose services in the JNDI (Java Naming and Directory Interface) is less powerful compared to OSGi native facilities and built-in service registry. The main protocol of service (inter)communication in Tomcat is HTTP, and while SIMPLI-CITY exposes services to the outside world via REST (based on HTTP), it also needs native Java method invocation possibilities between services inside the container.

OSGi Runtime Containers:

OSGi runtime containers like Apache Karaf⁵¹ or Eclipse Virgo⁵² are small OSGi-based runtimes providing a lightweight container onto which various components and applications can be deployed. These runtimes use a plugged-in core OSGi framework implementation (mainly Apache Felix⁵³, Eclipse Equinox⁵⁴ or Knopflerfish⁵⁵). In addition to standard OSGi features, a runtime container can provide console access to control the OSGi implementation, logging subsystem, deployment features, administration of the implementation and other valuable features not found in standalone OSGi implementations.

- **Apache Karaf** is a generic platform providing higher level features and services specifically designed for creating OSGi-based servers. Karaf is a small and lightweight OSGi-based runtime. This provides a small lightweight container onto which various bundles can be deployed. Best integrated with Apache Felix, Karaf supports any OSGi-compliant implementation.
- **Eclipse Virgo** is an open source, OSGi-based, Java application server. Virgo supports the deployment of OSGi bundles and unmodified Java web applications. Having features similar to Karaf, it also supports different OSGi implementation, Equinox being the most suited one.

The Service Runtime Environment can benefit from the underlying OSGi framework which is a module system and service platform for the Java programming language that implements a complete and dynamic component model, something that does not exist in standalone Java/VM environments. Applications or components (coming in the form of bundles for deployment) can be remotely installed, started, stopped, updated, and uninstalled without requiring a reboot; management of Java packages/classes is specified

⁵⁰ <http://tomcat.apache.org/>

⁵¹ <http://karaf.apache.org/>

⁵² <http://www.eclipse.org/virgo/>

⁵³ <http://felix.apache.org/>

⁵⁴ <http://www.eclipse.org/equinox/>

⁵⁵ <http://www.knopflerfish.org/>

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 207 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

in great detail. Application lifecycle management (start, stop, install, etc.) is done via APIs that allow for remote downloading of management policies. The service registry allows bundles to detect the addition of new services, or the removal of services, and adapt accordingly. On the other hand, the OSGi standard does not provide means for transparent monitoring / SLA fulfillment, nor does it envision pushing functionality needed in SIMPLI-CITY. These extra functionalities have to be developed separately.

There are several different OSGi implementations which could be chosen for the SIMPLI-CITY Service Runtime Environment. From the various possibilities, the following implementations are the most advanced ones and should therefore be discussed more closely:

- **Apache Felix** is a community effort to implement the OSGi R4 Service Platform and other interesting OSGi-related technologies under the Apache license. The OSGi specifications originally targeted embedded devices and home service gateways, but they are ideally suited for any project interested in the principles of modularity, component-orientation, and/or service-orientation. This implementation is one of the most mature and therefore having a wealth of documentation, developer discussion forums and other similar information. Being a general purpose implementation, Apache Felix is well received in the developers community and considered one of the easiest to use. Apache Felix does not aim to implementing the latest OSGi specification, but rather on being robust and proven platform.
- **Eclipse Equinox** is an implementation of the OSGi R4 core framework specification, a set of bundles that implement various optional OSGi services and other infrastructure for running OSGi-based systems. Its goal is to be a first class OSGi community and foster the vision of Eclipse as a landscape of bundles. As part of this, it is responsible for developing and delivering the OSGi framework implementation used for all of Eclipse. Equinox, being the reference implementation of OSGi, has the most up-to-date features of the OSGi specification, but is more complex in configuring and fine tuning compared to Apache Felix. In general, the learning curve for Equinox is steeper than Apache Felix.
- **Knopflerfish** is a leading universal open source OSGi service platform. Led and maintained by Makewave, Knopflerfish delivers significant value as the key container technology for many Java-based projects and products. Being a mature product implementing one of the latest versions of the specification, Knopflerfish is less known in the development world. The development team is supported and sponsored by only one company, compared to wider communities behind Apache Felix and Eclipse Equinox.

Table 41: Technology Selection Criteria and Comparison of Technologies for Service Runtime Environment

Parameter	Importance	Standalone Tomcat	Apache Felix	Eclipse Equinox	Knopflerfish	Apache Karaf (Runtime)	Eclipse Virgo (Runtime)
Generic Criteria							
Up-to-Datedness	+	10	8	10	10	10	10
Stability	+	8	8	8	7	8	8
Extensibility & Open Source/Standards	++	9	9	10	9	9	8
Familiarity	++	10	10	8	8	10	7
Performance	+	8	8	8	8	8	8
Interoperability	+	7	9	9	9	9	9
License	++	Apache License 2.0	Apache License 2.0	Eclipse Public License	BSD style license	Apache License 2.0	Eclipse Public License
Specific Criteria							
OSGi Framework/Runtime	N/A	N/A (Servlet container)	Framework	Framework	Framework	Runtime	Runtime
Hot Service Deployment	++	10	N/A	N/A	N/A	10	10
Supports embedded Tomcat	++	10	10	10	10	10	10
Security Features	+/-	10	8	8	8	8	8
Fault Tolerance Mechanisms	+	10	N/A	N/A	N/A	10	10
Scalability	+	10	N/A	N/A	N/A	10	10
Monitoring Features	++	8	8	8	8	10	10
Included Service Registry	++	0	10	10	10	10	10

6.1.3 Technology Selection

6.1.3.1 Selection for Service Runtime Environment

Apache Karaf based on Apache Felix was chosen as the foundation for the Service Runtime Environment (and especially the Service Host component) as providing a mature, scalable platform with all necessary features as hot deployment of services, their monitoring, isolation and interoperability as well as built-in service registry. Being well received in the developers community, there is a lot of documentation, examples as well as active forum discussions available for the combination of these two products.

Eclipse Virgo/Equinox both are excellent products, and choosing between them and Karaf/Felix or intermix of runtimes/OSGi implementations is rather subjective. An easier learning curve for Apache Felix and broader documentation helped to make the final decision. Knopflerfish is less known and would bear a risk of insufficient information should a need for solving peculiar issues arise.

It was decided not to use Tomcat as the basis of Service Runtime Environment for the reason of not being initially targeted at solving tasks SIMPLI-CITY poses. Being an industry standard and overall great servlet container, Tomcat implements the HTTP servlet concept which is not as good a match for what SIMPLI-CITY needs, compared to OSGi, especially in the area of native service intercommunication.

In the case for exposing RESTful service interfaces, OSGi implementation have an excellent possibility to run embedded Tomcat that is combining the best of two worlds

6.1.3.2 Missing Elements and Implementation Needs

Being a well-known framework used for years and having a wealth of information available online, the consortium does not expect any major drawbacks in the OSGi technology or in the Karaf/Felix implementations. However, some of the modules outlined below have to be developed to enhance functionality of Karaf/Felix in order to be used in the project:

Service Registry:

The service registry of the OSGi specification is deemed less satisfying for the project needs, therefore it will be only partly used and the remaining complementary part will be implemented using the Cloud-based Information Infrastructure as discussed in Section 6.4.

Monitoring and SLA Conformance:

In addition, the monitoring capabilities of Apache Karaf are not completely sufficient for the needs of SIMPLI-CITY. While the Monitoring component (see Section 6.2) is a SIMPLI-CITY subcomponent on its own, the Service Runtime Environment will nevertheless have to provide the possibility to integrate its functionalities. The same applies to functionality needed to check if the service instance complies with its Service Level Agreements (SLAs). This possibility will be realized in the Service Runtime Environment by the means of a REST Proxy described in Section 6.1.4.

Service Personalisation:

Per se, Apache Karaf does not provide any built-in functionalities for personalisation of service outputs. Similar to the Monitoring functionalities, SIMPLI-CITY will provide this in a separate subcomponent (see Section 6.3). However, once again, the Service Runtime Environment will have to make sure that these functionalities can be integrated and therefore used in arbitrary internal backend services.

Service Lifecycle Management:

Being an integral part of the OSGi specification, functionality like deploying, undeploying, starting and stopping of services is normally done manually by hand in these containers. SIMPLI-CITY needs to automate these processes by exposing publicly available interfaces, which might be called by the Service Development IDE plugin for the convenience of the developers.

REST Proxy for RESTful Service Invocation:

Functionality for seamless and transparent container located service invocation using a single entry point via REST calls (discussed below in the next section) is out of scope of OSGi specification. All this functionality has to be developed from scratch.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 210 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

6.1.3.3 Further Information and Conclusion from Technology Comparison

The standalone Tomcat provides less developed infrastructure for SIMPLI-CITY requirements. As there is a possibility to run embedded Tomcat within Apache Felix, it seems to be the natural choice to have the best of two worlds combining power and flexibility of OSGi and maturity of tried with time Tomcat for processing REST requests from client apps.

Selection of a particular OSGi implementation and runtime is rather subjective in nature. All discussed implementations are mature and well-known products, which are technologically well-suited for the project needs. Although significant extending and tweaking of these implementations will be required, all of them are deemed to be a sound and right basis for the SIMPLI-CITY Service Runtime Environment. Apache Karaf/Felix has been finally chosen as the easiest to use but still having an impressive array of features required in SIMPLI-CITY.

6.1.4 Component Structure

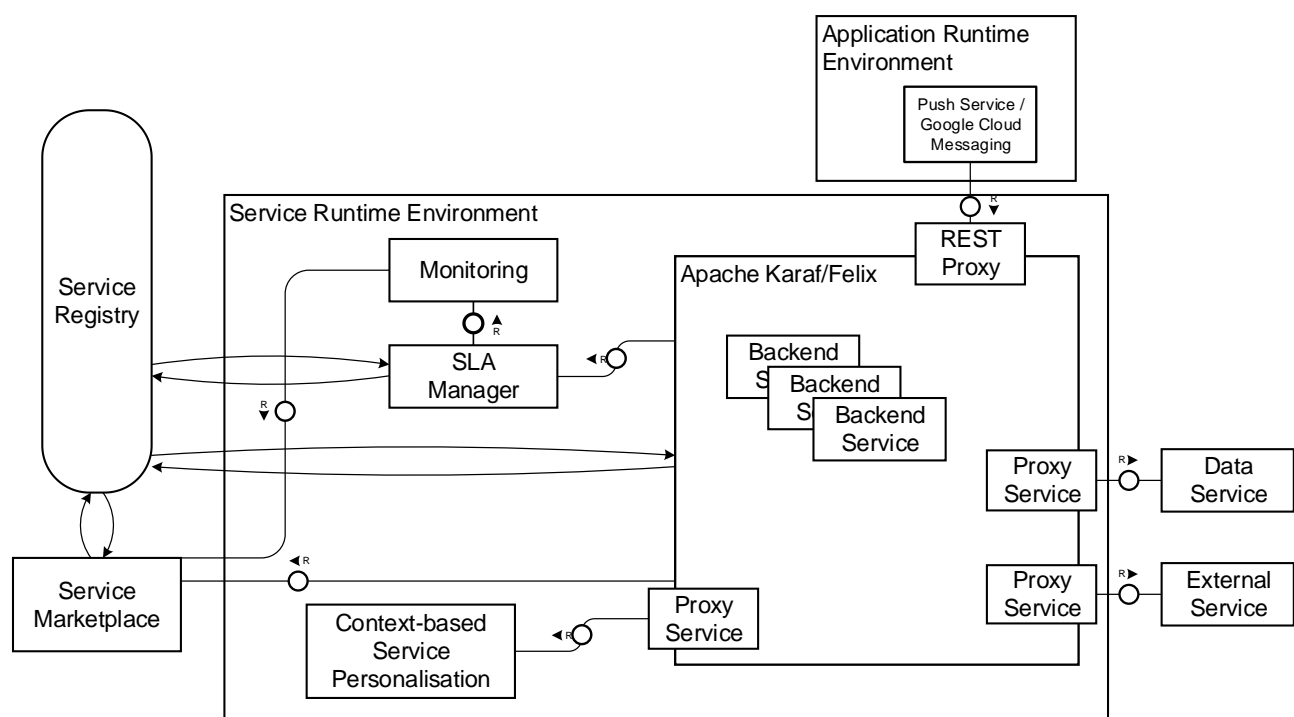


Figure 16: Service Runtime Environment Subcomponent and Interactions

Figure 16 shows the component diagram representing the Service Runtime Environment from the standpoint of the OSGi container that is focused on the services hosting framework. The differences between the Functional Specification (deliverable D3.2.1) and the updated component structure are described in the next paragraph.

If compared to the Functional Specification, it can be seen that the formerly envisioned Client Library was dropped in favor of using the standardized REST Proxy. The REST Proxy is in fact a separate service (bundle) deployed and running in the OSGi container. Exposing a single REST interface to the world, it allows other service invocations from the OSGi container, that is dispatching the calls. The proxy allows for extra functionality like transparent monitoring or gathering accounting data (discussed below). The Monitoring

(and now fused with it, Accounting) components are outside the scope of this section and are described in detail in Section 6.2, and the Proxy Generator is not used anymore as the REST Proxy is designed and implemented only once.

REST Proxy:

This component is designed to add an extra layer between the caller of the service and the service itself to supply monitoring and accounting data to the Monitoring component in addition to collecting statistics. This information can then be used either by the service or app provider. In addition, the REST Proxy provides the single entry point to invoke container services.

The REST Proxy is a service running in the container next to other services deployed there that relays all calls to other services through itself, collecting useful statistics. In addition, it calls the SLA Manager component upon every service invocation, providing user context information, e.g., the ID of the user. The SLA Manager in turn calls the Monitoring component and notifies it about starting service invocation. Upon completion of the service call, whether with an error or not, the REST Proxy will call the SLA Manager again to let it know the call is over. The SLA Manager will notify the Monitoring component about the service invocation end and can return instructions to the REST Proxy about how to recover from possible SLA violation (for instance, it can instruct the REST Proxy to repeat the service call in a case of failure).

An additional functionality of the REST Proxy is to check service usage license. Upon receiving the first call from a particular user, the proxy will invoke the `validateLicense()` method from the Payment API component of the Service Marketplace (see Section 6.5) to validate the license of the calling user. In the case of a license violation, the REST Proxy will return an error to the caller.

Proxy Services:

These are normal internal SIMPLI-CITY backend services running inside the container and delegating their functionality to other services (internal or external). They are mainly used to adapt an external service, running elsewhere, and fit it into the framework of SIMPLI-CITY. Exposing interfaces, which can be invoked by apps, these Proxy Services can do a variety of work ranging from aggregating data from several internal and external services to relaying calls to the external services, translating input from apps to the format the external service can understand and vice versa. Proxy Services are custom developed to fulfill their particular role.

Push Service / Google Cloud Messaging:

The Push Service is a part of the Application Runtime Environment. It is not a separate component of Service Runtime Environment, but rather a special service helping to realize the pushing functionality. It is discussed in detail in the Section 7.1 of this document.

Accounting:

The Functional Specification has foreseen a separate Accounting component that would provide accounting information related to usage of services to the Service Marketplace for subsequent billing. As this accounting functionality is tightly related to the Monitoring component, it was decided to merge these two components together into one Monitoring component (see Section 6.2).

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 212 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

SLA Manager:

The SLA (Service Level Agreement) Manager component is responsible for constant verifying of the overall OSGi container condition and performance as well as checking that services process requests according to the corresponding Service Level Agreements stored in the Service Registry. It checks, for example, that defined time limits for service method invocation or failure rate thresholds are not exceeded. If a violation of the SLA is detected, the SLA Manager will take adequate countermeasures.

The SLA Manager monitors behaviour of the container and the services in two ways:

1. Synchronously. This is the way described in the paragraph dedicated to the REST Proxy above. A service call failure can be detected this way and the SLA Manager then instructs the REST Proxy to reinvoke the failed service one more time.
2. Asynchronously. The SLA Manager has a separate background thread that runs parallel to other services and constantly checks the following parameters:
 - a. CPU and memory consumption by the OSGi container: Thresholds for these parameters are not configured in the service SLA agreements, but rather on the level of the OSGi container. The SLA Manager background thread periodically reads these parameters from the JMX, and if it detects, e.g., 100% CPU utilization for the preconfigured period of time or excessive memory consumption, it can take such countermeasures as notifying the system administrator via email (in the future it would be possible to start up another Virtual Machine instance on a separate computer to handle the extra load).
 - b. Service call hang-up: As the REST Proxy can never receive control back from a hung-up method of the service it is invoking, this situation has to be monitored asynchronously outside from the SLA Manager background thread. Once the Monitoring component receives notification from the REST Proxy about the beginning of the service invocation, it stores information about this event together with the timestamp in a database table. Upon the service invocation finish, the Monitoring component gets another notification from the REST Proxy, storing this information again in the same table. The SLA Manager background thread constantly checks this table and if it notices, that there is a service call that was initiated substantial time ago and not yet finished (and exceeding limits for that service in the SLA agreement), it can decide to take countermeasures ranging from sending an email to the system administrator to restarting the entire OSGi bundle containing hung-up service.

6.1.5 Interfaces

The Service Runtime Environment provides several methods for managing SIMPLI-CITY services running inside the OSGi container (this is done via Service Registry that exposes corresponding methods and is tightly integrated with the OSGi container) as well as obtaining auxiliary monitoring and accounting data (exposed by the Monitoring component).

Service invocations can be initiated by SIMPLI-CITY apps (via Application Runtime Environment) as well as other SIMPLI-CITY components. For this, the Service Runtime Environment offers functionalities, which are provided both via Java interfaces (Section 6.1.5.1) and REST interfaces (Section 6.1.5.2).

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 213 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

The RESTful interface exposed by the REST Proxy is shown in the diagram below.

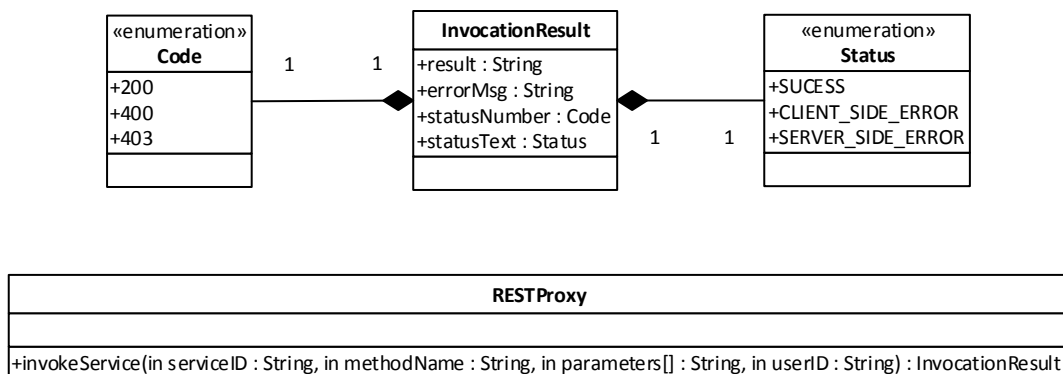


Figure 17: Class Diagram of Service Runtime Environment

6.1.5.1 Java Interfaces

6.1.5.1.1 Invoking a Service

This method offers functionality to invoke a SIMPLI-CITY backend service, registered in the Service Registry and running in the OSGi container (or externally via a corresponding Proxy Service, which is a normal internal backend service running in the container, delegating to the external one and performing necessary parameter mapping). In order to perform the invocation, the app needs to know the global unique service ID from the Service Registry. The ID of the user who is associated with the app will be passed on automatically by the Application Runtime Environment from the PMA side.

Parameters:

- **serviceID**: UUID of the service under which it is registered in the Service Registry.
- **methodName**: Name of the method exposed by the service to invoke.
- **parameters**: Array of method parameter values. These values will be cast to the types of the actual method parameters. Therefore, it is important that they are of correct types, otherwise the method invocation will fail.
- **userID**: UUID of the user in whose context the service is invoked (as registered in the Service Registry).

Return Value:

The service invocation result will be returned as an Object that can be cast to the type which is expected as result from the method call.

Error Handling:

- A **ServiceNotFoundException** is thrown if the service cannot be found in the Service Registry.
- A **ServiceInvocationException** is thrown in the case of an error in the working of the service.

Remarks:

The service return type is defined as an Object because of wide range of possible return results which can be a simple text or number, or can be an image or video, or even a part of an audio stream. Parameters are also loosely typed because of variety of services,

which have to be called using the single method. Supplying binary data as a parameter to the service usually requires encoding it first with base64 as this call normally will be done using the RESTful interface (see in the subsection below) and base64 is the preferred way of doing this. However, this is up to developer to decide on encodings and types of data, as it is their responsibility to parse it in the actual service implementation.

Listing 139 shows the API method signature for the invocation of a backend service running in the SIMPLI-CITY Service Runtime Environment.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 215 / 435
http://www.simpli-city.eu/	Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201			

Listing 139: Source Code Example – API Method Signature to Invoke a Service

```

/**
 * Invokes the service, identified by its id in the context of the
 * given user.
 *
 * @param serviceID: UUID of the service under which it is registered
 * in the Service Registry
 * @param methodName: Name of the method exposed by the service
 * to invoke
 * @param parameters: Array of method parameter values. These values
 * will be cast to the types of the actual method parameters
 * @param userID: UUID of the user in whose context the service is invoked
 * (as registered in the Service Registry).
 * @return Service invocation result
 * @throws ServiceNotFoundException in case the service cannot be found
 * in the Service Registry
 * @throws ServiceInvocationException in case invocation of the service
 * did not succeed
 */
Object invokeService(String serviceID, String methodName,
                    Object[] parameters, String userID)
                    throws ServiceNotFoundException,
                    ServiceInvocationException;

...

/**
 * Example service running in the OSGi container and registered
 * in the Service Registry under id: 528739A0-F508-4551-A12A-04A9B51718D0.
 */
public interface CityNameService {

    /**
     * Returns city name by its GPS coordinates.
     *
     * @param latitude latitude of the city
     * @param longitude longitude of the city
     * @return city name
     */
    String getCityName(double latitude, double longitude);
}

// Id under which CityNameService is registered in the Service Registry
String serviceID = "528739A0-F508-4551-A12A-04A9B51718D0";
// User who is associated with the given app (provided by ARE)
String userID = "138739A0-G508-1231-A12A-04A9B57464D0";
// Example coordinates
Object[] parameters = new Object[] { 48.208889d, 16.3725d };

String cityName = (String) sre.invokeService(serviceID, "getCityName",
                                             parameters, userID);

```


6.1.5.2 RESTful Interface

As mentioned above, the only method for invoking a service which will be exposed to the outside world is via a RESTful interface. Interfaces for getting monitoring and accounting data as well as managing lifecycle of services are provided by other components.

6.1.5.2.1 Invoking a Service

Mirroring the corresponding Java interface method (see Section 6.1.5.1), this interface offers the functionality to invoke a method on a SIMPLI-CITY service. Listing 140 and Listing 141 show example JSON messages for service request or service response, respectively.

Table 42: RESTful Interface Description – Invoking a Service

Method	POST	URL	\$API_ROOT/invokeService				
Description	Invokes the service, identified by its id in the context of the given user						
JSON Object	http://SIMPLI-CITY.eu/ServiceRuntimeEnvironment/JSON-Schema/InvokeServiceRequest						
JSON Attribute	serviceID	Required	yes	Possible Values	128 bit	Description	Unique service ID
JSON Attribute	methodName	Required	yes	Possible Values	String	Description	Name of the service method to invoke
JSON Attribute	parameters	Required	yes	Possible Values	array	Description	Array of parameter values to be passed to the method
JSON Attribute	userID	Required	yes	Possible Values	128 bit UUID	Description	Unique user ID in whose context the method is invoked (provided by the ARE)
Example URL	\$API_ROOT/invokeService						
Response	HTTP status code + JSON object						
HTTP Status Code		Required	yes	Possible Values	200 400 404	Description	Success Client error (incorrect parameters supplied) Server side error
JSON Object	http://SIMPLI-CITY.eu/ServiceRuntimeEnvironment/JSON-Schema/InvokeServiceResponse						
JSON Attribute	result	Required	yes	Possible Values	Serialized JSON	Description	Result of the service invocation in the form of serialized JSON
JSON Attribute	errorMsg	Required	yes	Possible Values	String	Description	In the case of error textual description of the error
Example Response	HTTP/1.1 200 OK						

Listing 140: Example JSON Supplied for Invoking a Service

```
{
  "serviceID": "528739A0-F508-4551-A12A-04A9B51718D0",
  "methodName": "getCityName",
  "parameters": [
    48.20889,
    16.3725
  ],
  "userID": "138739A0-G508-1231-A12A-04A9B57464D0"
}
```

Listing 141: Example JSON Response Containing Service Invocation Result

```
{
  "result": "Vienna",
  "errorMsg": null
}
```

6.1.6 Content Format

Data for service invocation contains different parameters related to invoked service details as well as parameters payload. Listing 142 shows the JSON schema for service invocation requests.

Listing 142: JSON Schema – Service Invocation Request

```
{
  "type": "object",
  "id": " http://SIMPLI-CITY.eu/ServiceRuntimeEnvironment/JSON-
Schema/InvokeServiceRequest",
  "properties": {
    "serviceID": {
      "type": "String",
      "required": true
    },
    "methodName": {
      "type": "String",
      "required": true
    },
    "parameters": {
      "type": "array",
      "required": true,
      "items": {
        "type": "object"
      }
    },
    "userID": {
      "type": "String",
      "required": true
    }
  }
}
```

Data returned from the service call contains either the JSON-serialized result of invocation or a textual description of the error occurred. Listing 143 shows the JSON schema for service invocation responses.

Listing 143: JSON Schema – Service Invocation Response

```
{
  "type": "object",
  "id": " http://SIMPLI-CITY.eu/ServiceRuntimeEnvironment/JSON-
Schema/InvokeServiceResponse",
  "properties": {
    "result": {
      "type": "object",
      "required": false
    },
    "errorMsg": {
      "type": "String",
      "required": false
    }
  }
}
```

6.1.7 Summary

The technical structure of the Service Runtime Environment has been detailed in this section. For this, a technology basis (OSGi) for hosting SIMPLI-CITY services was chosen and justified and aspects of the interaction with the outside world through the REST Proxy was laid out. In addition, the format of data exchanged has been defined and illustrated with examples and schemata.

An OSGi implementation Apache Felix running inside OSGi runtime Apache Karaf has been chosen as the basis of the Service Runtime Environment.

While being able to host independent services, provide access to their lifecycle (hot deployment, undeployment, starting, stopping of services) as well as having a built-in service registry and being overall a mature pair of products, the SIMPLI-CITY consortium needs to enhance the chosen technologies to fit for the project needs. This includes adding extended monitoring and SLA conformance checking, personalization functionality and extended programmatic lifecycle control.

6.2 Monitoring

6.2.1 Major Design Decisions

The purpose of SIMPLI-CITY's Monitoring component is to ensure defined Service Level Objectives targeting Quality of Service (QoS) aspects, e.g., the maximum response time or the minimum amount of concurrent requests a service should be able to handle, as well as to identify if services are responding to requests at all. This is done in order to detect Service Level Agreement (SLA) violations in advance and to identify possible bottlenecks with negative influence on the performance and induce respective counteraction. Aside from this, the Monitoring component also provides the monitored data to developers in order to provide feedback about the usage, potential errors or some general statistics to end users.

Monitoring provides the means to the Service Runtime Environment to provide feedback to end users as well as service developers about their services. Notably, the SIMPLI-CITY Monitoring component allows tracking the performance of both internal and external backend services as well as data services.

During the discussions of the Global Architecture (deliverable D3.1) and Functional Specification (deliverable D3.2.1), the following major design decisions were made:

Quantifiable Monitoring Aspects:

The Monitoring component has to be able to monitor at least the following different quantifiable aspects of services deployed in the SIMPLI-CITY Runtime Environment:

- Service invocations including information about the success of the invocation and the requester (user or app)
- QoS aspects, e.g., response time
- Interpolated resource consumption of single services and service invocations, e.g., CPU and RAM usage

Beside this, it allows to indirectly monitor external services, i.e., services which are not deployed within the SIMPLI-CITY Service Runtime Environment, through monitoring OSGi

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 219 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

services which are used as clients for external services. As external services are running outside of the control domain of SIMPLI-CITY, resource consumption will naturally not be monitored for them, but through internal monitoring, it is still possible to monitor if an external service is alive and has been carried out successful (see Section 6.2.3.1).

Extensibility:

Besides of the abovementioned quantifiable monitoring aspects, the Monitoring component should be designed in an extensible way, i.e., it should be possible to extend it by the means of monitoring further performance aspects in the future. Therefore, the Monitoring component needs to provide an extensible data model.

Up-to-Datedness and History:

The Monitoring component should provide up-to-date monitoring data of currently running services. This can be realized by continuously monitoring these services. Aside from this, the Monitoring component should also store historical data in a database in order to provide detailed reports about past timespans.

Lightweight Solution and Integration:

As monitoring is a helper functionality, it should be designed as lightweight as possible. Further, the integration into the SIMPLI-CITY Service Runtime Environment should not lead to too much effort and should only cover the functionalities discussed within the Functional Specification.

Scalability:

Since SIMPLI-CITY will host several different services at the same time and provide access to even more users to these services simultaneously, the Monitoring component should be able to scale in order to handle the different loads.

6.2.2 Technology Comparison

6.2.2.1 Comparison Criteria

For the selection of an appropriate technology, Table 43 gives the description of the different specific comparison criteria.

Table 43: Criteria for Technical Specification

Parameter	Importance	Description
Monitoring Aspects	++	<p>It minimizes the implementation efforts if the technology to be selected directly allows to monitor the abovementioned quantifiable monitoring aspects:</p> <ul style="list-style-type: none"> • Service invocations • QoS aspects, e.g., response time • Resource consumption of single services and service invocations, e.g., CPU and RAM usage

GUI	--	The data recorded by the Monitoring component can be presented in a GUI. However, this GUI will not be a standalone application but will be provided as part of the Service Marketplace, as this information is primarily interesting to service providers.
SLA Support	++	In order to provide the mentioned functionalities, it is mandatory to support SLAs. These are provided by the Service Registry and the functionality can be achieved based on an extensible data model.
Various Possibilities Regarding Database	--	In order to be as lightweight as possible, the Monitoring component should reuse an already available data storage instead of providing an own data store. As – if not motivated otherwise – all SIMPLI-CITY components make use of the Cloud-based Information Infrastructure as foundation for data storage, this will also be the case for the Monitoring component, i.e., an according bucket will be used.
Role Management	--	The access to the monitored data should be limited to authorized users only. Further, some data may be visible to users of a particular service and other data visible only for the developer who created this service. However, it is not the focus of the Monitoring component to develop an extensive role management system. Instead, it will reuse the roles defined by the Service Marketplace and allow access to monitoring data according to it.
Extensible Data Model	++	Since the Monitoring component is able to monitor services regarding different QoS attributes, it is a requirement for the data model to represent these different types correctly. Therefore, it is mandatory that the data model provided by the Monitoring component is extensible.
OSGi Compatibility	++	While there are no extensive monitoring solutions for OSGi which could be applied in the context of SIMPLI-CITY, a high degree of compatibility of OSGi frameworks and especially Apache Felix/Karaf is naturally of very high importance when choosing a monitoring technology for the Service Runtime Environment.

6.2.2.2 Possible Technologies and Comparison – External Monitoring Solutions

Monitoring of services can basically be done based on two different paradigms. An internal service monitoring is able to measure precisely all parameters of a service execution from the request of a service until the delivery of the result, including the observation of hardware usage parameters. However, this kind of observation is not able to take network properties into account. Another observation method is external service monitoring. This allows taking the network connection from different locations into account, but leads to difficulties for observing internal parameters in addition to security and legal problems because of the external storage of sensible data. The available technologies for external monitoring services are listed and described in the following as foundation for the technology comparison.

Cloudwatch:

Cloudwatch⁵⁶ is the in-house monitoring solution of the Amazon S3 cloud system. It is very easy to setup and delivers basic functionality, for example amount of disk used and write operations or CPU utilization. It is possible to auto scale the hosting power depending on these measurements. Cloudwatch provides a free of charge, very basic monitoring, but is mainly fee-based. The costs are not very transparent and are potentially open ended. Cloudwatch is only usable with applications hosted on Amazon servers, which renders it useless for SIMPLI-CITY.

Monitis:

Monitis⁵⁷ is a fee-based cloud monitoring solution which can easily be extended. Different monitoring locations around the world can be chosen and be configured accurately to cater for specific needs. In addition to that, Monitis delivers an extensive API and internal monitoring support. The GUI is browser-based and online, but monitored data can be exported. In addition to that, configuration is very easy and the GUI performance and ease of use is good with a rather fair pricing scheme.

Monitor.us:

Monitor.us⁵⁸ is the free monitoring solution from Monitis. It offers the same basic functionality, but possesses limitations in history, frequency, and selectable monitoring locations as well as the amount of monitors employable.

Membrane SOA:

Membrane SOA⁵⁹ is a complete Service-oriented Architecture with an own monitoring solution. It is open source (Apache 2.0) and includes the Membrane Monitor. This monitor is very flexible and can be used as proxy or even on client side. However, if the Membrane SOA is not used, the monitor configuration is rather oversized. Only HTTP and SOAP messages can be monitored, but the proxy provides powerful features like load balancing (if using the Membrane SOA as well).

ServiceMon:

ServiceMon⁶⁰ is a lightweight and easy to use solution, which is mainly useful for HTTP-GET monitoring. It was developed for the application RightCalc. It is programmed in .NET technology and therefore requires the .NET framework in version 4. The logging is text-based and based on an own syntax. ServiceMon's usage is mainly useful during development phases, but not suitable for larger monitoring. ServiceMon provides a Windows-based GUI.

Fiddler:

Fiddler⁶¹ is a debugging HTTP proxy, preferably used on the client side. It requires .NET framework version 4. The GUI is Windows-based and provides many configuration

⁵⁶ <http://aws.amazon.com/de/cloudwatch/>

⁵⁷ <http://www.monitis.com/>

⁵⁸ <http://www.monitor.us/>

⁵⁹ <http://www.membrane-soa.org/>

⁶⁰ <http://servicemon.codeplex.com/>

⁶¹ <http://fiddler2.com/>

possibilities. It is mainly useful during development phases in order to control traffic flow of an application.

Site 24x7:

Site 24x7⁶² is a cloud-based monitoring solution similar to Monitis, but its functionality is not as big as the one of Monitis. It is fee-based, but includes a small monitoring service for free. One of the most interesting features is the possibility for recording a user interaction within a web service which can be then monitored by replaying. The GUI is functional and browser-based.

SiteUptime:

SiteUptime⁶³ is also a cloud-based monitoring solution, but does not provide as much functionality as Monitis. It provides a small API and different monitoring locations. It supports HTTP, SMTP, POP, FTP, ping, DNS, HTTPS and different user roles. The GUI is browser-based. It has a very limited free of charge service, but is mainly fee-based.

Table 44: Comparison of Technologies for Service Monitoring – External Solutions

Parameter	Importance	CloudWatch	Monitor.Us	Monitis	Membrane SOA	ServiceMon	Fiddler	Site24x7	SiteUptime
Generic Criteria									
Up-to-Datedness	++	10	10	10	10	4	10	10	10
Stability	++	10	10	10	10	2	4	10	10
Extensibility & Open Source/Standards	++	0	8	8	6	6	0	6	5
Familiarity	+-	3	5	7	3	3	5	8	7
Performance	++	10	10	10	5	2	5	8	8
Interoperability	+-	0	10	10	6	0	0	8	8
License	(e.g., Apache 2.0)	Closed Source; Monthly fees	Closed source; for free	Closed source; Monthly fees	Apache 2.0	GPL	Closed source; for free	Closed source; Monthly fee	Closed source; Monthly fee
Specific Criteria									
Monitoring Aspects	++	10	7	10	8	5	5	10	10
SLA Support	++	7	10	10	0	0	0	10	10
GUI	+-	8	10	10	7	7	8	8	7
Various Databases	+-	0* (10)	0* (10)	0* (10)	8	0	0	0* (10)	0* (10)
Role Management	+-	10	8	8	0	0	0	10	0
Extensible Data Model	++	0	10	10	5	0	0	8	5

0*(10)=API access to data, storing in an own various database is possible

Table 44 shows the assessment of the different external monitoring solutions. Notably, most solutions allow to store data in different databases. However only indirectly, since the data is hosted at the provider of the monitoring solution and users can access them through an API and subsequently store them in their own database(s).

6.2.2.3 Possible Technologies and Comparison – Internal Monitoring Solutions

As initially mentioned, the second possible monitoring approach besides an external service monitoring is internal monitoring of services, i.e., using a monitoring solution as part of the OSGi-based Service Runtime Environment or at least run the monitoring solution on the same machine as the Service Runtime Environment. As sketched before, this offers the essential advantage to be able to observe all internal parameters in detail, including the hardware status. The available technologies for internal service monitoring are listed and described in the following as foundation for the technology comparison.

⁶² <http://www.site24x7.com/>

⁶³ <http://site-uptime.net/>

JMX:

Java Management Extension Technology⁶⁴ (JMX) is a part of the Java platform since version 2. It provides a simple, standard way of managing resources such as applications, devices, and services. JMX technology is dynamic and supports monitoring and managing resources as they are created, installed and implemented. It further provides the possibility to monitor and manage the Java Virtual Machine (JVM).

Jvmtop:

Jvmtop⁶⁵ (current version: 0.7.1) is a lightweight console application to monitor several JVMs running on the same machine. It is able to display JVM internal metrics of running Java processes including (but not limited to) used heap memory, max heap memory and CPU utilization.

Cacti:

Cacti⁶⁶ is an open source frontend for the monitoring tool RRDTool, which is mainly suitable for internal measurements, like CPU utilization, LAN throughput etc. The GUI is PHP-based. Cacti updates and development is community driven and has not been very active over the last year. Configuration is complex and error-prone. For the storage of measurement data, a MySQL database is needed.

PSI Probe:

PSI Probe⁶⁷ (current version: 2.3.3) is an open source community-driven fork of Lambda Probe. It is intended to manage and monitor an instance of Apache Tomcat and its deployed services. Unlike many other server monitoring tools, PSI Probe does not require any changes to existing apps, it provides all of its features through a web interface. The provided features include request monitoring (real-time traffic and response time on a per-application basis), session monitoring, JVM monitoring (memory) and system monitoring (CPU usage, memory usage and swap file usage). Naturally, PSI Probe is only able to provide full functionalities in an Apache Tomcat environment and is therefore only partially applicable in SIMPLI-CITY.

JavaMelody:

JavaMelody⁶⁸ (current version: 1.46.0) is a Java or Java EE application server monitor. It is able to measure and calculate statistics on real operation of several applications depending on its usage. JavaMelody is able to collect data about the average response time and number of invocations, including the mean execution times and percentage of errors of HTTP and SQL requests. In addition, JavaMelody is able to monitor the Java memory and Java CPU usage.

Apache Tomcat Manager:

Apache Tomcat Manager⁶⁹ delivers detailed application information hosted on Tomcat. It is possible to get an overview of running threads, Virtual Machine utilization and so on. The

⁶⁴ <http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html>

⁶⁵ <https://code.google.com/p/jvmtop/>

⁶⁶ <http://www.cacti.net/>

⁶⁷ <https://code.google.com/p/psi-probe/>

⁶⁸ <https://code.google.com/p/javamelody/>

⁶⁹ <http://tomcat.apache.org/tomcat-7.0-doc/manager-howto.html>

main drawback of this solution consists of the necessity to implement it in application code. The aforementioned PSI Probe is based on Apache Tomcat Manager, but provides more features and an easier implementation. Naturally, the Apache Tomcat Manager is only able to provide full functionalities in an Apache Tomcat environment and is therefore only partially applicable in SIMPLI-CITY.

Membrane SOA:

Membrane SOA⁷⁰ has already been presented as part of external monitoring solutions in the last subsection. It provides a complete Service-oriented Architecture with an own monitoring solution. It is open source (Apache 2.0) and includes the Membrane Monitor. This monitor is very flexible and can be used as proxy or even on client side. However if not using the Membrane SOA, the monitor configuration is oversized. Only HTTP and SOAP messages can be monitored, but the proxy provides powerful features like load balancing (in case it is used on top of a Membrane SOA as well).

Table 45: Comparison of Technologies for Service Monitoring – Internal Solutions

Parameter	Importance	JMX	Jvmtop	Cacti	PSI-Probe	JavaMelody	Apache Tomcat Manager	Membrane SOA
Generic Criteria								
Up-to-Datedness	++	10	7	5	8	8	3	10
Stability	++	10	4	7	10	7	5	10
Extensibility & Open Source/Standards	++	8	10	10	7	10	10	6
Familiarity	+-	6	2	3	6	7	5	3
Performance (includes configuration)	++	9	7	3	7	2	2	5
Interoperability	+-	10	2	3	4	2	2	6
License	(e.g., Apache 2.0)		GPLv2	GNU	GPLv2	LGPL	Apache 2.0	Apache 2.0
Specific Criteria								
Monitoring Aspects	++	9	5	0	10	6	0	8
GUI	+-	0	5	6	8	7	3	7
SLA Support	++	7	0	0	1	0	0	0
Various Databases	+-	5	5	0	5	10	5	8
Role Management	+-	0	0	8	0	6	0	0
Extensible Data Model	++	8	0	10	7	4	2	5
OSGi Compatibility	++	10	4	0	2	6	1	2

6.2.3 Technology Selection

6.2.3.1 Selection for Monitoring

External monitoring solutions provide quite powerful tools. However, they possess different disadvantages. One major disadvantage is the strong dependency on an external provider. In case an external solution provided by a corresponding external service provider would be employed within SIMPLI-CITY's monitoring solution, a lock-in effect would occur and the whole functioning of the monitoring solution within SIMPLI-CITY would be dependent on the external service provider providing this service in the way and to the conditions specified during the current technology selection. Thus, in case the external service provider would decide during the runtime of the project to change functionality and/or conditions for its service, or even worse would completely stop its service, severe

⁷⁰ <http://www.membrane-soa.org/>

problems would occur as a working monitoring solution on which SIMPLI-CITY relies would become non-functional from one moment to another. Additionally, as the gathered data would be completely under the sphere of control of the external service provider, severe data security and data privacy problems might be encountered. In consequence, it has been decided that even against the background that external monitoring solutions possess a huge range of functions and great visualization possibilities, these advantages are outweighed by the just mentioned disadvantages.

Therefore, the decision has been taken to realize an internal monitoring solution. However, this internal monitoring solution shall be able to not just monitor internal services, but as well externally provided services, e.g., by making use of a client OSGi service which serves as an endpoint to the external service, where the internally running client calls the required external service and the internally running client (OSGi service) is monitored. This would allow an indirect monitoring of the external service by monitoring the internal client's performance. This would furthermore circumvent the problem that active monitoring, e.g., via regular polling of an external service, might be legally questionable.

As it has been decided within the SIMPLI-CITY consortium that the Service Runtime Environment will be OSGi-based, the selection of JMX as basis for the SIMPLI-CITY Monitoring component is a natural decision, because JMX provides by far the best OSGi support compared to other examined solutions. Unfortunately, it seems as if there is not OSGi-specific monitoring solution which is publicly available.

Furthermore, JMX is competitive even in the context of the other parameters identified as important for the technology selection, like Up-To-Datedness and stability or measurement of service invocations or QoS parameters and possesses only slight disadvantages in the context of the minor criteria identified for technology selection in the former section. Therefore, it has been decided to use JMX as basis for SIMPLI-CITY's Monitoring component.

6.2.3.2 Missing Elements and Implementation Needs

As JMX brings along most of the functionalities needed within the SIMPLI-CITY Monitoring component, only some additional functionalities need to be implemented. This primarily includes the definition of APIs for storing monitoring data in a bucket in the Cloud-based Information Infrastructure and to receive SLA information from the Service Registry. Hence, it is necessary to implement the following functionalities.

Exposition of Monitoring Functionalities:

While JMX provides the basic monitoring functionalities for the SIMPLI-CITY Service Runtime Environment, these functionalities need to be integrated and helper methods should be provided which make it possible to easily integrate these functionalities into services. Hence, clearly defined methods are needed to monitor for a particular service invocation. This includes a method which can be used by the Service Watch subcomponent of the Service Registry (see Section 6.4).

Storage of Monitoring Data:

Monitoring data needs to be stored for analysis and accounting (see below) purposes. Monitoring data is also provided to service developers through the Service Marketplace. Within SIMPLI-CITY, the Cloud-based Information Infrastructure is used for data storage

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 226 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

and will therefore also be used here. However, it is necessary to provide an interface for data storage and retrieval.

Accounting Information:

Within SIMPLI-CITY, the Service Marketplace invoices the service users and checks if users actually have got licenses to make use of a service. Based on this information, a user can be charged for service usage. While the actual logic for this is part of the Service Marketplace, the Monitoring component needs to provide filtered accounting-related data through a well-defined interface. Notably, accounting for prefetched services will *not* be explicitly regarded in SIMPLI-CITY, i.e., all services to be prefetched are fully accounted.

User Roles:

While not providing a full-fledged user role model, the Monitoring component should provide data only to these users cleared for accessing data for particular services. Hence, the simple user role model from the Service Marketplaces should be adhered to.

6.2.3.3 Further Information and Conclusion from Technology Comparison

As mentioned before, the proposed solution for service monitoring is a custom solution that internally monitors all services and makes use of the functionalities of an OSGi-based monitoring framework, namely JMX. Monitoring information provided by such a framework is handled by an internal component called Monitoring Manager that acts as an adapter to the internal monitoring functionalities. Thus, the used framework can be replaced easily by adoption of the respective interfaces of the Monitoring Manager. In order to include external measurements, indirect monitoring of services hosted externally is provided through OSGi service representations ("clients") of the external services. These clients run inside the Service Runtime Environment and can therefore be easily monitored. The required components are described in detail in the following subsection.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 227 / 435
http://www.simpli-city.eu/	Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201			

6.2.4 Component Structure

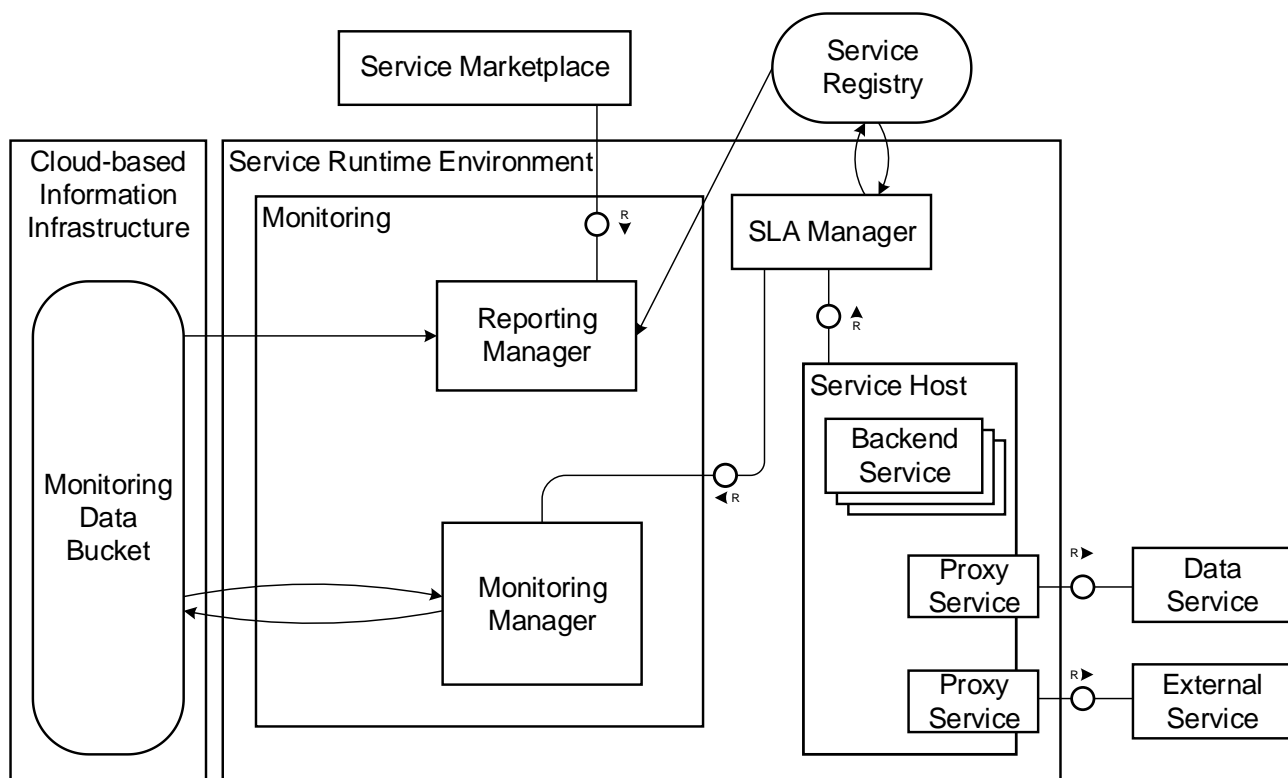


Figure 18: Component Structure of Monitoring

An overview of the different Monitoring subcomponents is given in the following paragraphs. The single components and dependencies are depicted in Figure 18. Since the Monitoring component is part of the Service Runtime Environment all other concerned components that interact with or provide data to the Monitoring component, are also illustrated. The main foundation for the proposed Monitoring solution is the use of OSGi as foundation for the Service Runtime Environment. This allows observations of running services in a comfortable way. The framework can be configured to automatically provide all basic monitoring information about services executed in the Service Runtime Environment. Nevertheless, interfaces to receive this information will be needed. Additionally, this rudimentary monitoring information has to be enriched with service-specific information to fulfil all needs of other SIMPLI-CITY components. Especially data preparation for accounting purposes is a complex task and the necessary observation and control of SLAs will be ambitious and challenging. All information about services running inside the Service Host will be observed by the Monitoring Manager.

Monitoring Manager:

The Monitoring Manager constitutes the central monitoring control. This component is invoked by the SLA Manager (see Section 6.1.4) in order to start monitoring a particular service invocation. It receives information about which service instance is invoked by which requester. The Monitoring Manager measures the service response time and the CPU and RAM usage. As soon as the Monitoring Manager gets invoked for a second time, i.e., stopMonitoring was called, the monitoring stops, and the recorded data is stored in the Monitoring Data Bucket thus it can be requested by the Reporting Manager.

Monitoring Data Bucket:

The Monitoring Data Bucket is responsible for storing information about executed services and the status of service requests to provide the Reporting Manager with the respective information. Actually, the Monitoring Data Bucket is a semi-structured database; a bucket in the Cloud-based Information Infrastructure will be used for this purpose. The main parameters that will be stored for all service calls is the service ID, the start and endpoint in time, the status of the service request, i.e., if the service execution was successful, the ID of the requester (user, app) of the service, and information about consumed hardware resources, e.g., CPU utilization and RAM, when applicable.

Reporting Manager:

The Reporting Manager provides the interfaces for the Service Marketplace to request monitoring and accounting data. If monitoring data about a specific service is requested, the Reporting Manager retrieves the necessary information from the Monitoring Data Bucket via an SQL query. In addition, the Reporting Manager can request according SLAs from the Service Registry.

6.2.5 Interfaces

The Monitoring component provides both Java and RESTful interfaces to other components: Java interfaces to components which are also part of the Service Runtime Environment and want to monitor services (or get monitoring data), and a RESTful interface to the Service Marketplace in order to request monitoring and accounting data. Figure 19 shows an overview of the provided interfaces of the Monitoring component. The class UUID is part of the java.util package and therefore not discussed in detail.

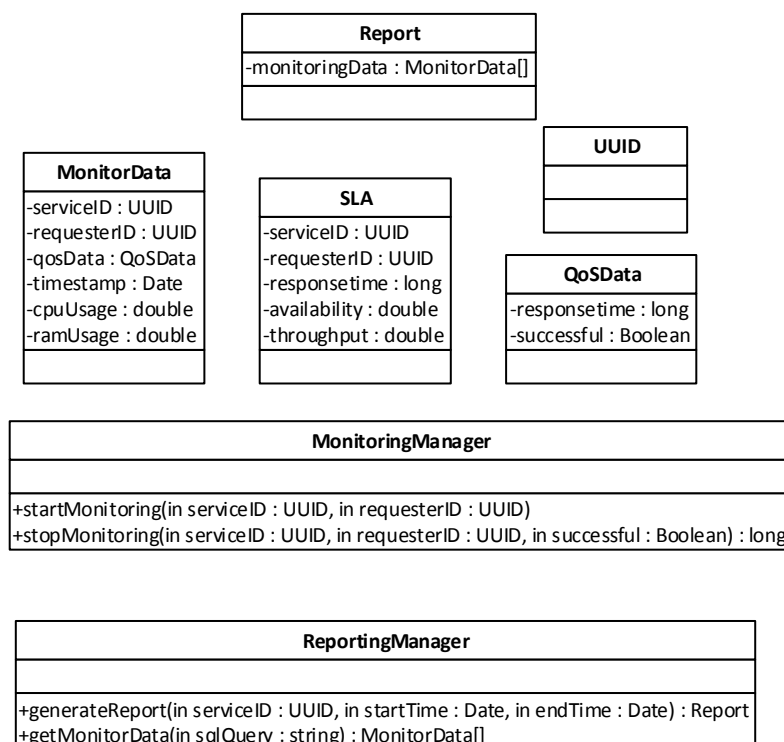


Figure 19: Class Diagram of the Monitoring Manager

6.2.5.1 Java Interfaces

6.2.5.1.1 Start Monitoring a Service Invocation

The Monitoring Manager gets invoked by the SLA Manager in order to measure the service execution time. In order to link this monitoring data to a particular service, it also receives a service ID. In addition to that, it receives a requester ID. This is needed for a later accounting process, i.e., if the call was successful the requester will be charged according to its agreed license.

Parameters:

- **serviceID:** ID of the service instance which will be invoked or of the service invocation
- **requesterID:** ID of the service requester

Return Value:

None

Error Handling:

A `ServiceNotFoundException` is thrown in case the `serviceID` is invalid.

Remarks:

It remains to mention, that the Monitoring Manager is not an active component. This means, it only measures the response time from the moment on which it got invoked the first time until the `stopMonitoring` method (see next subsection) got invoked.

Listing 144: Source Code Example – Invoke a Monitor for a Service Invocation

```
/**
 * Starts a Monitor for a particular service invocation; returns response
 * time
 *
 * @param serviceID, ID of the service instance to be invoked or service
 * invocation, respectively
 * @param requesterID, ID of the service requester *
 * @throws ServiceNotFoundException in case the service ID is invalid
 */
public void startMonitoring(UUID serviceID, UUID requesterID) throws
ServiceNotFoundException;

api.startMonitoring("2fad29d0-5355-11e3-8f96-0800200c9a66", "b07a3f80-5369-11e3-8f96-
0800200c9a66");
```

6.2.5.1.2 Stop Monitoring a Service Invocation

Whenever the monitoring of a service invocation should be ended, i.e., because the service invocation was finished, the `stopMonitoring()` method has to be called. It will stop the actual monitoring process, identified by a `serviceID` and `requesterID`. In addition to these two fields, a Boolean value will be added, defining whether the invocation was

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 230 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

successful or not. The recorded data will be stored in the Monitoring Data Bucket. In addition, the overall execution time will be returned.

Parameters:

- serviceID: ID of the service instance which was invoked or of the service invocation
- requesterID: ID of the service requester
- successful: Indicates if the service invocation was successful

Return Value:

The execution time in milliseconds will be returned.

Error Handling:

A ServiceNotFoundException is thrown in case the serviceID is invalid.

Remarks:

This method gets invoked by the SLA Manager as soon as the service invocation is finished. Internally in the Monitoring Manager, additional information about the CPU and RAM load will be recorded and stored in the Monitoring Data Bucket in combination with the service response time.

Listing 145: Source Code Example – Stop Monitoring a Service Invocation

```
/**
 * Stops monitoring a service and stores the results in the Monitoring Data
 * Bucket
 * @param serviceID, ID of the invoked service instance or service
 * invocation, respectively
 * @param requesterID, ID of the service requester
 * @param successful, indicates if the service invocation was successful
 * @return response time in milliseconds
 * @throws ServiceNotFoundException in case the service ID is invalid
 */
public long stopMonitoring(UUID serviceID, UUID requesterID, boolean successful)
    throws ServiceNotFoundException;

long responsetime = api.stopMonitoring("2fad29d0-5355-11e3-8f96-0800200c9a66",
    "b07a3f80-5369-11e3-8f96-0800200c9a66")
```

6.2.5.1.3 Retrieving Monitoring Data

The Reporting Manager provides an additional Java interface for retrieving historical monitoring data from the database. For that, the requester has to specify what monitor data it is interested in. This request is expressed in form of an SQL statement.

Parameters:

- sqlQuery: Specifies the wanted monitoring data

Return Value:

The found monitor data will be returned in form of a List filled with Plain Old Java Objects (POJOs) as described in Section 6.2.6.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 231 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

Error Handling:

If no monitoring data was found, an empty list will be returned. If an invalid SQL query was specified, a `SQLException` will be thrown.

Remarks:

It is expected that this method is error proof, i.e., if no monitoring data was found, because of an invalid service ID, timespan etc., an empty list will be returned.

Listing 146: Source Code Example – Request Monitor Data

```
/**
 * Retrieves historical monitoring data from the database
 *
 * @param sqlQuery, the SQL statement
 * @return A List of found monitor data.
 *
 * @throws SQLException if an invalid SQL statement was specified
 */
public List<MonitorData> getMonitorData(String sqlQuery) throws SQLException;

List<MonitorData> data = api.getMonitorData("Select * from monitoring where
serviceID='exampleID' and timestamp < new Date()");
```

6.2.5.1.4 Generate Report

The Reporting Manager may be requested by the Service Marketplaces in order to generate a Report over a defined timespan for a particular service and requester. The information will be gathered from the Monitoring Data Bucket. In addition, the Report Manager queries the Service Registry for the corresponding SLA and adds it as an attribute to the monitoring data.

Parameters:

- `serviceID`: Specifies a service
- `requesterID`: Specifies a requester
- `startTime`: The start time telling what data shall be included
- `endTime`: The end of this timespan

Return Value:

The generated Report will be returned in form of Plain Old Java Objects (POJOs) as described in Section 6.2.6.

Error Handling:

If no monitoring data was generated so far, an empty report will be returned. In case of a non-existent `serviceID` or an invalid timespan, a corresponding exception will be thrown, i.e., `ServiceNotFoundException` or `InvalidArgumentException`.

Remarks:

If a service has never been invoked so far, no monitor data may exist. This is not seen as an error and therefore an empty report will be returned.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 232 / 435
http://www.simpli-city.eu/ Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201				

Listing 147: Source Code Example – Request Report

```

/**
 * Generates a Report
 *
 * @param serviceID - a unique serviceID
 * @param requesterID - a unique requesterID
 * @param startTime - defines the start time of the report
 * @param endTime - defines the end time of the report
 *
 * @return A Report will be returned
 *
 * @throws ServiceNotFoundException if an invalid serviceID was defined
 * @throws InvalidArgumentException if the timespan is invalid
 */
public Report generateReport(UUID serviceID, UUID requesterID, Date startTime, Date
endTime) throws ServiceNotFoundException, InvalidArgumentException;

Report report = api.generateReport(serviceID,requesterID, new Date(1389953295), new
Date());

```

6.2.5.2 RESTful Interfaces

6.2.5.2.1 Request Monitoring and Accounting Data

For charging service requesters, the Service Marketplace needs information about successful service invocations. Further, the Service Marketplace provides service developers with information about the non-functional (i.e., QoS) behaviour of past service invocations, including the number of successful service invocations. For this, the Service Marketplace issues an SQL query to the Reporting Manager, which will retrieve the data accordingly. The analysis of this data is done within the Service Marketplace. Notably, the SQL query is not provided by the user of the Service Marketplace – instead, the user provides some parameters and the SQL query is assembled automatically.

The query result is returned as Java ResultSet mapped on a JSON object or a number of JSON objects if there are several monitoring data entries, respectively. Table 46 provides the interface details. However, naturally, based on the stated SQL query, the JSON attributes in the response will vary. Notably to mention, that in this table the whole QoS Data (Listing 156) and SLA object (Listing 218) are not explicitly shown. While

Listing 148 shows the interface call with an example SQL query asking for monitoring data, Listing 149 shows the example response. In this listening, only two entries are shown for the sake of simplicity, however, since the query specifies a date range, usually a list of monitoring entries will be returned.

Table 46: RESTful Interface Description – Query Monitoring Data

Method	GET	URL	\$API_ROOT/queryMonitoring?:sqlQuery				
Description	Returns monitored data according to the given SQL query						
Parameter	sqlQuery	Required	yes	Possible Values	String with SQL query	Description	SQL query based on user input (provided by Service Marketplace)
Example URL	\$API_ROOT/queryMonitoring?sqlQuery=Select+++from+monitoringDB+where+timestamp++>+2013-09-29T16:08:54:563Z+and+timestamp+<+2013-10-29T16:08:54:563Z+and+serviceID=sampleID						
Response	HTTP status code + JSON object						
HTTP Status Code		Required	yes	Possible Values	200	Description	Results found
					400		Service results not available
JSON Object	http://simpli-city.eu/Monitoring/JSON-Schema/MonitorData						
JSON Attribute	serviceID	Required	yes	Possible Values	128 bit UUID	Description	ID of the monitored service
JSON Attribute	requesterID	Required	yes	Possible Values	128 bit UUID	Description	ID of the service requester
JSON Attribute	cpu_usage	Required	yes	Possible Values	double	Description	CPU usage in percent (1-100)
JSON Attribute	ram_usage	Required	yes	Possible Values	double	Description	RAM usage in MB
JSON Attribute	start_time	Required	yes	Possible Values	date	Description	Date when the data record was started
JSON Attribute	end_time	Required	yes	Possible Values	date	Description	Date when the data record was ended
JSON Attribute	qos_data	Required	yes	Possible Values	object	Description	A QoS Data object
JSON Object	http://simpli-city.eu/Monitoring/JSON-Schema/QosData						
JSON Attribute	sla	Required	yes	Possible Values	object	Description	A SLA object
JSON Object	http://simpli-city.eu/Registry/JSON-Schema/SLA						
Example Response	HTTP/1.1 200 OK						

Listing 148: Input Message for Querying Monitoring Data

```
$API_ROOT/queryMonitoring?sqlQuery=Select+*+from+monitoringDB+where+timestamp
+>+2013-09-29T16:08:54:563Z+and+timestamp+<+2013-10-
29T16:08:54:563Z+and+serviceID=sampleID
```

Listing 149: Response Message for Querying Monitoring Data

```
[
  {
    "monitor_data": {
      "serviceID": "e8e61f50-4163-11e3-aa6e-0800200c9a66",
      "requesterID": "231c8a60-5357-11e3-8f96-0800200c9a66",
      "cpu_usage": "65.78",
      "ram_usage": "150",
      "startTime": "2013-01-29T16:04:54:563Z",
      "endTime": "2013-01-29T16:05:54:563Z",
      "qos_data": {
        "responsetime": "25000",
        "successful": "true"
      },
      "sla": {
        "responsetime": "30000",
        "availability": "99.9",
        "throughput": "1000"
      }
    }
  },
  {
    "monitor_data": {
      "serviceID": "e8e61f50-4163-11e3-aa6e-0800200c9a66",
      "requesterID": "231c8a60-5357-11e3-8f96-0800200c9a66",
      "cpu_usage": "75.87",
      "ram_usage": "89",
      "startTime": "2013-01-30T16:04:54:563Z",
      "endTime": "2013-01-30T16:05:54:563Z",
      "qos_data": {
        "responsetime": "27000",
        "successful": "false"
      },
      "sla": {
        "responsetime": "3000",
        "availability": "99.9",
        "throughput": "1000"
      }
    }
  }
]
```

6.2.5.2.2 Generate Report

As described above in Section 6.2.5.1.4, the Service Marketplace may request a Report about past service invocations for a particular service and requester. The JSON format for the Report can be found in Section 6.2.6.

Table 47: RESTful Interface Description – Generate a Report

Method	GET	URL	\$API_ROOT/generateReport?:serviceID&:requesterID&:startTime&:endTime				
Description	Returns a generated report for a particular service						
Parameter	serviceID	Required	yes	Possible Values	128 bit UUID	Description	a unique serviceID
Parameter	requesterID	Required	yes	Possible Values	128 bit UUID	Description	a unique requesterID
Parameter	startTime	Required	yes	Possible Values	Date	Description	a start time
Parameter	endTime	Required	yes	Possible Values	Date	Description	a end time
Example URL	\$API_ROOT/generateReport?serviceID=e54df900-7f5f-11e3-baa7-0800200c9a66&requesterID=557fba60-82c1-11e3-baa7-0800200c9a66&startTime='01.01.2013,00:00:00CET'&endTime='01.01.2014,00:00:00CET'						
Response	HTTP status code + JSON object						
HTTP Status Code		Required	yes	Possible Values	200	Description	Results found
					400		Service results not available
JSON Object	http://simpli-city.eu/Monitoring/JSON-Schema/Report						
JSON Attribute	monitor_data	Required	yes	Possible Values	object[]	Description	An array of monitoring data in that particular time span
JSON Object	http://SIMPLI-CITY.eu/Monitoring/JSON-Schema/MonitorData						
Example Response	HTTP/1.1 200 OK						

Listing 150: Input Message for Generating a Report

```
$API_ROOT/generateReport?serviceID=e54df900-7f5f-11e3-baa7-
0800200c9a66&requesterID=557fba60-82c1-11e3-baa7-
0800200c9a66&startTime='01.01.2013,00:00:00CET'&endTime='01.01.2014,00:00:00C
ET'
```

Listing 151: JSON Response Message for Generating a Report

```
{
  "report": [
    {
      "monitor_data": {
        "serviceID": "e8e61f50-4163-11e3-aa6e-0800200c9a66",
        "requesterID": "557fba60-82c1-11e3-baa7-0800200c9a66",
        "cpu_usage": "65.78",
        "ram_usage": "150",
        "startTime": "2013-01-29T16:04:54:563Z",
        "endTime": "2013-01-29T16:05:54:563Z",
        "qos_data": {
          "responsetime": "25000",
          "successful": "true"
        },
        "sla": {
          "responsetime": "30000",
          "availability": "99.9",
          "throughput": "1000"
        }
      },
      {
        "monitor_data": {
          "serviceID": "e8e61f50-4163-11e3-aa6e-0800200c9a66",
          "requesterID": "557fba60-82c1-11e3-baa7-0800200c9a66",
          "cpu_usage": "75.97",
          "ram_usage": "89",
          "startTime": "2013-01-30T16:04:54:563Z",
          "endTime": "2013-01-30T16:05:54:563Z",
          "qos_data": {
            "responsetime": "12000",
            "successful": "false"
          },
          "sla": {
            "responsetime": "3000",
            "availability": "99.9",
            "throughput": "1000"
          }
        }
      }
    ]
  }
}
```

6.2.6 Content Format

6.2.6.1 Monitoring Data

Since historical monitoring data has to be stored and should be transferable from the Monitoring Data Bucket to the Service Marketplace, a semi-structured data structure is provided for this data. As described in Section 6.2.1, the following criteria are monitored and stored:

- Service ID: The ID of the service which was monitored
- Requester ID: The ID of the service requester
- QoS aspects: e.g., response time in milliseconds or the information if the service invocation was successful
- Average resource consumption of the underlying VM: e.g., CPU usage in percent and RAM usage in MB
- Start and end time: The time span defining when this monitor data was recorded
- SLA: A SLA defined for the service requester and the service ID; the corresponding JSON schema can be found in Listing 218

The mentioned fields are only preliminary and especially the QoS attributes could be extended in the future. Listing 152 shows the schema of the JSON serialization for the answer to a monitoring data request through the RESTful Interface presented above; Listing 153 shows the according Java class for monitoring data. Listing 218 shows the SLA JSON schema, while Listing 219 shows the corresponding Java class.

Listing 152: Monitoring Data JSON Schema

```

{
  "type": "object",
  "id": "http://SIMPLI-CITY.eu/Monitoring/JSON-Schema/MonitorData",
  "properties": {
    "monitor_data": {
      "type": "object",
      "required": true,
      "properties": {
        "serviceID": {
          "type": "string",
          "required": true
        },
        "requesterID": {
          "type": "string",
          "required": false
        },
        "start_time": {
          "type": "date",
          "required": true
        },
        "end_time": {
          "type": "date",
          "required": true
        },
        "cpu_usage": {
          "type": "double",
          "required": true
        },
        "ram_usage": {
          "type": "double",
          "required": true
        },
        "qos_data": {
          "type": "object",
          "id": "http://SIMPLI-CITY.eu/Monitoring/JSON-Schema/QosData",
          "required": true
        },
        "sla": {
          "type": "object",
          "id": "http://SIMPLI-CITY.eu/Registry/JSON-Schema/SLA",
          "required": false
        }
      }
    }
  }
}

```

Listing 153: Java Class: MonitorData

```

public class MonitorData {

    /**
     * The ID of the service which was monitored
     */
    private UUID serviceID;

    /**
     * The ID of the service requester
     */
    private UUID requesterID;

    /**
     * QoS aspects: E.g., response time in milliseconds, success as boolean ...
     */
    private List<QoSData> qosData;

    /**
     * CPU usage of the single services and service invocations in percent
     */
    private double cpuUsage;

    /**
     * RAM usage of the single services and service invocations in mega bytes
     */
    private double ramUsage;

    /**
     * The point of time when monitor data recording was started
     */
    private Date startTime;

    /**
     * The point of time when monitor data recording has ended
     */
    private Date endTime;

    /**
     * A SLA defined for the service requester and service id
     */
    private SLA sla;

    //... getter() and setter()
}

```


6.2.6.2 Report

The following listening shows the JSON schema for a generated Report. It contains a list of monitoring data describing the non-functional behaviour of a service between a given start time and end time. In addition, the property `monitor_data` (see Listing 152) contains information about the CPU and RAM usage during the given time span as well as further information. Similar to Monitor Data (see above) the Report class could be extended to cater for further analysis functionalities. Additional attributes may be added during development phase, therefore, for now, only a list of `monitor_data` is returned. While Listing 154 shows the JSON Schema, Listing 155 shows the corresponding Java class.

Listing 154: Report JSON Schema

```
{
  "type": "object",
  "id": "http://SIMPLI-CITY.eu/Monitoring/JSON-Schema/Report",
  "properties": {
    "report": {
      "type": "object",
      "required": true,
      "properties": {
        "monitor_data": [
          {
            "type": "object",
            "required": true,
            "id": "http://SIMPLI-CITY.eu/Monitoring/JSON-
Schema/MonitorData"
          }
        ]
      }
    }
  }
}
```

Listing 155: Java Class: Report

```
public class Report {
    /**
     * A list of monitored data
     */
    private List<MonitorData> monitorData;

    //... getter() and setter()
}
```

6.2.6.3 QoS Data

The QoS Data class is a simple POJO (Plain Old Java Object) representing service-specific Quality of Service attributes. For now, only the service response time and the information if a service invocation was successful are relevant, however, this class may be extended during development of SIMPLI-CITY to cater for further scenarios. Listing 156 shows the QoS Data JSON schema, while Listing 157 shows the corresponding Java class.

Listing 156: QoS Data JSON Schema

```
{
  "qos_data": {
    "type": "object",
    "id": "http://SIMPLI-CITY.eu/Monitoring/JSON-Schema/QosData",
    "required": true,
    "properties": {
      "responsetime": {
        "type": "long",
        "required": true
      },
      "successful": {
        "type": "boolean",
        "required": true
      }
    }
  }
}
```

Listing 157: QoS Data Java Class:

```
public class QoSData {

    /**
     * The response time expressed in milliseconds
     */
    private long responsetime;

    /**
     * Indicating if the request was successful
     */
    private Boolean successful;

    //... getter() and setter()
}
```

6.2.7 Summary

It has been identified that two possible paradigms exist, which can be used to realize the monitoring within the SIMPLI-CITY Service Runtime Environment. These are external service monitoring and internal service monitoring. Both approaches exhibit advantages and disadvantages. For both approaches several technological realizations have been examined and evaluated. Finally, after weighing up the different advantages and disadvantages, the decision has been taken to realize an internal monitoring solution.

Concerning the technology to be employed for this internal monitoring solution, JMX has exhibited the most promising benefits during the technology comparison. In particular against the background that a deep integration of the monitoring solution with the OSGi-based Service Runtime Environment is aimed at, the selection of JMX as base technology to realize its own monitoring solution, specially tailored to the needs within SIMPLI-CITY, was a logical decision. Therefore, within the course of the project, an internal monitoring solution will be developed on the basis of JMX technology and in particular means to realize the monitoring of external services even with such an internal monitoring solution will be integrated as well as the basic API means required by the other SIMPLI-CITY components to request and receive the monitoring and accounting data they require. In addition, to cater as well for security issues, means to support different user roles within SIMPLI-CITY's monitoring solution will be provided.

6.3 Context-based Service Personalisation

6.3.1 Major Design Decisions

The responsibility of the Context-based Service Personalisation component is to provide the functionalities which are needed for personalizing service output based on a particular context. The service personalization follows different approaches including:

- Location-based data service selection, where a service outcome depends on a particular location of the user.
- Proactive user notifications, which is the functionality of recognizing whether a user should be provided with a certain piece of information based on the current situation.
- Support of data prefetching functionalities, which is the means of providing the functionality to tell apps or services when to start prefetching data.
- Context-based service invocation, which allows the automated usage of user context data in service executions.

The Context-based Service Personalisation component is an important component in SIMPLI-CITY, since it provides basic functionalities for realizing a number of the use case services. In detail, this means, that many apps need the personalized output of services based on a user's context in order to provide the wanted functionalities to the end user.

The Context-based Service Personalisation component is an important helper component for several services and the focus on it lies in its stability and reliability. So far, only a few, primarily research-driven, software solutions in this field exist and the provided functionalities are quite limited. Therefore, it is very likely that a new custom solution will be created which fits the needs of SIMPLI-CITY. For that, the following major design decisions have been made:

Interoperability:

As the Context-based Service Personalization provides a crucial functionality to several services, its interoperability should be high, i.e., the interface including the required parameters and return values have to be well-defined.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 243 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

Scalability:

In theory, SIMPLI-CITY is thought to serve thousands of users simultaneously. Since the Context-based Service Personalisation component provides a crucial functionality for many of the services deployed in SIMPLI-CITY, it is important to maintain a certain level of performance and stability in order to provide a high Quality of Experience (QoE) to the users. One way to realize this is to support scalability in the sense of having the ability of serving a lot of requests at the same time while serving only a few at off-peak time.

Extensibility:

Based on the application domain, Context-based Service Personalization can provide different functionalities based on different types of context data. While in the SIMPLI-CITY setting location-based service personalization will be in the focus, it is very likely that during the development of this component new data sources and data types have to be added. Therefore, the Context-based Service Personalization has to provide an extensible data model.

6.3.2 Technology Comparison**6.3.2.1 Comparison Criteria**

As will be further explained below, there is no existing context-based application which SIMPLI-CITY could reuse as a foundation (see Sections 6.3.2.2 and 6.3.2.3). However, what is needed as one important building block is a publish-subscribe software solution. Hence, the criteria listed in Table 48 are related to publish-subscribe solutions, not to service personalisation solutions.

Table 48: Criteria for Technical Specification

Parameter	Importance	Description
OSGi Compatibility	++	Since the Service Runtime Environment is based on OSGi, it is required that the publish-subscribe solution is compatible with it.
Durable Messages	++	In some use cases it is required that the messages transfer is durable. i.e., if a message is marked as durable, the publish-subscribe solution has to ensure that the message is delivered to the subscriber. This means, it has to be resend if the transfer failed, or stored and resend to a later moment if the subscriber is offline.
Notifications	++	The publish-subscribe solution has to provide the possibility to send a notification and receive notifications. This feature can be used to intercept send messages for the purpose of logging and/or accounting.
Filters	+-	The support of filters within the publish-subscribe solution is useful for saving resources in order to keep the system as lightweight as possible. Filters enable publishers to reuse the same queue/topic for different subscribers.

Lightweight	++	In order to provide a fast and durable system, a lightweight solution is desirable.
Interprocess Communication	++	The publish-subscribe solution is required to support interprocess and cross platform communication since it is very likely that different components run on different servers, e.g., sensor data sources through the Sensor Abstraction Interoperability Interfaces have to be subscribed to by the Context-based Service Personalization component.
Distributed	++	In order to create a scalable platform, it is required for the publish-subscribe solution to support distributed communication. This means, an n to n communication should be possible, i.e., several publishers and subscribers distributed among the net but listening on the same queue/topic.

6.3.2.2 Possible Technologies and Comparison

For the realization of the Context-based Service Personalization component, it is required to identify adequate technologies to manage and evaluate context information including technologies supporting active notifications. For that, two different kind of categories need to be investigated: First, existing context-based applications and their functionalities need to be discussed. Since most of these applications are research-driven prototypes with very different basic concepts and underlying technologies, and for most of them the documentation is rather sparse, it was not possible to identify common criteria which could have been used for the technical comparison. Instead, in Section 6.3.2.3, characteristics and distinct features of these applications will be described and analysed in detail. Second, as discussed in the Functional Specification (deliverable D3.2.1, Section 6.3), one particular building block for the Context-based Service Personalisation component is a publish-subscribe solution, since a messaging component is needed which is able to support active notifications to announce context changes and start subsequent service personalisation. For this, the technical comparison in terms of criteria will be done in Section 6.3.2.4.

6.3.2.3 Context-based Applications – Comparison

Context-based applications are software components or systems which are able to take a particular piece of information into account when producing outcome or a result of a service. A set of information, e.g., calendar data including the current location of a user is defined as a context. By taking this information into account, a service can be personalized. So far, a number of different approaches have been presented by the research community, while software products in this area are scarce. In the following, mainly context-based systems in the field of Service-oriented Computing will be investigated.

Location-Aware Systems:

Location-aware systems, where information about a particular location is used in order to provide a personalized service, are quite common. Especially on smartphones, several apps are available, which provide the user with certain information about the current

location of a user. General examples are travelling and touring apps where information about Points of Interests (POIs) is provided to a user. Similar to that are all navigation apps which provide routing information to users depending on their location, even more, whenever the user gets lost, e.g., takes the wrong highway exit, the navigation gets updated. For Android devices, a famous example is the newly published Google Now⁷¹ feature, which takes into account calendar data, the current location a user is, traffic situations, etc. in order to provide a perfect route to a particular location, e.g., the users working place.

From a functional point of view, Google Now provides some interesting features. However, since most of these features are in a preliminary state, e.g., they only support American cities or the Google in-house calendar, and in combination with the lack of extendibility, SIMPLI-CITY is not able to make use of it. In SIMPLI-CITY, the focus lays on extendibility and openness, thus third party developers are attracted to develop additional features and functionalities.

In contrast to closed-source and proprietary applications, location-aware systems are also in the focus of many researchers: Examples are mentioned by [EPS+01], [PCB00], and [LSE+12]. However, since these prototypes are for the purpose of research, they focus on providing one particular functionality in order to fulfil simplified use cases. These use cases are often not applicable in a real world scenario, because they are based on laboratory-like settings, i.e., they work on a higher abstraction level.

Mobility Mediation Layer [PMS+11]:

As the name implies, the Mobility Mediation Layer (MML) is a middleware for the Internet of Services, which provides mechanisms to make use of heavyweight, SOAP-based services on mobile devices with restricted computing resources. It allows to intercept service calls and to adapt the messaging in order to minimize the sent data. For this, some context data about the mobile device is used. The usage of context data for other purposes and especially functional personalization has basically been foreseen, but concrete personalization functionalities are not provided. Furthermore, the sources of the MML are not available.

From a technological point of view, the MML only provides limited compatibility with the SIMPLI-CITY Mobility Services Framework for two reasons: First, it makes use of SOAP-based web services, while SIMPLI-CITY applies RESTful services. Second, the MML does not fit into the SIMPLI-CITY global architecture (see Section 2.1), as it provides a middleware which would have to be integrated between the Service Runtime Environment and the Application Runtime Environment, and necessarily lead to some overhead, even in cases where no personalization is applied. Instead, the Context-based Service Personalization subcomponent provides add-on functionalities which are only requested if needed.

Context Broker Architecture (CoBrA) [CFA03]:

The Context Broker Architecture is an agent-based architecture for supporting context-aware computing in so-called intelligent spaces, e.g., living rooms, vehicles, offices or meeting rooms that are populated with intelligent systems that provide pervasive computing services to users. The central unit of CoBrA is an intelligent context broker that

⁷¹ <http://www.google.com/landing/now/>

maintains and manages shared contextual model on behalf of several agents, which can be applications hosted by mobile devices.

In general, the idea behind CoBrA is interesting for SIMPLI-CITY. However, the application area of CoBrA and the general usage differs to the one of SIMPLI-CITY. Where in CoBrA mostly indoor agents are used to gather context information, in SIMPLI-CITY more real world factors are of interest. Despite the fact that CoBrA's sources are open and freely available, it is still in an early version and it is unlikely that a newer version will be published since the last update was done in early 2004.

Hospital context-aware system:

[MGR+03] created a context-aware system which is able to deliver text messages to mobile phones considering the owner's current location and role. In detail, this system was used in a hospital for the staff, e.g., for physicians, nurses, etc. A user could formulate a message that should be delivered to a doctor that enters a particular room. In addition to that, the user was able to specify a time in between the message should be delivered. The contextual elements in this system are aware of includes location, time, the user's role and device state. The system is designed in three layers, first: the sensor layer, i.e., the sensor which gathers the information about the current location, second: the reasoning layer which is responsible in finding a required action, i.e., whether a message should be send or not and finally the action module which delivers the message.

Context Weaver:

The Context Weaver [CBC+04] was developed at IBM in 2004. It is a platform that simplifies writing context-aware applications. Context providers register with the Context Weaver in order to provide their data to applications through a simplified uniform interface. Applications are able to access the data not by naming a particular provider of the data, but by describing the kind of data they need. The system will then respond with a suitable provider.

During the comparison of the technologies for Context-based Service Personalisation, it became obvious that their functionalities are very specific and differ to such an extent that it is not possible to provide a comparison using different categories, as it has been done for all other SIMPLI-CITY software components. The presented technologies for context-based applications are either closed-source and proprietary or just research platforms which are not reusable within SIMPLI-CITY. In addition, most of the context-based applications only make use of one particular type of context data, e.g., the location, and do lack of extendibility for new context data types. Specific functionalities for services are sparse. Therefore, in SIMPLI-CITY, the Context-based Service Personalisation will be built from scratch.

Despite building the Context-based Service Personalisation component from scratch, particular functionalities, e.g., notifications about context changes, will be based on existing technologies. Most importantly, for context changes and other notifications, a publish-subscribe solution should be applied. Potential technologies will be discussed in the next subsection.

6.3.2.4 Publish-Subscribe Solutions – Comparison

As the name implies, publish-subscribe solutions are software components which provide the feature of subscribing to a particular channel or a topic. This enables the possibility to actively notify the subscriber with new information, i.e., new data is pushed through that channel.

HornetQ:

HornetQ⁷² is an open source project for building multi-protocol asynchronous messaging systems. HornetQ currently makes use of JMS (Java Message Service) 1.1 and provides a native Stomp (Simple Text Orientated Messaging Protocol) implementation thus it can be used with a range of non-Java clients. Stomp⁷³ provides the means for message clients and brokers to exchange messages across different programming languages and software platforms. In addition to that, HornetQ can be accessed via websockets from a browser and provides a RESTful messaging interface over HTTP.

Since it is very likely that the SIMPLI-CITY publish-subscribe component will be used in many different services, a main focus while selecting a publish-subscribe technology for SIMPLI-CITY lays on its lightweighness and extensibility. In addition, because of the missing notification support, HornetQ will not be considered as a possible ready to use technology.

Guava libraries: Event Bus:

The Guava⁷⁴ project provides several Google's core libraries for Java-based projects. A subcomponent of this library is the Event Bus which allows publish-subscribe-style communication between components. However, it is not a general-purpose publish-subscribe system and not intended for interprocess communication. Therefore this library is out of question for usage within SIMPLI-CITY.

Apache Camel:

Apache Camel⁷⁵ is an open source project which allows developers to define various routing and mediation rules. Apache Camel uses an URI model to work directly with several transport or messaging models such as HTTP via websockets, ActiveMQ, JMS 1.1 or others. This URI model allows to developers to dynamically change the transport protocol while still using the same API calls. Apache Camel is rich in features such as the support of different messaging transport protocols and supports different data formats such as JSON and XML. Depending on the use case and the components which have to communicate with each other, different protocols may be needed. However, while JMS Topics or Queues support notifications, filters and durable messaging, switching to a different protocol such as websockets may lead to the loss of these features.

⁷² <http://www.jboss.org/hornetq>

⁷³ <http://stomp.github.io/>

⁷⁴ <http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/eventbus/EventBus.html>

⁷⁵ <http://camel.apache.org/>

ActiveMQ:

ActiveMQ⁷⁶ is an open source messaging and integration pattern server. It supports cross-language clients and protocols. ActiveMQ fully supports JMS 1.1 and is released under the Apache 2.0 License. Beside of JMS, ActiveMQ also supports through its Ajax API HTML5 sockets which makes it a candidate for cross platform communications in real-time.

Table 49: Comparison of Technologies for Publish-Subscribe

Parameter	Importance	HornetQ	Guava-libraries: Event Bus	Apache Camel	ActiveMQ
Generic Criteria					
Up-to-Datedness	++	8	8	9	8
Stability	++	10	10	10	10
Extensibility & Open Source/Standards	++	9	10	10	10
Familiarity	++	6	6	5	5
Performance	++	8	8	9	9
Interoperability	++	9	9	10	10
License		Apache 2.0	Apache 2.0	Apache 2.0	Apache 2.0
Specific Criteria					
OSGI Compatibility	++	8	6	10	10
Durable Messages	++	9	7	0/10*	10
Notifications	++	5	10	0/10*	10
Filters	++	8	0	0/10*	10
Lightweight	++	5	8	8	8
Interprocess Communication	++	10	0	10	10
Distributed	++	10	0	10	10
* depending on the underlying protocol, this feature is either fully supported or not at all (see description for details)					

6.3.3 Technology Selection

The support of publish-subscribe by the messaging libraries described above are in general quite similar. Each of them supports the Java Message Service (JMS) in version 1.1 which is crucial for SIMPLI-CITY since many services are going to be written in Java. However, since some services might be written in other languages (such as Go) a different communication protocol may be needed in addition. Examples are the newly upcoming websockets in HTML5.

6.3.3.1 Selection for Publish-Subscribe Functionalities

Even though the Context-based Service Personalisation component will be implemented from scratch, it will be based on an existing publish-subscribe software, since (as can be seen in Section 6.3.2.4) there is a number of well-established technologies which can be applied within SIMPLI-CITY.

During the technology research and their comparison, it became obviously that the middleware providing the publish-subscribe functionality should be up-to-date to modern standards, be stable and provide a lightweight integration. Further, since some backend

⁷⁶ <http://activemq.apache.org/>

services may be written in a different language as Java, interoperability and performance are also crucial factors. Because of these requirements, HornetQ and the Guava-libraries are already excluded. Only ActiveMQ and Apache Camel are possible frameworks, as both support the crucial feature of durable messaging, i.e., it is possible to mark a message as durable which ensures that this message is delivered to the wanted subscriber(s). At the end of the day, it has been decided to make use of Apache Camel as it provides a higher abstraction level than ActiveMQ and supports several different messaging transport protocols such as JMS, ActiveMQ and websockets.

6.3.3.2 Missing Elements and Implementation Needs

As mentioned above, most of the subcomponents of the Context-based Service Personalisation software component will be implemented from scratch. In particular, the following functionalities have to be implemented since they are either currently missing in the available technologies or there is simply no information about how a particular system or application has realized it.

Reasoning Support:

The core component of the Context-based Service Personalization component is the Context Reasoner. It provides the logic for different use case scenario where context-based service personalisation is envisioned. Since these use case scenarios are diverse, the Context Reasoner has to be dynamically designed, thus it is able to make use of several different context data types in order to reason about whether a particular action has to be invoked or not. The focus of this component is the research of how to efficiently reason on several different data types.

In general, the Context-based Service Personalization component supports four different fields of reasoning:

- Location-based data service selection: In order to provide the functionality of location-based service selection, it is necessary that the Context Reasoner retrieves location data. For that purpose, the Context Manager subscribes to a Context Sensor which itself monitors a location sensor, such as the GPS module of a user's smartphone. Whenever a new location is available, this data has to be updated in the Context Database. The Context Reasoner on the other hand, retrieves this information and reasons on it, i.e., it determines if it is required that a particular service is selected. Therefore, it is connected to the Service Registry, in which it looks up for a fitting service. As soon as a particular service is found, the Context Reasoner induces the required action, e.g., it invokes the particular service.
- Proactive user notifications is the functionality which allows recognizing in which situation a user should be proactively provided with a certain piece of information depending on different constraints. In order to provide this functionality it is necessary that the Context Reasoner is "aware" of the users which can be informed. For that, the user has to have an app installed on his mobile device which is able to receive these notifications. These apps make use of the publish-subscribe functionality of the Context-based Service Personalization component. Besides of this, the Context Reasoner has to be aware of which information type and its according source can be delivered to a specific user.

- Support of prefetching-relevant context: While the actual prefetching logic is part of the Media Data Streams & Prefetching Logic component (see Section 5.4), the Context Reasoner provides reasoning capabilities helping to decide whether data has to be prefetched or not. For that, the Context Reasoner has to be aware of the services which provide such a prefetching logic including the situation in which prefetching should be done. The information about the services can be retrieved from the Service Registry and the information about whether data has to be prefetched or not can be retrieved from the Context Database (see below).
- Context-based service execution: The functionality of context-based service execution is from the software implementation view similar to a service execution without (context-based) personalisation, but in the expected results quite different. This means, the context-based service execution handles the functionality of invoking a particular service based on the currently available context information. This context information will then be automatically provided as input data to the respective service.

Data Storage:

Within the Context-based Service Personalization component, it is required to store context information in a persistent data storage. As with other data storages in SIMPLI-CITY; the Cloud-based Information Infrastructure will be used for this.

Publish-Subscribe for Context Manager:

In order to retrieve always up-to-date information from the Context Sensors, the Context Manager should implement a publish-subscribe functionality. This feature is already available in many different forms and versions for Java. Therefore, SIMPLI-CITY will make use of an existing software solution.

As discussed in Section 6.3.2.4, SIMPLI-CITY will make use of a particular publish-subscribe solution, i.e., Apache Camel. However, this software needs to be integrated into the overall software system and most importantly be connected to the particular Context Sensors, the Context Reasoner, etc. (see next subsection).

Secure Access to the System:

As defined in the functional requirements, it has to be possible to limit the access to context data. The access should be configurable by roles which are assigned to users and/or services. However, for this SIMPLI-CITY will also make use of already ready implemented software solutions. It is possible to make use of the internally available security functionality of the Service Runtime Environment.

6.3.4 Component Structure

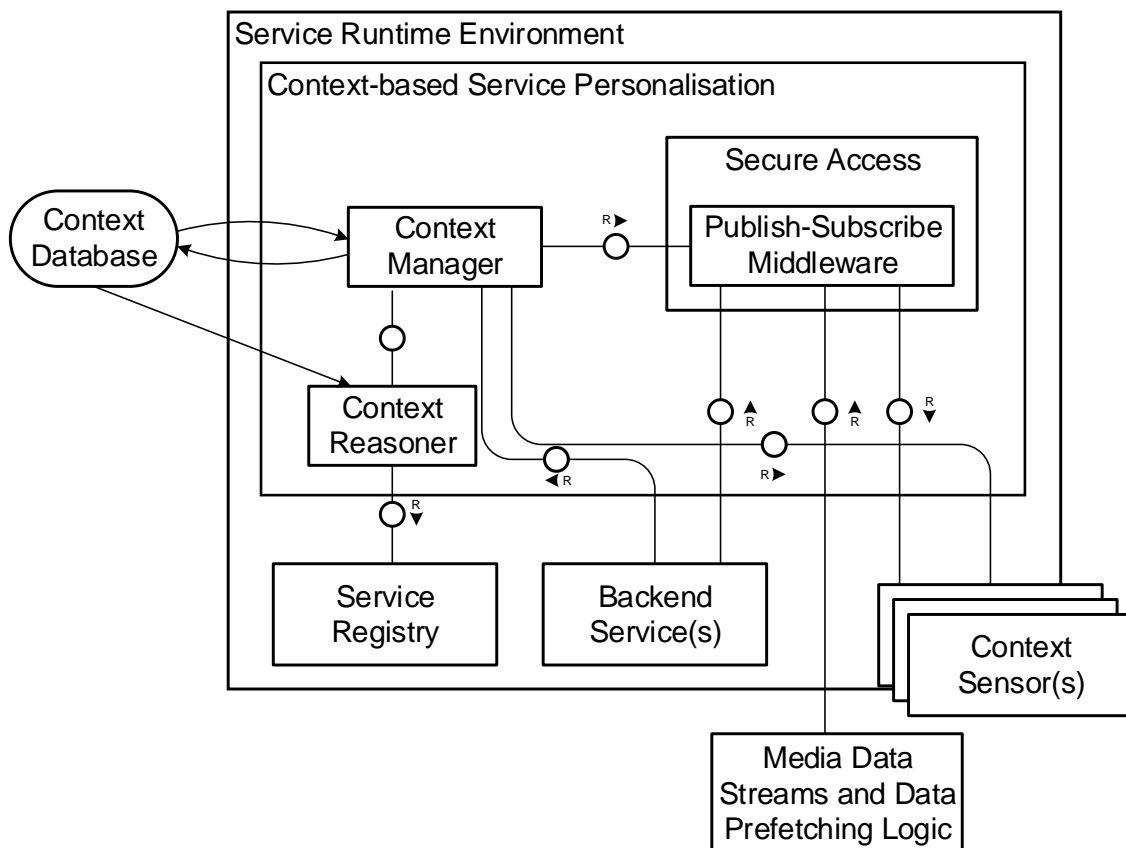


Figure 20: Component Structure of Context-based Service Personalization

Context Manager:

The Context Manager is the central subcomponent in the Context-based Service Personalization component. It is responsible for communicating internally with the Context Reasoner, which is on the one hand able to request new data from the Context Sensors and on the other hand able to notify subscribed Backend Services which are running inside the Service Runtime Environment, or the Media Data Streams and Data Prefetching Logic component.

In addition, it is able to communicate with the Context Sensors in order to retrieve updated context information immediately or it can send explicit requests. This data will then be written into the Context Database. Besides this, the Context Manager is able to retrieve historical data from the Context Database and forward it to a requesting component, such as backend services. For that, a publish-subscribe solution is deployed, which allows on the one hand, the registering at a Context Sensor in order to be informed if new data is available and on the other hand, backend services can register themselves at the Context Manager, thus the latter one can inform them if new context data is available.

Publish-Subscribe Middleware:

The Publish-Subscribe Middleware is the communication bridge between the backend services, Context Sensors, Media Data Streams and Data Prefetching Logic and the Context Manager.

- Backend services can subscribe to a particular type of context data or a particular Context Sensor in order to get informed if new data is available. Whenever new data is available, it will be pushed from the Context Manager through the Publish-Subscribe Middleware to the subscribed backend service. From the perspective of a service developer, accessing data from a Context Sensor is not different from subscribing to any other push-based data source which regularly provides data to the running service.
- The Context Manager can subscribe to Context Sensors in order to get informed when new context data is available. For that purpose, a threshold can be defined which defines the point at which the Context Sensor should send the data to the Context Manager, e.g., if a numerical number exceeds or falls below a particular threshold.
- The Media Data Streams and Data Prefetching Logic component can also subscribe itself to the Context Manager via the Publish-Subscribe Middleware in order to get informed whether it should start to prefetch a data or a media stream.

Secure Access:

The Publish-Subscribe Middleware is wrapped into a Secure Access component which is needed in order to restrict the request of context data to only allowed services. This means, that developers or data source provider are able to specify which services are allowed to read from the data source.

The Secure Access is a simple helper component which provides the functionality of limiting access to the data sources, thus only allowed services can request particular information.

Context Reasoner:

The Context Reasoner provides the core functionality of the Context-based Service Personalization component. It provides the logic which is needed in order to realize several use cases. In order to get the relevant data for the reasoning process, the Context Reasoner is connected to the Context Database and can retrieve the required information. In addition, for some use cases it might be necessary for the Context Reasoner to find a particular service, this information can be requested from the Service Registry which is located in the Service Runtime Environment. Further, the Context Reasoner is able to analyse the data from the Context Database and whenever it comes to the conclusion that a particular service has to be invoked, e.g., a particular backend service or if new data has to be requested from a Context Sensor. For that, the Context Reasoner is connected to the Context Manager.

Context Sensors:

The Context Sensors can be seen as data wrapping services, i.e., each Context Sensor is connected to one particular data source, be it a physical sensor such as a proximity sensor or GPS sensor, or to an Open Data service where it can retrieve free Open Data, or to a data processing component or to the Cloud-based Information Infrastructure where user-

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 253 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

specific data is stored. On the other hand, a Context Sensor is connected to the Publish-Subscribe Middleware in order to notify the Context Manager if new context data is available. The latter one is responsible for notifying the subscribed backend services and the Media Data Streams and Data Prefetching Logic component. At the end of the day, a Context Sensor is following an observer design pattern, i.e., it is the subject which informs its observers automatically about any state changes (via the Publish-Subscribe Middleware).

Context Database:

Since the Context Manager is subscribed to several Context Sensors, the amount of context data could become very large and should therefore not be kept in-memory. Therefore the Context Database will be used, which stores historical context data, if necessary. Storing historical data is required, since it serves as the basis for several reasoning approaches performed by the Context Reasoner. As with other databases in SIMPLI-CITY, a particular bucket within the Cloud-based Information Infrastructure (see Section 5.2) will be used. Because of the diversity of supported different Context Sensors it is not possible to define a common relational data model; however, all data types supported by the SIMPLI-CITY data model (as defined in deliverable D4.1.1) should be possibly supported. Therefore, a more dynamic data model will be used, and the data will be stored in a NoSQL database following a key-value storage principle. Naturally, this data model will follow the overall SIMPLI-CITY data model.

6.3.5 Interfaces

The Context-based Service Personalization component provides different methods to retrieve context-relevant data from data sensors or historical data from the Context Database. This information can be retrieved ad-hoc, i.e., another component sends a request to the Context Manager, or it can be retrieved in a publish-subscribe manner, i.e., a backend service registers itself at the Context Manager and gets informed whenever new context data is available. In detail, the Media Data Streams and Data Prefetching component, backend services, and Context Sensors are interacting with the Context-based Service Personalization component. For this, the Context-based Service Personalization component offers different methods which are presented in the next subsection: In order to access the features of the Context-based Service Personalization component, RESTful Interfaces (see Section 6.3.5.1) as well as Java Interfaces are provided (see Section 6.3.5.2). For the interfaces, it is assumed that an authentication of the user is assured and the user ID is provided additionally for each API call. As discussed in the last subsection, the Secure Access subcomponent will make sure that only eligible software components use functionalities of the Context-based Service Personalisation component.

Figure 21 represents the class diagram for the Context-based Service Personalisation. As in the interface description Sections 6.3.5.1 and 6.3.5.2, only externally available interfaces are represented. As the RESTful interfaces are only wrappers for the corresponding Java class, they are not stated in this diagram. Notably, the ContextData class depicted in the figure is a placeholder, since all data from the SIMPLI-CITY data model (see deliverable D4.1.1) could actually become context data.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 254 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

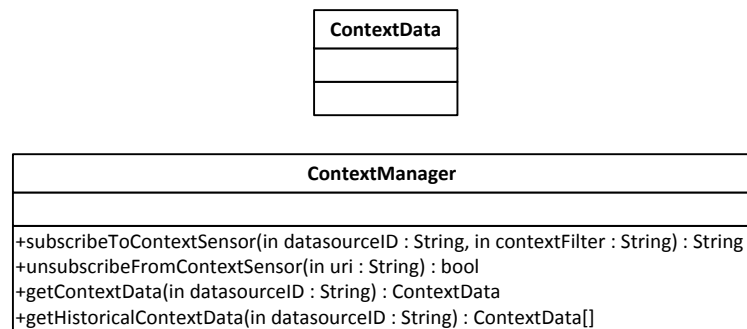


Figure 21: Context-based Service Personalization Class Diagram

6.3.5.1 RESTful Interfaces

The following RESTful interfaces wrap the functionality of their Java interface counterparts described in Section 6.3.5.2.

6.3.5.1.1 Get Subscription Endpoint

Through this interfaces, it is possible to send a request to the Context Manager in order to open a channel for a particular Context Sensor with given filter attributes (the JSON schema description for filters can be found in Section 6.3.6.2). However, as those criteria are highly dependent on the underlying use case, it is very likely that the proposed JSON schema may be extended in order to accept additional different filter criteria (e.g., support of radius filter, GPS coordinates ...).

The return value of this call is an instance of the URI JSON schema in Section 6.3.6.2. This return value specifies two fields, namely a URI given the endpoint address of the subscription channel and a type. The type tells the subscriber how the actual subscription has to be done. Depending on the used libraries on subscriber side, the subscriber is able to connect/subscribe to the channel. An example for a Java-based subscription can be found in Section 6.3.5.2.1. It makes use of the from SIMPLI-CITY recommended Java library Apache Camel⁷⁷.

⁷⁷ <http://camel.apache.org/>

Table 50: RESTful Interface Description –Get Subscription Endpoint

Method	GET	URL	\$API_ROOT/getSubscriptionEndpoint?:id			
Description	Requests a subscription endpoint for a particular Context Sensor					
Parameter	id	Required	yes	Possible Values	128 bit UUID	Description Unique ID of the requested sensor
JSON Object	http://Simpli-City.eu/ContextSensor/JSON-Schema/Filter					
JSON Attribute	upperthreshold	Required	no	Possible Values	Double	Description An upper threshold which has to be exceeded before the subscriber gets informed
JSON Attribute	lowerthreshold	Required	no	Possible Values	Double	Description An lower threshold which has to be exceeded before the subscriber gets informed
JSON Attribute	interval	Required	no	Possible Values	Long	Description An interval which defines how often new data should be send the subscriber
JSON Attribute	start	Required	no	Possible Values	Date	Description Defines the start date of the subscription
JSON Attribute	end	Required	no	Possible Values	Date	Description Defines the expiration date of the subscription
Example URL	\$API_ROOT/getSubscriptionEndpoint?id=528739A0-F508-4551-A12A-04A9B51718D0					
Response	HTTP status code + JSON object					
HTTP Status Code		Required	yes	Possible Values	200 204 404	Description Value returned No Value available ID not found
JSON Object	http://simpli-city.eu/ContextSensor/JSON-Schema/Subscription					
JSON Attribute	URI	Required	yes	Possible Values	any URI	Description the address for the socket
JSON Attribute	type	Required	yes	Possible Values	"websocket" "JMS"	Description the type, either a websocket or a JMS channel was opened
Example Response	HTTP/1.1 200 OK					

Listing 158 shows an example JSON message necessary to get the subscription endpoint of a Context Sensor. Importantly; only one websocket or JMS messaging queue at a time will be created for a particular service.

Listing 158: JSON Example: Get Subscription Endpoint (Input Message)

```
{
  "id": "528739A0-F508-4551-A12A-04A9B51718D0",
  "upperthreshold": "80",
  "lowerthreshold": "10",
  "interval": "60000",
  "start": "2014-07-04T12:08:00",
  "end": "2014-08-04T12:08:00"
}
```

Listing 159: JSON Example: Get Subscription Endpoint (Response Message)

```
{
  "URI": "websocket://simpli-city.eu:8080/sensor",
  "connectionType": "websocket"
}
```


6.3.5.1.2 Close Subscription Endpoint

This REST call sends a notification to the publisher that a particular channel is not needed anymore by the subscriber, and can therefore be closed. However, the channel will only be closed, if no other service is subscribed to it. In addition to this, the subscriber is responsible for closing all connections which remain open on his side. An example implementation for Java is presented in Listing 165.

Table 51: RESTful Interface Description – Close Subscription Endpoint

Method	GET	URL	\$API_ROOT/closeSubscriptionEndpoint?uri			
Description	Notify the publisher that a particular endpoint is no longer needed and can be closed					
Parameter	uri	Required	yes	Possible Values	URI	Description
						Unique URI for a particular channel
Example URL	\$API_ROOT/closeSubscriptionEndpoint?uri=websocket://simpli-city.eu:8080/sensor					
Response	HTTP status code only					
HTTP Status Code		Required	yes	Possible Values	200 404	Description
						Successfully unsubscribed URI not found.
Example Response	HTTP/1.1 200 OK					

While Listing 160 shows an example JSON message necessary to notify the publisher that a particular channel is not needed anymore, Listing 161 shows the HTTP response value. Importantly, the message channel will be closed only in the case, if there is no other subscriber left listening on that channel.

Listing 160: Example URI: Close Subscription Endpoint

```
$API_ROOT/closeSubscriptionEndpoint?uri=websocket://simpli-city.eu:8080/sensor
```

Listing 161: Close Subscription Endpoint (Response Message)

```
HTTP/1.1 200 OK
```

6.3.5.1.3 Request Context Data and Request Historical Context Data

This REST call sends a request for retrieving context data from a particular Context Sensor which is defined by its ID.

Table 52: RESTful Interface Description – Requesting Context Data

Method	GET	URL	\$API_ROOT/getcontextdata?id=:historical				
Description	Returns the sensor information of the sensor with the respective ID						
Parameter	id	Required	yes	Possible Values	128 bit UUID	Description	Unique ID of the requested sensor
Parameter	historical	Required	yes	Possible Values	boolean	Description	Defines whether historical or up-to-date data shall be returned
Example URL	\$API_ROOT/getcontextdata?id=528739A0-F508-4551-A12A-04A9B51718D0&historical=false						
Response	HTTP header + text/JSON						
HTTP Status Code		Required	yes	Possible Values	200	Description	Value returned
					204		No value available
					404		ID not found
JSON Object	The context data for a specific context sensor. Note: This is not following any particular data model, since any data format foreseen in the SIMPLI-CITY data model could be used for context data.						
Example Response	HTTP/1.1 200 OK						

While Listing 162 shows an example JSON message necessary to request context data from a particular Context Sensor, Listing 163 shows an example response message. Please note that the format of the context data actually depends on the data source which is observed by the Context Sensor. Hence, the data format could be of any data format foreseen in the SIMPLI-CITY data model (see deliverable D4.1.1 and Section 5 of the document at hand).

Listing 162: JSON Example: Request Data from Context Sensor (HTTP Request Message)

```
$API_ROOT/getcontextdata?id=528739A0-F508-4551-A12A-04A9B51718D0&historical=false
```

Listing 163: JSON Example: Request Data from Context Sensor (Response Message)

```
{
  "name": "volume",
  "value": "100",
  "unit": "liter",
  "accuracy": {
    "unit": "percent",
    "value": 100
  }
}
```

6.3.5.2 Java Interfaces

6.3.5.2.1 Get Subscription Endpoint

The Context-based Service Personalization component provides methods to request a subscription endpoint for a particular Context Data source. This means, if a new requests comes in, a new channel will be created and the endpoint will be returned to the requester. In addition to the endpoint address, its type is returned, which tells the requester how it can subscribe to the endpoint. An example implementation making use of Apache Camel can be found below the interface description in Listing 165.

Parameters:

- **datasourceID:** Defines the unique ID for the Data Source for which the requester will subscribe himself
- **contextFilter:** Is an optional parameter, which defines a filter which should be applied on the context data. Examples are the frequency of wished updates or a threshold which has to be exceeded before the new data is pushed. The Java class format of such a ContextFilter object is bound to the JSON schema description which can be found in Section 6.3.6.1, i.e., it contains the same fields as are defined within the JSON schema.

Return Value:

If the endpoint generation was successfully, a Subscription object (see Listing 171) is returned holding the URI and the channel type, i.e., information if the channel is either a websocket or a JMS topic/queue.

Error Handling:

If the datasource was not found, a DataSourceNotFoundException will be thrown.

Remarks:

Whenever an endpoint generation request arrives, the Context Manager verifies if already a channel with this filter is open. If this is not the case, a new channel will be created. However, in both cases, the corresponding URI will be returned.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 258 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

The actual messaging protocol depends on the requesting component; however, for the Context Manager this is not important since it makes use Apache Camel which unifies several different messaging protocols as described in Section 6.3.2.4. Further, an example for subscribing to such a channel on the client side can be found in Listing 165.

Listing 164: Method Signature – Get Subscription Endpoint

```
/**
 * @param datasourceID, unique ID for the Data Source for which the requester
 *           will subscribe
 * @param contextFilter, Is an optional parameter, which defines a filter which
 *           should be applied on the context data, examples are the
 *           frequency of wished updates or a threshold which has to be
 *           exceeded before the new data is pushed.
 * @throws DataSourceNotFoundException in case the datasource ID is invalid
 * @return on success a Subscription object holding the URI and its type
 */
public Subscription getSubscriptionEndpoint(String datasourceID, ContextFilter
contextFilter) throws DataSourceNotFoundException
...
//Requests an URI for a sensor with the id DATA_SOURCE_TEST_ID and the predefined
filter ContextFilters.HOURLY_UPDATE
Subscription uri = api.getSubscriptionEndpoint(DATA_SOURCE_TEST_ID,
ContextFilters.HOURLY_UPDATE);
```

Listing 165: Example Method – Subscribe to and Unsubscribe from an Endpoint⁷⁸

```

private void subscribe(Subscription subscription) throws
DataSourceNotFoundException {
    // create the Camel context that is the "heart" of Camel
    camel = new DefaultCamelContext();
    // Grab the endpoint where we should consume.
    Endpoint endpoint = camel.getEndpoint(subscription.getURI());

    // create the event driven consumer
    // the Processor is the code what should happen when there is an event
    Consumer consumer = endpoint.createConsumer(new Processor() {
        public void process(Exchange exchange) throws Exception {
            // get the body as a String
            String body = exchange.getIn().getBody(String.class);

            // simply print it to the console
            System.out.println("received message body" + body);
        }
    });

    // start the consumer, it will listen for new messages
    consumer.start();
}

private void unsubscribe(Consumer consumer){
    // stop the consumer, it will not listen anymore for new messages
    consumer.stop();
}

```

6.3.5.2.2 Close Subscription Endpoint

This Java interface provides the functionality of telling the publisher that a particular endpoint is not needed anymore and can therefore be closed. However, while this method call only tells the publisher that the channel is not needed anymore, the subscriber has to manage its own opened resources locally, i.e., close potential topics or message queues (see Listing 165).

Parameters:

- URI: The URI defines a particular message transport channel which can be closed as long as no other service is listening on it.

Return Value:

True will be returned if closing the subscription endpoint was successful. Notably, false will be returned if there are any other subscribers to this particular endpoint.

Error Handling:

No special error handling is required in this case as the whole method call is optional. If closing a channel fails on the server side, the error will be solved also on the server side, thus the subscriber will not notice anything

⁷⁸ Adapted from: <http://camel.apache.org/tutorial-example-reportincident-part3.html>

Remarks:

In order to save resources, it is possible to have several services listening on one particular messaging channel. This has to be considered when a service tries to unregister itself.

Listing 166: Method Signature – Close Subscription Endpoint

```
/**
 * @param URI, true will be returned if the un-subscription was successfully
 * @return true on success or false otherwise
 */
public boolean closeSubscriptionEndpoint(String URI)

...

//Unsubscribes from the sensor "http://simpli-city.eu/sensor/fuel"
Boolean success = api.closeSubscriptionEndpoint("websocket://simpli-
city.eu/sensor/fuel");
```

6.3.5.2.3 Get Context Data and Get Historical Context Data

This Java interface provides the means of returning the whole historical data or the up-to-date data from a particular context sensor.

Parameters:

- **dataSourceID:** Defines the unique ID for the Data Source for which the requester will subscribe

Return Value:

- **ContextData:** Returns the up-to-date context data from the defined Context Sensor. Note that the ContextData is *not* indicating a particular data format, since any data format foreseen in the SIMPLI-CITY data model could be used for context data.
- **ContextData[]:** In case of `getHistoricalContextData`, an array of historical context data from a particular Context Sensor will be returned

Error Handling:

In case of an invalid `dataSourceID`, a `DataSourceNotFoundException` will be thrown.

Remarks:

Beside of the publish-subscribe interfaces, the Context Manager offers an interface for retrieving ad-hoc context data from a particular sensor or historical data from the database. Please note that the format of the context data actually depends on the data source which is observed by the Context Sensor. Hence, the data format could be of any data format foreseen in the SIMPLI-CITY Data Model (see deliverable D4.1.1 and Section 5 of the document at hand).

Listing 167: Method Signature – Get Context Data

```

/**
 * @param datasourceID, defines the Data Source for which the context data shall be
 * returned
 * @return a list of context data or a single context data entry
 * @throws DataSourceNotFoundException in case the datasource ID is invalid
 */
public ContextData getContextData(datasourceID) throws
DataSourceNotFoundException

public ContextData[] getHistoricalContextData(datasourceID) throws
DataSourceNotFoundException

//Get the up-to-date Context Data from a Sensor with the ID
//DataSensors.FUEL_SENSOR_ID
ContextData result = api.getContextData(DataSensors.FUEL_SENSOR_ID);
//Get historical Context Data from a Sensor with the ID
//DataSensors.FUEL_SENSOR_ID
ContextData[] results = api.getHistoricalContextData(DataSensors.FUEL_SENSOR_ID);

```

6.3.5.2.4 Location-based Data Service Selection

This method is needed in order to provide a loose coupling between apps and location-based services, e.g., a parking slot finder app should only provide information about a free parking slot within the current area. This functionality is only provided through a Java interface, since only services running with the Service Runtime Environment will be allowed to use it.

Parameters:

- location: Defines the location for which a data service should be selected. The location class is defined in Section 6.3.6.4.
- category: Defines the category for the wanted service, e.g., parking service, POI service.

Return Value:

The service description of the best fitting service will be returned.

Error Handling:

In case of an invalid location as parameter, an InvalidLocationException will be thrown. Further, if no service was found within this location area, a NoServiceFoundException will be thrown.

Remarks:

Beside of retrieving context data from the Context-based Service Personalisation component, an additional interface is provided which accepts location information as a parameter and returns the most suitable service in a defined category.

Listing 168: Method Signature – Location-based Data Service Selection

```
/**
 * @param location, defines the area of interest for finding a suitable service
 * @param category, defines the category for the wanted service
 * @return a service description for the most suitable service
 * @throws InvalidLocationException if the defined location is invalid
 * @throws NoServiceFoundException if no service was found within the defined area
 */
public ServiceDescription getContextBasedDataService(Location location, String
category) throws InvalidLocationException, NoServiceFoundException

...
Location location = new Location(48.210627, 16.385727, 10);
//
ServiceDescription result = api.getContextData(location, "parkingService");
```

6.3.6 Content Format

6.3.6.1 Context Filter Schema

Since some services may only need a particular part of context information, it is possible to limit the data which is sent over an opened publish-subscribe channel. For that purpose, the Context-based Service Personalization makes use of a context filter which accepts different filtering attributes. The schema for this filter is described in Listing 169.

Listing 169: Context Filter Schema

```
{
  "type": "object",
  "id": "http://Simpli-City.eu/ContextSensor/JSON-Schema/Filter",
  "properties": {
    "contextfilter": {
      "type": "object",
      "required": true,
      "properties": {
        "upper_threshold": {
          "type": "string",
          "required": false
        },
        "lower_threshold": {
          "type": "string",
          "required": false
        },
        "interval": {
          "type": "long",
          "required": false
        },
        "start": {
          "type": "date",
          "required": false
        },
        "end": {
          "type": "date",
          "required": false
        }
      }
    }
  }
}
```

As Listing 169 shows, different filtering criteria are supported. However, those criteria are highly dependent on the use case and may change or be extended in the future (e.g., support of radios filter, GPS coordinates, etc.).

- `upper_threshold`: defines a threshold which means to be exceeded before new data is sent to the requesting service
- `lower_threshold`: if a value is fallen short of this lower threshold, the new data is sent to the requesting service
- `interval`: defines the interval of how often data should be updated. This value is defined in milliseconds. If 0, the update is only triggered once, e.g., either on the start date, or if not defined, immediately.
- `start`: defines a start date when the first update should be triggered

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 264 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

- end: defines the end date, i.e., the moment when the last update should be triggered

6.3.6.2 Subscription JSON Schema

Listing 171 represents the JSON schema which is needed for subscribing to a subscription endpoint, i.e., the response if using the interface described in Section 6.3.5.1.1. The response value gives information about the channel via which the new context data will be pushed. The according Java class is defined in the next subsection.

Listing 170: JSON Schema – Subscription Result Value

```
{
  "type": "object",
  "id": "http://Simpli-City.eu/ContextSensor/JSON-Schema/Subscription",
  "properties": {
    "URI": {
      "type": "string",
      "required": true
    },
    "type": {
      "required": true,
      "value": {
        "enum": [
          "websocket",
          "JMS"
        ]
      }
    }
  }
}
```

6.3.6.3 Subscription Java Class

Listing 171 represents the corresponding Java class to the JSON schema which is needed for subscribing to a subscription endpoint. The response value (for the interface provided in Section 6.3.5.2.1) gives information about the endpoint of the channel via which the new context data will be pushed.

Listing 171: Java Class Subscription

```
public class Subscription {

    /**
     * Endpoint URI
     */
    private String uri;

    /**
     * Endpoint type
     */
    private String type;

    //... getter() and setter()
}
```

6.3.6.4 Location Class Definition

The class Location represents either a particular point or an area. The size of the area is defined by the field radius. It creates a circle with the centre latitude and longitude.

Listing 172: Java Class Location

```
public class Location {

    /**
     * latitude value for a location
     */
    private double latitude;

    /**
     * longitude value for a location
     */
    private double longitude;

    /**
     * radius for the distance of the
     */
    private double radius;

    public Location(double latitude, double longitude, double radius) {
        this.latitude=latitude;
        this.longitude=longitude;
        this.radius=radius;
    }
    //... getter() and setter()
}
```

6.3.7 Summary

The Context-based Service Personalisation component provides the means to deliver relevant context data to services in order to personalize their outcome to fit for a particular user. These context data are retrieved from so called Context Sensors which are connected to different data sources including the Cloud-based Information Infrastructure, different sensors such as physical, proximity and GPS sensors, Open Data services, or the Data Processing component.

The core component of the Context-based Service Personalisation is the Context Reasoner, which is able to read context data from the Context Database (where historical context data is stored) and reason on it in order to trigger a particular action such as sending a notification to a backend service or to the Media Data Streams and Data Prefetching Logic component (on request).

For notifying different services, this component makes use of a publish-subscribe solution. In the technology comparison, it was decided to make use of Apache Camel, an open source publish-subscribe middleware. It supports several different messaging channels as underlying transport protocol. This kind of middleware enables backend services to subscribe to a particular data source, in order to be notified whenever new, relevant context data is available.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 266 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

6.4 Service Registry

6.4.1 Major Design Decisions

The Service Registry stores service software artifacts along with further information about the services. For internal backend services, i.e., services which will be deployed within the SIMPLI-CITY Service Runtime Environment, full service artifacts need to be stored, while for data services and external backend services, so-called Proxy Service artifacts (see Section 6.1) will be stored. In addition, descriptive (meta-)data about services and proxies needs to be persisted. From a technical perspective, Proxy Services and service artifacts provide the same content; however, regarding the descriptive data, different data is used for different kinds of services (for the data models, see Section 9.1). Apart from the actual storage functionality, the Service Registry offers the Create, Read, Update, Delete (CRUD) functionalities necessary in order to manipulate data. Furthermore, it provides the possibility to store and discover information about running instances of services – this functionality is needed for service interoperability and for system administration.

The Service Registry is an important helper component, but not in the immediate focus of the SIMPLI-CITY project. A lot of sophisticated service registry research approaches and software implementations have been presented in the last years (see Section 6.4.2.2). As such, the project will examine existing solutions as well as the usage of software components already foreseen within SIMPLI-CITY, instead of building a completely new software solution.

During the discussion of the Global Architecture (deliverable D3.1) and Functional Specification (deliverable D3.2.1) discussions, the following major design decisions have been made:

Service Registry and Repository:

The SIMPLI-CITY Service Registry has to provide the functionalities of both a registry and a repository: While a pure registry is a service catalogue, i.e., providing information about services, but not the services themselves, a repository also includes the actual service software artifacts. The latter is necessary in SIMPLI-CITY in order to allow the deployment of new service instances during the runtime of the system.

Integrated Approach:

As stated above, the SIMPLI-CITY Service Registry needs to be able to store different types of service artifacts for internal/external backend services, data services and differing information about these service types. If a service registry cannot differentiate between different service types and according data models, it would be necessary to offer three distinct registries.

Furthermore, in order to provide an holistic software component for the different SIMPLI-CITY components, the Service Registry needs to provide an extensible data model. Different components will request different data from the Service Registry. For instance, the Service Runtime Environment will request the actual software artifacts and technical data about services, the Service Marketplace will request marketing-related data, and the Monitoring component will request Service Level Agreement (SLA) data.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 267 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

Lightweight Solution:

Since the Service Registry is a helper component, it should not be heavyweight. A lot of the existing solutions provide functionalities that will not be used in SIMPLI-CITY but nevertheless complicate the understanding of the system. Therefore, the SIMPLI-CITY Service Registry should provide a solution without too much overhead which still covers the necessary functionalities as defined in the Functional Specification.

Integration:

As a helper component, the Service Registry should rather be capable to be integrated into a system landscape without too much effort; hence, it needs to be technologically compatible with the general SIMPLI-CITY system (see Section 3) and especially the OSGi-based Service Runtime Environment (see Section 6.1). It should not introduce redundant components into the system landscape, e.g., for user management.

Scalability:

In theory, SIMPLI-CITY could be used to host many different services. These services could be used concurrently by many apps, which makes it necessary to provide different instances of the same service at the same time. Since the Service Registry is used to discover running instances of internal backend services whenever an app wants to make use of a service, it needs to be able to serve a large number of requests at the same time, i.e., provide runtime scalability. Also, storage scalability is needed in order to make sure that a large number of service artifacts – which may be available in different versions – may be stored in the Service Registry.

6.4.2 Technology Comparison

6.4.2.1 Comparison Criteria

Table 53: Criteria for Technical Specification

Parameter	Importance	Description
GUI	-	The data stored in the Service Registry will only be indirectly accessed by software developers through the Service Marketplace or the Service Development API. Hence, there is no need for a GUI.
Extensible Data Model	++	Different SIMPLI-CITY components will need to store and read different information about services. Furthermore, information about different types of services (internal/external backend services, data services) needs to be stored. Hence, the SIMPLI-CITY Service Registry needs to provide an extensible data model.
SLA Support	+	SLAs are needed by the Monitoring component and therefore need to be provided by the Service Registry. However, this could be also achieved indirectly through an extensible data model, i.e., there is no need for direct SLA support by the underlying technology.

Registry AND Repository	++	The SIMPLI-CITY Service Registry needs to store both information about services and the according software artifacts. Hence, both registry and repository functionalities need to be provided.
Various Possibilities Regarding File Storage	++	In order to provide a lightweight solution, the SIMPLI-CITY Service Registry should reuse an already foreseen file storage instead of having specific requirements.
Various Possibilities Regarding Database	++	In order to provide a lightweight solution, the SIMPLI-CITY Service Registry should reuse an already foreseen database instead of having specific requirements.
User Management	+	The access to services needs to be restricted through some user management, i.e., in order to make sure that only authorized personnel can access, change, and submit services. However, this functionality could also be provided through integrating user management facilities from some other component. In the latter case, such facilities need to be integrable into the SIMPLI-CITY Service Registry.
Version Control	+	For internal backend services, it may make sense to provide different versions of the same service at the same time. For instance, this could be necessary in order to allow for apps to continue functioning even if they will not be able to work with a newer version of a service. In any case, it should be possible to update services; hence, some version control functionality is needed.

6.4.2.2 Possible Technologies and Comparison

Service registry software products can be distinguished into software that implements a particular standard, and research prototypes, which either provide a new approach or implement and extend existing standards. Furthermore, some frameworks and (OSGi-based) service runtime environments provide selected registry functionalities, which mainly cover these functionalities needed during system runtime, e.g., for service lookup.

Currently, the two most important service registry standards are the (outdated) Universal Description, Discovery and Integration (UDDI) standard, and the ebXML Registry standard. Since 2007, the current UDDI version 3.0⁷⁹ is available; however, there has been no further development of the standard since then and there is only little progress in software adopting the standard. In contrast, the OASIS ebXML RegRep Version 4.0⁸⁰ has been released at the beginning of 2012. As the name implies, the latter covers both registry and repository functionalities, while UDDI is limited to registry functionalities. More recently, OASIS has also started to standardise the SOA Repository Artifact Model and Protocol (S-RAMP) version 1.0⁸¹, which also offers both registry and repository functionalities. However, only a limited number of implementations of the S-RAMP standard are available so far. Last but not least, the OSGi service registry also allows registering services at a shared catalogue, but is primarily aiming at runtime functionalities.

⁷⁹ <https://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm#uddiv3>

⁸⁰ <http://docs.oasis-open.org/regrep/regrep-core/v4.0/os/regrep-core-overview-v4.0-os.html>

⁸¹ <http://docs.oasis-open.org/s-ramp/s-ramp/v1.0/s-ramp-v1.0-part1-foundation.html>

jUDDI and OpenUDDI:

jUDDI⁸² and OpenUDDI⁸³ are two open source implementations of the UDDI version 3.0 standard, which provides the current state of the art for UDDI. jUDDI is an official Apache project and therefore backed by a large community, while there have been no updates for OpenUDDI since 2010. In general, jUDDI provides more possibilities to extend the actual functionality and is also more flexibly integrable into an existing system landscape, which is important for SIMPLI-CITY. Nevertheless, both UDDI software products are integrable into a Tomcat-based service landscape. Both implementations do not explicitly provide the possibility to store SLA data; however, this could be achieved through extending the underlying service data model. Both software products allow the definition of users using a rather simple role model; OpenUDDI allows making use of LDAP for this purpose, while jUDDI expects the integration of an external authentication mechanism. Since the UDDI standard does not explicitly incorporate the means to describe different versions of the same service, both jUDDI and OpenUDDI do not support versioning.

Further UDDI implementations like UDDI4J⁸⁴ or Scoud⁸⁵ have not been regarded because there has been no activity on these projects since 2006 and 2004, respectively. Both projects only implement the out-dated UDDI version 2.0. Also, there are some UDDI extensions for Semantic Web Services (SWS), e.g., the FUSION Semantic Registry Architecture [KP08], which is in turn based on UDDI4J. However, since SIMPLI-CITY does not make use of a SWS standard, such solutions are not interesting to the project.

freebXML:

freebXML is the outcome of the OASIS ebXML Registry reference implementation project and the only open source implementation of this standard. However, there has been no progress since 2007 and it is unlikely that the abovementioned OASIS RegRep Version 4.0 will be implemented in time for SIMPLI-CITY. Hence, the current freebXML version 3.1, which implements the OASIS ebXML Registry 3.0 specification, will be discussed here. In general, the main drawback of this version is the low extensibility of some core packages. If compared to the UDDI implementations discussed above, freebXML provides more sophisticated features and also a combined registry/repository solution.

JBoss Overlord S-RAMP:

JBoss Overlord's S-RAMP⁸⁶ distribution (version 0.3.0, final) is an implementation of the S-RAMP standard and therefore provides a more recent registry/repository standard than the software products discussed so far. It is designed for usage with the JBoss Enterprise Application Platform, but provides open sources and could therefore be customized for usage in other system landscapes (based on an Atom REST binding). Furthermore, Maven is explicitly supported. Because S-RAMP is a more recent standard than UDDI and ebXML Registry, it supports SLAs (through WS-Policy), an extensible data model, and semantic classification through usage of the Web Ontology Language (OWL). Furthermore, a sophisticated query language is provided. However, versioning of

⁸² <http://juddi.apache.org/>

⁸³ <http://sourceforge.net/projects/openuddi/?source=directory>

⁸⁴ <http://uddi4j.sourceforge.net/>

⁸⁵ <http://sourceforge.net/projects/scoud/>

⁸⁶ <http://www.jboss.org/overlord/downloads/sramp>

services/artifacts is not provided. Modeshape⁸⁷ is used to store data, allowing a number of different database systems to be used. Some basic HTTP authentication is provided through the implementation of the abovementioned Atom binding. However, there is no sophisticated authentication or security concept.

OSGi Service Registry – Eclipse Equinox and Apache Felix:

Both Apache Felix and Eclipse Equinox (see Section 6.1) provide an implementation of the OSGi service registry standard. The OSGi service registry provides certain registry functionalities, which primarily aim at the system runtime, e.g., in order to lookup running services. Per se, the OSGi service registry is limited to describe services (typically a Java class or interface) in order to find them during runtime. However, the service descriptions could also be used for further purposes. Based on the actual OSGi implementation SIMPLI-CITY will make use of (see Section 6.1), it might nevertheless be necessary to provide some further data storage for marketplace-related data. Naturally, one particular benefit of an OSGi service registry is its integration into the actual Service Runtime Environment. Therefore, it would offer a very lightweight solution. Last but not least, the OSGi framework provides an enhanced versioning mechanism⁸⁸, which is supported by the OSGi service registry.

Bucket-based Solution:

While not explicitly providing any registry or repository functionalities, a solution based on the internal data storage of SIMPLI-CITY, i.e., the Cloud-based Information Infrastructure (see Section 5.2) could also provide the foundation for the SIMPLI-CITY Service Registry, since it meets important criteria, i.e., the provision to store any data, the possibility to choose between different file storages and database types, role management for authentication, and an extensible data model. For this, it would be necessary to define a bucket which is able to store data about services as well as the service artifacts themselves. One particular benefit of such a solution would be the stability and scalability of the data storage; furthermore, the system is well-known to the project participants. However, a bucket-based solution provides only very lightweight functionality, i.e., all actual service management functionalities as well as the runtime functionalities (most importantly: service lookup) would have to be covered by other components.

⁸⁷ <http://www.jboss.org/modeshape>

⁸⁸ <http://www.osgi.org/wiki/uploads/Links/SemanticVersioning.pdf>

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 271 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

Table 54: Comparison of Technologies for Service Registry

Parameter	Importance	jUDDI	OpenUDDI	freebXML	JBoss Overlord (S-RAMP)	Eclipse Equinox Registry (OSGi)	Apache Felix Registry (OSGi)	Bucket-based Solution
Generic Criteria								
Up-to-Datedness	→	6	4	4	8	10	8	10
Stability	+	8	6	8	6	8	8	10
Extensibility & Open Source/Standards	+	8	5	5	8	10	9	10
Familiarity	+	5	2	5	1	8	10	10
Performance	++	5	7	5	5	8	8	10
Interoperability	+	9	5	6	6	9	9	10
License		Apache 2.0	Apache 2.0	Proprietary Open Source License	Apache 2.0	Eclipse Public License	Apache 2.0	Does Not Apply
Specific Criteria								
GUI	-	4	4	6	6	10	10	0
SLA Support	++	2	2	5	9	8	8	0
Registry AND Repository	++	0	0	10	10	0	10	10
Various Possibilities Regarding File Storage	→	Not applicable	Not applicable	8	10	10	10	10
Various Possibilities Regarding Database	→	10	10	8	8	10	10	10
Role Management	+	2	2	8	2	10	10	10
Version Control	+	0	0	8	0	10	10	6
Extensible Data Model	++	6	6	8	8	4	4	10

6.4.3 Technology Selection

6.4.3.1 Selection for Service Data Storage

During the technology comparison, it became clear that the UDDI, ebXML, and S-RAMP standards provide a lot of functionalities which are actually not needed within SIMPLI-CITY. Furthermore, there are some doubts regarding the extensibility and customization of existing UDDI and ebXML software products. JBoss Overlord (S-RAMP) offers a more up-to-date approach, but also provides a lot of overhead not necessary in SIMPLI-CITY. None of these solutions provides runtime functionalities. However, JBoss Overlord could be integrated with the JBoss OSGi framework in order to deliver this. In fact, this would be a suitable solution for SIMPLI-CITY. Similar runtime functionalities are provided by both the Eclipse Equinox and Apache Felix implementations of the OSGi service registry. However, there might be the problem that not all marketplace-related data can be integrated into such a registry. The bucket-based solution provides a highly stable solution, but unfortunately does not offer any additional registry functionalities.

As a result, it has been decided to primarily make use of an OSGi service registry (based on the selection of a particular OSGi implementation as foundation for the Service Runtime Environment – see Section 6.1). Conditionally, marketplace-related information may be stored in some additional data storage based on the Cloud-based Information Infrastructure (i.e., a specific bucket). The definite decision is strictly related to the final service data model.

As described in the Functional Specification (deliverable D3.2.1, Section 6.4.4), an optional functionality of the Service Registry will be the provision of a Service Watch, i.e., a component which regularly polls data services in order to make sure that data sources are alive. For this, the Monitoring component (see Section 6.2) is used. According data about the status of data services has to be stored in an according database. As described in Section 4.1, such databases are realized in SIMPLI-CITY through a specific bucket in the Cloud-based Information Infrastructure (see Section 5.2). Since the Service Watch is an

optional component, it will only be realized if there are enough resources left for implementing this functionality.

6.4.3.2 Missing Elements and Implementation Needs

The main enhancement needs are in the following areas:

Exposition of Functionalities for Service Management and Service Discovery:

Service management includes the creation, updating, and deletion of service artifacts and descriptions in the Service Registry, i.e., it is a design time functionality. While the OSGi-based service registry allows this, it is necessary to expose these functionalities to the Service Development API, which in turn offers these functionalities to service developers. Hence, a corresponding subcomponent which provides these functionalities to the outside needs to be foreseen. Since OSGi offers the registration and deployment of different service versions at the same time, this functionality should also be exposed.

In the Functional Specification, we defined that the Service Registry has to offer a so-called “Service Lookup” functionality in order to make it possible for components like the Context-based Service Personalisation and Service Marketplace to find services. It was also foreseen to implement this functionality for the Service Runtime Environment, i.e., so that the Service Runtime Environment would be able to find running services during system runtime and also deploy services. However, this functionality is already covered by the OSGi framework service registry and does not have to be implemented separately.

Hence, service discovery in SIMPLI-CITY will be a design time functionality that other components can make use of in order to discover services and get information about them. This functionality can be covered as part of the service management, as it is a prerequisite for the latter anyway.

Notably, service management also includes the management of marketplace-related data. As defined above, such data might make it necessary to store additional data in a bucket-based data storage. If this is the case, service management will also have to make sure that this functionality is exposed to other components.

License Checking:

License checking is a functionality which is actually provided by the Service Marketplace. However, the SIMPLI-CITY Service Registry needs to be able to provide the license information to the Service Marketplace when requested. This is a specialization of the general service discovery functionality, as based on a particular service ID, general license information is handed back to the Service Marketplace.

Furthermore, the Service Registry needs to be able to store data about valid licenses (i.e., that a particular app or user has a valid license for a particular backend service) if the Service Marketplace requests this.

Notably, service licenses within SIMPLI-CITY may be given either to particular apps or particular developers/companies. As both of them are identified using a UUID, it makes no difference from the perspective of the Service Registry is a license has been granted for an app or a developer. However, this needs to be regarded when license checking is invoked by the Service Runtime Environment.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 273 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

Extensible Service Data Model:

While the service data model itself is not an intrinsic part of the SIMPLI-CITY Service Registry, the Service Registry has to store the SIMPLI-CITY service data model (service Manifest, see Section 9.1). Hence, this data model needs to be serialized into the OSGi format. If necessary, certain parts of the service Manifest, which should (e.g., for reasons of performance) not be stored in the OSGi service registry need to be stored in a separate bucket. For this, the SIMPLI-CITY Service Registry needs to be able to store and administrate the different parts of the SIMPLI-CITY service Manifest and, most importantly, be able to distinguish between these parts to be stored within the bucket and the OSGi service registry.

Service Status Bucket (Optional):

The Service Status Bucket will be used in order to store status-related data about services. This includes data from the Service Watch (see below). As the name implies, the Service Status Bucket will be a bucket as offered by the Cloud-based Information Infrastructure. Hence, there are no particular implementation needs. However, the bucket needs to be designed (e.g., a database schema needs to be defined) and set up.

Service Watch (Optional):

As described in the Functional Specification, the Service Watch functionality regularly polls data services and stores QoS data about them in an according database. This is done in order to be able to estimate if a data source is available or not. While the actual polling of data services can be done through the Monitoring component (see Section 6.2), the logic when to poll, where to store the according QoS data (but not the actual output from data services), and when to delete the QoS data from the database again, will be part of the Service Watch. Since this is not a typical functionality of a service registry, such functionality is completely missing in the OSGi service registry and therefore needs to be implemented from scratch. The Service Status Bucket will be used to store the Service Watch-generated QoS data.

6.4.3.3 Further Information and Conclusion from Technology Comparison

While there are several service registry and service repository standards and implementations available, most of them are either not easy to extend or provide a large functionality overhead. Through the selection of an OSGi-based service framework for the SIMPLI-CITY Service Runtime Environment (see Section 6.1), the choice of the OSGi-based service registry standard (and a corresponding implementation) became obvious for the SIMPLI-CITY Service Registry. As can be seen from the discussion above, the OSGi-based service standard provides a lightweight and fully functional technology, which eases the implementation efforts for this component to a large extent.

Hence, if compared with the technology-agnostic Functional Specification, it was possible to delete some originally foreseen subcomponents of the Service Registry without any loss of functionality. Because the Service Registry is a helper component, this is a very positive side effect of the technology selection.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 274 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

6.4.4 Component Structure

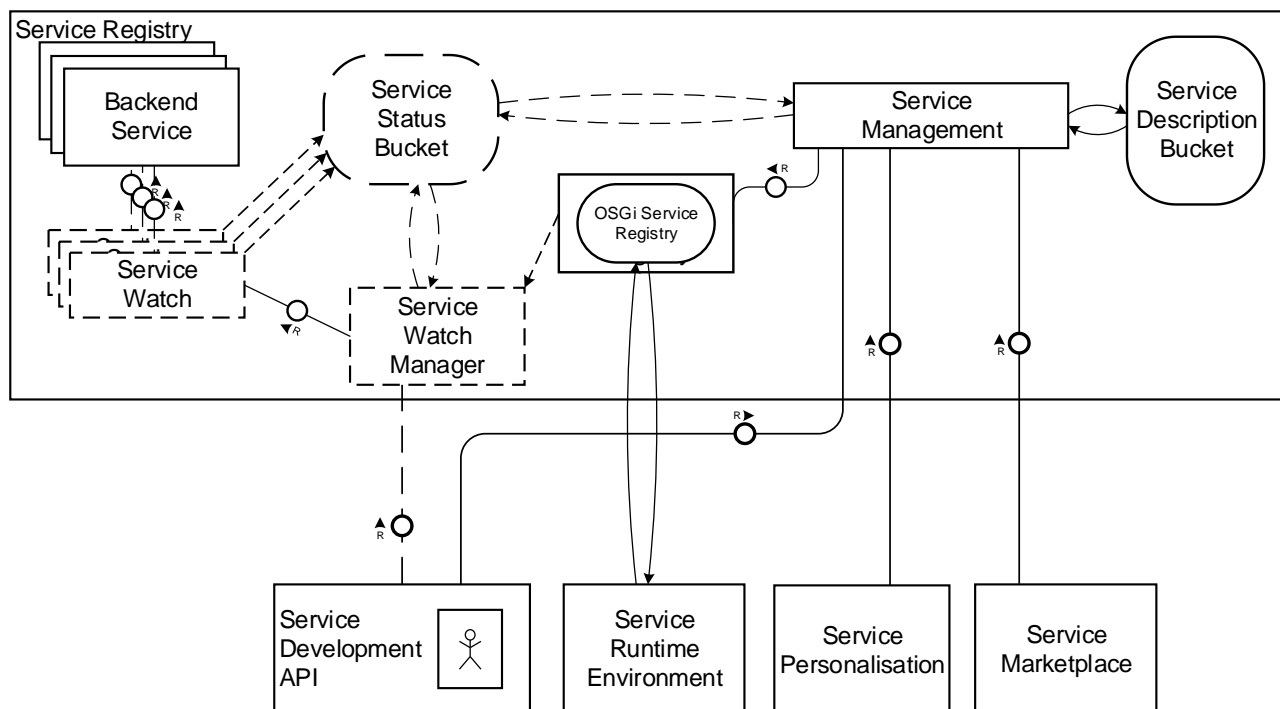


Figure 22: Component Structure of Service Registry

Figure 22 shows the updated component structure of the SIMPLI-CITY Service Registry. When compared to the Functional Specification, it can be seen that the formerly technology-agnostic data storages have now been replaced by the selected technologies, i.e., buckets from the Cloud-based Information Infrastructure for the storage of service status data and Service Marketplace-related data, and the OSGi Service Registry for the actual service registry functionalities. Within the Functional Specification, the Service Status Bucket has been foreseen as database for service statistics; however, it became obvious that this introduces some redundancies, because service statistics are also stored as part of the Monitoring component (see Section 6.2). Hence, this functionality of the Service Status Bucket has been deleted. The Functional Specification was missing a definition which component could setup a watch for a particular data service. This information has now been added, because this can be done through the Service Development API.

While the Functional Specification named a Service Artefact Storage, this is an intrinsic element of the OSGi service registry standard and therefore, this separate data storage is not necessary anymore. Notably, Figure 22 shows the OSGi Service Registry as a data storage combined with a wrapper which offers registry functionalities to external components, i.e., such software not running on the same server or outside of the OSGi-based Service Runtime Environment.

Since the functionalities of the formerly foreseen subcomponents Service Lookup and Service Registration are partially already covered by the OSGi service registry, their scope of operation has been lowered and the subcomponents have been summarised into the (previously not named) Service Management component.

Service Management:

The Service Management subcomponent offers the central API functionalities to all external components, e.g., the methods to create, delete, or update service artifacts and service descriptions within the OSGi Service Registry. It offers the same functionalities to store proxies for data services and external backend services. Hence, it provides a mediation layer between the actual data storage and other software components. In many cases, this subcomponent will expose functionalities provided by the OSGi Service Registry to external components. However, the Service Management subcomponent also controls the storage of different service data aspects in separate data storages (technical data and SLA information in the OSGi Service Registry, potentially Service Marketplace-related data in the optional Service Description Bucket). Last but not least, it provides any external component with the means to access data from the Service Status Bucket.

OSGi Service Registry:

The OSGi Service Registry is the basic subcomponent of the SIMPLI-CITY Service Registry and actually offers all registry and repository functionalities necessary for the operation of services within the SIMPLI-CITY Service Runtime Environment, i.e., the means to store services, look them up for service deployment, and lookup running service instances. For this, the OSGi Service Registry offers an internal data storage (not depicted in Figure 22) as well as the according methods.

Notably, the OSGi Service Registry is an intrinsic part of the Service Runtime Environment, since both are parts of the chosen service framework implementation. Hence, the interactions and message exchange between the Service Runtime Environment and OSGi Service Registry is given by the chosen OSGi implementation-

Service Description Bucket:

The Service Description Bucket is used as a supplemental data storage for service-related data that cannot be stored within the OSGi Service Registry, e.g., screenshots or other figures which are needed within the Service Marketplace. Notably, it depends on the final version of the SIMPLI-CITY service Manifest which data will be stored within this bucket (see Section 9.1). Also, the schema of the bucket depends on the data model and has therefore not been defined yet. As with all other data storages within SIMPLI-CITY, this bucket will use the means provided by the Cloud-based Information Infrastructure. Hence, for its realization, a definition of the schema etc. is necessary, but there are no further implementation needs. For external components, CRUD functionalities of this bucket are exposed through the Service Management subcomponent.

Service Watch, Service Watch Manager, and Service Status Bucket (Optional):

The optional Service Watch Manager and the Service Watch offer the possibility to regularly poll data services in order to make sure if a service is still alive. A watch can be set up for a particular data service through the Service Development API, by naming the data service to be watched, the time interval for the polls, and the storage lifetime of the data from these polls.

This data is stored in the Service Status Bucket and can be accessed by other components through the Service Management API, e.g., in order to provide a service developer with live data about the availability of a particular data source. The Service

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 276 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

Status Bucket is a database realized through the means of the Cloud-based Information Infrastructure.

6.4.5 Interfaces

The SIMPLI-CITY Service Registry provides different methods to store and retrieve service-related data. Per se, it could be requested by any other component, but as can be seen from Figure 22, especially the Service Runtime Environment, the Service Marketplace, Context-based Service Personalization, and the Service Development API make use of it. For this, the Service Registry offers different functionalities, which are provided both via Java interfaces (Section 6.4.5.1) and REST interfaces (Section 6.4.5.2). This includes methods/interfaces to create, update, and delete (CRUD) services and service descriptions, to read/get services, CRUD functionalities for licenses, and the interfaces offered in order to provide make use of the (optional) Service Watch subcomponent.

Figure 23 shows the class diagram for the Service Registry component and its subcomponents. Since the RESTful interfaces are only wrappers for the corresponding Java class, they are not stated in this diagram.

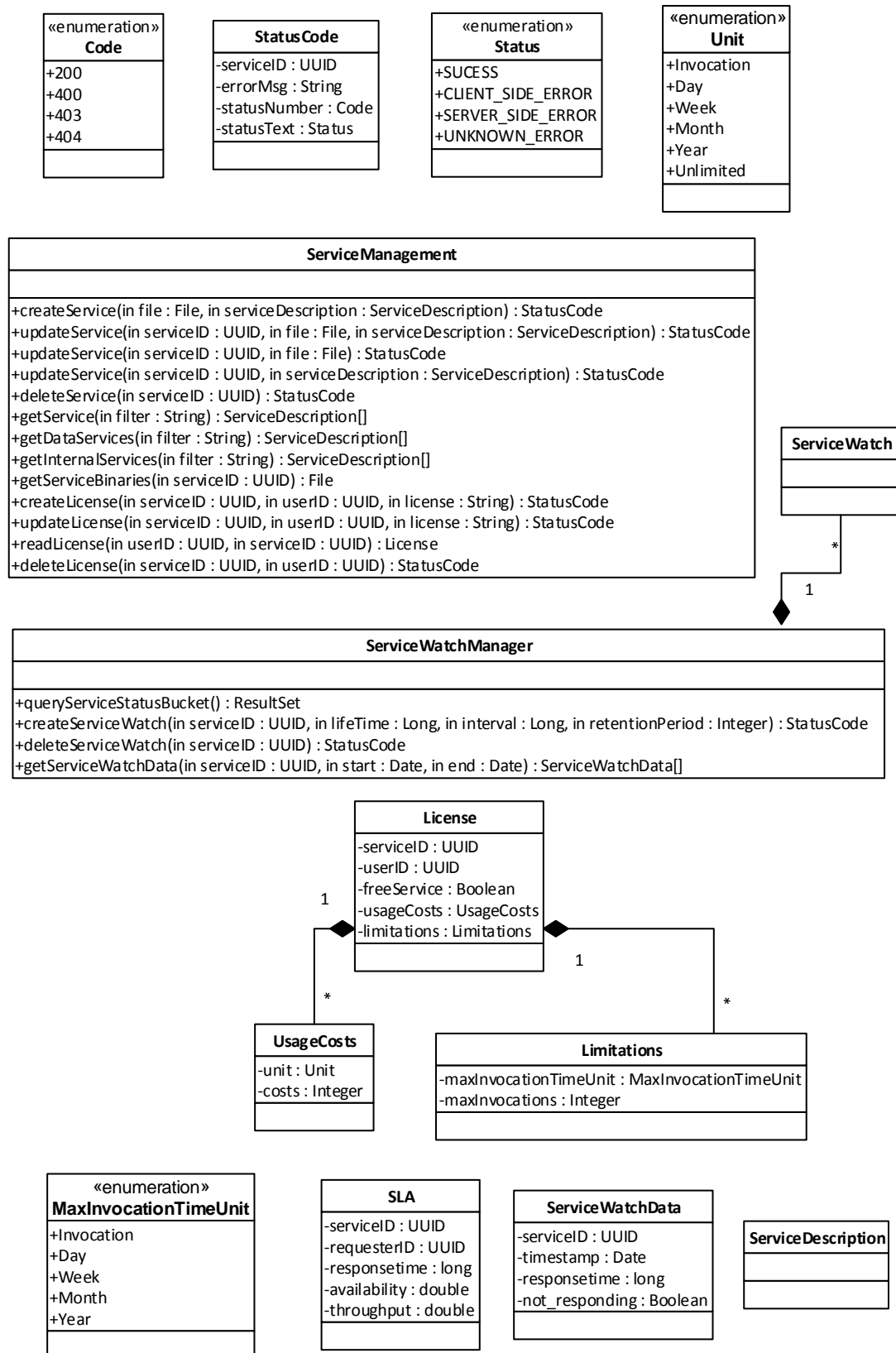


Figure 23: Class Diagram of Service Registry

6.4.5.1 Java Interfaces

6.4.5.1.1 Create Service

This method offers the functionality to create/store a service in the Service Registry. Because of the automated deployment of services in the chosen OSGi-based Service Runtime Environment, services which are stored in the Service Registry are automatically deployed.

Parameters:

- service: File containing the service bundle
- manifest: File (XML) containing the service description (including its license model(s)). See Section 9.1 for the service Manifest model.

Return Value:

Returns a StatusCode object which holds information about the success or failure of the creation, on success the ID is defined within this object. The Java class StatusCode is defined in Listing 217.

Error Handling:

In case of an error, the error cause, e.g., the stacktrace, will be added to the StatusCode object.

Remarks:

Complex parts of the service description, e.g., videos which could be used for advertising a service on the Service Marketplace, have to be manually uploaded to the Service Description Bucket. The service Manifest file will then include the URI of the according video in the bucket. The license models for a particular service are optional and may be added later. As long as no license model has been defined, the usage of a service is unrestricted and free of charge.

Listing 173: Source Code Example – API Method Signature to Create a Service

```
/**
 * @param service, the service which should be created
 * @param manifest, a textual description of the service
 * @return a StatusCode object containing information about the success or failure of
 * this action */
public StatusCode createService(File service, File manifest)

...

//Creates and deploys a new service including its description in form of a Manifest
file
StatusCode status= api.createService(service, manifest);
String serviceID = status.getServiceID();
```

6.4.5.1.2 Update Service

This method offers the functionality to update an existing service in the Service Registry, i.e., to provide a new version of the service or the service description. Notably, information about version compatibility is given in the Manifest file.

Parameters:

- service: File containing the service bundle
- manifest: File containing the service description (including its license model(s)). See Section 9.1 for the service Manifest.
- serviceID: Unique service ID of the service to be updated

Return Value:

Returns a StatusCode object indicating if the service has been updated successfully or not. The Java class StatusCode is defined in Listing 217.

Error Handling:

In case of an error, the error cause, e.g., the stacktrace, will be added to the StatusCode object.

Remarks:

As it can be seen in Listing 174, either the complete service can be updated (by submitting both the service bundle file and the service Manifest file), or only the actual bundle or the description (Manifest) can be changed.

Listing 174: Source Code Example – API Method Signature to Update a Service

```
/**
 * @param service, the service which should be updated
 * @param manifest, a textual description of the service
 * @param serviceID, a unique service ID which shall be updated
 * @return a StatusCode object containing information about the success or failure of
 * this action
 */
public StatusCode updateService(UUID serviceID, File service, File manifest)
public StatusCode updateService(UUID serviceID, File service)
public StatusCode updateService(UUID serviceID, File manifest)

...

//Updates and deploys a new service including its description in form of a manifest
file
StatusCode statusCode = api.updateService(serviceID, service, manifest);

//Updates and deploys a service defined by a ID
StatusCode statusCode = api.updateService(serviceID, service);

//Updates and deploys a service manifest file defined by a ID
StatusCode statusCode = api.updateService(serviceID, manifest);
```


6.4.5.1.3 Delete Service

This method offers the functionality to delete an existing service from the Service Registry.

Parameters:

- `serviceID`: ID of the service to be deleted

Return Value:

Returns a `StatusCode` object indicating if the service has been updated successfully or not. The Java class `StatusCode` is defined in Listing 217.

Error Handling:

In case of an error, the error cause, e.g., the stacktrace, will be added to the `StatusCode` object.

Remarks:

The generic deletion method will delete the complete service as well as the service description. For this, information is deleted from the Service Registry and the Service Description Bucket.

Listing 175: Source Code Example – API Method Signature to Delete a Service

```
/**
 * @param serviceID, a unique ID of the service which shall be deleted
 * @return a StatusCode object containing information about the success or failure of
 * this action
 */
public StatusCode deleteService(UUID serviceID)

...

//Deletes a service including its description in form of a manifest file
StatusCode status = api.deleteService(serviceID);
```

6.4.5.1.4 Get Services Descriptions

Reading service descriptions from the Service Registry is the key functionality of this component. There are different ways to request services, but all have in common that an XML Path Language (XPath) expression needs to be defined.

Parameters:

- `filter`: Filter indicating which services should be given back to the requester. Filters are defined using XPath expressions (as can be found in Listing 211) and therefore have to comply with the service Manifest (see Section 9.1). If no particular filter is given, all services of a particular type will be returned.

Return Value:

An array of XML elements which meet the XPath expression is returned. For a valid output example, see Listing 212.

Error Handling:

In case of an error, a null value is returned. Note: An empty result set does not indicate an error, but merely that there are no services fitting the XPath expression.

Remarks:

As it can be seen in the following listing, different methods are provided in order to retrieve all types of services stored within the Service Registry, or only data services, or only internal backend services. However, the method signature for different service types remains the same.

Listing 176: Source Code Example –
API Method Signature to Retrieve Service Descriptions

```
/**
 * @param xpath, an xpath expression representing a filter indicating which services
 * should be given back to the requester.
 * @return an array of XML elements (service Manifests) matching the provided filter
 */
public ServiceDescription[] getServices(String xpath)
public ServiceDescription[] getDataServices(String xpath)
public ServiceDescription[] getInternalServices(String xpath)
...

String filter = "[type=parkingslotservice]";
//Retrieve a list of services according to the defined filter
ServiceDescription[] services = api.getServices(filter);
ServiceDescription[] services = api.getDataServices(filter);
ServiceDescription[] services = api.getInternalServices(filter);
```

6.4.5.1.5 Get Service Binaries

In some cases, it is necessary to retrieve the binaries of a specific service from the Service Registry. For that, the requester has to pass the wanted service ID to the Service Registry and will retrieve an archive file which includes the according binaries.

Parameters:

- serviceID: a service ID which indicates the service binaries

Return Value:

Returns an archive file which includes the requested service binaries.

Error Handling:

If the defined service was not found, a `ServiceNotFoundException` will be thrown.

Remarks:

Since SIMPLI-CITY uses an OSGi container (Apache Karaf with Felix), the used file extension will be .jar.

Listing 177: Source Code Example – API Method Signature to Retrieve Service Binaries

```
/**
 * @param serviceID, the service ID for which the binaries shall be returned
 * @return an archive file holding the service binaries
 * @throws ServiceNotFoundException if the service was not found
 */
public File getServiceBinaries(UUID serviceID) throws ServiceNotFoundException
...
//Retrieve a list of services according to the defined filter
File serviceBinary = api.getServiceBinaries("528739A0-F508-4551-A12A-04A9B51718D0");
```

6.4.5.1.6 Create License

As described above, the actual license models offered by a service are part of the service description, i.e., the Manifest file. In addition, the Service Registry needs to store the actual granted licenses for single services for single users or apps (see Section 6.5).

Parameters:

- serviceID: Defines the unique ID of the service for which a license should be created
- userID: Defines the unique ID of the user or app for which the license is granted
- license: Defines the license for the particular service. The format of such a license (model) is defined in Section 6.4.6.2.

Return Value:

Returns a StatusCode object indicating if the license has been created successfully or not. The Java class StatusCode is defined in Listing 217.

Error Handling:

In case of an error, the error cause, e.g., the stacktrace, will be added to the StatusCode object.

Remarks:

A license could be granted to a particular user or an app. Both are identified through an according UUID which is stored when a license is granted.

Listing 178: Source Code Example – Create License

```

/**
 * @param serviceID, defines the unique service ID for which a license should
 * be created
 * @param userID, defines the unique user ID for which a license should
 * be created
 * @param license, defines the license for the particular service and user
 * @return a StatusCode object containing information about the success or failure of
 * this action
 */
public StatusCode createLicense(UUID serviceID, UUID userID, String license)
...

//Creates a license for a particular service and user, each defined by their IDs
//
StatusCode status = api.createLicense(servcieID, userID, license);

```

6.4.5.1.7 Read License

This method provides the functionality to read an existing license for a particular service from the Service Registry.

Parameters:

- serviceID: Defines the unique service ID of a service.
- userID: Defines the unique user ID of the user or app for which the licenses for the identified service should be returned.

Return Value:

A license will be returned if it was defined for the specified service and user. Otherwise, null will be returned. The License Java class can be found in Section 6.4.6.2.

Error Handling:

If the defined service was not found, a `ServiceNotFoundException` will be thrown. In addition, if an invalid userID was defined, a `UserNotFoundException` will be thrown. Further, if no license was found for a particular user and service a `LicenseNotFound` exception will be thrown.

Remarks:

This method returns the license which has been granted for a particular service/user combination. If no license has been granted, null will be returned. In this case, the invoking component (e.g., the Payment API of the Service Marketplace – see Section 6.5) should also check if there has been a license model defined at all for a particular service. If this is not the case, the service can be used unrestricted and free of charge.

Listing 179: Source Code Example – Read License

```

/**
 * @param serviceID, defines the unique service ID
 * @param userID, defines the unique user ID for which a license should
 * be returned (in combination with serviceID)
 * @return a license if found, otherwise null
 * @throws ServiceNotFoundException if the service was not found
 * @throws UserNotFoundException if the user was not found
 * @throws LicenseNotFoundException if the license was not found
 */
public License readLicense(UUID serviceID, UUID userID) throws
ServiceNotFoundException, UserNotFoundException, LicenseNotFoundException

...

//Read license for a particular service and user combination
License license = api.readLicense(serviceID, userID);

```

6.4.5.1.8 Update License

This method offers the functionality to update an existing license in the Service Registry.

Parameters:

- serviceID: Defines the unique ID of a service.
- userID: Defines the unique ID of the user or app for which the license for the identified service should be updated.
- license: Defines the updated license for the particular service and user. The License Java class can be found in Section 6.4.6.2.

Return Value:

Returns a StatusCode object indicating if the service has been updated successfully or not. The Java class StatusCode is defined in Listing 217.

Error Handling:

In case of an error, the error cause, e.g., the stacktrace, will be added to the StatusCode object.

Remarks:

Using this method allows a developer to update a license for a particular service. The structure for this license can be seen in Section 6.4.6.2.

Listing 180: Source Code Example – Update License

```
/**
 * @param serviceID, defines the unique service ID for which a license should
 * be updated
 * @param userID, defines the unique user ID for which a license should
 * be updated
 * @param license, defines the updated license for the particular service/user
 combination
 * @return a StatusCode object containing information about the success or failure of
 this action
 */
public StatusCode updateLicense(UUID serviceID, UUID userID, License license)

...

//Updates a license for a particular service defined by the ID
StatusCode status = api.updateLicense (serviceID, userID, license);
```

6.4.5.1.9 Delete License

This method offers the functionality to delete an existing license from the Service Registry.

Parameters:

- serviceID: Defines the unique service ID of a service.
- userID: Defines the unique user ID of the user or app for which the licenses for the identified service should be deleted.

Return Value:

Returns a StatusCode object indicating if the service has been updated successfully or not. The Java class StatusCode is defined in Listing 217.

Error Handling:

In case of an error, the error cause, e.g., the stacktrace, will be added to the StatusCode object.

Remarks:

Deleting a license means complete erasure, i.e., information about licenses will not be stored and is therefore lost. However, the license model will be kept, since it is part of the service Manifest and can only be deleted through altering the service Manifest.

Listing 181: Source Code Example – Delete License

```

/**
 * @param serviceID, defines the unique service ID for which a license should
 * be deleted
 * @param userID, defines the unique user ID for which a license should
 * be deleted
 * @return a StatusCode object containing information about the success or failure of
 this action
 */
public StatusCode deleteLicense(UUID serviceID, UUID userID)

...

//Deletes a license for a particular service/user combination
StatusCode status = api.deleteLicense(serviceID, userID);

```

6.4.5.1.10 Create and Update Service Watch

The (optional) Service Watch is used in order to regularly poll data services. Hence, mechanisms to create, update, and delete according service watches as well as mechanisms to retrieve the recorded data, need to be provided. Notably, data from a Service Watch is per se open to all interested parties, i.e., there is no restriction to access this data. Service Watch data has to be pulled from the subcomponent, i.e., the Service Watch will not actively provide data to other subcomponents.

Parameters:

- serviceID: Defines the ID of the data service which should be watched.
- lifetime: Lifetime of the service watch for the particular service. If no lifetime is defined, it will be automatically set to “0”, which indicates unlimited lifetime, i.e., the particular Service Watch will run until it is deleted.
- interval: Defines the interval (in milliseconds) in which a data service should be polled.
- retentionPeriod: Defines how long data about one particular data poll should be kept in the Service Status Bucket.

Return Value:

Returns a StatusCode object indicating if the service has been updated successfully or not. The Java class StatusCode is defined in Listing 217.

Error Handling:

In case of an error, the error cause, e.g., the stacktrace, will be added to the StatusCode object.

Remarks:

A data service needs to be previously registered (i.e., have received an ID) in order to be capable to be watched. Only one Service Watch per service instance may be created; i.e., the creation of a second Service Watch automatically leads to an update of the existing one. Hence, there is no separate method/interface for updating a Service Watch and a user could overwrite the Service Watch of another user.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 287 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

Listing 182: Source Code Example – Create/Update Service Watch

```

/**
 * @param serviceID, defines the unique service ID for which a Service watch should
 * be created
 * @param lifetime, defines how long the Service Watch will run
 * @param interval, defines in which interval the data service will be polled
 * @param retentionPeriod, defines the period of safekeeping of the Service Watch
 data
 * @return a StatusCode object containing information about the success or failure of
 this action
 */
public StatusCode createServiceWatch(UUID serviceID, long lifetime, long interval,
int retentionPeriod)

...

//Creates and invokes a Service Watch for a particular data service, given the
defined lifetime of the Service Watch, interval between data service polls, and
retention period of the monitored data
//
StatusCode status = api.createServiceWatch(serviceID, lifetime, interval,
retentionPeriod);

```

6.4.5.1.11 Delete Service Watch

This method offers the functionality to delete an existing Service Watch.

Parameters:

- serviceID: Defines the unique service ID of the data service to be watched.

Return Value:

Returns a StatusCode object indicating if the service has been updated successfully or not. The Java class StatusCode is defined in Listing 217.

Error Handling:

In case of an error, the error cause, e.g., the stacktrace, will be added to the StatusCode object.

Remarks:

Deleting a Service Watch does not mean that the monitored and stored data about the single data service polls will also be deleted. In fact, this will only happen once the retention period for the data has passed.

Listing 183: Source Code Example – Delete Service Watch

```
/**
 * @param serviceID, defines the unique service ID for which a Service Watch should
 * be deleted
 * @return a StatusCode object containing information about the success or failure of
 * this action
 */
public StatusCode deleteServiceWatch(UUID serviceID)

...
//Deletes a Service Watch for a particular service
StatusCode status = api.deleteServiceWatch(serviceID);
```

6.4.5.1.12 Get Service Watch Data

This method offers the functionality to retrieve a service watch data record for a particular service

Parameters:

- serviceID: Defines the unique service ID.
- start: The start time telling what data shall be included
- end: The end of this time span

Return Value:

Returns a Java List of ServiceWatchData objects. The Java class ServiceWatchData is defined in Listing 215.

Error Handling:

In case of an invalid serviceID or a non-existing Service Watch, an according exception will be thrown: i.e., ServiceNotFoundException, ServiceWatchNotFoundException.

Remarks:

None

Listing 184: Source Code Example – Retrieve Service Watch Data

```

/**
 * @param serviceID, defines the unique service ID
 * @param start, defines the start of the time span for which data will be retrieved
 * @param end, defines the end of the time span
 * @return List<ServiceWatchData> containing QoS information about the service
 * @throws: ServiceNotFoundException if the service was not found
 * @throws: ServiceWatchNotFoundException if no service watch exists
 */
public List<ServiceWatchData> getServiceWatchData(UUID serviceID, Date start, Date
end) throws ServiceNotFoundException, ServiceWatchNotFoundException

...
//Retrieve service watch data
List<ServiceWatchData> serviceWatchData = api.getServiceWatchData(serviceID);

```

6.4.5.2 RESTful Interfaces

6.4.5.2.1 Create Service

Analogue to the corresponding Java interface (see Section 6.4.5.1), this interface offers the functionality to create/store a service in the Service Registry. Because of the automated deployment of services in the chosen OSGi environment, services which are stored in the Service Registry are automatically deployed.

As can be seen in Table 55, two input parameters are necessary to create a service: A service file and a XML-based Manifest file.

Table 55: RESTful Interface Description – Create Service

Method	PUT		URL		\$API_ROOT/createService		
Description	Registers a service in the Service Registry						
File Stream	service	Required	yes	Possible Values	filestream	Description	A file which represents a service
File Stream	manifest	Required	yes	Possible Values	filestream	Description	XML description of the service
Example URL	\$API_ROOT/createService						
Response	HTTP status code + JSON object						
HTTP Status Code		Required	yes	Possible Values	200	Description	Service successfully created
					400		Bad request (Invalid service file...)
					403		Server side error
					404		Service to be created not found
					424		Unknown error occurred
JSON Object	http://SIMPLI-CITY.eu/ServiceRegistry/JSON-Schema/StatusCode						
JSON Attribute	serviceID	Required	yes	Possible Values	128 bit UUID	Description	ID for new service
JSON Attribute	serviceVersion	Required	yes	Possible Values	any string	Description	The new version of the created service
JSON Attribute	statusCodeNum	Required	yes	Possible Values	number	Description	Status code as number (same value as the HTTP status code)
JSON Attribute	statusCodeText	Required	yes	Possible Values	any string	Description	Status code in form of text
Example Response	HTTP/1.1 200 OK						

Listing 185 shows an example XML message containing a valid example for a service description. Importantly, only one service may be created during each invocation. For the sake of simplicity, the service binaries (i.e., the file stream) are not shown. In addition, the listing does not show the complete service description; instead, only a subset of the parameters are shown. For the full service description, see the description of the service Manifest in Section 9.1.

Listing 185: Service Description XML Example: Creation of a Service (Input Message)

```
<?xml version="1.0" encoding="UTF-8"?>
<ServiceDescription>
  <name>Example_Service</name>
  <version>1.1.0</version>
  <description>This is an example service description</description>
  <runtimeVersion>1.0.0-SNAPSHOT</runtimeVersion>
  <export>
    <interface>
      <class>eu.simpli-city.example.ExampleServiceInterface</class>
    </interface>
  </export>
  <sla>
    <responseTime>1000</responseTime>
    <throughput>300</throughput>
    <availability>99.9</availability>
  </sla>
  <license>
    <usageCosts>
      <costs>0.1</costs>
      <limitation>
        <maxInvocationTimeUnit>per_day</maxInvocationTimeUnit>
        <maxInvocations>100</maxInvocations>
      </limitation>
      <unit>per_invocation</unit>
    </usageCosts>
  </license>
</ServiceDescription>
```

Listing 186: JSON Example: Creation of a Service (Response Message)

```
{
  "serviceID": "89f47220-31bf-11e3-aa6e-0800200c9a66",
  "serviceVersion": "1.2.0",
  "statusCodeNum": "200",
  "statusCodeText": "SUCCESS"
}
```

6.4.5.2.2 Update Service

Similar to the corresponding Java interface (see Section 6.4.5.1.2), this interface offers the functionality to update an existing service in the Service Registry, i.e., provide a new version of the service or the service description. Notably, information about version compatibility is given in the Manifest file.

As can be seen in Table 56, one interface exists for updating the service. Like the corresponding Java method from Section 6.4.5.1.2, the RESTful interface accepts three different combinations of parameters:

- The serviceID, the service binaries and the service Manifest (XML)
- The service ID and the service binaries
- The service ID and the service Manifest (XML)

Table 56: RESTful Interface Description – Update Service

Method	PUT	URL	\$API_ROOT/updateService?:serviceID				
Description	Updates a service in the Service Registry.						
Parameter	serviceID	Required	yes	Possible Values	128 bit UUID	Description	Unique ID of the service to be updated
Filestream	service	Required	yes	Possible Values	filestream	Description	A file which represents a service
Filestream	manifest	Required	yes	Possible Values	filestream	Description	XML description of the service
Example URL	\$API_ROOT/updateService?serviceID=528739A0-F508-4551-A12A-04A9B51718D0						
Response	HTTP status code + JSON object						
HTTP Status Code		Required	yes	Possible Values	200	Description	Service successfully updated
					400		Bad request (Invalid service file...)
					403		Server side error
					404		Service to be updates not found
					424		Unknown error occurred
JSON Object	http://SIMPLI-CITY.eu/ServiceRegistry/JSON-Schema/StatusCode						
JSON Attribute	serviceID	Required	yes	Possible Values	128 bit UUID	Description	ID for updated service
JSON Attribute	serviceVersion	Required	yes	Possible Values	any string	Description	The new version of the updated service
JSON Attribute	statusCodeNum	Required	yes	Possible Values	number	Description	Update status code as number (same value as the HTTP status code)
JSON Attribute	statusCodeText	Required	yes	Possible Values	any string	Description	Update status code in form of text
Example Response	HTTP/1.1 200 OK						

Listing 187 shows an example XML message containing a valid example for a service description. This is needed to update an existing service in the Service Registry. In this case, both the service itself as well as its description (i.e., the Manifest) are updated. However, for the sake of simplicity, the service binaries (i.e., the filestream) are not shown.

Listing 187: Service Description XML Example: Service Update (Input Message)

```
<?xml version="1.0" encoding="UTF-8"?>
<ServiceDescription>
  <name>Example_Service</name>
  <version>1.1.0</version>
  <description>This is an example service description</description>
  <runtimeVersion>1.0.0-SNAPSHOT</runtimeVersion>
  <export>
    <interface>
      <class>eu.simpli-city.example.ExampleServiceInterface</class>
    </interface>
  </export>
  <sla>
    <responseTime>1000</responseTime>
    <throughput>300</throughput>
    <availability>99.9</availability>
  </sla>
  <license>
    <usageCosts>
      <costs>0.1</costs>
      <limitation>
        <maxInvocationTimeUnit>per_day</maxInvocationTimeUnit>
        <maxInvocations>100</maxInvocations>
      </limitation>
      <unit>per_invocation</unit>
    </usageCosts>
  </license>
</ServiceDescription>
```

Listing 188: JSON Example: Successful Service Update (Output Message)

```
{
  "serviceID": " 89f47220-31bf-11e3-aa6e-0800200c9a66",
  "serviceVersion": "1.2.2",
  "statusCodeNum": "200",
  "statusCodeText": "SUCCESS",
}
```

6.4.5.2.3 Delete Service

Analogue to the corresponding Java interface (see Section 6.4.5.1.3), this interface offers the functionality to delete an existing service from the Service Registry. As can be seen in Table 57, the interaction with the interface is straightforward – a unique service ID is needed as parameter and exactly this service will then be deleted.

Table 57: RESTful Interface Description – Delete Service

Method	DELETE	URL	\$API_ROOT/deleteService?:serviceID				
Description	Deletes a service from the Service Registry						
Parameter	serviceID	Required	yes	Possible Values	128 bit UUID	Description	Unique ID of the service to be deleted
Example URL	\$API_ROOT/deleteService?serviceID=528739A0-F508-4551-A12A-04A9B51718D0						
Response	HTTP status code + JSON object						
HTTP Status Code		Required	yes	Possible Values	200	Description	Service successfully deleted
					400		Bad request (Invalid service file...)
					403		Server side error
					404		Service to be deleted not found
					424		Unknown error occurred
JSON Object	http://SIMPLI-CITY.eu/ServiceRegistry/JSON-Schema/StatusCode						
JSON Attribute	serviceID	Required	yes	Possible Values	128 bit UUID	Description	ID of deleted service
JSON Attribute	serviceVersion	Required	no	Possible Values	any string	Description	The version of the deleted service
JSON Attribute	statusCodeNum	Required	yes	Possible Values	number	Description	Update status code as number (same value as the HTTP status code)
JSON Attribute	statusCodeText	Required	yes	Possible Values	any string	Description	Update status code in form of text
Example Response	HTTP/1.1 200 OK						

Listing 189 shows an example JSON message necessary to delete an existing service from the Service Registry.

Listing 189: Example: Service Deletion (Input Message)

```
http://simpli-city.eu/deleteService?serviceID=528739A0-F508-4551-A12A-04A9B51718D
```

Listing 190: JSON Example: Deletion of a Service (Response Message)

```
{
  "serviceID": "89f47220-31bf-11e3-aa6e-0800200c9a66",
  "serviceVersion": "1.2.0",
  "statusCodeNum": "200",
  "statusCodeText": "SUCCESS"
}
```

6.4.5.2.4 Get Service Descriptions

Reading service descriptions from the Service Registry is the key functionality of this component. There are different ways to request services, but all have in common that a filter in terms of an XPath expression (see Section 6.4.6.3) needs to be defined. The result of the request is in every case a list of service descriptions.

Table 58: RESTful Interface Description – Get Service Description

Method	GET	URL	\$API_ROOT/getServices?:filter				
Description	Retrieves service descriptions according the specified filter						
Parameter	filter	Required	yes	Possible Values	String	Description	A filter defining criteria a service should fulfil
Example URL	\$API_ROOT/getServices?filter=[type=parkingslotservice]						
Example URL	\$API_ROOT/getDataServices?filter=[type=parkingslotservice]						
Example URL	\$API_ROOT/getInternalServices?filter=[type=parkingslotservice]						
Response	HTTP status code + service descriptions in form of XML						
HTTP Status Code		Required	yes	Possible Values	200	Description	Services found
					400		Client side error (invalid request)
					404		No service found
					424		Unknown error
Example Response	HTTP/1.1 200 OK						

Table 58 shows an example for requesting service descriptions from the Service Registry. For the sake of simplicity, all three different interfaces are shown in the same table, since input and output are equally for each case.

Listing 191 shows the usage of the interface for a retrieving all service descriptions of parking slot services. Listing 192 shows the example output of such a service request, i.e., a service description. In order to save some space, this example service description is only a short version of the full Manifest described in Section 9.1.

Listing 191: Example URLs: Get Service Descriptions Fitting to the Defined Filter

```
$API_ROOT/getServices?filter=[type=parkingslotservice]
$API_ROOT/getDataServices?filter=[type=parkingslotservice]
$API_ROOT/getInternalServices?filter=[type=parkingslotservice]
```

Listing 192: XML Example: Output from Get Service Description Request

```
<?xml version="1.0" encoding="UTF-8"?>
<ServiceDescription>
  <name>Example_Service</name>
  <version>1.1.0</version>
  <description>This is an example service description</description>
  <runtimeVersion>1.0.0-SNAPSHOT</runtimeVersion>
  <export>
    <interface>
      <class>eu.simpli-city.example.ExampleServiceInterface</class>
    </interface>
  </export>
  <sla>
    <responseTime>1000</responseTime>
    <throughput>300</throughput>
    <availability>99.9</availability>
  </sla>
  <license>
    <usageCosts>
      <costs>0.1</costs>
      <limitation>
        <maxInvocationTimeUnit>per_day</maxInvocationTimeUnit>
        <maxInvocations>100</maxInvocations>
      </limitation>
      <unit>per_invocation</unit>
    </usageCosts>
  </license>
</ServiceDescription>
```

6.4.5.2.5 Get Service Binaries

In some cases, it is necessary to retrieve the binaries of a specific service from the Service Registry. For that, the requester has to pass the wanted service ID to the registry and will retrieve an archive file which includes the according binaries.

Table 59: RESTful Interface Description – Get Service Binaries

Method	GET	URL	\$API_ROOT/getServiceBinaries?:serviceID				
Description	Retrieves service binaries as an archive file for a defined service						
Parameter	serviceID	Required	yes	Possible Values	128 bit UUID	Description	Specifies the wanted service
Example URL	\$API_ROOT/getServiceBinaries?serviceID=528739A0-F508-4551-A12A-04A9B51718D0						
Response	HTTP status code + binary						
HTTP Status Code		Required	yes	Possible Values	200	Description	Services found
					404		No service found
					424		Unknown error
Example Response	HTTP/1.1 200 OK						

Table 59 shows the Service Registry RESTful interface for retrieving service binaries for a particular service.

Listing 193 shows the usage of the interface for retrieving the binaries for a particular service. The result of this request is an archive file in form of a binary filestream (not depicted).

Listing 193: JSON Example: Get Particular Service Binaries

```
http://simpli-city.eu/getServiceBinaries?serviceID=528739A0-F508-4551-A12A-04A9B51718D0
```

6.4.5.2.6 Create License

As described above, the actual license models offered by a service are part of the service description, i.e., the Manifest file. In addition, the Service Registry needs to store the actual granted licenses for single services for single users or apps (see Section 6.5). Analogue to the corresponding Java interface (see Section 6.4.5.1.6), this interface offers the possibility to create a license for an already registered service.

Table 60: RESTful Interface Description – Create License

Method	PUT	URL	\$API_ROOT/createLicense				
Description	Assigns a license to a particular service which is only valid of a defined user						
JSON Object	http://SIMPLI-CITY.eu/ServiceRegistry/JSON-Schema/License						
JSON Attribute	serviceID	Required	yes	Possible Values	128 bit UUID	Description	Unique service ID
JSON Attribute	userID	Required	yes	Possible Values	128 bit UUID	Description	Unique user ID
JSON Attribute	freeService	Required	yes	Possible Values	boolean	Description	Defines if the service is free of charge or not
JSON Attribute	usageCosts	Required	no	Possible Values	object	Description	Defines the usage costs for the defined service
JSON Object	http://SIMPLI-CITY.eu/ServiceRegistry/JSON-Schema/usageCosts						
JSON Attribute	unit	Required	no	Possible Values	string (see description)	Description	Defines the units of the costs, e.g. "per invocation", "per day", "per week", "per month", "per year"
JSON Attribute	cost	Required	no	Possible Values	integer	Description	Defines the costs which apply according the defined unit
JSON Attribute	limitation	Required	no	Possible Values	object	Description	Defines a limitation for the payment model
JSON Object	http://SIMPLI-CITY.eu/ServiceRegistry/JSON-Schema/Limitations						
JSON Attribute	maxInvocationTimeUnit	Required	no	Possible Values	string (see description)	Description	Defines the unit for the limitation criteria, e.g. "per invocation", "per day", "per week", "per month", "per year"
JSON Attribute	maxInvocations	Required	no	Possible Values	integer	Description	Defines the max invocations
Example URL	\$API_ROOT/createLicense						
Response	HTTP status code + JSON object						
HTTP Status Code		Required	yes	Possible Values	200 400 403 404 424	Description	License successfully defined License invalid Server side error No service found Unknown error
JSON Object	http://SIMPLI-CITY.eu/ServiceRegistry/JSON-Schema/StatusCode						
JSON Attribute	serviceID	Required	yes	Possible Values	128 bit UUID	Description	Unique service ID
JSON Attribute	statusCodeNum	Required	yes	Possible Values	number	Description	Create status code as number (same value as the HTTP status code)
JSON Attribute	statusCodeText	Required	yes	Possible Values	any string	Description	Create status code in form of text
Example Response	HTTP/1.1 200 OK						

Table 60 shows the interaction to create a license for a particular service. As it can be seen, three parameters are needed for this interaction – user ID, service ID, and the actual license. Based on this information, it is later possible to query if a particular user (e.g., a company or an app) are allowed to make use of a particular service.

Listing 194 shows the necessary input message necessary to create/store a license for a particular user ID (here: 138739A0-G508-1231-A12A-04A9B57464D0), service ID (here: 528739A0-F508-4551-A12A-04A9B51718D0), and the actual license.

Listing 194: JSON Example: Creation of a License (Input Message)

```
{
  "license": {
    "serviceID": "528739A0-F508-4551-A12A-04A9B51718D0",
    "userID": "138739A0-G508-1231-A12A-04A9B57464D0",
    "freeService": false,
    "usageCosts": {
      "unit": "per invocation",
      "costs": "0.1 €"
    },
    "limitation": {
      "maxInvocations": "100",
      "maxInvocationTimeUnit": "per day"
    }
  }
}
```

Listing 195: JSON Example: Create a License (Response Message)

```
{
  "serviceID": "528739A0-F508-4551-A12A-04A9B51718D0",
  "statusCodeNum": "200",
  "statusCodeText": "SUCCESS"
}
```

6.4.5.2.7 Read License

Analogue to the corresponding Java interface (see Section 6.4.5.1.7), this interface offers the functionality to read an existing license for a particular service and user (requester) from the Service Registry.

As it can be seen in Table 61, reading a license is straightforward – it is necessary to provide a service ID and a user ID, and the license is returned. If no license is available, a null value will be returned. Listing 196 shows the interaction necessary to request the license for a particular user/service combination; Listing 197 shows the result of this (successful) request.

Table 61: RESTful Interface Description – Read License

Method	GET	URL	\$API_ROOT/readLicense?:serviceID&:userID				
Description	Retrieves a license for a particular service and user						
Parameter	serviceID	Required	yes	Possible values	128 bit UUID	Description Unique service ID	
Parameter	userID	Required	yes	Possible values	128 bit UUID	Description Unique user ID	
Example URL	\$API_ROOT/readLicense?serviceID=528739A0-F508-4551-A12A-04A9B51718D0&userID=138739A0-G508-1231-A12A-04A9B57464D0						
Response	HTTP status code + JSON object						
HTTP Status Code		Required	yes	Possible Values	200	Description License found	
					400		No service found
					404		No license found
JSON Object	http://SIMPLI-CITY.eu/ServiceRegistry/JSON-Schema/License						
JSON Attribute	serviceID	Required	yes	Possible Values	128 bit UUID	Description Unique service ID	
JSON Attribute	userID	Required	yes	Possible Values	128 bit UUID	Description Unique user ID	
JSON Attribute	freeService	Required	yes	Possible Values	boolean	Description Defines if the service is free of charge or not	
JSON Attribute	usageCosts	Required	no	Possible Values	object	Description Defines the usage costs for the defined service	
JSON Object	http://SIMPLI-CITY.eu/ServiceRegistry/JSON-Schema/usageCosts						
JSON Attribute	unit	Required	no	Possible Values	string (see description)	Description Defines the units of the costs, e.g. "per invocation", "per day", "per week", "per month", "per year"	
JSON Attribute	cost	Required	no	Possible Values	integer	Description Defines the costs which apply according the defined unit	
JSON Attribute	limitation	Required	no	Possible Values	object	Description Defines a limitation for the payment model	
JSON Object	http://SIMPLI-CITY.eu/ServiceRegistry/JSON-Schema/Limitations						
JSON Attribute	maxInvocationTimeUnit	Required	no	Possible Values	string (see description)	Description Defines the unit for the limitation criteria, e.g. "per invocation", "per day", "per week", "per month", "per year"	
JSON Attribute	maxInvocations	Required	no	Possible Values	integer	Description Defines the max invocations	
Example Response	HTTP/1.1 200 OK						

Listing 196: Example URL: Request a License (Input Message)

```
$API_ROOT/readLicense?serviceID=528739A0-F508-4551-A12A-04A9B51718D0&userID=138739A0-G508-1231-A12A-04A9B57464D0
```

Listing 197: JSON Example: Request a License (Response Message)

```
{
  "license": {
    "serviceID": "528739A0-F508-4551-A12A-04A9B51718D0",
    "userID": "138739A0-G508-1231-A12A-04A9B57464D0",
    "freeService": false,
    "usageCosts": {
      "unit": "per invocation",
      "costs": "0.1 €"
    },
    "limitation": {
      "maxInvocations": "100",
      "maxInvocationTimeUnit": "per day"
    }
  }
}
```

6.4.5.2.8 Update License

Analogue to the corresponding Java interface (see Section 6.4.5.1.8), this interface offers the functionality to update an existing license for a particular service in the Service Registry.

Table 62: RESTful Interface Description – Update License

Method	PUT	URL	\$API_ROOT/updateLicense				
Description	Updates a license to a particular service and user						
JSON Object	http://SIMPLI-CITY.eu/ServiceRegistry/JSON-Schema/License						
JSON Attribute	serviceID	Required	yes	Possible Values	128 bit UUID	Description	Unique service ID
JSON Attribute	userID	Required	yes	Possible Values	128 bit UUID	Description	Unique user ID
JSON Attribute	freeService	Required	yes	Possible Values	boolean	Description	Defines if the service is free of charge or not
JSON Attribute	usageCosts	Required	no	Possible Values	object	Description	Defines the usage costs for the defined service
JSON Object	http://SIMPLI-CITY.eu/ServiceRegistry/JSON-Schema/usageCosts						
JSON Attribute	unit	Required	no	Possible Values	string (see description)	Description	Defines the units of the costs, e.g. "per invocation", "per day", "per week", "per month", "per year"
JSON Attribute	cost	Required	no	Possible Values	integer	Description	Defines the costs which apply according the defined unit
JSON Attribute	limitation	Required	no	Possible Values	object	Description	Defines a limitation for the payment model
JSON Object	http://SIMPLI-CITY.eu/ServiceRegistry/JSON-Schema/Limitations						
JSON Attribute	maxInvocationTimeUnit	Required	no	Possible Values	string (see description)	Description	Defines the unit for the limitation criteria, e.g. "per invocation", "per day", "per week", "per month", "per year"
Parameter	maxInvocations	Required	no	Possible Values	integer	Description	Defines the max invocations
Example URL	\$API_ROOT/updateLicense						
Response	HTTP status code + JSON object						
HTTP Status Code		Required	yes	Possible Values	200	Description	License successfully updated
					400		License invalid
					403		Server side error
					404		No service found
					424		Unknown error
JSON Object	http://SIMPLI-CITY.eu/ServiceRegistry/JSON-Schema/StatusCode						
JSON Attribute	serviceID	Required	yes	Possible Values	128 bit UUID	Description	Unique service ID of updated service
JSON Attribute	statusCodeNum	Required	yes	Possible Values	number	Description	Update status code as number (same value as the HTTP status code)
JSON Attribute	statusCodeText	Required	yes	Possible Values	any string	Description	Update status code in form of text
Example Response	HTTP/1.1 200 OK						

As it can be seen from Table 62, the input to update a license is analogue to the interaction necessary to create a new license, i.e., the service ID, user ID, and license itself are needed. Listing 198 shows an according example input message while Listing 199 shows an example response message if the update has been successful.

Listing 198: JSON Example: Update a License (Input Message Body)

```
{
  "license": {
    "serviceID": "528739A0-F508-4551-A12A-04A9B51718D0",
    "userID": "138739A0-G508-1231-A12A-04A9B57464D0",
    "freeService": false,
    "usageCosts": {
      "unit": "per invocation",
      "costs": "0.1 €",
      "limitation": {
        "maxInvocations": "100",
        "maxInvocationTimeUnit": "per day"
      }
    }
  }
}
```

Listing 199: JSON Example: Update a License (Response Message)

```
{
  "serviceID": "89f47220-31bf-11e3-aa6e-0800200c9a66",
  "statusCodeNum": "200",
  "statusCodeText": "SUCCESS"
}
```

6.4.5.2.9 Delete License

Analogue to the corresponding Java interface (see Section 6.4.5.1.9), this interface offers the functionality to delete an existing license for a particular service from the Service Registry.

Table 63: RESTful Interface Description – Delete a License

Method	DELETE	URL	\$API_ROOT/deleteLicense?:serviceID&:userID				
Description	Deletes a license to a particular service and user						
Parameter	serviceID	Required	yes	Possible Values	128 bit UUID	Description Unique service ID	
Parameter	userID	Required	yes	Possible Values	128 bit UUID	Description Unique user ID	
Example URL	\$API_ROOT/deleteLicense?serviceID=528739A0-F508-4551-A12A-04A9B51718D0&userID=138739A0-G508-1231-A12A-04A9B57464D0						
Response	HTTP status code + JSON object						
HTTP Status Code		Required	yes	Possible Values	200	Description License successfully deleted	
					400		Bad request
					403		Server side error
					404		License or service to be deleted not found
					424		Unknown error occurred
JSON Object	http://SIMPLI-CITY.eu/ServiceRegistry/JSON-Schema/StatusCode						
JSON Attribute	serviceID	Required	yes	Possible Values	128 bit UUID	Description ID for deleted service	
JSON Attribute	statusCodeNum	Required	yes	Possible Values	number	Description Status code as number (same value as the HTTP status code)	
JSON Attribute	statusCodeText	Required	yes	Possible Values	any string	Description Status code in form of text	
Example Response	HTTP/1.1 200 OK						

Table 63 shows that it is necessary to define a service/user combination to delete a license. Listing 200 shows an according input message while Listing 201 shows an example response if deletion was successful.

Listing 200: Example URL: Delete a License (Input Message)

```
http://simpli-city.eu/deleteLicenseModel?serviceID=528739A0-F508-4551-A12A-04A9B51718D0&userID=138739A0-G508-1231-A12A-04A9B57464D0
```

Listing 201: JSON Example: Delete a License (Response Message)

```
{
  "serviceID": "89f47220-31bf-11e3-aa6e-0800200c9a66",
  "statusCodeNum": "200",
  "statusCodeText": "SUCCESS"
}
```

6.4.5.2.10 Create and Update Service Watch

The (optional) Service Watch is used in order to regularly poll data services. Hence, mechanisms to create, update, and delete according service watches as well as mechanisms to retrieve the recorded data, need to be provided. Notably, data from a Service Watch is per se open to all interested parties, i.e., there is no restriction to access this data. The following RESTful interface offers the functionality to create and update a service watch (analogue to the corresponding Java interface as described in Section 6.4.5.1.10).

As can be seen in Table 64, for creating a Service Watch additional parameters are required (see Listing 202).

Table 64: RESTful Interface Description – Create a Service Watch

Method	PUT	URL	\$API_ROOT/createServiceWatch?:serviceId				
Description	Creates a new Service Watch for a particular service						
Parameter	serviceId	Required	yes	Possible Values	128 bit UUID	Description	Specifies the service to be watched
JSON Object	http://simpli-city.eu/ServiceRegistry/JSON-Schema/ServiceWatch						
JSON Attribute	lifetime	Required	yes	Possible Values	Date	Description	Defines the lifetime for the service watch
JSON Attribute	interval	Required	yes	Possible Values	Integer	Description	Defines the interval for the service watch
JSON Attribute	retentionPeriod	Required	yes	Possible Values	Integer	Description	Defines the retention period for the service watch
Example URL	\$API_ROOT/createServiceWatch?serviceId=528739A0-F508-4551-A12A-04A9B51718D0						
Response	HTTP status code + JSON object						
HTTP Status Code		Required	yes	Possible Values	200	Description	Service watch created successfully
					400		Bad request
					404		No service found
					424		Unknown error occurred
JSON Object	http://SIMPLI-CITY.eu/ServiceRegistry/JSON-Schema/StatusCode						
JSON Attribute	serviceId	Required	yes	Possible Values	128 bit UUID	Description	ID for the service
JSON Attribute	statusCodeNum	Required	yes	Possible Values	number	Description	Status code as number (same value as the HTTP status code)
JSON Attribute	statusCodeText	Required	yes	Possible Values	any string	Description	Status code in form of text
Example Response	HTTP/1.1 200 OK						

Listing 202 shows an example JSON message for creating a new Service Watch in the Service Registry. Importantly, only one Service Watch can exist per service. This means, if an additional request comes into the system, the existing Service Watch will be overwritten. Listing 203 shows the according example response message.

Listing 202: JSON Example: Creation of a Service Watch (Input Message)

```
{
  "serviceID": "exampleServiceID",
  "lifetime": "21253674",
  "interval": "8000",
  "retentionPeriod": "16000"
}
```

Listing 203: JSON Example: Creation of a Service Watch (Response Message)

```
{
  "serviceID": "89f47220-31bf-11e3-aa6e-0800200c9a66",
  "statusCodeNum": "200",
  "statusCodeText": "SUCCESS"
}
```

6.4.5.2.11 Delete Service Watch

Analogue to the corresponding Java interface (see Section 6.4.5.1.11), this interface offers the functionality to delete an existing Service Watch.

Table 65 shows the RESTful interface description for deleting a Service Watch.

Table 65: RESTful Interface Description – Deleting a Service Watch

Method	DELETE	URL	\$API_ROOT/deleteServiceWatch?:serviceId				
Description	Deletes a Service Watch from the Service Registry						
Parameter	serviceId	Required	yes	Possible Values	128 bit UUID	Description	Specifies the service watch to be deleted
Example URL	\$API_ROOT/deleteServiceWatch?serviceID=528739A0-F508-4551-A12A-04A9B51718D0						
Response	HTTP status code + JSON object						
HTTP Status Code		Required	yes	Possible Values	200 404	Description	Service Watch successfully deleted Service Watch to be deleted not found
JSON Object	http://SIMPLI-CITY.eu/ServiceRegistry/JSON-Schema/StatusCode						
JSON Attribute	serviceID	Required	yes	Possible Values	128 bit UUID	Description	ID of the service
JSON Attribute	statusCodeNum	Required	yes	Possible Values	number	Description	Status code as number (same value as the HTTP status code)
JSON Attribute	statusCodeText	Required	yes	Possible Values	any string	Description	Status code in form of text
Example Response	HTTP/1.1 200 OK						

Listing 205 shows the output for the request shown in Listing 204.

Listing 204: Example URL – Deletion of a Service Watch (Input Message)

```
http://simpli-city.eu/$API_ROOT/deleteServiceWatch?serviceID=528739A0-F508-4551-A12A-04A9B51718D0
```

Listing 205: JSON Example: Deletion of a Service Watch (Response Message)

```
{
  "serviceID": "528739A0-F508-4551-A12A-04A9B51718D0",
  "statusCodeNum": "200",
  "statusCodeText": "SUCCESS"
}
```

6.4.5.2.12 Retrieve Service Watch Data

Analogue to the Java method, a RESTful interface is also provided in order to retrieve a collection of Service Watch data entries for a particular service in a defined timespan.

Table 66 shows the RESTful interface description for retrieving Service Watch data entries. The JSON schema can be found in Listing 214.

Table 66: RESTful Interface Description – Retrieve Service Watch Data Entries

Method	GET	URL	\$API_ROOT/getServiceWatchData?:serviceId&:start&:end				
Description	Retrieves a collection of Service Watch data entries for a particular service in a defined timespan						
Parameter	serviceId	Required	yes	Possible Values	128 bit UUID	Description	Specifies service for which the serviceWatchData shall be returned
Parameter	start	Required	yes	Possible Values	date	Description	Specifies the start date for the first entry
Parameter	end	Required	yes	Possible Values	date	Description	Specifies the end date for the last entry
Example URL	\$API_ROOT/getServiceWatchData?serviceID=528739A0-F508-4551-A12A-04A9B51718D0&start='01.01.2013,00:00:00CET'&end='01.01.2014,00:00:00CET'						
Response	HTTP status code + JSON object						
HTTP Status Code		Required	yes	Possible Values	200 404	Description	Service Watch successfully deleted Service Watch to be deleted not found
JSON Object	http://SIMPLI-CITY.eu/ServiceRegistry/JSON-Schema/ServiceWatchData						
JSON Attribute	serviceID	Required	yes	Possible Values	128 bit UUID	Description	ID of the service
JSON Attribute	timestamp	Required	yes	Possible Values	date	Description	The date when this entry was recorded
JSON Attribute	responsetime	Required	yes	Possible Values	long	Description	The responsetime of the watched service
JSON Attribute	not_responding	Required	yes	Possible Values	boolean	Description	Indicates if the service was responding
Example Response	HTTP/1.1 200 OK						

Listing 207 shows the output for the request shown in Listing 206.

Listing 206: Example URL: Retrieving Service Watch Data Entries (Input Message)

```
$API_ROOT/getServiceWatchData?serviceID=528739A0-F508-4551-A12A-04A9B51718D0&start='01.01.2013,00:00:00CET'&end='01.01.2014,00:00:00CET'
```

Listing 207: JSON Example: Retrieving Service Watch Data Entries (Response Message)

```
[
  {
    "service_watch_data": {
      "serviceID": "528739A0-F508-4551-A12A-04A9B51718D0",
      "timestamp": "10.01.2014:10:10:53",
      "responsetime": "8000",
      "not_responding": "16000false"
    }
  },
  {
    "service_watch_data": {
      "serviceID": "528739A0-F508-4551-A12A-04A9B51718D0",
      "timestamp": "09.01.2014:10:10:53",
      "responsetime": "4000",
      "not_responding": "16000false"
    }
  }
]
```

6.4.6 Content Format

This section provides the content formats for the interfaces presented in the last subsections.

6.4.6.1 License JSON Schema

The License shown in Listing 208 represents the template for a license. A particular license is assigned to one single user (which could be either a user or an app) and one service. Those two fields are required in each license. The third field “freeService” allows a particular user to use the service free of charge. If this field is false, then it is required to define a usage cost field, i.e., if a user has to pay “per invocation”, “per day”, “per week”, “per month” or “per year”. In addition to that, the applying costs for this unit have to be defined. Further, it is possible to define a maximum of resulting costs which are defined in the field “limitation”. The license model will be integrated into the service Manifest file (see Section 9.1).

Listing 209 provides additional information by showing the example for the default license model.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 304 / 435
http://www.simpli-city.eu/	Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201			

Listing 208: JSON Schema – License

```

{
  "type": "object",
  "id": "http://SIMPLI-CITY.eu/ServiceRegistry/JSON-Schema/License",
  "properties": {
    "serviceID": {
      "type": "string",
      "required": true
    },
    "userID": {
      "type": "string",
      "required": true
    },
    "freeService": {
      "type": "Boolean",
      "required": true
    },
    "usageCosts": {
      "type": "object",
      "id": "http://SIMPLI-CITY.eu/ServiceRegistry/JSON-Schema/usageCosts",
      "required": false,
      "properties": {
        "unit": {
          "type": {
            "enum": [
              "per invocation",
              "per day",
              "per week",
              "per month",
              "per year",
              "unlimited"
            ]
          },
          "required": false
        },
        "cost": {
          "type": "integer",
          "required": false
        }
      }
    },
    "limitation": {
      "type": "object",
      "id": "http://SIMPLI-CITY.eu/ServiceRegistry/JSON-Schema/limitation",
      "required": false,
      "properties": {
        "maxInvocationTimeUnit": {
          "type": {
            "enum": [
              "per invocation",
              "per day",
              "per week",
              "per month",
              "per year"
            ]
          },
          "required": false
        },
        "maxInvocations": {
          "type": "integer",
          "required": false
        }
      }
    }
  }
}

```

Listing 209: JSON Example – Default License Model

```
{
  "license": {
    "serviceID": "528739A0-F508-4551-A12A-04A9B51718D0",
    "freeService": "true"
  }
}
```

6.4.6.2 License Java Class

The class License is the Java representation of the License JSON schema presented in Section 6.4.6.1. It contains the same fields. Notably, the same class is also used to describe license models.

Listing 210: Java Class: License

```
public class License{
    /**
     * unique service id
     */
    private UUID serviceID;
    /**
     * unique user id
     */
    private UUID userID;
    /**
     * indicates if the service is free of charge
     */
    private Boolean freeService;
    /**
     * defines the usage costs per time unit
     */
    private UsageCosts usageCosts;
    /**
     * defines the max allowed number of invocations per time unit
     */
    private Limitation limitation;

    static class UsageCosts {
        public enum Unit {Invocation, Day, Week, Month, Year, Unlimited};
        /**
         * defines the time unit for the costs
         */
        private Unit unit;
        /**
         * defines the costs per unit
         */
        private int costs;
    }

    static class Limitation{
        public enum MaxInvocationTimeUnit {Invocation, Day, Week, Month,
            Year};
        /**
         * defines the time unit for the max invocations
         */
        private MaxInvocationTimeUnit maxInvocationTimeUnit;
        /**
         * defines the max invocations per time unit
         */
        private int maxInvocations;
    }
    //... getter() and setter()
}
```

6.4.6.3 Filter

Filters are used to specify what kind of information about one or more services should be returned based on the individual service Manifests (see Section 9.1). In general, if there is no filter defined at all, a list of all service descriptions will be returned. A filter is represented by an XPath expression⁸⁹ (see example in Listing 211). A valid output example is presented in Listing 212.

Listing 211: Filter Format Example – Input

```
/service[type='parkingslotservice']/serviceID
```

Listing 212: Filter Format Example – Output (JSON)

```
{
  "serviceID": "0A644671-F5EC-41F7-B594-8F79F811CEB5",
}
```

6.4.6.4 Service Watch

The JSON schema for a Service Watch is used to create a new Service Watch for a particular service. Listing 213 shows the JSON schema for the description of a Service Watch.

Listing 213: JSON Schema – Service Watch

```
{
  "type": "object",
  "id": "http://SIMPLI-CITY.eu/ServiceRegistry/JSON-Schema/ServiceWatch",
  "serviceID": {
    "type": "String",
    "required": true
  },
  "lifetime": {
    "type": "date",
    "required": true
  },
  "interval": {
    "type": "integer",
    "required": true
  },
  "retentionPeriod": {
    "type": "integer",
    "required": true
  }
}
```

⁸⁹ http://www.w3schools.com/xpath/xpath_syntax.asp

6.4.6.5 Service Watch Data Model

The JSON schema below describes the monitored data recorded by a Service Watch. It contains the following fields:

- **serviceID**: The unique id for a data sensor
- **timestamp**: The timestamp when this data was created
- **responsetime**: The response time of the service
- **not_responding**: Indicates if the service was not responding

Listing 214: JSON Schema – Service Watch Data

```
{
  "type": "object",
  "id": "http://SIMPLI-CITY.eu/ServiceRegistry/JSON-
Schema/ServiceWatchData",
  "serviceID": {
    "type": "String",
    "required": true
  },
  "timestamp": {
    "type": "date",
    "required": true
  },
  "responsetime": {
    "type": "long",
    "required": true
  },
  "not_responding": {
    "type": "boolean",
    "required": true
  }
}
```

6.4.6.6 Service Watch Data Java Class

The class ServiceWatchData represents a record created by a Service Watch:

Listing 215: Java Class: ServiceWatchData

```
public class ServiceWatchData {

    private UUID serviceID;
    private Date timestamp;
    private Long responsetime;
    private Boolean notResponding;

    //... getter() and setter()
}
```

6.4.6.7 StatusCode

The StatusCode JSON schema depicted in Listing 216 is used to provide feedback about a performed request. In addition, a numeric status code as well as a textual representation about the overall status is returned; possible values are defined below:

- 200 or “SUCCESS”: Request successful.
- 400 or “CLIENT_SIDE_ERROR”: An error occurred, the error’s origin was caused by the client.
- 403 or “SERVER_SIDE_ERROR”: The error was caused on the server side.
- 404 or “SERVICE NOT FOUND”: The specified service was not found.
- 424 or “UNKNOWN_ERROR”: An unknown error occurred

Listing 216: JSON Schema – StatusCode

```
{
  "type": "object",
  "id": " http://SIMPLI-CITY.eu/ServiceRegistry/JSON-Schema/StatusCode",
  "serviceID": {
    "type": "String",
    "required": true
  },
  "serviceVersion": {
    "type": "String",
    "required": false
  },
  "statusCodeNum": {
    "type": {
      "enum": [
        "200",
        "400",
        "403",
        "404",
        "424"
      ]
    },
    "required": true
  },
  "statusCodeText": {
    "type": {
      "enum": [
        "SUCCESS",
        "CLIENT_SIDE_ERROR",
        "SERVER_SIDE_ERROR",
        "SERVICE_NOT_FOUND",
        "UNKNOWN_ERROR"
      ]
    },
    "required": true
  }
}
```

6.4.6.8 StatusCode Java Class

The class StatusCode (depicted in Listing 217) is used in order to pass information of a performed action (e.g., create, update or delete) to the requester. The status code gives information about the action's success or failure, in addition, in case of an error, the stack trace can also be stated.

Listing 217: Java Class: StatusCode

```
public class StatusCode {
    /**
     * numerical representation of the status
     */
    public enum Code {
        200, 400, 403, 404, 424
    }

    /**
     * textual representation of the status
     */
    public enum Status {
        SUCCESS, CLIENT_SIDE_ERROR, SERVER_SIDE_ERROR, SERVICE_NOT_FOUND,
        UNKNOWN_ERROR,
    }

    /**
     * Error message
     */
    private String errorMsg;

    /**
     * Status code as number
     */
    private Code statusNumber;

    /**
     * Status code as text
     */
    private Status statusText;

    /**
     * unique service id
     */
    private UUID serviceID;

    //... getter() and setter()
}
```

6.4.6.9 Service Level Agreement Model

The SLA JSON Model is a simple JSON Schema representing a SLA between a user and service. It contains the following properties:

- **serviceID:** A unique service ID
- **userID:** A user ID, can either be a user, or an app ID
- **responsetime:** The maximum response time for the defined service in milliseconds
- **availability:** The service availability in percent which has to be ensured in a particular timespan
- **throughput:** The minimal throughput which has to be ensured

For now, only the mentioned fields are important, however, this class may be extended during development of SIMPLI-CITY. Listing 218 shows the SLA JSON schema, while Listing 219 shows the corresponding Java class.

Listing 218: SLA JSON Schema

```
{
  "sla": {
    "type": "object",
    "id": "http://SIMPLI-CITY.eu/Registry/JSON-Schema/SLA",
    "required": true,
    "properties": {
      "responsetime": {
        "type": "long",
        "required": true
      },
      "availability": {
        "type": "double",
        "required": false
      },
      "throughput": {
        "type": "double",
        "required": false
      }
    }
  }
}
```


6.4.6.10 Service Level Agreement Java Class

The SLA Java class is a simple POJO representing a SLA between a user (defined by userID) and service. It contains the same fields as the corresponding JSON schema in Section 6.4.6.9.

Listing 219: SLA Java Class

```
public class SLA {
    /**
     * The ID of the monitored service
     */
    private UUID serviceID;

    /**
     * The ID of the service requester (user)
     */
    private UUID requesterID;

    /**
     * The maximum response time for the defined service in milliseconds
     */
    private long responsetime;

    /**
     * The service availability in percent which has to be ensured
     */
    private double availability;

    /**
     * The minimal throughput which has to be ensured
     */
    private double throughput;

    //... getter() and setter()
}
```

6.4.7 Summary

Within the SIMPLI-CITY Mobility Services Framework, the Service Registry is an important helper component providing functionalities needed to create and update services, check service licenses, and regularly check if data services are still alive. As OSGi provides an extensive service registry concept, which largely fits the requirements defined for the SIMPLI-CITY Service Registry, the internal service registry of the chosen OSGi runtime environment (see Section 6.1) will be the fundamental technology applied within SIMPLI-CITY.

In addition, mechanisms for license checking and to define Service Watches will be implemented. Furthermore, existing functionalities will be extended in order to cater for the SIMPLI-CITY service data model and to realize the corresponding data management and service discovery functionalities.

6.5 Service and App Marketplaces

6.5.1 Major Design Decisions

The Service Marketplace is the main resource for developers that want to improve apps with new data or services offered by services actively marketed in the Service Marketplace. The Service Marketplace may also be used by service developers if they want to find and reuse services offered by other developers. Additionally, the App Marketplace is the platform for customers that want to increase their mobile experience with new functionalities and features offered by apps in the App Marketplace.

As there will be four kinds of roles used in this section, the following list will clarify the responsibilities and a short description for each role:

- **Authors:** Authors are users that are responsible for apps or services. They might not be developers themselves, but they upload, edit and remove apps and services at the respective marketplaces.
- **Developers:** Developers create apps and services for the SIMPLI-CITY platform and provide authors with the compiled bundles that are created. Developers can be authors as well, but this is not compulsory.
- **Reviewers:** Reviewers are responsible for accepting newly uploaded apps and services. Once apps or services are uploaded, they need to be reviewed by a person in this role. Once a new app or service is accepted, it is listed and available via the respective marketplace.
- **Customers:** Customers are the end users of SIMPLI-CITY. They are the users of the Personal Mobility Assistant and the target group for the App Marketplace. Customers do not have access to the Service Marketplace.

The App Marketplace is offered to customers in two forms: the web- and the PMA-variants. Both of them enable users to open and operate the marketplace.

The Service Marketplace is only available as a web view, providing developers with information about available services. So developers who want to make use of SIMPLI-CITY services or add a new service to the marketplace have to invoke it from a web browser.

The App Marketplace is the central place for users to find and install new apps for their Personal Mobility Assistant. Customers can use it as a PMA application that provides all the functionality to browse and install apps in the PMA. Moreover, the App Marketplace also provides web UIs for developers and reviewers. All the information associated with apps is stored in an internal database of the App Marketplace.

The payment of apps and services is also done with internal components that will be further described in the Section 6.5.4.

During the discussion of the Global Architecture (deliverable D3.1) and the Functional Specification (deliverable D3.2.1), the following major design decisions have been made:

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 314 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

Ease of Use:

The Service and especially App Marketplaces should be easy to use, as customers are not supposed to be technology experts. This implies an easy-to-use GUI for both the web and the PMA UIs. The Service Marketplace is not restricted to a simple UI since it is used by developers that have a technical background.

Availability:

The App Marketplace will be the most frequently used app on the PMA and the web views of the Service and App Marketplaces will be highly visited websites. All components (e.g., the used webserver and the technologies) used by these marketplaces need to be highly available for every frontend described in this section.

6.5.2 Technology Comparison

6.5.2.1 Comparison Criteria

Table 67: Criteria for Technical Specification

Parameter	Importance	Description
Distributed Architecture	++	Since SIMPLI-CITY will have to handle a large number of end user requests, it is necessary to be able to handle a big amount of requests simultaneously. Besides parallelism the Service and App Marketplaces need a highly concurrent and scalable backend system to serve data as fast and reliable as possible.
Adaptable UI	+	The Service Marketplace is used by developers that need a different level of information than customers that make use of the web- and PMA-based App Marketplace. Therefore, the App Marketplace needs to be much more general and user friendly, especially for non-technical customers.
Connectivity for Backend Data	++	In order to provide the end users/developers with the newest apps/services, the Service and App Marketplaces should serve stable and fast communication to the storage backend.
Licensing Support	++	For the implementation, the used technologies should provide a free (or open source) license, so the developers will not have any constraints during programming.

6.5.2.2 Possible Technologies and Comparison – App Marketplace

The App Marketplace will be based on a solution that fulfils the necessary functionalities and views needed to provide customers with an easy to use solution. The App Marketplace makes use of its own approach for storing information about apps, hence there is no need for the solution to provide its own database access and schema. The data will be received via the App Registry, which stores information inside the Cloud-based Information Infrastructure.

Google Play:

Google Play is the most widely known store for Android apps, as it is the store operated by Google, which owns the major share of Android OS variations in the market. It meets nearly all the listed criteria except the open source ability. Unfortunately it is licensed with a non-free license, which renders it not suitable for SIMPLI-CITY.

GetJar:

GetJar is a freeware app for Android that enables users to download over 150,000 apps for several mobile systems for free. It requires Android OS 2.1 or higher. GetJar makes use of a custom, non-free, license and is therefore not suitable for the App Marketplace of the SIMPLI-CITY project.

F-Droid:

The F-Droid repository includes more than 500 open source apps for free download. These apps are sorted in categories as well as in subcategories to simplify the user search. F-Droid makes use of the GPLv3 License and requires all of the apps in the store to be released under this license as well and is therefore not suitable for the App Marketplace component of SIMPLI-CITY.

MyMarket:

MyMarket is a marketplace platform written in Java. It is highly customizable due to access to the source code. SIMPLI-CITY project partners have a lot of experience with this marketplace solution, which makes it the best fitting solution for the App Marketplace component. The MyMarket comes with a special license, which is granted to the SIMPLI-CITY project as well as the source code of the market software. This provides feature extensibility and further visual customization of the given solution to the SIMPLI-CITY project.

Table 68: Comparison of Technologies for the App Marketplace

Parameter	Importance	F-Droid	Google		
			GetJar	Play	MyMarket
Generic Criteria					
Up-to-Datedness	++	10	8	10	10
Stability	+	8	8	10	10
Extensibility & Open Source/Standards	+/-	6	6	6	10
Familiarity	+	0	0	0	10
Performance	++	6	6	10	10
Interoperability	++	0	0	0	10
License	(e.g., Apache 2.0)	GPLv3	Custom License	Custom License	Custom License
Specific Criteria					
Distributed Architecture	++	1	1	1	10
Adaptable UI	+	6	6	10	10
Connectivity for Backend Data	++	1	1	1	10
Licensing Support	++	10	10	1	10

6.5.2.3 Possible Technologies and Comparison – Service Marketplace

The number of possible basic technologies for the SIMPLI-CITY Service Marketplace is quite limited. The limited number of possible existing solutions which, despite their provision of features listed in the requirements, are considered for the Service Marketplace is even reduced due to their limitation to physical items. The Service Marketplace needs to provide several views to create, update and delete services to service developers as well as functionality to browse and search for new services to app developers. The Service Marketplace will be handled by a web application, so service developers who want to develop and host services on their own service platform can still publish their services to the Service Marketplace.

PHPB2B:

The PHPB2B marketplace script is a full featured marketplace for bidding purposes. It is based on PHP and uses MySQL as storage. It also makes use of the Model View Controller (MVC) design pattern that allows users and developers (in this case the SIMPLI-CITY project) to customize it in any possible way. Since PHPB2B is a marketplace for physical items the only way to make use of it is to modify the existing code, which means to read and understand the existing codebase as well as changing almost everything. While PHPB2B in general meets the requirements of the SIMPLI-CITY Service Marketplace, it is published under the viral GPL license, and therefore not suitable for SIMPLI-CITY

B2B Marketplace Script:

The B2B Marketplace Script is a modern looking yet customizable B2B script written by a company named “Eicra”. It provides all features needed to create a B2B portal without any programming knowledge. The B2B Marketplace script is targeted at physical items and therefore not suitable for the Service Marketplace of SIMPLI-CITY.

Both discussed marketplaces are open systems but their general purpose is to create a business-to-business platform similar to eBay. As this is not the goal for the Service Marketplace none of the shown technologies fit to our requirements.

Table 69: Comparison of Technologies for the Service Marketplace

Parameter	Importance Generic Criteria	PHPB2B Script	B2B Marketplace Script
Up-to-Datedness	++	10	10
Stability	+	10	8
Extensibility & Open Source/Standards	+/-	8	6
Familiarity	+	10	8
Performance	++	10	8
Interoperability	++	10	10
License	(e.g., Apache 2.0)	GPLv3	Custom License
Specific Criteria			
Distributed Architecture	++	10	10
Adaptable UI	+	8	6
Connectivity for Backend Data	++	10	0
Licensing Support	++	10	2

6.5.2.4 Service Marketplace – Comparison Addendum

As seen in the comparison tables in the last subsections, there is no solution that fulfills all requirements, since PHPB2B fits the requirement very well but is published under a GPL license. Hence, the SIMPLI-CITY Service Marketplace needs to be implemented from scratch. Table 70 therefore compares different web frameworks/technologies that could be used for the implementation of the Service Marketplace. Afterwards, different frameworks/technologies which could be applied, will be discussed.

Table 70: Description of Specific Criteria for the Service Marketplace Comparison

Parameter	Importance	Description
Templates	++	Templates are used to divide frontend web pages from programming code and are therefore useful to manage the codebase and to keep it clean and organized.
Scaffolding	+	Scaffolding is the possibility to create a skeleton for new modules in the framework application. This usually includes a number of generic views for a module as well as a very simplified codebase.
ORM	++	An Object Relational Mapper (ORM) automatically maps data from a database to a model in the codebase. An ORM is helpful because it keeps developers from writing database statements (e.g., SQL) themselves.
Caching	++	Caching accelerates the rendering of server pages by saving pages that have not changed and respond to requests with those saved pages instead of re-rendering and re-executing the complete code for a single page.
Form Validation	+	Upon letting the customer enter data via forms, this input data needs to be validated. A component to automatically check the data on SQL Injection or other security threats is helpful and therefore part of the criteria.

Revel:

Revel is a small Web Framework for the Go programming language. It provides basic framework functionality like routing, templates, a testing framework and internationalization. Revel is built on the internal Go HTTP server, which is a very fast performing web server. This factor and the high extensibility of the generated web apps are the main factors for Revel to fit into the requirements needed for the Service Marketplace.

Rails:

Rails was the entrance of the MVC design pattern to the populace of developers, risen with version 2.0 in 2007. It is written in the programming language Ruby, which is an object oriented programming language with a steep learning curve. It provides many required features out of the box and many add-ons via so called RubyGems that can extend the environment of a Rails app. There are different web servers for Rails apps, that provide different core features, but none of them is easy to configure, so the configuration

of the web server as well as the bad performance of the programming language are the main factors to not choose Rails as the target framework for the Service Marketplace.

CakePHP:

CakePHP is a web framework similar to Rails, building on top of the MVC design pattern. It resembles the RubyGem approach with a portal called “Bakery”, containing plugins to extend the base framework. PHP is known to be the most used programming language for web applications today and therefore a popular target for security attacks. Additionally, the framework consists of many technical layers and is therefore prone to faults that can block the whole system in case of a minor error.

Django:

Django resembles the Rails approach for the programming language Python, which similar to Ruby has a steep learning curve. Django needs the usage of a Web Server Gateway Interface (WSGI) and is therefore not runnable on every web server available. Thus the used server needs to be highly configured due to the WSGI-extension. Django provides poor runtime performance due to the different layers of execution (Webserver → WSGI module → interpreted Python Code → Database). Therefore, Django is not the best suitable technology for the Service Marketplace.

Table 71: Comparison of Possible Programming Languages and Frameworks for the Service Marketplace

Parameter	Importance	Revel	Rails	CakePHP	Django
Generic Criteria					
Up-to-Datedness	+	9	9	8	7
Stability	+	8	6	7	8
Extensibility & Open Source/Standards	+/-	10	10	8	9
Familiarity	+	9	7	7	3
Performance	+/-	10	2	2	3
Interoperability	++	10	10	7	9
Specific Criteria					
Templates	++	8	8	10	8
Scaffolding	+	0	8	8	8
ORM	++	4	8	8	8
Caching	++	10	10	10	10
Form Validation	+	8	8	9	8

6.5.3 Technology Selection

6.5.3.1 Selection for App Marketplace

As seen in the comparison of existing application marketplaces, the MyMarket solution fulfils most of the requirements needed by the SIMPLI-CITY app marketplace, and is therefore chosen for development. The MyMarket solution will be used as a basis of the App Marketplace, but further development will be needed in order to provide all the requirements of the App Marketplace and in order to integrate it within the different server-side and PMA-side components of SIMPLI-CITY.

6.5.3.2 Selection for Service Marketplace

As discussed above, none of the presented solutions can be applied as foundation for the SIMPLI-CITY Service Marketplace – since PHPB2B fits the requirement very well but is published under a GPL license. Hence, this marketplace will be implemented from scratch.

Although some specific criteria are not provided by the framework itself, the programming language Go with the Revel web framework is the best choice due to the speed of the Go programming language combined with the extensibility of the Revel framework. In a real world environment, speed is the highest rated performance characteristic as well as stability and reliability. Additionally, providing a lightweight solution will make it much easier to exchange bottleneck components (e.g., ActiveRecord in Rails) with faster performing components (e.g., DataMapper in Rails).

6.5.3.3 Missing Elements and Implementation Needs

The following functionalities have to be implemented since they are either currently missing in the available technologies or there is simply no information about how a particular system or application has realized it.

Design:

The common interface design of the App and Service Marketplaces needs to be defined to create an integrated user experience of all end user components of SIMPLI-CITY.

App Marketplace:

The MyMarket solution is used as a basis for the App Marketplace. The current MyMarket version will be used as it is currently implemented in the cases that it works within the SIMPLI-CITY project, including interfaces between the different components. But further implementation is needed to meet the needs of SIMPLI-CITY. The most important extension needs are the following:

- Modification of the app upload-process in order to include the required parameters to be used in SIMPLI-CITY.
- Simplification of the review process.
- Modification of the app search process in order to include the required search options.
- Improve the management of the installed apps in the client-side application of the marketplace, as the current implementation only provides a basic mechanism.
- Integration of the client-side of the marketplace within the PMA.
- Integration of the server-side of the marketplace with the other server-side components of SIMPLI-CITY.
- Add payment functionalities.
- Add support for the newest versions of Android.
- Removal of iOS support.

Service Marketplace:

The Service Marketplace will be written from scratch, because there is no suitable solution that fulfils all the criteria (see Section 6.5.3). This includes all components of the frontend as well as all the backend functionality. The backend does not include much implementation needs since the data storage is handled via the Cloud Storage API of the

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 320 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

Cloud-based Information Infrastructure. The frontend will be developed in the Go programming language with the Revel web framework due to the speed and the caching mechanisms Revel provides out of the box. This interaction keeps the overhead on the web server small and provides customers with fast response times at any time.

The Service Marketplace includes the functionality to check licenses of services for developers. This can include three types of licenses:

- Free of use
- Pay per use
- Flatrate

Author UI:

The Author UI provides authors with the possibility to upload, edit and remove apps and services. While the Author UI for apps will be provided by the chosen solution for the App Marketplace (MyMarket), the Author UI for services needs to be completely developed from scratch. This includes functionality to upload new services to the Service Registry, edit their descriptions and license model, and changing the visibility of services.

Reviewer UI:

The Reviewer UI is used by persons in the reviewer role to test, validate and accept new services. While the solution for apps is already covered within the chosen solution (MyMarket) the Reviewer UI for services will be newly developed.

6.5.3.4 Further Information and Conclusion from Technology Comparison

Several technologies that could serve as base technologies for the App Marketplace and the Service Marketplace have been examined in regards of different requirements. Especially critical requirements like used programming language and framework, performance characteristics, extensibility and domain features as well as less critical but also telling indicators of a suitable technology like up-to-datedness have been examined and rated to make final technology choices.

The App Marketplace will make use of the existing solution MyMarket because it fits the requirements in most cases. The main reason for using this solution is that it is the only solution that is focused on providing the same functionalities needed by the SIMPLI-CITY App Marketplace, including the processes of uploading apps by the developers, reviewing of apps by the reviewers, and browsing and downloading of apps by the end users. Moreover, it has been implemented using the same technologies used in SIMPLI-CITY (mainly Java and Android for the client-side), so it will be easy to extend and to integrate with the other SIMPLI-CITY components.

The Service Marketplace has to be written from scratch because there is no suitable solution concerning the given requirements. Due to that fact and the need for a well scaling solution the Service Marketplace will be written in the Go programming language while making use of the lightweight web framework Revel, which provides a minimal implementation of the MVC design pattern, granting fast development times for trivial tasks (e.g., it automatically creates views for a model).

6.5.4 Component Structure

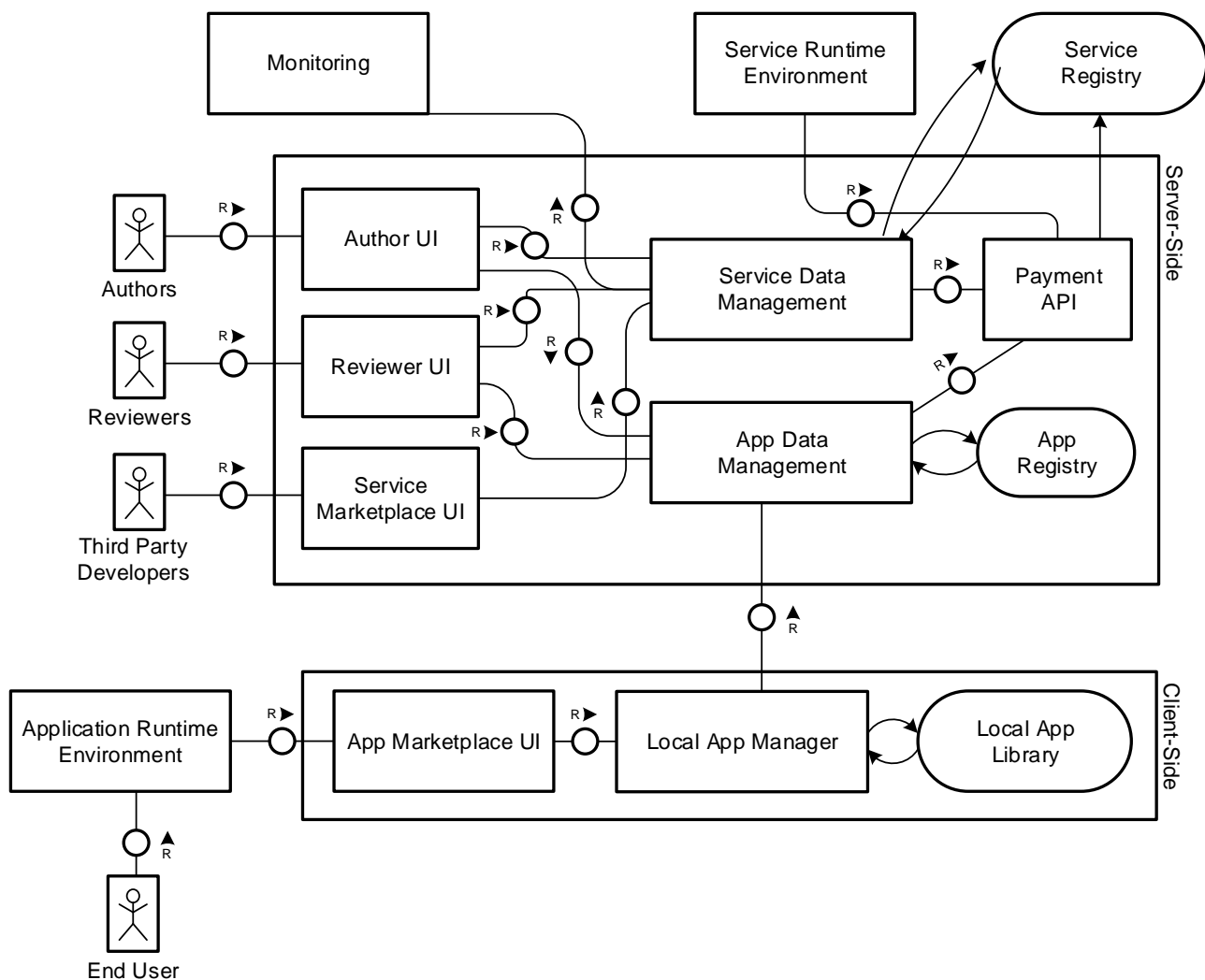


Figure 24: Service and App Marketplaces Subcomponents

This component contains two separate marketplaces that are split by their target users. The App Marketplace is used by customers and only contains apps to be used within the Personal Mobility Assistant (PMA). These apps will extend the features and possibilities of the PMA and enhance the Quality of Experience (QoE) for customers.

The Service Marketplace is used by developers that either want to publish their own services, which may contain new data sources or provide backend functionality to app and service developers. These app developers want to make use of the Service Marketplace to browse and search for new services to increase the feature set of their apps.

As can be seen in Figure 24, the two marketplaces share certain subcomponents (e.g., Payment API), but most subcomponents are either App Marketplace- or Service Marketplace-specific. This is due to the fact that the user groups of the marketplaces differ to quite some extent.

App Marketplace UI:

The App Marketplace UI is the main app to find and install new SIMPLI-CITY apps on the Personal Mobility Assistant (PMA). It runs on the PMA which contains the App Marketplace UI as an app named "SIMPLI-CITY Store".

The App Marketplace UI allows end users to browse apps published in the marketplace and install them on their devices. It provides a search functionality that permits to search for apps matching specific filter parameters, and to read the app description. Moreover, users are able to provide feedback concerning the app, to help other users decide about the quality of apps they may want to install.

Author UI:

The Author UI subcomponent is a web GUI aimed at authors that permits to manage the publication of apps and services in the App Marketplace and Service Marketplace, respectively. It permits to upload new apps and services, update their descriptions, visualize usage information, and remove them from the marketplaces. The Author UI integrates two main functionalities offered to the developer: service management and app management.

The creation and uploading of new services to SIMPLI-CITY is carried out through the Service Development API, which registers the information of a service in the Service Registry. The Author UI allows the publication of these services in the Service Marketplace. When a developer accesses the Author UI, the Author UI queries the Service Registry to receive the list of services associated to the developer and then shows the list of these services, both the published services and the services created but not published yet. Then, the developer can edit the information associated to a service, like the licensing information, and publish the service in the Service Marketplace. The developer can also visualize the information of a published service, and modify it if necessary. Finally, the Author UI allows the removal of a service from the Service Marketplace.

In a similar way, the Author UI permits to manage the publication of apps in the App Marketplace. In this case, the Author UI also covers the creation of new apps in SIMPLI-CITY, by providing a form where the developer can introduce all the information of an app and upload the app bundle. All this information is registered in the App Registry, but the application still needs to be reviewed and accepted in order to be published in the App Marketplace. The Author UI also permits to upload new version of an app, visualize the information of an app, and remove an app from the Marketplace.

Reviewer UI:

The Reviewer UI is a web GUI for the reviewers of SIMPLI-CITY that will permit to review and validate the apps and services uploaded by authors/developers prior to its publication in the Service and App Marketplaces.

The Reviewer UI will show the list of pending services and apps to be reviewed. The reviewer is able to use and test the services and apps, accept or reject them, and provide a written feedback to the developers. This feedback is then sent to the developer via e-mail.

Once a service or app is approved by a reviewer, it is published in the Service or App Marketplace in order to be used by other developers or end users, respectively.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 323 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

Service Marketplace UI:

The Service Marketplace UI is a web GUI aimed at third party developers that permits to search services published in the Service Marketplace in order to (re-)use them within their applications.

It provides a search form in order to look for services based on different input parameters like the name of the service, the description, or keywords. The result of the search provides a list of services matching the search parameters. The user is able to visualize the detailed information of a service, and also the information about how to use the service within their applications.

Local App Manager:

The Local App Manager is a subcomponent of the PMA responsible for the management of the apps within the PMA, including the installation and uninstallation of apps. It also takes charge of the communication with the server-side of the App Marketplace, specifically with the App Data Management subcomponent.

The Local App Manager receives requests from the App Marketplace UI, like a search request or a request to install an app, and then queries the App Data Management component to perform the request. The Local App Manager stores data about installed apps in the Local App Library.

App Data Management:

The App Data Management is the main backend subcomponent of the App Marketplace, handling the server-side functionality for managing apps and providing access to the information stored in the App Registry.

It provides a Java interface for the Author UI and Reviewer UI, and a REST interface for the Local App Manager subcomponent. Moreover, it interacts with the Payment API for all the payment-related functionalities.

Service Data Management:

The Service Data Management is responsible for handling the data management for services in the Service Marketplace and provides the data to other components. It takes care of the license status validation for each single service and is therefore used by the Service Runtime Environment (see Section 6.1). In addition to that, the Service Data Management communicates with the Payment API in order to claim payment for a particular service. Furthermore, it retrieves information from the Monitoring component (see Section 6.2.5.2); this information can then be presented to third party service developers or authors through the corresponding user interfaces, showing the non-functional performance and other statistics of the services.

App Registry:

The App registry is a server-side database that stores all the information related to apps, including the app bundles and all the information associated with an app like the description, images, keywords reviews, or licensing information.

The App Registry provides all the needed methods for creating, modifying and accessing information about apps in the Cloud-based Information Infrastructure.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 324 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

Payment API:

The payment of SIMPLI-CITY apps and services will be handled internally by the Android System with the "Google Play In-app Billing API", which is usable since the SIMPLI-CITY Application Runtime Environment itself is an app running in the Android base system. The payment itself is then being handled with Google Wallet, which is a system by Google that processes payments.

To create a license for a particular user or an app, which is allowed to call a particular service, corresponding interfaces are defined in the following section.

6.5.5 Interfaces

This section describes the interfaces of the App and Service Marketplaces. The first and second subsections contain interfaces of the App Marketplace, whereas the third and fourth are focused on the Payment API subcomponent of the Service Marketplace, which bundles all functionalities of the Service Marketplace provided to external (i.e., non-marketplace) software components of SIMPLI-CITY.

The App Marketplace provides different methods to manage apps. The main subcomponent of the App Marketplace is the App Data Management subcomponent, which is the link between the different web UIs, the subcomponents of the PMA, and the App Registry. It provides different Java interfaces to be used by the Author and Reviewer UIs, and a RESTful interface to be used by the Local App Manager subcomponent of the PMA.

6.5.5.1 Java Interfaces of the App Marketplace

This section describes the Java interface provided by the App Data Management subcomponent. This interface is used by the Author UI and the Reviewer UI subcomponents of the App Marketplace. Figure 25 represents the class diagram of the App Data Management subcomponent. Only externally available interfaces are represented. It also shows the class diagram of the App class, returned in the getApp method of the App Data Management subcomponent.

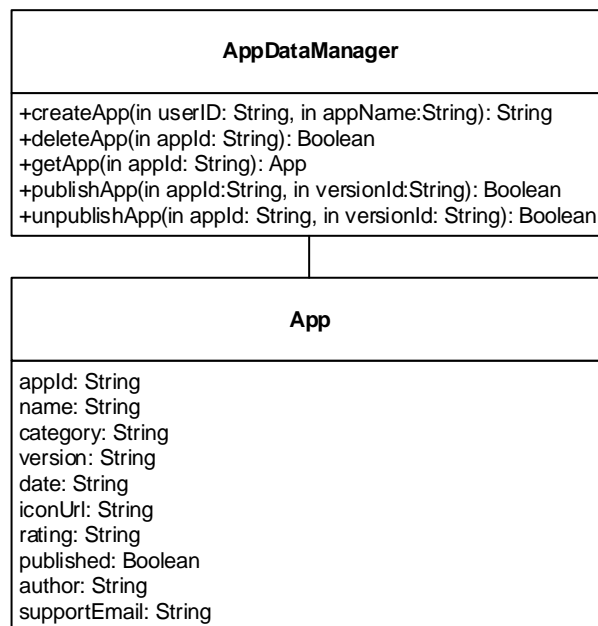


Figure 25: Class Diagrams of App Data Management Subcomponent

6.5.5.1.1 Create App

When the user creates an app using the Author UI, different parameters of the app have to be specified. Once all these parameters have been specified, different Java methods are executed. The first one permits to create the app in the App Registry. In this Java method, the main parameters of the app are specified, like the user that creates the app and the name of the app. The method returns the identifier of the app created, that will be used in the following Java calls for saving further information of the app.

Parameters:

- **userId:** Specifies the user identifier of the developer creating the app
- **appName:** Defines the name of the app to be created

Return Value:

Returns the ID of the created app.

Error Handling:

An `AppCreationException` is thrown in case of error during app creation.

Remarks:

`hibernateDao` is an internal Java class of the MyMarket solution.

Listing 220: Source Code Example – Create App

```

/**
 * @param userId, specifies the user identifier of the developer creating the app
 * @param appName, defines the name of the app to be created
 * @return appId, app identifier
 * @throws AppCreationException, if an error occurs during app creation
 */
String createApp(String userId, String appName) throws AppCreationException {
    //...

    String appId = hibernateDao.createApp(userId, appName);
    return appId;
}

```

6.5.5.1.2 Delete App

This method will delete an app from the App Marketplace. This method is called by the Author UI when the developer selects the option to delete an app.

Parameters:

- appId: App identifier

Return Value:

Returns a Boolean indicating if the app deletion has been successful.

Error Handling:

An AppDeletionException is thrown in case of error during app deletion.

Remarks:

hibernateDao is an internal Java class of the MyMarket solution.

Listing 221: Source Code Example – Delete App

```

/**
 * @param appId, app identifier
 *
 * @return appDeleted, app identifier
 * @throws AppDeletionException, if an error occurs during app deletion
 */
Boolean deleteApp(String appId) throws AppDeletionException {
    //...

    Boolean appDeleted = hibernateDao.deleteApp(appId);
    return appDeleted;
}

```

6.5.5.1.3 Get App

This method will return the information of an app stored in the App Marketplace. This method is used by the Author UI and the Reviewer UI in order to display the information of an app.

Parameters:

- appId: App identifier

Return Value:

Returns the app object containing all the information of an app. This app object has all the parameters that contain information of the app, e.g., name of the app, version identifier, etc. The App class is described in Section 6.5.6.3.

Error Handling:

An AppSearchException is thrown in case of error during app search.

Remarks:

hibernateDao is an internal Java class of the MyMarket solution.

Listing 222: Source Code Example – Get App

```
/**
 * @param appId, app identifier
 *
 * @return app, app object
 * @throws AppSearchException, if an error occurs during app search
 */
App getApp(String appId) throws AppSearchException {
    //...
    App app = hibernateDao.getApp(appId);
    return app;
}
```

6.5.5.1.4 Publish Version of an App

Once an app has been approved within the review process, it is ready for publication in the App Marketplace. The developer can then submit the app for publication by clicking the corresponding button in the Author UI, which calls the method described in this section.

Parameters:

- appId: App identifier
- versionId: Version identifier of the app

Return Value:

Returns a Boolean indicating if the publication has been successful.

Error Handling:

An AppPublicationException is thrown in case of error during app publication.

Remarks:

hibernateDao is an internal Java class of the MyMarket solution.

Listing 223: Source Code Example – Publish an App

```
/**
 * @param appId, App identifier
 *
 * @return versionId, Version identifier of the app
 * @throws AppPublicationException, if an error occurs during app publication
 */
Boolean publishApp(String appId, String versionId) throws AppPublicationException {
//...

    Boolean appPublished = hibernateDao.publishApp(appId, versionId);
    return appPublished;
}
```

6.5.5.1.5 Unpublish Version of an App

This method permits to unpublish the version of an app already published in the App Marketplace.

Parameters

- appId: App identifier
- versionId: Version identifier of the app

Return Value:

Returns a Boolean indicating if the unpublication has been successful.

Error Handling:

An AppUnpublicationException is thrown in case of error during app unpublication.

Remarks:

hibernateDao is an internal Java class of the MyMarket solution.

Listing 224: Source Code Example – Unpublish Version of an App

```
/**
 * @param appId, App identifier
 *
 * @return versionId, Version identifier of the app
 * @throws AppUnpublicationException, if an error occurs during app unpublication
 */
Boolean unpublishApp(String appId, String versionId) throws
AppUnpublicationException {
//...

    Boolean appUnpublished = hibernateDao.unpublishApp(appId, versionId);
    return appUnpublished;
}
```

6.5.5.2 RESTful Interfaces of the App Marketplace

This section described the RESTful interface provided by the App Management Data subcomponent. This interface will be used by the Local App Manager subcomponent of the PMA in order to communicate with the server-side subcomponents of the App Marketplace. These interfaces are already implemented in the current version of the MyMarket App Marketplace. The modification of these interfaces is not foreseen within the project although in some cases they may not use the best implementation approach, because these interfaces are already working and there is a low priority for this modification. Instead, the resources for the development of the App Marketplace will be dedicated to adapt the MyMarket solution to be used within SIMPLI-CITY and to add or modify the required functionalities which are described in Section 6.5.3.

6.5.5.2.1 Information of an App

The following RESTful methods provide detailed information of an app. The method accepts as a parameter the identifier of the app.

Table 72: RESTful Interface Description – Information of an App

Method	GET	URL	\$API_ROOT/market/app/:appId				
Description	Provides information of the last version of a published app						
Parameter	appld	Required	yes	Possible Values	any string	Description	Id of the app
Example URL	\$API_ROOT/market/app/45						
Response	HTTP status code + JSON object						
HTTP Status Code		Required	yes	Possible Values	200	Description	OK
					400		Invalid parameters
					404		App not found
JSON Object	http://SIMPLI-CITY.eu/AppMarketPlace/JSON-Schema/App						
JSON Attribute	appld	Required	yes <th>Possible Values</th> <td>any string</td> <th>Description</th> <td>The ID of the app</td>	Possible Values	any string	Description	The ID of the app
JSON Attribute	name	Required	yes <th>Possible Values</th> <td>any string</td> <th>Description</th> <td>The name of the app</td>	Possible Values	any string	Description	The name of the app
JSON Attribute	category	Required	yes <th>Possible Values</th> <td>any string</td> <th>Description</th> <td>The category of the app</td>	Possible Values	any string	Description	The category of the app
JSON Attribute	version	Required	yes <th>Possible Values</th> <td>any string</td> <th>Description</th> <td>The id of the version of the app</td>	Possible Values	any string	Description	The id of the version of the app
JSON Attribute	date	Required	yes <th>Possible Values</th> <td>any string</td> <th>Description</th> <td>The date of the publication</td>	Possible Values	any string	Description	The date of the publication
JSON Attribute	iconUrl	Required	yes <th>Possible Values</th> <td>any string</td> <th>Description</th> <td>The url of the icon of the app</td>	Possible Values	any string	Description	The url of the icon of the app
JSON Attribute	rating	Required	yes <th>Possible Values</th> <td>any string</td> <th>Description</th> <td>The rating of the app</td>	Possible Values	any string	Description	The rating of the app
JSON Attribute	published	Required	yes <th>Possible Values</th> <td>boolean</td> <th>Description</th> <td>If the app is published</td>	Possible Values	boolean	Description	If the app is published
JSON Attribute	author	Required	yes <th>Possible Values</th> <td>any string</td> <th>Description</th> <td>The author of the app</td>	Possible Values	any string	Description	The author of the app
JSON Attribute	supportEmail	Required	yes <th>Possible Values</th> <td>any string</td> <th>Description</th> <td>Support email of the author</td>	Possible Values	any string	Description	Support email of the author
Example Response	HTTP/1.1 200 OK						

The JSON response message will contain the detailed information of an app. It follows the JSON Schema described in Listing 237.

Listing 225: JSON Example: App Information

```
{
  "appId": "1cbe3af0-8378-11e3-baa7-0800200c9a66",
  "name": "Traffic restrictions",
  "category": "Navigation",
  "version": "1.1",
  "date": "15/10/2013",
  "iconUrl": "http://SIMPLI-CITY.eu/AppMarketPlace/app/1cbe3af0-8378-11e3-baa7-0800200c9a66/icon/icon.png",
  "rating": "4.2",
  "published": true,
  "author": "TrafficrestrictionsS.L.",
  "supportEmail": "support@trafficrestrictions.com"
}
```

6.5.5.2.2 Information of an App by Version

The following method of the RESTful interface provides the detailed information of a concrete version of an app, accepting as parameters the identifier of the app and the identifier of the version.

Table 73: RESTful Interface Description – Information of an App by Version

Method	GET	URL	\$API_ROOT/market/app/:appId/:versionNumber				
Description	Provides information of a specific version of a published app						
Parameter	appld	Required	yes	Possible Values	any string	Description	Id of the app
Parameter	versionNumber	Required	yes	Possible Values	any string	Description	Version umber
Example URL	\$API_ROOT/market/app/45/2.0.1						
Response	HTTP status code + JSON object						
HTTP Status Code		Required	yes	Possible Values	200	Description	OK
					400		Invalid parameters
					404		App not found
JSON Object	http://SIMPLI-CITY.eu/AppMarketPlace/JSON-Schema/App						
JSON Attribute	appld	Required	yes	Possible Values	any string	Description	The ID of the app
JSON Attribute	name	Required	yes	Possible Values	any string	Description	The name of the app
JSON Attribute	category	Required	yes	Possible Values	any string	Description	The category of the app
JSON Attribute	version	Required	yes	Possible Values	any string	Description	The id of the version of the app
JSON Attribute	date	Required	yes	Possible Values	any string	Description	The date of the publication
JSON Attribute	iconUrl	Required	yes	Possible Values	any string	Description	The url of the icon of the app
JSON Attribute	rating	Required	yes	Possible Values	any string	Description	The rating of the app
JSON Attribute	published	Required	yes	Possible Values	boolean	Description	If the app is published
JSON Attribute	author	Required	yes	Possible Values	any string	Description	The author of the app
JSON Attribute	supportEmail	Required	yes	Possible Values	any string	Description	Support email of the author
Example Response	HTTP/1.1 200 OK						

The response JSON uses the schema described in Listing 237. An example is provided in Listing 225.

6.5.5.2.3 Get List of Installed Apps

The following RESTful method provides the list of installed apps in the device, because this information is stored in the App Registry. This method is used by the device to check the list of installed apps.

Table 74: RESTful Interface Description – Get List of Installed Apps

Method	GET	URL	\$API_ROOT/market/installedapps?:deviceId&:sorting&:start&:elem				
Description	Provides the list of installed apps in the device that performs the request						
Parameter	deviceId	Required	yes	Possible Values	any string	Description	Id of the device
Parameter	sorting	Required	no	Possible Values	dating	Description	Date sorting
					rating		Rating sorting
Parameter	start	Required	no	Possible Values	any integer	Description	Start of the pagination
Parameter	elements	Required	no	Possible Values	any integer	Description	Number of elements in response
Example URL	\$API_ROOT/market/installedapps?start=30						
Response	HTTP status code + JSON object						
HTTP Status Code		Required	yes	Possible Values	200	Description	OK
					400		Invalid parameters
JSON Object	http://SIMPLI-CITY.eu/AppMarketPlace/JSON-Schema/AppList						
JSON Attribute	apps	Required	yes	Possible Values	array	Description	List of apps
Example Response	HTTP/1.1 200 OK						

The JSON response will include a list of apps. Listing 226 shows a list of installed apps with the information of one app. It uses the schema described in Listing 241.

Listing 226: JSON Example Response – List of Installed Apps

```
{
  "apps": [
    {
      "appId": "1cbe3af0-8378-11e3-baa7-0800200c9a66",
      "name": "Traffic restrictions",
      "category": "Navigation",
      "version": "1.1",
      "date": "15/10/2013",
      "iconUrl": "http://SIMPLI-CITY.eu/AppMarketPlace/app/1cbe3af0-8378-11e3-baa7-0800200c9a66/icon.png",
      "rating": "4.2",
      "published": true,
      "author": "TrafficrestrictionsS.L.",
      "supportEmail": "support@trafficrestrictions.com"
    }
  ]
}
```

6.5.5.2.4 Binary of an App

The following method of the RESTful interface returns the binary of a concrete version of an app. As request message, the ID of the app and of the version is needed. The response will be the binary data of the app.

Table 75: RESTful Interface Description – Binary of an App

Method	GET	URL	\$API_ROOT/market/binary/:appId/:versionId				
Description	Provides the binary of an app						
Parameter	appld	Required	yes	Possible Values	any string	Description	Id of the app
Parameter	versionId	Required	yes	Possible Values	any string	Description	Version of the app
Example URL	\$API_ROOT/market/binary/43/1.0.2						
Response	HTTP status code + binary data						
HTTP Status Code		Required	yes	Possible Values	200	Description	OK
					400		Invalid parameters
					404		Application not found
Example Response	HTTP/1.1 200 OK						

6.5.5.2.5 App Search

This section contains a set of RESTful methods that allow searching for apps. They are used in different cases, but follow the same response format as depicted in Listing 227, which provides a valid example for all interfaces presented in Table 76 to Table 79. In all of them the response will be a JSON structure containing a list of apps. The schema is shown in Listing 241 and will not be depicted for every interface description to reduce redundancy of already given information. The first method permits a generic search of apps, which accepts different parameters in the query.

Table 76: RESTful Interface Description – App Search

Method	GET	URL	\$API_ROOT/market/apps?:country&:category&:hasPromoImage&:sorting&:start&:elements			
Description			Provides the list of apps that match the search parameters			
Parameter	country	Required	no	Possible Values	any integer	Description Country
Parameter	category	Required	no	Possible Values	any integer	Description Category
Parameter	hasPromoImage	Required	no	Possible Values	yes no	Description Select apps with at least one image Select apps without an image
Parameter	sorting	Required	no	Possible Values	dating rating	Description Date sorting Rating sorting
Parameter	start	Required	no	Possible Values	any integer	Description Start of the pagination
Parameter	elements	Required	no	Possible Values	any integer	Description Number of elements in response
Example URL	\$API_ROOT/market/apps					
Response	HTTP status code + JSON object					
HTTP Status Code		Required	yes	Possible Values	200 400	Description OK Invalid parameters
JSON Object	http://SIMPLI-CITY.eu/AppMarketPlace/JSON-Schema/AppList					
JSON Attribute	apps	Required	yes	Possible Values	array	Description List of apps
Example Response	HTTP/1.1 200 OK					

The interface in Table 77 is used for a refined search of apps by category.

Table 77: RESTful Interface Description – App Search by Category

Method	GET	URL	\$API_ROOT/market/apps/category/:category?:sorting&:start&:elements			
Description			Provides the list of apps of a category			
Parameter	category	Required	yes	Possible Values	any integer	Description Category
Parameter	sorting	Required	no	Possible Values	dating rating	Description Date sorting Rating sorting
Parameter	start	Required	no	Possible Values	any integer	Description Start of the pagination
Parameter	elements	Required	no	Possible Values	any integer	Description Number of elements in response
Example URL	\$API_ROOT/market/apps/category/2					
Response	HTTP status code + JSON object					
HTTP Status Code		Required	yes	Possible Values	200 400	Description OK Invalid parameters
JSON Object	http://SIMPLI-CITY.eu/AppMarketPlace/JSON-Schema/AppList					
JSON Attribute	apps	Required	yes	Possible Values	array	Description List of apps
Example Response	HTTP/1.1 200 OK					

The interface in Table 78 is used for a refined search of apps by words.

Table 78: RESTful Interface Description – App Search by Words

Method	GET	URL	\$API_ROOT/market/apps/search/:words?:sorting&:start&:elements			
Description			Provides the list of apps that contain in their name some of the words of the search			
Parameter	words	Required	yes	Possible Values	any string	Description Words acting as a filter within the name of the app
Parameter	sorting	Required	no	Possible Values	dating rating	Description Date sorting Rating sorting
Parameter	start	Required	no	Possible Values	any integer	Description Start of the pagination
Parameter	elements	Required	no	Possible Values	any integer	Description Number of elements in response
Example URL	\$API_ROOT/market/apps/search/routing					
Response	HTTP status code + JSON object					
HTTP Status Code		Required	yes	Possible Values	200 400	Description OK Invalid parameters
JSON Object	http://SIMPLI-CITY.eu/AppMarketPlace/JSON-Schema/AppList					
JSON Attribute	apps	Required	yes	Possible Values	array	Description List of apps
Example Response	HTTP/1.1 200 OK					

The interface in Table 79 is used during reviewing time, and provides the list of apps that are under review. It is used by the mobile devices of reviewers.

Table 79: RESTful Interface Description – List of Apps under Review

Method	GET	URL	\$API_ROOT/market/apps/testing?:sorting&:start&:elements			
Description			Provides the list of apps that are under the testing / review process			
Parameter	sorting	Required	no	Possible Values	dating rating	Description Date sorting Rating sorting
Parameter	start	Required	no	Possible Values	any integer	Description Start of the pagination
Parameter	elements	Required	no	Possible Values	any integer	Description Number of elements in response
Example URL	\$API_ROOT/market/apps/testing/					
Response	HTTP status code + JSON object					
HTTP Status Code		Required	yes	Possible Values	200 400	Description OK Invalid parameters
JSON Object	http://SIMPLI-CITY.eu/AppMarketPlace/JSON-Schema/AppList					
JSON Attribute	apps	Required	yes	Possible Values	array	Description List of apps
Example Response	HTTP/1.1 200 OK					

In all the cases of app search included in this section, the JSON response will include a list of apps. The following listing shows an example returned during an app search operation.

Listing 227: JSON Example – List of Apps of a Search

```
{
  "apps": [
    {
      "appId": "1cbe3af0-8378-11e3-baa7-0800200c9a66",
      "name": "Traffic restrictions",
      "category": "Navigation",
      "version": "1.1",
      "date": "15/10/2013",
      "iconUrl": "http://SIMPLI-CITY.eu/AppMarketPlace/app/1cbe3af0-8378-11e3-baa7-0800200c9a66/icon/icon.png",
      "rating": "4.2",
      "published": true,
      "author": "TrafficrestrictionsS.L.",
      "supportEmail": "support@trafficrestrictions.com"
    }
  ]
}
```

6.5.5.3 RESTful Interfaces for the Payment API

Analogue to the corresponding Java interfaces from Section 6.5.5.4, the Payment API also provides RESTful interfaces to create and validate a license and to generate an invoice for an already granted license.

6.5.5.3.1 Create License

The create license RESTful interface takes a service ID, for which the license shall be created, a user or app ID, i.e., for whom it should be granted as parameters and a filled out license model as body. The return value is a valid license. Example input and output messages can be found in Listing 228 and Listing 229. The corresponding JSON schema can be found in Listing 208.

Table 80: RESTful Interface Description – Creating a License

Method	POST	URL	\$API_ROOT/paymentAPI/createLicense?:serviceID&:userID				
Description	Creates a valid License for a user or an app.						
Parameter	serviceID	Required	Yes	Possible Values	any string	Description	Id of the service
Parameter	userID	Required	Yes	Possible Values	any string	Description	Id of the app or user
JSON Object	http://SIMPLI-CITY.eu/ServiceRegistry/JSON-Schema/License						
JSON Attribute	serviceID	Required	yes	Possible Values	128 bit UUID	Description	Specifies the wanted service
JSON Attribute	userID	Required	yes	Possible Values	128 bit UUID	Description	Specifies the license owner
JSON Attribute	freeService	Required	yes	Possible Values	boolean	Description	Defines if the service is free of charge or not
JSON Attribute	usageCosts	Required	no	Possible Values	object	Description	Defines the usage costs for the defined service
JSON Attribute	unit	Required	no	Possible Values	string (see description)	Description	Defines the units of the costs, e.g. "per invocation", "per day", "per week", "per month", "per year"
JSON Attribute	cost	Required	no	Possible Values	integer	Description	Defines the costs which apply according the defined unit
JSON Attribute	limitations	Required	no	Possible Values	object	Description	Defines a limitation for the payment model
JSON Attribute	maxInvocationTimeUnit	Required	no	Possible Values	string (see description)	Description	Defines the unit for the limitation criteria, e.g. "per invocation", "per day", "per week", "per month", "per year"
JSON Attribute	maxInvocations	Required	no	Possible Values	integer	Description	Defines the max invocations
Example URL	\$API_ROOT/paymentAPI/createLicense?serviceID=528739A0-F508-4551-A12A-04A9B51718D0&userID=exampleID						
Response	HTTP status code + JSON object						
HTTP Status Code		Required	yes	Possible Values	200 404	Description	License successfully created Service or user not found to be deleted not found
JSON Object	http://SIMPLI-CITY.eu/ServiceRegistry/JSON-Schema/StatusCode						
JSON Attribute	serviceID	Required	yes	Possible Values	128 bit UUID	Description	ID for updated service
JSON Attribute	statusCodeNum	Required	yes	Possible Values	number	Description	Update status code as number (same value as the HTTP status code)
JSON Attribute	statusCodeText	Required	yes	Possible Values	any string	Description	Update status code in form of text
Example Response	HTTP/1.1 200 OK						

Listing 228: JSON Example: Creation of a License (Input Message)

```
{
  "license": {
    "serviceID": "528739A0-F508-4551-A12A-04A9B51718D0",
    "userID": "62474782-H154-1256-C12C-14G6R32741R1",
    "usageCosts": {
      "unit": "per invocation",
      "costs": "0.1 €",
      "limitation": {
        "maxInvocations": "100",
        "maxInvocationTimeUnit": "per day"
      }
    }
  }
}
```


Listing 229: JSON Example: Creation of a License (Response Message)

```
{
  "serviceID": "89f47220-31bf-11e3-aa6e-0800200c9a66",
  "statusCodeNum": "200",
  "statusCodeText": "SUCCESS"
}
```

6.5.5.3.2 Validate License

Since it is possible that the developer of a service makes changes to it, the license model defined for that service could also change. However, once a license was granted, it remains valid for the defined time period. The Payment API provides a RESTful interface for validating an already granted license. The corresponding license model can be found in Listing 208.

Table 81: RESTful Interface Description – Validating a License

Method	POST	URL	\$API_ROOT/paymentAPI/validateLicense				
Description	Validates a license model for a particular service						
Parameter	none	Required		Possible Values		Description	
JSON Object	http://SIMPLI-CITY.eu/ServiceRegistry/JSON-Schema/License						
JSON Attribute	serviceID	Required	yes	Possible Values	128 bit UUID	Description	Specifies the wanted service
JSON Attribute	userID	Required	yes	Possible Values	128 bit UUID	Description	Specifies the license owner
JSON Attribute	freeService	Required	yes	Possible Values	boolean	Description	Defines if the service is free of charge or not
JSON Attribute	usageCosts	Required	no	Possible Values	object	Description	Defines the usage costs for the defined service
JSON Attribute	unit	Required	no	Possible Values	string (see description)	Description	Defines the units of the costs, e.g. "per invocation", "per day", "per week", "per month", "per year"
JSON Attribute	cost	Required	no	Possible Values	integer	Description	Defines the costs which apply according the defined unit
JSON Attribute	limitations	Required	no	Possible Values	object	Description	Defines a limitation for the payment model
JSON Attribute	maxInvocationTimeUnit	Required	no	Possible Values	string (see description)	Description	Defines the unit for the limitation criteria, e.g. "per invocation", "per day", "per week", "per month", "per year"
JSON Attribute	maxInvocations	Required	no	Possible Values	integer	Description	Defines the max invocations
Example URL	\$API_ROOT/paymentAPI/validateLicense						
JSON Attribute	HTTP header						
HTTP Status Code		Required	yes	Possible Values	200	Description	License valid
					400		License invalid
					404		No service found
Example Response	HTTP/1.1 200 OK						

Listing 230: JSON Example: Validation of a License (Input Message)

```
{
  "license": {
    "serviceID": "528739A0-F508-4551-A12A-04A9B51718D0",
    "userID": "62474782-H154-1256-C12C-14G6R32741R1",
    "usageCosts": {
      "unit": "per invocation",
      "costs": "0.1 €",
      "limitation": {
        "maxInvocations": "100",
        "maxInvocationTimeUnit": "per day"
      }
    }
  }
}
```


Listing 231: JSON Example: Validation of a License (Response Message)

```
{
  "serviceID": "89f47220-31bf-11e3-aa6e-0800200c9a66",
  "statusCodeNum": "200",
  "statusCodeText": "SUCCESS"
}
```

6.5.5.3.3 Generating an Invoice

A user can see how much he/she has to pay for an already granted license; the Payment API provides a RESTful interface which generates an invoice for the granted license. The input is a license for a passed time period and the result is an invoice in form of a JSON format, see Listing 238.

Table 82: RESTful Interface Description – Generating an Invoice

Method	GET	URL	\$API_ROOT/paymentAPI/generateInvoice?:serviceID				
Description	Generates an invoice for a particular license						
Parameter	serviceID	Required	yes	Possible values	128 bit UUID	Description	Unique service ID
JSON Object	http://SIMPLI-CITY.eu/ServiceRegistry/JSON-Schema/License						
JSON Attribute	serviceID	Required	yes	Possible Values	128 bit UUID	Description	Specifies the wanted service
JSON Attribute	userID	Required	yes	Possible Values	128 bit UUID	Description	Specifies the license owner
JSON Attribute	freeService	Required	yes	Possible Values	boolean	Description	Defines if the service is free of charge or not
JSON Attribute	usageCosts	Required	no	Possible Values	object	Description	Defines the usage costs for the defined service
JSON Attribute	unit	Required	no	Possible Values	string (see description)	Description	Defines the units of the costs, e.g. "per invocation", "per day", "per week", "per month", "per year"
JSON Attribute	cost	Required	no	Possible Values	integer	Description	Defines the costs which apply according the defined unit
JSON Attribute	limitations	Required	no	Possible Values	object	Description	Defines a limitation for the payment model
JSON Attribute	maxInvocationTimeUnit	Required	no	Possible Values	string (see description)	Description	Defines the unit for the limitation criteria, e.g. "per invocation", "per day", "per week", "per month", "per year"
JSON Attribute	maxInvocations	Required	no	Possible Values	integer	Description	Defines the max invocations
Example URL	\$API_ROOT/paymentAPI/generateInvoice?serviceID=528739A0-F508-4551-A12A-04A9B51718D0						
Response	HTTP header + text/JSON						
HTTP Status Code		Required	yes	Possible Values	200 400 404	Description	License valid License invalid No service found
JSON Object	http://SIMPLI-CITY.eu/ServiceRegistry/JSON-Schema/Invoice						
JSON Attribute	serviceID	Required	yes	Possible Values	128 bit UUID	Description	Specifies the wanted service
JSON Attribute	id	Required	yes	Possible Values	128 bit UUID	Description	Specifies the license id
JSON Attribute	timeperiod	Required	yes	Possible Values	string	Description	Defines the time period for the invoice
JSON Attribute	cost	Required	no	Possible Values	number	Description	Defines the costs which have to be paid
Example Response	HTTP/1.1 200 OK						

Listing 232: JSON Example: Generation of an Invoice (Input Message)

```
{
  "license": {
    "serviceID": "528739A0-F508-4551-A12A-04A9B51718D0",
    "userID": "62474782-H154-1256-C12C-14G6R32741R1W",
    "usageCosts": {
      "unit": "per invocation",
      "costs": "0.1 €",
      "limitation": {
        "maxInvocations": "100",
        "maxInvocationTimeUnit": "per day"
      }
    }
  }
}
```

Listing 233: JSON Example: Generation of an Invoice (Response Message)

```

{
  "invoice": {
    "serviceID": "528739A0-F508-4551-A12A-04A9B51718D0",
    "usageCosts": {
      "timePeriod": "30 days",
      "costs": "0.30 €"
    }
  }
}

```

6.5.5.4 Java Interfaces for Payment API

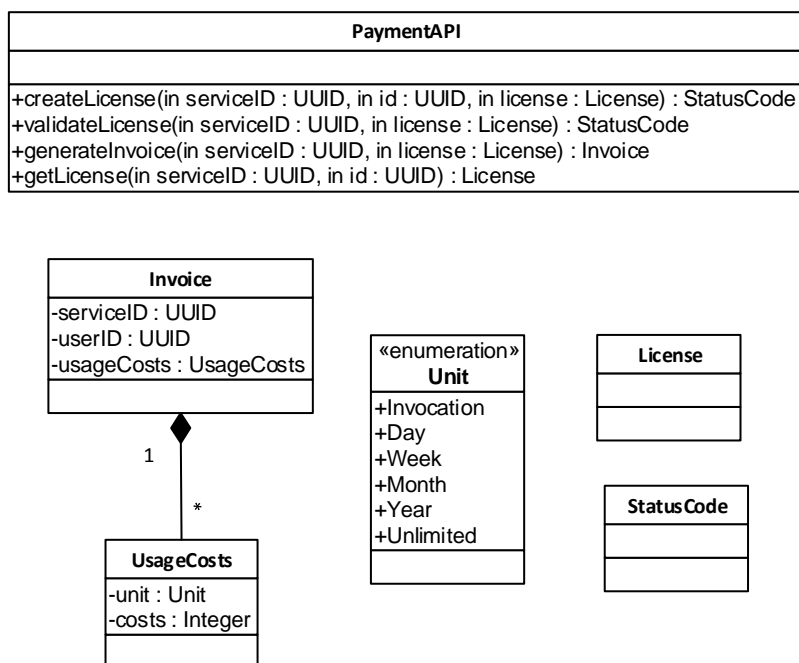


Figure 26: Class Diagrams of Payment API

Figure 26 shows the UML class diagram of the Java interfaces of the Payment API. Notably, the Java classes **License** and **StatusCode** have already been described in Section 6.4.6 and are therefore depicted in a simplified format.

6.5.5.4.1 Create License

Creating a license for a particular user or App includes two steps: First, a license will be created which is forwarded to the requester. This license can be seen as a contract and allows the caller to use the service under the defined conditions. Second, a payment order will be created using the Google Wallet API.

Parameters:

- **serviceID**: Specifies the service for which the license shall be created
- **id**: Defines the user or the app for whom the license shall be created
- **license**: The License which should be granted for a particular user or an App

Return Value:

Returns a `StatusCode` object. This object contains a field for the granted license if successful. Using this `licenseId`, the user can retrieve the license via the method `getLicense` (see Section 6.5.5.4.2).

Error Handling:

If an error occurs, the corresponding field in the `StatusCode` object will be filled in accordingly.

Remarks:

The license which is passed as a parameter is similar to the `LicenseModel` (see Section 6.4.6.1), i.e., the `LicenseModel` can be seen as a template and the license itself is the filled in template. The returned license differs in one field, i.e., a signed hash value which can be used for verifying the license.

Listing 234: Function Signature – Create a License

```
/**
 * @param serviceID, specifies the service for which the license should be
 *         created
 * @param id, defines the user or app ID for whom the license shall be created
 * @param license, specifies the license which should be created for the
 *         defined user
 * @return StatusCode, a status code will be returned indicating if the license creation was successful
 */
StatusCode createLicense(UUID serviceID, UUID id, License license);
//...

StatusCode statusCode = api.createLicense("an-example-ID", "user1", EXAMPLE_LICENSE);
```

6.5.5.4.2 Get License

The Payment API component also provides an interface for retrieving a license for a particular user or a particular app.

Parameters:

- `serviceID`: Specifies the service against which the license should be validated
- `id`: Specifies a user or an app ID

Return Value:

Returns the currently active and valid license for that user/app and service.

Error Handling:

If no license is specified, an exception will be thrown.

Remarks:

The JSON schema of the `License` class is defined in the Service Registry (see Section 6.4.6.1) and will be reused by the Payment API.

Listing 235: Function Signature – Retrieve a License

```

/**
 * @param serviceID, specifies the service for which the license should be
 *         validated
 * @param license, the license for the specified service
 *
 * @return boolean, returns true if the license is valid, otherwise false
 * @throws NoLicenseFoundException, if no valid license is defined for the specified user/App and
 *         service
 */
License retrieveLicense(UUID serviceID, UUID id);
//...
License license = api.retrieveLicense("an-example-ID".exampleAppID);

```

6.5.5.4.3 Validate License

The Payment API component also provides an interface for validating if a particular license is still valid for a particular service, i.e., if a user or an app is still authorized for using the service.

Parameters:

- serviceID: Specifies the service against which the license should be validated
- license: The license which should be validated

Return Value:

Returns a StatusCode object. This object contains information whether the license is still valid or not. Its definition can be found in Listing 217.

Error Handling:

If an error occurs, e.g., no license is specified at all or the service was not found, the particular field in the StatusCode object will be specified.

Remarks:

Since a license model may change for a particular service, the Payment API component is able to validate a given license. However, once the license was granted for a particular time period, it remains to be valid also in the case the license model for the service may change.

Listing 236: Function Signature – Validate a License

```

/**
 * @param serviceID, specifies the service for which the license should be
 *         validated
 * @param license, the license for the specified service
 *
 * @return boolean, returns true if the license is valid, otherwise false
 */
StatusCode validateLicense(UUID serviceID, License license);
//...
StatusCode valid = api.validateLicense("an-example-ID", EXAMPLE_LICENSE);

```

6.5.5.4.4 Generate Invoice

The Payment API component also provides an interface for generating an invoice for a particular license, i.e., how much a user had to pay for holding that license since it was granted.

Parameters:

The following parameters are expected:

- **serviceID**: Specifies the service for which the invoice should be created
- **license**: The license for the specified service

Return Value:

An invoice in form of a POJO⁹⁰ will be returned. The format is derived from the JSON format described in Section 6.5.6.

Error Handling:

If an error occurred or no invoice can be found, an `InvoiceErrorException` will be thrown.

Remarks

Since some license models are based on a “pay-as-you-go” model, i.e. the user has to pay per invocation; it should be possible for users to retrieve an invoice for the time period from the moment the license was granted to now.

Listing 237: Function Signature – Generate Invoice

```
/**
 * @param serviceID, specifies the service for which the invoice should be created
 * @param license, the license for the specified service
 *
 * @return invoice, returns the generated invoice or null
 * @throws InvoiceErrorException, if an error occurred*/
Invoice generateInvoice(UUID serviceID, License license) throws InvoiceErrorException;
//...
Invoice invoice = api.generateInvoice("an-example-ID", EXAMPLE_LICENSE);
```

⁹⁰ Plain Old Java Object, shorthand for a usual Java Object without special enterprise features

6.5.6 Content Format

6.5.6.1 Invoice Model JSON Schema

The invoice is used in the Service Marketplace component. An invoice can be requested by a particular user for a particular time period. It contains a field describing the corresponding service to whom it is assigned to and the resulting usage costs which are defined for a time period.

Listing 238: JSON Schema – Invoice Model

```
{
  "type": "object",
  "id": "http://SIMPLI-CITY.eu/CloudStorage/JSON-Schema/Invoice",
  "properties": {
    "serviceID": {
      "type": "string",
      "required": true
    },
    "id": {
      "type": "string",
      "required": true
    },
    "usageCosts": {
      "type": "object",
      "id": "http://SIMPLI-CITY.eu/ServiceRegistry/JSON-
Schema/usageCosts",
      "required": false,
      "properties": {
        "timePeriod": {
          "type": "string",
          "required": true
        },
        "cost": {
          "type": "integer",
          "required": true
        }
      }
    },
    "required": true
  }
}
```

The App Data Management subcomponent will provide a RESTful interface that will be used by the mobile apps in order to access the information related with apps. The data format used within the RESTful interface will be JSON. Section 6.5.5.2 describes the RESTful interface and the JSON format used by the App Data Management subcomponent, respectively.

6.5.6.2 Invoice Java Class

The class Invoice is the Java representation of the in Section 6.5.6.1 presented Invoice JSON Schema and contains the same fields.

Listing 239: Java Class: Invoice

```
public class Invoice {
    /**
     * unique service ID
     */
    private UUID serviceID;
    /**
     * unique user ID
     */
    private UUID id;
    /**
     * defines usage costs including a timespan
     */
    private UsageCosts usageCosts;

    static class UsageCosts {
        public enum Unit {Invocation, Day, Week, Month, Year, Unlimited};
        /**
         * defines the time unit for the costs
         */
        private Unit unit;
        /**
         * defines the costs per unit
         */
        private int costs;
    }

    //... getter() and setter()
}
```

6.5.6.3 App Information

Listing 240: Java Class – App

```
public class App {
    public java.util.UUID appId;
    public String name;
    public String category;
    public String version;
    public String date;
    public String iconUrl;
    public String rating;
    public Boolean published;
    public String author;
    public String supportEmail;

    //... getter() and setter()
}
```

6.5.6.4 License Model

The JSON schema for licenses as well as the license model are the same as described for the Service Registry (see Section 6.4.6.1).

6.5.6.5 List of Apps

This JSON object defines a list of apps. It contains a list of app objects as defined in Section 6.5.6.1. This object is used in different cases, as it is shown in Section 6.5.6.3, including the one that provides the list of installed apps or for app search.

Listing 241: JSON Schema – List of Apps

```
{
  "type": "object",
  "id": "http://SIMPLI-CITY.eu/AppMarketPlace/JSON-Schema/AppList",
  "properties": {
    "apps": {
      "type": [
        "App",
        "array"
      ],
      "required": true
    }
  }
}
```

6.5.7 Summary

As seen in the comparison and selection sections related with the App Marketplace, the MyMarket solution is chosen for its usage within SIMPLI-CITY. The main reason for using this solution is that it is the only solution that is focused on providing the functionalities needed by the SIMPLI-CITY App Marketplace, including the processes of uploading apps by the developers, reviewing of apps by the reviewers, and browsing and downloading of apps by the end users. The MyMarket solution will be used as a basis of the App Marketplace, but further development will be needed in order to provide all the requirements of the App Marketplace and in order to integrate it within the different server-side and PMA-side components of SIMPLI-CITY.

As seen in the comparison and selection sections, there is no suitable solution for the Service Marketplace, hence this component will be written completely from scratch using the chosen technology with the selected framework that fits the needs for the SIMPLI-CITY Service Marketplace. The Go programming language with the Revel framework fits the requirements perfectly and will therefore be the technology for the Service Marketplace.

7 Technical Specification: Personal Mobility Assistant

7.1 Application Runtime Environment

In SIMPLI-CITY, users access SIMPLI-CITY services using the Personal Mobility Assistant (PMA), which can be an Android smartphone or a specialized device. The central software component on this device is the Application Runtime Environment. Joined with the Multimodal Dialog Interface (Section 7.2), it will be implemented as an Android app and is hosting all SIMPLI-CITY apps installed on a device. The Application Runtime Environment is responsible for every technical interaction between the user interface and the running SIMPLI-CITY apps on the device as well as every interaction between the SIMPLI-CITY apps and the components running in the SIMPLI-CITY Service Runtime Environment (e.g., services and other software components). Interactions between the user and the Application Runtime Environment in terms of the user interface are handled by the Multimodal Dialogue Interface (see Section 7.2).

7.1.1 Major Design Decisions

The Application Runtime Environment needs to handle all communication between the user interface and the SIMPLI-CITY apps running on the device, as well as all communication between the SIMPLI-CITY apps and the backend services running in the Service Runtime Environment. Hence, several components handle the communication with the respective targets (e.g., the Command Handler for the user interface/app communication or the Push Service for the app/service communication). These components wrap the access to the targets, i.e., a service on the device or on the web, to provide easy accessibility and functionality to apps. The specific components are described in detail in Section 7.1.3.2 since the Application Runtime Environment is written from scratch, as will be motivated during the technology comparison and technology selection (see Sections 7.1.2 and 7.1.3). Therefore, all the components depicted in the diagram (see Figure 28) have to be newly created for the Application Runtime Environment.

The technology comparison only covers possible frameworks for the target system. Since the Application Runtime Environment is created from scratch, the major design decisions contain higher-level topics than specific requirements from the Functional Specification, as there is no solution that can fulfil the requirements discussed in the Functional Specification (see deliverable D3.2.1, Section 7.1). These requirements are described and defined in Section 7.1.4 in the component-specific sections (e.g., Section 7.1.4.1 for the Message Handler). Thus, the following design decision section (Section 7.1.1) and the technology section (Section 7.1.3) are more focused on the evaluation which operating systems and implementation platforms should be used, rather than technologies to implement the different components that compose the Application Runtime Environment.

Operating System:

The operating system is one of the most important choices regarding the Application Runtime Environment since there are many requirements to the runtime itself. The chosen operating system of the device should be widely available to reach the highest possible customer range. The following figure will show the market share on the worldwide smartphone market to ease the choice for the operating system.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 345 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

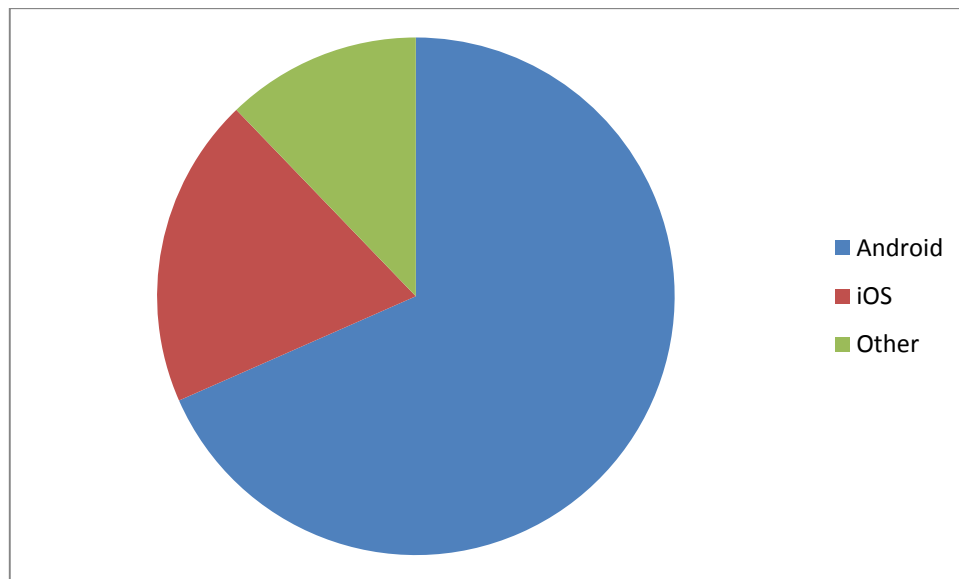


Figure 27: World-wide Smartphone Mobile OS Marketshare (January 2013)⁹¹

According to the statistic by Strategy Analytics, the highest market share of mobile operating systems is taken by the Android platform, which is a reason to choose Android. Another reason is the more commonly known programming language Java instead of the highly specialized programming language Objective-C, which is mandatory for the most valid Android contender, the iOS platform.

Other platforms are not considered in this section since their market share is too low to reach the targeted audience, and as they also fail the criteria in the following sections. Operating systems of other software and hardware solutions to bring mobility-related data to the road user, e.g., In-Vehicle Infotainment (IVI) solutions are also not regarded at this point of time as they are usually closed source and only available for a particular automotive brand.

Since SIMPLI-CITY is a research project, for the prototypical implementations, only Android will be taken into account. As defined in the Requirements Analysis Report (deliverable D2.3) and the Functional Specification (deliverable D3.2.1), support for iOS and other mobile operating systems may be added after the project lifetime if the SIMPLI-CITY PMA is going to be commercialized.

Operating System Extensibility and Hardware Access:

The target platform has to be open for hardware access to make use of internal sensors like the GPS module of the device and outputs like the audio device. Additionally, the Application Runtime Environment is used as an internal wrapper for SIMPLI-CITY apps. This feature will abstract hardware and operating system access for third party apps and keep developers from accessing and modifying the operating system or the hardware in an unexpected way. By abstracting these functionalities, the SIMPLI-CITY Application Runtime Environment can supervise and control third party access to the operating system or the hardware in order to reduce the possibility of the device being a bottleneck and as a

⁹¹ Source: <http://venturebeat.com/2013/01/28/android-captured-almost-70-global-smartphone-market-share-in-2012-apple-just-under-20/>

security feature to prevent misuse of operating system or hardware features by third party developers.

Error Handling Support:

The Application Runtime Environment needs to catch all operating system-specific errors as well as to replace the usual method of displaying them on the device. In case of highly severe errors like hardware failure, this functionality cannot be provided, since non-operating-system software cannot handle hardware failure. The error handling support therefore has to be able to catch and handle exceptions on the software part of the PMA, which contains the Application Runtime Environment itself and every third party app running in the Application Runtime Environment.

Push Service:

The Push Service is placed on the Server Side of Application Runtime Environment and provides apps running in the PMA with data from services running in the Service Runtime Environment. To fulfil the criteria given in Table 84, there is a need for a two-part-based Push Service, which means that the connection-establishment and maintenance is done by a different solution (namely the Base Push Messaging Solution) than the transfer of SIMPLI-CITY data, which will be handled by a custom solution. For the Base Push Messaging Solution, the use of an already existing technology is planned. The according technical comparison can be found in Section 7.1.2.4. The custom solution will make use of the connection negotiated by the Base Push Messaging Solution.

The complete Push Service works like a publish/subscribe service, which pulls changes from the Service Runtime Environment and pushes them to subscribed apps as soon as a data change happens.

7.1.2 Technology Comparison

7.1.2.1 Comparison Criteria

The selection criteria for the Application Runtime Environment technologies are defined in Section 7.1.7 of deliverable D3.2.1 (Functional Specification). In this section, a brief description of these parameters is presented.

Table 83: Description of the Comparison Criteria for the App Execution Framework

Parameter	Importance	Description
Inter-App Communication	++	To achieve a better way of exchanging data between the SIMPLI-CITY apps, developing technologies must allow an Inter Process Communication (IPC) mechanism by which data can be transferred and apps can influence each other.
Push Support	+	The Server Side of the PMA should be able to push well-formed messages from services to specific PMA devices.
Pull Support	++	Classic pull messages requested by the PMA should be handled by the Application Runtime Environment.

Error Handling Support	++	The Application Runtime Environment must be also able to catch exceptions during app execution and to report them to the marketplace for displaying in the author UI.
------------------------	----	---

The following comparison criteria are not part of the Functional Specification, since the split of the Push Service is introduced in this document. According to the Major Design Decisions for the Application Runtime Environment (see Section 7.1.1) the Base Push Messaging Solution is based on an existing solution, which needs to fulfil its own set of criteria, which is described in the following table.

Table 84: Description of the Comparison Criteria for the Base Push Messaging Solution

Parameter	Importance	Description
Android Support	++	Android support is necessary as this is the mobile platform chosen to be used within SIMPLI-CITY as stated in Section 7.1.1. Hence, the Base Push Messaging Solution needs to be easily integrable into and used on the Android platform.
XMPP Protocol	++	The XMPP ⁹² Protocol is a lightweight, high-speed protocol used for messaging between two parties. The speed is mandatory for this component, since the mobile client will not always have a fast connection to the server.
Payloads	+	A payload is data attached to a message. SIMPLI-CITY makes use of its own message formats (see deliverable D3.2.1) which should be attachable to the base Push Messaging solution.

7.1.2.2 Possible Technologies and Comparison

To develop the Application Runtime Environment component, it is necessary to select a framework that allows the execution of apps on Android devices. The possibilities include the Android Application Framework or one of several cross-platform development frameworks allowing hardware resources access. The descriptions and a comparison table of the possibilities are shown in Section 7.1.2.3. One particular functionality to be provided by the Application Runtime Environment is the support of push messages between backend services and apps. Hence, in Section 7.1.2.4, technologies for push messages will be compared. The results of these comparisons can be found in the next section (see Section 7.1.3).

⁹² XMPP is a standard for sending XML messages over a socket connection, see <http://xmpp.org/>

7.1.2.3 App Execution Framework – Comparison

Android Application Framework:

The Android Application Framework⁹³ is a set of APIs that allows developers to quickly and easily create apps for Android devices. It contains tools for designing User Interfaces and a documentation for the API. Essentially an Android app consists of activities (programs that the user interacts with), services (programs that run in the background or provide some function to other apps), and broadcast receivers (programs that catch information important to your app).

Xamarin.Mobile:

Xamarin.Mobile⁹⁴ is a library that exposes a single set of APIs for accessing common mobile device functionality across the iOS, Android, and Windows Phone platforms. This increases the amount of code developers can share across mobile platforms, making mobile app development more productive. Xamarin.Mobile currently abstracts the contacts, camera, and geo-location APIs across iOS, Android and Windows Phone platforms. Future plans include notifications and accelerometer services. It is based on Mono⁹⁵ and uses C#, so developers can reuse existing .NET Framework libraries that are successfully ported to the Mono Framework.

Titanium Framework:

The Titanium Framework⁹⁶ makes use of HTML5 for platform-independent mobile development. It provides developers with its own Integrated Development Environment (IDE). The Titanium Framework combines many mobile devices and platforms inside one central development platform to provide fast access to as many users as possible. Due to the focus on web technologies, it can only provide very basic and high-level functionality of hardware access APIs of the mobile devices.

PhoneGap Framework:

PhoneGap⁹⁷ is a mobile development framework that enables software developers to build apps for mobile devices using JavaScript, HTML5 and CSS3, instead of device-specific languages. The resulting apps are hybrid, meaning that they are neither truly native (because all layout rendering is done via web views instead of the platform's native UI framework) nor purely web-based (because they are not just web apps, but are packaged as apps for distribution and have access to native device APIs).

⁹³ <http://developer.android.com/guide/faq/framework.html>

⁹⁴ <http://xamarin.com/mobileapi>

⁹⁵ http://www.mono-project.com/Main_Page

⁹⁶ <http://www.appcelerator.com/titanium/>

⁹⁷ <http://phonegap.com/>

Table 85: Comparison of Technologies for App Execution Framework

Parameter	Importance	Android Application Framework	Xamarin	Titanium	PhoneGap
Generic Criteria					
Up-to-Datedness	+	10	10	6	8
Stability	+	10	10	8	8
Extensibility & Open Source/Standards	+	10	10	8	8
Familiarity	+/-	10	4	6	10
Performance	+/-	10	8	6	8
Interoperability	++	10	10	6	8
License	(e.g., Apache 2.0)	Proprietary	Proprietary	Apache 2.0, Proprietary	Apache 2.0
Specific Criteria					
Inter-App Communication	++	10	8	8	8
Push Support	+	10	10	8	10
Pull Support	++	10	10	8	10
Error Handling Support	++	10	10	10	10

7.1.2.4 Base Push Messaging Solution – Comparison

Apache ActiveMQ:

ActiveMQ⁹⁸ is a popular open source message queuing system by Apache released under the Apache 2.0 license. It supports diverse programming languages like C#, Java, PHP and many more. It also provides support for many different transport protocols like UDP, TCP, HTTP or multicast. It is based on Java and needs a Java Virtual Machine (JVM) to run on a server.

Google Cloud Messaging:

Google Cloud Messaging⁹⁹ (GCM) is a message queuing service by Google developed for Android devices, though it is available for other platforms as well. It allows developers to send lightweight messages as well as messages with a payload to Android devices of every kind. It also provides a functionality called “send to sync” which invokes a synchronisation process on the device. GCM uses the Extensible Messaging and Presence Protocol (XMPP)¹⁰⁰ for message transfer instead of the Advanced Message Queuing Protocol (AMQM), which makes messages much more lightweight and faster to transfer especially when having a slow connection on the device.

RabbitMQ:

RabbitMQ¹⁰¹ is a solution that implements the AMQM and is written in Erlang. Due to the choice of the functional programming language Erlang, RabbitMQ is a very fast message queuing service. Apart from the speed criterion, RabbitMQ does not fulfil the criteria set described in Table 84 as the AMQM protocol has already been evaluated as too heavyweight for the needed features (i.e., a small overhead is needed; otherwise the

⁹⁸ <http://activemq.apache.org/>

⁹⁹ <http://developer.android.com/google/gcm/index.html>

¹⁰⁰ XMPP is a standard for sending XML messages over a socket connection, see <http://xmpp.org/>

¹⁰¹ <http://www.rabbitmq.com/>

speed of messaging especially on slow connections is impeded). RabbitMQ is maintained by the commercial vendor of operating systems virtualization software VMWare¹⁰².

IronMQ:

IronMQ¹⁰³ is a service by the company IronIO, which provides message queuing in the cloud. The main benefits from using the cloud for message queuing are the instant availability and the abstraction of requests by services and apps into a cloud infrastructure, which simplifies dealing with performance and scalability. Due to the Software-as-a-Service (SaaS) approach, this solution has subscription fees, where the cheapest professional plan has a monthly cost of 495 US dollars.

Table 86: Comparison of Technologies for Base Push Messaging Solution

Parameter	Importance	Apache ActiveMQ	Google Cloud Messaging	RabbitMQ	IronMQ
Generic Criteria					
Up-to-Datedness	+	8	10	10	8
Stability	+	8	8	8	8
Extensibility & Open Source/Standards	+	6	10	6	8
Familiarity	+/-	4	8	4	4
Performance	+/-	6	10	8	8
Interoperability	++	8	10	6	6
License	(e.g., Apache 2.0)	Apache 2.0	Google API ToS	Mozilla Public License	None (Cloud Service)
Specific Criteria					
Android Support	++	4	10	6	2
XMPP Protocol	++	0	10	0	0
Payloads	+	8	2	6	8

7.1.3 Technology Selection

7.1.3.1 Selection for App Execution Framework

The Client Side software of the Application Runtime Environment (more precisely: the App Execution Framework) targets the PMA as host device and is therefore limited to the programming language and libraries that are offered by the publisher of the operating system of the device. In the case of SIMPLI-CITY, Android is chosen. Therefore, the Client Side software is developed using the Java programming language and the Android Application Framework. Every other framework described in the technology comparison (Section 7.1.2) is neither capable of providing the needed libraries nor of accessing the hardware layer in necessary degrees

7.1.3.2 Selection for Base Push Messaging Solution

After evaluating the aforementioned technologies for the Push Service (Section 7.1.2), GCM is chosen because of its lightweight implementation, its usage of XMPP and the implied synergies using the Android platform. The more heavyweight AMQM protocol, paid

¹⁰² <http://www.vmware.com/>

¹⁰³ <http://www.iron.io/mq>

services and services running in the cloud are reasons for other solutions not fulfilling the SIMPLI-CITY requirements. Since the App Execution Framework and GCM are both provided by Google, the integration of GCM is more effectively accomplishable than the integration of any other solution. The approach provided by GCM keeps the focus on the business logic implementation instead of implementation of basic infrastructure parts.

However, during the evaluation, it became obvious that GCM does not perfectly fit the requirements for the following reasons: GCM does not allow payloads over 4KB which makes transferring data from services hard, because GCM does not automatically split bigger data into several messages and is therefore unable to transfer SIMPLI-CITY relevant data on its own.

Hence, the Server Side part of the Application Runtime Environment consists of a hybrid solution with GCM as basic push messaging implementation and a wrapper to expose the messaging functionality to third party developers as well as to add functionality needed for the SIMPLI-CITY project. GCM only handles connection establishment and management to keep connections alive and to close connections, while the extended Push Service is responsible for sending and receipt of messages. Therefore, the chosen push messaging solution has to be as lightweight and fast as possible to keep the used bandwidth for these low-level tasks as low as possible. Also, since the Push Service is accessed by many apps and services simultaneously, the solution needs to have a high performance and reliability.

According to the criteria given in the last paragraph, the SIMPLI-CITY Push Service is realized using the Go programming language, as Go is a highly concurrent, high performance language for distributed architectures. Other choices would also be reasonable, but as there is no need for a certain technology base and as many languages and frameworks could be used to realize this component, a developer-driven choice is pursued. The main argument to choose Go is its specialization on being a fast backend programming language with an easy to use concurrency pattern that allows developers to focus on the application logic implementation needs instead of the more complex multi-threading patterns that are provided by programming languages like Java or C#, which promises working with this language to yield faster and more stable outputs.

7.1.3.3 Missing Elements and Implementation Needs

This section will cover every subcomponent of the Application Runtime Environment that needs to be developed or extended in the development phase of SIMPLI-CITY.

Push Service:

The Push Service is the only component on the Server Side of the Application Runtime Environment (see Figure 28) and offers push message functionality to all other components. As described above, the Push Service is a wrapper around GCM.

GCM provides an interface to the Android device which allows easy integration of push functionality. GCM supports two kinds of messages: send-to-sync messages and payload messages, which will either trigger a client action (send-to-sync message) or send up to 4 KB of data to the receiver (payload message). GCM is used for connection establishment and connection maintenance.

The specific implementation of a Push Service at the Server Side (of the Application Runtime Environment) will handle all messages with a payload or other data regarding the

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 352 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

SIMPLI-CITY system. This behaviour makes the Push Service flexible for developers since they are not responsible for trivial tasks like connection handling.

The complete solution for the Push Service will be developed as a fake Publish/Subscribe service, which will pull for changes from different sources (e.g. Context-Based Service Personalisation for sensor data or the Cloud-Based Information Infrastructure for historical and crowdsourced data) and populate them to subscribed Apps as soon as a data change happened.

Message Handler:

The Message Handler is the direct connection between the Push Service of the Application Runtime Environment and the PMA. An app has to subscribe itself to the Push Service to be able to receive messages from backend services running in the Service Runtime Environment. The Message Handler is also responsible for delivering messages to the correct recipients.

Error Handler:

Error handling is theoretically part of the Message Handler but is implemented separately to keep it transparent to the app developer. The difference to the generic Message Handler is that errors are predefined messages only sent to the App Marketplace UI via the ARE Controller. Hence, this is a static implementation of the Message Handler with only one function, which is the sending of error messages. Apart from custom error messages this subcomponent handles exceptions and errors with a higher severity (e.g., errors that can crash the app or corrupt the Application Runtime Environment) automatically.

ARE Controller:

The ARE Controller handles the inter-app communication inside the Application Runtime Environment instance. For this local communication, the Push Service is bypassed in order to reduce the overhead created by the transmission of data to the Server Side of the Application Runtime Environment.

Execution Manager:

The Execution Manager handles the execution of apps inside the Application Runtime Environment. This subcomponent needs to be able to manage all running apps and prepare newly started apps with the current context information. The Execution Manager is also responsible for getting information about running apps, as well as executing actions like closing apps and killing apps that are not responding anymore.

ARE Foundation Provider:

The ARE Foundation Provider provides apps with access to the basic functionalities of the Application Runtime Environment. This includes the Message Handler to communicate with services, the ARE Controller for inter-app communication, the Cloud-based Information Infrastructure for storing and retrieving data from the Local Key Storage and the Command Handler to handle voice controlled actions with the Multimodal Dialogue Interface (see Section 7.2).

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 353 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

Command Handler:

The Command handler is used for the communication from apps to the Multimodal Dialogue Interface. It dispatches responses from the apps to the MMDI, which will then take care of the interaction with the user (see Section 7.2).

7.1.3.4 Further Information and Conclusion from Technology Comparison

As can be seen from the discussion above, all listed requirements were very specific and no listed multi-platform framework for mobile apps was able to fulfil a critical mass of the requirements. Due to these facts, the Application Runtime Environment needs to be a completely custom solution not based on existing software. This implies a deep research background since the Application Runtime Environment has to fulfil a set of requirements from other components and has to be a well-designed and –implemented solution with a clean and well working APIs since third party app developers are supposed to make use of it. This can affect the error-proneness of technical and user-interaction-sourced errors and mistakes when using the device.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 354 / 435
http://www.simpli-city.eu/	Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201			

7.1.4 Component Structure

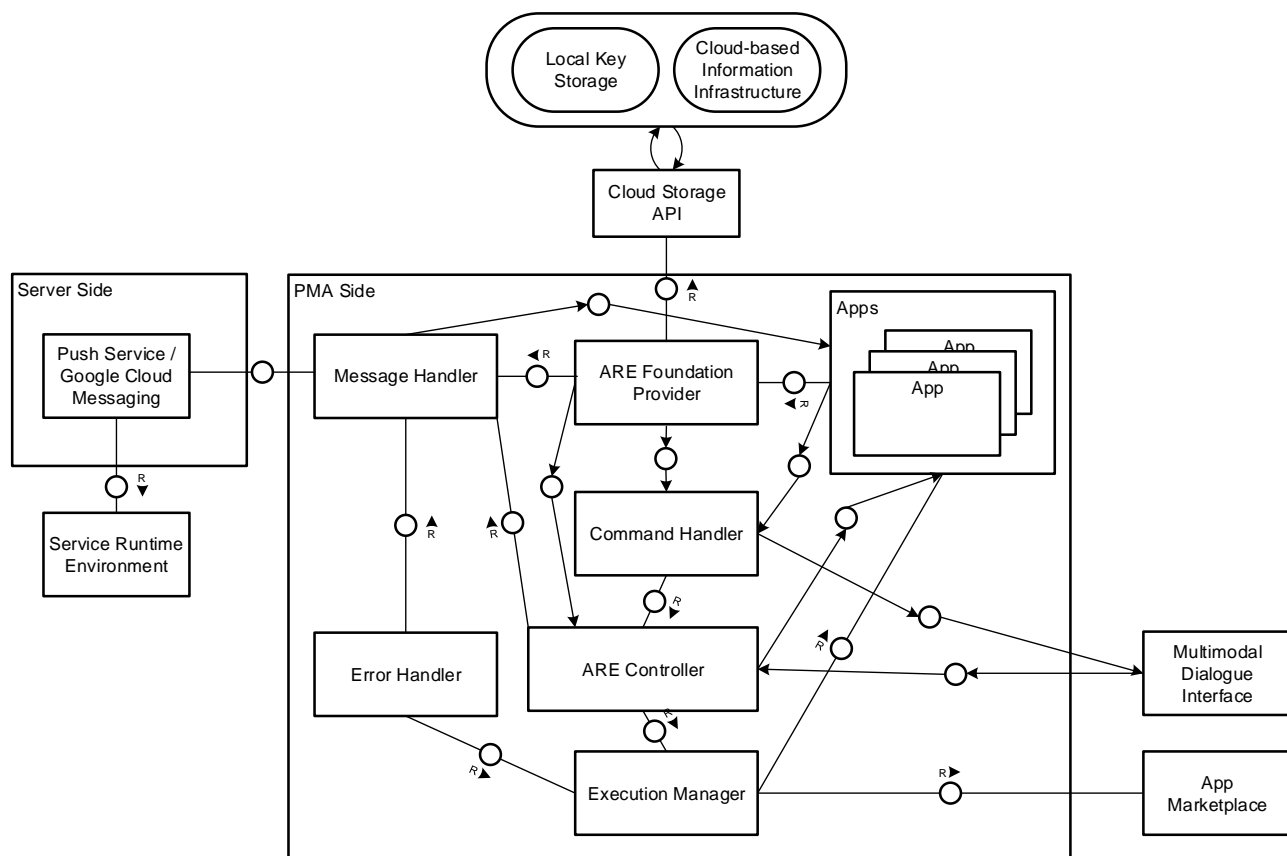


Figure 28: Component Structure of Application Runtime Environment

7.1.4.1 Message Handler

The Message Handler manages push and pull notifications via the Push Service on the Server Side of the Application Runtime Environment and provides a central communication center for apps. This component interacts with the Push Service via a REST Interface. The common format for the communication is described in Section 7.1.6. The main functionalities of the Message Handler are the sending and receiving of messages, this means it is the main communication point between apps and services.

7.1.4.2 ARE Foundation Provider

The ARE Foundation Provider will provide an API with core methods used by apps. It is the main connection point to apps and will also wrap the access to the Message Handler, the Cloud-based Information Infrastructure, the command Handler and the ARE Controller.

These functionalities are exposed via a Java API, since they will only be used on the PMA and Java is the recommended programming language used for the Android operating system.

7.1.4.3 Error Handler

The Error Handler catches app exceptions and reports them to the App Marketplace via the Execution Manager as depicted in Figure 28. The Error Handler is a specific

implementation of the Message Handler. Hence, it offers the same functionality but only sends and receives specifically defined messages, which are pre-specified error messages and serve as a wrapper for the native exceptions to make them more user-friendly and have them not crash the environment.

7.1.4.4 Command Handler

The Command Handler represents the connection to the Multimodal Dialogue Interface (MMDI) and handles the communication with the MMDI that is invoked by the Application Runtime Environment. It also provides the data needed by the MMDI to show the user interface. This includes all interactions that origin at an app. The Command Handler will format these interactions and messages into a format that is understandable by the MMDI.

7.1.4.5 ARE Controller

The ARE Controller controls the user interaction and inter-app communication facilities. It receives user input from the MMDI and dispatches them to the corresponding apps. Additionally, the ARE Controller initiates the sending of PMA-wide messages as well as messages between different apps on the PMA.

7.1.4.6 Execution Manager

The Execution Manager launches and handles apps during execution time. It is responsible for checking for app updates on the App Marketplace and will invoke the Local App Manager of the PMA Side of the Service and App Marketplaces (see Section 6.5) to install an update on the current device, since the Execution Manager is responsible for the execution of apps. The Execution Manager also forwards error messages to the App Marketplace as invoked by the Error Handler.

7.1.4.7 Push Service

This service allows services running in the Service Runtime Environment to notify apps in case of events. The Push Service handles the delivery of messages and distributes them to the different PMA instances (e.g., for informing an app installed on many PMA devices). The Push Service can also be used to inform a specific app of a specific user (e.g., to inform the user about upcoming traffic jams on the current route).

7.1.5 Interfaces

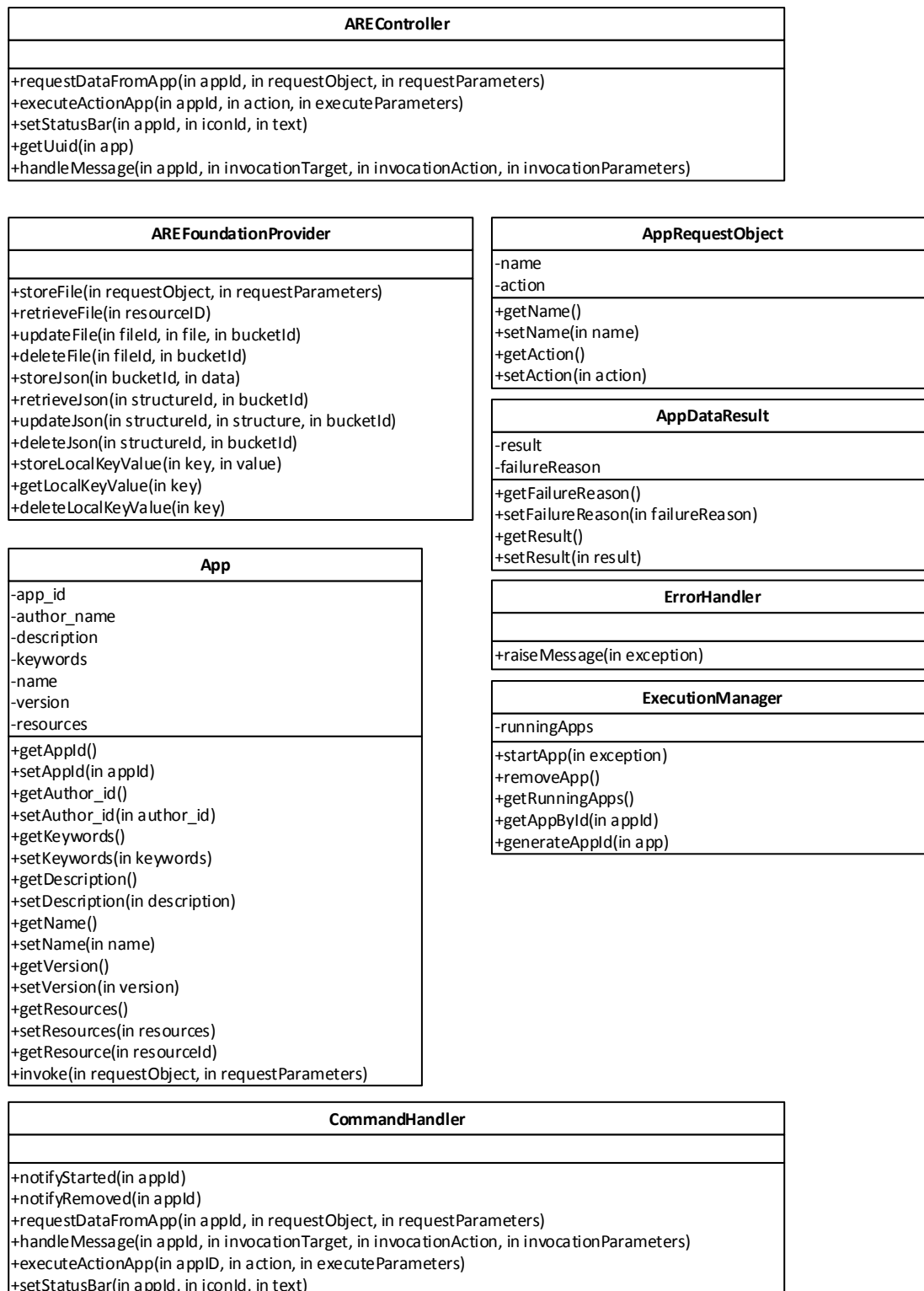


Figure 29: Class Diagrams of the Application Runtime Environment

The PMA Side of the Application Runtime Environment provides several interfaces to the Multimodal Dialogue Interface (MMDI) since it is the only other component running on the

PMA on its own. These interfaces will be provided as Java Interfaces and explained in detail in Section 7.1.5.1.

The Push Service on the Server Side is exposed via a REST Interface (see Section 7.1.5.2) so it is accessible for all third party services and applications as well as all other SIMPLI-CITY components.

7.1.5.1 Java Interfaces

7.1.5.1.1 Store File

This is the function to store a binary file via the Cloud Storage API. This function is called by apps that want to store files.

Parameters:

- bucketId: The id of the bucket where the file should be stored
- data: The data of the file that needs to be stored

Return Value:

The Cloud Storage API returns an identifier for the file in the specific bucket, which is used for further operations concerning the stored file.

Error Handling:

In case of an error, the returned value will be null and an error is sent to the Error Handler (see Listing 258).

Remarks:

There is no further information needed since the file is stored in a binary bucket and the identifier is used instead of a filename.

Listing 242: Source Code Example – Store File

```
/**
 * @param bucketId The ID of the bucket where the file should be stored
 * @param data The data of the file that needs to be stored
 * @return The Cloud Storage API returns an identifier for the file in the specific
 * bucket, which is used for further operations concerning the stored file.
 */
public String storeFile(String bucketId, CloudStorageFile file) {
    try
    {
        CloudStorage storage = CloudStorage.createConnection(bucketId);
        CloudStorageFile binaryFile = storage.storeFile(file);
        return binaryFile.getKey();
    }
    catch (Exception ex)
    {
        ErrorHandler.raiseMessage(ex);
        return null;
    }
}
```

7.1.5.1.2 Retrieve File

This function retrieves a binary file from the Cloud Storage according to the given id and the given bucket id.

Parameters:

- `fileId`: The id of the file to be retrieved
- `bucketId`: The bucket to receive the file from

Return Value:

The file from the Cloud Storage API is returned if the assigned id was found in the specific bucket.

Error Handling:

If the file was not found or the Cloud Storage API is not reachable, an error is sent to the Error Handler (see Listing 258) and the method returns a null value.

Remarks:

`CloudStorageFile` is an internal type of the Cloud Storage API and will be used for all operations concerning binary data in the Cloud-based Information Infrastructure.

Listing 243: Source Code Example – Retrieve File

```
/**
 * @param fileId the id of the file to be retrieved
 * @param bucketId the bucket to receive the file from
 * @return the file that was requested
 */
public CloudStorageFile retrieveFile(String fileId, int bucketId) {
    try
    {
        CloudStorage storage = CloudStorage.createConnection(bucketId);
        CloudStorageFile file = storage.getFile(fileId);
        return file;
    }
    catch (Exception ex)
    {
        ErrorHandler.raiseMessage(ex);
        return null;
    }
}
```

7.1.5.1.3 Update File

This function will update a file via the Cloud Storage API with new data, which overwrites a file with a specific id with a new file.

Parameters:

- **fileId:** The id of the file that will be updated
- **file:** The new file that contains the updates data
- **bucketId:** The id of the bucket where the file is saved in

Return Value:

Returns the boolean value “true” if the file was successfully updated and “false” if an error occurs.

Error Handling:

If the file was not found or could not be updated, an exception is raised to the ErrorHandler (see Listing 258) and the function returns the value “false”. The file is not changed for reasons of consistency and isolation if an error occurs.

Remarks:

CloudStorageFile is an internal type of the Cloud Storage API and will be used for all operations concerning binary data in the Cloud-based Information Infrastructure.

Listing 244: Source Code Example – Update File

```
/**
 * @param fileId the id of the file that will be updated
 * @param file the new file that contains the updates data
 * @param bucketId the bucket where the file is saved in
 * @return true when the file was successfully updated
 */
public boolean updateFile(String fileId, CloudStorageFile file, int bucketId) {
    try
    {
        CloudStorage storage = CloudStorage.createConnection(bucketId);
        CloudStorageFile upFile = storage.updateFile(fileId, file);
        return true;
    }
    catch (Exception ex)
    {
        ErrorHandler.raiseMessage(ex);
        return false;
    }
}
```


7.1.5.1.4 Delete File

This function will delete a file in a specific bucket according to the given file id.

Parameters:

- **fileId:** the id of the file that has to be removed
- **bucketId:** the id of the bucket the file is saved in

Return Value:

Returns the boolean value “true” if the file was successfully removed and “false” if there was an error in the removal process.

Error Handling:

If the file was not found or the connection to the bucket could not be established an exception is thrown and forwarded to the Error Handler component.

Remarks:

The file will not be temporarily saved in a recycle bin but is directly deleted; there is no way to revert this process.

Listing 245: Source Code Example – Store File

```
/**
 * @param fileId the id of the file that has to be removed
 * @param bucketId the bucket where the file is saved in
 * @return true if the file was successfully removed
 */
public boolean deleteFile(String fileId, int bucketId) {
    try
    {
        CloudStorage storage = CloudStorage.createConnection(bucketId);
        boolean deleteSuccess = storage.deleteFile(fileId);
        return deleteSuccess;
    }
    catch (Exception ex)
    {
        ErrorHandler.raiseMessage(ex);
        return false;
    }
}
```

7.1.5.1.5 Store JSON

This is the method to store structured data via the Cloud Storage API. Since JSON is a plain text data structure, it is used to store descriptive structures and serialized objects in a bucket for structured data. JSON is more developer-friendly than a storage optimized binary data structure, provides storage specific functions, its schema can be changed at runtime and it provides advanced querying features to third party developers.

Parameters:

- **bucketId:** The id of the bucket where the JSON object should be stored
- **data:** The data of the JSON object that needs to be stored

Return Value:

The Cloud Storage API returns a structureId string for the JSON object in the specific bucket, which is used for further operations concerning the stored object.

Error Handling:

In case of an error, an exception is raised and sent to the Error Handler (see Listing 258).

Remarks:

There is no further information needed since the structure is stored in a JSON bucket and the identifier is used instead of a filename.

Listing 246: Source Code Example – Store JSON

```
/**
 * @param bucketId The id of the bucket where the JSON object should be stored
 * @param data The data of the JSON object that needs to be stored
 * @return The Cloud Storage API returns an identifier (key) for the JSON object in
 the specific bucket, which is used for further operations concerning the stored
 object.
 */
public String storeJSON(String bucketID, CloudStorageJson data) {
    try
    {
        CloudStorage storage = CloudStorage.createConnection(bucketId);
        CloudStorageJSON jsonFile = storage.storeJson(data);
        return jsonFile.getKey();
    }
    catch (Exception ex)
    {
        ErrorHandler.raiseMessage(ex);
        return null;
    }
}
```

7.1.5.1.6 Retrieve JSON

This function tries to retrieve a JSON structure via the Cloud Storage API according to the given id and the given bucket id.

Parameters:

- **structureId:** The id of the JSON structure to be retrieved
- **bucketId:** The bucket to receive the file from

Return Value:

Returns the JSON structure from the Cloud Storage API if the assigned id was found in the specific bucket.

Error Handling:

If the JSON structure was not found or the Cloud Storage API is not reachable, an error is sent to the Error Handler (see Listing 258).

Remarks:

CloudStorageJSON is an internal type of the Cloud Storage API and will be used for all operations concerning structured JSON data in the Cloud-based Information Infrastructure.

Listing 247: Source Code Example – Retrieve JSON

```
/**
 * @param structureId the ID of the JSON structure to be retrieved
 * @param bucketId the bucket to receive the file from
 * @return the JSON structure that was requested
 */
public CloudStorageFile retrieveJSON(String structureId, int bucketId) {
    try
    {
        CloudStorage storage = CloudStorage.createConnection(bucketId);
        CloudStorageJSON json = storage.getJson(fileId);
        return json;
    }
    catch (Exception ex)
    {
        ErrorHandler.raiseMessage(ex);
        return null;
    }
}
```

7.1.5.1.7 Update JSON

This function will update a stored JSON structure with new data. Since JSON is an object notation language stored in plain text, there will not be any overwriting of data, because the structures are directly accessible and editable.

Parameters:

- **structureId:** The id of the JSON structure that will be updated
- **structure:** the new structure that contains the updates data
- **bucketId:** the bucket where the JSON structure is saved in

Return Value:

Returns the boolean value true if the structure was successfully updated and false if an error occurred.

Error Handling:

If the structure is not found or could not be updated an exception is handed to the ErrorHandler (see Listing 258) and the function returns the boolean value “false”. The JSON will be unchanged in this case for consistency reasons.

Remarks:

CloudStorageJSON is an internal type of the Cloud Storage API and will be used for all operations concerning structured JSON data in the Cloud-based Information Infrastructure.

Listing 248: Source Code Example – Update JSON

```
/**
 * @param structureId the id of the structure that will be updated
 * @param structure the new structure that contains the updated json
 * @param bucketId the bucket where the file is saved in
 * @return true when the structure was successfully updated
 */
public boolean updateJSON(String structureId, CloudStorageJSON structure, int
bucketId) {
    try
    {
        CloudStorage storage = CloudStorage.createConnection(bucketId);
        CloudStorageJSON file = storage.updateJson (structureId, structure);
        return true;
    }
    catch (Exception ex)
    {
        ErrorHandler.raiseMessage(ex);
        return false;
    }
}
```

7.1.5.1.8 Delete JSON

This function will delete a JSON structure in a specific bucket according to the given structure id.

Parameters:

- **structureId:** The id of the structure that has to be removed
- **bucketId:** The id of the bucket the structure is saved in

Return Value:

Returns true if the structure was successfully removed and false if there was an error in the removal process.

Error Handling:

If the structure was not found or the connection to the bucket could not be established, an exception is thrown and forwarded to the Error Handler component (see Listing 258).

Remarks:

CloudStorageJSON is an internal type of the Cloud Storage API and will be used for all operations concerning structured JSON data in the Cloud-based Information Infrastructure.

Listing 249: Source Code Example – Delete JSON

```
/**
 * @param structureId the id of the saved structure
 * @param bucketId the id of the bucket the structure is saved in
 * @return true if the structure was successfully removed
 */
public boolean deleteJSON(String structureId, int bucketId) {
    try
    {
        CloudStorage storage = CloudStorage.createConnection(bucketId);
        boolean deleteSuccess = storage.deleteJson(structureId);
        return deleteSuccess;
    }
    catch (Exception ex)
    {
        ErrorHandler.raiseMessage(ex);
        return false;
    }
}
```

7.1.5.1.9 Store Local Key Value Pair

This is the method to store a key value pair in the Local Key Storage of the device. It uses the component provided by the Cloud-based Information Infrastructure (Section 5.2.4).

Parameters:

- key: The key of the value to be stored
- value: The value of the data to be stored (here: string)

Return Value:

Returns the old value for the key or null if it was not set.

Error Handling:

According to the Cloud Storage Java API (see Listing 15), this function can throw an IOException on connect or store failures.

Remarks:

No further information is needed, as this function is just a wrapper for the Local Key Storage.

Listing 250: Source Code Example – Store Local Key Value

```
/**
 * @param key The key of the value to be stored
 * @param value The string value to be stored with this key
 * @throws IOException in case a connect or store failure occurred
 * @return Returns the old value set for the key or null
 */
public String storeLocalKeyValue(String key, String value) throws IOException {
    ILocalKeyStorage localKeyStorage = new LocalKeyStorage();
    return localKeyStorage.put(key, value);
}
```

7.1.5.1.10 Get Local Key Value Pair

This is the method to get a key value pair from the Local Key Storage of the device. It uses the component provided by the Cloud-based Information Infrastructure (Section 5.2.4).

Parameters:

- key: The key of the value to be retrieved

Return Value:

Returns the value for the key or null if it was not set.

Error Handling:

According to the Cloud Storage Java API (see Listing 15), this function can throw an IOException on connect or store failures.

Remarks:

No further information is needed, as this function is just a wrapper for the Cloud Storage Local Key Storage.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 366 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

Listing 251: Source Code Example – Get Local Key Value

```

/**
 * @param key The key of the value to be retrieved
 * @throws IOException in case a connect or store failure occurred
 * @return Returns the value set for the key or null
 */
public String getLocalKeyValue(String key) throws IOException {
    ILocalKeyStorage localKeyStorage = new LocalKeyStorage();
    return localKeyStorage.get(key);
}

```

7.1.5.1.11 Delete Local Key Value Pair

This is the method to delete a key value pair from the Local Key Storage of the device. It uses the component provided by the Cloud-based Information Infrastructure (Section 5.2.4).

Parameters:

- key: The key of the value to be deleted

Return Value:

Return the value for the key or null if it was not set.

Error Handling:

According to the Cloud Storage Java API (see Listing 15), this function can throw an IOException on connect or store failures.

Remarks:

No further information is needed, as this function is just a wrapper for the Cloud Storage Local Key Storage.

Listing 252: Source Code Example – Delete Local Key Value

```

/**
 * @param key The key of the value to be deleted
 * @throws IOException in case a connect or store failure occurred
 * @return Returns the value set for the key or null
 */
public String deleteLocalKeyValue(String key) throws IOException {
    ILocalKeyStorage localKeyStorage = new LocalKeyStorage();
    return localKeyStorage.remove(key);
}

```

7.1.5.1.12 Request Data from App

This method requests data from an app that is identified by the app id. This data can then be obtained via another function or be stored as soon as an app is started. Apps can hold data needed by the MMDI to produce specific output to the customer. Since apps are running inside the Application Runtime Environment, the MMDI needs to use the Application Runtime Environment as a gateway to the app. For instance when a user

wants to request his eco-index, the Multimodal Dialogue Interface reroutes this question to the app to retrieve the result.

Parameters:

The following parameters are expected:

- **appId:** The id of the app that is queried.
- **requestObject:** The app-given request object, that is queried. This is an object that functions as a query, all objects that match the set properties of the object are returned.
- **requestParameters:** The parameters that are used to query the app

Return Values:

This method returns an object with the data that the app returned after invoking the request. This can later be changed into a generic type that was defined in the Application Design Studio in Section 8.1.

Error Handling:

In case of an exception, a message is sent to the Error Handler (see Listing 258).

Remarks:

The AppRequestObject (see Listing 255) is an interface specifically inherited by the app to generate app-specific requestObjects.

Listing 253: Source Code Example – Request Data from App

```
/**
 * @param appId The id of the app that is queried.
 * @param requestObject The request object provided by the app, that is queried
 * @param requestParameters The parameters that are used to query the app
 * @return This method returns an object with the data that the app returned after
 * invoking the request.
 */
public object requestData(String appId, AppRequestObject requestObject, Map<string,
string> requestParameters) {
    App app = ExecutionManager.getAppById(appId);
    try
    {
        AppDataResult result = app.invoke(requestObject, requestParameters);
        Return result
    }
    catch (Exception ex)
    {
        ErrorHandler.raiseException(ex);
        return null;
    }
}
```


7.1.5.1.13 Execute App Action

This function executes a specific action of an app with the given parameters. The action has to be a function within the called app. This function is used by the MMDI to execute functions that a certain app provides. During the start of an app, the MMDI backend is handed the according appld. So, it can get further metadata about this app (see Section 7.2.5). This is used in the communication between the customer and the device.

Parameters:

- **appld:** The id of the app that is queried.
- **action:** The name of the action that is invoked
- **executeParameters:** The parameters that is passed to the action

Return Values:

Returns the result of the app action; this is an `AppActionResult`, which wraps state representing varying data because the result can vary from app to app.

Error Handling:

In case of an exception, an error is sent to the Error Handler (see Listing 258).

Remarks:

The action needs to be a function of the app and will be called via reflection, which is an internal function of the Application Runtime Environment. The Application Runtime Environment needs to be aware of existing functions inside an app.

Listing 254: Source Code Example – Execute App Action

```
/**
 * @param appId The id of the app that is queried.
 * @param requestObject The request object provided by the app, that is queried
 * @param requestParameters The parameters that are used to query the app
 * @return This method returns an object with the data that the app returned after
 * invoking the request.
 */
public object requestData(int appId, AppRequestObject requestObject, Map<string,
string> requestParameters) {
    App app = ExecutionManager.getAppById(appId);
    try
    {
        AppDataResult result = app.invoke(requestObject, requestParameters);
        return result;
    }
    catch (Exception ex)
    {
        ErrorHandler.raiseException(ex);
        return null;
    }
}
```

Listing 255: Java Class Definition – AppRequestObject

```

/**
 * Class to define a data request to an app
 */
public class AppRequestObject {
    private String name;
    private String action;

    public String getName() { /* ... */ }
    public void setName(String name) { /* ... */ }
    public String getAction() { /* ... */ }
    public void setAction(String action) { /* ... */ }
}

```

Listing 256: Java Class Definition – AppDataResult

```

/**
 * Class to define a data result from an app
 */
public class AppDataResult {
    private static final AppDataResult Error = new AppDataResult();
    private Boolean result;
    private String failureReason;

    public AppDataResult(String reason) { /* ... */ }
    public AppDataResult() { /* ... */ }
    public String getFailureReason() { /* ... */ }
    public void setFailureReason(String failureReason) { /* ... */ }
    public Boolean getResult() { /* ... */ }
    public void setResult(Boolean result) { /* ... */ }
}

```

Listing 257: Java Class Definition – App

```

/**
 * Classdefinition for an app
 */
public class App {
    private java.util.UUID app_id;
    private String author_name;
    private String description;
    private String[] keywords;
    private String name;
    private String version;
    private List<Resource> resources;

    public final java.util.UUID getAppId() { /* ... */ }
    public final void setAppId(java.util.UUID appId) { /* ... */ }

    public final java.util.UUID getAuthor_id() { /* ... */ }
    public void setAuthor_id(java.util.UUID author_id) { /* ... */ }
    public String getAuthor_name() { /* ... */ }
    public void setAuthor_name(String author_name) { /* ... */ }
    public String[] getKeywords() { /* ... */ }
    public void setKeywords(String[] keywords) { /* ... */ }
    public String getDescription() { /* ... */ }
    public void setDescription(String description) { /* ... */ }
    public String getName() { /* ... */ }
    public void setName(String name) { /* ... */ }
    public String getVersion() { /* ... */ }
    public void setVersion(String version) { /* ... */ }
    public List<Resource> getResources() { /* ... */ }
    public void setResources(List<Resource> resources) { /* ... */ }
    public Resource getResource(String resourceID) { /* ... */ }
    public AppDataResult invoke(AppRequestObject requestObject, Object[]
requestParameters) { /* ... */ }
}

```

Listing 258: Java Class Definition – ErrorHandler

```

/**
 * Definition of ErrorHandler
 */
public class ErrorHandler {
    public static void raiseMessage(Exception ex) { /* ... */ }
}

```

Listing 259: Java Class Definition – ExecutionManager

```

/**
 * Definition of ExecutionManager
 */
public class ExecutionManager {
    private List<App> runningApps = new Vector<App>();

    public boolean startApp() { /* ... */ }
    public void removeApp() { /* ... */ }
    public void updateApps() { /* ... */ }
    public List<App> getRunningApps() { /* ... */ }
    public App getAppById(String appId) { /* ... */ }
    public App generateAppId(App app) { /* ... */ }
}

```

7.1.5.1.14 Set Statusbar Information

The statusbar is similar to the Android OS status bar. It contains one entry for every app that wants to make use of it. This function is used to set some information concerning a specific app. To forward the statusbar entry to the User Interface, it is handed to the CommandHandler (Listing 262).

Parameters:

- **appId:** The id of the application that sets the status bar information
- **iconID:** The id of the icon that is displayed in the status bar
- **text:** The text that is displayed when asked for detailed information

Return Values:

A boolean value that indicates whether the status bar information could be set or not.

Error Handling:

If the information could not be set, an error is sent to the Error Handler (see Listing 258) and null is returned.

Remarks:

The Execution Manager needs to handle the function internally.

Listing 260: Source Code Example – Set Statusbar

```

/**
 * @param appId The id of the app that sets the status bar information
 * @param iconId The id of the icon that is displayed in the status bar
 * @param text The text that is displayed when asked for detailed information
 * @return A boolean value that indicates whether the status bar information could
be set or not
 */
public boolean setStatusbar(int appId, int iconId, String text) {
    try
    {
        App app = ExecutionManager.getAppById(appId);
        Resource res = app.getResource(iconId);
        boolean statusbarRes = CommandHandler.setStatusbar(appId, res, text);
        return statusbarRes;
    }
    catch (Exception ex)
    {
        ErrorHandler.raiseMessage(ex);
        return false;
    }
}

```

7.1.5.1.15 Handle Message

This function tells the MMDI to handle a certain message triggered by the user via the voice control of the PMA.

Parameters:

- **appId:** The id of app that is invoked.
- **invocationTarget:** The function of the app that is invoked.
- **invocationAction:** The action of the invocationTarget that is passed as a parameter and is only parsed by the invocationTarget itself.
- **invocationParameters:** The parameters that describe the context of the invocationAction.

Return Values:

Boolean value that if the message handling succeeded.

Error Handling:

In case of an exception, an error is sent to the Error Handler (see Listing 258) and the function returns the boolean value “false”.

Remarks:

This function will make use of Java reflection due to the dynamics of the invocation parameters. This could also be done via interfaces or abstract classes similar to the solution for requestData (see Section 7.1.5.1.9) function.

Listing 261: Source Code Example – Handle Message

```

/**
 * @param appId The id of the app that is invoked.
 * @param invocationTarget The function of the app that is invoked.
 * @param invocationAction The action of the invocationTarget that is passed as a
parameter and is only parsed by the invocationTarget itself
 * @param invocationParameters The parameters that describes the context of the
invocationAction
 * @return Boolean value if the message handling succeeded.
 */
public boolean handleMessage (int appId, String invocationTarget, String
invocationAction, Map<String, String> invocationParameters) {
    try
    {
        return CommandHandler.notifyStarted(appId, invocationTarget,
invocationAction, invocationParameters);
    }
    catch (Exception ex)
    {
        ErrorHandler.raiseMessage(ex);
        return false;
    }
}

```

Listing 262: Class Definition – CommandHandler

```

/**
 * Definition of CommandHandler
 */
public class CommandHandler {
    public void notifyStarted(String appId) { /* ... */ }
    public void notifyRemoved(String appId) { /* ... */ }
    public AppDataResult requestDataFromApp(String appId, AppDataRequestObject
requestObject, Map<String, String> requestParameters) { /* ... */ }
    public void handleMessage(String appId, String invocationTarget, String
invocationAction, Map<String, String> invocationParameters) { /* ... */ }
    public void executeActionApp(String appId, String action, Map<String, String>
executeParameters) { /* ... */ }
    public boolean setStatusBar(String appId, int iconId, String text)
    { /*... */ }
}

```

7.1.5.1.16 Get UUID

This method returns the Unique User Identifier UUID for the current user of the PMA. Please note that this UUID is app-specific and returns the UUID of the app that invokes this method.

Parameters:

- `appId`: The id of app that is invoked.

Return Values:

The return value of this method is the UUID of the user, respectively the app.

Error Handling:

The UUID is generated by the Application Runtime Environment (on OS level via an Android API method¹⁰⁴ call) to identify the user. It is designed to return a unique identifier in any case so the generation will not fail.

Remarks:

The UUID is the main identifier for apps and will be used to identify the app on every service call. This allows the service to account every invocation done by a particular app running on the PMA.

Listing 263: Source Code Example – Get the UUID of the Device

```
/**
 * @return Returns the UUID of the user.
 */
public String getUuid() {
    return ExecutionManager.generateAppId(app);
}
```

¹⁰⁴ [http://developer.android.com/reference/java/util/UUID.html#randomUUID\(\)](http://developer.android.com/reference/java/util/UUID.html#randomUUID())

7.1.5.2 RESTful Interface

7.1.5.2.1 Send Message

The Push Service offers the possibility to send a message to certain apps on devices via a REST API call. It enables the service developer to provide a subscription mechanism to app developers. Hence, the service developer is responsible to implement the subscriber management. The service developer has to take care that the app developer calls the service subscribe method with the PMA ID and app ID. So the service can notify the App in the intended cases. Also, this interface is used to answer a service request by an app, where the PMA ID and App ID are supplied automatically by the Application Runtime Environment in the request (Section 7.1.4.1 and Section 7.1.6.1).

Parameters:

The following parameters are expected:

- pma_id: The id of the receiving device

In the JSON Object attached to this call the following parameters are expected:

- sender: The id of the service that sent the message
- receiver: The id of the receiving app
- sent_time: The time the message was sent
- type: The type of the message, the name is assigned to a specific app
- message: The message (see Section 7.1.6.1) that is sent via the Push Service

Return Values:

Returns an HTTP status code according to the result of the call. The list of possible return codes is specified in the figure below.

Error Handling:

The error handling is described in the REST interface description in Table 87.

Remarks:

The REST API assures that all the receiving and sending IDs are correct and existent.

Table 87: Push Service Send Message REST Interface Description

Method	POST or PUT		URL	\$API_ROOT/are/push_service/:pma_id			
Description	Sends message to a specific app						
Parameter	pma_id	Required	yes	Possible Values	any device id	Description	The receiving device of the message
JSON Object	http://simpli-city.eu/ApplicationRuntimeEnvironment/PushService/Message						
JSON Attribute	sender	Required	yes	Possible Values	any UUID	Description	The sender of the message
JSON Attribute	receiver	Required	yes	Possible Values	any UUID	Description	The receiver of the message
JSON Attribute	sent_time	Required	yes	Possible Values	string	Description	The time the message was sent
JSON Attribute	type	Required	yes	Possible Values	string	Description	The type of the message
JSON Attribute	message	Required	yes	Possible Values	string	Description	The message itself, can be text or binary data.
Example URL	\$API_ROOT/are/push_service/7aa640627a5fb2650e2718df7113863a						
Response	Updated HTTP status code						
HTTP Status Code		Required	yes	Possible Values	200 400 502	Description	200: Message sent. 400: Bad request: Invalid parameter. 502: Message could not be sent.
Example Response	HTTP/1.1 200 OK Content-Length: 0 Connection: Close						

The message format for the push messages is defined in Section 7.1.6.1 and is therefore not duplicated here. An example of a push message is provided in Listing 264.

Listing 264: JSON Example – Send Message

```
{
  "sender": "a2ae6cc9a7acfff494422585a43459c2",
  "receiver": "86c61d2597857a96ae3a6048b5e7c5a7",
  "sent_time": "2013-09-09T13:02:24.002Z",
  "type": "eu.simplicity.services.routing.messages.reroute",
  "message": "f7 8e [...] 71 e4"
}
```

7.1.6 Content Format

7.1.6.1 JSON Message Format Schema

Listing 265 shows the schema of the standard message format for SIMPLI-CITY apps, which is used for internal communication; Listing 266 shows an example with 512 bytes of data (binary data abbreviated).

Listing 265: JSON Schema – Push Service Message

```
{
  "type": "object",
  "$schema": "http://json-schema.org/draft-03/schema",
  "id": "http://jsonschema.net",
  "required": false,
  "properties": {
    "message": {
      "type": "object",
      "id": "http://jsonschema.net/message",
      "required": false,
      "properties": {
        "message": {
          "type": "string",
          "id": "http://jsonschema.net/message/message",
          "required": false
        },
        "receiver": {
          "type": "string",
          "id": "http://jsonschema.net/message/receiver",
          "required": false
        },
        "sender": {
          "type": "string",
          "id": "http://jsonschema.net/message/sender",
          "required": false
        },
        "sent_time": {
          "type": "string",
          "id": "http://jsonschema.net/message/sent_time",
          "required": false
        },
        "type": {
          "type": "string",
          "id": "http://jsonschema.net/message/type",
          "required": false
        }
      }
    }
  }
}
```

Listing 266: JSON Example – Push Service Message

```
{
  "message": {
    "sender": "86c61d2597857a96ae3a6048b5e7c5a7",
    "receiver": "a2ae6cc9a7acfff494422585a43459c2",
    "sent_time": "2013-09-09T13:03:24.002Z",
    "type": "eu.simplicity.services.routing.messages.danger_ahead",
    "message": "f7 8e [...] 71 e4"
  }
}
```

7.1.7 Summary

The Android Framework was selected as runtime framework, as it allows full access to the hardware resources as opposed to other application development frameworks that just provide limited hardware access to handle events (e.g. several sensors on the PMA).

For the Server Side of the Application Runtime Environment, GCM was chosen, because of its integration of lightweight technologies for simple tasks like connection establishment and connection maintenance. It is also mainly developed for devices that make use of the Android platform and is therefore the perfect choice for the Push Service since the integration and development needs on the PMA Side and Server Side of the PMA is reduced to a minimum.

This also means that all the components described in the Functional and Technical Specifications need to be developed from scratch as no suitable modules or parts, which can serve as a base technology, currently exist. The components will be written in two different languages – all PMA Side components of the Application Runtime Environment will be using the native Android Application Framework, hence using Java as the main programming language. The Server Side components of the Application Runtime Environment (i.e., the Push Service) will be written in Go.

7.2 Multimodal Dialogue Interface

7.2.1 Major Design Decisions

The Multimodal Dialogue Interface is the user interface layer of SIMPLI-CITY and the PMA. It provides both verbal and graphical/haptic interaction capabilities. In other words, it is both the Graphical User Interface (GUI) and the Voice User Interface (VUI), serving as a layer between the user and the SIMPLI-CITY apps.

The Multimodal Dialogue Interface receives spoken and haptic (touch) input from the user, and asks questions and gives feedback to the user. It also sends requests about information and actions to the apps.

Although the Talkamatic Dialogue Manager (TDM) has support for multiple languages, only English will be prioritized in the SIMPLI-CITY project as decided in the course of deliverables D2.3 (Requirements Analysis) and D3.2.1 (Functional Specification).

During the definition of the Global Architecture (deliverable D3.1) and Functional Specification (deliverable D3.2.1), a number of different aspects of the Multimodal Dialogue Interface were discussed, which primarily have to do with the interface between

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 379 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

the dialogue definitions of the Multimodal Dialogue Interface and the Application Runtime Environment. In order to function properly, the Multimodal Dialogue Interface needs a description of the domain of the dialogue in terms of which actions, questions, answers, individuals, utterances etc. are available. These definitions can be done in a number of different ways, and this was the subject of most of the discussions. The following major design decisions were made:

Hybrid Ontology/Taxonomy:

Each application or domain used by the Talkamatic Dialogue Manager (TDM) needs an ontology, describing the facts in the world that an application can reason about. The question is whether each domain description contains an ontology of its own, or if there is a global ontology – common to all the domains. Hybrid solutions have also been considered, where there is a common – but extensible – ontology. This kind of hybrid approach was chosen for SIMPLI-CITY.

One Dialogue Domain:

SIMPLI-CITY will consist of many different apps, ranging over different domains, for instance parking, navigation, or eco-driving. To facilitate speech recognition, it can be a good idea to use a separate dialogue domain (dialogue application) for each domain, handling one single domain at a time. To enhance flexibility, it is, on the other hand, a better idea to use one single dialogue domain for all the apps. Since the quality of speech recognizers is constantly improving, this flexible single-domain approach was chosen.

Hierarchical Ontologies:

The ontologies used in the dialogue domain descriptions can be flat or hierarchical. Flat ontologies are easier to manage, but are less expressive than hierarchical ontologies. For example, a hierarchical ontology can be organized into broad categories on a high level, such as “transportation” and “entertainment”, and broken down into more specific categories on lower levels. In contrast, in a flat hierarchy, all concepts are aligned in a single level. The choice in SIMPLI-CITY will be hierarchical ontologies, with flat ontologies as a backup plan if needed.

Division in Initiative between the Application Runtime Environment and the Multimodal Dialogue Interface:

It is possible to shift control between the Application Runtime Environment and the Multimodal Dialogue Interface. One can put lots of logic into the Multimodal Dialogue Interface and very little logic in the Application Runtime Environment or the other way around. If more control is given to the Multimodal Dialogue Interface, the Application Runtime Environment becomes simpler, as the apps running in the ARE will become more like wrappers around the SIMPLI-CITY services. If more control is put in the Application Runtime Environment, the dialogue domains can be made simpler.

The language used to define the dialogue logic in the Multimodal Dialogue Interface is not particularly well suited to handle general programming tasks. Therefore, development is easier and faster if the Dialogue Domain Descriptions in the Multimodal Dialogue Interface take care of the high-level tasks, while the Command Handler in the Application Runtime Environment take care of the details.

7.2.2 Technology Comparison

For the Dialogue Manager toolkit, there will be no real technology selection, since the Talkamatic Dialogue Manager (TDM) is one of the very few available toolkits available for building spoken dialogue systems. TDM is one of the technologically leading dialogue managers on the market today, with built-in support for rapid development, multimodal interaction, grounding (making sure that all participants in a dialogue agree on what has been said and what it meant), topic-shifts (the ability to keep track of multiple topics), meta-dialogue on learned user preferences (to talk about user model information) and finally also for accommodation (support for integration of relevant, but not requested, user utterances). It is questionable whether any other dialogue manager (supporting the larger parts of the list above) is commercially available at all.

TDM uses an external Automatic Speech Recognizer; hence, an according technology comparison and selection has to be done (see Section 7.2.2.1). The same applies for the needed Grammar Tools (see Section 7.2.2.2).

7.2.2.1 Automatic Speech Recognizer

Table 88: Criteria for Technical Specification of Automatic Speech Recognizer

Parameter	Importance	Description
Realtime/Speed	++	Doing speech recognition in real time with a minimum of delays is essential to create a positive experience from using the system. Responsiveness is essential.
Large Coverage	++	The product should be able to handle large grammars, language models etc. without compromising speed and quality.
Cost	++	From an economic point of view, the costs for using the technologies should be included in the selection decision.
Quality	++	The quality of the output is essential. Misrecognitions, recognition failures and parse failures always have a large negative impact on the usability of a speech based system.
Platform Availability	++	The availability of a technology on a certain platform (here: Android, see Section 7.1) is essential.

The market for Automatic Speech Recognition (ASR) on mobile devices is dominated by two large players: Google and Nuance. Nuance provides different products for different needs: Vocon (for recognition locally on embedded devices), DragonMobile (for cloud-based unrestricted recognition) and Vocon Hybrid (combining Vocon with Dragon Mobile). Google offer their cloud-based and unrestricted recognition on the Android platform. In addition to the two dominants, there is also an open source GPL technology, PocketSphinx, which offers unrestricted as well as grammar-based recognition. Nuance has, in recent years, acquired all their major competitors except Google (SVOX, Loquendo, VLingo).

Below follows a more detailed list of the possible ASR options:

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 381 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

Nuance Vocon:

The Vocon recogniser has been the industry standard recogniser for automotive for a long time, and is considered by the industry as the best recogniser for embedded use. It is fast since it runs directly on the device. It can cope with relatively large grammars with reasonable speed, and can handle several hundred thousand utterances. The Software Development Kit (SDK) costs €1000 + €1000 per language that one would like to use. It also requires a small runtime licence fee for every unit.

Nuance DragonMobile:

This is the cloud-based recogniser from Nuance. It has a general language model, but there are also specialised language models available for instance for talking about music and TV. It is the recogniser used in Apple's SIRI. There are different service levels, and the one that you can get for free allows 500 transactions per day. Since the recogniser is cloud-based, a good network connection is required when using speech input. Latency can also be higher than for an embedded recogniser.

Nuance Vocon Hybrid:

Nuance Vocon Hybrid¹⁰⁵ provides a combination of Vocon and Dragon Mobile. An embedded recogniser (Vocon) is running on the actual handset (or tablet etc.), providing fast recognition for a smaller grammar, but the software is connected to a cloud-based recogniser (Dragon Mobile) as well, providing slower, network-dependent, but wider-coverage recognitions. In advanced applications, the network-based recogniser can be used to update, confirm or reject hypotheses given by the embedded recogniser.

Google ASR:

The ASR from Google¹⁰⁶ is free to use within the Android platform. It is an open recogniser, and Google currently provides two different language models, "web search" and "free form". The web-search language model is trained to recognise web search queries (isolated words, keywords, no complete sentences), while the free form model is trained on running text, utterances, commands etc. Like Nuance DragonMobile, Google ASR suffers from the limitations of cloud-based recognisers (network requirement and latency).

PocketSphinx:

PocketSphinx¹⁰⁷ is an open source recogniser, based on the CMU Sphinx ASR, is available under GPL and supports open dictation recognition as well as grammar based recognition. It is however very slow when it comes to recognising speech. Any grammar larger than toy examples will cause it to be unusable. It is also really slow when changing context or updating the recognition grammar.

¹⁰⁵ <http://www.nuance.com/for-business/speech-recognition-solutions/vocon-hybrid/index.htm>

¹⁰⁶ <https://play.google.com/store/apps/details?id=com.nll.asr&hl=de>

¹⁰⁷ <http://www.speech.cs.cmu.edu/pocketsphinx/>

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 382 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

Table 89: Comparison of Technologies for Automatic Speech Recognition

Parameter	Importance	Vocon	Dragon Mobile	Vocon Hybrid	Google ASR	Pocket-Sphinx
Generic Criteria						
Up-to-Datedness	--	5	5	5	5	5
Stability	++	8	10	8	10	4
Extensibility & Open Source/Standards	+/-	6	4	5	5	10
Familiarity	+	10	4	7	7	7
Performance	See Specific Criteria for a breakdown					
Interoperability	++	8	6	7	7	9
License (e.g., Apache 2.0)	--	Proprietary	Proprietary	Proprietary	Proprietary	GPL
Specific Criteria						
Realtime/speed	++	10	4	8	5	2
Large Coverage	++	8	10	10	10	2
Cost	++	3	3	3	10	10
Recognition Quality	++	8	7	7	7	5
Platform availability	++	10	10	10	10	7

7.2.2.2 Grammar Tools

The market for grammar toolkits, for fulfilling the need for parsing and generating natural language, are quite limited. Three different technologies have been considered:

Grammatical Framework:

The Grammatical Framework (GF)¹⁰⁸ is a grammar library developed at the Chalmers University of Technology, and takes a functional approach to grammar. It is an ambitious toolkit capable of generating parsers and generators for many languages, and also for compiling Context-Free Grammars in Backus-Naur Form (BNF) for use with grammar-based ASRs. GF itself is written in Haskell, and is very fast. GF development tools are released as open source under a GPL licence. Runtime libraries are also available under LGPL or BSD licences.

Natural Language Tool Kit:

The Natural Language Tool Kit (NLTK)¹⁰⁹ is a Python package for dealing with various problems in the Natural Language Processing (NLP) domain. There is support for context-free grammars, with parsers and generators. However, the parsers are relatively slow. NLTK is an open source project and its software is available under the Apache 2.0 license.

Information Retrieval:

Instead of using traditional grammars, one can use the approach of information retrieval, more or less using search tools for identifying relevant information in the user input. For instance: if the speech recogniser returns an hypothesis “play like a prior”, an information retrieval system designed to work with music titles should be able to identify the song title “Like a Prayer” from the recognised string using spell checkers, phonological indices etc.

¹⁰⁸ <http://www.grammaticalframework.org/>

¹⁰⁹ www.nltk.org

built from the music database. Solutions for this need to be tailor made, as the needs in this project differs from the mainstream use. Talkamatic has worked with a tool from Findwise¹¹⁰ in this area, but other vendors exist.

Table 90: Comparison of Technologies for Grammar Tools

Parameter	Importance	GF	NLTK	IR
Generic Criteria				
Up-to-Datedness	--	5	5	5
Stability	++	10	7	5
Extensibility & Open Source/Standards	+/-	10	10	2
Familiarity	+	7	1	5
Performance	See Specific Criteria for a breakdown			
Interoperability	++	5	8	8
License (e.g., Apache 2.0)	--	GPL/LGPL	GPL	Proprietary
Specific Criteria				
Realtime/speed	++	10	2	8
Large Coverage	++	5	6	8
Cost	++	10	10	3
Platform availability	++	8	8	3

7.2.3 Technology Selection

7.2.3.1 Automatic Speech Recognition

For the Automatic Speech Recognition, the ASR from Google will be used. It is freely available on the Android platform, with APIs included in the Android SDK. For targeting other platforms, such as iOS or Windows Phone, one would need to select another ASR, but for the prototypical implementations provided by SIMPLI-CITY, the Google ASR is nevertheless sufficient. The Google ASR has good quality and a good coverage of the kinds of utterances that can be expected in the apps foreseen in this project. Since it is not running in the handset but in the cloud, the speed could be an issue, but according to conducted tests with good network connections, this seems to be a minor problem. The ASR will however be unavailable in cases where the network coverage is bad.

7.2.3.2 Grammar Tools

For the Grammar Tools, GF will be selected. GF is fast and freely available, and is familiar to the developers within SIMPLI-CITY. What lacks from GF is the robustness, which means that it is not suitable for use directly with the kind of large-coverage ASR units that will be used in this project. In the TDM system, there is a build system which automatically builds parsing and generation grammars as well as standard ASR grammars (for grammar-based recognition).

¹¹⁰ www.findwise.com

7.2.3.3 Missing Elements and Implementation Needs

Talkamatic Dialogue Manager (TDM):

TDM is conceived for use on automotive head units, running under Linux. It is a Python application, and it can also run on Windows PCs and Apple Macintosh computers, or any other platform where Python is available. In this project, TDM will need to be deployed on the Android platform, running on Android handsets. This means that some of the components will need to be redesigned for the Android environment. For instance, the components for ASR and Text-to-Speech (TTS) as well as the GUI component will need to run in Android Java and communicate with the backend component over a network connection. These components are lightweight and uncomplicated (e.g., the TTS component in Python is 48 lines of code) compared to the DME etc. Additionally, the backend components need to be adapted to run as a server, either on the handset (using for instance Scripting Layer for Android (SL4A) or another Python platform for Android) or on a server.

GF & Google ASR:

The GF grammar system parser is fast and efficient for parsing phrases and sentences that has been foreseen by the grammar author. It is however not robust to unexpected input. If the author specified that “I would like to go to <city name>” is the expected way to answer questions about destinations, the system can only handle exactly such utterances and some regular variations described in the grammar. If not specified in the grammar, phrases like “I’m going to New York” will not be understood. When using a grammar-based recogniser such as Nuance Vocon, this is not a problem (or rather a problem on a higher level), since the grammar will also specify what utterances can be recognised – only utterances that can be parsed will be recognised by the ASR. Of course this can be avoided if all possible ways of expressing some certain intent is foreseen and covered in the grammar by the author, but this is not realistic. The GF parser needs to be made more robust to varying input from the user. There have been successful initiatives in the GF community to obtain this behaviour, which can be used as a basis for a successful solution.

One or Many Dialogue Domains:

As a single Dialogue Domain Description will be used for all apps, it will need to be extended as new apps are installed by the user. Such support for extending Dialogue Domain Descriptions on the fly needs to be implemented. It is also probable that some kind of namespace support in the TDM will be needed, for handling cases where multiple apps define plans, actions, etc. with similar names.

7.2.3.4 Further Information and Conclusion from Technology Comparison

The technology selection for the Multimodal Dialogue Interface is limited to deciding what technologies should be considered for Grammar Tools and for the ASR solution, as there are few (if any) real alternatives to the actual dialogue manager.

The selection of ASR solution will be limited to the current platform, as the Google ASR is only available in Google products (Chrome and Android). As soon as another platform is targeted, a different ASR product will need to be chosen. However, for the prototypical software to be developed within SIMPLI-CITY, this is sufficient.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 385 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

7.2.4 Component Structure

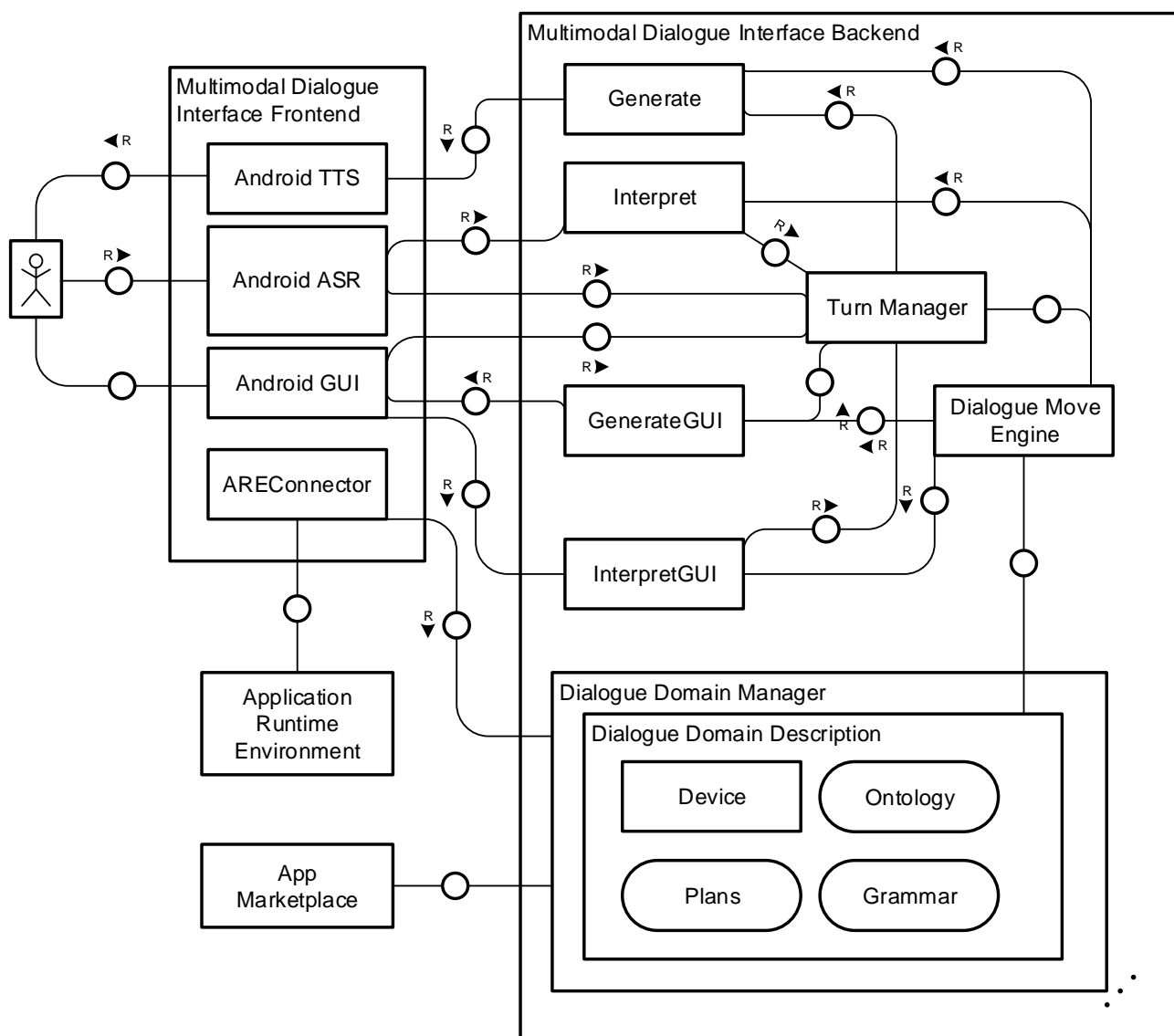


Figure 30: Component Structure of Multimodal Dialogue Interface

As can be seen in Figure 30, the subcomponents of the Multimodal Dialogue Interface include both frontend components, which necessarily are running on the client machine (in SIMPLI-CITY: a mobile device), and backend components, running locally (in the mobile device) or on a TDM server.

In order to clarify the structural relations between components, the connector components in the frontend and backend have been omitted from the figure. In reality, all calls between the frontend and backend are channelled via connector components. The connector components are described, along with the other components, in the following sections.

7.2.4.1 Frontend Components

Android ASR:

The Android ASR wraps the ASR service for speech input available in the Android API. The Android ASR module takes as input a speech signal from the user (something which is handled by the Android ASR service itself). The output is a number of hypotheses of what the user said, annotated with the probability/confidence that the individual hypothesis is actually correct. The output is transported to the server side, and broadcast to other modules which subscribe to this kind of event.

The Android ASR is activated by pushing the Push-To-Talk (PTT) button or – in the case of a Push-To-Initiate (PTI) button¹¹¹ – either by pushing that button, or by the Turn Manager in the backend signalling that the user is requested to speak.

Android GUI:

The GUI component is an Android application which offers menu choices and information to the user, and receives haptic/graphic input from the user. The user input is then sent as output from the module to the subscribing modules. The GUI takes as input an XML description from the GenerateGUI module and renders it on the screen.

Android TTS:

The Android TTS (Text-to-Speech) component wraps the TTS service available via the Android API. The TTS module takes as input a string of characters to speak, sent from the Generate module, but also focus information from the GUI in order to read out screen alternatives to the user. It also listens to notifications for if the PTT button is pressed, or if an item is selected in the GUI, in which cases the TTS is silenced. Finally, it also sends information to subscribers about when the current utterance was spoken or aborted.

AREConnector:

The AREConnector is a component which acts as a connection to the Application Runtime Environment. It must necessarily be on the frontend. It receives calls from the backend (more precisely the Dialogue Move Engine – see below) and requests data from the apps running in the Application Runtime Environment and requests the Application Runtime Environment to start and stop actions in the apps.

Backend Connector:

The Backend Connector, running in the frontend, is responsible for all communication with the backend. All messages to the backend from the frontend are channelled via the Backend Connector, as well as all messages from the backend to the frontend. In the case where the backend is running in the cloud, the frontend and backend will be physically connected over a network channel.

¹¹¹ The PTI button is pressed in order to initiate a dialogue, and the system and the user take turn speaking until the dialogue is finished. See D.3.1 for details.

7.2.4.2 Backend Components

Dialogue Move Engine:

The Dialogue Move Engine (DME) is the central component of the TDM, and is responsible for the actual dialogue logic. It keeps a model for the current dialogue context, which is updated by user and system utterances. This context model, the Total Information State, and a set of Update Rules are used in order to determine the next system move. This can be to make a certain utterance, to start an activity in an app or to request some data from an app. The DME is triggered by the Turn Manager, described below. The DME also disambiguates the input, when there are several interpretation hypotheses of the user utterance.

Interpret:

The Interpret module uses the grammar from the Dialogue Domain Description in order to interpret an utterance recognised by the ASR. The input consists of a set of hypotheses annotated with a confidence score. The module adds a semantic description to each of the hypotheses.

Generate:

The Generate module generates natural language utterances from semantic representations. It receives as input a semantic description of the utterance, and adds a surface representation to it in the form of a natural language utterance.

InterpretGUI:

The InterpretGUI module makes semantic interpretations of user activities in the GUI. InterpretGUI will take as input a GUI event, representing a menu choice, a button being pressed or similar. It will transform this event into a semantic description, which will be passed on to subscribers – primarily the Turn Manager.

GenerateGUI:

The GenerateGUI component is the GUI counterpart of the Generate module. It receives as input a semantic description of an utterance, and generates an XML description of a screen suitable for accompanying the utterance which will be generated and spoken.

Turn Manager:

Distributes the “turn” (the right and opportunity to speak) between the user and the Multimodal Dialogue Interface. It controls the activity of the output modules and the DME. It listens to input:

- From the Interpret and InterpretGUI modules about the latest user moves
- From the TTS module about when the last system utterance ended
- From the DME module about when it has stopped processing and if it has selected a next utterance
- From the AREConnector about started and stopped app actions

It feeds the DME with user and system dialogue move information and information about started and stopped actions.

Dialogue Domain Description:

The Dialogue Domain Description describes the ontology, the plans and the grammar of a particular dialogue domain. In SIMPLI-CITY, there will be a Core Domain Description, describing the dialogue domain for the core functionality of the PMA, as well as app-specific domain descriptions. The Core Domain Description will be relatively small, as most functionality in the PMA will be provided by the apps. The app-specific domain descriptions will be generated in the Application Design Studio and references to the actual description will be a part of the App Manifest. The actual software artifact will be stored in the App Marketplace (see Sections 6.5 and 8.1) and retrieved from the MDI when the app is installed.

When no app has yet been installed, the Dialogue Domain Description consists of only the Core Domain Description. When the user installs a new app from the App Marketplace, the Dialogue Domain Description is merged with the application-specific Domain Description fetched from the App Marketplace. Each subsequent app installation results in a similar merge.

Frontend Connector:

The Frontend Connector, running in the backend, is responsible for all communication with the frontend. All messages from the backend to the frontend are channelled via the Frontend Connector, as well as all messages from the frontend to the backend.

7.2.5 Interfaces

AREConnector
<pre>+appAdded(in appId : string) : void +notifyStarted(in appId : string, in action : string, in parameters : HashMap) : void +notifyEnded() : void[]</pre>

Figure 31: Public Methods in the Multimodal Dialogue Interface

Figure 31 shows the public methods in the Multimodal Dialogue Interface. The number of methods in the interface is quite small. The main reason for this is that the communication between the user and the Multimodal Dialogue Interface is handled within the component itself. Other components invoke methods in the Multimodal Dialogue Interface for two different reasons: to notify that a new app has been installed, and to notify that an action in an app started or ended. The methods are described in more detail below.

All interfaces of the Multimodal Dialogue Interfaces are implemented in Java. No RESTful interfaces are provided.

7.2.5.1 App Added

This method in AREConnector will be invoked by the Application Runtime Environment to inform the Multimodal Dialogue Interface that a new app has been added. When invoked, the Multimodal Dialogue Interface fetches the app's grammar, plans and ontology from the App Marketplace.

Parameters:

- **appId:** A unique identifier of the added App. In practice the identifier is a reference to the location of the App manifest file, which in turn contains references to the App's resources.

Return Value:

The method has no return value. In the case of an error, an exception is raised (see below).

Error Handling:

If an error is encountered, e.g., the same app id is added a second time, the method throws an exception describing the nature of the error.

Remarks:

Note that the Application Runtime Environment is not responsible for providing any information about the app. Instead, the Multimodal Dialogue Interface retrieves information about the app from the App Marketplace, using the app identifier.

Listing 267: Source Code Example – App Added

```
/**
 * @param appId, a unique identifier of the added app
 */
public void appAdded(String appId)

...

// Notify the Multimodal Dialogue Interface that an application has been added
String appId = app.getId();
api.appAdded(appId);
```

7.2.5.2 Notify Started

This method should be invoked when an app has notified the Application Runtime Environment that an action has been started. When called, the Multimodal Dialogue Interface updates its dialogue state and may decide to provide feedback to the user. For example, the ARE invokes the method when a media player App notifies that it is starting to play a specific song, and the Multimodal Dialogue Interface tells the user that the song is now being played by yielding spoken and graphical output. This method should be called both for actions initiated by the user and for actions initiated by some app.

Parameters:

- `appId`: A unique identifier of the app. In practice the identifier is a reference to the location of the app manifest file.
- `action`: The action is defined in the app manifest, and may also be covered by the global taxonomy.
- `parameters`: The parameters dictionary may contain action-specific fields such as name of song and artist.

Return Value:

The method has no return value. In the case of an error, an exception is raised (see below).

Error Handling:

If an error is encountered, e.g., the parameters were invalid in relation to the specified action, the method throws an exception describing the nature of the error.

Remarks:

In the source example below, the Application Runtime Environment notifies the Multimodal Dialogue Interface that a song is being played. Note that the action and its parameters are just illustrative examples. In reality, the Application Runtime Environment will not have any hard-coded knowledge about apps. Instead, it will just forward notifications that it receives from apps.

Listing 268: Source Code Example – Notify Started

```
/**
 * @param appId, a unique identifier of the app
 * @param action, an identifier of the action that was started
 * @param parameters, a dictionary of action-related parameters
 */
public void notifyStarted(String appId, String action, Map parameters)

...

// Notify the Multimodal Dialogue Interface that an action was started
Map parameters = new HashMap<String,String>() {
    {
        put("artist", "madonna");
        put("song", "like a prayer");
    }
};
api.notifyStarted(appId, "PlayMusic", parameters);
```

7.2.5.3 Notify Ended

This method should be invoked when an app has notified the Application Runtime Environment that an action has ended. When called, the Multimodal Dialogue Interface updates its dialogue state and may decide to provide feedback to the user. For example, the ARE invokes the method when a ticket-purchase app signals that a payment has been completed, and the Multimodal Dialogue Interface informs the user about the completion of the payment by yielding spoken and graphical output.

Parameters:

- **appld:** a unique identifier of the added app. In practice, the identifier is a reference to the location of the app manifest file, which in turn contains references to the app's resources.
- **action:** The action is defined in the app manifest, and may also be covered by the global taxonomy.
- **parameters:** The parameters dictionary may contain action-specific fields.

Return Value:

The method has no return value. In the case of an error, an exception is raised (see below).

Error Handling:

If an error is encountered, e.g., the parameters were invalid in relation to the specified action, the method throws an exception describing the nature of the error.

Remarks:

In the source example below, the Application Runtime Environment notifies the Multimodal Dialogue Interface that a payment was completed. Note that the action and its parameters are just illustrative examples. In reality, the Application Runtime Environment will not have any hard-coded knowledge about apps. Instead, it will just forward notifications that it receives from apps.

Listing 269: Source Code Example – Notify Ended

```

/**
 * @param appId, a unique identifier of the app
 * @param action, an identifier of the action that was ended
 * @param parameters, a dictionary of action-related parameters
 */
public void notifyEnded(String appId, String action, Map parameters)

...

// Notify the Multimodal Dialogue Interface that an action was started
Map parameters = new HashMap<String,String>() {
    {
        put("amount", "4.50");
        put("currency", "EUR");
        put("description", "Parking from 2014-10-10 13:43 to 2014-10-10 15:43");
    }
};
api.notifyEnded(appId, "PayParking", parameters);

```

7.2.6 Content Format

The Multimodal Dialogue Interface will mainly communicate with the Application Runtime Environment. The components send data to each other by invoking the Java methods, and by returning data in response to such calls.

For example, in an interaction where the user is about to pay for parking, the Multimodal Dialogue Interface may first request price information from the Application Runtime Environment (see Section 7.1.5.1.9). The price information is delivered back to the Multimodal Dialogue Interface as a return value, and is communicated to the user. If the user approves the purchase, the Multimodal Dialogue Interface sends a request to the Application Runtime Environment to execute an app action for completing the parking payment (see Section 7.1.5.1.13). In the case of a successful payment, the Application Runtime Environment sends a notification to the Multimodal Dialogue Interface (see Section 7.2.5.3).

Method invocations between the Multimodal Dialogue Interface and the Application Runtime Environment primarily contain two types of information: names (e.g., of actions or requested data) and parameters. Apart from this, there is no strict content format governing the content. Instead, the data structure – for example the parameter names – is app specific and defined by the app developer using the Application Design Studio.

7.2.7 Summary

For the Multimodal Dialogue Interface, Talkamatic's TDM technology will be used, as there are few if any real alternatives available. Google's ASR solution will be chosen since it offers the right combination of quality, speed and coverage for free. The Grammar Tools selected are GF, since the TDM developers are already familiar with the technology, and because it is free and efficient.

Implementation-wise, the TDM needs to be adapted to running on an Android Handset, and this may include moving the backend components onto a server or into the cloud. The

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 393 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

Google ASR also needs to be wrapped in a subcomponent. As Google ASR is a wide-coverage ASR (not grammar-based) one can expect the utterances to be very varied. The GF/Interpret module needs to be adapted to be able to handle these variations. Since a single instance of a Dialogue Domain Description will be used, it needs to be made dynamic, i.e., it needs to be able to receive updates when new apps are added and old apps removed.

The interface to the Multimodal Dialogue Interface is very small (with respect to the offered methods), since the component is at the very start of the interaction chain and since the interaction between the user and the Multimodal Dialogue Interface is handled internally within the component.

7.3 PMA-based Sensor Abstraction

7.3.1 Major Design Decisions

Since the PMA-based Sensor Abstraction is in fact a subcomponent of the Sensor Abstraction and Interoperability Interfaces (see Section 5.3), there is no separate technology selection necessary. The PMA-based Sensor Abstraction will be developed as Android background service in Java. The component directly interacts with the server-side Sensor Abstraction and Interoperability Interfaces and provides Java interfaces to the local running apps to access local sensors and sensors of directly connected devices, e.g., the car and its sensors. This allows locally running apps to access all sensors in the surrounding in a unified way and to receive data in a common data format. However, the main functionality of the PMA-based Sensor Abstraction component is the interaction with the previously mentioned server-side Sensor Abstraction and Interoperability Interfaces.

The PMA-based Sensor Abstraction provides the complex functionality to integrate the local sensors into the server-based interfaces. Thus, this component must provide to the server-based components the functionality to request for sensor data of the PMA or connected devices with very low latency and independent of the network and network address the PMA currently uses. The functionality and the components are described in the following subsections.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 394 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

7.3.2 Component Structure

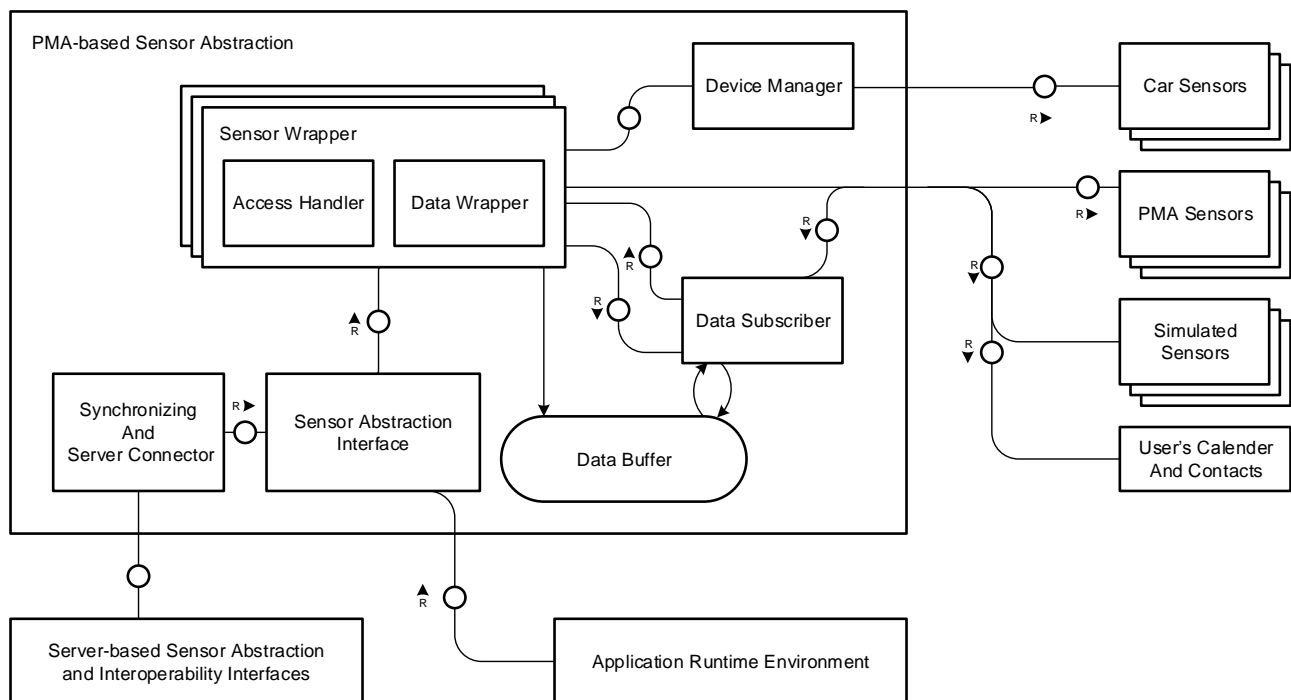


Figure 32: Component Structure – PMA-based Sensor Abstraction

The component structure of the PMA-based Sensor Abstraction has been updated in comparison with the Functional Specification (deliverable D3.2.1) and is depicted in Figure 32. Since a custom implementation is intended, there have been only minor changes.

The figure is simplified with respect to external sensor data sources. To increase the readability there is only one box, e.g., Car Sensors, depicted as placeholder for all external devices that are data sources. The new subcomponent Device Manager is introduced. This subcomponent of the PMA-based Sensor Abstraction will act as control unit to interact with external data sources, e.g., the user's car.

To achieve the envisioned functionalities, the PMA-based Sensor Abstraction is providing the following subcomponents as depicted in the figure above. A local buffer is used to store sensor values received by events until they are requested.

7.3.2.1 Sensor Wrapper

The Sensor Wrapper is the basic component to access external sensor data sources and translate the external data format into the SIMPLI-CITY data format. It mainly consists of the two subcomponents Access Handler, which is responsible to access the external sensor data source, and the Data Wrapper that provides the wrapping facilities for transforming diverse proprietary sensor data to data (formats) usable by the other SIMPLI-CITY components, respectively SIMPLI-CITY apps and services.

7.3.2.2 Synchronizing and Server Connector

The Synchronizing and Server Connector component has two major functionalities. On the one hand, it is responsible for synchronizing some of the PMA or user related data with the server side to make it even available if the PMA is offline. On the other hand it is

responsible to allow the server to establish a connection and provides the necessary interfaces to allow the server access to the local sensors. The component allows the server to send sensor requests to the PMA. For this, a kind of push notification will be used. The response is then sent into a message queue on the server where the complementary component retrieves this information. This allows an asynchronic communication approach.

7.3.2.3 Sensor Abstraction Interface

This component is the API that is exposing the interfaces to apps in the Application Runtime Environment and therefore allows apps to exploit data from device-internal data sources. These APIs are provided as Java interfaces.

7.3.2.4 Data Subscriber

This component is responsible for receiving data triggered by an external event. It provides the functionality to subscribe to external event bus systems and allows receiving event triggered sensor readings. Such data is buffered in the Data Buffer until the value is requested via the Sensor Wrapper or a new value is provided by a subsequent event.

7.3.2.5 Device Manager

The Device Manager will act as a control unit to interact with external data sources. It is responsible for managing and establishing a wireless connection to the external devices, e.g., the user's car.

7.3.3 Interfaces

The PMA-based Sensor Abstraction will only provide Java interfaces to locally running apps. No RESTful interfaces are provided. The Server-based Sensor Abstraction and Interoperability Interfaces will provide the interfaces to allow the server components to access the PMA sensors. Each Java interface is described in detail in the following by the description of the input and output parameters and a code example. The functionality of the single interfaces is naturally related to the interfaces of the same name of the Server-based Sensor Abstraction and Interoperability Interfaces (see Section 5.3.5). The custom data types Device and Sensor are described in Section 7.3.2 and further specified in Section 7.3.4. For the unique identifiers, the Java standard type `java.util.UUID` will be used.

PMASensorAbstraction
<pre> +getSensor(in sensorID : UUID) : Sensor +getDevice(in deviceID : UUID) : Device +getPmaInformation() : Map[] +getContacts(in filterValues : Map) : Sensor[] </pre>

Figure 33: Class Diagram of PMA-based Sensor Abstraction

7.3.3.1 Get Sensor Data

The Java interface Get Sensor Data (see Listing 270) returns the actual data of a particular sensor.

Parameters:

- `sensorID`: UUID of the respective sensor

Return Value:

The sensor data as `eu.simplicity.Sensor` object.

Error Handling:

A `SensorNotFoundException` is thrown in case the sensor ID is invalid.

Remarks:

None

Listing 270: Source Code Example – Get Sensor Data

```
/**
 * Requests for the actual data of a particular sensor
 *
 * @param sensorId of the sensor
 * @throws SensorNotFoundException in case the sensor ID is invalid
 */
public Sensor getSensor(UUID sensorID) throws SensorNotFoundException;

...

// Request sensor
try {
    Sensor s = getSensor(sensorID);
} catch (SensorNotFoundException e) {
    ...
}
```

7.3.3.2 Get Device Data

The Java interface Get Device Data (see Listing 271) returns the actual data of a particular device.

Parameters:

- `deviceId`: UUID of the respective device

Return Value:

The sensor data as `eu.simplicity.Device` object.

Error Handling:

A `DeviceNotFoundException` is thrown in case the device ID is invalid.

Remarks:

None

Listing 271: Source Code Example – Get Device Data

```
/**
 * Requests for the actual data of a particular device
 *
 * @param deviceId of the sensor
 * @throws DeviceNotFoundException in case the device ID is invalid
 */
public Device getDevice(UUID deviceId) throws DeviceNotFoundException;

...

// Request device
try {
    Device d = getDevice(deviceID);
} catch (DeviceNotFoundException e) {
    ...
}
```

7.3.3.3 Get Contact Data

Data sources for personal data are handled as virtual sensors of the PMA and the corresponding IDs are returned amongst others by the method *getPmaInformation* (see Section 7.3.3.4). To allow requesting a particular data set, e.g., the contact data of one particular person, this method allows filtering the data source for particular values.

Parameters:

- **filterValues:** (key,value) pairs to filter the data source as Strings in a Map, e.g., (street, Mainstreet) or (name, Smith)

Return Value:

The contact data as array of *eu.simplicity.Sensor* objects with one array object per matched contact. This allows treating all information provided by the Sensor Abstraction Interfaces in the same way as virtual sensors. If no available contact matches to the **filterValues**, the return value is null.

Error Handling:

None

Remarks:

None

Listing 272: Source Code Example – Get Contact Data

```

/**
 * Requests for the actual data of a particular sensor
 * Filtered by the input parameters
 *
 * @param filterValues Map<String,String> as Key Value pairs to filter
 * the data source
 *
 */
public Sensor[] getContacts(Map filterValues);

...

// Request sensor
Sensor[] s = getContacts (filterValues);

```

7.3.3.4 Get PMA Information

The Java interface Get PMA Information (see Listing 273) provides the IDs of the devices and sensors connected to this PMA.

Parameters:

None (as this interface is locally at a particular device)

Return Value:

The device data of the PMA as eu.simplicity.Device object.

Error Handling:

None

Remarks:

None

Listing 273: Source Code Example – Get PMA Information

```

/**
 * Request the actual data of this particular device
 *
 *
 */
public Device getPmaInformation();

...

// Request PMA device
Device d = getPmaInformation();

```

7.3.4 Content Format

Naturally, the PMA-based Sensor Abstraction makes use of the same data formats as the Server-based Sensor Abstraction and Interoperability Interfaces. Hence, refer to Section 5.3.6 for a description of the applied data formats.

For the sake of better readability, the structure of the data format is described here again and is depicted in Figure 34. The Java interfaces will return custom Java objects of the same structure as described for the RESTful interfaces in Section 5.3.6. In the hierarchical view, depicted in Figure 34, on the top level there will be an object class that brings the basic properties for devices and sensors. Each object brings the following properties:

- Object: The respective class descriptor (sensor, device)
- Timestamp: The point in time of object creation
- ObjectID: A unique 128 bit identifier
- Parent: The object ID of the parent object if available

However, the type object will never be directly initiated. Instead, objects of the derived classes sensor or device will be used. A device can be connected from 0 to n sensors and 0 to m other devices. That means there could be a cascade of devices, each in between device can have sensors but does not need to have. Sensors are always connected to devices and have 1 to n values. That means a sensor has at minimum one corresponding value. All sensors have a field describing their type. Sensor values are provided as an array; so, complex sensor values can be represented as an array of single values. Each sensor value can have, but does not have to have, an accuracy property that has a value describing the accuracy range and a corresponding unit. If a data object is not available the value null is provided.

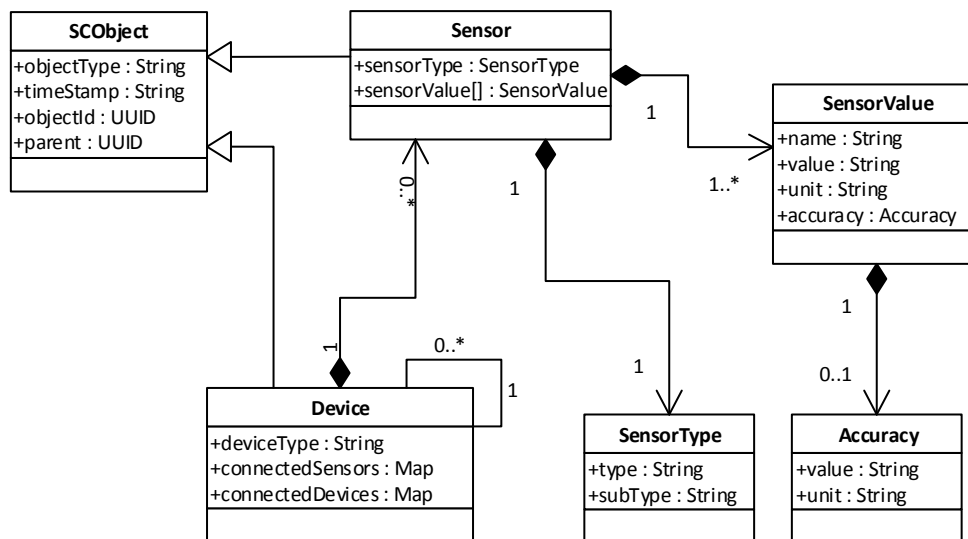


Figure 34: UML Class Diagram – Sensor- and Device-related Data Objects

7.3.4.1 SObject Java Class

The Java class SObject (see Listing 274) contains the basic properties of the objects Sensor and Device that extend this class and are used for instantiation. The class SObject will never be instantiated itself.

Listing 274: Java Class – SObject

```
import java.util.UUID;

public abstract class SObject {

    public String objectType;
    public String timeStamp;
    public UUID objectID;
    public UUID parent;

    public SObject() {
        setNewTimeStamp();
        setNewRandomUUID();
    }

    public SObject(String timestamp, String id, String parent) {
        this.timeStamp = timestamp;
        this.objectID = UUID.fromString(id);
        if (!parent.equals("null") && !parent.isEmpty()) {
            this.parent = UUID.fromString(parent);
        }
    }
}
```

7.3.4.2 Device Java Class

The Java class Device (see Listing 275) extends the class SObject and represents a physical device that has Sensors and connected Devices.

Listing 275: Java Class – Device

```
import java.util.ArrayList;
import java.util.Map;

public class Device extends SObject {

    public static final String IDENT = "device";

    public Map<String, SensorType> connectedSensors;

    public ArrayList<String[]> connectedDevices;

    public String deviceType;

    public Device() {
        super();
        objectType = IDENT;
    }

    public Device(String timestamp, String id, String parent) {
        super(timestamp, id, parent);
        objectType = IDENT;
    }

    public Device(String timestamp, String objectID, String parent, String
deviceType) {
        super(timestamp, objectID, parent);
        objectType = IDENT;
        this.deviceType = deviceType;
    }

    public Map<String, SensorType> getConnectedSensors() {
        return connectedSensors;
    }

    public void setConnectedSensors(HashMap<String, SensorType> connectedSensors) {
        this.connectedSensors = connectedSensors;
    }

    public ArrayList<String[]> getConnectedDevices() {
        return connectedDevices;
    }

    public void setConnectedDevices(ArrayList<String[]> connectedDevices) {
        this.connectedDevices = connectedDevices;
    }

    public String getDeviceType() {
        return deviceType;
    }
}
```

7.3.4.3 Sensor – Java Class

The Java class Sensor (see Listing 276) extends the SObject and represents a Sensor.

Listing 276: Java Class – Sensor

```
public class Sensor extends SObject {

    public static final String IDENT = "sensor";
    public SensorType type;
    public ArrayList<SensorValue> sensorValue;

    public Sensor(String timestamp, String id, String parent){
        super(timestamp, id, parent);
        objectType = IDENT;
    }

    public SensorType getSensorType() {
        return type;}

    public void setSensorType(SensorType sensorType) {
        this.type = sensorType;}

    public void setSensorValue(ArrayList<SensorValue> sensorValues) {
        this.sensorValue = sensorValues;
    }

    public ArrayList<SensorValue> getSensorValues() {
        return sensorValue;
    }

}
```

7.3.4.4 SensorType – Java Class

The Java class SensorType (see Listing 277) is used to represent the different types of Sensor objects.

Listing 277: Java Class – SensorType

```
public class SensorType {

    public String type;
    public String subtype;

    public SensorType(String type, String subtype) {
        this.type = type;
        this.subtype = subtype;
    }

    public String getType() {
        return type;
    }

    public void setType(String type) {
        this.type = type;
    }

    public String getSubtype() {
        return subtype;
    }

    public void setSubtype(String subtype) {
        this.subtype = subtype;
    }

}
```

7.3.4.5 SensorValue – Java Class

The Java class SensorValue (see Listing 278) is used to represent the values of Sensor objects.

Listing 278: Java Class – SensorValue

```
public class SensorValue {

    public String name;
    public String value;
    public String unit;
    public Accuracy accuracy;

    public SensorValue(String name, String value, String unit, Accuracy accuracy) {
        this.name = name;
        this.value = value;
        this.unit = unit;
        this.accuracy = accuracy;
    }

    public String toString() {
        String sensorValue = "sensorValue:\n\tname: " + name + "\n\tvalue: " + value +
            "\n\tunit: " + unit
            + "\n\taccuracy: \n\t\tunit: " + accuracy[0] + "\n\t\tvalue: " + accuracy[1];
        return sensorValue;
    }

    public String getName() {
        return name;
    }

    public String getValue() {
        return value;
    }

    public String getUnit() {
        return unit;
    }

    public Accuracy getAccuracy() {
        return accuracy;
    }
}
```

7.3.4.6 Accuracy – Java Class

The Java class Accuracy (see Listing 279) is used to represent the accuracy of SensorValue objects.

Listing 279: Java Class – Accuracy

```
public class Accuracy {  
  
    public String unit;  
    public String value;  
  
    public String getUnit() {  
        return unit;  
    }  
  
    public String getValue() {  
        return value;  
    }  
}
```

7.3.5 Summary

In this section, the PMA-based Sensor Abstraction has been introduced as a subcomponent of the Sensor Abstraction and Interoperability Interfaces. Thus there was no separate technology selection necessary. The PMA-based Sensor Abstraction will be developed as an Android background service in Java that directly interacts with the Server-based Sensor Abstraction and Interoperability Interfaces. This enables the Server-based Sensor Abstraction and Interoperability Interfaces to access the sensors of the PMA and the connected devices. The PMA-based Sensor Abstraction provides Java Interfaces to the local running Apps to access local sensors and sensors of directly connected devices, e.g., the car and its sensors, directly and not only via the Server-based Sensor Abstraction and Interoperability Interfaces.

8 Technical Specification: Developer Support

8.1 Application Design Studio

The Application Design Studio is a tool with which SIMPLI-CITY apps (applications running in the Personal Mobility Assistant) will be designed and implemented. In addition to assisting developers while programming, the Application Design Studio will allow them to manage the source code, debug the app, add documentation, compile, run, bundle the app and send it to the App Marketplace. As such, the Application Design Studio will not interact with other SIMPLI-CITY components like the Cloud-based Information Infrastructure (see Section 5.2) or PMA-based Sensor Abstraction (see Section 7.3) itself, but will rather help the developer to make use of them in the app during design time by means of providing a complete set of component usage examples, guidelines, HowTos and JavaDocs, so that developers can easily find guidance in the programming phase.

The Application Design Studio is one of the two main design components of the SIMPLI-CITY project (the other one being the Service Development API as described in Section 8.2). It will assist the developer in starting a new app project from scratch, creating the recommended directory structure, filling it with necessary building stones like main class definition templates, configuration and documentation file templates, or auxiliary helper library classes with related Java documentation. By informing the Application Design Studio about SIMPLI-CITY services the developer intends to use in the app, developers will be given access to documentation on these particular services provided by the service authors.

The outcome of the Application Design Studio is the newly created app and all the information related to the configuration of the PMA for optimal execution. This configuration information is provided in the form of an app Manifest file (see Section 9.2) which specifies the runtime environment requirements, properties such as the app icon or even provides the dependencies between the apps.

8.1.1 Major Design Decisions

During the discussions of the Global Architecture (deliverable D3.1) and Functional Specification (deliverable D3.2.1), the following major design decisions have been made:

Access to APIs:

App developers are going to make use of different APIs provided by services (either developed by the SIMPLI-CITY project or by other developers) or provided by SIMPLI-CITY components such as the Cloud-based Information Infrastructure. The Application Design Studio has to assist the developer in getting access to and using these APIs for proper app development.

Access to Documentation:

When developing a new app, the developer can make use of different resources in the form of components and services documentation, examples of apps already developed by the consortium and best practices for a professional development and a smooth running of the future app.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 407 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

Connectivity to External Data Sources during Design Time:

Part of app development consists in getting access to different repositories for reading/updating information from them, e.g., an app for finding public parking spaces would have to access Open (Government) Data repositories, or sensors which are available in cars, for example, the fuel level sensor.

As not all of these external repositories and sensors might be available during design time, it is the responsibility of the app developer to provide mock versions of these repositories and configure them in the PMA for the local testing of the app. Once tested, bundled, sent to the App Marketplace and then deployed in a user PMA, the app will seamlessly use the real repositories as they are programmed against the same interfaces as the mock ones. If a repository is available during design time, there is no need to provide a mock version. The developer can use the real repository straight away.

Publication of Apps:

Any developed app that is supposed to be executed by the PMA has to be published in the App Marketplace before it can be used. The publication of apps is an independent process in which the Application Design Studio provides only initial assistance and the rest of the process is executed in the App Marketplace. These first steps are the following:

- Compilation of the app
- Creation of the app bundle
- Manual submission of the bundle to the App Marketplace

The bundle will be comprised of the following files:

- App main JAR
- App Manifest (see Section 9.2)
- App auxiliary files, required for using the app in the SIMPLI-CITY PMA, such as the grammar, ontology and plans for the Multimodal Dialogue Interface.

8.1.2 Technology Comparison

8.1.2.1 Comparison Criteria

This subsection will compare existing Integrated Development Environments (IDEs) suitable for the development of SIMPLI-CITY apps. One of these IDEs will be used as a base for the development phase. The selection criteria for the Application Design Studio technologies were defined in Section 8.1.7 of the Functional Specification (deliverable D3.2.1).

Table 91: Criteria for Technical Specification

Parameter	Importance	Description
Android Development Support	++	As SIMPLI-CITY apps will be native Android apps (see the Application Runtime Environment section), the IDE being chosen has to be tailored toward Android development (support package management, assistance, debugging, etc.).

Plugin Support	++	The IDE being selected must be customizable as there will be a plugin to serve examples of code and HowTos in addition to a manifest editor.
Ease of Use	++	App developers prefer IDEs that are easy to use and provide a known interface. Therefore, the IDE for the Application Design Studio must have a friendly UI and be easy to operate.

8.1.2.2 Possible Technologies and Comparison

To develop native apps for Android within the scope of the SIMPLI-CITY project, an IDE that fulfils the mentioned criteria above must be provided. This IDE could be the native Android Studio as well as other IDEs that support building apps for the Android platform by use of a plugin.

Android SDK (Eclipse and ADT-Plugin):

The Android SDK¹¹² contains the API libraries and developer tools necessary to build, test and debug apps for Android. It is especially recommended for beginners in Android development to quickly start developing apps. It includes the essential Android SDK components and a version of the Eclipse IDE with built-in ADT (Android Developer Tools) to ease Android app development. The Android SDK also includes the latest Android platform, an SDK manager that allows managing the available platforms and a virtual device manager that makes it possible to run apps without having an Android Device.

Android Studio:

The Android Studio¹¹³ is an open IDE from Google based on IntelliJ IDEA¹¹⁴. It was essentially made to facilitate developing for the Android platform. Several features are expected to be rolled out to the developers such as the live layout, specific refactoring and quick fixes and app-signing capabilities. IntelliJ has a modular architecture and supports custom plugins.

One feature of the Android Studio is live code updates and live renderings of an app in real time. This allows seeing what the app will look like as one types in code and makes changes to it. The IDE also lets one see how the app looks on different screen sizes, and there is an extensive library of devices supplied. There is not only a variety of emulators for previewing apps but it is possible to see the design on various screen sizes while making changes to it. This is a very useful feature for Android developers who must design their apps to fit a large number of different screens. Review analysis shows superiority of Android Studio implementation in this aspect.

As for developing apps for an international audience, Android Studio also includes different language tools. One can get a real-time view of how the app works with different languages, see how the text changes and affects other elements in the app as the language is changed. Android Studio seems to be unique in implementing this feature.

¹¹² <http://developer.android.com/sdk/index.html>

¹¹³ <http://developer.android.com/sdk/installing/studio.html>

¹¹⁴ <http://www.jetbrains.com/idea/>

NetBeans with NBAndroid-Plugin:

NetBeans¹¹⁵ is one of the most popular general purpose IDEs and through its extensibility it also supports Android app development. This is provided by adding the NBAndroid¹¹⁶ plugin to the IDE. This plugin supports the full Android app development cycle. The NBAndroid plugin covers all general aspects of Android Development.

NetBeans NBAndroid plugin is an active project supported by a small group of enthusiasts. Having mixed reviews, it seems more future-proof to build SIMPLI-CITY Application Design Studio on the basis of a product backed by the developer of the Android platform (Google) via either Eclipse ADT plugin or the Android Studio.

AIDE:

AIDE¹¹⁷ is a mobile IDE which runs on Android Devices. AIDE supports the full edit-compile-run cycle: writing code with a feature-rich editor offering advanced features like code completion, real-time error checking, refactoring, smart code navigation, and running developed apps with a single click. AIDE can only work on an Android powered device and not on a personal computer.

Whilst innovative and a promising product, AIDE is designed to run on the Android device itself and not on a PC. In addition, AIDE lacks the plugin support. SIMPLI-CITY aims to extend a well-established free desktop IDE with own plugins, therefore plugin support is mandatory. In addition, the free AIDE version is targeted at smaller scale projects (less than five files) and for developing bigger SIMPLI-CITY apps an expensive premium key would be necessary, which is against the free and open nature of SIMPLI-CITY.

¹¹⁵ <https://netbeans.org/>

¹¹⁶ <http://www.nbandroid.org/>

¹¹⁷ <http://www.android-ide.com/>

Table 92: Technology Selection Criteria and Comparison of Technologies for the Application Design Studio

Parameter	Importance	Android SDK (Eclipse + ADT)	Android Studio	NetBeans & NBAndroid	AIDE
Generic Criteria					
Up-to-Datedness	+	9	10	9	9
Stability	++	10	10	8	8
Extensibility & Open Source/Standards	++	10	10	8	6
Familiarity	++	9	9	8	6
Performance	-	10	10	8	10
Interoperability	+/-	9	10	10	8
License (e.g., Apache 2.0)		Proprietary	Apache 2.0	CDDL	Proprietary
Specific Criteria					
Android Development Support	++	9	10	8	8
Plugin Support	++	10	10	10	6
Ease of Use	++	10	10	8	4

8.1.3 Technology Selection

8.1.3.1 Selection of the Foundation IDE

The IDEs, chosen as possible candidates on which SIMPLI-CITY will build its Application Design Studio, are all products well-known in the development community. The majority of them have a long history such as Eclipse (used together with Google ADT plugin in Android development), IntelliJ IDEA (the new Google Android Studio is based on this product from JetBrains) or NetBeans. AIDE is a niche product for developing Android apps on the Android device itself, preferably a tablet.

The results from the comparison table mainly argue for the use of Android Studio versus the Eclipse ADT plugin. The choice between Eclipse with the ADT plugin and Android Studio is rather subjective in nature. Both options are actually plugins built on top of very mature and highly regarded IDEs (Eclipse and IntelliJ IDEA respectively). Both of these IDEs support modular structure and custom plugins. In addition, these plugins are developed by Google, maintainer of the Android platform. This means both options are well suited for implementing extra functionality SIMPLI-CITY wishes to provide for the app developer.

Android Studio seems more promising than Eclipse with the ADT plugin because Google is already talking about adding more features to it. The company wants to integrate more and more services into Android Studio in the future. Google mentioned it is going to continue supporting the ADT plugin, but the Android Studio is a new product released by Google in close collaboration with JetBrains (developer of IntelliJ IDEA) after already having the Eclipse ADT for quite a while.

8.1.3.2 Missing Elements and Implementation Needs

Android Studio is based on the well-known IntelliJ IDEA IDE. This IDE allows seamless integration of third party modules known as plugins that can enhance the functionality of the IDE. The SIMPLI-CITY consortium will implement the Manifest Editor, the App Bundle Manager and the Documentation Centre in the form of such a plugin:

Manifest Editor:

The Manifest Editor will contain a GUI to easily enter needed information about a SIMPLI-CITY app. The Manifest will also be manually editable via a simple text editor, though it is recommended to make use of the Manifest Editor. The app Manifest and its contents are defined and described in Section 9.2.

App Bundle Manager:

The App Bundle Manager will be part of the finishing steps of the app creation. It builds the whole app with the Manifest and possible generated documentation and bundles it to a format that is used SIMPLI-CITY-wide for the recognition of apps. These bundles will be used for the App and Service Marketplaces, which will create app pages out of the information stored in the bundle.

Documentation Center:

The Documentation Center will provide the following functionality to the developer while creating an app:

- Helper documents and API documentation will be integrated into the central IDE help menu.
- Examples will be provided as projects that make use of the Application Design Studio features.
- The API will also be available for the code completion feature of IntelliJ.

8.1.3.3 Further Information and Conclusion from Technology Comparison

From the investigation outlined above, it followed that two of potential candidate technologies (NetBeans with NBAndroid plugin and AIDE) did not provide functionality set or solidness required by the SIMPLI-CITY project. The choice between the other two (Eclipse with the ADT plugin and Android Studio) was rather subjective as they provided almost the same level of functionality, extensibility and developer usefulness. However, it should be noted that Android Studio is currently supported by Google as a “flagship IDE”.

As the goal was to choose a developer-friendly, actively supported and feature rich IDE, backed by the maintainer of the Android platform, the Android Studio has finally been chosen.

8.1.4 Component Structure

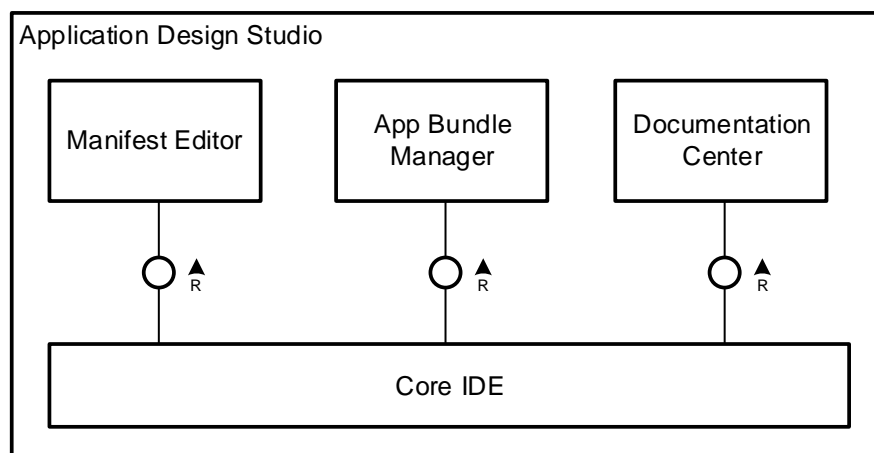


Figure 35: Component Structure of the Application Development Studio

8.1.4.1 Core IDE

The Core IDE is comprised of the Google Android Studio based on the IntelliJ IDEA IDE. It provides the developer with basic source code editor functionality such as syntax highlighting, project organisation and a build system. The other components described below will be implemented in the form of a plugin for IntelliJ IDEA and plugged into the Core IDE.

8.1.4.2 Manifest Editor

The Manifest Editor will be used by the developer to store information about the app that is to be created. This information will be saved in the app Manifest and be used by several components of SIMPLI-CITY. This editor contains some automation logic to fill some information (e.g., build version, build date, author) by itself. This keeps the maintenance of the Manifest file as low as possible.

8.1.4.3 App Bundle Manager

The App Bundle Manager combines the compiled source code and the manifest to a bundle that is compiled into a format that is understandable within the whole SIMPLI-CITY ecosystem. It provides a separate UI to the developer to easily bundle and deliver an app for further use.

8.1.4.4 Documentation Center

The Documentation Center will provide a broad variety of helper documents for the creation of SIMPLI-CITY apps. It is the main resource for developers to gather information about the creation and publishing of SIMPLI-CITY apps. It also supports the developer with helpful functions inside the Application Design Studio like code completion and snippets to fasten the development process.

8.1.5 Interfaces

As the Application Design Studio is a product on its own, it does not have any interfaces to expose to other components in the SIMPLI-CITY ecosystem. By convention, only

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 413 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

interfaces crossing borders are defined in this Technical Specification, therefore definitions of interfaces between subcomponents of Application Design Studio as well as UML class diagrams are omitted in this document.

8.1.5.1 Example of a SIMPLI-CITY App Creation

A developer using the Application Design Studio will be consulting examples and code snippets which will be included in HowTos supplied with the Application Design Studio. Examples to be delivered are described below. Examples are divided by major component the developer of an app is likely to use with sub-examples for using different functionality of these components.

1. Local Key Storage (on the device)
 - a. Storing data / Updating Data
 - b. Deleting data
 - c. Retrieving data
2. Cloud-based Information Infrastructure
 - a. Storing data / Updating Data
 - b. Deleting data
 - c. Retrieving data
3. PMA-based Sensor Abstraction
 - a. Accessing and reading data from the GPS sensor
 - b. Retrieving current car speed
4. Marketplace and Licenses
 - a. Validating a user license to use an app
5. App Manifests (Application Runtime Environment)
 - a. Several examples of app manifest files with detailed description of the structure
6. Data Prefetching
 - a. Prefetching data from a remote service based on the current geospatial position of the device
7. Media Streaming / Playback
 - a. Receiving and playing back an internet radio stream
 - b. Playing back a pre-recorded sound
8. Service Invocation / Accessing Data
 - a. Invoking a generic RESTful data service
 - b. Invoking a generic RESTful backend service

8.1.6 Summary

The Application Design Studio is a standalone development tool based on the industry-leading Android Studio IDE from the maintainer of the Android platform Google, which is being actively developed and supported. The Application Design Studio will provide all features a developer would expect from a modern tool. While not interacting directly with other SIMPLI-CITY components, the Application Design Studio will provide a wealth of HowTo help documents, containing examples of solving the most typical SIMPLI-CITY scenarios as well as precompiled libraries for accessing these components for the convenience of the developer.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 414 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

8.2 Service Development API

8.2.1 Major Design Decisions

The Service Development API is a component aiming at third party developers that allows them to create and configure their own services running in the SIMPLI-CITY Service Runtime Environment and use them in apps running in the PMA.

The main objective of the Service Development API component is to provide an API that allows service developers to create and manage services in the SIMPLI-CITY Mobility Services Framework. As part of this API, it provides functionalities to handle service bundles and their metadata management operations from outside the Service Runtime Environment. The actual CRUD functionalities to manage services in the Service Registry is provided by the Service Registry (see Section 6.4), but the Service Development API implements a process to validate the uploaded service bundles before they are stored in the Service Registry.

The Service Development API is aimed to be used during development phase, enforcing a standard taxonomy for services and providing a set of interfaces to work at runtime, and deployment phases of the bundles, applying a set of constraints and checks prior the submission of services to the Service Registry and service execution in the Service Runtime Environment. The API functionalities will be provided through a plugin for an Integrated Development Environment (IDE).

During the discussion of the Global Architecture (deliverable D3.1) and Functional Specification (deliverable D3.2.1), the following major design decisions have been made:

Service-Management Operations:

The Service Development API will provide a defined set of operations to create and manage services. The created services will be stored in the Service Registry, which also offers the necessary CRUD functionalities (see Section 6.4). Nevertheless, the Service Development API will perform validation operations to check the correct implementation of each service before storing it in the Service Registry.

Service Manifest:

The service Manifest is an XML file containing all the associated SIMPLI-CITY service metadata. Structured by a set of XSDs (XML Schema Definitions), it will contain all the information of the service in SIMPLI-CITY. The full service Manifest data model is presented in Section 9.1.

The Service Development API will validate the structure and the semantics exposed in this Manifest during service deployment time.

Integration and Communications:

The Service Development API will provide several mechanisms to allow communication among applications and different types of services.

- A *REST Proxy* (part of the Service Runtime Environment, Section 6.1.5.2) makes all developed services consumable by apps through a RESTful interface. It also calls the Monitoring component and checks the SLA compliance.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 415 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

- *Service Templates* are customizable artifacts containing custom code developed in a programming language different from Java. Service Templates will make it easier to access External and Data Services providing a way for (external) developers to utilize a programming language of their choice for retrieving data from the externally located data service and seamlessly integrating this code into the SIMPLI-CITY service model. The concept of Service Templates is discussed in depth in the corresponding paragraph in the Section 8.2.3.2.

Data Services:

Table 93: Supported Data Service Runtimes

Language	Runtime
perl	Perl 5.18 r7+
python	Jython 2.5.3+ and SciPy 0.12.0+ (not including IPython)
xslt	Saxon HE 9.5.1.1+

The main purpose of Service Templates is to deal with data sources as provided by the SIMPLI-CITY Mobility Data as a Service concept (see Section 5), so third-party service developers will be able make use of data sources independent from their underlying technologies.

In order to perform this language-agnostic data handling, a representative selection (see Table 93) of languages is provided to data service developers.

Service Development IDE:

To develop services within the scope of the SIMPLI-CITY project, an IDE has to be provided. Next to the standard functionality of modern IDEs like code completion and real-time syntax checking, the IDE has to support SIMPLI-CITY-specific functionality (like already pre-packaged libraries or a help centre with SIMPLI-CITY-related documentation and HowTos).

As a modern IDE is a very complex piece of software, it would make sense to choose the best of the existing IDEs conforming to project criteria and enrich it with extra functionality for SIMPLI-CITY developer support by means of plugins.

8.2.2 Technology Comparison

8.2.2.1 Comparison Criteria

The technology for a similar IDE with plugins supporting the necessary functionality has already been discussed in Section 8.1 in order to find a foundation for the Application Design Studio. The goal was to choose the most developer-friendly, actively supported and feature-rich IDE, preferably backed by the maintainer of the Android platform, and the Android Studio has finally been chosen as the basis of the Application Developer Studio. In order to avoid double efforts, there will be no separate technology comparison for the Service Development API.

The Android Studio itself is based on the well-known and highly regarded (in the developer community) IDE IntelliJ IDEA (from the company JetBrains). IntelliJ has a modular architecture and supports custom plugins.

8.2.3 Technology Selection

8.2.3.1 Selection for IDE

It seems reasonable to reuse the same top-notch platform for the Application Design Studio as well as for the here discussed Service Development API. Thus, SIMPLI-CITY partners developing plugins for the Application Design Studio and Service Development API can reuse each other's knowledge and source code.

Android support provided by the IDE, chosen for the Application Design Studio in the Section 8.1.3.1 (Android Studio by Google) is quite specific, however the general programming support, required for development of SIMPLI-CITY services, is fully present in that IDE as well. Android Studio is an IDE built on top of the well-known IntelliJ IDEA IDE, hence the service developers will use the functionality provided by the base IDE (as they do not need Android support for development of services). As the result, the consortium has selected the Android Studio (and consequently its basis IntelliJ IDEA) as the foundation for the Service Development IDE. Nevertheless, service developers may choose to use another IDE – in this case, however, SIMPLI-CITY-specific functionalities including uploading the services to the Service Registry will not be supported and have to be done manually.

8.2.3.2 Missing Elements and Implementation Needs

The Service Development API needs to go beyond the standard functionalities provided by the selected IDE. Thus, the remaining logic to fulfil the required needs must be hand-made, enhancing and coordinating such provided functionalities used as a development framework. Many of the missing functionalities are not IDE related and have to be implemented from scratch and will run on the SIMPLI-CITY server.

Metadata Validation:

Metadata validation involves executing an ordered set of rules, involving advanced Java class data manipulation. The Service Development API runs a specific set of validations that the services must pass, whenever they are persisted, in order to register only those services that are conformant. The list below describes also the actions to do when the validation is not met.

Metadata validation involves executing an ordered set of rules, involving advanced Java class data manipulation. The Service Development API runs a specific set of validations that the services must pass, whenever they are persisted, in order to persist only those services that fulfil these rules. This validation occurs on the server side and is done by the Service Development API component (see Figure 36).

- Check the service descriptor conforms to its XSD schema
- Check service name is not reserved (e.g., not eu.simplicity, eu.simpli-city, etc.)
- Check all datasource UUIDs exists in the server
- Check all sensor UUIDs exists in the server

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 417 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

- Check the service SLA is valid
- Check the SLAs of other SIMPLI-CITY services used by the service are tighter than the SLA constraints of the service
- Check for the presence of marketplace information (description of the service, image icon, developer's contacts)
- Check for existence and well-formedness of /META-INF/simpli-city/service.xml in the bundle being deployed
- Check for non-existence of /OSGI-INF folder in the bundle being deployed
- Check for existence of /META-INF/manifest.mf in the bundle being deployed
- Check that no other versions of the bundle being deployed exist previously
- Check that service-development-api.jar is not bundled inside

Service Templates:

External and Data Services are a special type of services that handle information from external data sources. Example of an external data source could be a weather information website with a webpage containing historical temperature readings. As it is possible that such data sources do not have a defined API and even more could be maintained by external developers without knowledge of Java (the main programming language used in SIMPLI-CITY), but rather some other language, e.g., Python or Perl, it was foreseen to allow them to write code allowing SIMPLI-CITY to get access to their sources using their familiar programming language. A developer of the imaginary weather website above knows the format of their webpage and can write code in, a programming language, e.g., Perl, they know, that properly extracts that data for further use in SIMPLI-CITY.

For that reason the concept of Service Templates will be implemented. A Service Template is an XML file in predefined format that contains fragments of code in the programming language supported by the external developers. That is, service developers will develop parts of code that accesses the external datasource and returns this data in the format SIMPLI-CITY understands. Service developers do not need to know Java or internal SIMPLI-CITY structures, but rather have to follow format of the provided Service Template XML file. Once delivered, the Service Template XML would be deployed as part of the corresponding service in the OSGi container. Using Java libraries for interoperability with other languages, the container will execute the custom code provided in the Service Template XML file (say, Perl code, since Java has support for many such languages) and then the Perl code would access the datasource the right way and the results will be transferred back to the consumer of the SIMPLI-CITY service.

Service Templates –Sandbox:

Access to External and Data Services is done via execution of the custom code provided in the Service Template XML in a programming language different from Java (see the explanation of the Service Template concept in the paragraph above). Hence, the foreign code has to run in isolation from the main SIMPLI-CITY code to protect SIMPLI-CITY from possible errors in that code such as memory leaks or malicious code. For that reason, such services have to be executed in a so-called sandbox, where there will not be access or possibility to harm SIMPLI-CITY services running in the same container. Some of these code fragments can be excluded from sandboxing, for instance XSLT (XML transformations), as they are native to Java and are considered harmless in the current project model.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 418 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

Service Development IDE Integration – IntelliJ IDEA Plugin:

While providing a rich development platform, IntelliJ IDEA lacks some important functionality needed by developers of SIMPLI-CITY services. The consortium is going to implement the following additional functionality in the form of an IDE plugin to deliver a complete solution ready to be used in real life development:

- Service XML metadata file. A separate editor supporting the metadata format will be provided, including syntax checking.
- Bundling of the service into a deliverable JAR file ready for submission to the Service Registry via the RESTful interface of the Service Development API.
- Uploading the deliverable JAR to the Service Registry via the RESTful interface of the Service Development API.

8.2.3.3 Further Information and Conclusion from Technology Comparison

The technologies used for the implementation of the server side of the Service Development API will be the same as the ones used by other server side components of SIMPLI-CITY, like OSGi R4 as the underlying runtime for the execution of the Service Development API (see Section 6.1.2), and REST and JSON (see Sections 4.3 and 4.4) for data exchange. The component will provide an IntelliJ IDEA plugin to service developers as an IDE for the development of services.

8.2.4 Component Structure

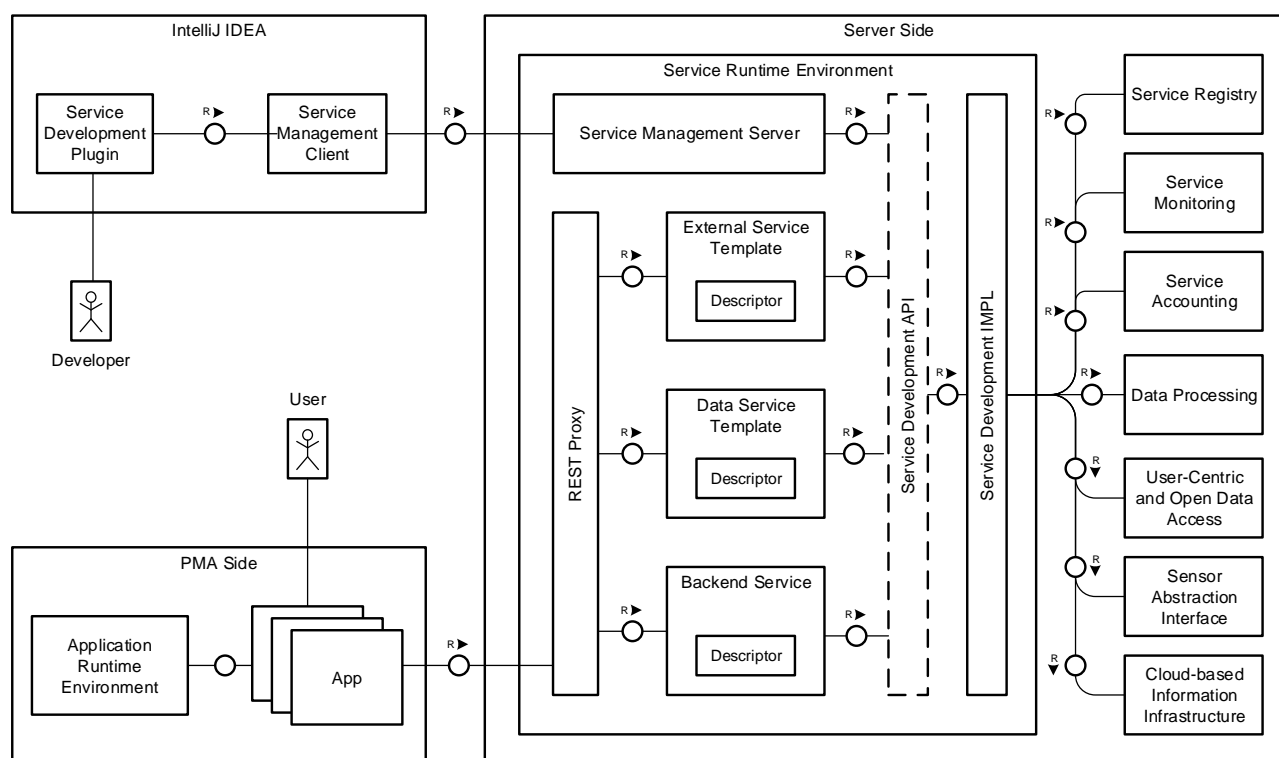


Figure 36: Component Structure of the Service Development API

Figure 36 shows the component diagram of the Service Development API. The component structure is an evolution of the structure presented in the Functional Specification.

Service Development IDE:

The consortium will extend functionality of the IntelliJ IDEA by supplying a plugin for the abovementioned functionality, which is the service metadata XML editor and bundling the developed service into a deliverable JAR. In addition, a convenient way for uploading the deliverable to the Service Registry will be facilitated by the plugin.

Service Management Client:

The Service Management Client is composed of a bundle which is a standard JAR that connects directly to the Service Management Server REST interface, helping the Service Development API plugin to interact with.

This component implements the bundle management RESTful interface provided by the Service Management Server subcomponent. It also interacts with the Service Registry for the rest of service management CRUD operations.

Service Management Server:

The Service Management Server is an OSGi bundle that runs inside the Service Runtime Environment. It exposes an interface that provides bundle persistence functionality.

Beyond the REST request handling, the functionalities are all delegated to the Service Registry.

REST Proxy:

As part of the Service Runtime Environment, a single REST Proxy will be provided for SIMPLI-CITY services running in the OSGi container, which will allow the communication with the apps running in the PMA.

External or Data Service Templates:

The External or Data Service Template is a special SIMPLI-CITY service that is bundled together with the Service Template XML file, provided by (external) developers and containing parts in a programming language different from Java, to access respective external data sources and retrieve data in the format, which can be used by native SIMPLI-CITY service consumers.

Backend Service:

These services are standard OSGi bundles composed by the following third-party developed items:

- Standard Java classes
- A service XML descriptor (i.e., the service Manifest)

Service Development API:

This module exposes functionalities to be used during service development time. These functionalities can be used by service developers by means of a helper library provided as a JAR file with a set of Java interfaces.

The exposed functionality covers the following topics: context-related functionalities, data management, sensor information, service management and unit handling.

- Context-related functionality allows the usage of service personalisation as described in Section 6.3.
- Data management leverages cloud information management operations provided by the Cloud-based Information Infrastructure component.
- Sensor information functionality permits to access and make use of vehicles' and other entities' sensor information.
- Service management provides notifications about service lifecycle events and metadata validation methods.
- Unit handling takes care of automatic and semi-automatic unit conversion and comparison of several units.

Service Development IMPL:

This OSGi bundle contains the implementation of the whole server-side functionality provided by the Service Development API.

8.2.5 Interfaces

8.2.5.1 RESTful Interfaces

The Service Management Server functionalities are provided by a RESTful interface that works as the endpoint for the Service Management Client. This RESTful interface is used by the IntelliJ IDEA Plugin to communicate with the server side subcomponents.

8.2.5.1.1 Persist a Bundle

This method of the Service Development API allows a service developer to persist (save or update) a service bundle in the Service Registry. Each bundle contains all the required information of a service as defined in the service Manifest (see Section 9.1). The method returns the generated ID (UUID) of the service. Every submitted bundle will be validated before being registered.

Table 94: RESTful Interface Description – Bundle Persistence

Method	POST or PUT	URL	\$API_ROOT/bundle?:replace				
Description	Uploads / persists the bundle posted in the message body.						
Parameter	replace	Required	no	Possible Values	Boolean	Description	Persistence replacing mode. Defaults to true .
Example URL	\$API_ROOT/bundle?replace=true						
Response	HTTP status code + JSON object						
HTTP Status Code		Required	yes	Possible Values	200	Description	Bundle replaced
					201		Bundle created
					400		Bad request: Invalid bundle or parameter
					502		Bundle could not be uploaded
JSON Object	http://SIMPLI-CITY.eu/sdapi/JSON-Schema/BundlePersistence						
JSON Attribute	serviceld	Required	yes	Possible Values	any String	Description	Id of the affected service, as a 128-bit UUID
Example Response	HTTP/1.1 200 OK						

Listing 280: JSON Example: Return Value for Bundle Persistence

```
{
  "serviceId": "528739A0-F508-4551-A12A-04A9B51718D0"
}
```

8.2.5.2 Library Interfaces for Service Developers

The Service Development API will provide a helper library (JAR file) with a set of auxiliary Java classes to be used by service developers during service development time to increase their productivity and avoid common errors. These classes cover several topics like access to context and sensor information, data and services management, and unit handling.

8.2.6 Summary

The Service Development IDE is a standalone development tool comprised of a Service Development IDE plugin combined with the IntelliJ IDEA IDE. It will provide all features a service developer will need in order to develop and submit services to the SIMPLI-CITY Service Marketplace as well as directly persisting them in Service Registry through the backend part of the Service Development API with all corresponding validations.

Moreover, it will provide a helper library as a JAR file with a set of interfaces to be used by service developers during development time.

In addition, through the IntelliJ IDEA plugin, SIMPLI-CITY will provide guidelines, HowTos, examples etc. for service developers.

9 Manifest Data Models

9.1 Service Manifest Model

Table 95 and Listing 281, the data model for the service descriptions, i.e., the service Manifest is presented. It is used to describe services and their capabilities in SIMPLI-CITY. Since the data model is rather complex, only the top level entities are presented. Further, during the development phase of the project it is very likely that further entries will be added or changed. However, this is not an issue, since the implementations takes into account extendability of the service Manifest data model.

The service description is crucial for the SIMPLI-CITY Mobility Services Framework, especially the Monitoring component (see Section 6.2), the Service Registry (see Section 6.4, Context-based Service Personalisation (see Section 6.3), the Service Marketplace (Section 6.5), and the Service Development API (see Section 8.2).

Table 95: Preliminary Service Manifest Data Model

Field Name	Comments
Name	This mandatory property contains the name of the service. All types of services have to contain this property to be identified and properly accessed by other components of SIMPLI-CITY.
Version	This mandatory property specifies the version of the service. All types of services have to contain this property to be identified and properly accessed by other components of SIMPLI-CITY.
Description	This optional property that specifies human-readable description of the service. Any type of service can contain this property.
Keywords	This optional property specifies a collection of keywords that can be associated with the corresponding service. Any type of service can contain this property.
Runtime Version	This mandatory property specifies a range of Service Runtime Environment versions that the service is compatible with. All types of services have to contain this property to be identified and properly accessed by other components of SIMPLI-CITY.
SLA	This optional complex property describes the set of SLA (Service Level Agreement) requirements to this service. Any type of service can contain this property. SLA property allows defining service availability requirements, response time restrictions and throughput obligations.
License	This optional complex property describes the licensing restrictions of the service. Any type of service can contain this property. If licensing restrictions are specified, they must be enforced by other components of SIMPLI-CITY.
Marketplace Info	This optional complex property describes marketplace-related information of the service. Any type of service can contain this property.
Import	This optional complex property describes a set of features, datasources and sensors service requires access to. Any type of service can contain this property.

Export	This mandatory property defines a set of interfaces that define functionality that can be accessible from other components. Only Backend Services are allowed to have this property.
External	This mandatory property defines a set of SOAP- and RESTful-based web-services that a described service operates with. Only External Services are allowed to have this property
Data	This obligatory property defines a data source that a corresponding service can operate with. Only Data Services are allowed to have this property.

Listing 281: Preliminary Service Manifest Data Model (XSD)

```

<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" version="1.0">
  <xs:element name="BackendServiceDescription"
type="backendServiceDescription" />
  <xs:element name="DataServiceDescription" type="dataServiceDescription" />
  <xs:element name="ExternalServiceDescription"
type="externalServiceDescription" />
  <xs:element name="ServiceDescription" type="serviceDescription" />
  <xs:complexType name="serviceDescription" abstract="true">
    <xs:sequence>
      <xs:element name="name" type="xs:string" />
      <xs:element name="version" type="xs:string" />
      <xs:element name="description" type="xs:string" minOccurs="0" />
      <xs:element name="keywords" minOccurs="0">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="keyword" type="xs:string" minOccurs="0"
maxOccurs="unbounded" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="runtimeVersion" type="xs:string" />
      <xs:element name="import" type="import" minOccurs="0" />
      <xs:element name="sla" type="sla" minOccurs="0" />
      <xs:element name="license" type="license" minOccurs="0" />
      <xs:element name="marketPlaceInfo" type="marketPlaceInfo"
minOccurs="0" />
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="import">
    <xs:sequence>
      <xs:element name="features" minOccurs="0">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="feature" type="feature" minOccurs="0"
maxOccurs="unbounded" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="datasources" minOccurs="0">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="datasource" type="xs:string"
minOccurs="0" maxOccurs="unbounded" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="sensors" minOccurs="0">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="sensor" type="xs:string" minOccurs="0"
maxOccurs="unbounded" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>

```

```

</xs:complexType>
<xs:complexType name="sla">
  <xs:sequence>
    <xs:element name="throughput" type="xs:double" />
    <xs:element name="responseTime" type="xs:long" />
    <xs:element name="availability" type="xs:double" />
  </xs:sequence>
</xs:complexType>
<xs:complexType name="license">
  <xs:sequence>
    <xs:element name="serviceID" type="xs:string" minOccurs="0" />
    <xs:element name="userID" type="xs:string" minOccurs="0" />
    <xs:element name="freeService" type="xs:boolean" minOccurs="0" />
    <xs:element name="usageCosts" type="usageCosts" minOccurs="0" />
    <xs:element name="limitations" type="limitations" minOccurs="0" />
  </xs:sequence>
</xs:complexType>
<xs:complexType name="usageCosts">
  <xs:sequence>
    <xs:element name="unit" type="unit" minOccurs="0" />
    <xs:element name="costs" type="xs:int" />
  </xs:sequence>
</xs:complexType>
<xs:complexType name="limitations">
  <xs:sequence>
    <xs:element name="maxInvocationTimeUnit"
type="maxInvocationTimeUnit" minOccurs="0" />
    <xs:element name="maxInvocations" type="xs:int" />
  </xs:sequence>
</xs:complexType>
<xs:complexType name="marketPlaceInfo">
  <xs:sequence />
</xs:complexType>
<xs:complexType name="backendServiceDescription">
  <xs:complexContent>
    <xs:extension base="serviceDescription">
      <xs:sequence>
        <xs:element name="export" type="export" />
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="export">
  <xs:sequence>
    <xs:element name="interfaces">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="interface" type="interface"
maxOccurs="unbounded" />
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>

```

```

<xs:complexType name="interface">
  <xs:sequence>
    <xs:element name="class" type="xs:string" />
  </xs:sequence>
</xs:complexType>
<xs:complexType name="dataServiceDescription">
  <xs:complexContent>
    <xs:extension base="serviceDescription">
      <xs:sequence>
        <xs:element name="data" type="data" />
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="data">
  <xs:sequence>
    <xs:element name="uuid" type="xs:string" />
    <xs:element name="handler" type="handlerType" />
    <xs:element name="transformers" minOccurs="0">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="transformer" type="transformer"
minOccurs="0" maxOccurs="unbounded" />
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="transformer">
  <xs:sequence>
    <xs:element name="id" type="xs:string" />
    <xs:element name="type" type="handlerType" />
  </xs:sequence>
</xs:complexType>
<xs:complexType name="externalServiceDescription">
  <xs:complexContent>
    <xs:extension base="serviceDescription">
      <xs:sequence>
        <xs:element name="external" type="external" minOccurs="0" />
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="external">
  <xs:sequence>
    <xs:element name="applications" minOccurs="0">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="application" type="xs:string"
minOccurs="0" maxOccurs="unbounded" />
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="wsdls" minOccurs="0">
      <xs:complexType>
        <xs:sequence>

```

```

        <xs:element name="wsdl" type="xs:string" minOccurs="0"
maxOccurs="unbounded" />
    </xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
<xs:simpleType name="feature">
    <xs:restriction base="xs:string">
        <xs:enumeration value="NETWORK_ACCESS" />
        <xs:enumeration value="DYNAMIC_DATASOURCES" />
        <xs:enumeration value="DYNAMIC_SENSORS" />
    </xs:restriction>
</xs:simpleType>
<xs:simpleType name="unit">
    <xs:restriction base="xs:string">
        <xs:enumeration value="Invocation" />
        <xs:enumeration value="Day" />
        <xs:enumeration value="Week" />
        <xs:enumeration value="Month" />
        <xs:enumeration value="Year" />
        <xs:enumeration value="Unlimited" />
    </xs:restriction>
</xs:simpleType>
<xs:simpleType name="maxInvocationTimeUnit">
    <xs:restriction base="xs:string">
        <xs:enumeration value="Invocation" />
        <xs:enumeration value="Day" />
        <xs:enumeration value="Week" />
        <xs:enumeration value="Month" />
        <xs:enumeration value="Year" />
    </xs:restriction>
</xs:simpleType>
<xs:simpleType name="handlerType">
    <xs:restriction base="xs:string">
        <xs:enumeration value="FREEMAKER" />
        <xs:enumeration value="GROOVY" />
        <xs:enumeration value="JAVASCRIPT" />
        <xs:enumeration value="LUA" />
        <xs:enumeration value="PYTHON" />
        <xs:enumeration value="RUBY" />
        <xs:enumeration value="STRINGTEMPLATE" />
        <xs:enumeration value="VELOCITY" />
        <xs:enumeration value="XSLT" />
    </xs:restriction>
</xs:simpleType>
</xs:schema>

```

9.2 App Manifest Model

An app is described via the Manifest file stored in the App Registry. The Manifest contains all relevant information for the App Marketplace to generate a marketing view for the app as well as the needed information to install an app on the customer's PMA.

Table 96: Preliminary App Manifest Data Model

Field Name	Comments
Basic Data	This mandatory group contains the basic information to identify and describe the app.
Used Services	Contains a list of the used services for an app (identified by IDs), along with the used version of these services.
Files	Contains a list of all files delivered with this app. This list is mandatory to help the Market Data Management subcomponent (see Section 6.5) to generate dynamic updates for customers. An example of these files is the grammar to be used by the Dialogue Interface and, eventually, an update of the taxonomy employed by the app.
Resources	This holds a list of resources used for the app. Resources must be part of the files inside the Files list and may be used for promotional reasons (e.g., screenshots, icons).

In the following listing, the app Manifest will be described in more details: Listing 282 provides the XSD for the app Manifest data model, while Listing 283 and Listing 284 provide examples in XML and JSON, respectively.

Listing 282: App Manifest Data Model (XSD)

```

<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="application">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="app">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="author">
                <xs:annotation>
                  <xs:documentation>Basic data</xs:documentation>
                </xs:annotation>
                <xs:complexType>
                  <xs:sequence>
                    <xs:element type="xs:string" name="id"/>
                    <xs:element type="xs:string" name="name"/>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
              <xs:element type="xs:string" name="description"/>
              <xs:element type="xs:string" name="keywords"/>
              <xs:element type="xs:string" name="name"/>
              <xs:element type="xs:string" name="namespace"/>
              <xs:element type="xs:string" name="version"/>
              <xs:element name="icon">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element type="xs:string" name="resource_id"/>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
              <xs:element name="properties">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element type="xs:float" name="price"/>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
              <xs:element type="xs:string" name="reviewed"/>
              <xs:element name="used_services">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="service" maxOccurs="unbounded"
minOccurs="0">
                      <xs:complexType>
                        <xs:sequence>
                          <xs:element type="xs:string" name="namespace"/>
                          <xs:element type="xs:string" name="version"/>
                        </xs:sequence>
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
              <xs:element name="files">

```

```

        <xs:complexType>
          <xs:sequence>
            <xs:element name="file" maxOccurs="unbounded"
minOccurs="0">
              <xs:complexType>
                <xs:sequence>
                  <xs:element type="xs:string" name="id"/>
                  <xs:element type="xs:string" name="name"/>
                </xs:sequence>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="resources">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="resource">
              <xs:complexType>
                <xs:sequence>
                  <xs:element type="xs:string" name="file_id"/>
                  <xs:element type="xs:string" name="type"/>
                </xs:sequence>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>

```

Listing 283: App Manifest Example (XML)

```

<?xml version="1.0" encoding="UTF-8"?>
<application>
  <app>
    <!-- Basic data -->
    <author>
      <id>6579e96f76baa00787a28653876c6127</id>
      <name>John Doe</name>
    </author>
    <description>Lorem Ipsum Dolor Sit Amet</description>
    <keywords>sample, lorem, ipsum, example, application</keywords>
    <name>Sample Application</name>
    <namespace>eu.simplicity.app.sample_application</namespace>
    <version>0.1-RC1</version>
    <icon>
      <resource_id>826e8142e6baabe8af779f5f490cf5f5</resource_id>
    </icon>
    <properties>
      <price>2.99</price>
    </properties>
    <reviewed>true</reviewed>
    <used_services>
      <service>
        <namespace>eu.simplicity.svc.sample-trading</namespace>
        <version>1.2</version>
      </service>
      <service>
        <namespace>eu.simplicity.svc.sample-routing</namespace>
        <version>0.5</version>
      </service>
      <service>
        <namespace>eu.simplicity.svc.sample-emailing</namespace>
        <version>3.2.2</version>
      </service>
    </used_services>
    <files>
      <file>
        <id>1c1c96fd2cf8330db0bfa936ce82f3b9</id>
        <name>application.war</name>
      </file>
      <file>
        <id>826e8142e6baabe8af779f5f490cf5f5</id>
        <name>logo.png</name>
      </file>
      <file>
        <id>4f618ab239c03befffa89ddab54247a5</id>
        <name>grammar.gr</name>
      </file>
    </files>
    <resources>
      <resource>
        <file_id>826e8142e6baabe8af779f5f490cf5f5</file_id>
        <type>image/png</type>
      </resource>
    </resources>
  </app>
</application>

```


Listing 284: App Manifest Example (JSON)

```

{
  "application": {
    "app": {
      "author": {
        "id": "6579e96f76baa00787a28653876c6127",
        "name": "John Doe"
      },
      "description": "Lorem Ipsum Dolor Sit Amet",
      "keywords": "sample,lorem,ipsum,example,application",
      "name": "Sample Application",
      "namespace": "eu.simplicity.app.sample_application",
      "version": "0.1-RC1",
      "icon": {
        "resource_id": "826e8142e6baabe8af779f5f490cf5f5"
      },
      "properties": {
        "price": "2.99"
      },
      "reviewed": "true",
      "used_services": {
        "service": [
          {
            "namespace": "eu.simplicity.svc.sample-trading",
            "version": "1.2"
          },
          {
            "namespace": "eu.simplicity.svc.sample-routing",
            "version": "0.5"
          },
          {
            "namespace": "eu.simplicity.svc.sample-emailing",
            "version": "3.2.2"
          }
        ]
      },
      "files": {
        "file": [
          {
            "id": "1c1c96fd2cf8330db0bfa936ce82f3b9",
            "name": "application.war"
          },
          {
            "id": "826e8142e6baabe8af779f5f490cf5f5",
            "name": "logo.png"
          },
          {
            "id": "4f618ab239c03beffffa89ddab54247a5",
            "name": "grammar.gr"
          }
        ]
      },
      "resources": {
        "resource": {
          "file_id": "826e8142e6baabe8af779f5f490cf5f5",
          "type": "image/png"
        }
      }
    }
  }
}

```

10 Conclusion

This deliverable defines the detailed Technical Specification of the SIMPLI-CITY software framework with regard to individual software components and their interfaces. Based on the Requirements Analysis Report (deliverable D2.2), the Global Architecture Definition (D3.1), and – most importantly – the Functional Specification (D3.2.1), the Technical Specification provides the foundation for the Research, Technology, and Development (RTD) work to be carried out in SIMPLI-CITY. Notably, this deliverable should be read together with the Holistic Security and Privacy Concept (deliverable D3.3), which complements the Technical Specification.

Each of the SIMPLI-CITY software components is analysed with respect to these aspects:

- **Major Design Decisions:** Technology-independent software design decisions that define the scope and general direction of the discussed SIMPLI-CITY software component.
- **Technology Comparison:** Comparison of technologies which could be applied as a foundation for the respective software component. The comparison is based on generic and component-specific criteria defined in the Functional Specification (deliverable D3.2.1) and further refined within this Technical Specification.
- **Technology Selection:** Selection of technology/technologies based on the aforementioned comparison. Furthermore, missing elements are identified. This lays the foundation for the necessary development work within the project.
- **Component Structure:** An update of the components' structure from the Functional Specification (deliverable D3.2.1). The update reflects the continuation in the software engineering process. To depict the structure, FMC notation has been used.
- **Interfaces:** Based on the used programming languages, all software components provide Java (respectively Go and C++) and/or RESTful interfaces. For each interface, the necessary input and provided output is defined. Furthermore, example request and response JSON messages for the interaction with RESTful interfaces are provided.
- **Content Format:** Includes the definition of message format *schemas* for interaction with the respective interfaces (in contrast to the *examples* provided as part of the interface descriptions).

As a result, for all software components in SIMPLI-CITY, it is clearly defined which functionalities to expect and how to access the provided functionalities.

Throughout the technical specification of the individual software components, the SIMPLI-CITY consortium has been able to identify previously existing inconsistencies between the components and remodel the internal and external structure of software components, wherever necessary. Nevertheless, naturally, during the course of the project, certain aspects of the software architecture may be changed for technical reasons and due to unforeseen issues. Nevertheless, this Technical Specification provides a stable and reliable foundation for the currently ongoing and upcoming software development tasks within SIMPLI-CITY. While not leading to additional deliverables of the SIMPLI-CITY project, changes to the Technical Specification will necessarily lead to an update of the document at hand. However, no major changes regarding the defined functionalities are foreseen.

D3.2.2_Technical_Spec_v1.00_EC_Approved.docx	Document Version: 1.00	Date: 2015-04-21	Status: Approved	Page: 434 / 435
http://www.simpli-city.eu/		Copyright © SIMPLI-CITY Project Consortium. All Rights Reserved. Grant Agreement No.: 318201		

References

- [ALE04] A. Alexandrescu, “Lock-free data structures,” *C/C++ User Journal*, 2004.
- [CFA03] H. Chen, T. Finin, and J. Anupam. “An intelligent broker for context-aware systems,” in *Adjunct Proceedings of The Fifth International Conference on Ubiquitous Computing*, vol. 3, Seattle, Washington, 2003, pp. 183-184.
- [EPS+01] F. Espinoza, P. Persson, A. Sandin., H. Nystrom, E. Cacciatore and M. Bylund, “GeoNotes: social and navigational aspects of location-based information systems,” in *Third International Conference on Ubiquitous Computing*, Atlanta, Georgia, USA, 2001, pp. 2–17.
- [GA07] G. Gigan and I. Atkinson, “Sensor Abstraction Layer: a unique software interface to effectively manage sensor networks,” in *3rd International Conference on Intelligent Sensors, Sensor Networks and Information (ISSNIP 2007)*, Melbourne, Australia, 2007, pp.479-484.
- [GLF+08] M. Girolami, S. Lenzi, F. Furfari and S. Chessa, “SAIL: A Sensor Abstraction and Integration Layer for Context Awareness,” in *34th Euromicro Conference on Software Engineering and Advanced Applications (SEAA '08)*, Parma, Italy, 2008, pp.374-381.
- [KP08] D. Kourtesis and I. Paraskakis, “Combining SAWSDL, OWL-DL and UDDI for Semantically Enhanced Web Service Discovery,” in *The Semantic Web: Research and Applications, 5th European Semantic Web Conference (ESWC 2008)*, Tenerife, Spain, 2008, pp. 614-628.
- [LM03] T. Liu and M. Martonosi, “Impala: A Middleware System for Managing Autonomic, Parallel Sensor Systems,” in *Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, California, 2003, pp. 107-118.
- [LSE+12] J. Levandoski, M. Sarwat, A. Eldawy, M. Mokbel, “LARS: A location-aware recommender system,” in *2012 IEEE 28th International Conference on Data Engineering*, Washington DC, 2012, pp. 450-461.
- [PCB00] N.B. Priyantha, A. Chakraborty and H. Balakrishnan, “The cricket location-support system,” in *6th Annual International Conference on Mobile Computing and Networking*, Boston, MA, 2000, pp.32–43.
- [PMS+11] A. Papageorgiou, A. Miede, D. Schuller, S. Schulte, and R. Steinmetz, “Always Best Served: On the behaviour of QoS- and QoE-based Algorithms for Web Service Adaption,” in *8th IEEE International Workshop on Managing Ubiquitous Communications and Services (PerCom Workshops 2011 – MUCS)*, Seattle, Washington, 2011, pp. 71-76.
- [SGV+06] E. Souto, G. Guimarães, G. Vasconcelos, M. Vieira, N. Rosa, C. Ferraz and J. Kelner, “Mires: a publish/subscribe middleware for sensor networks,” *Personal and Ubiquitous Computing*, vol. 10, no. 1, pp. 37-44, 2006.