simpli·city

The Road User Information System Of The Future

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

D3.2.1: Functional Specification

Deliverable Lead: TU Vienna

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

Delivery Date: 09/2013

Dissemination Level: Public

Version 1.10

This document provides an in-depth definition of all technical SIMPLI-CITY components and their subcomponents, including the functional specification of their functionalities, behaviours, and interactions.

Document Status	
Deliverable Lead	Stefan Schulte, TU Vienna
Internal Reviewer 1	Michaela Kargl, FGM
Internal Reviewer 2	Daniele Presta, CRF
Type	Deliverable
Work Package	WP3: Architecture, Functional & Technical Specification, Security & Privacy Concept, Integration
ID	D3.2.1: Functional Specification
Due Date	31.05.2013
Delivery Date	20.09.2013
Status	Approved

Document History	
Draft Version	V0.11, TUDA, 04.12.2012
Contributions	V0.20, TUV, ASC, TEMP, IBM, TUDA, TALK, TEMP V0.21, TUV V0.22, TUV, IBM, ASC V0.23, TUV, ASC, TUDA, IBM V0.25, TUV, ASC V0.26, TUV, ASC, TIE V0.27, ASC, TALK V0.28, ASC V0.30, TUV (2nd For Review Version) V1.00, TUV (approved by the internal reviewers)
Final Version	V1.10, TUV (Approved by the European Commission)

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

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

Tempos 21, Spain

Centro Ricerche FIAT, Italy

SRM – Reti e Mobilità, Italy

Executive Summary

Within this document, the functionalities of the SIMPLI-CITY software components are specified. 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), and the upcoming Technical Specification (D3.2.2), it provides the foundation for the Research, Technology, and Development work to be conducted within work packages WP4-WP6. The outcome of this Functional Specification is an in-depth functional specification of all SIMPLI-CITY software components.

Per se, the technical discussions within this document are done in a product- and software-independent way in order to avoid any preliminary decision for a particular technology. Instead, this deliverable provides the foundation for the actual technology and software selection within the subsequent Technical Specification (D3.2.2).

The deliverable starts with a recapitulation of the outcomes from the Global Architecture Definition (D3.1) and a basic explanation of the interactions between the single SIMPLI-CITY components. Afterwards, the functional specification of the SIMPLI-CITY Data Integration, Service Framework, Personal Mobility Assistant, and Developer Support software components are presented: An in-depth discussion of the functionalities of each component is provided by discussing their overall functional specifications, necessary subcomponents, related requirements (as defined in deliverable D2.3), interactions with other components, the provided (Graphical) User Interface (if applicable), the used data model (if applicable), and the parameters to take into account for the technical specification of the component.

Afterwards, the basic data models used in SIMPLI-CITY are introduced. The document ends with a summarisation of its outcomes.

Table of Contents

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 definition of all technical SIMPLI-CITY components, including the specification of their functionalities, behaviours, and interactions.

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, a 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 Service 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 modelled, 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.

1.2 Deliverable Purpose, Scope and Context

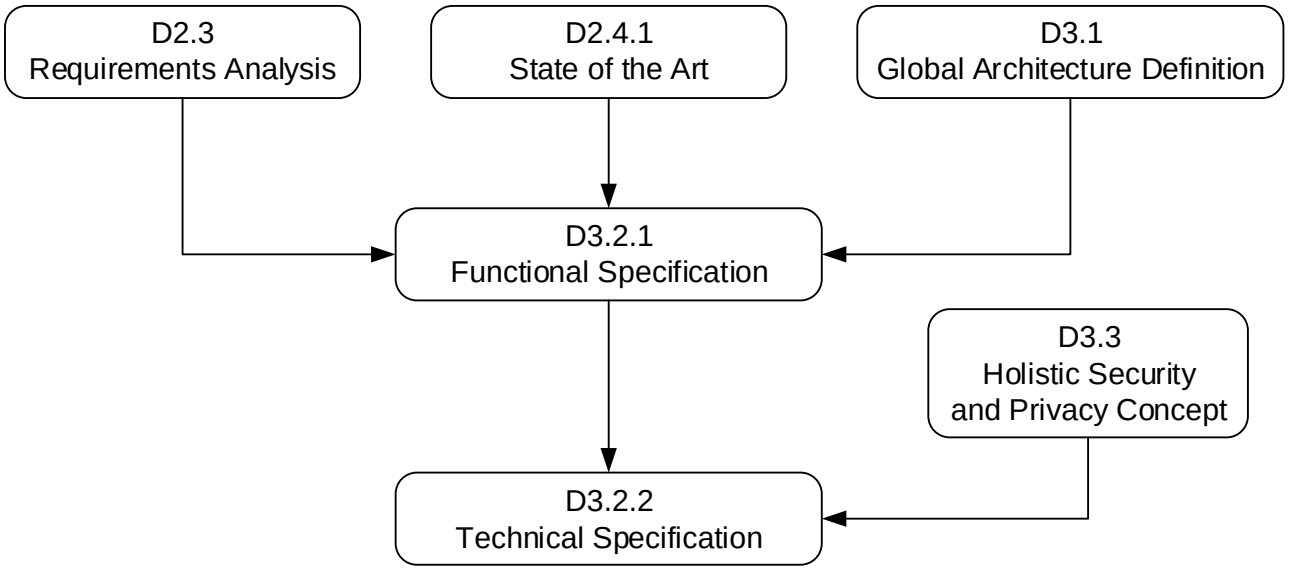

Figure 1: Context of Deliverable D3.2.1

The purpose of this document is to provide the detailed functional specification of all SIMPLI-CITY software components. At this purpose, this document recapitulates the high-level functional specification (from deliverable D3.1) as well as the identified requirements for each single component (from deliverable D2.3), and takes into account the current state of the art as identified in deliverable D2.4.1. Thereupon, this document defines the individual subcomponents, interaction patterns with other components, Graphical User Interfaces (optional), and necessary data models (optional), and finally the parameters to be taken into consideration for the Technical Specification of each component. In addition, the overall SIMPLI-CITY data models will be defined preliminary.

As depicted in Figure 1, this deliverable provides the foundation for the upcoming SIMPLI-CITY Technical Specification (D3.2.2).

1.3 Document Status and Target Audience

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

While the document primarily aims at 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.

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 to 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 provides a generic scenario of the software component interaction within SIMPLI-CITY, using Unified Modeling Language (UML) sequence diagrams.

Section 4 introduces the structure of the component descriptions in Sections 5-8.

Section 5 presents the functional specification of the data integration components, recapitulating their overall functional specification, introducing subcomponents, discussing related requirements and interactions with other components, providing mockups for Graphical User Interfaces (if applicable), and defining preliminary data models. Furthermore, parameters to be taken into account for technology/software selection in deliverable D3.2.2 are determined.

Section 6 does the same as Section 5 for the service framework components.

Section 7 does the same for the Personal Mobility Assistant.

Section 8 does the same for the software developer support components.

Section 9 provides an overview of the high-level data models applied in SIMPLI-CITY, explaining the rationale behind the different data models and where they will be applied.

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

2 System Overview

Within this section, a brief overview about SIMPLI-CITY is given from a technical perspective. In Section 2.1, the general architecture is summarized as a recapitulation from the deliverable D3.1 (Global Architecture Definition) outcomes. 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 at within SIMPLI-CITY. Thus, the context for the detailed component analysis and specification 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).

2.1 SIMPLI-CITY Architecture Summary

SIMPLI-CITY is supposed to be used by a large number of users from different countries using 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 generic – 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 covers 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 which 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.

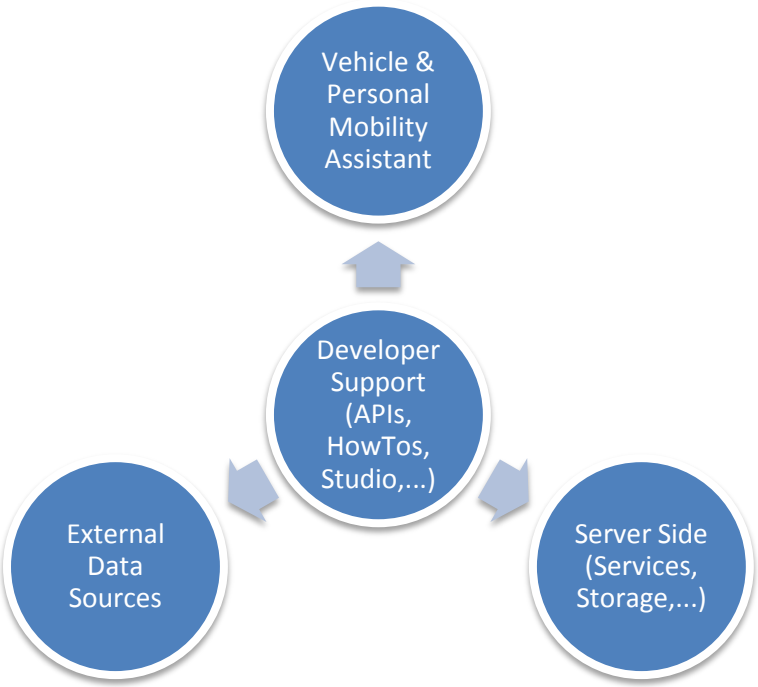

Figure 2: Core Areas of the SIMPLI-CITY Architecture

The “Vehicle & PMA” area focuses on the mobile device which 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 provided. User interaction is performed via the Multimodal User Interface and its 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 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

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 as 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 the component of the same name. 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 about the components and their interconnection as introduced in deliverable D3.1.

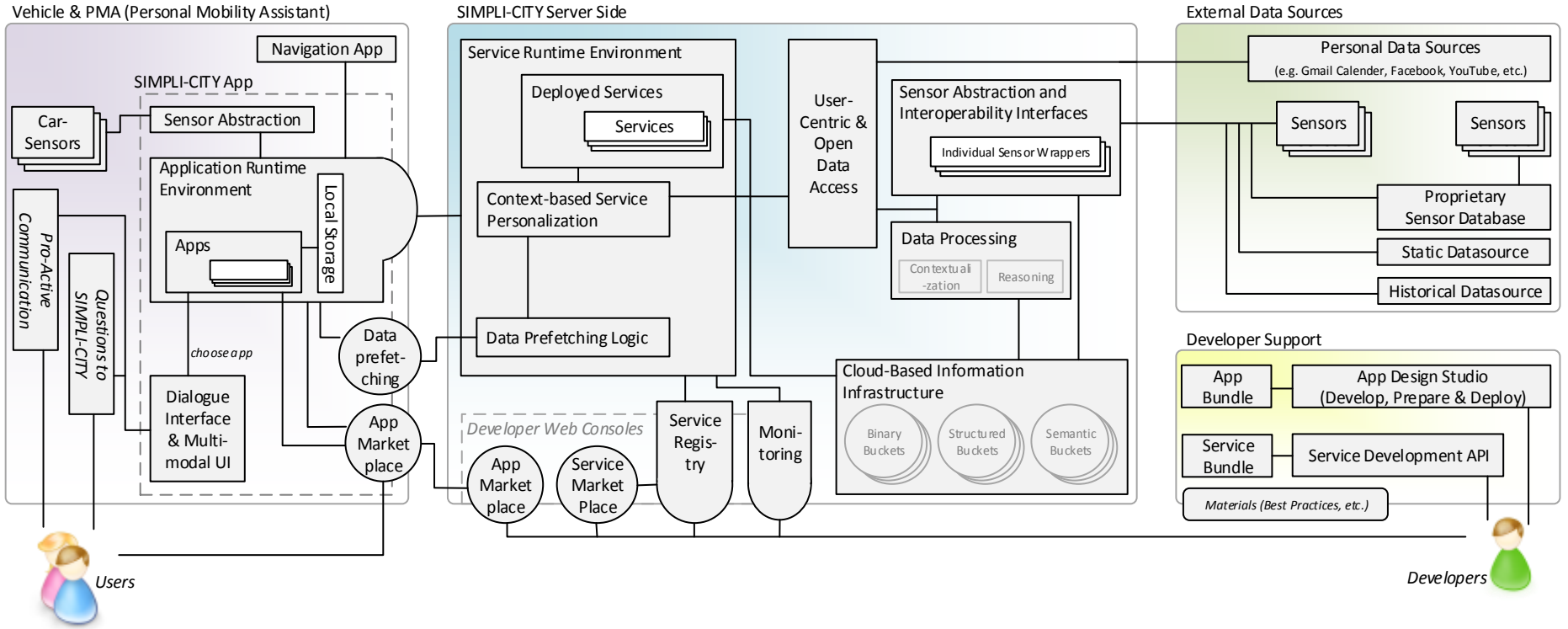

Figure 3: SIMPLI-CITY Architecture

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 employs 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
- Dialogue Interface and Multimodal User Interface.

The Application Runtime Environment provides 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 Dialogue Interface and the Multimodal User Interface is the second component with which the users directly interact. Thus, these 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 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. The Multimodal User Interface extends the Dialogue Interface with means of a GUI. 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 with finally allowing developers to correspondingly deploy their developed apps. For this purpose, different tools are offered within the Application Design Studio. These tools comprise guideline documents, 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 whole 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 own *services* in the context of SIMPLI-CITY, 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, as 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 an eased 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 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 created services. Furthermore, by realizing the approach of Mobility-related Data as a Service within SIMPLI-CITY, service developers get an easy 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, 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 as well 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 from event-based sensing systems that transmit data only in case a certain event occurred as well as proprietary databases that 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 as well as 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 provides as well homogeneous access possibilities to heterogeneous sensor data. However, the focus of the PMA-based Sensor Abstraction is on access possibilities of 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 analogous 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 allows 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 good 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 interrupted 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 different 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. As sometimes, and for some apps or services, data has to be stored as well on the PMA, a correspondingly adapted Local Storage is offered on the PMA, as well.

3 Generic Scenario

In the following, a generic scenario showing typical interactions between software components in SIMPLI-CITY is presented. In contrast to the high-level scenarios of deliverable D2.3, this scenario is presented on a much more technical level including sequence diagrams and component interaction descriptions. However, in contrast to the sequence diagrams that will be presented in Sections 5-7, the generic scenario only shows the interaction between the high-level software components of SIMPLI-CITY, but not between the subcomponents. It also shows only the most important interactions in order to help the reader to understand the generic approach used in SIMPLI-CITY.

The generic scenario is organized according to the single steps that need to be carried out in order to make use of an app and its underlying services. While an example is used to illustrate this, the interaction between the user and the system as well as single software components is per se generic. Importantly, the story below is *illustrative only.* SIMPLI-CITY is not bound to specific data sources or already existing services. SIMPLI-CITY, by design, is for potential use with arbitrary data sources and services.

3.1 Prerequisites

Jane Doe is an avid user of the SIMPLI-CITY PMA and has a number of apps she has downloaded and installed from the SIMPLI-CITY App Marketplace. Amongst others, she makes use of BestTransportMode, an app able to recommend the most appropriate transport mode for a particular trip.

BestTransportMode is able to automatically integrate calendar data in order to identify the place of departure and the destination of a particular trip, and calculate the most suitable transportation plan (train, car, bus, bike, walk, or a combination thereof) under consideration of time constraints, Jane's personal preferences, and carbon efficiency. Once the user has chosen a transport mode for a particular trip, BestTransportMode integrates the necessary routing information, informs the user about congestions, and recomputes its recommendation if necessary.

In the following, it is assumed that Jane has allowed BestTransportMode to make use of local data like her location as well as web-based data like her personal calendar. The app is also allowed to access further user profile data, e.g., that she owns a car, and information about social network accounts. Furthermore, it is assumed that Jane's smartphone has a stable internet connection – via WIFI at home and 3G or LTE on the road. Her mobile internet contract allows 1 Gigabyte of incoming traffic each month; hence, Jane is interested to minimize data transfer en route. All backend services needed to carry out BestTransportMode are registered in the Service Registry and are available in SIMPLI-CITY even if Jane will not directly have to care about this as she will only have to use the app within her PMA.

In the basic story, Jane wants to go to a football game at a specific venue. She has added this event to her online calendar weeks ago. The morning before the game, she wants to know what her best transport option is. In the following, simplified sequence diagrams will

be used in order to depict the scenario. For each major activity, one particular sequence diagram is depicted and the according interaction is explained.[1]

3.2 User Interaction (Input)

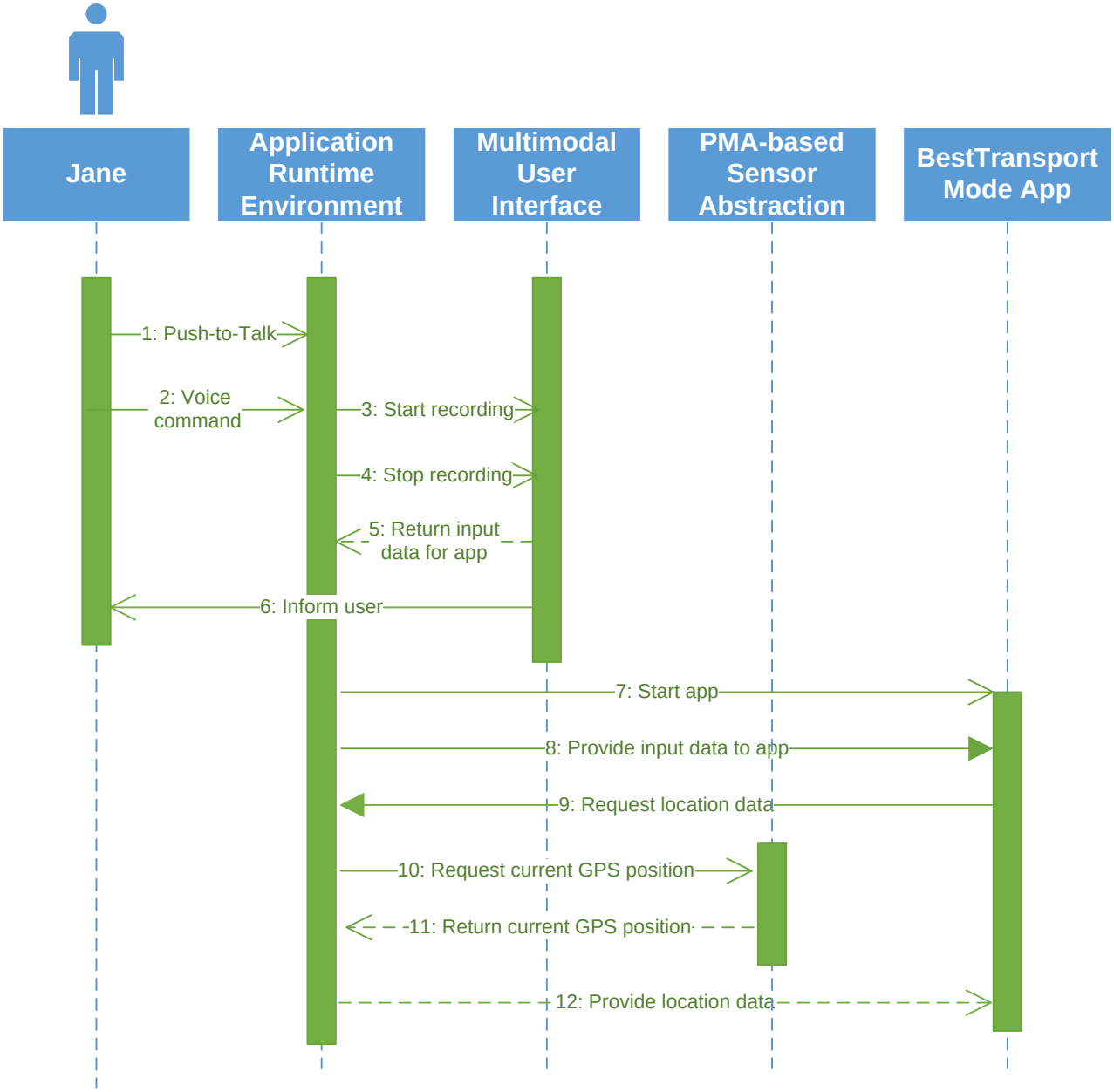

Figure 4: User Interaction – Input

In order to make use of the functionalities of an app like BestTransportMode, Jane needs to interact with the PMA, which is running on her smartphone, and give relevant spoken commands (voice utterances) to the PMA. The PMA will then recognize which app should

[1] The presentation of this particular story does not indicate that exactly this story will be part of the services and apps developed within the SIMPLI-CITY use case work packages WP7 and WP8. The definition of the services and end user apps to be implemented within the project will be done in the Preliminary (deliverables D7.1.1 and D8.1.1) and Final Use Case Specifications (deliverables D7.1.2 and D8.1.2).

be addressed by interpreting the command lists of the app manifests, which have been provided by app developers.

In the example depicted in Figure 4, the interaction with the BestTransportMode app is started through pressing the Speak With the PMA (SWP) Push-To-Talk (PTT) button on Jane's smartphone, which enters the PMA into a "ready for commands"-state (Step 1). Afterwards, Jane can utter voice commands (Step 2), which will be recorded by the Application Runtime Environment (Step 3) until the voice command is finished and recording has been stopped (Step 4). Note: The voice command is of course running until Step 4 – the duration of this action is not representable using UML sequence diagrams.

Here, the voice command will be "I want to go to the football match. What is the best transport mode for me?" Please note that Jane does not state any explicit information about which football match she wants to go to and neither when it is. This information will be acquired from other sources (cf. Section 3.3ff) and be confirmed by her.

The speech-based input is recognized by the Dialogue Interface (not depicted), which provides a data structure to the Multimodal User Interface. As it will be explained in Section 7.1, the Application Runtime Environment is providing a framework in which apps are executed; hence, it is also providing input to and getting output from an app. Hence, in the example, the Multimodal User Interface also forwards the data to the Application Runtime Environment (Step 5) both in order to identify which app needs to be started and the input data for it. In Step 6, the user is informed that the command has been understood and the according app will be started (Step 7). Finally, the necessary input data is provided to the app – here: BestTransportMode (Step 8) – and the app executes the command and processes the input data. BestTransportMode recognizes that Jane's current location will be needed and requests it from the PMA-based Sensor Abstraction through the Application Runtime Environment (Steps 9-10). The PMA-based Sensor Abstraction queries the smartphone-internal GPS device and sends the location data back to the app via the Application Runtime Environment (Steps 11-12).

While this step ends at this point, the life cycle of the involved software components including the BestTransportMode app does not end. This will be taken up in the step (Section 3.3), where the actual service execution will be discussed as well as the aspect of providing results to the end user (Section 3.4).

3.3 Service Execution

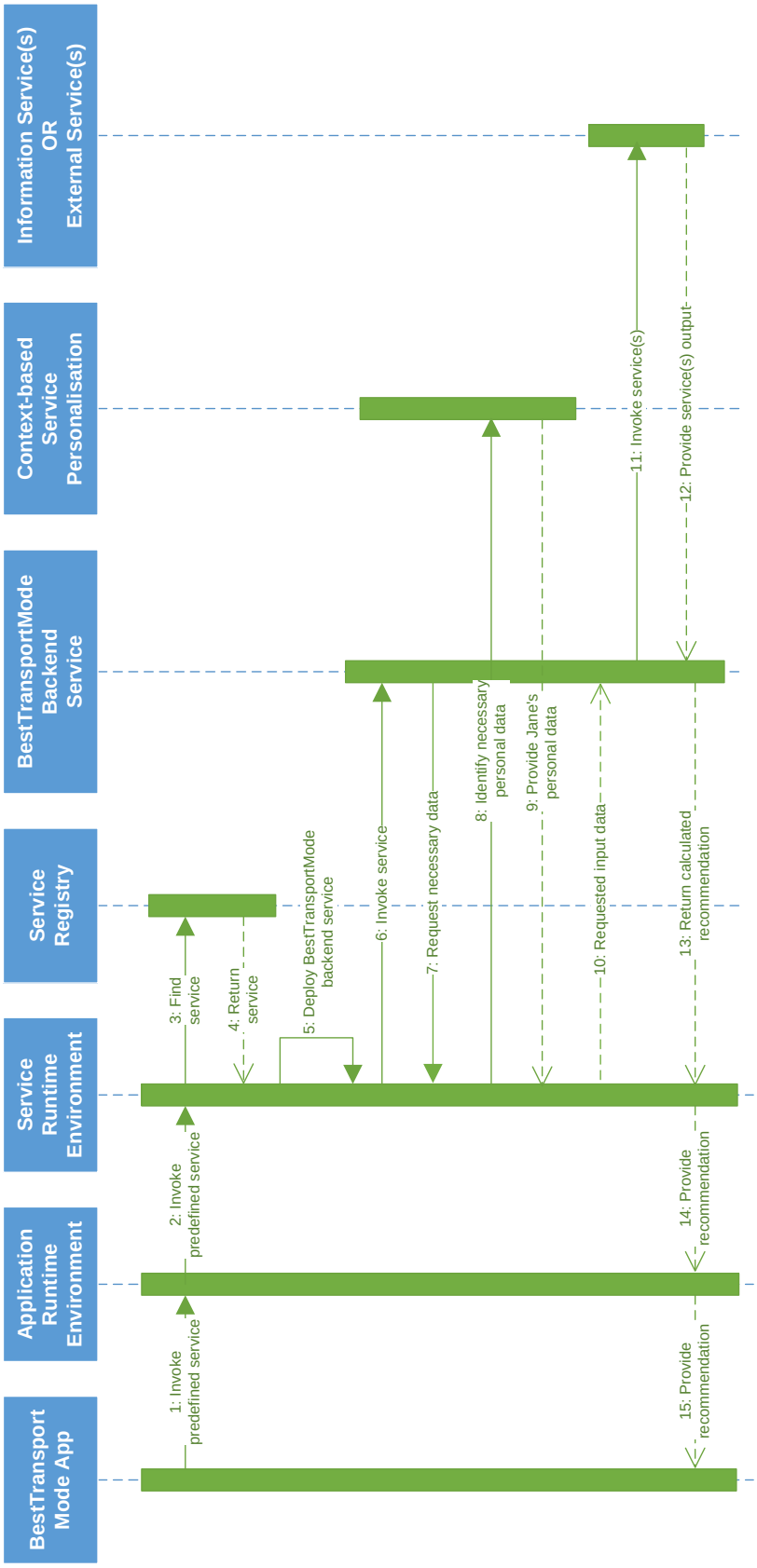

Figure 5: Service Execution

SIMPLI-CITY makes a clear distinction between the end user app, which is running in the PMA (in the Application Runtime Environment) and provides the user frontend, and services invoked by the app, which are running in the backend and provide the actual service logic and functionality. Nevertheless, an end user app could provide some additional (smartphone-) local logic, i.e., in order to get data from smartphone-internal devices as has been described in the previous section with regard to retrieve Jane's current location. In any case, the app will provide the means to interact with the user interface and all other means so that it is possible to run it within the PMA.

In the example (see Figure 5), it is assumed that so far BestTransportMode does not apply any local logic. Instead, it invokes a backend service and provides the service context data (in this case, the GPS location is the most important data item) to the backend service through the Application Runtime Environment (Steps 1 and 2). The Service Runtime Environment identifies which service it needs to invoke by looking it up in the Service Registry (Step 3). Notably, the service is tightly-coupled with the app, i.e., the service to be invoked is predefined. After the Service Registry has returned the necessary service description (Step 4), the Service Runtime Environment is able to deploy the BestTransportMode backend service (Step 5), which provides the actual backend functionalities. Afterwards, the service can be invoked (Step 6). In this case, these functionalities include looking up the itinerary for a number of transport modes, i.e., to take the train, car, bus, bike, or walk to the venue, and recommend the best transport mode to Jane.

Naturally, the BestTransportMode backend service needs at least two data items as input – the current location indicating the point of departure, and the destination. The necessary data will be requested by the backend service through the Service Runtime Environment (Step 7). Jane did not state her current location, but this information is part of the app context and is therefore already available in the PMA and has been handed over to the Service Runtime Environment, which stores this data for future usage, as part of the service invocation in Steps 1 and 2. This complex data item is called "service context".

Apart from her location, the only information available so far is that Jane wants to go to a football match, but not which match and when it will take place. In order to identify this data, personal user context data of Jane will be retrieved by invoking Context-based Service Personalisation (Step 8). This component is able to identify which sources of user data need to be queried and retrieves the information. This is a basic interaction pattern in its own right and therefore depicted separately in Section 3.5. After the user data is retrieved, it is filtered for the necessary information needed in BestTransportMode, stored as part of the service context, and handed back to the backend service (Steps 9 and 10). In the example, this would be the venue and time of the football match. It is assumed that the correct data has been found since the football match is on this particular day; however, it would be possible to include a verification step here, i.e., ask Jane if the data found is correct.

Next, the BestTransportMode backend service can make use of the input data (current location and destination) to invoke services providing the best itinerary (Step 11). These services could have been implemented in the project or constitute external third party services. To simplify matters, it is assumed that the data from the external services does not need to be converted into an appropriate input format for the BestTransportMode backend service and is already in the "correct" format.

Based on the input, the invoked services will provide information about the transport mode, routing (including distance), duration, and costs (Step 12), which the backend service analyses in order to find the best transport mode. For reasons of simplification, it is assumed here that the shortest travelling time is always indicating the best transport mode, but there is also the possibility to take into account Jane's personal preferences, which could be taken from her user profile. This would make it necessary to request these preferences from Jane in the first place, i.e., by a short questionnaire the app asks Jane once it is first installed.

Once the recommendation has been computed, it is given back as output to the Service Runtime Environment (Step 13) and provided to the end user app via the Application Runtime Environment (Steps 14 and 15), where the output will be presented to the user (see Section 3.4).

Please note: The complete sequence of actions (Steps 6-15) as presented in this subsection will be carried out in regular intervals based on Jane's location in order to get up-to-date information at all times. Hence, from a technical perspective, this sequence is arranged within a loop which is repeated until Jane arrives at her destination.

3.4 User Interaction (Output)

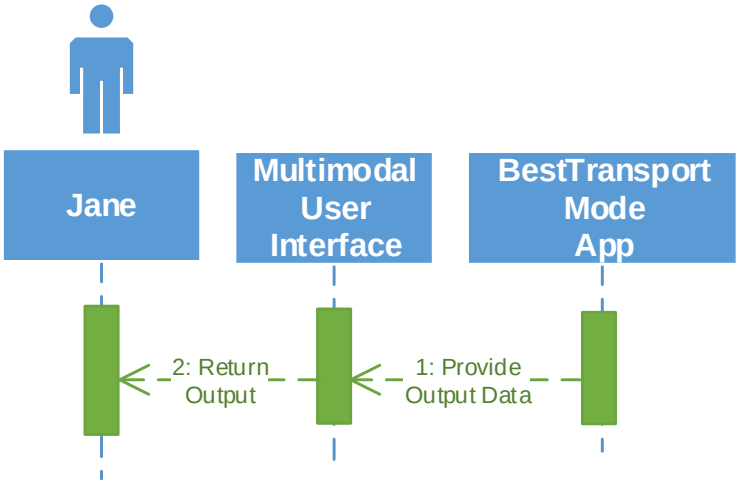

Figure 6: User Interaction – Output

The provision of output data to the end user is following the service execution presented in the previous section and can be seen as a continuation of the user interaction presented in Figure 4, since the same components are involved (Figure 6). However, interaction is started by providing some output data from an app, not by manual invocation through the end user.

As can be seen in Figure 6, Step 1 of the output is the provision of data from the BestTransportMode app to the Multimodal User Interface – this is a direct continuation of Step 15 from the previous section. Step 2 shows the actual provision of output to the end user. Notably, the Application Runtime Environment is not involved in providing output data to the user.

3.5 Context Data Integration

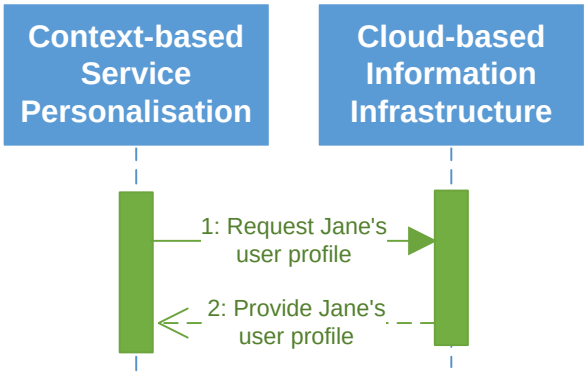

Figure 7: Context Data Integration

(User-)Context-based service personalisation in SIMPLI-CITY is always conducted through the according software component (see Section 6.3), i.e., the Context-based Service Personalisation component. As described in Section 6.3, context data can be used for different purposes, i.e., as non-explicitly stated input data for services, control statements for services, or in order to select a particular service instance/variant which provides data relevant in a particular context. In any case, the context data needs to be retrieved. For this, there are two possibilities: Either, the context data is handed over by the Service Runtime Environment, as data regarding the invocation (e.g., user location) is originally provided by the PMA, or context data has to be retrieved from a data source or data store. In many cases, a combination of both approaches to gather user data will be conducted, because, e.g., the input data coming from the PMA will be used in order to identify a user profile stored in the Cloud-based Information Infrastructure. However, context data could of course be also used locally, i.e., within an app, without invoking the Context-based Service Personalisation component (task T5.2) and sending data to it. In this case, the functionalities of this component will not be available.

Figure 7 is directly connected to Step 8 within Figure 5 (see Section 3.3). As can be seen in Figure 7, the interaction between the Context-based Service Personalisation component and some data source (here: Cloud-based Information Infrastructure) is straight-forward – the former component simply requests the data needed from a particular source, which could be the Cloud-based Information Infrastructure, but also some sensor, open data repository, or the Data Processing component. Once the user profile has been provided, it will be analysed and only relevant data fields will be handed back to the requester.

In the example, Jane's user profile is identified based on a unique identifier and requested from the Cloud-based Information Infrastructure. Further user context data could be gathered from other sources, e.g., a web-based calendar, address book, etc. However, in this example, it is assumed that there is calendar data directly provided through the online profile. Furthermore, Jane's transport modes preferences are also part of the online profile. The next step would be to continue with Step 10 from Section 3.3.

3.6 Data Acquisition

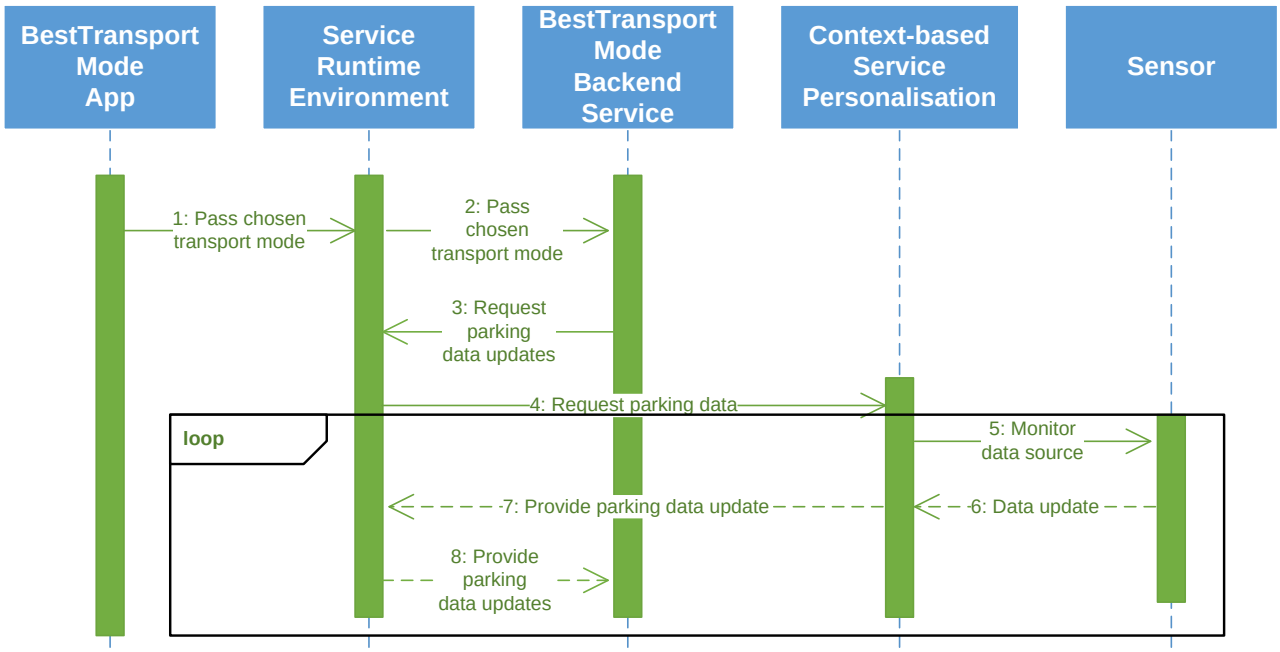

Figure 8: Data Acquisition

In the previous section, the focus was on user profile data, as it is part of the underlying story. The acquisition of data from other sources is comparable but extends the so far regarded scenario. General data acquisition will be explained in the following based on Figure 8.

In the story, the recommendation of the BestTransportMode has been to take the car, so Jane enters her car at the right time and starts to drive to the stadium. During her journey, the BestTransportMode app regularly interacts with its backend service as was presented in Section 3.3 in order to request if there has been an update in the parking situation. It is assumed that the necessary BestTransportMode backend service is still running after the recommendation of a particular transport mode has been sent to the app as seen in Step 15 of Section 3.3 and that this service still possesses all relevant data about the different transport modes, most importantly Jane's starting point and location, destination, and the routing.

In the scenario, the BestTransportMode app will provide its backend service with the information that Jane has decided to make use of her car (Step 1 and Step 2). Notably, the usually involved Application Runtime Environment has been omitted to keep the figure well-arranged.

Please note that the identification of the service from the Service Registry as shown in Section 3.3 has been omitted from Figure 8 for reasons of simplification. The backend service will then derive from this information that it needs to provide updates about the parking situation at Jane's destination and request according data from parking spaces. This step has been skipped in Section 3.3 in order to not extend the number of involved components/tasks too much.

Luckily, the city Jane is heading to provides information about all public parking spaces using sensor technology. In order to make use of such data, the backend service requests generic parking information data from the Service Runtime Environment (Step 3). Notably,

the backend service does not provide information about what exact parking space needs to be monitored to the Service Runtime Environment; this information will be derived by the Context-based Service Personalisation based on the destination information (Step 4). Afterwards, this component subscribes itself to relevant events from sensors (Step 5). In SIMPLI-CITY, a publish/subscribe model is applied for this – the according publish/subscribe subcomponent is part of the Context-based Service Personalisation (see Section 6.3.2). When there is a relevant sensor event, i.e., the parking situation changes to some extent, this information is provided to the Service Personalisation (Step 6), which provides it to the Service Runtime Environment (Step 7) and finally the data is handed over to the BestTransportMode backend service (Step 8). Notably, Steps 5 and 6 are therefore part of a loop which will be carried out until a service is unsubscribed from a particular context data source. Steps 7 and 8 are also part of this loop, but only fired if the Context-based Service Personalisation identifies a relevant data change.

It is assumed that the data format is already compatible with the SIMPLI-CITY Data Model (see Section 8.2). If this would not be the case, data transformation needs to be applied. Furthermore, it is assumed that live data can be retrieved directly from the sources (through the publish/subscribe functionality).

This part of the story ends here, but of course the information needs to be taken into account by the backend service and it could be necessary to calculate a new routing. For this, Steps 12-15 from Section 3.3 would be carried out. Afterwards, Data Acquisition as presented here would be carried out again until Jane arrives at her destination.

In this example, the focus is on the parking situation, but a similar sequence of actions could be followed in order to integrate data from traffic sensors, e.g., in order to identify congestions and adapt the routing information based on it. Last but not least, some cities provide parking space data through open data sources. Again, the inclusion of such a data source would be similar to the approach presented here. A more detailed technical discussion of this will be part of Section 5.3

3.7 Data Prefetching

In the previous sections, it has been shown how data can be integrated ad hoc into apps, i.e., the data is requested and provided at the point of time it is necessary. In addition, SIMPLI-CITY will also allow to request data a priori, i.e., before it is actually needed. For this, it is necessary to predict which data will be needed at what point of time. In the following, it will be shown how this can be achieved for media streams, but the presented approach could also be applied in other situations.

It is assumed that Jane plans a trip by car. She indicates that she wants to make use of her favourite Internet music streaming site, Last.fm, during the journey. For reasons of simplification, the steps where she plans the trip, chooses a transport mode, and also specifies that she wants to listen to Last.fm during her travel, will be omitted. Instead, it is assumed that all this information has already been forwarded to the Application Runtime Environment and that the according backend service has been identified (see Section 3.3). The following steps are actually conducted *before* Jane starts her journey.

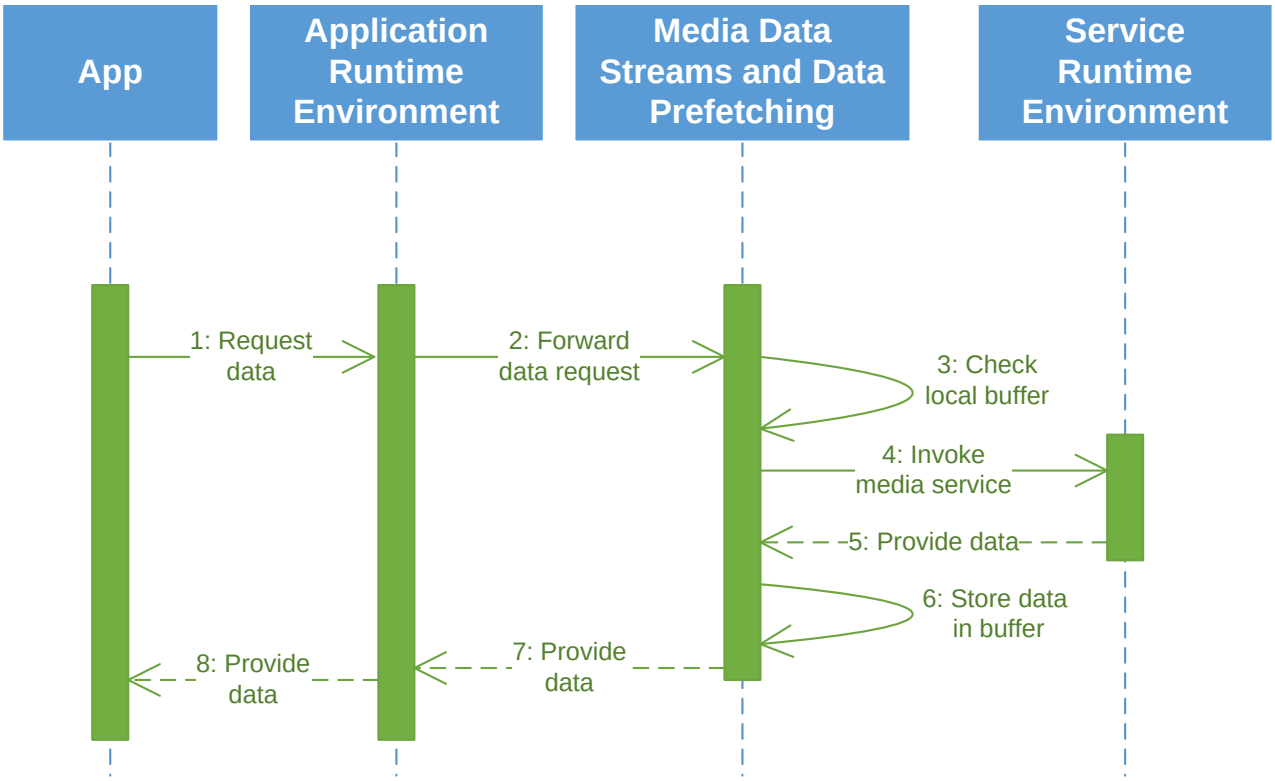

Figure 9: Data Prefetching

As stated above, all necessary data is already available in the Application Runtime Environment (Step 1). Hence, in Figure 9, this component is the primary actor. The Application Runtime Environment invokes the Media Data Streams and Data Prefetching component (Step 2) and provides it with the necessary information, i.e., which music streaming provider Jane wants to make use of and her user credentials for this provider. Afterwards, in Step 3, the Media Data Streams and Data Prefetching Logic component checks if a stream has already been prefetched in a local buffer (not depicted), which is located on the client (here: mobile device). If this is not the case, an according media service is invoked through the Service Runtime Environment (Step 4). The actual media service is not depicted here.

After the media stream has been provided to the Media Data Streams and Data Prefetching component (Step 5), it is stored in the local buffer (Step 6) and can afterwards be provided to the app via the Application Runtime Environment (Steps 7 and 8).

4 Foundation for Functional Specification

In Sections 5-8, different aspects of the functional specification for all SIMPLI-CITY software components are defined. As an introduction, the single aspects are explained in the following subsections. In Section 4.1, the actual aspects of the functional specification are covered, while Section 4.2 explains the rationale behind the parameters to take into account for the upcoming Technical Specification deliverable D3.2.2.

4.1 Functional Specification

Each of the following aspects of the functional specification will be discussed in a specific subsection for the individual software components in Sections 5-8.

4.1.1 Overall Functional Specification

The goal of this subsection is to name the most important functionalities that will be provided by the individual component. This short summarization of the functionalities is based on the component's definition from the Global Architecture deliverable D3.2.1.

4.1.2 Subcomponents

While the Global Architecture deliverable has already provided information about the subcomponents of the major SIMPLI-CITY software components, these definitions were quite high-level. As the ongoing discussion has led to further insights and more details have been developed, this subsection aims at a brief clarification of the needed subcomponents. A figure which makes use of the FMC modelling notation[2] is also provided in order to show the interaction between the subcomponents and external components. Some subcomponents may be marked as "optional", if they are needed to fulfil a requirement with a low priority (see next subsection).

It should be noted that the choice of subcomponents is preliminary and does not restrict the technical specification. If during the technical specification software frameworks are evaluated which make use of a distinct partitioning of functionalities, this will not restrict their selection, as long as a software framework is able to provide the needed functionalities and interfaces for other components.

4.1.3 Related Requirements

To make the Functional Specification deliverable self-contained, this subsection lists all requirements that have been assigned to a particular component within the Requirements Analysis Report (deliverable D2.3). It describes which subcomponents are needed to meet the requirement. Notably, both primary and secondary task assignment are covered.

Furthermore, the requirements are prioritized using the well-known *MoSCoW* method into:

- Must Have Requirements: Requirements that need to be met in any case, as they provide a functionality that is essential for the overall project. These requirements need to be implemented in any way.
- Should Have Requirements: Important requirements which are nevertheless not as crucial for the overall project. Hence, they may be implemented if resources allow.

[2] http://www.fmc-modeling.org/

- Could Have Requirements: Not so important requirements, which nevertheless provide an interesting functionality. If resources allow, they may be implemented, but the chances are rather small.
- Will Not Have For Now Requirements: Non-relevant requirements, which will not be covered in the project.

Within a MoSCoW category, requirements are sorted according to their number as defined in the Requirements Analysis Report. The sorting does not indicate any prioritisation of the single requirements within one MoSCoW category.

4.1.4 Interactions with other Components

The goal of this subsection is to show the interactions between the component at hand and other components in a more specific way than has been done in the Global Architecture Definition (D3.1). For this, UML sequence diagrams are used, which show the interaction between the subcomponents of the currently discussed component, and external components. Importantly, the interaction with external components is not done on the level of the external components' subcomponents – the external component is a "blackbox".

As an example, if describing the interaction between the Service Runtime Environment and the Service Registry (as covered in Section 6.1.4.1), the Service Runtime Environment's subcomponents will be shown, but the Service Registry's ones will not.

4.1.5 User Interface

If applicable, this subsection shows a mockup of the Graphical User Interface for the discussed component. However, not all components feature a GUI.

4.1.6 Conceptual Data Model

This subsection is dedicated to the discussion of the needed data models in a generic, technology-independent way. Therefore, it contains information about, e.g.:

- Which data needs to be stored in which format?
- Which data is needed to describe stored data, i.e., a data metamodel?
- Which data format is needed for messaging between components?

4.2 Parameters to Take into Account for Technical Specification

At the end of each component in Sections 5-7, a summary table is given which defines a set of criteria which are perceived to be important for that specific component (see the example for the Service Registry in Table 1). The generic criteria are the same across all components, but their importance for each component will differ. Other criteria are specific to the components – these are usually rather functional criteria. The importance measurement is on a scale from "--" to "++".

This table is then reused in the Technical Specification (D3.2.2) in order to assess and analyse the different technical options against the criteria from the table. This approach will provide conclusions for the actual selection of a technology.

Specific criteria are defined within the single component descriptions. The generic criteria will be considered as follows:

- Up-to-Datedness: In general, it is the aim of the consortium to apply and extend cutting-edge technologies which already offer the latest state-of-the-art. This will allow the consortium to create more innovative solutions which will attract more attention during the project dissemination and exploitation
- Stability: The technologies to be chosen should be stable and mature, i.e., preferably, not be in beta mode. This is especially crucial if a product is chosen which is not regularly updated.
- Extensibility & Open Source/Standards: 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.
- 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.
- 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.

There might be a trade-off between the different criteria. In this case, this trade-off needs to be assessed during the technical specification.

Table 1: Example Criteria for the Service Registry

Parameter	Importance (--, -, -+, +, ++)
Generic Criteria	
Up-to-Datedness	-+
Stability	+
Extensibility & Open Source/Standards	+
Familiarity	+
Performance	++
Interoperability	+
Specific Criteria (max. 10)	
GUI	-
Service Level Agreement (SLA) Support	++
Registry AND Repository	++
Various Possibilities Regarding File Storage	-+
Various Possibilities Regarding Database	-+
Role Management	+
Version Control	+
Extensible Data Model	++

5 Functional Specification: Data Integration

5.1 Data Processing

5.1.1 Overall Functional Specification

The Data Processing component functionality comprises subcomponents which combine and process incoming data from different sources. The data coming from these different sources will have been transformed into a *Unified Data Model*. This model allows contextualization and reasoning functionalities to be exercised across the datasets, i.e., vehicle based sensor data available to the PMA, external sensor data sources, user-profile and personal data, open data and stored history data. Furthermore, the Unified Data Model eases the usage of SIMPLI-CITY-enabled data sources for service developers.

5.1.2 Subcomponents

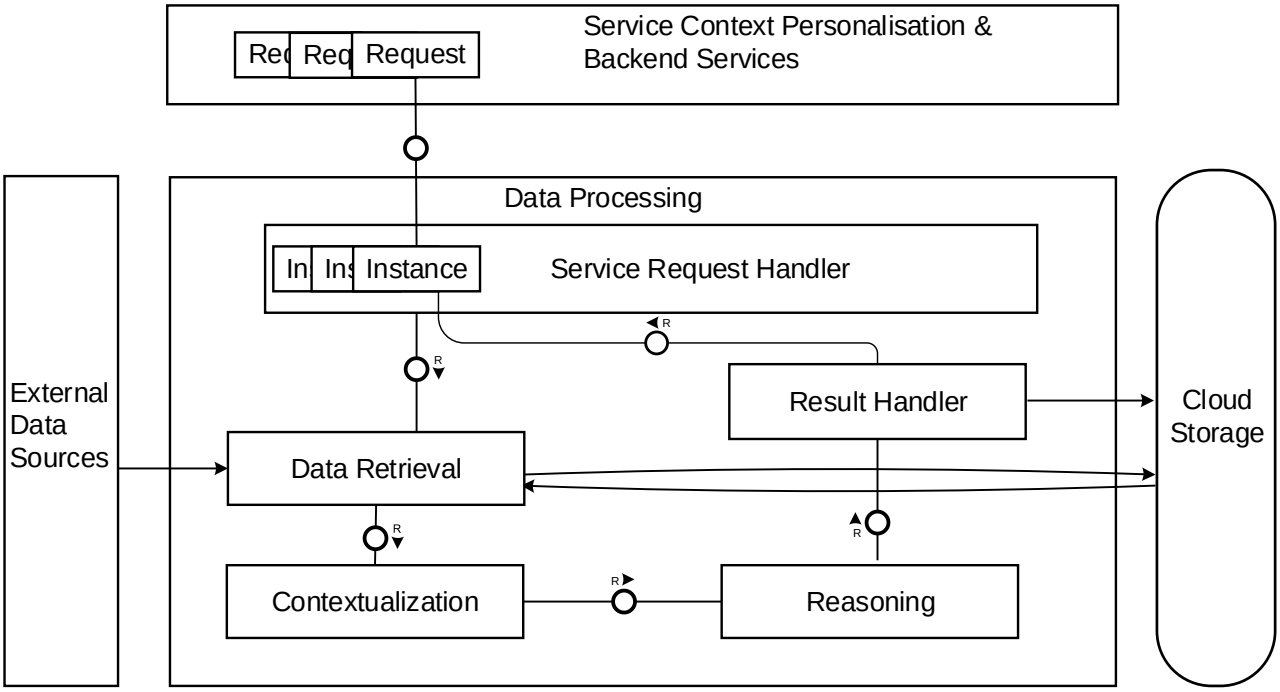

Figure 10: Data Processing Subcomponents and Interactions

The Data Processing Component comprises the following proposed functional subcomponents:

- Data Retrieval: Provides an interface that allows to state data requests to get open (government) data source data and historical semantic data and other data required by the request from other components (e.g., the Sensor Abstraction and Interoperability Interface, the user profile in the cloud and user personal data).
- Contextualization: Of static and stream data to form a combined semantic structure.
- Reasoning: To perform prediction and diagnosis on the combined semantic structure.

- Result Handler: To push results to the cloud for retrieval by services (through according service requests) and notify the Service Request Handler of completion status.
- Service Request Handler: To receive requests and notify status and outcomes to the requesting service. This will create distinct processing instances for different service requests with role-based access to relevant datasets and calling privacy related routines such as data encryption where required. Service requests may be issued by internal backend services running in the Service Runtime Environment (more precisely: the Service Host).

All externally pulled datasets are processed in the Contextualization subcomponent vocabulary and ontology. This produces a combined semantic structure tree for the datasets and is analysed in the Reasoning component which does prediction and diagnosis. This includes such operations as filtering, correlation, merging, fusion, aggregation, summarisation and splitting of data. The tree, the outcome and relevant data of the Reasoning component is available to be stored in the Cloud-based Information Infrastructure for return to the user via the service.

5.1.3 Related Requirements

Table 2: Requirements Related to the Cloud-based Information Infrastructure

Requirement	Handled by Subcomponent	Comment
Must Have Requirements		
U31: Reaction to time of the day (P1) U34: Reaction to history of usage (P1) U35: Reaction to traffic information, like traffic jams, train schedules, road works, accidents, and strikes (P1)	Contextualization	Supports basic integration of time-based data, fulfilling the SIMPLI-CITY spatio-temporal contextualisation requirements.
U34: Reaction to history of usage (P1)	Contextualization	Contextualization history of end-user information will be retrieved and used in processing user requests.
U35: Reaction to traffic information, like traffic jams, train schedules, road works, accidents, and strikes (P1)	Data Retrieval	Data related to U35 will be retrieved from relevant open-data sources and taken into account in contextualization and reasoning.
U59: User centric data services (P1)	Data Retrieval Contextualization	User personal data sources and profile data will be combined with the other requested data sources and used for contextualization.

Requirement	Handled by Subcomponent	Comment
U80: Profile in the cloud (P4)	Data Retrieval	User profile data will be retrieved from the Cloud-based Information Infrastructure and used for contextualization.
U86: Transparency (P1) U87: Confidentiality. Does not give away data to third parties (P1) U88: Data encryption (P2) U89: Certification. Only certified apps are allowed to access users data (P2) U92: Secure access to system (P3)	Service Request Handler	Fulfils data privacy-protection, using separate request instances, RBAC, encryption where required, checking user profile information and using secure transmissions and encrvption where specified in the service request.
U112: Support of Open Data (P1)	Data Retrieval	Open Data sources will be accessible via the Data Retrieval subcomponent which will fetch the data related to the particular data sources for each service request. The particular request may use this data directly or also request contextualization and reasoning over the dataset(s).
U114: Configuration of the frequency of update of the data from data sources (P1)	Data Retrieval	Allows a service to set the frequency of update for data sources directly queried by the Data Processing component relating to Open Data.
U115: Transformation support (P1)	Data Retrieval	Transforms open data and user personal/profile date to the common data representation (SIMPLI-CITY Unified Data Model).
U116: Unified Data Model (P1)	Contextualization	The Unified Data Model is used for two purposes: First as a foundation for the contextualisation and reasoning capabilities of the Data Processing component. Second to ease the integration of (new) data sources for service developers based on a well-known data model.
U117: Data filtering (P1) U118: Data correlation (P1)	Reasoning	The Data Processing subcomponent Reasoning will provide the functionality to do data filtering and correlation.

Requirement	Handled by Subcomponent	Comment
U119: Data Summarization (P1)	Data Retrieval	The Data Processing subcomponent Data Retrieval will provide the functionality to do Data Summarization.
U120: Handle data streams (P3)	Data Retrieval Contextualization	The data Retrieval and contextualisation subcomponents will make use of stream handling software to process stream data sources.
U189: Unified interface for accessing sensors	Unified Data Model	The Sensor Abstraction Interface will handle access to the vehicle sensors and the smart device sensors and provide data in a format which feeds into and conforms with the Unified Data Model.
U190: Unified interface for accessing user centric data (P1)	Data Retrieval Unified Data Model.	Will use Sensor Abstraction Interface which in turn will use Unified Interface for access to user centric data.
U202: Diagnosis of abnormal traffic condition in real-time (P1) U203: Prediction of abnormal traffic condition (P3) U204: Support for querying diagnosis historic (P1) U205:Support for querying impact factor on traffic condition (P3) U206:Optimization of financial resources (P1) U207: Support suggestions for road/trip optimization if conditions change (P1)	Data Retrieval Contextualization Reasoning	Open data-sources relating to traffic conditions are processed along with stored contextualized historical data and these are then used to contextualize and reason over.
U208: Possibility to continue a previously started trip planning on the same device or different device (P4)	Data Retrieval Contextualization	Stored User profile and history data can be accessed in the Data Retrieval sub-component and used in contextualization.

Requirement	Handled by Subcomponent	Comment
Should Have Requirements		
U50: Prefetching of media data & Offline access (P1) U52: Offline access of data used within apps (P2) U113: Handling of multimedia data (P1)	Data Retrieval	The Data Retrieval subcomponent will support data-prefetching accordingly as any service requests from T4.5 are forwarded. Prefetching is handled as described by T4.5 and should support the streaming of data from user centric (and other) data sources.
U85: Interaction with car sensors (P1) U107: Access to sensors of the vehicle (P1) U108: Access to smart device sensors(P1) U109: Access to remote sensors (P1) U110: Remote control of car components, e.g., air conditioning, heating, battery charge timing (P3)	Unified Data Model	The Sensor Abstraction Interface will handle access to the vehicle sensors and the smart device sensors. T4.1 provides a unified model for data description and access.

5.1.4 Interaction with other Components

The Data Processing component requires interaction with the following components as detailed in the following subsections.

5.1.4.1 Interaction with the External Data Sources

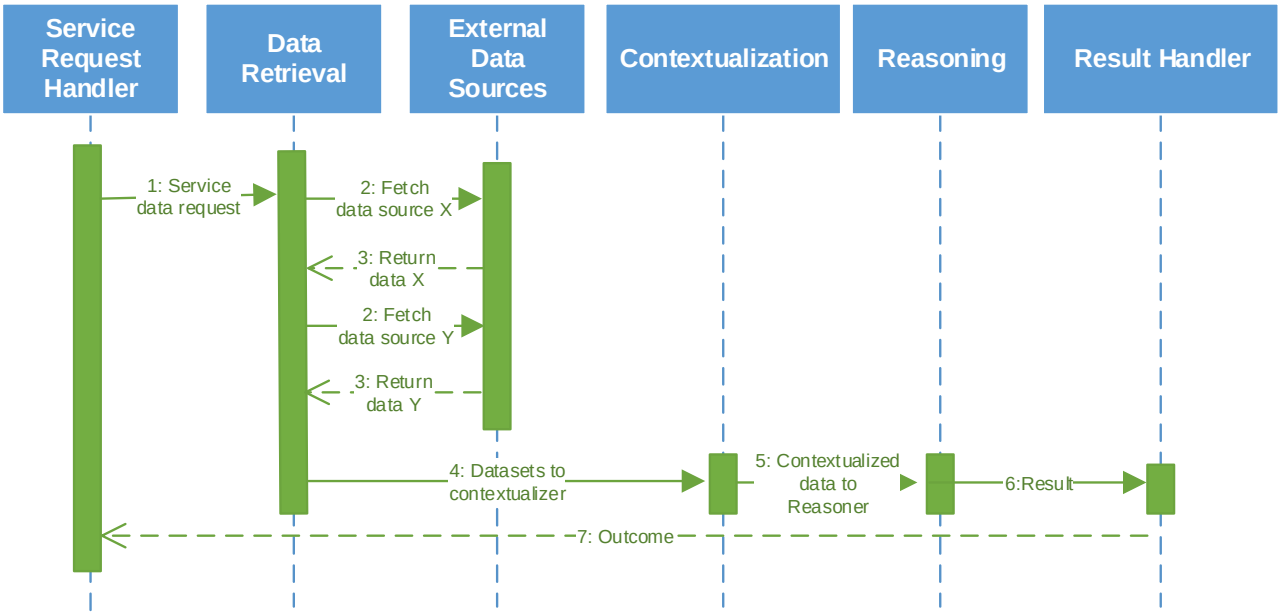

Figure 11: Interaction of Data Processing and Data Sources

Figure 11 shows the interaction between the Data Processing component and SIMPLI-CITY-enabled, external data sources, i.e., accessible through the Sensor Abstraction and Interoperability Interface, Open Data sources and user personal data sources. Essentially, the Data processing component requests to provide data for a particular data source.

Data sources will be queried to return the data relevant to the current service request using the relevant APIs provided by the components. It is proposed from a performance perspective that requests for several data sources could be done in parallel (or asynchronously) to avoid long sequential delays in retrieving data from different sources. Afterwards, data can be contextualised, reasoning is conducted, and the Result Handler returns the outcome of these actions to the requesting component via the Service Request Handler. The requesting component could be either the Context-based Service Personalisation (see Section 6.3) or the Service Runtime Environment (Section 6.1).

5.1.4.2 Interaction with the Cloud-based Information Infrastructure

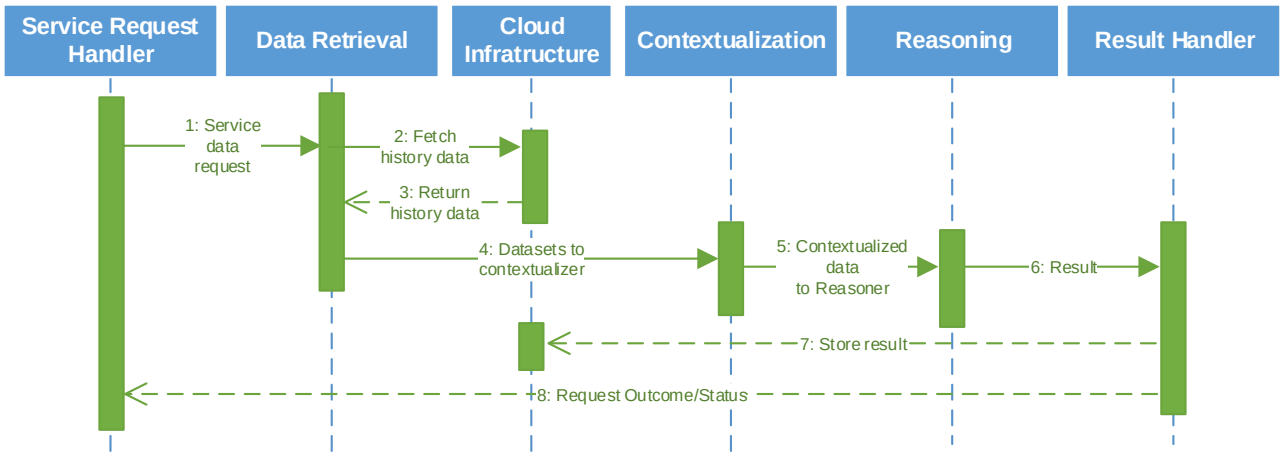

Figure 12: Interaction of Data Processing and Cloud Infrastructure

The figure below shows the interaction between the Data Processing Component and the Cloud-based Information Infrastructure. In the first case (Steps 1-3), it is used to fetch semantic history data and return updated semantic data from the Contextualization subcomponent. It also stores the result of any Reasoning subcomponent functionality exercised (Step 7).

The Data Retrieval subcomponent will query the Cloud-based Information Infrastructure to get semantic history information related to the current request. The returned data will be passed to the Contextualization and then reasoned over in the Reasoning subcomponent execution along with data from the other data sources for this request. Results will be updated to the Cloud-based Information Infrastructure as will updates to the stored semantic history information.

The Cloud-based Information Infrastructure APIs will be used to access semantic history information and to store updated semantic information, service prefetch data and other service request results.

5.1.4.3 Interaction with the Service Runtime Environment and Context-based Service Personalisation

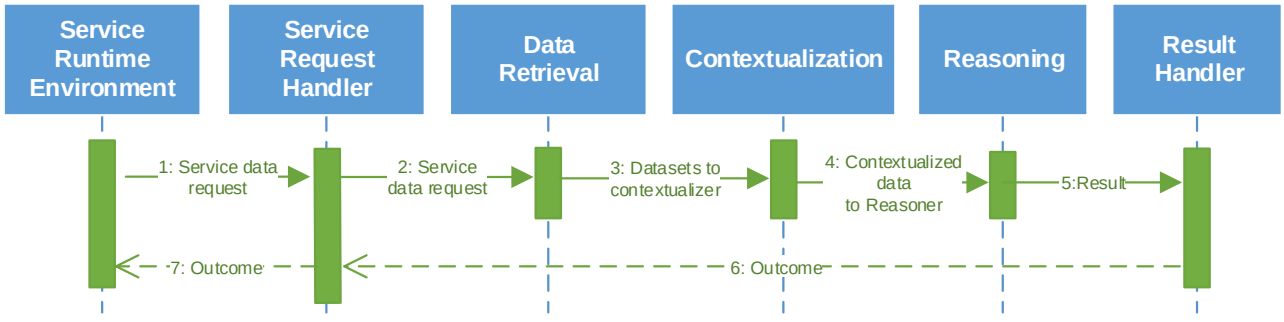

Figure 13: Interaction of Data Processing and Service Runtime Environment (and Context-based Service Personalisation)

Figure 13 shows how the Data Processing component interacts with the Service Runtime Environment, i.e., backend services running on the Service Host (see Section 6.1). Backend services initiate a request with a fetch data source command which is followed by any number of relevant data source fetch requests relevant to that service. The Service

Runtime Environment could also provide relevant anonymized user profile info and personal data to the data processing component for this particular request.

This component will interact with the Service Runtime Environment to receive user and service requests and to notify the Service Runtime Environment of the status of requests and results. The Data Processing component will provide APIs that can be called by the Service Runtime Environment. The Data Processing component will also call Service Runtime Environment APIs to return status and result information to the service.

The details of the Data Retrieval subcomponent are shown in Section 5.1.4.1. This facilitates the data processing to asynchronously handling the data retrieval from different data sources.

Notably, the interaction for the Context-based Service Personalisation component is exactly the same as for the Service Runtime Environment, but instead of backend services, Context Sensors will request data and contextualisation outcomes from the Data Processing component. The interactions from the perspective of these components are described in Section 6.1.4 for the Service Runtime Environment and Section 6.3.4 for the Context-based Service Personalisation.

5.1.5 User Interface

The only interfaces to the Data Processing component which can be called are the Data Processing APIs which will be defined in detail in the technical specification (deliverable D3.2.2). There will be no separate GUI for the Data Processing component.

5.1.6 Conceptual Data Model

The Data Processing component will make use of a Unified Data Model in order to allow Contextualization and Reasoning over heterogeneous datasets in the form of structured semantic data. The Unified Data Model has the implication that each heterogeneous data sources undergoes a transformation of its data into a common format whether the data is coming from sensor, personal, user profile or open data. For new data sources, this could mean that a service developer who wants to make use of the data source has to map inputs and outputs during the registration of a new data service (see Section 8.2).

Stream handling software will be used to combine static and stream data from different sources (sensor, open, personal, user profile) using vocabulary and ontologies to produce a common semantic structured tree which can then be reasoned over.

5.1.7 Parameters to Take into Account for Technical Specification

Table 3: Criteria for Technical Specification

Parameter	Importance (--, -, -+, +, ++)
Generic Criteria	
Up-to-Datedness	-+
Stability	++
Extensibility & Open Source/Standards	+
Familiarity	++
Performance	++
Interoperability	++
Specific Criteria	
Data stream Handling Capability	++
Static and Multi-stream Data Combining Capability	++
Data Transformation Capability	++
Data Contextualization Capability	++
Role Based Access	++
Scalability	+
Asynchronous Request/Data Handling	++
Ontology/Vocabulary Handling	++
Semantic Structure Generation	++
Standard Way to Represent Heterogeneous Data	++
Robust Way to Contextualise/Filter Data	+
Querying Mechanism over Combined Structured Data	++

5.2 Cloud-based Information Infrastructure

5.2.1 Overall Functional Specification

The Cloud-based Information Infrastructure will provide persistence functionality for SIMPLI-CITY. It will act as a service which is designed for managing different types of data in a persistent, scalable and efficient storage. Hence, access, storage, and retrieval of mobility-related data are performed through the Cloud-based Information Infrastructure. The component will act as a data storage solution for information and serves as a source for static data that can be used by apps and services.

The cloud-based storage will be fed with information from apps (e.g., to store data from users), from services (e.g., to store settings or data for backend services) or from external data sources such as sensors or user centric data. It will be accessed by services, other components or apps but there will be no direct access from apps to the data storage: All app access will be performed via the Service Runtime Environment.

The component will allow services, apps and components to create isolated data storage spaces which will not overlap with those of other components, services or apps. Those isolated storage spaces are referred to as “Buckets”, which is a concept originating from the Amazon S3 storage solution and has proved to be robust in many Cloud-based storage solutions. The bucket concept allows the usage of different storage backends in order to support different types of data storage. The project will provide a NoSQL storage for structured JSON-based data, a semantic storage and a storage for binary data.

5.2.2 Subcomponents

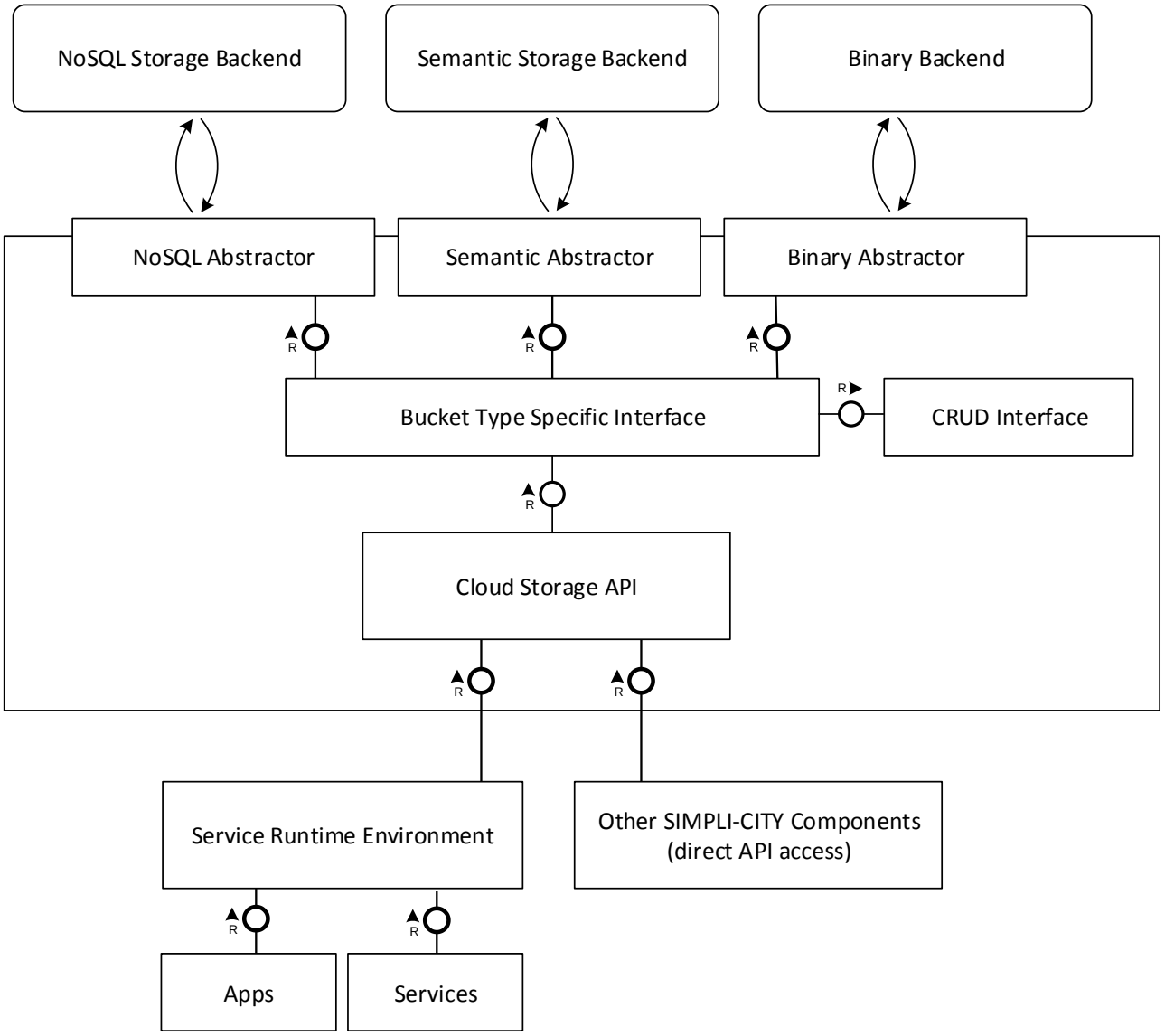

Figure 14: Cloud-based Information Infrastructure Subcomponents and Interactions

To achieve the functionalities described in the previous 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, Binary and Semantic 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: Will provide standard functionality for Create, Read, Update and Delete (CRUD), which will be available for all storage types. The CRUD Interface is invocated by the Bucket Type Specific Interface, which is the main connection to the Cloud Storage API used by developers.

- Bucket Type Specific Interface: Will provide more advanced query facilities than the CRUD Interface, such as SPARQL.
- Cloud Storage API: Will provide a holistic interface to external components like the Service Runtime Environment.
- Local Key Storage (Not shown in diagram): While not a subcomponent of the Cloud-based Information Infrastructure, the Local Key Storage is nevertheless necessary to fulfil the requirements towards data storage within SIMPLI-CITY. The Local Key Storage provides a 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 storage component is used by other SIMPLI-CITY components:

- All other SIMPLI-CITY components: To store component-specific data and settings. Importantly, the Service Runtime Environment will be used by apps and services to forward requests, i.e., apps and services will interact with the Cloud-based Information Infrastructure through other components.

5.2.3 Related Requirements

Table 4: Requirements Related to the Cloud-based Information Infrastructure

Requirement	Handled by Subcomponent	Comment
Must Have Requirements		
U80: Profile in the cloud (P1)	Cloud Storage API CRUD Interface NoSQL Abstractor NoSQL Backend	An essential functionality of the cloud-based storage infrastructure is to store structured information. For this purpose, the NoSQL storage facility of the Cloud-based Information Infrastructure will be provided by the storage via an API. This will also support the storage of the user profile.
U81: Storage of data in the cloud (P4) U122: Storage of data in the cloud: binary, semantic and structured (P1)	Cloud Storage API CRUD Interface All Abstractors and Backends	The storage of different types of data has to be provided. For this purpose, the bucket type concept has been introduced.
Should Have Requirements		
U121: Local storage of data (P1)	Local Key Storage	This will be an extra component of the Cloud-based Information Infrastructure sitting at the PMA side with a simplistic key-value store for small data via a minimalistic API for mobile devices.

Requirement	Handled by Subcomponent	Comment
U123: Scalability of the cloud infrastructure (P1)	Storage Backends	Storage Backends will be chosen in a way that auto-sharding may be applied. For example, MongoDB or Amazon S3 may be used for scalable backend systems. The Cloud Storage API may be replicated via a cluster in case of having to cover many parallel requests.

5.2.4 Interaction with other Components

5.2.4.1 Interaction with External SIMPLI-CITY Components

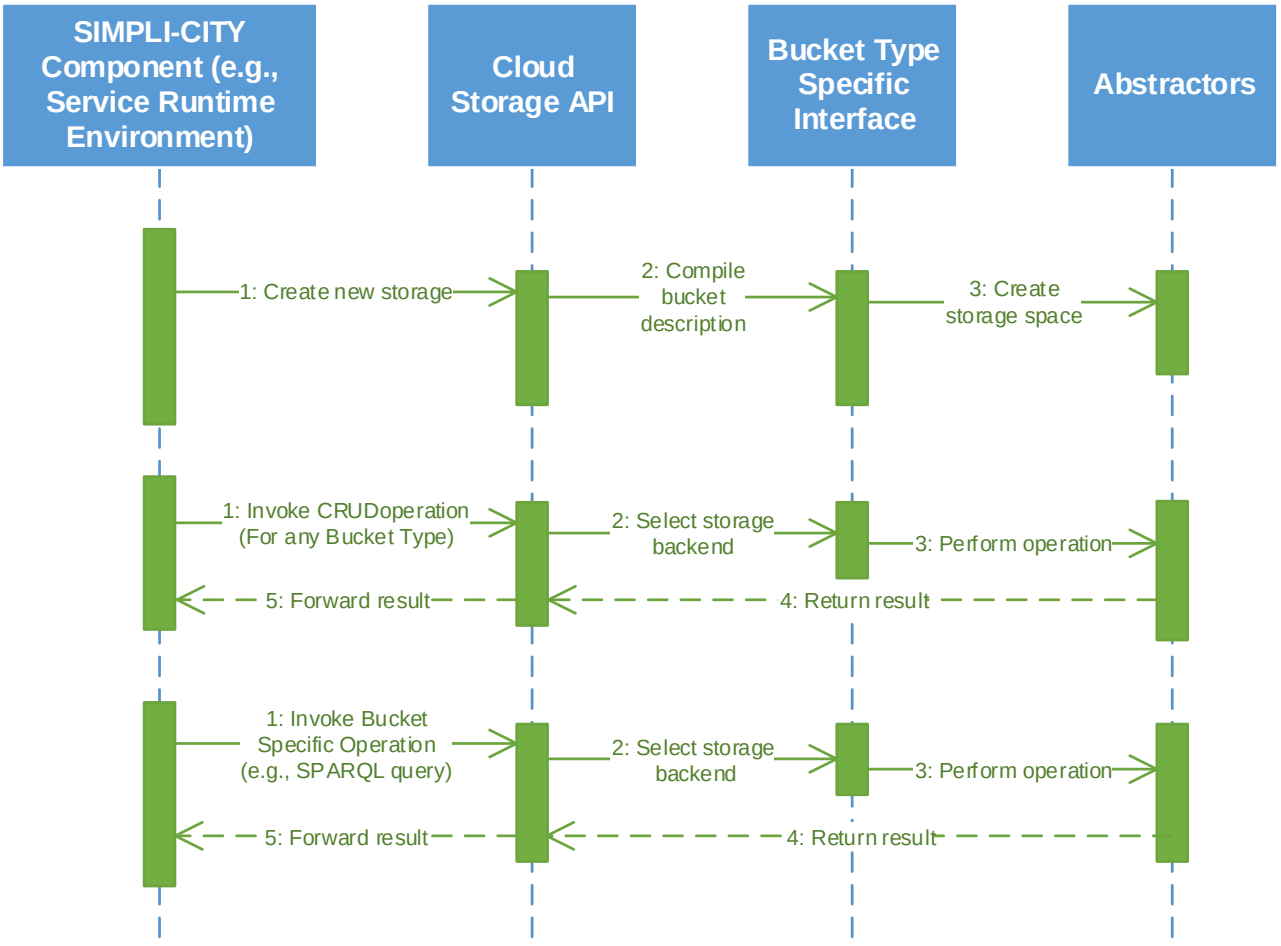

Figure 15: Invoking the Cloud Storage Component

The figure above shows the different possible interactions between the SIMPLI-CITY components and the cloud storage. This covers the possibility to create new, isolated storage spaces and to perform operations on them. Operations may use the generic CRUD functionality which is available for all bucket types and encapsulated by the Bucket Type Specific Interface component. CRUD functionality will be available for all Storage Backends. The Bucket Type Specific Interface will also handle all operations that are not covered by the standard CRUD Interface such as a SPARQL query, which will only be

supported by the semantic storage backend. These queries need to be explicitly written and executed via the Bucket Type Specific Interface, since there is no uniform approach for standardised queries. Hence the Cloud Storage API is able to handle both standardised CRUD-Operations and manually written Queries (e.g., via SPARQL). Data that needs to be stored will automatically be distinguished by the Abstractors and stored in the adequate Storage Backend. The Storage Backends are not depicted in Figure 15 since the only interactions with the Abstractors are to read and write data into the Storage Backend.

5.2.5 User Interface

This component does not have a user interface for end users.

5.2.6 Conceptual Data Model

The component will provide an open interface by making all its functionality available as service calls. This means that components will not be bound to a specific operating system or development language. Consequently, data that is received and delivered will also be wrapped in an open format based on, e.g., JSON or a combination of JSON and RDF.

The data storage provides different data bucket types in order to support data storage of different kind of data: e.g., Binary, NoSQL, and Semantics. The concept of data buckets is used since each of those data types is best managed by different implementations and concepts. For example, binary storage may be realized by a distributed file system and may even make use of existing cloud storage services such as Amazon S3. In contrast to this, semantic data usually has requirements such as advanced query functionality (e.g., applying SPARQL), which is usually not needed for binary data. As such, another data storage implementation may be chosen.

In order to bring together those different requirements, the cloud storage will provide one holistic wrapper for different cloud storage types. This wrapper will provide basic CRUD functionalities for all data types as well as advanced interfaces (e.g., SPARQL) for specific data bucket types. Each component may create multiple data buckets and specify their type.

The storage wrapper will hide the physical implementation of each data store. Different technologies may be used for this and will be analyzed and selected during the functional and technical specification of SIMPLI-CITY. The different technologies should, however, be chosen in a flexible way, e.g., by using jClouds to access Amazon S3 storage instead of directly accessing it.

5.2.7 Parameters to Take into Account for Technical Specification

Table 5: Criteria for Technical Specification

Parameter	Importance (--, -, -+, +, ++)
Generic Criteria	
Up-to-Datedness	++
Stability	+
Extensibility & Open Source/Standards	+
Familiarity	-
Performance	++
Interoperability	++
Specific Criteria	
Scalability	++
Response Time	++
Costs for Data Storage	++
Cloud Compatibility	+
CRUD Operation Support	++
Open Format Compatibility	+

5.3 Sensor Abstraction and Interoperability Interfaces

5.3.1 Overall Functional Specification

Diverse sensors, respectively sensor data, provide a major data source within SIMPLI-CITY. 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, a homogeneous access method eases the efforts for software/service developers who want to exploit these data sources.

To provide this homogeneous access, the Sensor Abstraction and Interoperability Interfaces component is used. This component will provide the seamless integration of heterogeneous sensor sources and sensor readings within SIMPLI-CITY, e.g., by providing corresponding wrappers. In this context, the component will take care of

providing pull-based and (event-based) push-based data access as well as on demand accessible data sources.

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 (see Section 7.3).

5.3.2 Subcomponents

To achieve the functionalities described in the previous subsection, the Sensor Abstraction and Interoperability Interfaces provide the following subcomponents as depicted in Figure 16:

- Sensor Wrapper: Includes:
  - Access Handler: Allows the access to the diverse sensor data.
  - Data Wrapper: 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.
- Data Source Pull Scheduler: Takes care of pulling data from sensors according to a pre-defined time interval.
- Data Subscriber: Provides the functionality to subscribe to external event bus systems and allows receiving event triggered sensor readings.
- Historical Data Handler: Provides the means to access historical data, locally stored sensor data as well as historical sensor data in external databases.
- Sensor Abstraction Interface: An API exposing the interfaces to other SIMPLI-CITY components, respectively apps and services over which sensors and sensor data are accessed.

The Sensor Abstraction and Interoperability Interfaces are used by other SIMPLI-CITY components:

- Data Processing: To provide a means for the Data Processing component to access sensor data and to allow a processing of sensor data, e.g., for aggregation purposes.
- Service Runtime Environment: Whereas the Data Processing component will provide access to processed and contextualized sensor data, the Sensor Abstraction and Interoperability Interfaces also provide direct access to raw sensor data to the Service Runtime Environment.
- Context-based Service Personalisation: Through the usage of Context Sensors and the Context Manager (see Section 6.3), the Context-based Service Personalisation component allows service developers to subscribe to particular data sources in order to regularly pull data from different sources and check if there has been a significant data/context change.

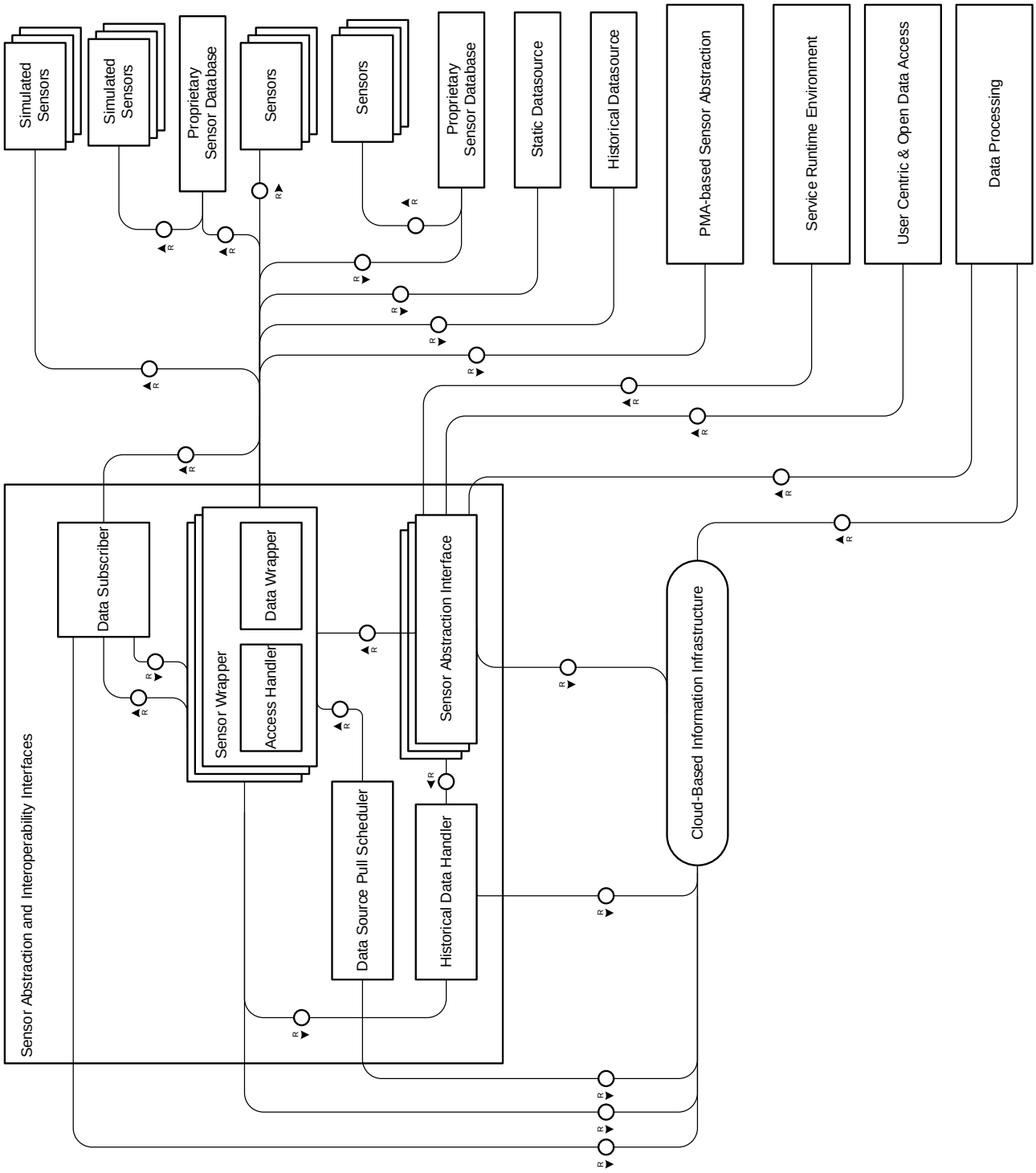

Figure 16: Sensor Abstraction and Interoperability Subcomponents and Interactions

5.3.3 Related Requirements

Table 6: Requirements Related to the Sensor Abstraction and Interoperability Interfaces

Requirement	Handled by Subcomponent	Comment
Must Have Requirements		
U31: Reaction to time of the day	Data Source Pull Scheduler	This requirement describes the functionality to capture sensor readings of predefined points in time for late usage. Thus this component will request such values at the scheduled times and store them in the Cloud-based Information Infrastructure.
U32: Reaction to sensors	Data Subscriber	This requirement describes the functionality to react on sensor changes that cause an event. Such sensors only provide their data at the moment the event occurs, thus this component must react to this event and store the sensor value for later usage.
U109: Access to remote sensors (P1) U202: Diagnosis of abnormal traffic condition in real-time U203: Prediction of abnormal traffic condition U204: Support for querying diagnosis historic U205: Support for querying impact factor on traffic condition	Sensor Wrapper	This requirement describes one of the basic functionalities of the Sensor Integration and Interoperability Interfaces. It has to provide means for SIMPLI-CITY apps and services to access data from various sensor sources in a unified manner and thus realize the concept of Mobility-related Data as a Service. Therefore, a unified data description and access model has to be provided. However, it has to be noted, that within this component data sources from the car and the PMA are not considered. Such data sources are addressed within the PMA-based Sensor Abstraction component (see Section 7.1).

Requirement	Handled by Subcomponent	Comment
U189: Unified interface for accessing sensors (P1)	Sensor Abstraction Interface	This requirement describes the basic functionality of the Sensor Integration and Interoperability Interfaces of providing a unified access method to heterogeneous sensors for app and service developers. Thus, building on a unified data description and access model (see requirement U109), an interface has to be provided for app and service developers, which allows a unified access to sensors independent of their (heterogeneous) types. However, it has to be noted, that within this component, car sensors and PMA-based sensors are not considered. These sensor sources are addressed within the PMA-based Sensor Abstraction component (see Section 7.1).
Should Have Requirements		
U112: Support of open data	Sensor Wrapper	This requirement describes the need to provide the means for SIMPLI-CITY apps and services to access data from various open data sensor sources in a unified manner and thus realize the concept of Mobility-related Data as a Service. Therefore, a unified data description and access model has to be provided. Such open data sensor sources could be, e.g., parking slot sensors of a parking facility.
U114: Configuration of the frequency of update of the data from data sources (P1)	Data Source Pull Scheduler	This requirement describes the necessity of being able to manage data streams from sensor sources in the sense of configuring them to, e.g., required data bandwidths, etc., for example by adjusting update intervals.

Requirement	Handled by Subcomponent	Comment
U120: Handle data streams (P1)	Sensor Wrapper, Data Subscriber	This requirement describes the need for being able to handle sensor data coming in as streaming data. In this context, an analysis and interpretation of this streaming data might be necessary to manipulate and transform the incoming data to specific system needs.
U191: Simulation of sensors (P1)	Sensor Simulation	This requirement describes the need to provide developers with simulated sensors and simulated sensor data. This is necessary to decouple the development process for apps and services from the actual access to real sensors. This would significantly simplify the development, as it is expected that developers do not necessarily have easy access to real sensors. Thus, by using the simulated sensors and simulated sensor data they can still develop their solutions without the requirement of accessing real sensors.

5.3.4 Interaction with other Components

5.3.4.1 Interaction with Sensors

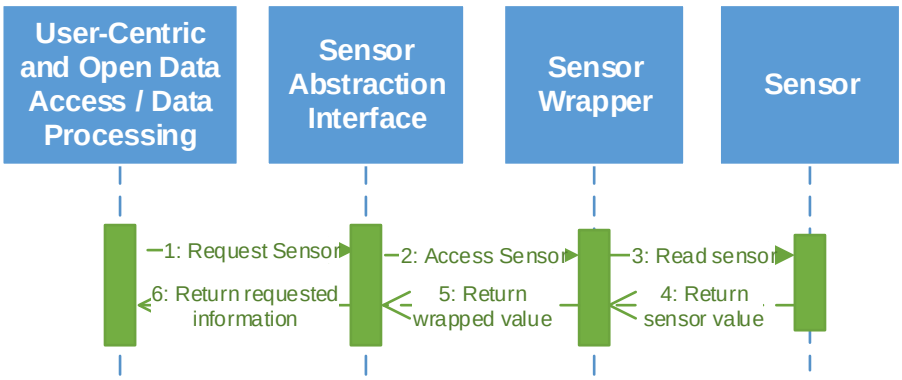

Figure 17: Interaction of User-Centric and Open Data Access / Data Processing Components and Sensors

Figure 17 shows the interaction between the User-Centric and Open Data Access component, the Data Processing component, and the Sensor Abstraction and Interoperability Interfaces. Both components will communicate with the Sensor Abstraction and Interoperability Interfaces to request sensor data. Also sensors attached to the PMA

can be accessed in a very similar way; this is described in more detail in Section 5.3.4.5. If sensor data is requested within SIMPLI-CITY, the request is passed to the Sensor Abstraction and Interoperability Interfaces that allow accessing sensors by the use of Sensor Wrappers. This component contains the Access Handler (not depicted) that is responsible to access the sensor and the Data Wrapper (not depicted) that transforms the returning sensor data into the common data format.

For each data source initially a Sensor Wrapper, comprised of an Access Handler and a Data Wrapper has to be defined to integrate the data source to the system. In the next step, the Sensor Wrapper passes the transformed data back to the Sensor Abstraction Interface that is now able to return the requested information. For all sensors or sensor data sources that are accessible on demand, the data flow is the same, thus the “Sensor” depicted in Figure 17 could be sensors, simulated sensors or static data sources. In addition, the Service Runtime Environment is also able to access the Sensor Abstraction and Interoperability Interfaces and a request for sensor data is handled in the same way as previously described for the other components.

5.3.4.2 Interaction based on Event-Based Sensor Readings

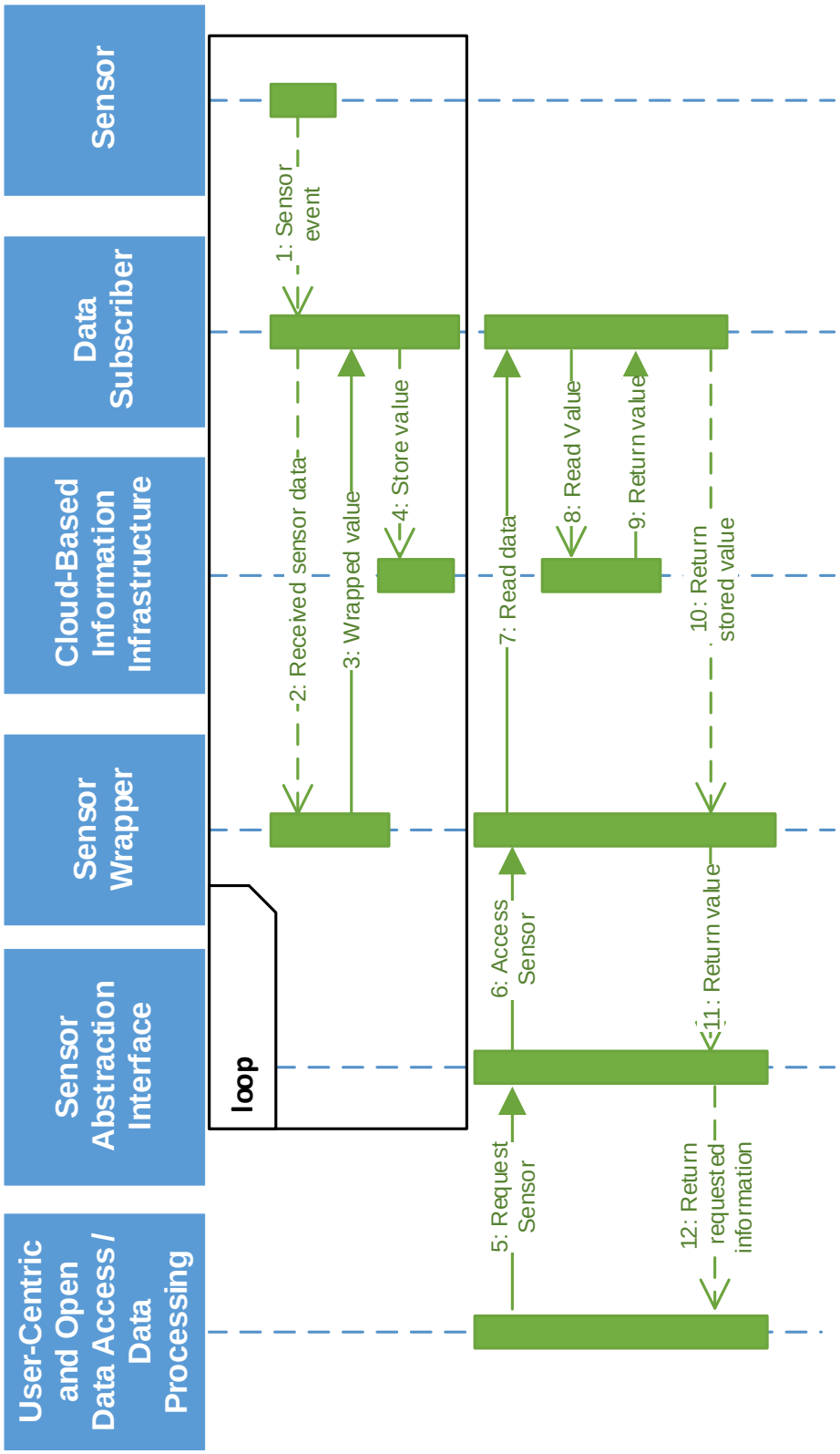

Figure 18: Interaction of User Centric and Open Data Access / Data Processing Components and Sensor Data of Sensor Events

In case of sensors that provide their data only if a respective event occurs, the interaction is depicted in the figure above. To handle this type of data source the Data Subscriber provides the means to subscribe to external event bus systems. If an event occurs, the received data is passed to the Sensor Wrapper to transform the data into the common data format and then the wrapped data is stored in the Cloud-Based Information Infrastructure for later usage (Steps 1-4). To enable this functionality, some preparative steps have to be done that are not depicted in the figure above. In the first step, the address of the Data Subscriber component has to be registered at the data source to enable the transmission of the respective sensor events. In the second step, a customized Data Wrapper has to be prepared by a developer to enable the translation of the proprietary external data format into the common SIMPLI-CITY data format.

If such data is requested by the User Centric and Open Data Access or the Data Processing component from the Sensor Abstraction Interface, these components asks the Sensor Wrapper to access the respective data. The Sensor Wrapper's Access Handler knows how to request this type of data from the Data Subscriber that reads the requested value from the Cloud-Based Information Infrastructure and returns the answer of the query back to the Sensor Wrapper that is now able to return the data to the Sensor Abstraction Interface that passes back the requested information.

5.3.4.3 Interaction based on Pulled Data

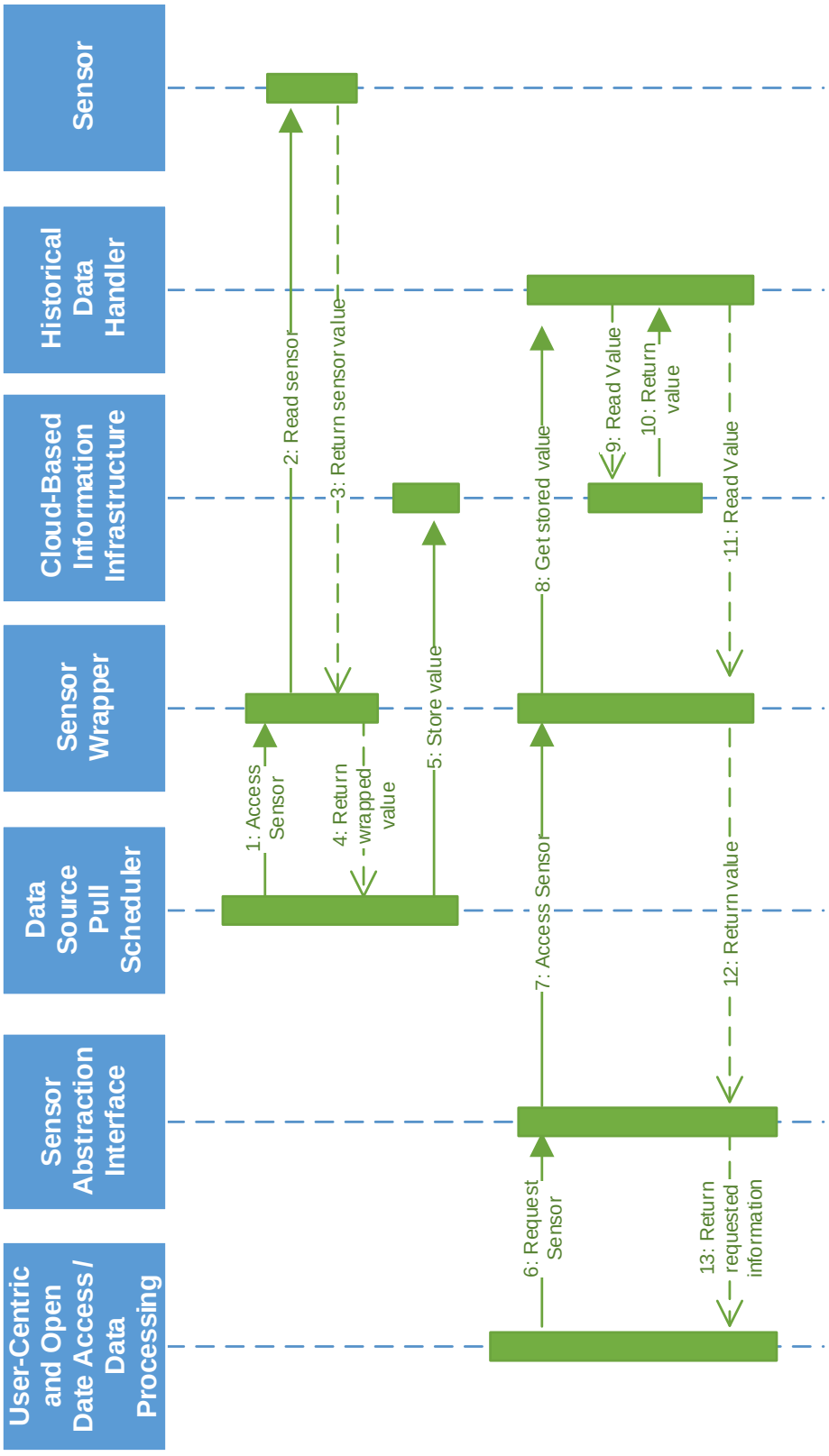

Figure 19: Interaction of User-Centric and Open Data Access / Data Processing Components and Actual Values of Stored Sensor Data

The Data Source Pull Scheduler provides the means to request for particular sensor data periodically in specific intervals or special points in time. Such data sources are registered at the Data Source Pull Scheduler that requests the respective data from the Sensor Wrapper at the target time. The Sensor Wrapper accesses the external data source, requests the data and wraps the received data into the common data format. This data is then returned to the Data Source Pull Scheduler that stores the wrapped data in the Cloud-Based Information Infrastructure for later usage (Steps 1-5).

If such data is requested by the User Centric and Open Data Access or the Data Processing component from the Sensor Abstraction Interface, this components asks the Sensor Wrapper to access the respective data. The Sensor Wrapper's Access Handler (not depicted) knows how to request this type of data from the Data Source Pull Scheduler that reads the requested value from the Cloud-Based Information Infrastructure and returns the answer of the query back to the Sensor Wrapper that is now able to return the data to the Sensor Abstraction Interface that passes back the requested information. The corresponding interaction is given in the figure above.

For all sensors or sensor data sources that are accessible on demand, the data flow is the same, thus the "Sensor" depicted in the figure above could be sensors, simulated sensors or static data sources.

5.3.4.4 Interaction with Historical Data

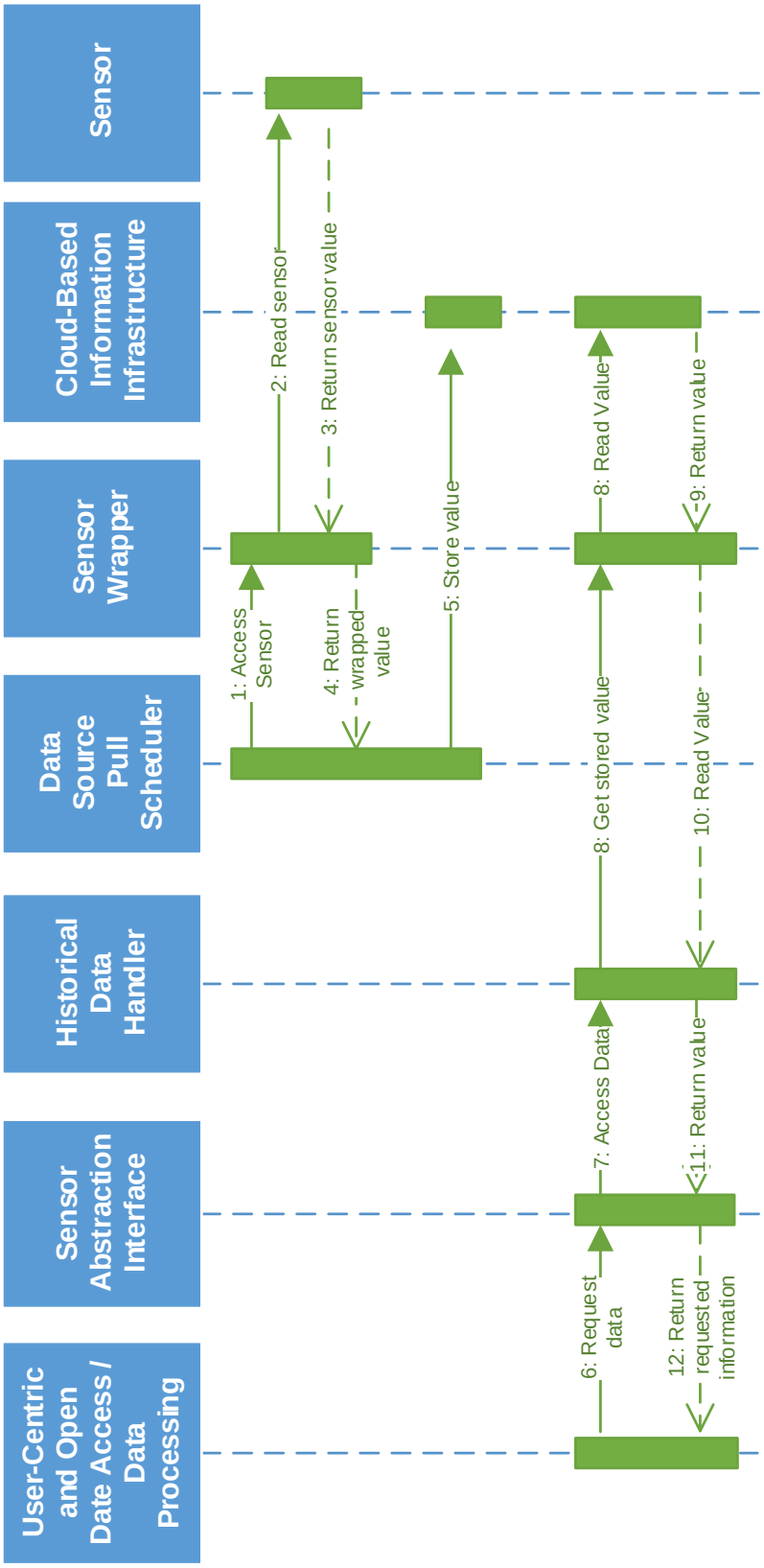

Figure 20: Interaction of User-Centric and Open Data Access / Data Processing Components and Actual Values of Stored Sensor Data

The Historical Data Handler provides the means to access particular sensor data from the past. The precondition to provide historical values of sensor readings is to store these data. As mentioned in Sections 5.3.4.2 and 5.3.4.3, the Data Source Pull Scheduler as well as the Data Subscriber have the capability to store sensor readings in the Cloud-Based Information Infrastructure. If such historical data is available in the Cloud-Based Information Infrastructure, requests for such data to the Sensor Abstraction are queried to the Sensor Wrapper that is able to directly read the respective data out of the Cloud-Based Information Infrastructure. The interaction of the involved components of the data storage and retrieval process is depicted in Figure 20.

In case of external data sources, the interaction is very similar. As depicted in Figure 21, in this case the Historical Data Handler directly accesses the external data source via the Sensor Wrapper. Obviously this data flow is only possible, if the external source provides historical data. If historical data of other sources is needed, the previously mentioned mechanism provides the means.

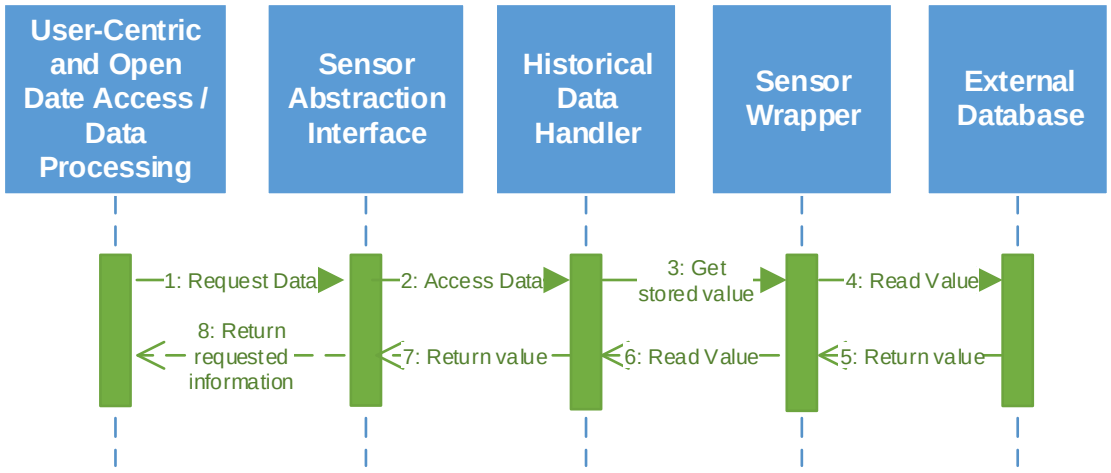

Figure 21: Interaction of User-Centric and Open Data Access / Data Processing Components and Externally Stored Historical Data

5.3.4.5 Interaction with User-Related Data

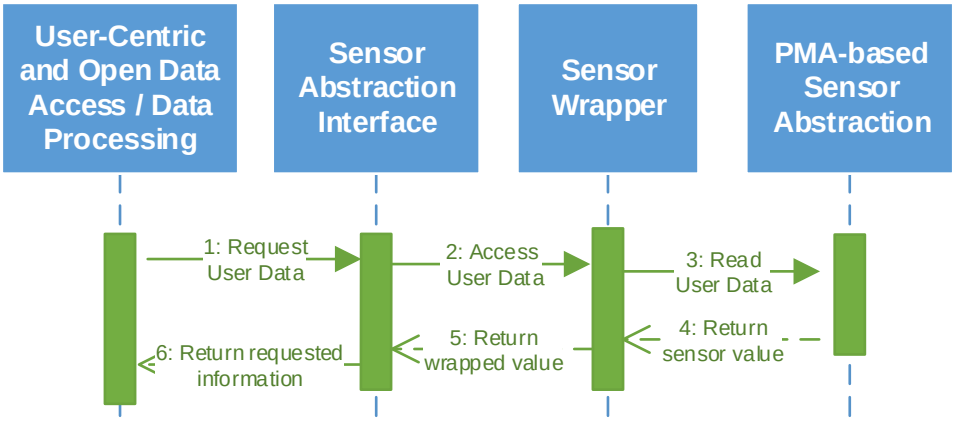

Figure 22: Interaction of User-Centric and Open Data Access / Data Processing Components and User-Related Data Accessible on the PMA

Figure 22 shows the interaction between the User-Centric and Open Data Access and user-related data that is accessible on the PMA. Since the access to user-specific contact

and calendar accounts is already configured by the user on the PMA, the PMA-based Sensor Abstraction service can easily access all these data in a common way without the necessity to have knowledge about the different user specific logins for the different third party services. If user-related data is requested within SIMPLI-CITY, the request is passed to the Sensor Abstraction and Interoperability Interfaces that allows accessing this data by the use of Sensor Wrappers.

This component contains the Access Handler (not depicted) that is responsible to access the PMA-based Sensor Abstraction and the Data Wrapper (not depicted) that transforms the returned user data into the common data format. A special wrapper will be responsible to access the user-related data on the PMA via the PMA-based Sensor Abstraction Service. In the next step, the Sensor Wrapper passes the transformed data back to the Sensor Abstraction Interface that is now able to return the requested information.

User data will also be cached on the server side to allow the data processing components to access this data even if the PMA is offline. Sensors readings of sensors attached to the PMA or other sensors accessible through the PMA-based Sensor Abstraction, described in Section 7.3, are in the same way accessible via the Sensor Abstraction Interfaces. So the interaction for such data sources is the same as depicted in Figure 22.

5.3.5 User Interface

The Sensor Abstraction Interoperability Interfaces will not feature a specific user interface.

5.3.6 Conceptual Data Model

The conceptual sensor data model is described in Section 9.1.

5.3.7 Parameters to Take into Account for Technical Specification

Table 7: Criteria for Technical Specification

Parameter	Importance (--, -, -+, +, ++)
Generic Criteria	
Up-to-Datedness	+
Stability	+
Extensibility & Open Source/Standards	-+
Familiarity	-+
Performance	+
Interoperability	++
Specific Criteria	
Lightweight	--

Parameter	Importance (--, -, -+, +, ++)
Ease of Use	+
Extensible Data Model	++
Platform (Portability)	+

5.4 Media Data Streams and Data Prefetching Logic

5.4.1 Overall Functional Specification

This component will realize media handling facilities in terms of media data streaming and media prefetching (e.g., for music streaming). Secondly, this component will provide service prefetching functionalities as well (for preinvocation of data services and backend services).

Media streaming and media data prefetching may be used to create apps that support the playback of music or other media information. By nature, the consumption of media is sensitive to interruptions: Even a connectivity loss of a few seconds will give the user a bad experience, if it happens in the middle of a song the user is listening to. For this purpose, the Media Data Streams and Data Prefetching Logic will integrate a media buffering solution by prefetching relevant data in a local buffer.

Service prefetching will invoke services, which the user is likely to use in the future. The prediction/selection of these services is done by monitoring the user's behaviour and combining it with historical behaviour of SIMPLI-CITY users. The service selection will be made on the mobile device, based on local user settings and local sensor data, and on the server side, based on historical behaviour of SIMPLI-CITY users. The results of service preinvocations will be buffered on the mobile device.

5.4.2 Subcomponents

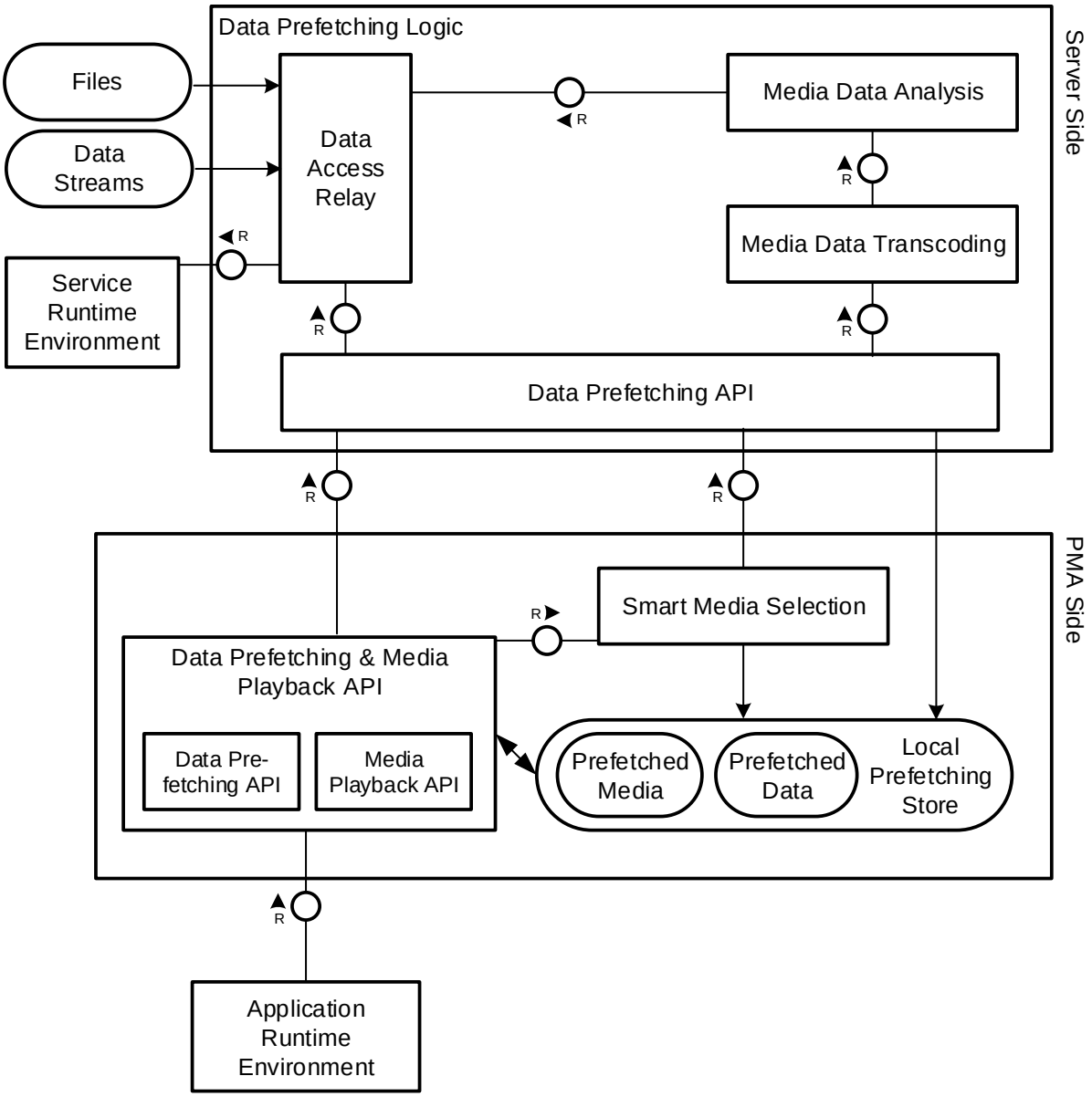

Figure 23: Media Data Streams and Data Prefetching Subcomponents and Interactions

To achieve the functionalities described in the previous subsection, this component provides the following subcomponents as depicted in the figure above:

PMA Side:

- Data Prefetching & Media Playback API: Provides an API that can be accessed from outside the component. This component consists of two parts:
  - Data Prefetching API (PMA Side): Provides data prefetching (direct access to the Data Prefetching API) and mainly targets app developers.
  - Media Playback API: Provides basic media playback facilities for Cloud-based media files (via the Smart Media Selection) and mainly targets app developers. Files are played from the Local Prefetching Store after the prefetching process was initiated.

- Smart Media Selection: Automatically selects files to be prefetched and initiates the download process. Files may also be streamed on-demand if a file is requested by the Local Prefetching Store. The decision on which data is automatically selected for the prefetching process is described in Section 5.4.4.3.
- Local Prefetching Store: Provides a local PMA-specific data buffer, which is part of the Local Storage (see Figure 3 – Global Architecture). If a file is requested that is currently not in the store, the process depicted in Section 5.4.2.2 will be initiated for Data Prefetching.

Server Side:

- Data Prefetching API (Server Side): Provides the access point for accessing the prefetching logic. This is the only Server Side subcomponent able to store data in the Local Prefetching Store.
- Data Access Relay: Receives (media) data from external data sources.
- Media Data Analysis: Analyses the quality of the media data stream or file.
- Media Data Transcoding: Adopts the media data to the context of the PMA.

5.4.2.1 Media Playback

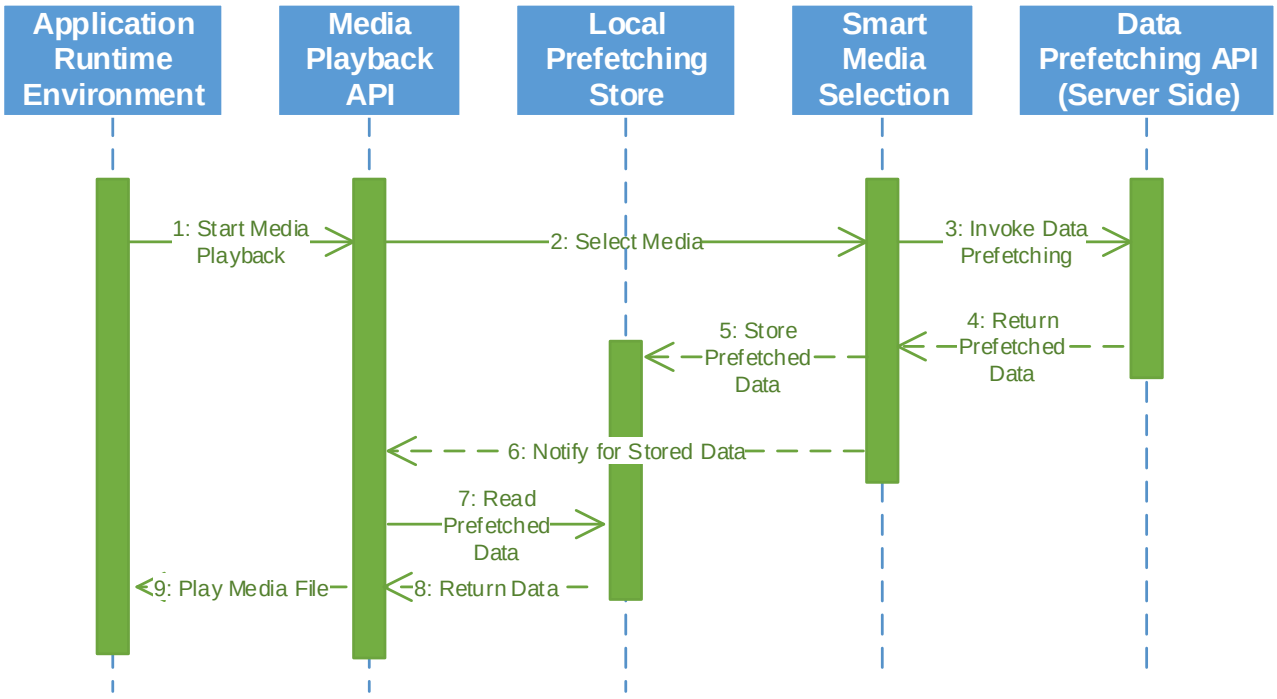

Figure 24: Media Playback via the Media Data Streams & Data Prefetching Logic on the PMA Side

App developers, who want to make use of services that use data prefetching, can do so by marking their app likewise, thus the Application Runtime Environment is able to differ between:

- Prefetchable data (see Figure 28) and
- Data that should be prefetched from a backend service (see Figure 29).

In the first case, media data will be directly accessed by querying the data source of the media. A data source may be a file being available via direct web requests (e.g., by pointing to a URL of an MP3 file). This access is performed by the Data Access Relay,

which delivers the media data to the Data Prefetching API (Server Side), which is itself called by the PMA side via the Media Playback API. Additionally, media streams will be supported, e.g., by supporting public audio streams from web radios. Section 5.4.4.1 shows this direct interaction with data sources in more detail.

In the second case, data will not be directly accessed but received from a backend service as described in Section 5.4.4.2. As such, data will be accessed via the Application Runtime Environment, which will forward a service invocation to the Service Runtime Environment.

In both cases, prefetching is handled in the background and not used by other SIMPLI-CITY components directly. As such, app developers will make use of the Media Playback API to play media within their app without having to care about any prefetching logic.

As can be seen in Figure 24, the Media Data Streams and Data Prefetching Logic component is invoked via the Application Runtime Environment, which triggers the Media Playback API to create an environment for media files to be played. The Smart Media Selection calls the Data Prefetching API, which prepares the remote file/stream to be stored in the Local Prefetching Store. As soon as the environment is created and the runtime parameters have been set (see Media Data Analysis in Figure 25) segments of the media file are periodically stored in the Local Prefetching Store and sent to the Application Runtime Environment via the Media Playback API.

The Server Side (see Figure 25) of the component handles the preparation and transfer of media files. The Media Data Transcoding subcomponent receives an initial request for a media file by the Data Prefetching API; this would happen directly after Step 3 from Figure 24. Afterwards, the context of the media file is forwarded to the Media Data Analysis subcomponent, which receives the raw media file from the source and adds relevant metadata to the data package (e.g., length, possible sum of segments regarding available bandwidth), which is then transferred to the Data Access Relay. In general, the source (depicted in Figure 25 as “Media Files”) could be any media data provider on the web, e.g., a media stream provider or a Cloud-based data store.

The Data Access Relay then provides the Data Prefetching API with the relevant metainformation as well as the segmented data transfer. As shown in Figure 24 (Step 4 and Step 5), this segmented data is stored in the Local Prefetching Store and handled locally (i.e., on the mobile device) for a fluent playback of media.

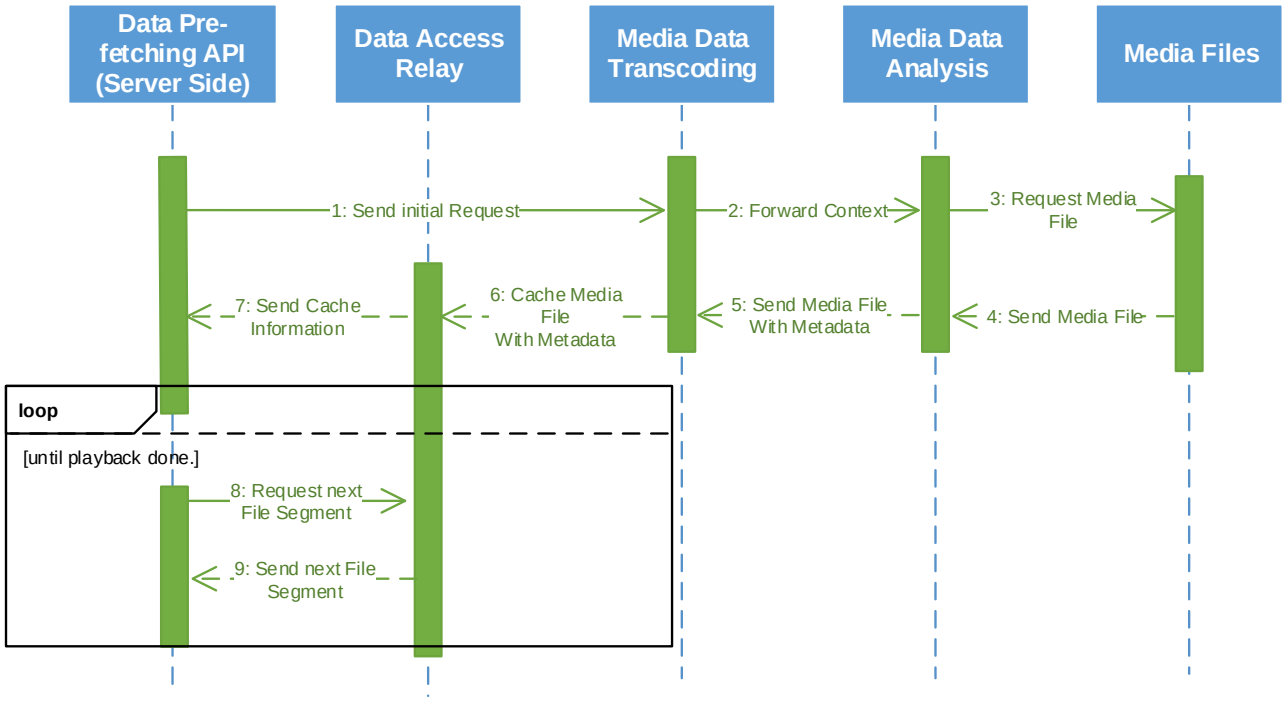

Figure 25: Media Playback via the Media Data Streams & Data Prefetching Logic on the Server Side

5.4.2.2 Data Prefetching

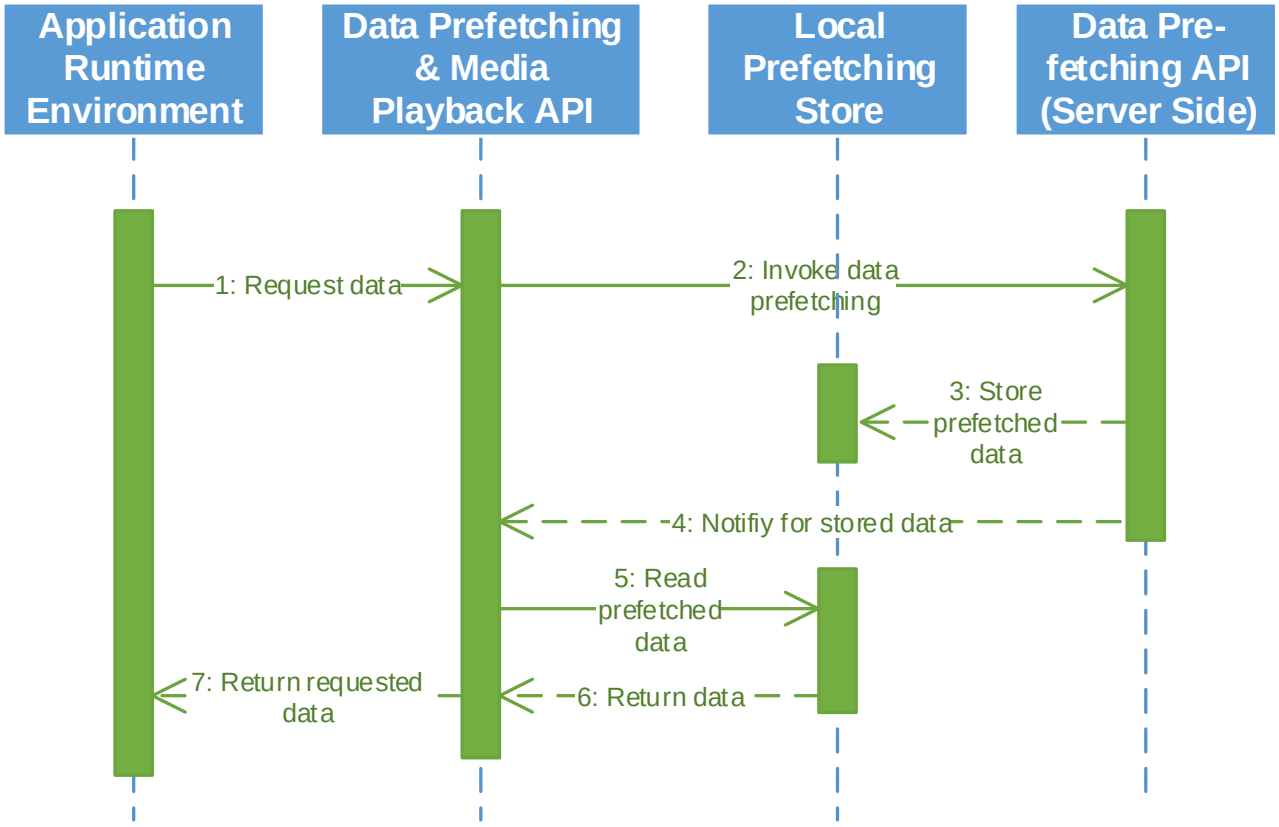

Figure 26: Data Prefetching via Media Data Streams & Data Prefetching Logic Component on the PMA Side

Data prefetching is quite similar to the media streaming presented in the previous subsection. The difference between the methods is that the Media Data Encoding and Media Data Analysis subcomponents are not used for data prefetching. The check for stored prefetched data is omitted in Figure 26 to keep the focus on the main functionality. The Data Prefetching API is directly invoked by the Data Prefetching & Media Playback API to receive data via the Data Access Relay and store the received data in the Local Prefetching Store. The Data Prefetching API then reads the prefetched segments and returns a notification to the caller. The Data Prefetching & Media Playback API now reads the prefetched data and finally returns it to the Application Runtime Environment.

Figure 27, shows the direct access of a data file from the server side. For this purpose, an initial request via the Data Access Relay is performed, which reads the data file and handles the concrete access of data from the server side. Data files are buffered and then forwarded to the Data Prefetching API. The transfer of files is performed in a loop allowing the partial delivery of data from the server side to the PMA side.

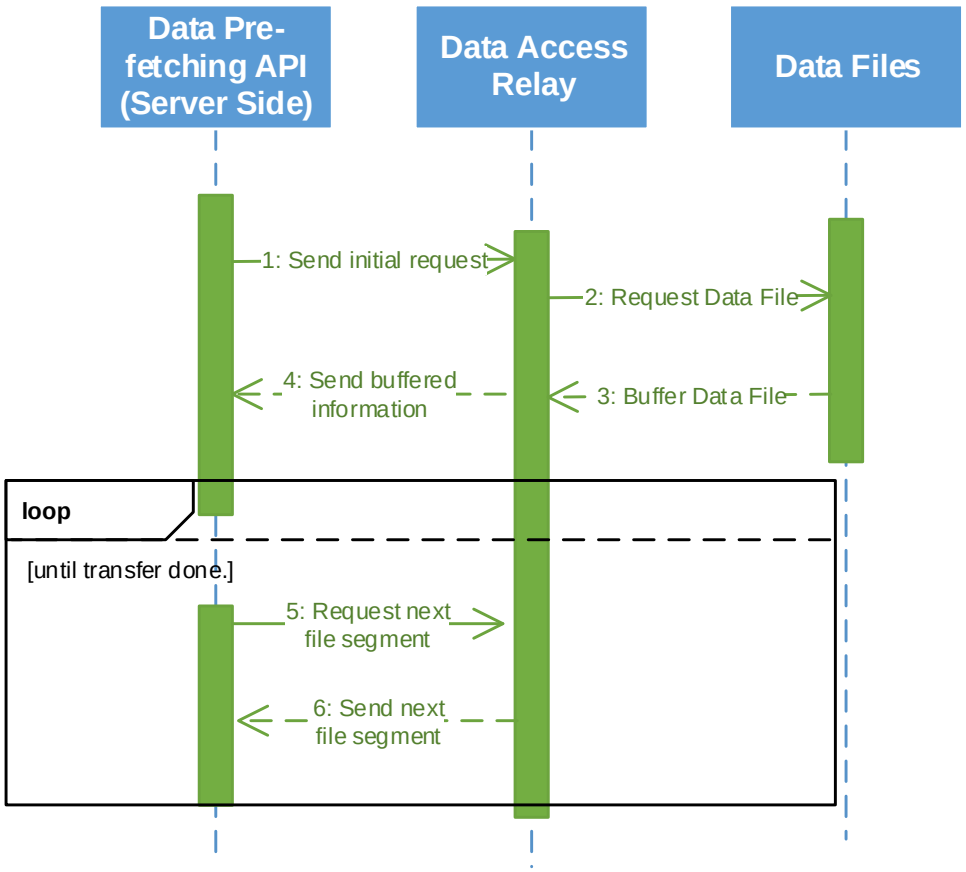

Figure 27: Data Prefetching via Media Data Streams & Data Prefetching Logic Component on the Server Side

5.4.3 Related Requirements

Table 8: Requirements Related to the Media Data Streams and Data Prefetching Logic

Requirement	Handled by Subcomponent	Comment
Must Have Requirements		
U50: Prefetching of media data + Offline access (P1)	Local Prefetching Store Smart Media Selection Data Prefetching API Media Data Transcoding Media Data Analysis Data Access Relay	This is the main functionality of this component and therefore will be supported by all subcomponents.
U113: Handling of multimedia data (P1)	Data Prefetching & Media Playback API	Multimedia data streaming is facilitated through an API, which allows integrating media streaming in apps.
Should Have Requirements		
U51: Avoid the download of data from 3G (P2)	Smart Media Selection Local Prefetching Store Media Data Transcoding	In many countries, there is a bandwidth cap in 3G networks. Hence, it should be avoided to download large amounts of data in such networks. In SIMPLI-CITY, large amounts of (media) data will be therefore prefetched as long as the mobile device is connected through a WIFI.
U52: Offline access of data used within apps (P2)	Local Prefetching Store Data Prefetching API Data Access Delay	It might be helpful to allow offline access of data, e.g., in order to make sure that media playback can be used even in areas with bad coverage or abroad (without having to pay roaming fees). Hence, SIMPLI-CITY will prefetch such data. Naturally, prefetched data will be outdated at a particular point of time. Data that is used more than once (e.g., media data) could be stored with a longer lifecycle to avoid data transfer redundancies.

Requirement	Handled by Subcomponent	Comment
U54: Expiration of data (P4)	Local Prefetching Store	Allows storing data that is valid for the current context. The prefetched data has a lifecycle that is only valid for a certain time or context.
Could Have Requirements		
U53: App recommendations (P4)	Data Prefetching API	The PMA can recommend apps due to the driver's driving behaviour and regularly visited places.

5.4.4 Interaction with other Components

The Media Data Streams and Data Prefetching Logic component interacts with other components of SIMPLI-CITY. The Media Playback API makes use of the prefetching component to retrieve and store stream-data on the PMA; such data transfer can be done with or without actual prefetching data.

Data may be accessed by the Data Access Relay in two ways:

- In case of streaming data or accessing files, the data is requested via the server side of the Media Data Streams and Data Prefetching Logic component directly using the Data Access Relay subcomponent.
- Data from services are requested via the Application Runtime Environment for invoking the Service Runtime Environment which will provide access to the (internal) backend services, handling the communication with and transfer of data from the data sources and then return the data to the Data Prefetching & Media Playback API, which will store the data in the Local Prefetching Store.

In the following subsections, the different interactions between the Media Data Streams and Data Prefetching Logic component and other SIMPLI-CITY components will be explained in more detail.

5.4.4.1 Interaction with Data Source

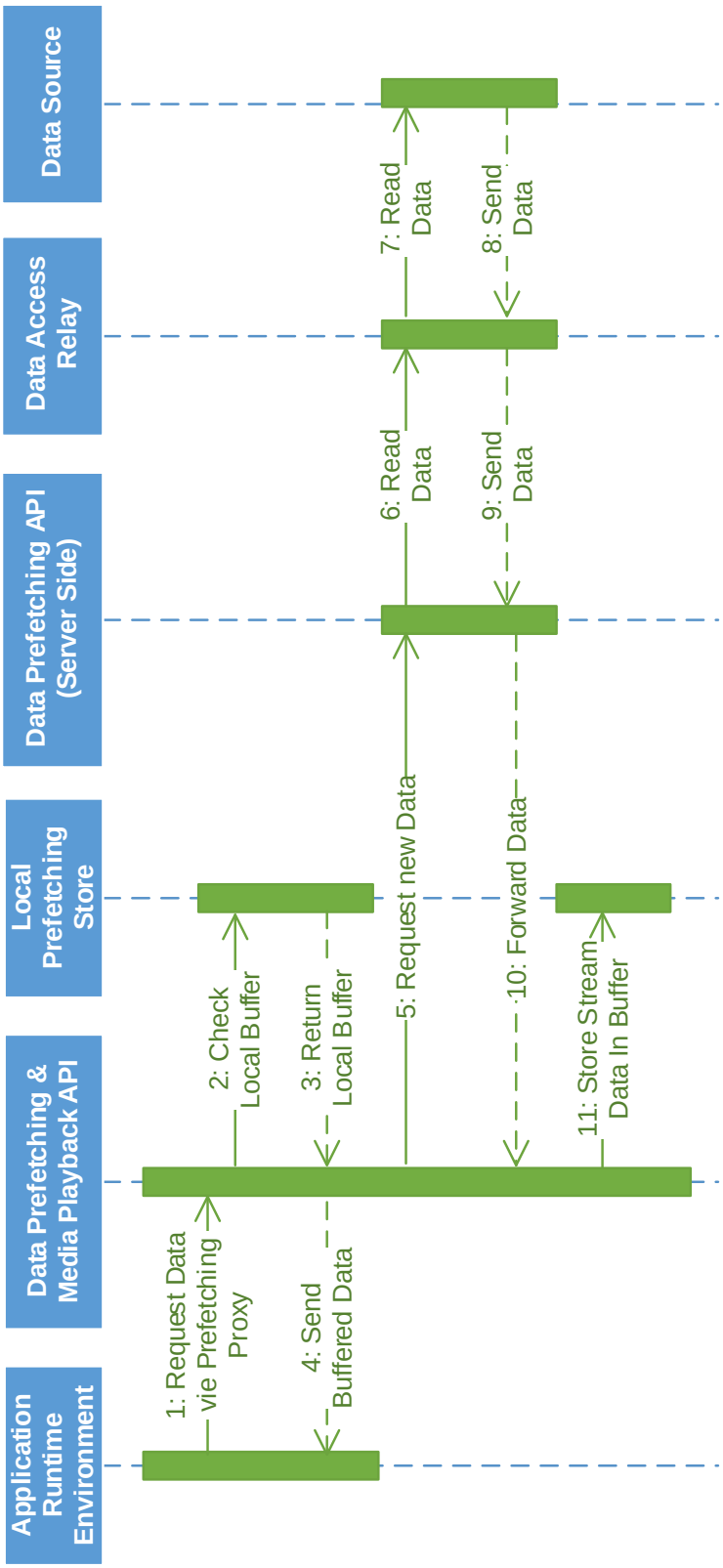

Figure 28: Interaction between Media Data Streams and Data Prefetching Logic and Data Sources

As can be seen in Figure 28, the Media Data Streams and Data Prefetching Logic component uses the Local Prefetching Store as a local buffer for data streams. As the Application Runtime Environment calls the Data Prefetching & Media Playback API, the local buffer (Local Prefetching Store) is checked for unread data, which is then forwarded to the Application Runtime Environment (Steps 1-4). The Data Prefetching & Media Playback API then requests new data from the Data Source via the server side and the Data Access Relay to ensure that the buffer is always filled with new data (Steps 5-11). The Data Source could be any source of data which provides direct data access (e.g., information from the Cloud-based Information Infrastructure) or streaming information (e.g., a public media stream). Please note that backend services provided by the Service Runtime Environment may also be data sources – thus, SIMPLI-CITY allows backend service prefetching.

Importantly, the Media Data Streams and Data Prefetching Logic component allows apps to automatically make use of its prefetching functionalities during runtime. For this purpose, app developers will need to implement an abstract class and to make use of annotations as described in Section 5.4.4.3.

5.4.4.2 Interaction with Service Runtime Environment and Data Sources

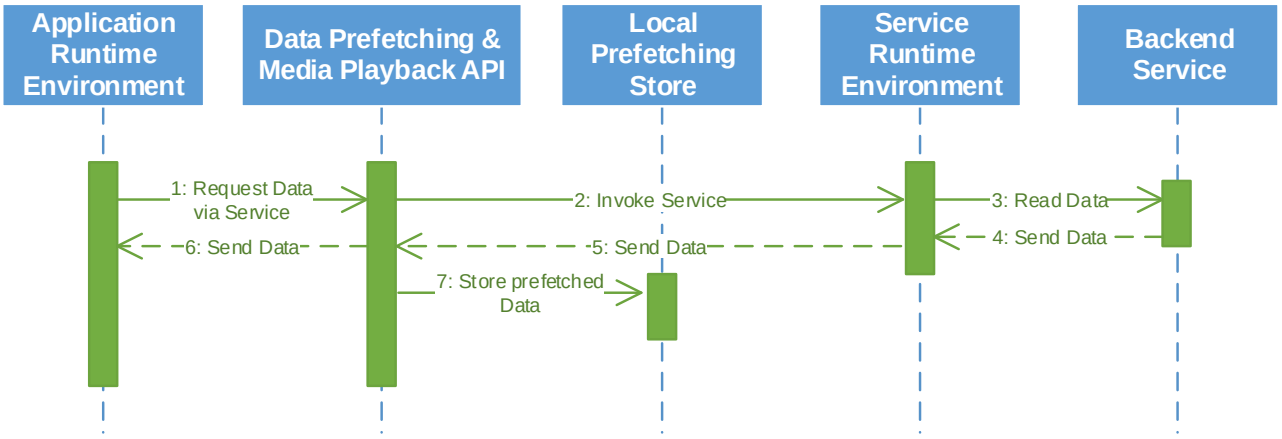

Figure 29: Interaction between Media Data Streams & Data Prefetching Logic and the Service Runtime Environment – Prefetching Data from Backend Services

External components, i.e., components which are part of the SIMPLI-CITY framework but not part of the Media Data Streams and Data Prefetching Logic component, can use the Data Prefetching & Media Playback API to access prefetched data via the Application Runtime Environment, regardless of the source of the data. Figure 29 shows how the Data Prefetching & Media Playback API accesses a Backend Service via the Service Runtime Environment, if data is not available in the Local Prefetching Store. The decision whether to just access data from the Local Prefetching Store or first perform a prefetching is analogue to the one depicted in Figure 28. Notably, a backend service could encapsulate any data service, such as a sensor, data from the Cloud-based Information Infrastructure, etc.

Prefetched data which is not directly accessed (as described in Section 5.4.4.1) is always linked to a particular backend service. As such, the Service Runtime Environment is able to send the data itself via the specific service. As shown in Figure 29, the data that is received is stored in the Local Prefetching Store. Once data is stored in the Local Prefetching Store, it may be received directly from there without the need to query the

service (as shown in Figure 26). As such, the process usually starts with querying the Local Prefetching Store and will only invoke the service if data is not already on the store. For simplicity reasons, this part is not shown in Figure 29 as it is identical to Steps 2 and 3 of Figure 28.

App developers may make use of an abstract class to mark which data is prefetchable and which is not. If data is not marked as prefetchable, the Local Prefetching Store is not invoked at all and data is directly received from the Backend Service (see Figure 30).

As described in Section 5.4.4.1, in order to make use of the functionalities of the Media Data Streams and Data Prefetching Logic component, it is necessary to access services and data sources via proxies.

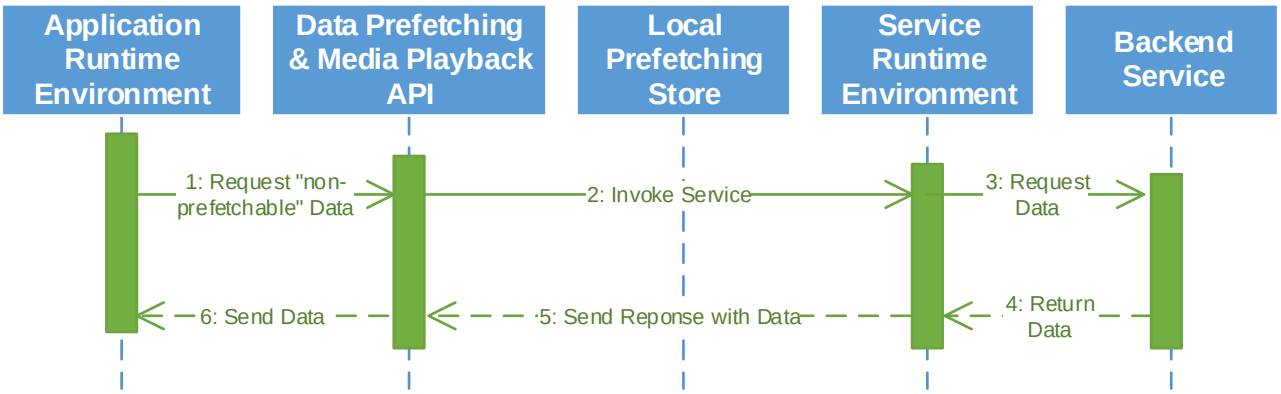

Figure 30: Interaction between Media Data Streams & Data Prefetching Logic and the Service Runtime Environment if Prefetched Data is not Available

5.4.4.3 Interaction with the Application Runtime Environment

It is important to note that apps do not need to actively care about the use of the Media Data Streams and Data Prefetching Logic except for implementing an abstract class during development time. This implementation marks those requests for accessing data which are considered to be prefetchable and therefore will be handled by the Meta Data Streams and Data Prefetching Logic.

During runtime, apps will continue to access their data as they would do without the Media Data Streams and Data Prefetching Logic. If data is available in the Local Prefetching Store, the Application Runtime Environment will deliver it directly to the apps via the Data Prefetching Playback API as shown in Section 7.1.4.

The Data Prefetching & Media Playback API will fill the Local Prefetching Store with data based on information about apps that it receives from the Application Runtime Environment. For this purpose, three different methods , which allow the component to automatically decide which data is to be prefetched, will be used:

- Continuous polling for Prefetchable Data
- Prefetching based on Historical Data
- Prefetching based on Events from the Context-Based Service Personalisation Component

For the purpose of clarity, it should be noted that the Media Data Streams and Data Prefetching Logic component will use all three methods in parallel based on the information from the Application Runtime Environment. Whether or not a data set is prefetched is decided by the Data Prefetching & Media Playback API autonomously based

on the size of the Local Prefetching Store, the connection details, the context, the user settings and type of data.

Method 1: Continuous polling for Prefetchable Data

App developers may implement an abstract class when implementing a class which requests data via the Application Runtime Environment and the Service Runtime Environment. Calls by classes which implement this abstract class will be routed via the Meta Data Streams and Data Prefetching Logic component and receive data directly from the Local Prefetching Store if available. It should be noted that the usage of the abstract class is optional so that app developers may or may not make use of the prefetching functionality.

As soon as an app, which has implemented this abstract class, is started, data is prefetched automatically in the background. Developers may use annotation to specify the polling interval, the expiration timing and the maximum size of the data set to be prefetched. Annotations may also specify a maximum duration in case of prefetching media files and a priority for marking data that the Data Prefetching & Media Playback API should consider to be more important, therefore raising the likeliness of prefetching it. For realizing this feature, the data source will be accessed via the Media Data Streams and Data Prefetching Logic component as described in Section 5.4.4.2. Figure 31 only shows the polling process for data from the services being invoked via the Service Runtime Environment since the requesting and receiving of direct data sources is already described in Figure 28.

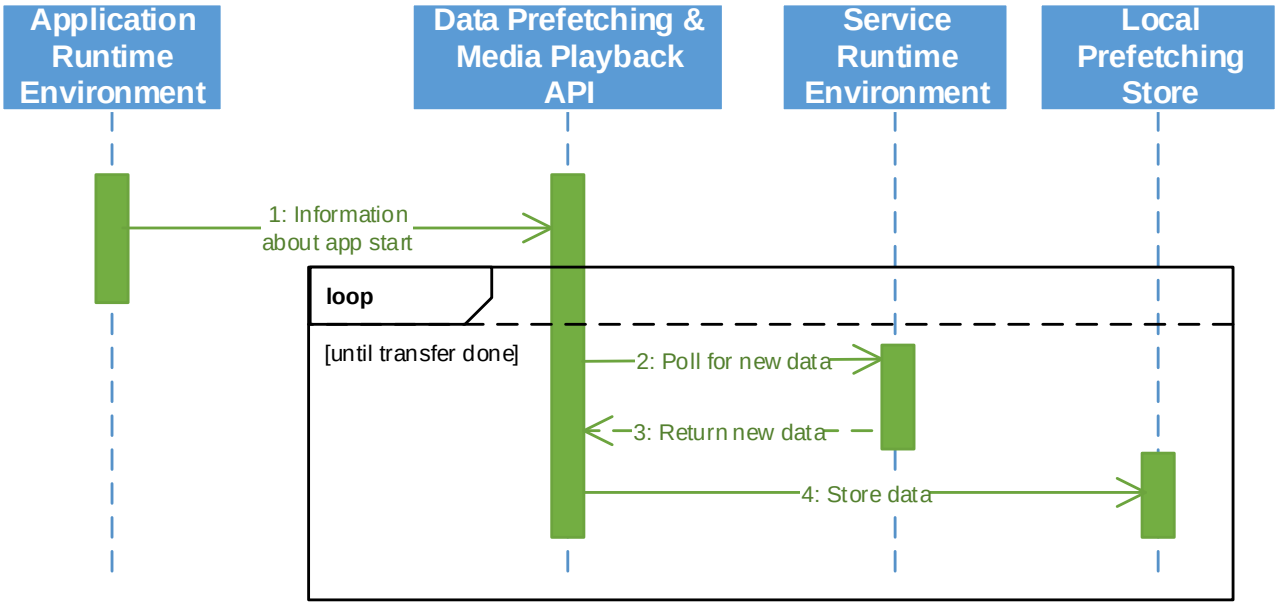

Figure 31: Interaction between Media Data Streams & Data Prefetching and the Application Runtime Environment (Method 1: Polling for Prefetchable Data)

Method 2: Prefetching based on Historical Data

Another method to invoke prefetchable services is depending on the Cloud-based Information Infrastructure, which stores information about service calls in relation to the current app. For example, such information could be that users of an app called the “Barcelona Info App” always tend to request information about free parking lots after using the app for a few minutes.

As such, SIMPLI-CITY will prefetch this data so that it can quickly be provided to the app when needed. Figure 32 shows this scenario. For enabling it, the Application Runtime Environment will inform the Data Prefetching & Media Playback API whenever an app is started at the PMA. The Data Prefetching & Media Playback API will then issue a call to the Cloud-based Information Infrastructure to request if there is any data that is regularly requested by users of this specific app. This is performed via a normal service call involving the Application Runtime Environment and the Service Runtime Environment (not shown in the figure for readability reasons). This historical data is then used as a base for requesting the data via the Service Runtime Environment and for storing it into the Local Prefetching Store. From this moment on, the data becomes available for app requests as described in Section 5.4.4.2.

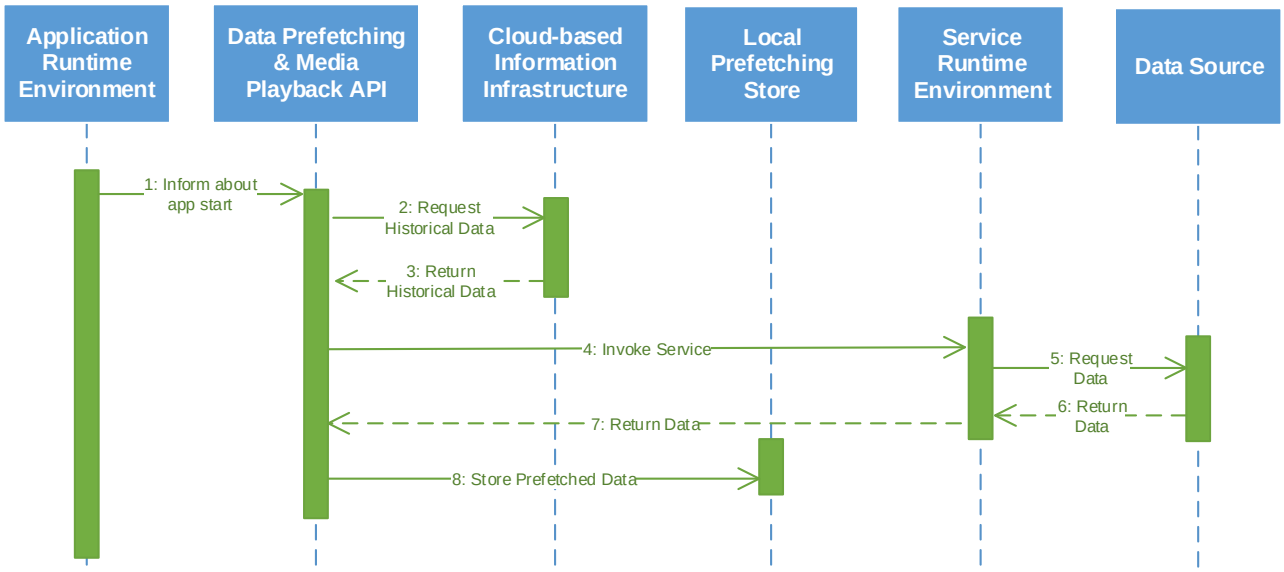

Figure 32: Interaction between Media Data Streams & Data Prefetching and the Application Runtime Environment (Method 2: Prefetching based on Historical Data)

Method 3: Prefetching based on Events from the Context-Based Service Personalisation

The third method depends on the Context-Based Service Personalisation component (see Section 6.3.4.2) which publishes context changes via a publish/subscribe service (which is part of the Context Manager).

As soon as a context change happens (e.g., fuel level is under 20%) the service that is subscribed to the relevant event channel receives a notification about the change in context (Step 4 in Figure 33). The service then pushes data to the Data Prefetching & Media Playback API to prefetch data related to the context change (here: retrieve a list of nearby gas stations). The data is written into the Local Prefetching Store (not shown in Figure 33 since it is an internal process depicted in Figure 24) and is directly accessible for apps that can handle the data. Note that the Service Runtime Environment is not depicted in Figure 33 in order to keep the figure simple; however, as described in Section 6.1, backend services are always invoked through the Service Runtime Environment.

Context changes will be grouped via topics (e.g., one topic is “fuel level”). Services can define value thresholds in order to react only on significant context changes that will trigger data prefetching. The service is unsubscribed from the publish/subscribe service as soon

as the app sends a closing signal or if the service gets unreachable. Else, the loop depicted in Figure 33 (Steps 4-6) will continue to operate.

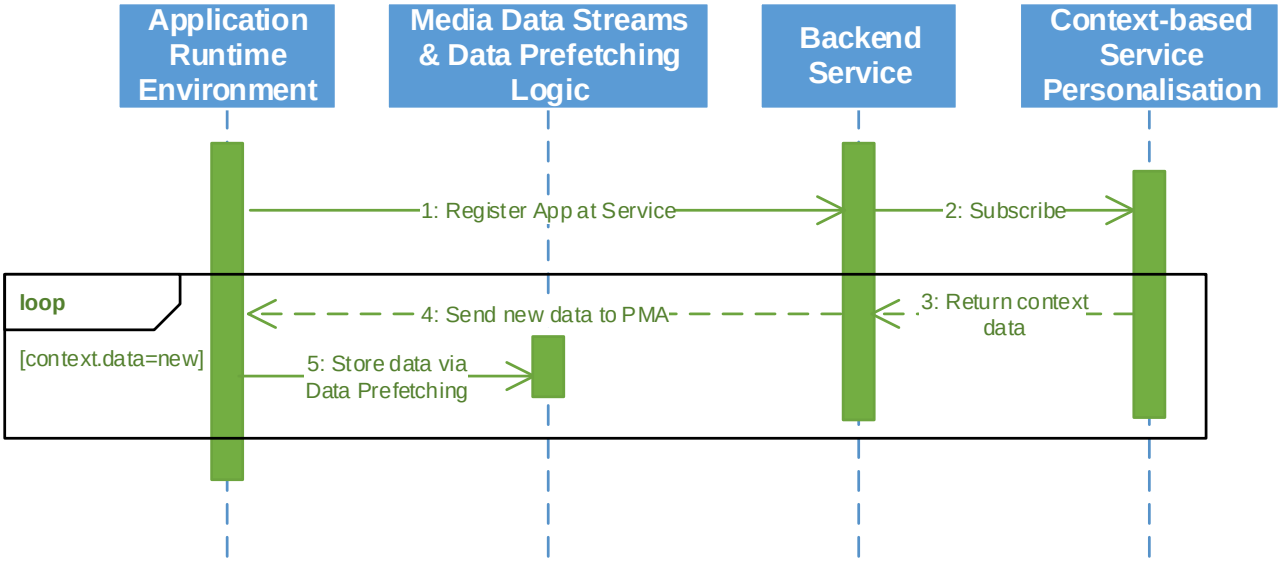

Figure 33: Interaction between Media Data Streams & Data Prefetching and the Application Runtime Environment (Method 3: Prefetching based on Events from the Context-based Service Personalisation Component)

5.4.4.4 Media Playback

Despite the prefetching functionality, the Media Data Streams and Data Prefetching Logic component also provides the functionality of media playback, e.g., for the purpose of playing music files. The process is started via the Application Runtime Environment allowing app developers to indirectly make use of the Media Data Streams and Data Prefetching Logic component and its playback functionality via the Application Runtime Environment. Figure 34 depicts the interaction, while the internal process including the data access is described in Section 5.4.2.1. The result returned to the app will be a status handler, e.g., for accessing the current location of the media file that is played.

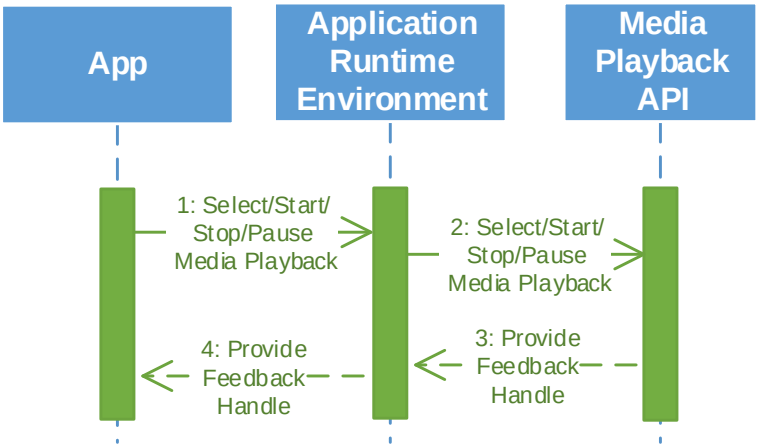

Figure 34: Interaction between Media Data Streams and Data Prefetching Logic and the Application Runtime Environment (Media Playback)

5.4.5 User Interface

This component does not have a user interface.

5.4.6 Conceptual Data Model

Apps that want to make use of the Media Data Streams and Data Prefetching Logic component have to mark their data sources as prefetchable by implementing an abstract class. Prefetchable data needs some extra information for the Media Data Streams and Data Prefetching Logic component to describe relevant information (e.g., expiration date).

5.4.7 Parameters to Take into Account for Technical Specification

Table 9: Criteria for Technical Specification

Parameter	Importance (--, -, -+, +, ++)
Generic Criteria	
Up-to-Datedness	-+
Stability	+
Extensibility & Open Source/Standards	+
Familiarity	-
Performance	++
Interoperability	++
Specific Criteria	
Zero Configuration	++
Multi-Format Support	++
Data Expiration Support	++
Streaming Capabilities	++
Small Data and Memory Footprint	++

6 Functional Specification: Mobility Services Framework

6.1 Service Runtime Environment

6.1.1 Overall Functional Specification

The Service Runtime Environment is the basic framework for the execution 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.

Usually, service requests will be done by apps running in the Application Runtime Environment, the request will then be processed in the Service Runtime Environment and the result will be routed back through the Service Runtime Environment and the Application Runtime Environment to the app. For internal backend services, the Service Runtime Environment will also control the execution in terms of resource allocation and load balancing. As SIMPLI-CITY allows pushing of information from services to apps, after apps have registered themselves to get pushed data from particular services, the Service Runtime Environment needs to be able to support active, stateful services, instead of the usually applied stateless services.

6.1.2 Subcomponents

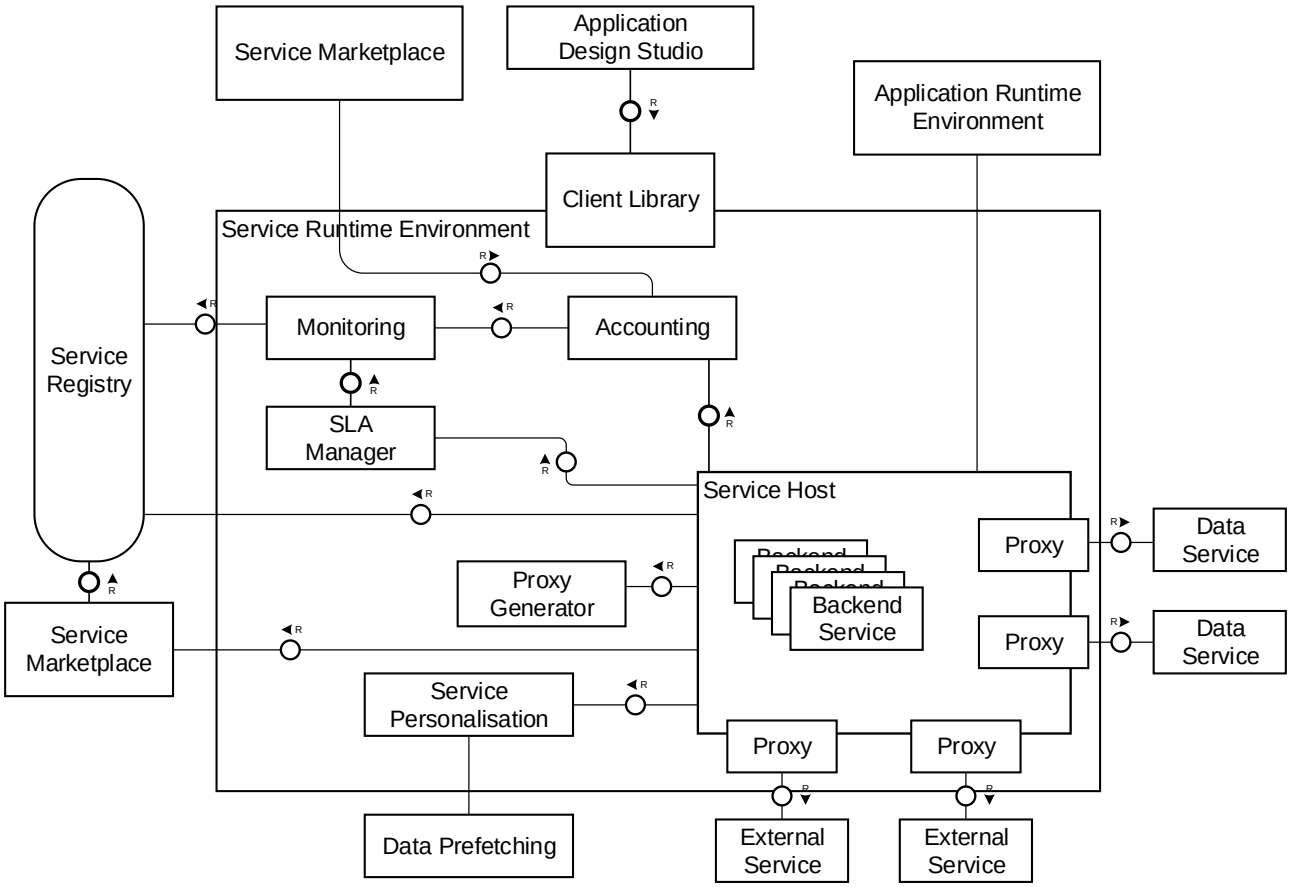

Figure 35: Service Runtime Environment Subcomponents and Interactions

To achieve the functionalities described in the previous subsection, the Service Runtime Environment provides subcomponents as depicted in Figure 45. Notably, this figure does *not* show the interaction with the subcomponents of, e.g., the Service Monitoring component or the Service Registry, and also only the most important interactions between Service Monitoring, Service Registry, Service Personalisation and components not part of the Service Runtime Environment.

Subcomponents of the Service Runtime Environment:

- Service Host: The Service Host provides the basic service execution environment, i.e., a platform able to install, host, start, stop, and uninstall services. In SIMPLI-CITY, the Service Host will be able to make use of computing resources in a scalable way, i.e., resources are leased and released based on the current and future workload.
- SLA Manager: Allows to make use of the (service) Monitoring component for observation of non-functional service aspects and to start according countermeasures, if necessary.
- Accounting: Allows counting the number of service invocations and therefore providing according accounting information to service providers.
- Accounting Database (not depicted): Data storage for accounting-related information.

- Proxy Support: Allows the automatic generation of monitoring proxies as well as the execution of these proxies. Monitoring proxies make use of the Monitoring component (not depicted in the figure).
- Client Library: The Client Library is not an intrinsic subcomponent of the Service Runtime Environment, but essentially a helper component, offering functionalities of the Service Runtime Environment to the app developer (via the Application Design Studio).
- Monitoring: See Section 6.2.
- Context-based Service Personalisation: See Section 6.3.

6.1.3 Related Requirements

Table 10: Requirements Related to the Service Runtime Environment

Requirement	Handled by Subcomponent	Comment
Must Have Requirements		
U27: Proactive behaviour (P1) U195: Proactive user notifications (P1) U196: Prioritization of notifications (P2)	Service Host Context-based Service Personalisation	Must enable services to forward information to the end user app (via the Application Runtime Environment).
U106: Access to cloud services	Service Host Proxy Generator	Must allow (services) to access data stored in the Cloud-based Information Infrastructure.
U139: Provision of service statistics (P1) U182: Provision of statistics, e.g., usage, traffic	Accounting Accounting Database Proxy Generator	In combination with the Monitoring component, the Accounting component should be able to provide data about service usage.
U192: Service deployment (P1)	Service Host SLA Manager Proxy Generator	Must allow starting and stopping internal backend services. Must allow interacting with external services through proxies. (Note: this is the central functionality of the Service Runtime Environment).
U193: Exchange of information from apps to server (P1) U194: Exchange of information from server to apps (P1)	Service Host Proxy Generator	The Service Runtime Environment should allow the interaction between apps (running in the Application Runtime Environment) and (internal and external) backend services.

Requirement	Handled by Subcomponent	Comment
U207: Support suggestions for road/trip optimization if conditions change (P1) U209: Provision of real-time information about the current route (P1) U212: Notification to end users about the proximity of Points of Interest (P2)	Service Host	Must support to *push* relevant data items to apps via the Push Service (part of the Application Runtime Environment).
Should Have Requirements		
U86: Transparency (P1) U87: Confidentiality (P1) U88: Data encryption (P2) U89: Certification (P2)	Service Host Proxy Generator	The Service Runtime Environment needs to make sure that data privacy is met.
U90: Availability (P3) U91: Integrity (P3) U92: Secure access to system (P3) U93: Third party access to the system (P3)	Proxy Generator Service Host	The Service Runtime Environment should allow making use of secure channels for communication with the Application Runtime Environment.
U103: Fault tolerance (P2)	SLA Manager Service Host	The SLA Manager should be able to react if a service is faulty. It controls corresponding actions by the Service Host (e.g., reinstantiation of a service).
U104: Stability (P2)	All components	All components need to take into account general policies and best practices to achieve the stability of the system.
U125: Open interfaces (P2) U126: Openness of the system (P2) U127: Extensibility (P2)	All components	It should be possible to extend any component in order to add more functionality.

Requirement	Handled by Subcomponent	Comment
U188: Composition of services (P3)	Service Host Proxy Generator	The Service Runtime Environment should allow invoking (data) services from within other services. While service developers may choose to implicitly mashup or compose services, the composition of services into value-added services will not be explicitly supported.
Could Have Requirements		
U124: Scalability of the service platform (P4)	Service Host All components	The Service Host must be able to lease and release (virtualized) computing resources in order to meet changing resource demands by the hosted backend services. The same requirement applies to all other components, if they could become a bottleneck.
Will not have for now		
U167: Service hot updates (P4)	Service Host	The Service Runtime Environment should cater for exchanging/updating services on the fly, i.e., during system runtime.

6.1.4 Interaction with other Components

6.1.4.1 Interaction with Application Runtime Environment, Service Registry and Backend Services

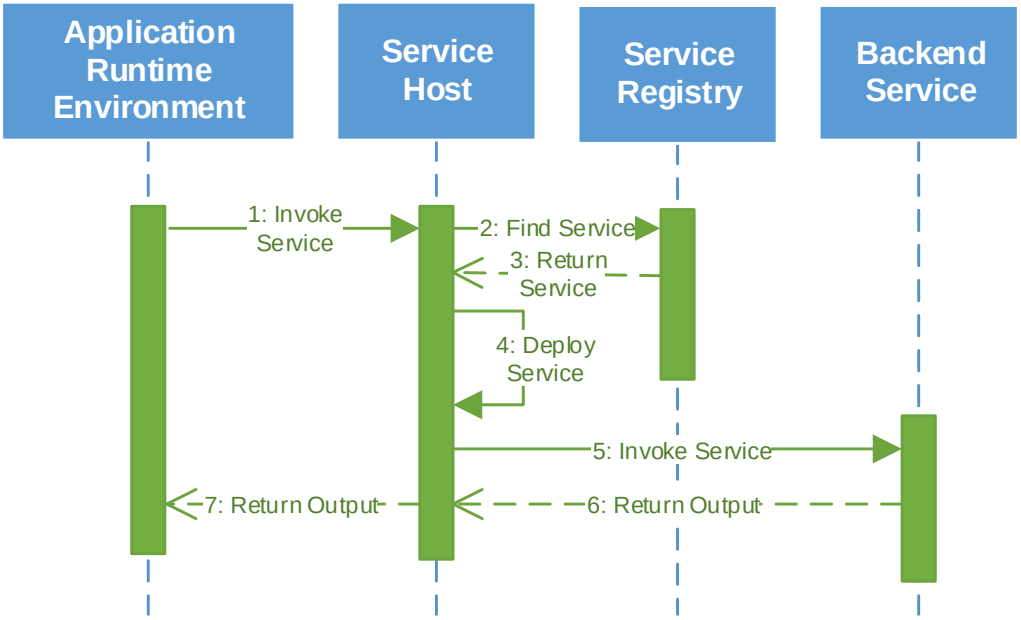

Figure 36: Interaction of Service Runtime Environment, Application Runtime Environment, and Backend Services with Deployment

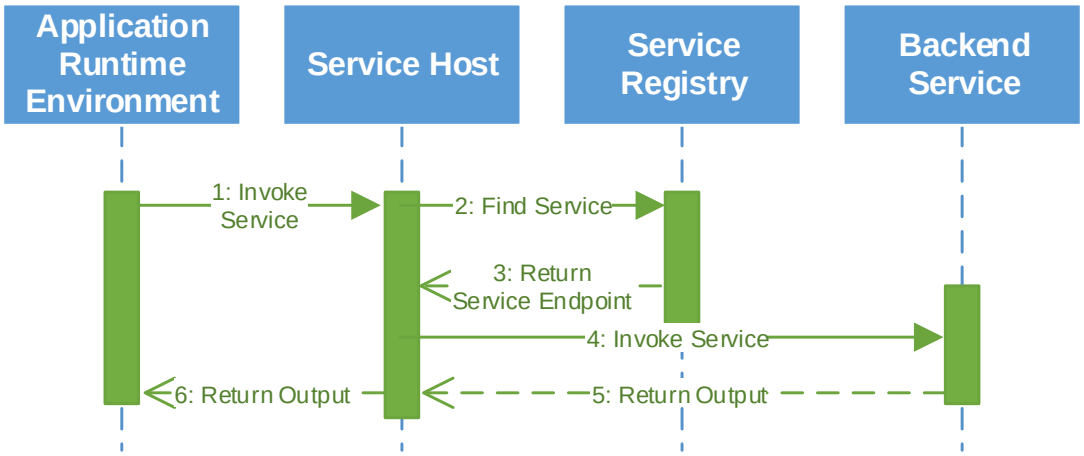

Figure 37: Interaction of Service Runtime Environment, Application Runtime Environment, and Backend Services without Deployment

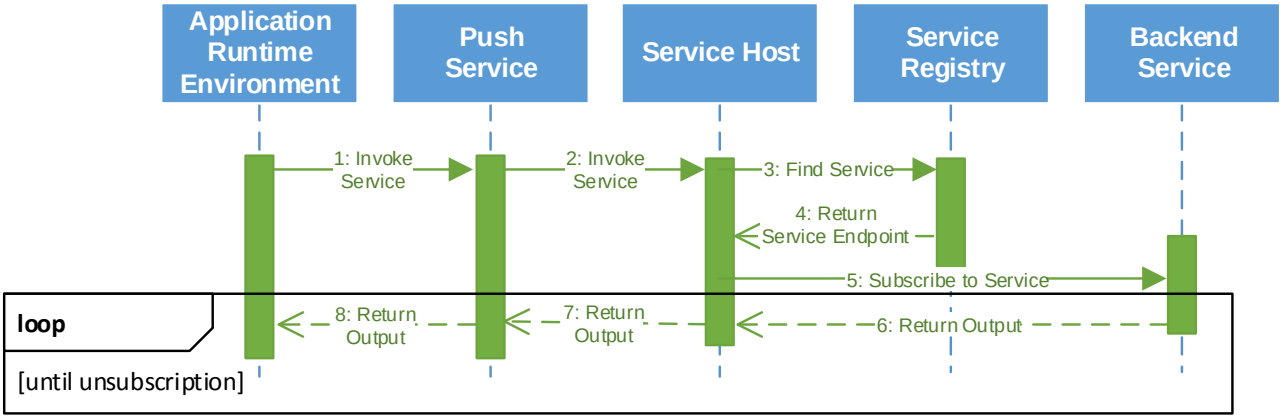

Figure 38: Interaction of Service Runtime Environment, Application Runtime Environment, and Backend Services for Push Services

Figure 36 shows the interaction of the Service Runtime Environment and the Application Runtime Environment and also the interaction with internal backend services. In this example, the service has not been deployed yet. Figure 37 shows the interaction without deployment, i.e., a backend service instance is already running. The necessary interaction from the perspective of the Service Registry is described in Section 6.4.4.2.

So far, internal backend services, i.e., services deployed within the Service Host, have been discussed. However, SIMPLI-CITY also allows the invocation of external backend services, which are not deployed within the Service Host. External backend services have to be registered in the Service Registry, using functionalities provided by the Service Development API. It is *not* possible to deploy an external backend service within the Service Host, as the corresponding service artefact is not available. During the invocation of an external backend service, a proxy needs to be used for monitoring and other purposes. Figure 39 shows the interaction between the components for external services.

Figure 36 and Figure 37 show the interaction between the Application Runtime Environment and the Service Runtime Environment if apps *pull* data from services. In the case of *pushing*, an invocation of a backend service (Step 5 in Figure 36 and Step 4 in Figure 37) would include a *subscription* of a particular service. This service remains active after it has been invoked and pushes information to the Application Runtime Environment instead of simply answering requests. This is depicted in Figure 38 (based on Figure 37). As it can be seen, in the case of pushing, the Application Runtime Environment does not directly interact with the Service Host. Instead, service invocation is done via the Push Service (actually part of the Application Runtime Environment), which takes care of routing push messages back to the Application Runtime Environment and also manages a list of subscribed apps. The Push Service forwards service requests to the Service Host, which then looks up the particular service and subscribes to it. For this, the Service Host has to provide the functionality as a publish/subscribe bus. To allow Backend Services to push data to the Service Host, they need to implement an according abstract class (see Section 8.2.4.4). Apps will be provided with pushed messages until they unsubscribe themselves from a particular Backend Service again via the Push Service and the Service Host (not depicted in the figure).

In general, the interactions discussed in this subsection are very basic, i.e., monitoring, accounting, or service personalisation functionalities have been omitted in order to not overcrowd the depictions. Further information about these interactions will be provided in the next subsections.

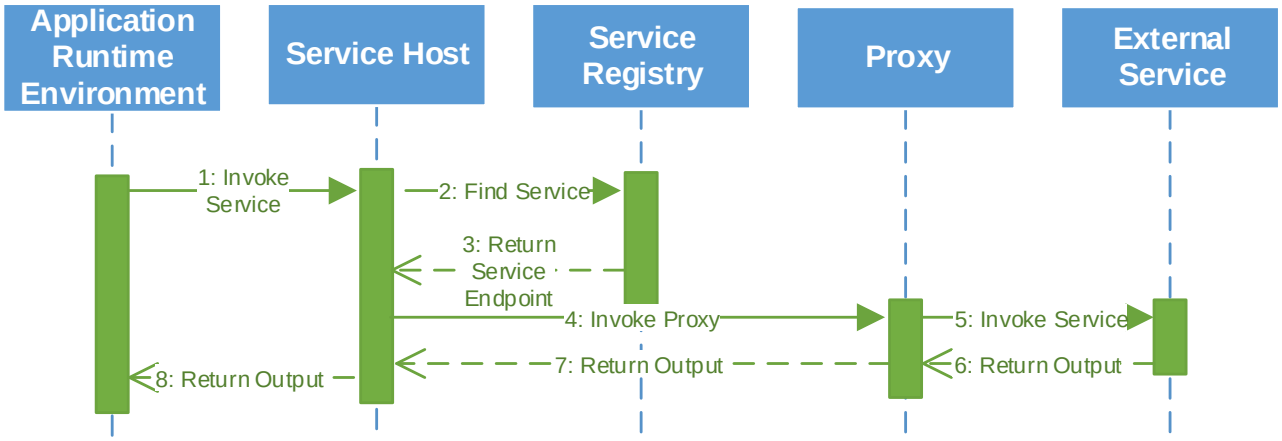

Figure 39: Interaction of Service Runtime Environment, Application Runtime Environment, and External Services

6.1.4.2 Interaction with Data Services

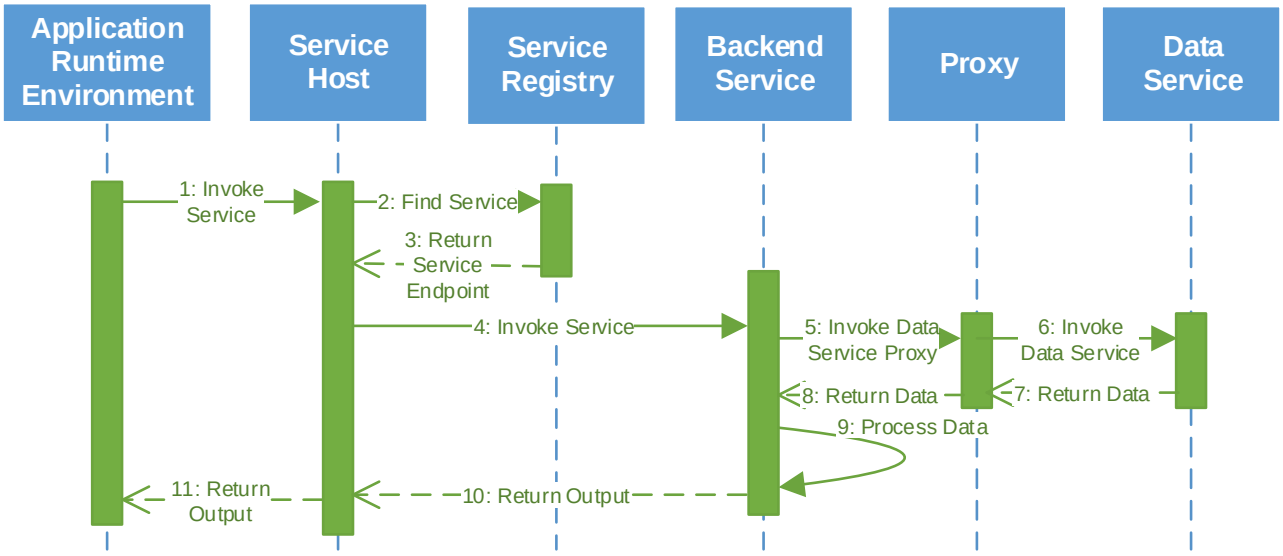

Figure 40: Interaction of Service Runtime Environment and Data Sources

Figure 40 is an extension of Figure 37, and includes also the usage of a Data Source (Data Service) by an *internal* backend service. Notably, an *external* backend service is not allowed to make use of these proxy functionalities of the Service Runtime Environment. Hence, if an external backend service makes use of a Data Service, this happens in a transparent way, i.e., the Service Host will not know that there is any Data Service involved in the background.

As it can be seen, Data Services are requested through a proxy, which provides an active layer to per se passive Data Services. The definition of a proxy is part of the service registration of Data Services and therefore part of the Service Development API (see Section 8.2). In SIMPLI-CITY; a Data Service acts as a wrapper for an arbitrary Data Source. It provides therefore an additional layer that is, e.g., for sensors based upon the Service Abstraction Interfaces (see Section 5.3), which are providing the actual technical integration of data sensors.

Note: Whenever a data service or backend service is invoked for the first time, a proxy needs to be generated for it. For this, the Service Host will call the Proxy Generator.

A Data Service could be a sensor, or an open government data repository. Furthermore, a proxy could directly interact with the Cloud-based Information Infrastructure in order to request data. A Data Service could also wrap functionalities from the Data Processing component through the Service Request Handler, as described in Section 5.1. In general, Data Sources do not have to make use of a particular technology to be applicable in SIMPLI-CITY, but only certain protocols and technologies will be directly supported and implemented during the course of work package WP4. For all other protocols and technologies, service developers have to integrate the according technologies by themselves as part of the registration of Data Service.

Importantly, a proxy for a Data Service can be either hard- or loosely-coupled. In the first case, the Backend Service will know the unique identifier of the proxy and therefore directly access it. If loose coupling is applied, the Backend has to lookup the proxy in the Service Registry. This happens analogue to the service lookup as depicted in Steps 2 and 3 of Figure 40.

A proxy could also allow subscribing to a particular Data Source so that it “pushes” data to the requester. This is realized through the usage of Context Sensors (via the Context Manager) which regularly check if there has been a relevant context (here: output) change (see Section 6.3). If this is the case, Steps 4-11 would be part of a (non-depicted) loop, which would provide data updates until an app (through the Application Runtime Environment) signals that it does not want to get pushed data anymore.

6.1.4.3 Interaction with Service Marketplace

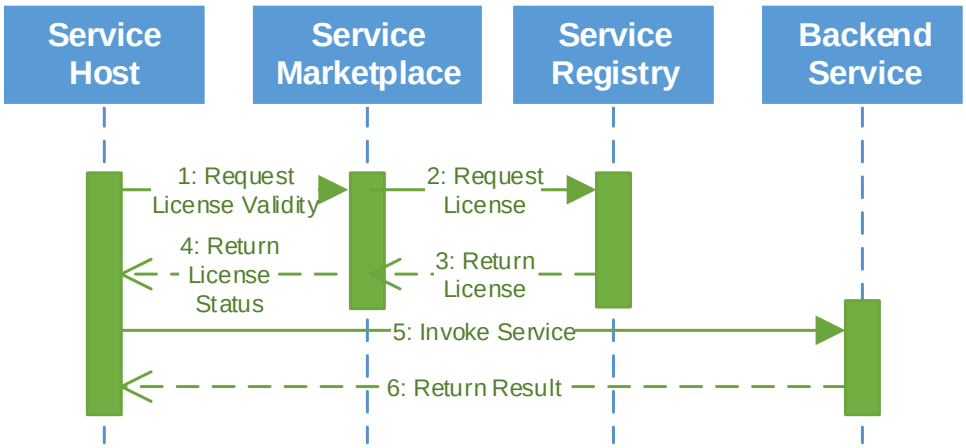

Figure 41: Interaction with Service Marketplace

Before an already deployed service may be invoked, it should be checked if the service user in question has a license to make use of this particular service. This needs to be done for both internal and external backend services as well as for data services. In order to check for licensing information, the Service Runtime Environment interacts with the Service Marketplace, which manages and validates licensing information. As can be seen in Figure 41, the licenses themselves are stored in the Service Registry, but the Service Marketplace takes care of validating the license information.

Note: This interaction could be part of all interactions described in Section 6.1.4.1, but has been omitted there in order to keep the figures well-arranged.

In addition, the Service Marketplace is the user interface for accounting-related data, i.e., this data will be displayed to service providers via the marketplace (not depicted). For this, the Service Marketplace accesses accounting and monitoring data for particular services from the respective databases.

6.1.4.4 Interaction with Context-based Service Personalisation

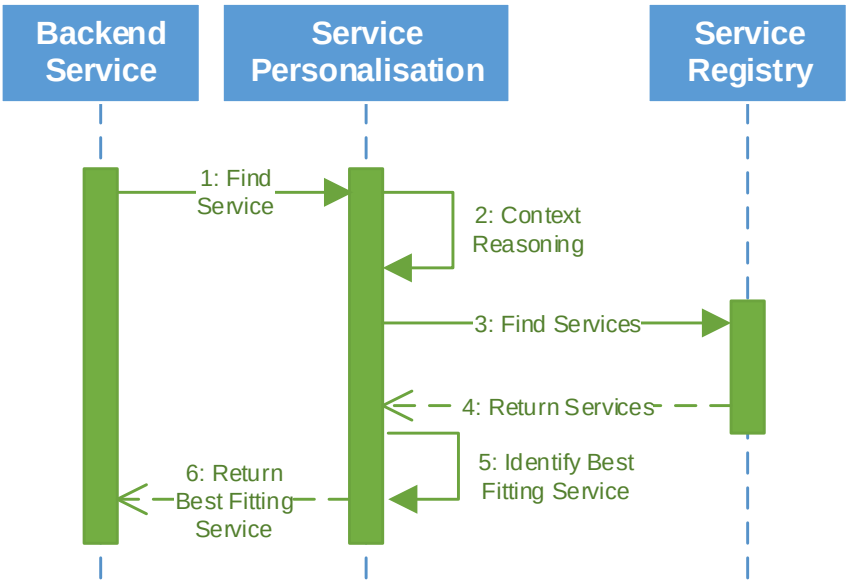

Figure 42: Interaction for Location-Based Data Service Selection

There are different interactions between the Service Runtime Environment and the Context-based Service Personalisation component. As can be seen in Figure 35, the latter is actually a subcomponent of the former environment. Hence, all functionalities of the Context-based Service Personalisation are exposed through the Service Runtime Environment. As described in more detail in Section 6.3.1, the Context-based Service Personalisation component offers the following functionalities:

- Location-based data service selection: As depicted in Figure 42, the selection of a particular data service based on the location of the end user is triggered by the backend service providing a particular functionality. However, the selection logic is provided by the Context-based. Service Personalisation component. Note: The Service Host (where the backend service is located) is not depicted for reasons of simplicity.
- Proactive user notifications: This is an API functionality that needs to be implemented by a service. Hence, there is no specific interaction between the Service Runtime Environment and the Service Personalisation component.
- Support of prefetching-relevant context: While the actual prefetching logic is part of Section 5.4, the Service Personalisation component provides context information to the Data Prefetching component, if requested.
- Context-based service execution: This is the generic service personalization mechanism in SIMPLI-CITY. Hence, the specific interactions depend on the actual functionality. Figure 43 shows this generic interaction. Once again, the Service Host is not depicted for reasons of simplicity.
- Detection of context changes: As mentioned above, the Context-based Service Personalisation component allows to sign up for context/data changes of a

particular Data Source. To make use of this functionality, a subscribe/publish functionality is provided through the Context Manager (see Section 6.3).

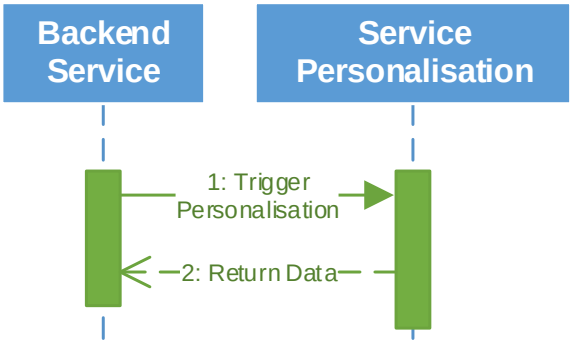

Figure 43: Context-based Service Execution (Generic Interaction)

6.1.4.5 Interaction with Monitoring and Accounting

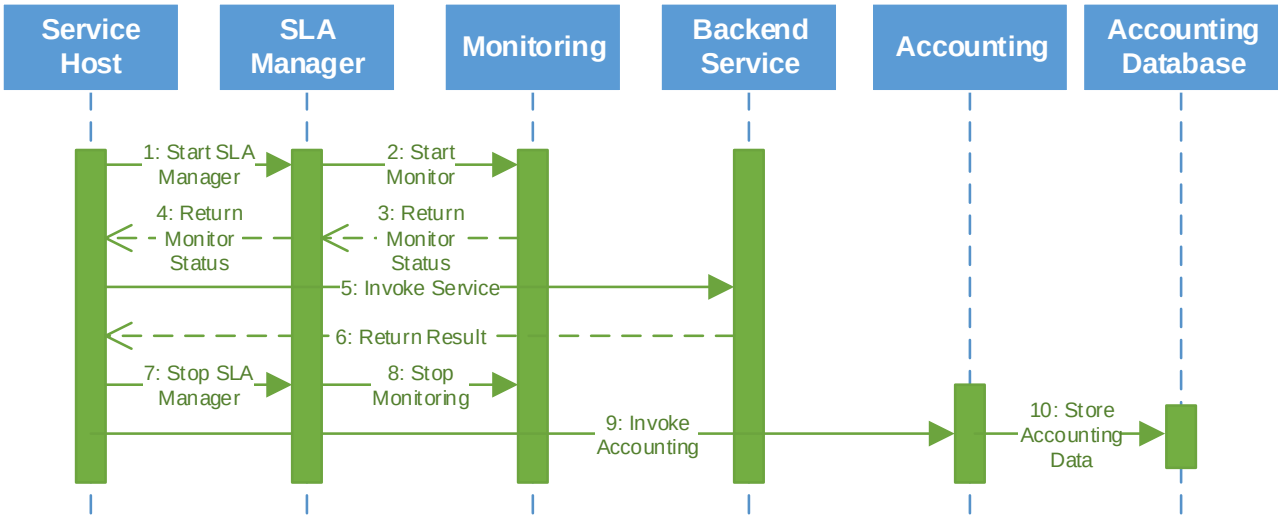

Figure 44: Interaction with Monitoring and Accounting

As discussed above, so far, the interaction between the Service Host and backend services has been simplified and supporting components like the Monitoring and Accounting components have not been covered. Figure 44 shows the interaction between the Service Host and these components. Notably, accounting will only be conducted once a service has been successfully invoked. Accounting data is stored in an according database and might be requested by the Service Marketplace as discussed in Section 6.5.4.5.

The monitoring is controlled through the SLA Manager, which is also the component controlling the adherence to necessary Service Level Agreement (SLA) parameters, and starts countermeasures if there is a deviation from the expected behaviour. According countermeasures (not depicted here) could be that a service needs to be reinvoked by the Service Host, another service instance needs to be instantiated, etc.

Note: The actual monitoring interaction is discussed in Section 6.2.

6.1.5 User Interface

The Service Runtime Environment will not feature a specific user interface. User interactions during service design time will be handled by the Service Development API and its APIs. Information from the Service Registry as well as from the Accounting subcomponent will be displayed in the Service Marketplace. Most importantly, there is no direct user interaction with the backend services. Instead, service functionalities are offered to the end user via apps.

6.1.6 Conceptual Data Model

The conceptual service data model is described in Section 9.3.

6.1.7 Parameters to Take into Account for Technical Specification

Table 11: Criteria for Technical Specification

Parameter	Importance (--, -, -+, +, ++)
Generic Criteria	
Up-to-Datedness	-+
Stability	++
Extensibility & Open Source/Standards	++
Familiarity	+
Performance	++
Interoperability	+
Specific Criteria	
Virtualization of Service Platform	+
Virtualization of Service Host	++
Support of Active, Stateful Services	++
Support of Pushing Functionality	+
Security Features	++
Fault Tolerance Mechanisms	+
Hot Service Deployment	++
Scalability	+

Parameter	Importance (--, -, -+, +, ++)
Monitoring Features	+
Included Service Registry	+

6.2 Monitoring

6.2.1 Overall Functional Specification

SIMPLI-CITY offers the functionalities for registering, discovering and invoking several services. Each service may come with an optional SLA consisting out of different Service Level Objectives (SLOs). SLOs target the Quality of Service (QoS) aspects like the maximum response time or the minimum amount of concurrent requests a service should be able to handle, i.e., if a service is not able to provide the defined QoS, there will be a breach of the SLA.

In order to ensure these SLAs, i.e., in order to detect potential SLA violations in advance and counteract if necessary, the Monitoring component is needed. Aside from this, the Monitoring component also provides the monitored data to the developers in order to provide feedback about usage and potential errors. Further, some general statistics about the services are provided to the end users.

This component runs in the Service Runtime Environment. The Monitoring component can either monitor a service on server-side if the service is provided by the SIMPLI-CITY developers and if it runs in the Service Runtime Environment, i.e., if it is an internal backend service. Alternatively, the component can monitor a service from a client-side perspective via a proxy or within the client on its own. This will be done for external services, i.e., services not running within the Service Runtime Environment.

6.2.2 Subcomponents

To achieve the functionalities described above, the Monitoring component provides the following subcomponents as depicted in Figure 45:

- Monitoring Manager: Allows configuring the actual monitoring and interprets SLAs.
- Monitor: Provides the actual monitoring of the following quantifiable aspects of a service:
  - Number of service invocations
  - QoS aspects, e.g., availability and response time
  - Resource consumption of the single services and service invocations (only for internal services)
  - Produced costs (only for internal services)
- Monitoring Proxy: Allows monitoring external services, i.e., services which are not deployed within the SIMPLI-CITY Runtime Environment.
- Monitoring Data Storage: Allows storing and retrieving monitored data.

Notably, it is yet to be decided if the monitoring of internal backend services can be done by observing services invoked within the Service Runtime Environment, or if the Monitor has to be used as an additional proxy if a service is invoked. This decision depends on the

monitoring tools for the Service Runtime Environment software to be chosen within the Technical Specification (deliverable D3.2.2).

The Monitoring component is used by:

- Reporting Component: The Monitoring Manager is able to provide the monitored data to several different Reporting Components, examples are (but not limited to)
  - Service Marketplace: To show information about a specific service to other service developers
  - Report Service: To provide up-to-date feedback in form of a report, e.g., an email, RSS, etc.
- Service Runtime Environment: To provide the monitoring functionality of internal services and to deliver information about the Service Runtime Environment on its own.

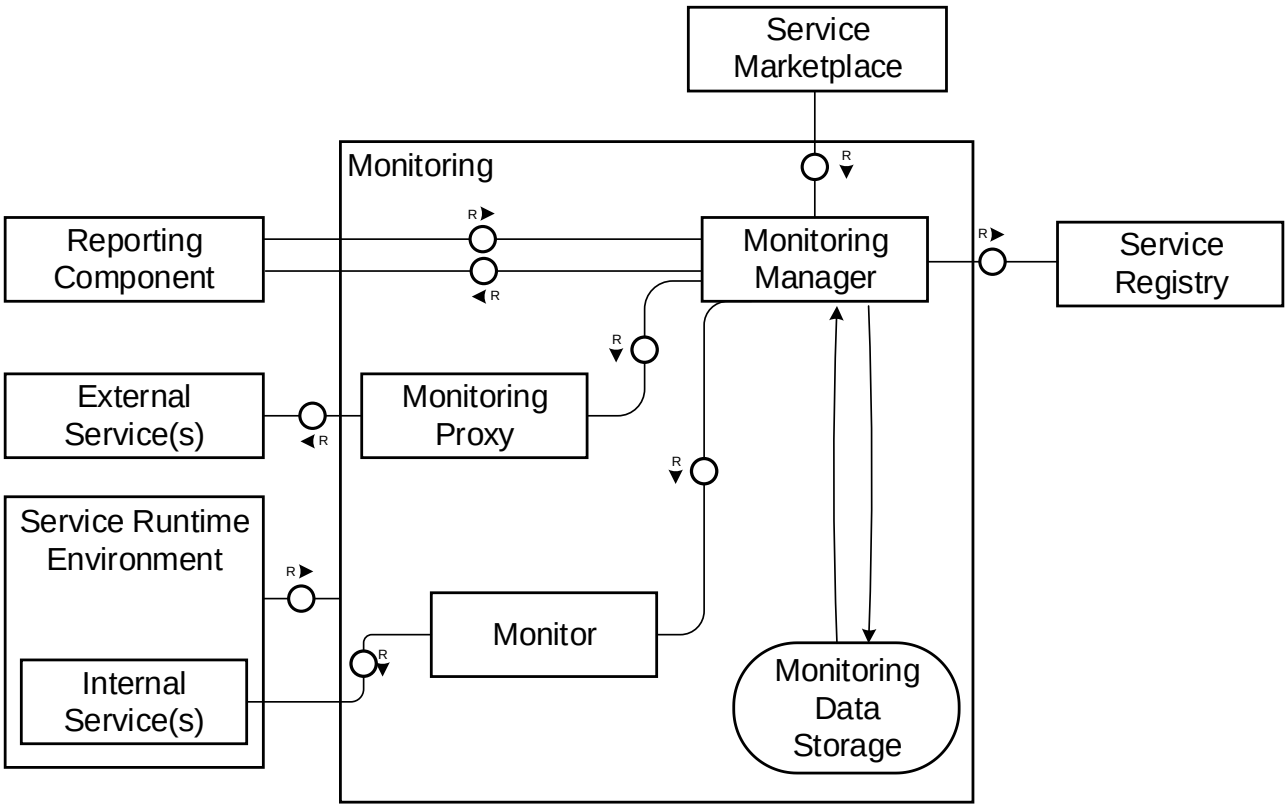

Figure 45: Monitoring Components

6.2.3 Related Requirements

Table 12: Requirements Related to the Monitoring Component

Requirement	Handled by Subcomponent	Comment
Must Have Requirements		
U103: Fault tolerance (P2) U104: Stability (P2)	Monitoring Manager Monitoring Proxy Monitor Monitoring Data Storage	It is crucial that every subcomponent of the Monitoring component provides a certain level of fault tolerance and stability. Furthermore, the Monitoring component provides information to other components which indicate if a service is available and provides the necessary QoS. Hence, the Monitoring component facilitates fault tolerance and stability within the Service Runtime Environment.
U152: SLA support (P1) U153: Usage of an official SLA standard (P4) U154: Simple SLA description standard (P2)	Monitoring Manager	It must be possible to support SLAs, thus the Monitoring Manager can be configured in terms of what kind of QoS attributes should be monitored.
Should Have Requirements		
U127: Extensibility (P2)	Monitor Monitor Proxy Monitoring Manager	It should be possible to extend the current monitoring components in order to be able to monitor additional attributes.
U124: Scalability of the service platform (P4)	Monitor Monitoring Proxy Monitoring Manager	It should be possible to acquire and release additional resources if the current ones are not enough in order to provide a stable monitoring for every service.

6.2.4 Interaction with other Components

6.2.4.1 Monitoring for Internal Backend Services

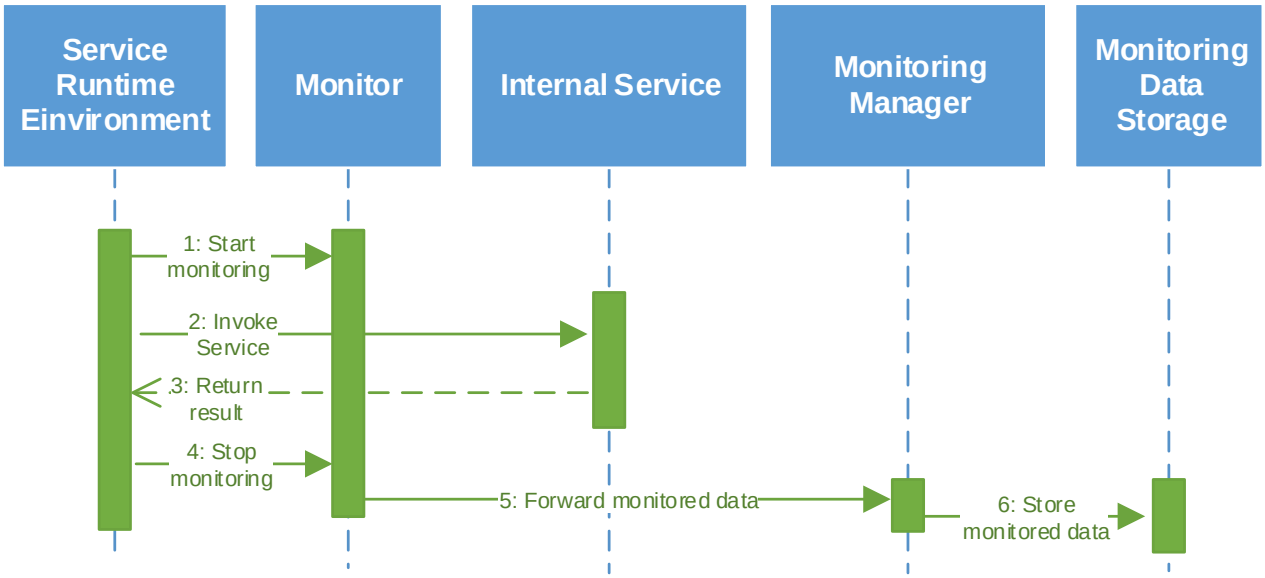

Figure 46: Interaction of Internal Services and Monitoring Components

Figure 46 shows the monitoring workflow for internal services. Whenever a call for a particular service arrives in the Service Runtime Environment, the Monitor subcomponent starts the actual monitoring. Each call is intercepted by the Monitor, thus information about different QoS attributes can be collected. After the invocation, the monitored data is forwarded to the Monitoring Manager which saves the data in the Monitoring Data Storage for later usage.

Notably, Figure 46 shows the usage of an “observing” monitor, i.e., a monitor which does not have to be invoked as a proxy (see next section). However, as described above, the actual decision if a proxy- or observer-based monitoring approach will be used within the Service Runtime Environment depends on the actual underlying software. Hence, this sequence could change within the Technical Specification (deliverable D3.2.2).

6.2.4.2 Monitoring for External Services

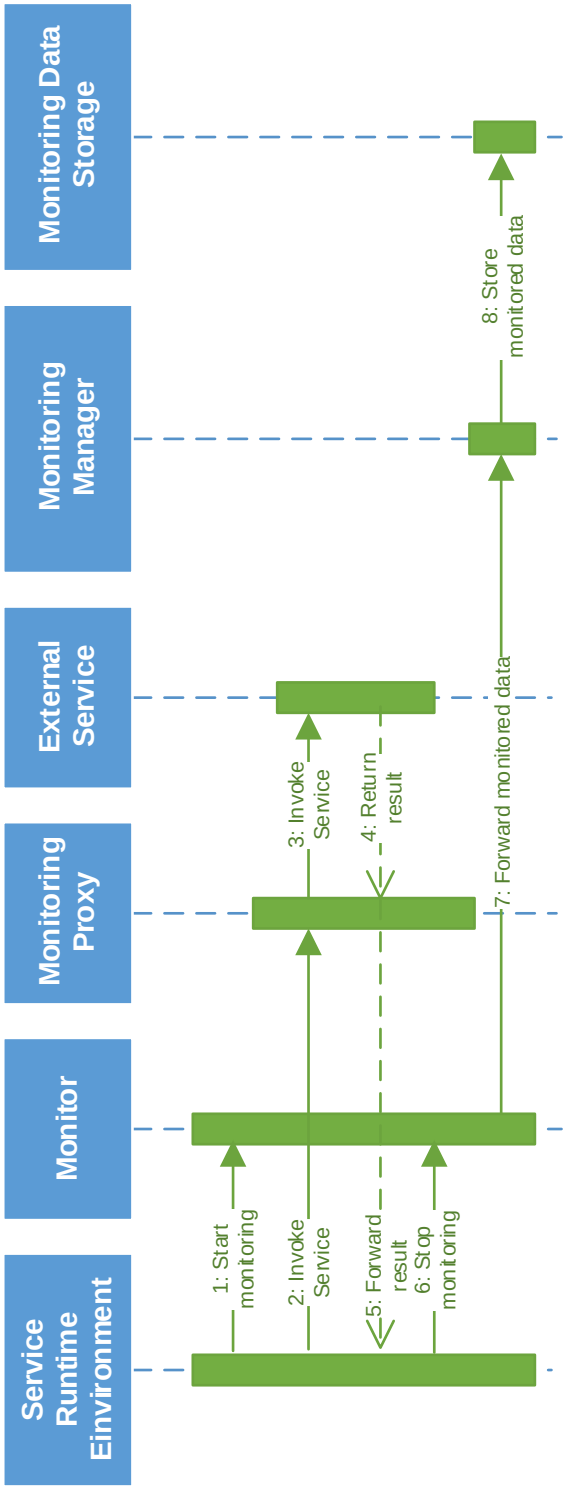

Figure 47: Interaction of External Services and Monitoring Components

Figure 47 shows the interaction of the Monitoring's subcomponents and external (backend or data) services. In contrast to internal services, external services are not under control of the Service Runtime Environment. Hence, the actual monitoring is still performed by the Monitor, but a Monitoring Proxy is needed which can intercept the service's invocations and monitors this call in terms of different QoS attributes. Every invocation of the external

service is piped through this proxy in order to monitor the calls. The Monitor monitors basically the proxy and implicitly the external service. The retrieved data is forwarded to the Monitoring Manager which stores it in the Monitoring Data Storage.

6.2.4.3 Reporting Components

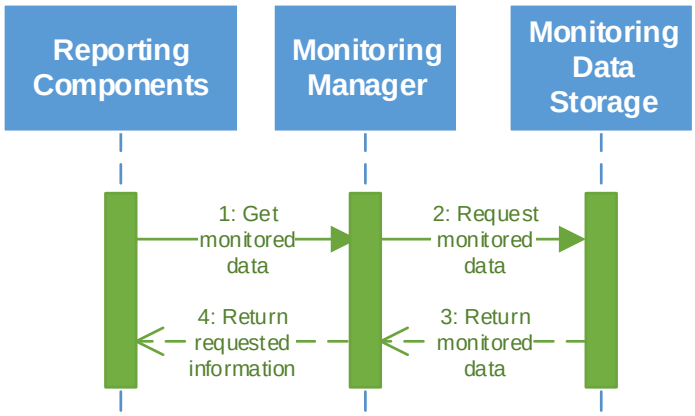

Figure 48: Interaction Reporting Components and the Monitoring Manager

Figure 48 shows the interaction of a Reporting Component (e.g., the Service Marketplace) with the Monitoring's subcomponents. A Reporting Component could be any SIMPLI-CITY component which is interested in the monitoring data, e.g., in order to show it as part of information in the Service Registry or Service Development API.

In order to provide monitored data to users and developers a Reporting Component requests the relevant data from the Monitoring Manager which reads the data from the Monitoring Data Storage and returns it to the requestor.

6.2.4.4 Interaction with the Service Registry

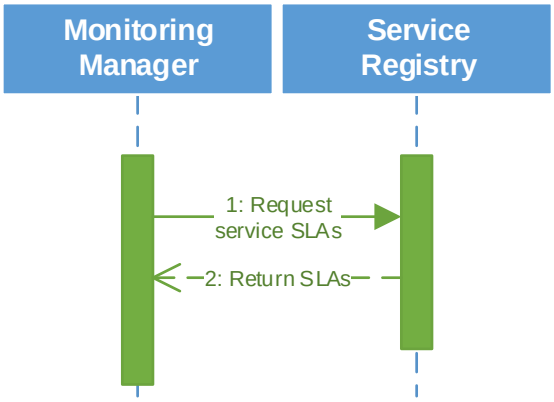

Figure 49: Interaction of Monitoring Manager with Service Registry

Figure 49 shows the interaction of the Monitoring Manager and the Service Registry in order to retrieve service-specific SLAs. This information can then be used in order to detect deviations from the expected/necessary non-functional behaviour of a service and subsequently to start according countermeasures.

6.2.5 User Interface

Figure 50 shows an example for a user interface mockup for the Monitoring component. It is crucial to see on a first sight whether the service is online (i.e., working as expected) or offline (i.e., if the service is not online anymore). Further, the UI mockup shows information about which service is monitored (here: ParkingLotFinder) about the overall invocations and the average invocations per hour. In addition, it shows the minimal and maximal measured response time, as well as the overall response time average. Beside of the textual representation, the UI should provide a real-time chart, showing the actual amount of concurrent invocations and QoS attributes such as the response time.

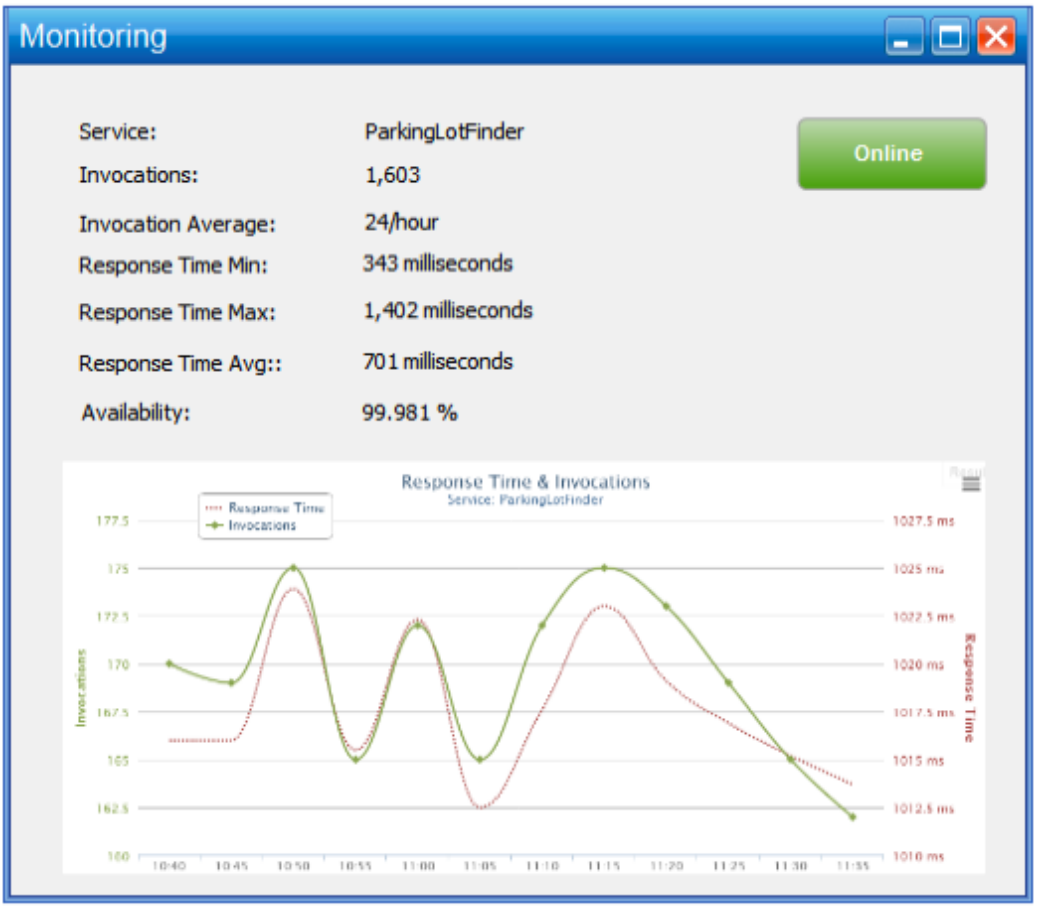

Figure 50: Monitoring UI Mockup

6.2.6 Conceptual Data Model

As described above, the Monitor components need to store the monitored data in a database for later retrieval. For this, the following data items will be stored (see Section 9.3):

- ServiceID: The service's identification
- ResponseTime: The response time of a particular service invocation
- Invocations: The number of invocations
- Throughput: The average throughput in a particular timespan

6.2.7 Parameters to Take into Account for Technical Specification

Table 13: Criteria for Technical Specification

Parameter	Importance (--, -, -+, +, ++)
Generic Criteria	
Up-to-Datedness	++
Stability	++
Extensibility & Open Source/Standards	+
Familiarity	+
Performance	++
Interoperability	+
Specific Criteria	
GUI	-+
SLA Support	++
Various Possibilities Regarding Database	-+
Role Management	-+
Extensible Data Model	++

6.3 Context-Based Service Personalisation

6.3.1 Overall Functional Specification

As the name implies, the Context-based Service Personalisation component offers the base functionalities for personalizing a service output based on the context of a particular user. Context-based service personalisation follows different approaches:

- Location-based data service selection: This allows adapting a service outcome based on the location of a server, i.e., the service outcome will be location-aware.
- Proactive user notifications: This allows recognizing in which situations a user may be provided with a certain piece of information. For this, it is necessary to determine the value of the information.
- Support of prefetching-relevant context: This allows controlling the data prefetching functionalities as discussed in Section 5.4. Also, provides relevant context data to the Media Data Streams and Data Prefetching Logic component.
- Context-based service execution: This allows the automated usage of user context data in service executions.

- Context changes: Allows to subscribe to a particular data source and be informed when a (predefined) change occurs.

6.3.2 Subcomponents

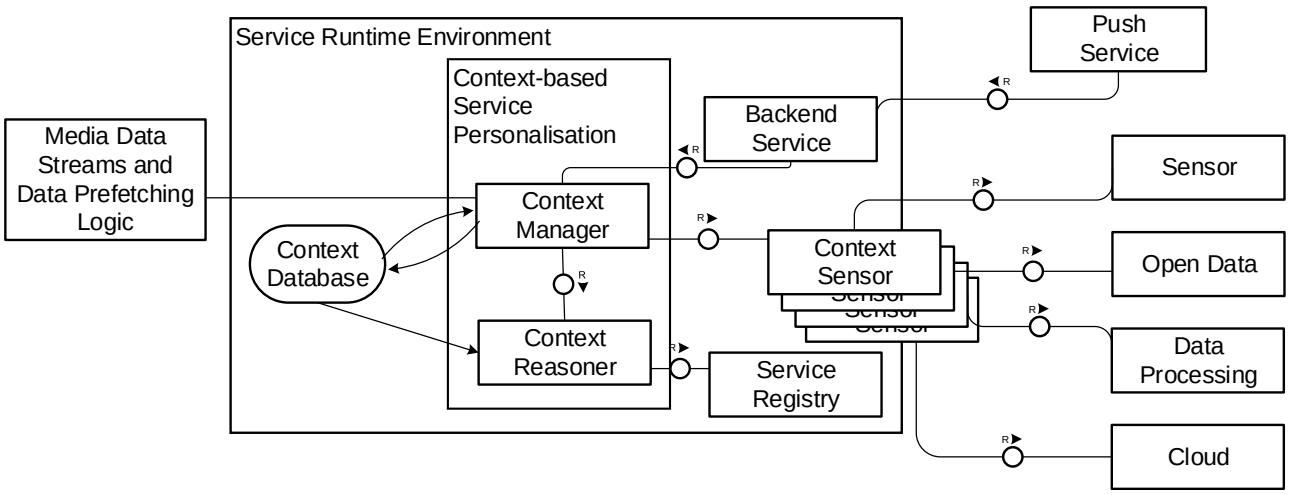

Figure 51: Context-based Service Personalization Components and Interactions

To achieve the functionalities described in the previous subsection, the Context-based Service Personalisation provide the following subcomponents as depicted in Figure 51. As it can be seen, the component is actually a subcomponent of the Service Runtime Environment.

- Context Manager: Receives requests from internal services, i.e., running inside of the Service Runtime Environment. The Context Manager manages the relevant context data and returns the information to the requesting service. For this, it offers a publish/subscribe functionality, i.e., an external component, e.g., a proxy or backend service, can subscribe to a particular context data source and will be informed about context changes until the component unsubscribes itself again.
- Context Reasoner: Provides the logic for different use case scenarios where context-based service personalisation is envisioned.
- Context Database: In this database, up-to-date context relevant information is stored
- Context Sensors: Context Sensors provide the Context Manager and further the Context Reasoner with new data.

The Context-based Service Personalisation is used by:

- Internal backend services: To accomplish the functionalities described above.
- Media Data Streams and Data Prefetching Logic: To get information about whether data prefetching should be performed or not. Also, in order to trigger data prefetching.
- Push Service (part of the Application Runtime Environment): The Context-based Service Personalization component is indirectly used by the Push Service, i.e., through a backend service, in order to proactively notify a user. Notably, backend services could also make use of the Context-based Service Personalisation component without being invoked by the Push Service.

6.3.3 Related Requirements

Table 14: Requirements Related to the Context-Based Service Personalisation

Requirement	Handled by Subcomponent	Comment
Must Have Requirements		
U27: Proactive behaviour (P1) U195: Proactive user notifications (P1)	Context Manager Context Reasoner Context Database Context Sensor	This is one of the essential functionalities of the Context-based Service Personalisation component. It must provide the means to trigger actions/services/notifications in advance. It must also provide the reasoning functionality, i.e., to detect whether a proactive action is acquired or not. Further, Context Sensors are either requested explicitly in order to retrieve new context data, or the data is pushed from the sensors towards the Context Manager.
U59: User centric data services (P1)	Context Manager Context Reasoner Context Database	It should be possible to include user-related information in the context reasoning for user personalized services
U90: Availability (P3) U91: Integrity (P3) U92: Secure access to system (P3) U93: Third party access to the system (P3) U103: Fault tolerance (P2) U104: Stability (P2)	Context Manager Context Reasoner Context Database Context Sensor	It is crucial that every component of this component provides a certain level of fault tolerance and stability. Further, the Context Manager should also be accessible for third party developers and provide a high availability and ensure a secure access and data integrity.
U106: Access to cloud services (P1)	Context Manager	It should be possible to read data from an arbitrary (data) service in order to reason about it.
U201: Provision of personalized info, e.g., travel, costs (P1)	Context Manager Context Reasoner Context Database	The Context Reasoner should be able to recognize which information is relevant to the user depending on the user's current context.

Requirement	Handled by Subcomponent	Comment
Should Have Requirements		
U125: Open interfaces (P2) U126: Openness of the system (P2) U127: Extensibility (P2)	Context Manager Context Reasoner Context Database	The Context Manager should provide an open interface, thus other software components can make use of its functionalities such as the Context Reasoner. Further, it should be possible to extend the Context-based Service Personalisation component by the means of using different attributes in a different context
U196: Prioritization of notifications (P2)	Context Manager Context Sensors	It should be possible to prioritize notifications depending on the user's context.
Could Have Requirements		
U124: Scalability of the service platform (P4)	Context Manager Context Reasoner Context Database Context Sensor	It should be possible to scale the system when the available resources are not sufficient to provide a certain level of efficiency.
Will not have for now		
U28: The system learns from feedback (P3)	Context Reasoner	It should be possible to learn from past interaction between the user and apps.

6.3.4 Interaction with other Components

6.3.4.1 Request of Context-relevant Data by Services

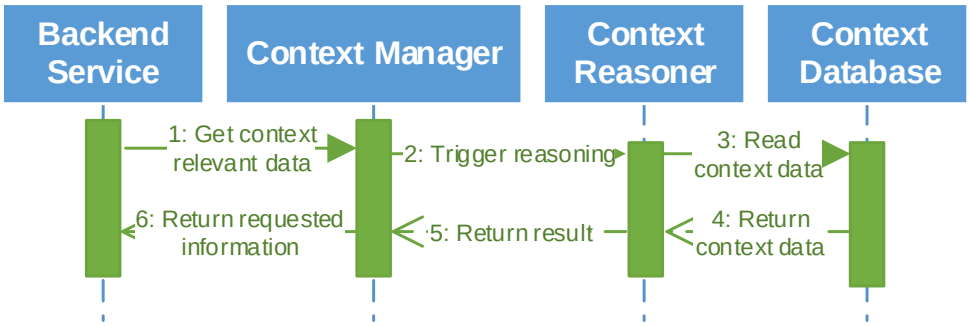

Figure 52: Interaction of Backend Services and Context-based Service Personalization

Figure 52 shows the interaction between backend services (running inside the Service Runtime Environment) and the Context-based Service Personalization subcomponents.

For simplicity reason the interaction with the Service Runtime Environment is not shown (see Section 6.1.4.4). Whenever a backend service requests a context-specific action, it sends a request to the Context Manager. The latter one triggers the Context Reasoner which retrieves the required information about the current context from a database. The result of the actual reasoning (not depicted) is passed to the Context Manager and further to the requesting backend service.

6.3.4.2 Context Data Access through Context Manager

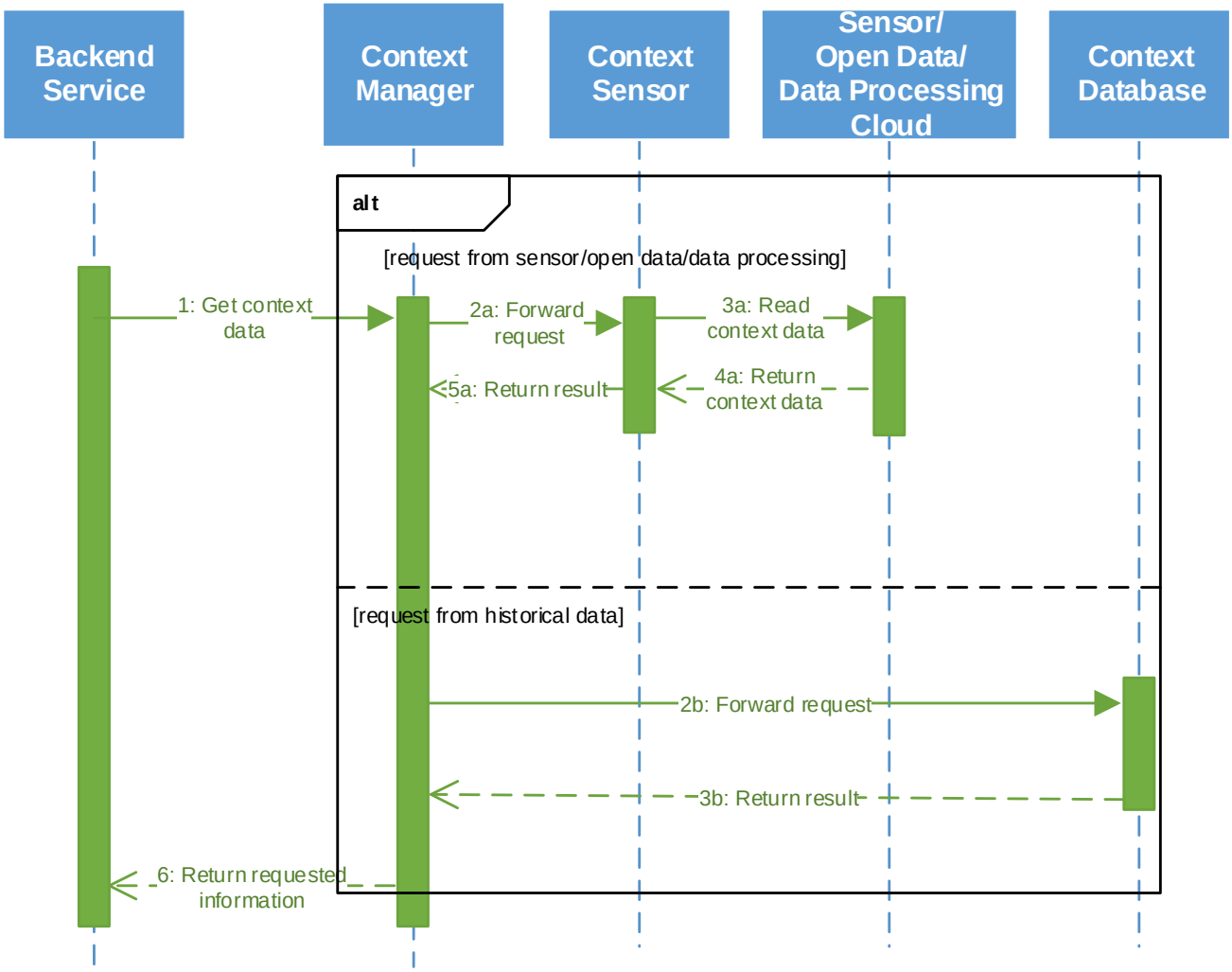

Figure 53: Unified access to Data Sources through the Context Manager

Figure 53 shows the unified access to arbitrary data sources through the Context Manager. Whenever a backend service needs context data, it sends a request to the unified interface of the Context Manager which decides whether the data should be read from the Context Database or if the request should be forwarded through the Context Sensors in order to retrieve data from sensors, open data, data processing or data from the Cloud such as a user's profile. As it can be seen, the Context Sensor can be applied for arbitrary types of data sources, e.g., sensor data, open data, or data from the Data Processing component.

Figure 53 only shows the pulling of data from a Context Sensor, i.e., the data is requested exactly ones by a backend service. Alternatively, the backend service could also subscribe to a Context Sensor which then would regularly push data to the service (see Figure 54).

The according publish/subscribe functionality will be part of the Context Manager. Whenever the Context Sensor determines that there is new data from a data service (not depicted in Figure 54) it is monitoring, it pushes this information back to the Context Manager and (via the publish/subscribe functionality of the Context Manager) to the backend service. The Context Manager also forwards this information to the Context Reasoner, thus, it can reason on it and stores it in the Context Database. The backend service could then in turn push the data to an app via the Push Service of the Application Runtime Environment (not depicted in the figure).

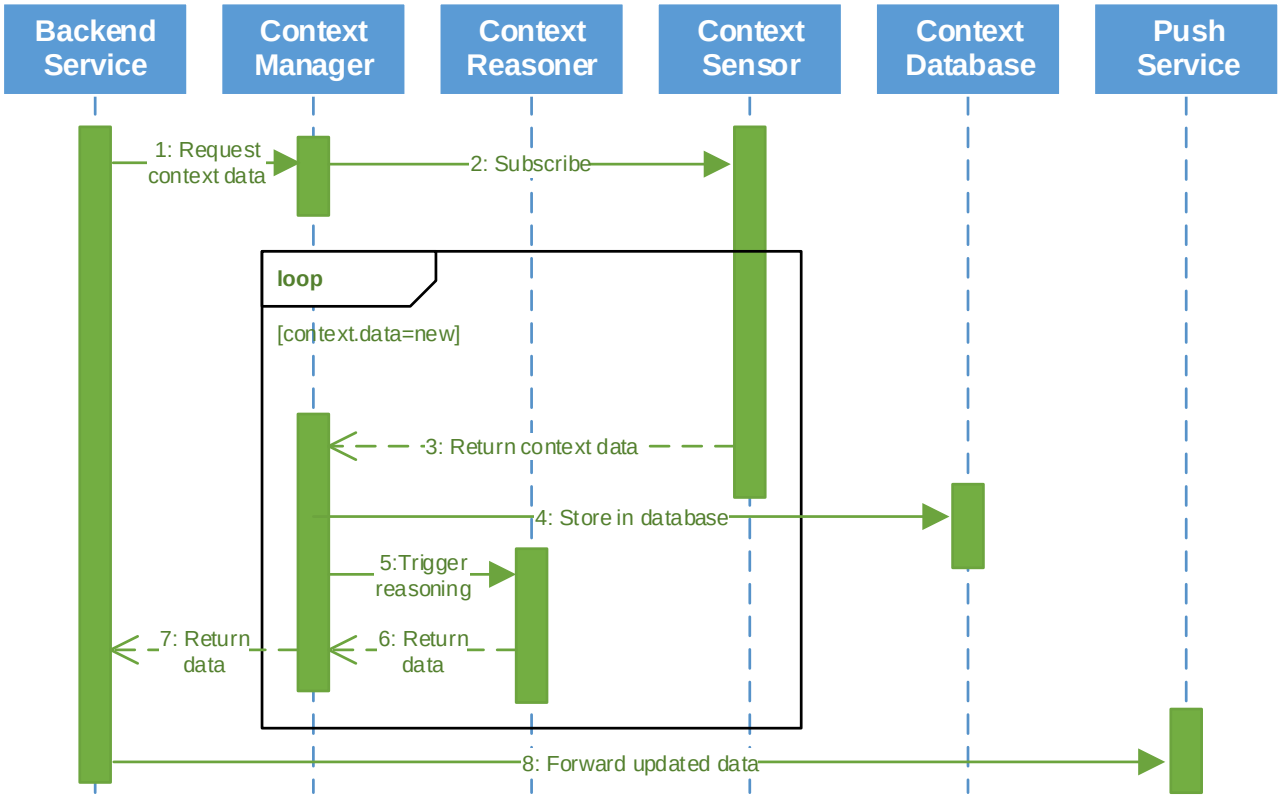

Figure 54: Interaction between Context Sensors and User Notification

6.3.4.3 Proactive Context-based User Notification

Figure 54 also shows the interaction of the Context-based Service Personalisation subcomponents in order to proactively notify a user through the Application Runtime Environment's Push Service. A particular backend service registers itself at the Context Manager which further subscribes itself at the required Context Sensor. Whenever the Context Sensor pushes new information to the Context Manager, it firstly stores the data in a Context Database and secondly triggers the Context Reasoner in order to reason on the updated information. The result is returned to the Context Manager and further to the sending Backend Service, which forwards this update to the Push Service in order to notify a particular user.

6.3.4.4 Context-based (Data) Service Selection

Figure 55 shows the interactions which are performed in a context (e.g., location)-based service selection scenario; the same interaction from the perspective of the Service Runtime Environment has been presented in Section 6.1.4.4.

As it can be seen, a backend service requests the Context Manager in order to find the best fitting service for a particular context (e.g., location). This request contains the information about the current context, e.g., the current GPS position. The Context Manager forwards these requests to the Context Reasoner in order to reason about it. The Context Reasoner looks up for the best fitting service within the Service Registry and returns this information to the Context Manager which forwards the result to the requesting backend service.

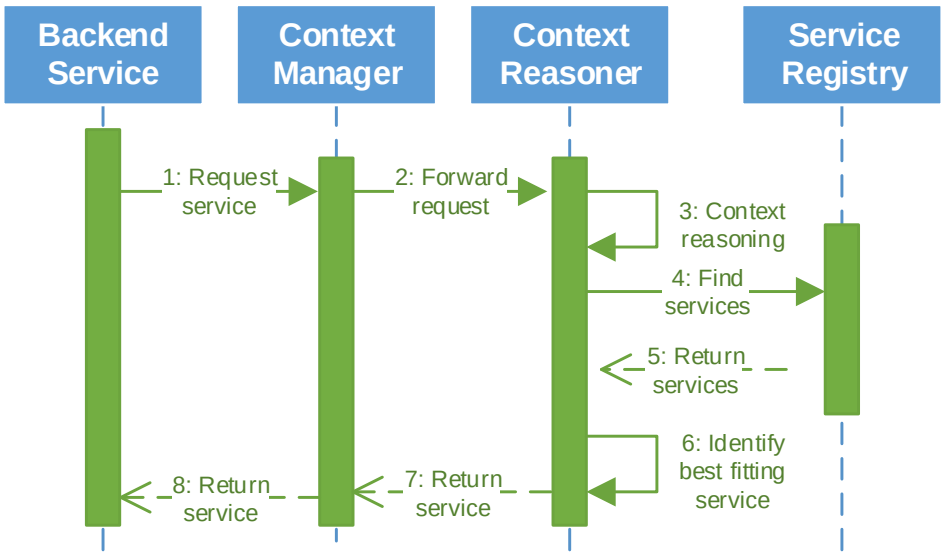

Figure 55: Interaction for Location-Based Data Service Selection

6.3.5 Conceptual Data Model

As depicted in the figures and in the description above, the Context-based Service Personalisation components need to store context-relevant data in a database. In order to provide useful information to the Context Reasoner and other components, the stored data should provide information about the exact date when the data was gathered by a sensor, the type of the data and its value. Other fields of this database are:

- SourceID: Defines the source which produced the entry, e.g., a car's GPS sensor.
- UserID: Defines the user to which the record belongs.
- ContextType: Refers to the category of context such as temperature, time, location, etc.
- ContextValue: Defines the current raw value gathered by a sensor. The unit depends on the context type, e.g., longitude/latitude, Celsius, etc.
- PastValue: Defines the value for the latest known entry.
- TimeStamp: Defines the date and time when this entry was recorded.
- Confidence: Since not every sensor provides accurate data, this field describes the uncertainty of this entry.

6.3.6 Parameters to Take into Account for Technical Specification

Table 15: Criteria for Technical Specification

Parameter	Importance (--, -, -+, +, ++)
Generic Criteria	
Up-to-Datedness	++
Stability	-+
Extensibility & Open Source/Standards	+
Familiarity	+
Performance	++
Interoperability	+
Specific Criteria	
GUI	--
Various Possibilities Regarding File Storage	-+
Various Possibilities Regarding Database	-+
Role Management	++
Extensible Data Model	++

6.4 Service Registry

6.4.1 Overall Functional Specification

(Backend) Services are at the core of all functionalities the PMA will provide. They either run within the Service Runtime Environment or are provided SIMPLI-CITY-externally, i.e., run on a host offered by an external, third-party service provider. External services can be either data services or backend services. In any case, it is necessary to store data about where a service is located and further information describing the service. For services to be hosted within the Service Runtime Environment, also the actual software artefact needs to be stored so the service can be deployed. For external services, a proxy artefact needs to be stored.

Hence, the SIMPLI-CITY Service Registry offers the functionalities to register, update, and discover services. Services are software objects made up from two parts: The actual software (or proxy) artefacts, i.e., a collection of classes, servlets, XML files, libraries that together form a software application (or proxy), and some description of the service, e.g., in the form of metadata.

6.4.2 Subcomponents

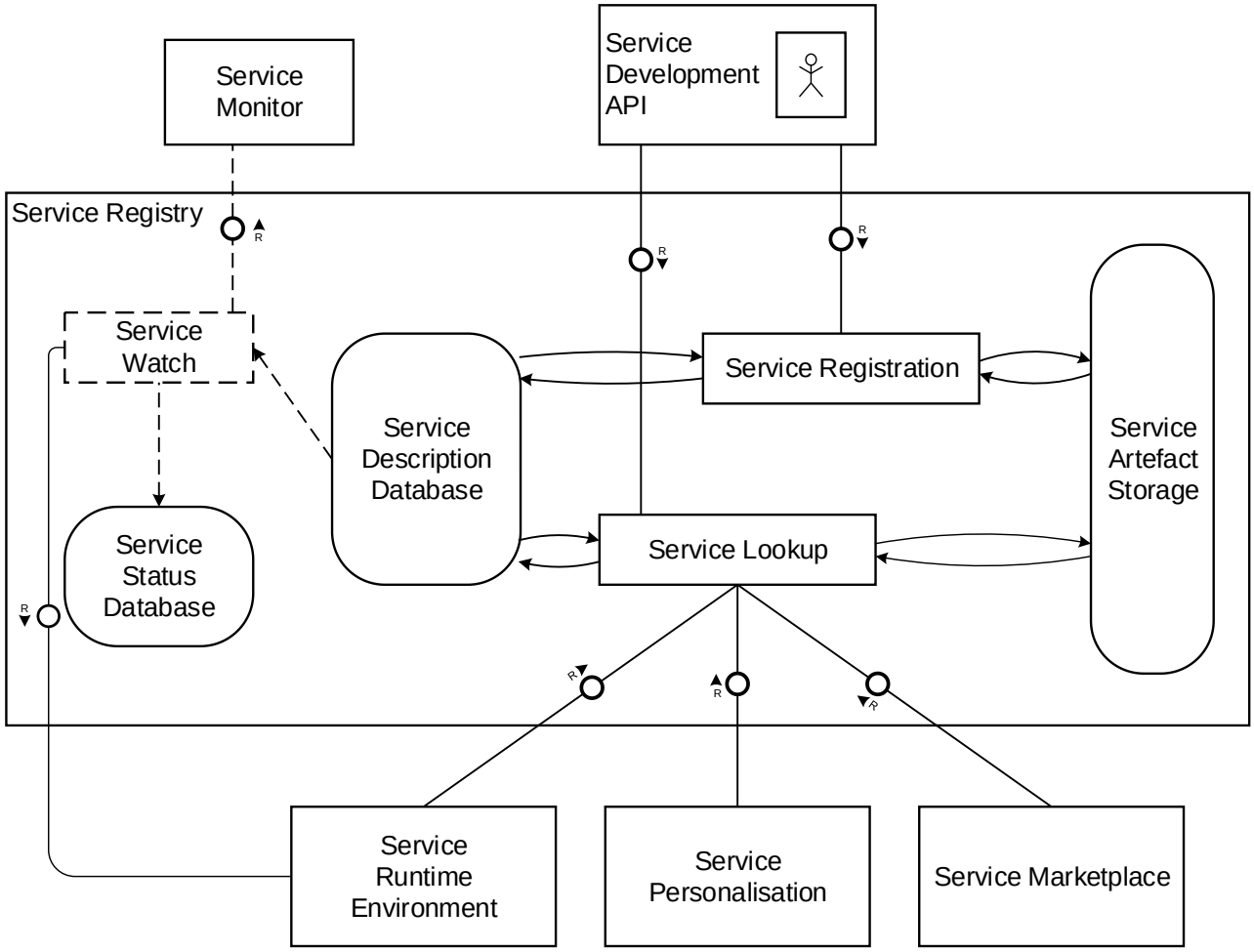

Figure 56: Service Registry Subcomponents and Interactions

To achieve the functionalities described in the previous subsection, the Service Registry provides the following subcomponents as depicted in Figure 56:

- Service Registration: Allows registering new services and updating them.
- Service Artefact Storage: Allows storing service artefacts and proxy artefacts.
- Service Description Database: Allows storing service descriptions as well as information about service developers.
- Service Status Database: Allows storing real-time data about a service's status (i.e., is the service still alive?) as well as about service invocations.
- Service Lookup: Allows to lookup registered services.
- Service Watch: An optional component allowing polling registered data services regularly in order to make sure that data sources are alive. For this, the Service Monitor (see Section 6.2) is used.

The Service Registry is used by other SIMPLI-CITY components:

- Service Development API: To store and update services.
- Service Marketplace: To lookup service information and to store marketplace-related information about services.
- Context-based Service Personalisation: To lookup services.

- Service Runtime Environment: To lookup services; to lookup and store service status information.

6.4.3 Related Requirements

Table 16: Requirements Related to the Service Registry

Requirement	Handled by Subcomponent	Comment
Must Have Requirements		
U148: Service registration (P1)	Service Artefact Storage Service Description Database Service Registration Service Lookup	This is the essential functionality of the Service Registry. It must provide the means to register services by submitting service artefacts and service descriptions. It must also allow updating this information. Must also provide the means to register “external” services, i.e., services not deployed within SIMPLI-CITY – this could be both data and backend services. Instead of storing a service artefact for such services, a proxy artefact will be stored. Must also provide lookup functionality, i.e., to find (internal or external) services based on some search parameters.
U149: Service extension and modification (P1) U150: Service control (P1)	Service Registration Service Artefact Storage Service Description Database	The Service Registry must allow to change service/proxy artefacts and service descriptions. Must allow to delete services.
U151: Service versioning (P1)	Service Registration Service Artefact Storage Service Description Database Service Lookup	Must allow to update service descriptions and service/proxy artefacts. Must be able to store and link different versions of the same service. Must store information about compatibility of different versions. Must distinguish between sales and technical version of a service.
U152: SLA support (P1)	Service Description Database Service Registration Service Lookup	Must allow to define, store, and update SLA information. Must be able to provide this information to the service consumer, if requested.

Requirement	Handled by Subcomponent	Comment
U161: Data services (P1) U162: Backend services (P2)	Service Registration Service Artefact Storage Service Description Database	Must permit to register a created data and backend service and update this information. Must also permit to lookup these services.
U179: Service Metadata (P1)	Service Registration Service Lookup Service Description Database	Must allow to store and lookup services based on metadata, i.e., the service descriptions.
Should Have Requirements		
U92: Secure access to system (P3) U93: Third party access to the system (P3)	Service Lookup Service Registration Service Status Database	Subcomponents need to make sure that only authorised (third) parties are able to lookup and register services. However, this information (e.g., a security token) needs to be provided by the component making use of the Service Registry.
U127: Open interfaces (P2) U128: Openness of the system (P2) U129: Extensibility (P2)	All components	Should provide open interfaces and allow that the system is extended.
U153: Usage of an official SLA standard (P4) U154: Simple SLA description standard (P2)	Service Description Database Service Registration Service Lookup	Must provide an SLA formalism and store SLA data according to it.
U166: Service Developer Signature (P1)	Service Registration Service Lookup Service Description Database	Must allow signing services using a service signature and store this information. Must also allow to lookup this information.
Could Have Requirements		
U101: Backwards compatibility of services (P2)	Service Registration Service Description Database Service Lookup	Should be able to store information about different versions of a service (see U151) and their compatibility. Must enable service consumers to lookup this information.
U182: Service crash reports (P4)	Service Watch Service Status Database	Must allow to regularly poll data services in order to check if they are alive. Must inform interested observers about deviances from expected behaviour.

Requirement	Handled by Subcomponent	Comment
Will not have for now		
U167: Service hot updates (P4)	Service Registration Service Description Database Service Lookup Service Artefact Storage	Must allow to update a service in real time. This means that both service descriptions and service artefacts must be updated on-the-fly. This functionality will only be implemented if there are according resources left.

6.4.4 Interaction with other Components

6.4.4.1 Interaction with the Service Development API

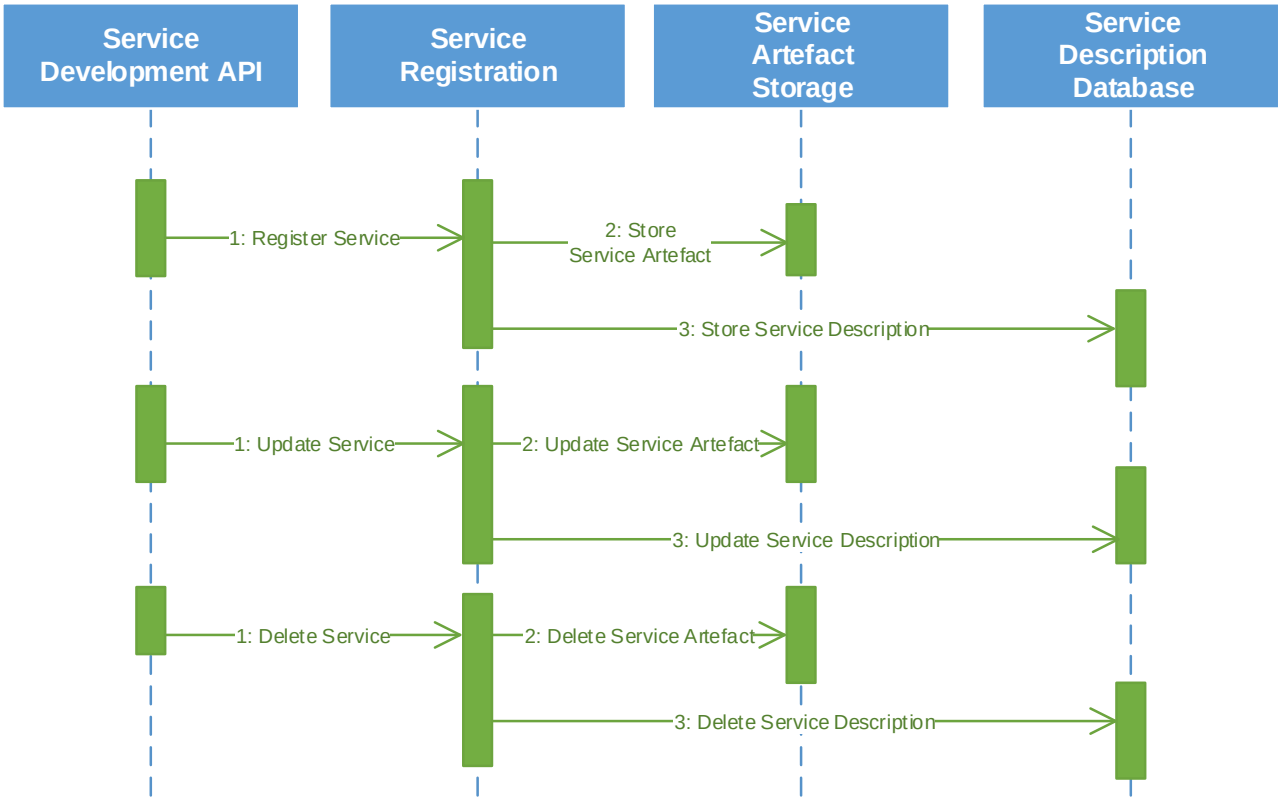

Figure 57: Registration Interaction of Service Development API and Service Registry

Figure 57 shows the different possible interactions between the Service Development API (T5.1) and the Service Registry, i.e., the possibility to register, update, and delete services. The same interactions from the perspective of the Service Development API have been described in Section 8.2.4.2. Notably, the necessary actions for all three kinds of services (data services, backend services, external services) are the same. However, instead of a service artefact, data services and external services require to store a proxy artefact. While not depicted here, the service developer signature (as defined in requirement U166) could be an additional parameter to be stored in the Service Description Database.

When a service is updated, versioning information for this service needs to be provided, i.e., if a service is compatible to the last version(s) and can therefore be used by apps (or other services) which in fact invoke an older version. Furthermore, it is necessary (during the Technical Specification) to define a mechanism which allows to run different service versions in parallel.

Figure 58 shows the interaction between the Service Development API (T5.1) and the Service Registry if it is necessary to lookup a service. The same interactions from the perspective of the Service Development API have been described in Section 8.2.4.1.

Here, interaction is between the Service Development API and the Service Lookup subcomponent instead of the Service Registration subcomponent. Notably, the interaction between the Service Personalisation or the Service Marketplace and the Service Registry is exactly the same as depicted in Figure 58. But of course, instead of the Service Development API, the mentioned components (Service Marketplace and Service Registry) would interact with the Service Lookup subcomponent.

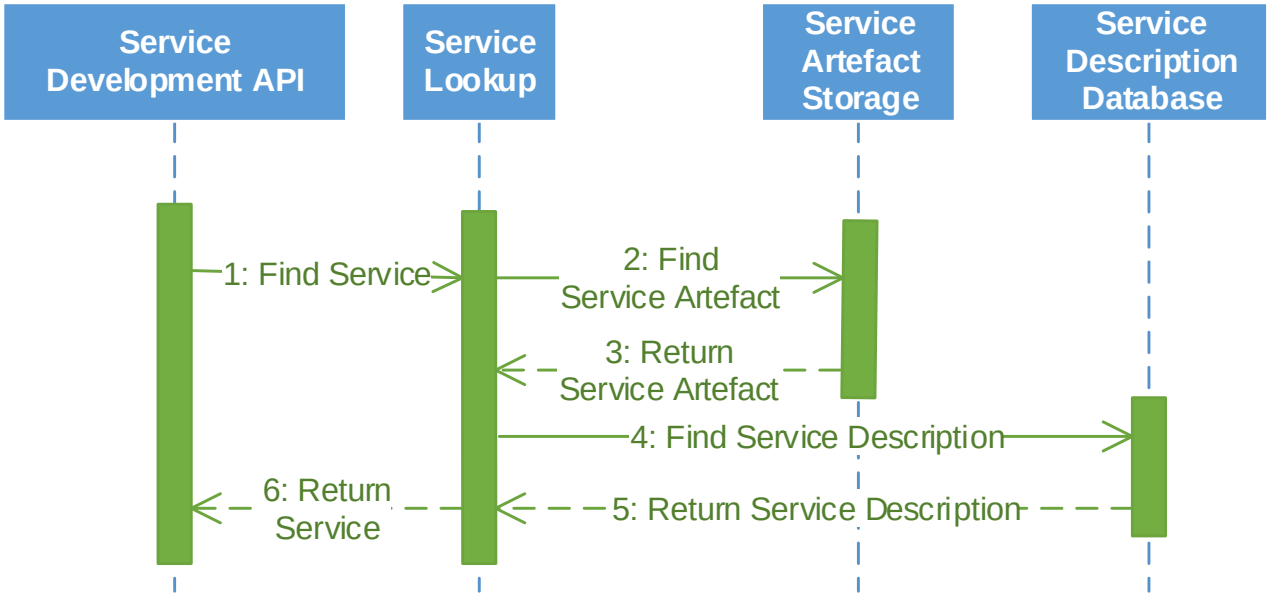

Figure 58: Lookup Interaction of Service Development API and Service Registry

6.4.4.2 Interaction with the Service Runtime Environment

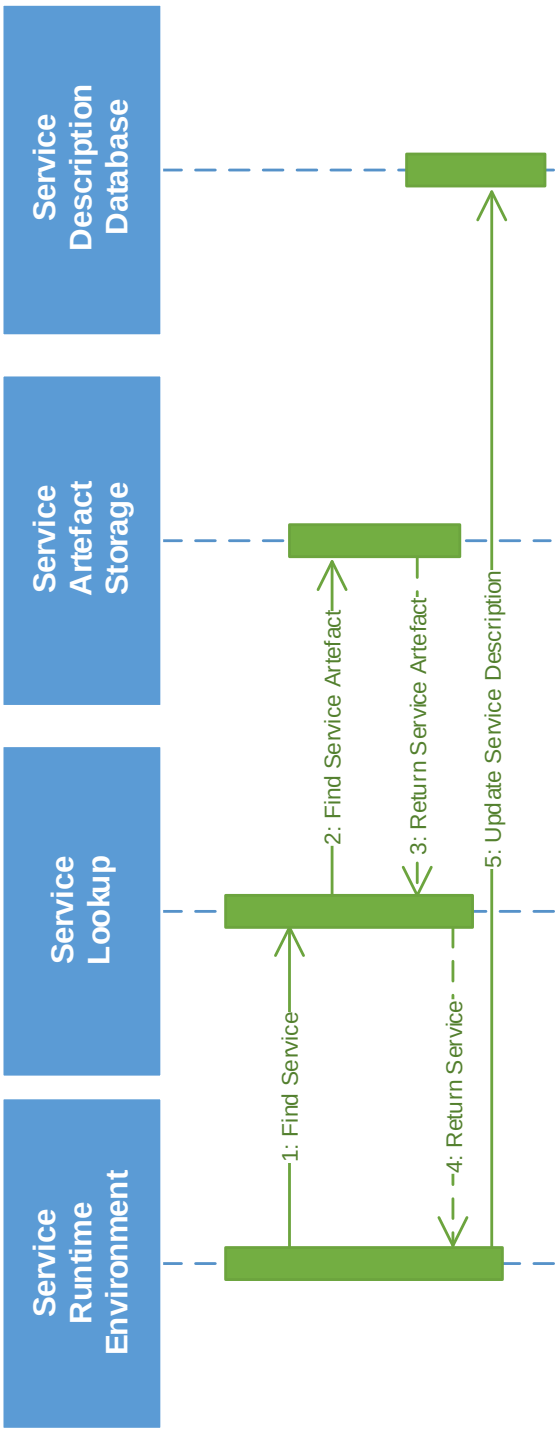

Figure 59: Interaction of Service Runtime Environment and Service Registry

Figure 59 shows the interaction necessary for the Service Runtime Environment to deploy a (internal) backend service. As can be seen, for this, the Service Runtime Environment retrieves a service identified by a particular ID. Once the service artefact is returned, it can be deployed in the Service Runtime Environment (see Section 6.1.4.1). Notably, the Service Runtime Environment updates the service description (more precisely: the service endpoint) once the service has been deployed.

6.4.4.3 Interaction with the Monitoring

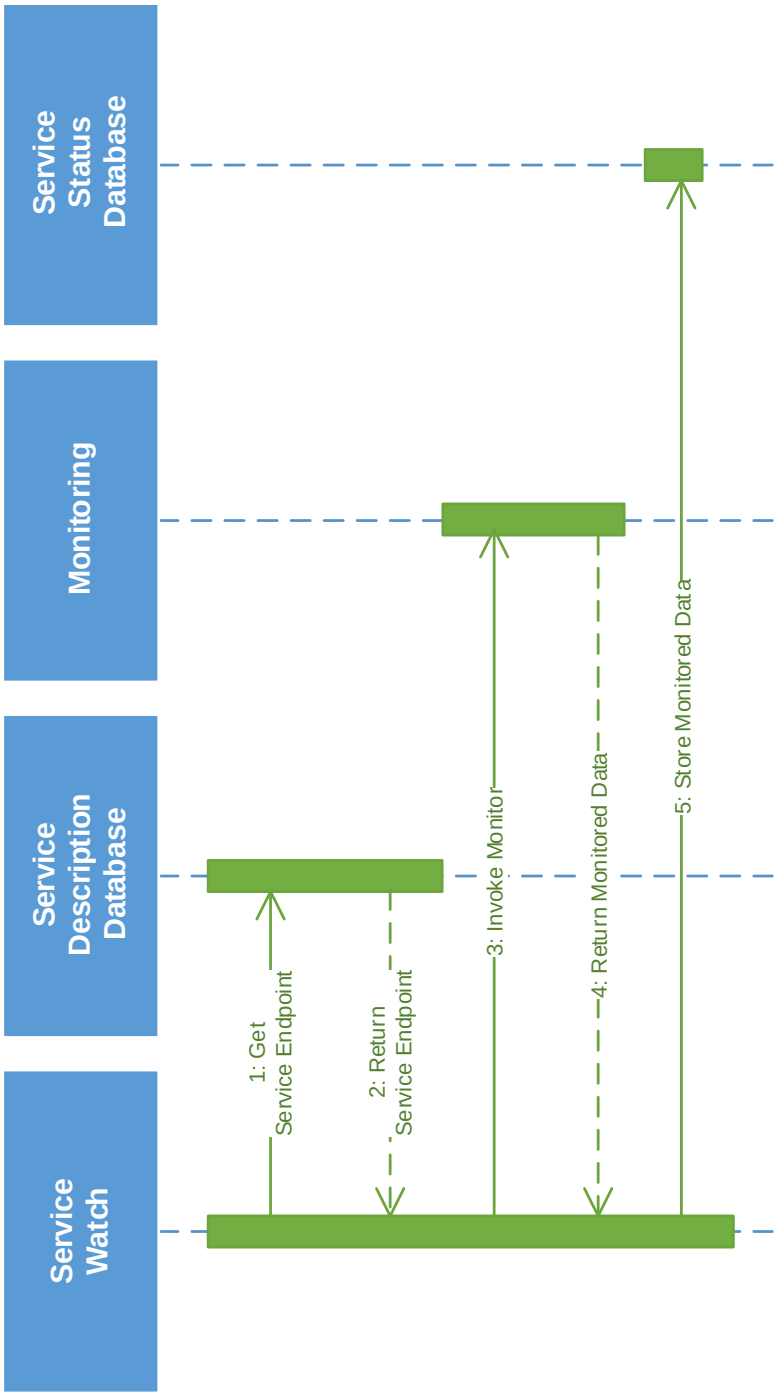

Figure 60: Data Service Watch

As depicted in Figure 60, the Service Watch makes use of the Service Monitor in order to regularly poll data services. The goal is to check if these services are still alive. In order to achieve its functionality, the Service Watch first requests the necessary data about the data services to be invoked from the Service Description Database. Status information about the services is stored in the Service Status Database. Analogue to Section 6.1.4.5, the Service Watch also needs to invoke the actual data service through the Service Runtime Environment (not depicted here for reasons of simplicity).

Notably, the Service Watch is an optional component and will only be realized if there are enough resources left.

6.4.5 User Interface

Service management is done through the Service Development API and the Service Marketplace, which offer all necessary functionalities to describe and update services. Hence, the Service Registry does not offer a separate UI.

6.4.6 Conceptual Data Model

As defined above, the Service Registry needs to store information about services. Three different kinds of services need to be taken into account:

- Backend services running in the Service Runtime Environment: For these services, both the service artefact and the service description need to be stored.
- External backend services provided and hosted by third parties: For these services, the service description needs to be stored. In addition, information about the proxy used to invoke this external service, needs to be stored in terms of a proxy artefact.
- Data services provided and hosted by third parties: Comparably to external backend services, no service artefact will be stored for data services. However, the actual service description as well as a proxy artefact needs to be stored. Notably, data service descriptions feature a link to a taxonomy which indicates the data category provided by this service.

The necessary service artefacts will be stored using an archive file format, which makes it possible to deploy them in the Service Runtime Environment. As long as a file format is able to encapsulate a service, it can be chosen for usage in SIMPLI-CITY. Examples for this are the JAR file format for some OSGi implementations, the WAR format as used in application servers like Apache Tomcat or Glassfish, or the SCDL format as used in Apache Tuscany.

For the description of services, a common standard like USDL or WSDL will be applied; if necessary, the description standard will be extended with further, SIMPLI-CITY-specific attributes. One example for such a specific attribute is the taxonomical information for the data services. The data format for the proxy artefacts still needs to be discussed. Furthermore, an SLA data format needs to be chosen. As defined in the requirements analysis, SIMPLI-CITY should make use of a standardised SLA format, which is preferably quite lightweight. For a more detailed preliminary discussion of the service data model, see Section 9.3.

6.4.7 Parameters to Take into Account for Technical Specification

Table 17: Criteria for Technical Specification

Parameter	Importance (--, -, -+, +, ++)
Generic Criteria	
Up-to-Datedness	-+
Stability	+
Extensibility & Open Source/Standards	+
Familiarity	+
Performance	++
Interoperability	+
Specific Criteria	
GUI	-
Extensible Data Model	++
SLA Support	+
Registry AND Repository	++
Various Possibilities Regarding File Storage	-+
Various Possibilities Regarding Database	-+
User Management	+
Version Control	+

6.5 Service and App Marketplaces

6.5.1 Overall Functional Specification

One of the main achievements that SIMPLI-CITY will deliver is its flexible way of adding new functionalities for road users. Those new functionalities will be provided by installing new apps on the PMA. Each app will usually be wrapped around one or more backend services, which will deliver data and perform server-side computations.

Apps will be offered by the SIMPLI-CITY App Marketplace in a PMA-based UI. This allows users to discover, buy, install, upgrade and uninstall apps via the App Marketplace.

Analogously, services will be offered in the Service Marketplace. The Service Marketplace is also part of task T5.4 (Mobility Service and App Marketplaces). While the app side mainly targets end users, the Service Marketplace targets developers and will therefore not be directly seen by end users as they will access all functionalities indirectly via apps installed on their PMA.

The main functionality of the marketplaces is to provide end users with the functionality to discover, download and install apps on the PMA (App Marketplace) and to provide software developers with the functionality to discover and study services as a base for new developments within the project (Service Marketplace).

The project strongly favours free software and as such it strongly fosters the development of apps and services on a free base. However, in order to provide a maximum exploitation, the project will also allow developers to establish a commercial market in order to make money by providing apps and services to other people (users and/or developers). This will leverage a multiplier effect and will motivate developers to develop on top of the project APIs. As such, apps may be offered in the marketplace commercially to end users and services may be offered commercially to developers. This component will support those two types as follows:

Services

Services will be used as a base for developers for creating apps. End users will not notice this as they will purely interact with apps without having to know what a service is. As such, services sold via the project Service Marketplace will be targeting developers. Licenses of commercial services may be offered on a pay-per-use base or on a time-specific base (payment on a weekly/monthly/yearly base). A developer may use the Service Marketplace for purchasing a service. On each invocation, the Service Runtime Environment will check if the user has a valid license to use a service and will otherwise not execute the service request (see Section 6.1.4.3). The Service Marketplace will provide developers with a UI to browse services and buy licenses that allow them to use that service in their app. Service developers have access to another UI that provides them with statistics (e.g., downloads, usage) concerning their service.

Apps

Apps will be offered to end users via the PMA in an own mobile UI. From an end user perspective, this concept will be identical to those of existing mobile marketplaces such as the Google Play Market or Apple's App Store. As such, the licensing offered by SIMPLI-CITY will be identical to those markets. More precisely, the App Marketplace will allow app developers to offer apps for purchase by specifying a one-time fee. This will allow an end user to download and install the app as well as all future updates. For achieving this, the

PMA will provide a UI for performing the full purchase process from the mobile device, i.e., without having to use a web browser. The PMA will only allow users to download and install apps and updates after a valid license has been purchased. Additionally, a second UI will be provided with a web-based view for developers in order to publish and control the sales process for offering new apps.

6.5.2 Subcomponents

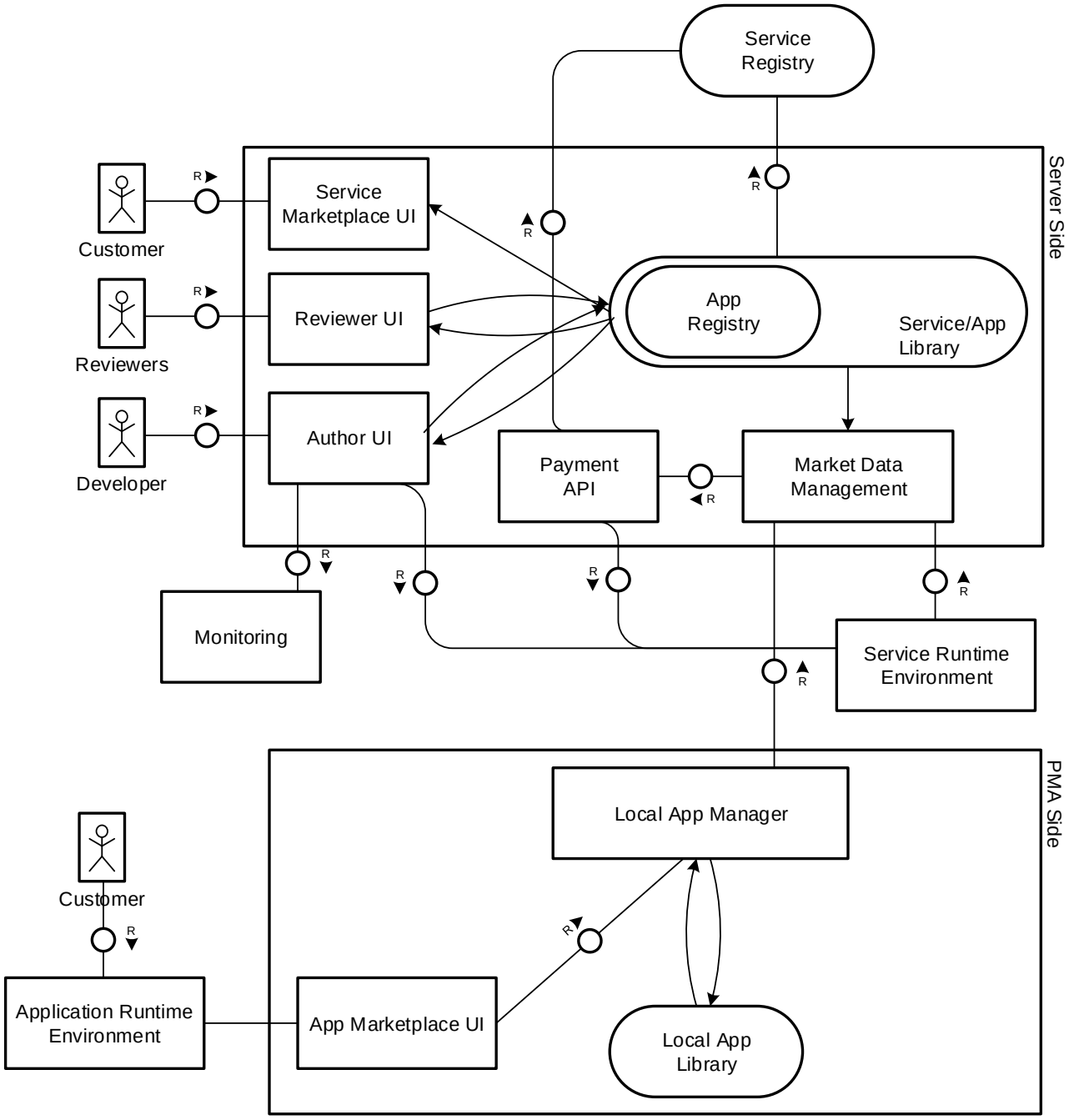

Figure 61: Service and App Marketplaces Subcomponents and Interactions

To achieve the functionalities described in the previous subsection, this component provides the following subcomponents as depicted in the figure above:

PMA Side:

- App Marketplace UI: Provides an end user interface for installing and uninstalling apps on the mobile device. Furthermore, it provides the means to discover apps.
- Local App Library: Manages a list of installed apps on the PMA, their manifests and their dependencies.
- Local App Manager: Handles installation, upgrade and uninstallation procedures.

Server Side:

- Author UI: Allows developers to add new apps and services to SIMPLI-CITY. Also allows developers to check monitoring and accounting data for a particular service.
- Reviewer UI: Provides a web-based view for reviewers in order to test and accept/reject apps and services. Only after a reviewer has cleared an app or a service for publication, it will be listed on the respective marketplace.
- Service Marketplace UI: Provides a web-based UI for service consumers for finding services
- Service/App Library: Is the place where the apps and services and their manifests are stored. As it can be seen in Figure 61, the Service/App Library encapsulates both the App Registry and the already introduced Service Registry (see Section 6.4).
- Market Data Management: Handles the overall data management for apps and services and provide them to the other subcomponents. For example, the Market Data Management is the component that takes care of the license status validation and is therefore used by the Service Runtime Environment (see Section 6.1.4.3).
- Payment API: Handles payment options and the payment process with different payment providers (for example PayPal, Google Wallet). It interacts with the Service Registry and the Accounting subcomponent of the Service Runtime Environment in order to get the necessary information to start a payment process.

The Marketplace component is used by other SIMPLI-CITY components:

- Service Runtime Environment: To receive license information for a service.
- Application Runtime Environment: To receive data about installed apps. This is not explicitly depicted in Figure 61, but implied by the channel between the Customer and the App Marketplace UI.

Figure 62 shows the interactions between the subcomponents of the App Marketplace in order to provide the necessary CRUD functionalities. The analogue interactions for services have been described in Section 6.4.

The App Marketplace internally provides part of the CRUD Operations via the Author UI, since authors are the only group of customers that are allowed to upload, update and remove apps. The retrieval of apps is handled via the Application Runtime Environment and explained later (see Section 6.5.4.3).

The upload of a new app is done via the Author UI, which sends the new App Manifest and app description data to the Service/App Library, which stores it in the App Registry. The app is now open for the review process (see Section 6.5.4.2) and will eventually be published to the App Marketplace once it has been reviewed.

Updating of Apps works in a similar way, because the updates will need a review as well. The new Manifest file coming with an update will be stored in the App Registry, but does not overwrite old Manifests to keep the PMA from re-downloading whole apps every time there is an update for a particular SIMPLY-CITY app (see Section 6.5.4.1).

The removal of apps is only done on the Server Side since apps should still be able to run within the PMA even if they are not available for download anymore. The only thing that needs to be concerned is the removal of service usage licenses. If a service usage license expires, the App Marketplace should suggest the removal of an app from the PMA for stability and availability reasons.

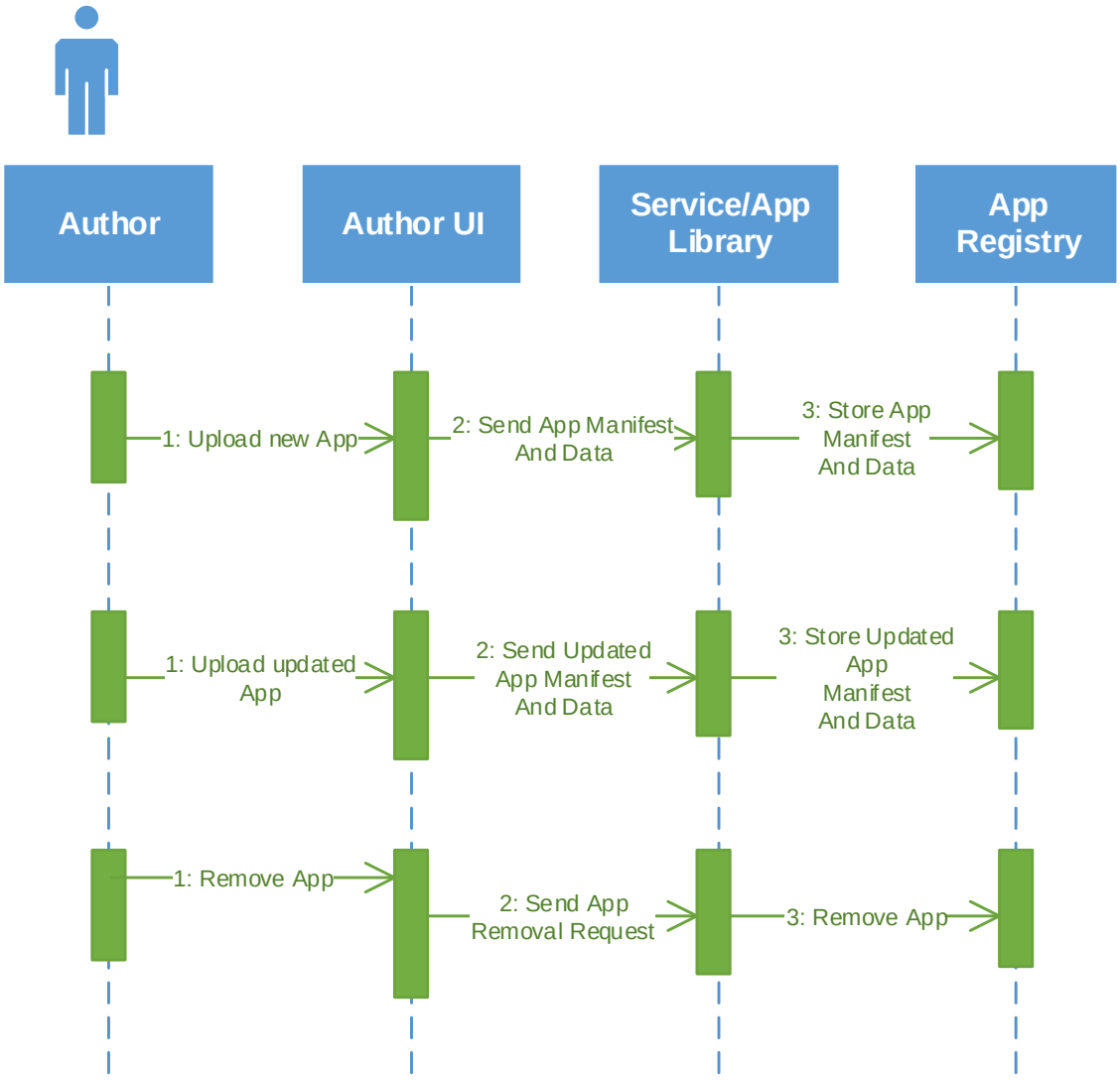

Figure 62: CRUD Operations of the App Marketplace via the Author UI

6.5.3 Related Requirements

Table 18: Requirements Related to the Marketplace

Requirement	Handled by Subcomponent	Comment
Must Have Requirements		
U62: Versioning of apps (P1) U63: Upgrading of apps (P1) U144: App versioning (P1)	Market Data Management Local App Manager Local App Library App Marketplace UI Service/App Library App Registry	These functionalities will allow the App Marketplace to keep control of installed apps and their versioning on the customer's PMA. Support for different app versions will allow the PMA to run old versions of apps and services as long as they are not marked as deprecated.
U64: Quality assurance – Mockup (P1) U142: Quality/checking certification (P1)	Reviewer UI	Reviewers will be provided with an according user interface which makes it possible to review and clear apps and service. Apps and services will be offered on the respective marketplaces only after a successful review. Reviews will also be done for updates. This requirement describes a mockup implementation of the quality assurance. The full implementation is discussed below.
U66: App marketplace easy to use (P1) U67: Apps are easy to buy (P1) U68: Apps are easy to search (P1)	Service Marketplace UI App Marketplace UI	Keep the usability of the App Marketplace high and intuitive to especially allow non tech-savvy customers easy access to new apps.
U69: App download to the device (P1) U70: App installation (P1) U71: App uninstallation (P1)	Local App Manager App Marketplace UI Local App Registry	Provide automatic app installation and removal with as little user interaction as possible (e.g., download and install an app with just one click).
U138: Provision of app statistics (P2) U139: Provision of service statistics (P1) U182: Provision of statistics, e.g., usage, traffic (P3)	Author UI Market Data Management	Simple user interface for service providers and app developers to get an overview of the usage of their services and apps. This will allow them to find bottlenecks.

Requirement	Handled by Subcomponent	Comment
U143: App publication (P1) U145: App removal (P1) U147: Easy to publish (P2)	Author UI Service/App Library App Registry	Provides a workflow for app developers to upload a new app to the Service/App Library and a view for publishers to describe and publish new services that are assigned to them.
Should Have Requirements		
U55: Rating of apps (P3)	App Marketplace UI	Gives customers the ability to view the rating of apps before downloading and installing them. Also provides an interface for customers to rate and review apps after having used an app on their PMA.
U56: Feedback to developers through the marketplace (P3)	Service Marketplace UI App Marketplace UI Author UI	Gives customers the ability to contact app or service developers for questions, enhancement ideas and general feedback.
Could Have Requirements		
U57: Social network functionality with the objective of spreading the word (P3) U141: Spreading the word (P3)	App Marketplace UI	Will provide functionality for customers to advertise apps via common social networks. This will only be available in the App Marketplace, since services are not globally visible for end users.
U58: User recommendations (P3)	Service Marketplace UI App Marketplace UI	Will provide app recommendations based on already installed apps (e.g., “users who installed this app also installed the following apps”). This can also work for removed apps. (e.g., “you removed this app, maybe another app will suit your requirements”).

Requirement	Handled by Subcomponent	Comment
U60: Payment for services – Mockup (P1)	Service Marketplace UI App Marketplace UI Author UI Payment API	Will provide an interface for customers to buy services and apps via different payment methods (e.g., PayPal, Google Wallet, and Credit Card). This is a simple wrapper for the different payment methods that will be called by an internal component. This requirement describes the mockup implementation. The full implementation is discussed below.
U128: The system allows app developers to make money (P1) U129: The system allows the consortium to make money from apps (P2) U130: Apps are free but supported, i.e., ed hat model (P4) U131: Free and premium versions of apps (P4) U132: Guarantee and minimum service: life time, support time (P5) U133: The system allows service developers to make money(P1) U134: The system allows the consortium to make money from services (P2) U135: Services are free but supported, i.e. red hat model (P4) U136: Free and premium versions of services (P4) U137: Guarantee and minimum service: life time, support time (P5)	All components	Allows developers to generate money from bought/subscribed apps and services. The fees and payment methods (e.g., monthly fee, traffic based, call based) can vary and will depend on the choices developers make. The fees can be changed by developers while the service/app is on the market, but the “updated” fee will only affect new subscriptions and purchases. Alternatively, apps and services can be offered for free.

Requirement	Handled by Subcomponent	Comment
U65: Quality assurance – Fully deployed (P3)	Reviewer UI Author UI	User Interfaces for the respective user groups (reviewers, developers), which allow them to create and read reviews concerning their apps and services. This requirement describes the full implementation. A mockup implementation is discussed above.
U140: Promotional aspects (P4)	Service Marketplace UI App Marketplace UI	Allows developers to promote their apps to increase visibility. This will include options to get them on a list of promoted apps or increase their rank globally.
Will not have for now		
U61: Payment for services – Fully deployed (P3)	Service Marketplace UI App Marketplace UI Author UI Payment API	Will provide an interface for customers to buy services and apps via different payment methods (e.g., PayPal, Google Wallet, and Credit Card). This is a simple wrapper for the different payment methods that will be called by an internal component. This requirement describes the full implementation. A mockup implementation is discussed above.
U146: Call for developers (P5)	N/A	Will not be realized due to its low priority combined with relatively large time that would be needed for its realization. It may, however, be realized after the project.

6.5.4 Interaction with other Components

6.5.4.1 Interaction with the PMA

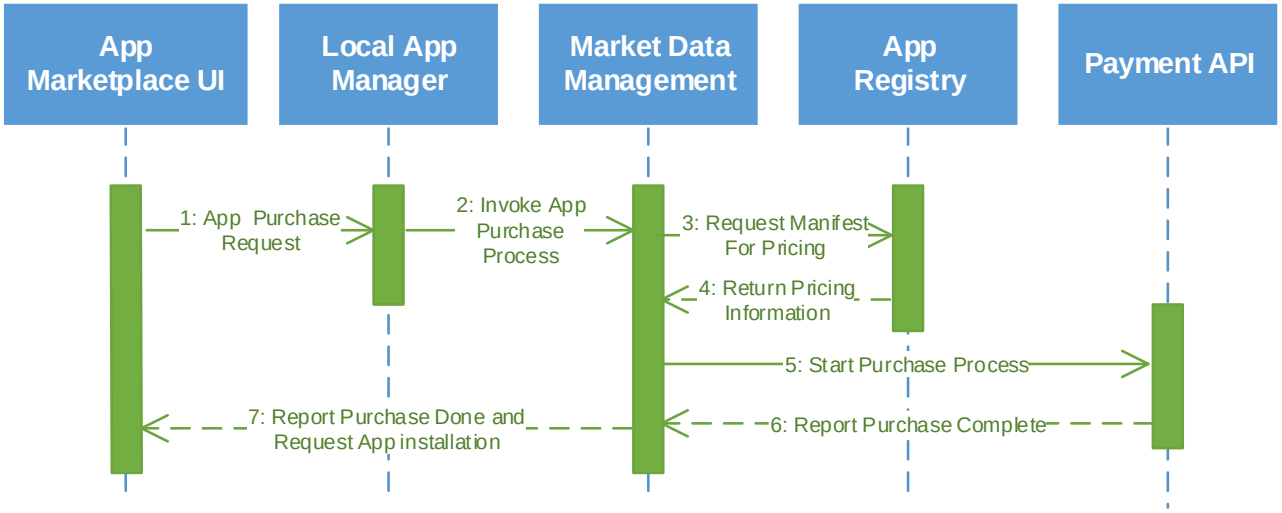

Figure 63: Process of App Purchase

In the following, the different interactions which are necessary in order to make use of apps within the PMA, will be presented. This includes the possibility to purchase apps (Figure 63), install them (Figure 64), update them (Figure 65), and remove them from the PMA again (Figure 66).

In order to make use of an app, it first needs to be purchased by the customer (see Figure 63). Naturally, this is done through the App Marketplace. As soon as a customer decides to buy an app, the App Marketplace UI triggers the Local App Manager to initiate the purchase process. As the pricing options are stored in the Manifest file (see Section 6.5.6) the App Registry (part of the not-depicted Service/App Library) needs to be invoked via the Market Data Management to receive the information, which will then be forwarded to the Payment API. The internal logic of the Payment API then handles the purchase request by invoking the selected payment method (e.g., Google Wallet) and by returning the result of the specific purchase. If the purchase is done, a success message will be reported to the App Marketplace which will then trigger the App Installation as described in Figure 64.

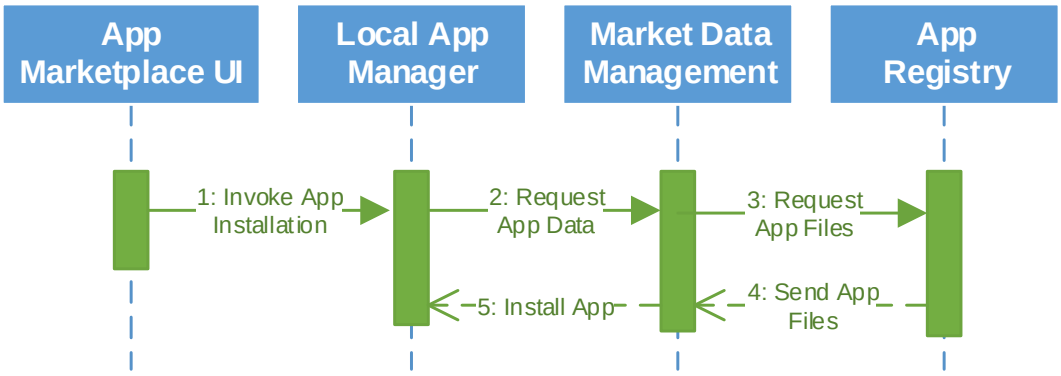

Figure 64: Process of Installing an App

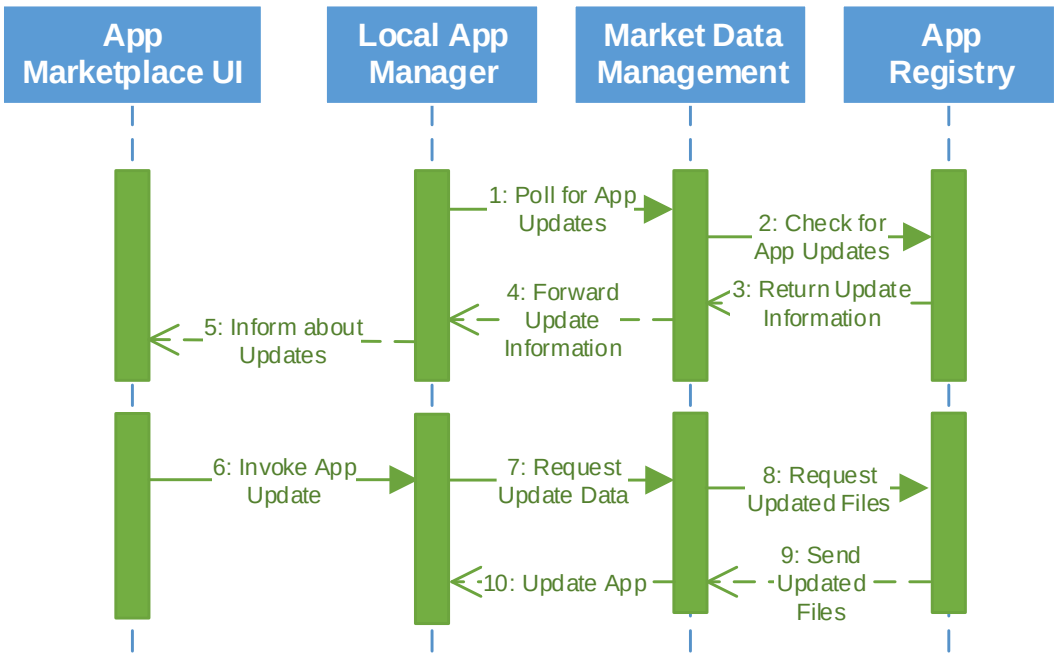

Figure 65: Process of Updating an App

For the installation, the App Marketplace triggers the Local App Manager, which itself is responsible for a correct and complete installation of an app. The Local App Manager requests the specific app data (e.g., the App Manifest containing a description of the app and the files the app contains). The Market Data Management subcomponent requests the specific app files from the App Registry and forwards it to the Local App Manager, which then copies the app files to the Local App Library (not depicted in Figure 64) via the Local App Manager.

SIMPLI-CITY allows updating both apps and services. The according process is depicted in Figure 65 and is quite similar to updates in mobile operating systems like Apple iOS or Google Android. After an update has been provided by an app developer, the Local App Manager can poll for updates (this is done on a regular basis, e.g., once per day or when the App Marketplace UI is started on the PMA). The updates are provided by the Market Data Management, which itself receives update information from the App Registry. Notably, as can be seen in Figure 65, there can be a gap between being informed about new updates (Step 5) and actually starting the update process (Step 6).

As soon as a customer starts an app update, the App Marketplace invokes the Local App Manager to initiate the update. An update always contains a renewed Manifest file, which includes a change log between the current and the previous version. This can be used to generate updates between any two versions of an app. Naturally, the renewed Manifest file is available in the App Registry and can be requested by the Market Data Management. The Market Data Management will now generate the exact updates between the PMA version of the app and the current App Registry version and copy the new and updated files to the Local App Library (not depicted) via the Local App Manager.

If a user does not want to make use of a particular app anymore, it can be removed from the PMA (see Figure 66). App removal is invoked by the costumer by invoking the removal via the App Marketplace. The Local App Manager then requests the App Manifest from the Local App Library which contains all files included in the specific app. The Local App

Manager then removes the files as described in the local App Manifest and will then delete the manifest itself.

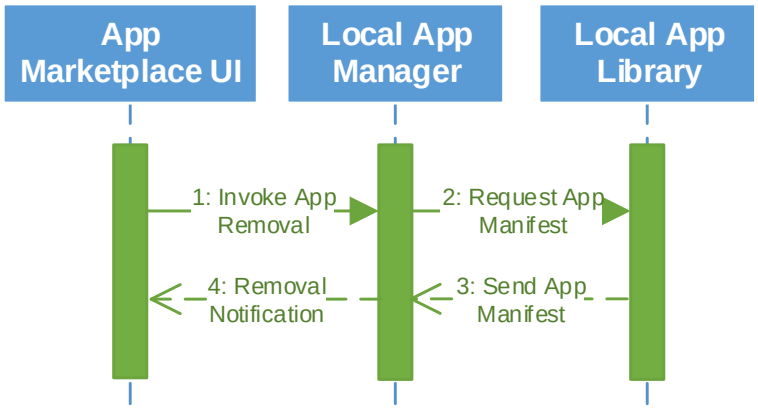

Figure 66: Removing an App

6.5.4.2 App and Service Reviews

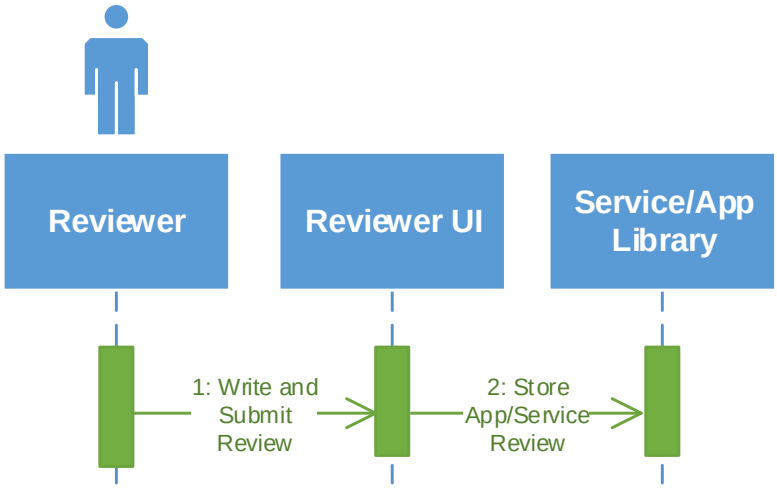

Figure 67: App and Service Review Process

The Reviewer UI provides an according UI to write and submit app and service reviews. The interaction for this is identical for both apps and services; however, the information about the successful review will be stored either in the App Registry or Service Registry (not depicted). Once the review has been done and the app/service is accepted, the app/service is published in the respective marketplace (not depicted). The customers can then find the reviewed app in the App Marketplace UI via the Application Runtime Environment (see next subsection) and services in the Service Marketplace UI (see Figure 61).

6.5.4.3 Interaction with the Application Runtime Environment

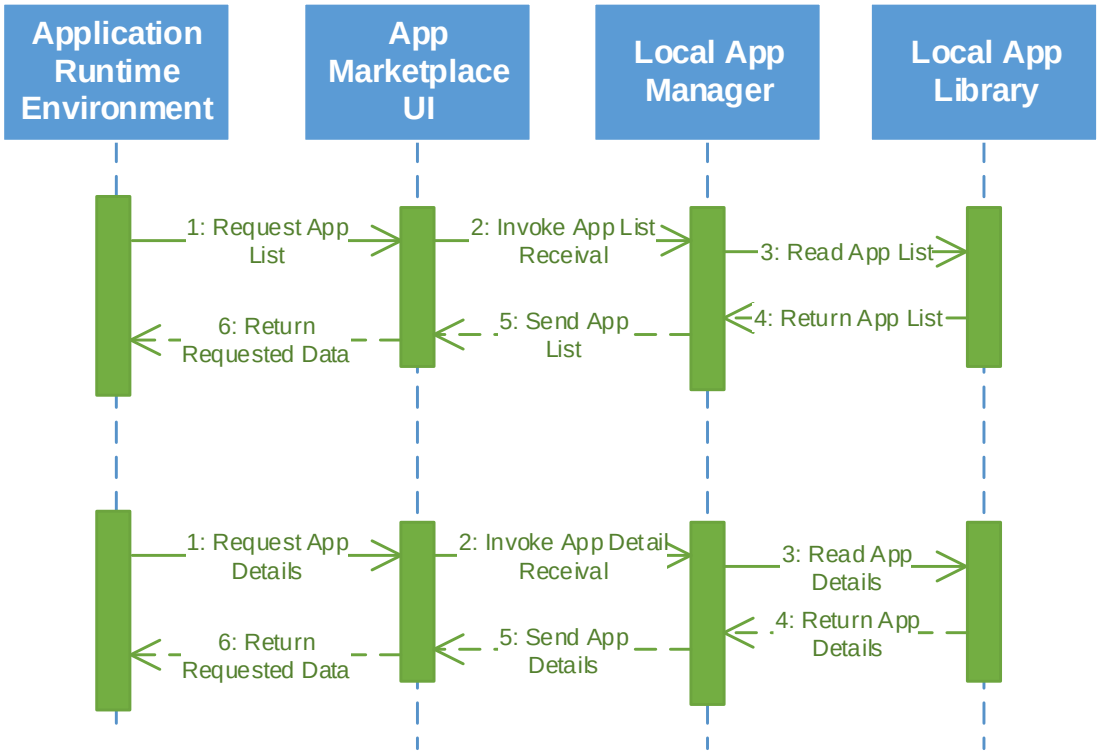

Figure 68: Interaction of the Application Runtime Environment and Marketplace

The App Marketplace is invoked by the Application Runtime Environment (by the user, see Figure 61 – not depicted in Figure 68 to keep the figure simple) to receive a list of installed apps or details about a single app via the App Marketplace UI. The App Marketplace UI triggers the Local App Manager to read the data from the Local App Library and returns the data to the App Marketplace UI, where it can be used for further operations – this procedure is depicted in the upper part of Figure 68.

Furthermore, it is possible to receive information about a specific app from the Local App Library (depicted in the lower part of Figure 68). To achieve this, the Application Runtime Environment invokes the App Marketplace UI to request app details from the Local App Manager, which has access to the Local App Library. The requested information is then forwarded to the Application Runtime Environment via the Local App Manager and the App Marketplace UI.

Each component is responsible for a certain task in this interaction. The Application Runtime Environment is the central caller for each action that is shown in the App Marketplace UI. The Local App Manager is managing the apps on the PMA and responsible for accessing the Local App Library, where all the information about installed apps is stored.

6.5.4.4 Interaction with the Service Registry

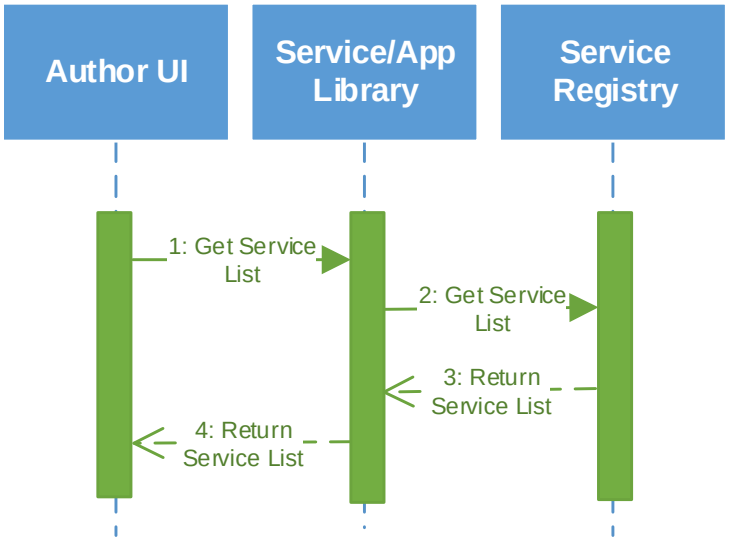

Figure 69: Interaction with Service Registry to Get Service List

The information related with a service is stored in the Service Registry (see Section 6.4). So, different subcomponents of the Service Marketplace interact with the Service Registry in order to access or update this information. These subcomponents are the Author UI, the Reviewer UI, and the Service Marketplace UI. In all the cases, the interaction with the Service Registry is performed through the Service/App Library.

One set of these operations is done by the service developer by means of the Author UI (see Figure 69). This subcomponent allows the service developer to edit the information of a service and to manage its publication in the Service Marketplace. When the service developer accesses the Author UI, it queries the Service Registry, which returns the list of services created by the service developer. These services include the services already published in the Service Marketplace and the services created but not published yet.

The service developer can view all the information associated with a service by choosing it from a list in the Author UI. In this case, the Author UI queries for the detailed information to the Service Registry, as illustrated in Figure 70.

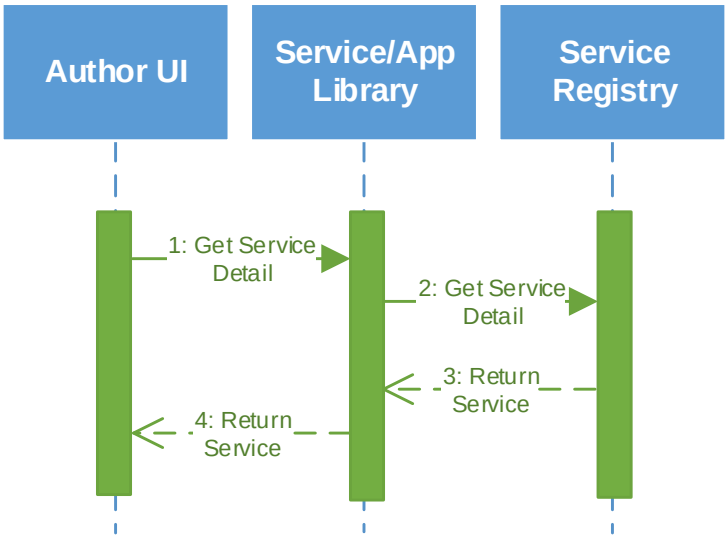

Figure 70: Interaction with the Service Registry to Get Service Details

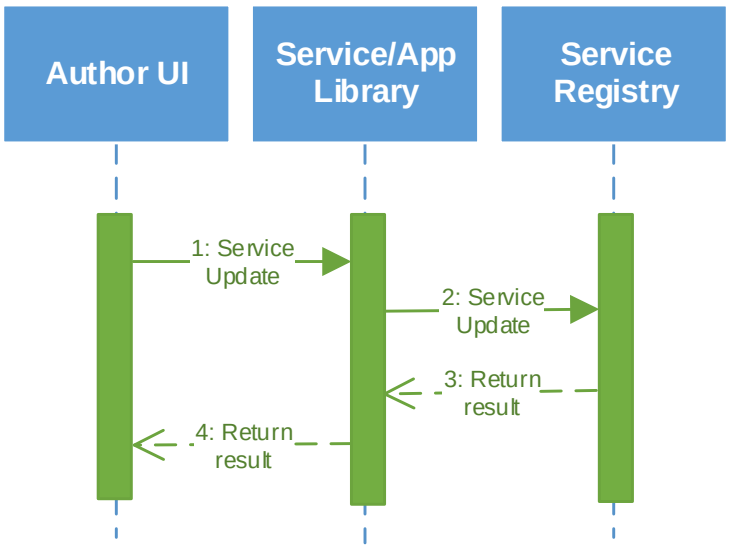

Figure 71: Interaction with the Service Registry for Service Data Updates

The service developer can edit the information of a service and publish it in the Service Marketplace by means of a service update query, as depicted in Figure 71. The same query can be used in order to update the information of a service, once it is already published in the Service Marketplace, or to delete a service from the Service Marketplace.

Once the service developer has updated all the information of a service and has published it in the Service Marketplace, the service has to be reviewed and accepted by the reviewers before being visible in the Service Marketplace and thus before being able to be found and used by other developers (see Section 6.5.4.2).

Service developers use the Service Marketplace UI in order to look for published services in the marketplace, and to use these services within their apps. Developers can search for services by means of different search criteria like the name of the service or the keywords to describe the service. The Service Marketplace UI interacts with the Service Registry via the Service/App Library to perform this search, as shown in Figure 72.

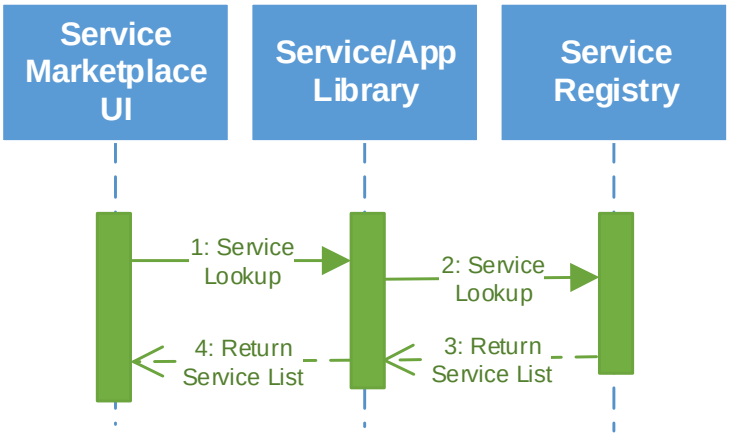

Figure 72: Interaction with the Service Registry for Service Lookups

6.5.4.5 Interaction with Service Monitoring and Accounting

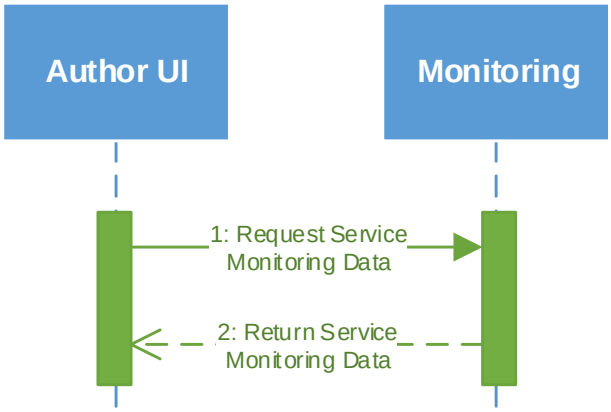

Figure 73: Interaction with (Service) Monitoring

As described above, the Service Marketplace UI will also offer the possibility to show developers accounting and monitoring data. As depicted in Figure 73, the according interaction is pretty simple – the Author UI requests data (based on the user data or for some specific service) for a particular service, which is then returned to the user via the Author UI. For accounting data, the interaction is exactly the same and therefore not depicted.

Apart from showing accounting data to the user, this data is obviously also needed by the Service Marketplace for payment purposes. As can be seen in Figure 74, the Payment API regularly (e.g., on a daily basis) polls the Service Registry in order to identify all services which have a license model that makes it necessary to start a payment process. Obviously, payment has not to be done if a service offers a free license; however, if the service developer offers a pay-per-use or flat-rate model, the payment process needs to be started. Based on the knowledge about which services need to be billed for, the Payment API then requests the accounting data for these services from the Service Runtime Environment, as the Accounting component is a subcomponent of it. Once the data is returned, the Payment API can trigger the actual payment for service usage.

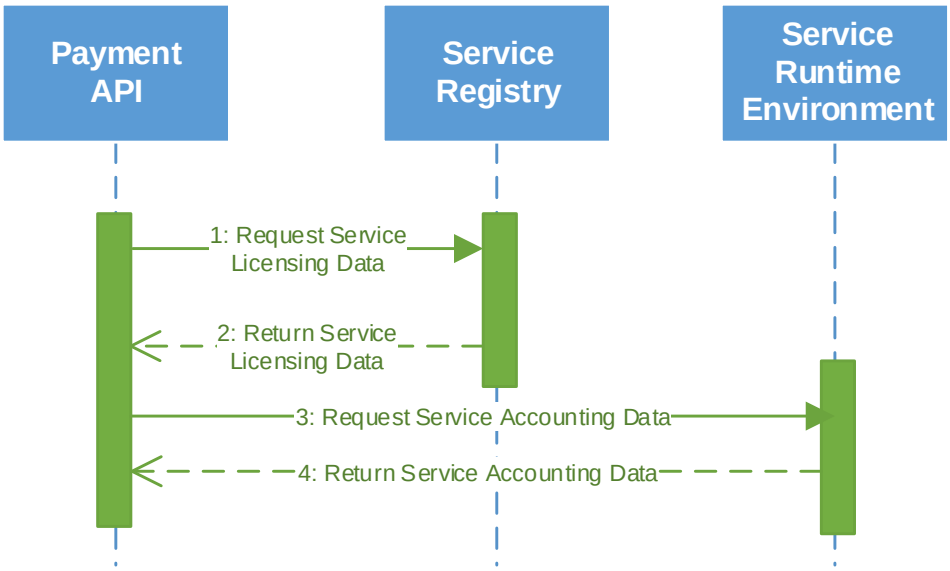

Figure 74: Interactions for Service Payments

6.5.5 User Interface

The following interfaces show examples for possible designs.

6.5.5.1 Service Marketplace UI

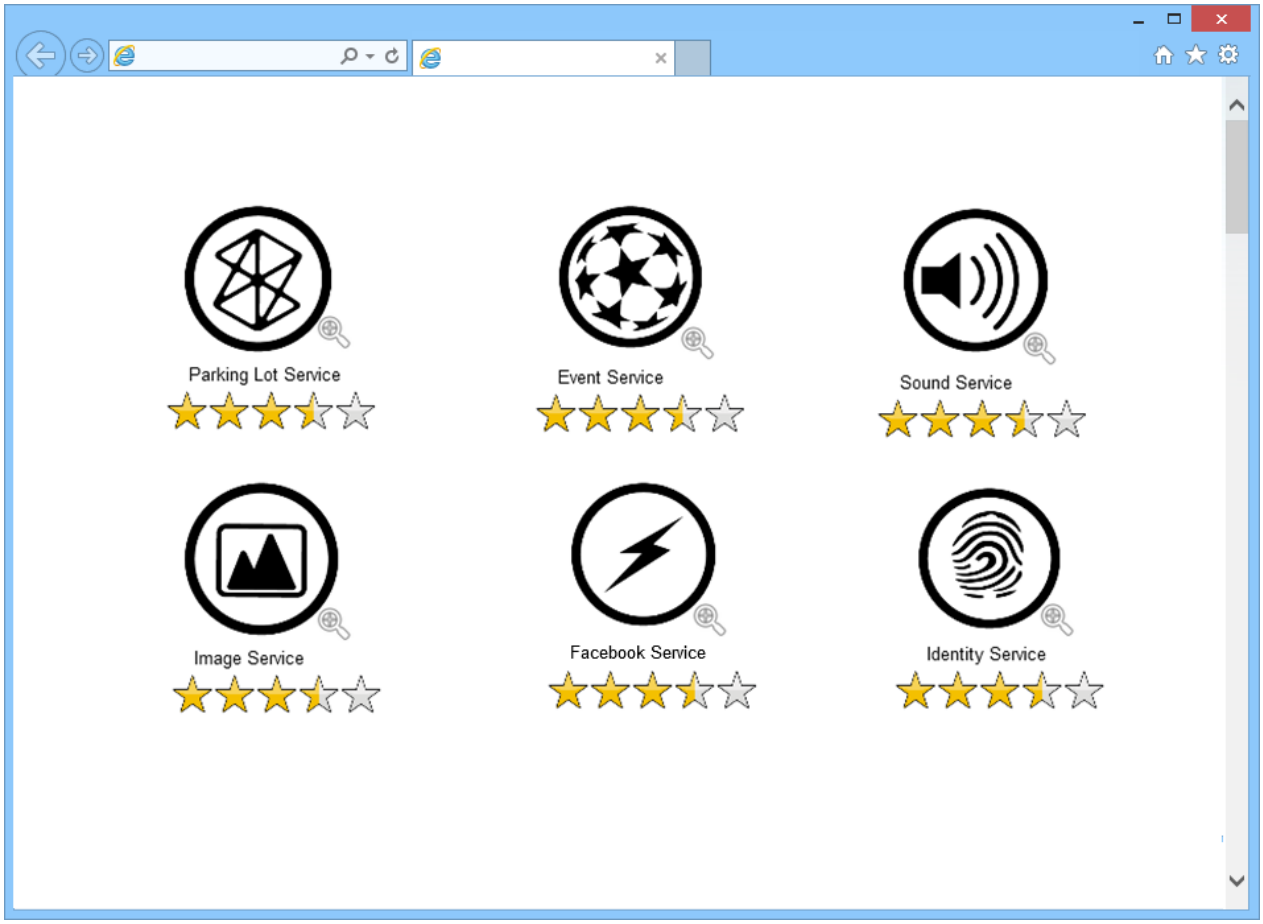

Figure 75: Service Marketplace UI

The web-based view of the Service Marketplace allows browsing and discovering services from any modern web browser. It allows the discovery of services by category and provides an overview about services in a list. A click on each service entry shows details about the service such as descriptions and author information.

The depiction of monitoring data is presented in Section 6.2.5.

6.5.5.2 App Marketplace UI

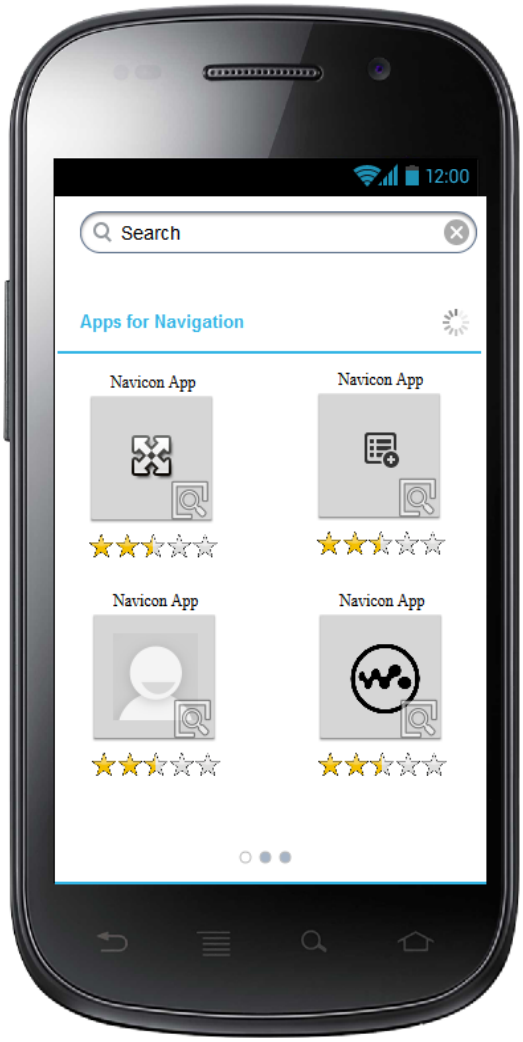

Figure 76: App Marketplace UI

The PMA version of the marketplace allows users to discover apps and to view their details such as descriptions or ratings. Users may then use the PMA version of the marketplace to install apps or to uninstall and upgrade them.

6.5.6 Conceptual Data Model

Apps are described via their Manifest file, which contains common information about the app (e.g., name, description, author, used services …). This is stored in the server-side App Registry. The apps themselves may use other data models including the SIMPLI-CITY Unified Data Model, but this is not applicable in this general description of the Service and App Marketplaces.

For an overview of the applied data models, please see Section 9.3 for services and Section 9.4 for the App Manifest.

6.5.7 Parameters to Take into Account for Technical Specification

Table 19: Criteria for Technical Specification

Parameter	Importance (--, -, -+, +, ++)
Generic Criteria	
Up-to-Datedness	+
Stability	-+
Extensibility & Open Source/Standards	++
Familiarity	++
Performance	-
Interoperability	-
Specific Criteria	
Distributed Architecture	++
Adaptable UI	++
Connectivity for Backend Data	++
Licensing Support	++

7 Functional Specification: Personal Mobility Assistant

7.1 Application Runtime Environment

7.1.1 Overall Functional Specification

The very heart of SIMPLI-CITY is to provide a next generation service platform for building the road user information system of the future. The concept of SIMPLI-CITY foresees that apps are lightweight by design. This means that the main logic should be performed by services. Services will be located at the server side and are managed by the Service Runtime Environment. They are consumed by apps running in a software on a mobile device – the Personal Mobility Assistant.

The Application Runtime Environment component provides an environment for apps running on the mobile device, analogous to the SIMPLI-CITY Service Runtime Environment, which provides a host for services - the backend functionality of apps. The execution of apps includes the local operations on the user's device as well as the interaction with the services in the backend via the Service Runtime Environment. It also makes use of the Local Key Storage provided by the Cloud-based Information Infrastructure of SIMPLI-CITY to store App- and User Data in order to restore a session after a connection timeout or a device change. The SIMPLI-CITY Application Runtime Environment also brings the technological foundation for apps to be deployed on different end user devices.

Succinctly, the Application Runtime Environment will be acting as a controller for executing apps and it will provide an API with a set of core functionalities. These core functionalities will extend the API of the operating system with typical tasks needed by SIMPLI-CITY apps.

7.1.2 Subcomponents

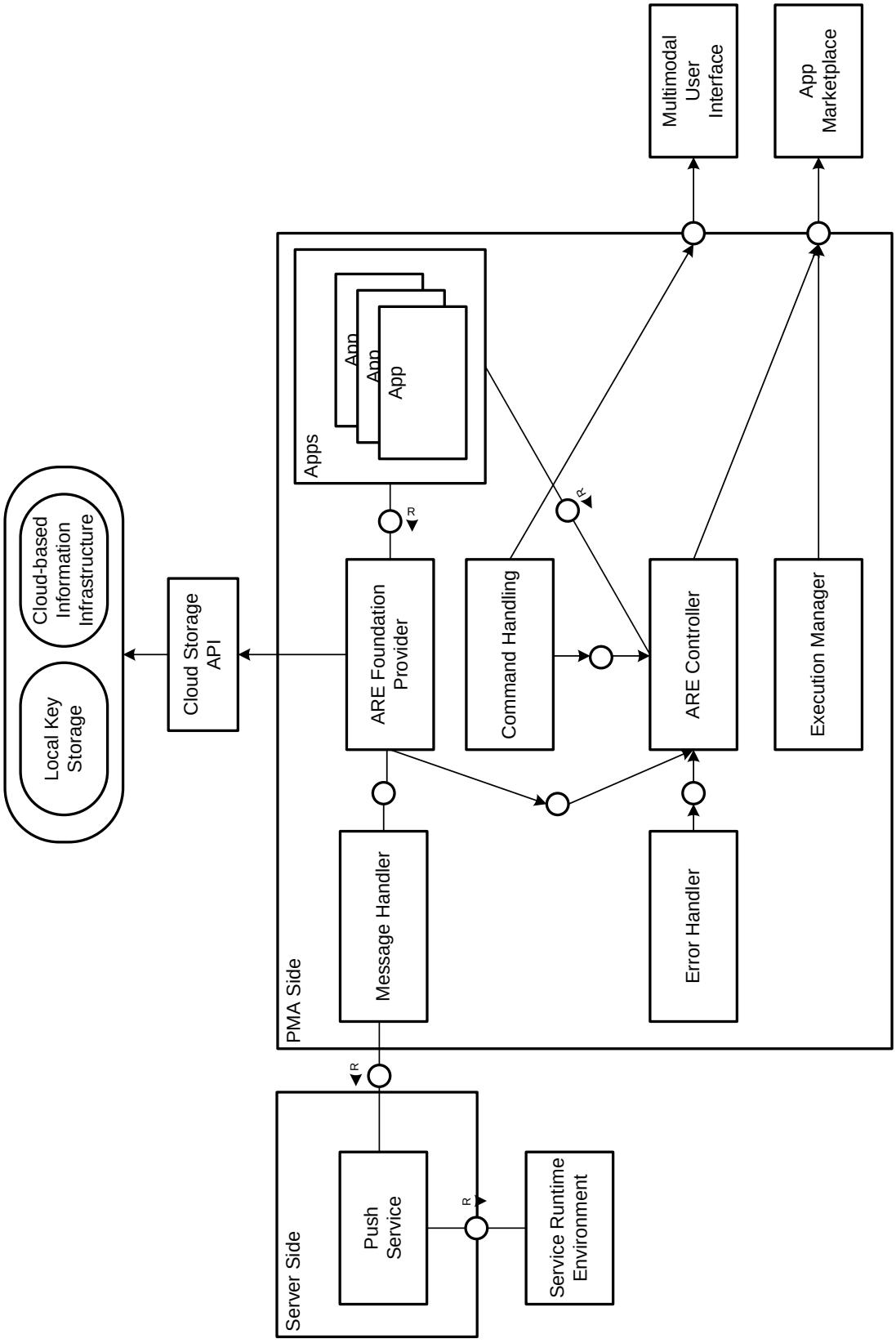

Figure 77: Application Runtime Environment Subcomponents and Interactions

To achieve the functionalities described in the previous subsection, this component provides the following subcomponents as depicted in the figure above:

- 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.
- ARE Foundation Provider: Provides an API with core commands to be used by apps. It is the main connection point to apps and also wraps the access to the Message Handler and the Local Key Storage and the Cloud-based Information Infrastructure (the latter with making use of the Cloud Storage API, see Section 5.2.2)
- Error Handler: Catches app exceptions and reports them to the App Marketplace for displaying them in the Author UI.
- Command Handler: Handles user commands and coordinates the dialogue handling calls
- ARE Controller: Controls the user interaction and cross-app communication facilities
- Execution Manager: Launches and manages apps during execution time
- Push Service (located at server side): This service allows internal backend services 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 may also be used to inform a specific app of a specific user (e.g. to inform the user that his/her train is late).

The Application Runtime Environment component is not used by other SIMPLI-CITY components except for the PMA-based Sensor Abstraction (as described in Section 7.3). Additionally, the Application Runtime Environment will directly and indirectly be used by apps as it will control and coordinate all apps during runtime and as it will allow them to invoke services by handling all communication facilities. Finally, the Service Runtime Environment may make use of the Push Service of this component to allow services to send messages to the Message Handler, which delivers it directly to the Application.

7.1.3 Related Requirements

Table 20: Requirements Related to the Application Runtime Environment

Requirement	Handled by Subcomponent	Comment
Must Have Requirements		
U185: Standardized messages between apps (P1)	Message Handler	Provide a unified message model for all apps running on the PMA. This model will be used for all communication between SIMPLI-CITY apps.
U186: Registry of installed SIMPLI-CITY apps (P1)	ARE Controller Execution Manager ARE Foundation Provider	Keep a registry of installed apps on the PMA which other SIMPLI-CITY components can access.

Requirement	Handled by Subcomponent	Comment
U193: Exchange of information from apps to server (P1) U194: Exchange of information from server to apps (P1)	Message Handler ARE Foundation Provider Push Service	Handles message transfers from the client to the server and vice versa. These functionalities make use of requirement U185 to have a mutual understanding of messages.
U72: Android support (P1)	All components	Provides the mobile environment for smartphones running the Android operating system. Will run in a native environment.
Should Have Requirements		
U48: App crashes are minimized (P2) U49: A crash should not impact other apps (P3) U103: Fault tolerance (P2) U104: Stability (P2)	Error Handler	Errors should only affect the current app inside the PMA and not affect other apps (e.g., music player crash should not stop the routing app). The error messages are logged and transmitted to the respective source (app or service) in the Author UI of T5.4.
U100: Backwards compatibility of apps (P2)	Command Handling	Provide guaranteed functionality of apps that are written for an older version of the PMA even on devices running a newer version of the PMA.
U208: Possibility to continue a previously started trip planning on the same device or different device (P4)	ARE Foundation Provider	Automatically save current Application Runtime Environment information periodically to be able to restore all information. Note: The part of switching between different devices will not be supported as listed below for U79.
Will not have for now		
U73: iOS support (P3) U74: Blackberry support (P5) U75: Windows phone support (P5) U76: Tablet support (P3)	N/A	Support for other widely used mobile devices and tablets. Within the project's lifetime, only Android-based systems will be supported as it is the goal to focus on creating one vertical prototype in the project with rich set of functionalities.

Requirement	Handled by Subcomponent	Comment
U79: Support to exchange devices (P4)	N/A	Keep a copy of a user profile in the cloud to have the same apps and settings on multiple devices. Profiles should only be synchronized if there's a stable and fast connection available (e.g. wireless or LTE). Support of switching the device is not planned for the prototype implementation as it is assumed that the situation does not appear often, is hard to implement and has a low priority in the requirements

7.1.4 Interaction with other Components

7.1.4.1 Interaction with the Service Runtime Environment

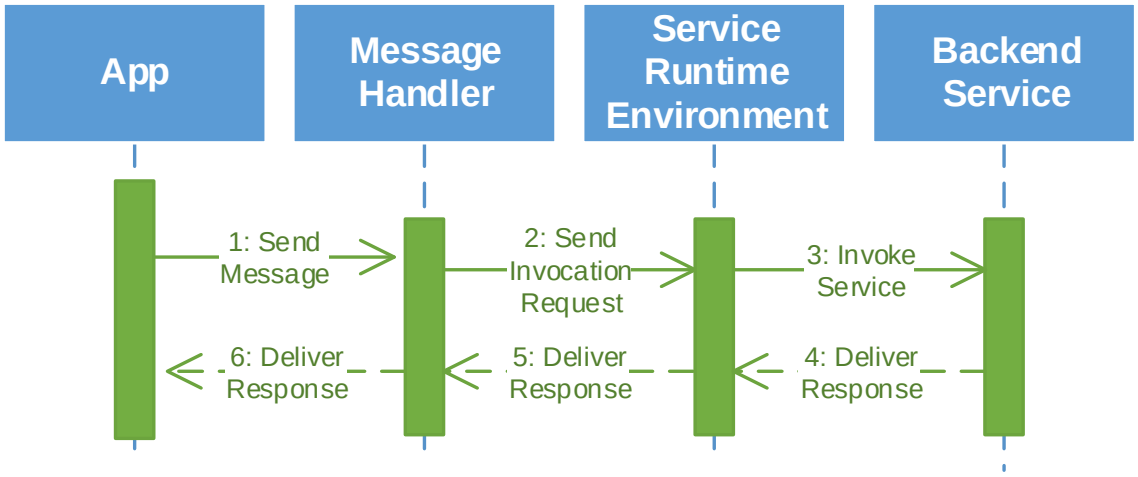

Figure 78: Interaction of the Application Runtime Environment and Backend Services via Request/Response Messages

Figure 78 shows a traditional way of receiving Messages from a particular backend service. An app sends a single message by sending it to the Message Handler, which then invokes the Service Runtime Environment to deliver the message. The App then awaits a message containing an acknowledge signal, which is used to confirm that the message was successfully received at the destination. If no acknowledge signal is received by the sender service/app there will be an automatic repetition of the service call (not depicted) to re-send the original message until the recipient confirms that the message was received correctly.

In some cases, backend services may return values to the app (i.e., more than an acknowledgement of the service call). For example, a parking service may return the geo location of the nearest parking space. In those cases, the response will contain the return value of the service, which will in most cases be a JSON or XML structure (to be defined in deliverable D3.2.2).

Alternatively, internal backend services may notify apps using the Push Service of the server side. In those cases, the Push Service will deliver the push messages to the Application Runtime Environment.

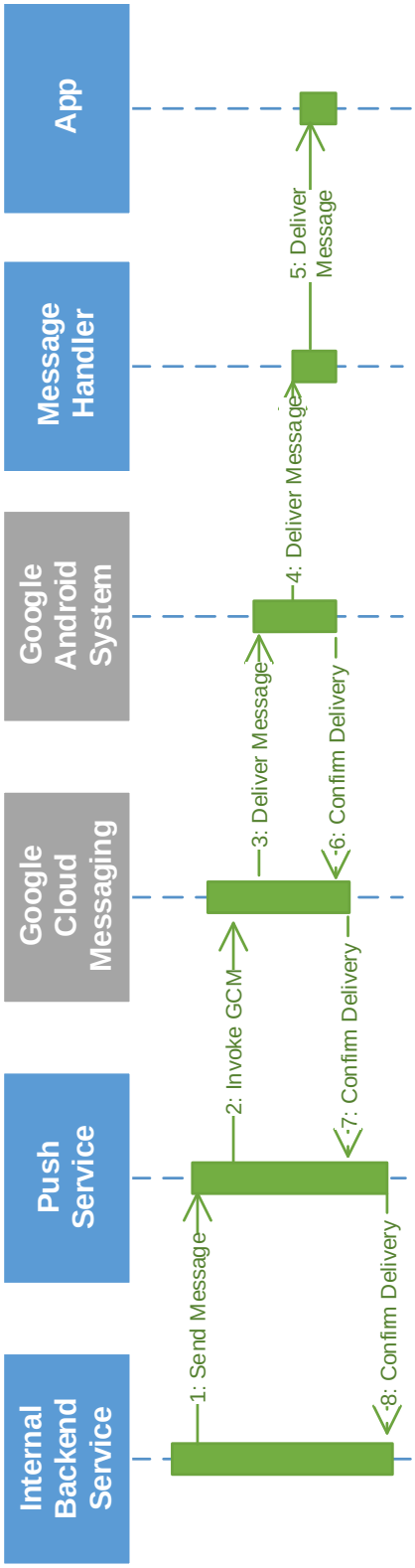

Figure 79: Interaction of the Push Service (Server Side) with an App

The Application Runtime Environment allows internal Backend Services to make use of a notification functionality to inform Apps about events. For this purpose, the Application Runtime Environment provides a server side subcomponent called the Push Service. This subcomponent provides a message interface for internal backend services as depicted in Figure 79. It may be used to inform app installations of all users, e.g. to inform all users of the “Barcelona Parking App” (broadcast) and it may be used to send a message to an app of one specific user (individual push notification). For the purpose of notifying apps, the Push Service manages a list of app subscriptions which is described in Section 7.1.4.1 allowing apps to actively subscribe to events.

As depicted in Figure 79, push messages are directly sent to the Message Handler subcomponent. This is technically handled by the Google Android base system with its integrated push functionality via the Google Cloud Messaging (GCM). For the purpose of clarity this has been depicted in the sequence diagram in grey colour even though this is outside SIMPLI-CITY.

Afterwards, the Message Handler analyses the message content which consists of two elements: An app ID for identifying the app which should receive the message, and a payload parameter containing XML or JSON content (to be defined in D3.2.2) for the App to interpret. Note that the Push Service will support the confirmation of the delivery of messages to the PMA but not the usage of it inside the App.

7.1.4.2 Interaction with the Cloud-based Information Infrastructure

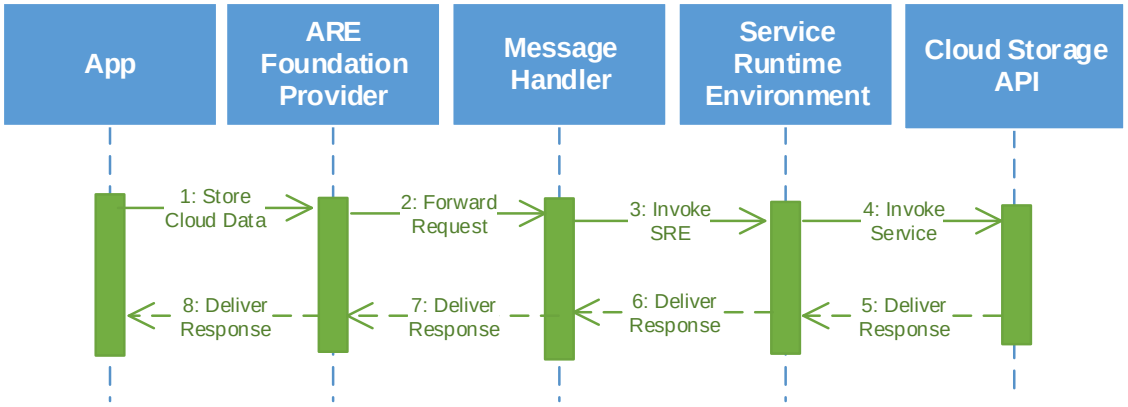

Figure 80: Interaction of the Application Runtime Environment and the Cloud Storage API

Apps may use the Application Runtime Environment to store data in the cloud as depicted in Figure 80. For this purpose, the Cloud Storage API will be invoked. This is essentially a normal service call via the SRE but it is encapsulated from the ARE Foundation Provider for easing the usage. The App therefore calls the ARE Foundation Provider which then calls the Service Runtime Environment via the Message Handler. This process is used to store all cloud-based data. For example, the Error Handler uses it to deliver error reports to the cloud storage which can later be viewed in the Web UI of the Marketplace by the app developer (Author UI, see Section 6.5). Obviously, the same interface can be used to not only store but also to receive data or to make use of all other functionality of the Cloud Storage API.

In some cases, it is not necessary or not wanted to store data in the cloud. In those cases a local storage is used, which is technically provided as a PMA side of the Cloud-based Information Infrastructure as the so called “Local Key Storage”. The Application Runtime

Environment will allow apps to make use of this Local Key Storage for storing key-value entries such as local settings. Figure 81 shows this process. The App therefore calls the ARE Foundation Provider which then calls the Local Key Storage via the Message Handler. Obviously, the same interface can be used to not only store but also to receive data or to update and delete it.

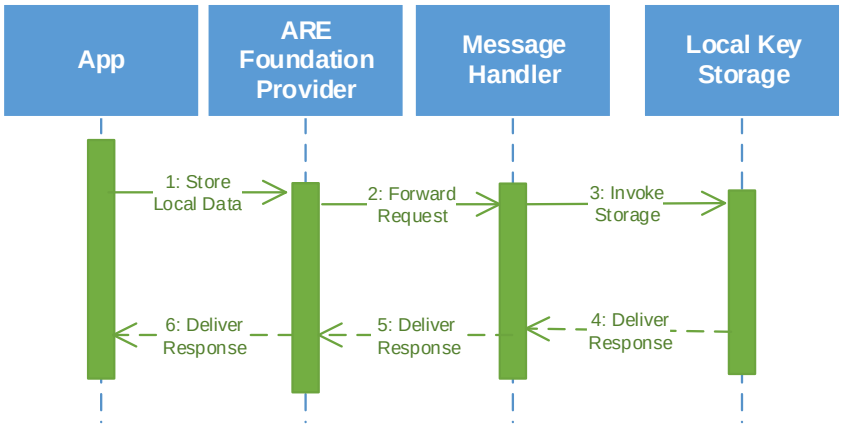

Figure 81: Interaction of the Application Runtime Environment and the Local Key Storage

7.1.4.3 Interaction with Apps

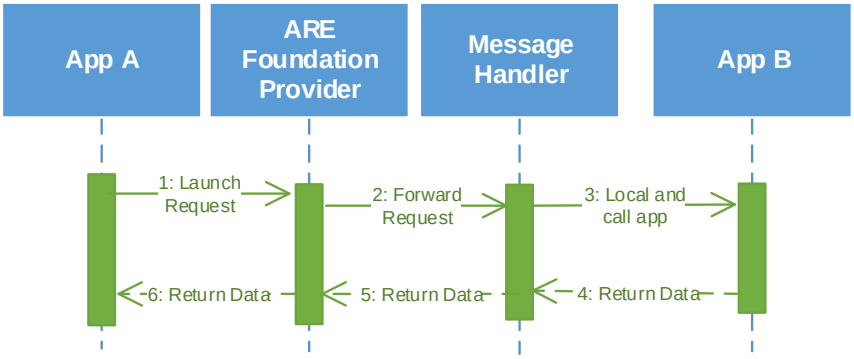

Figure 82: Interaction of the Application Runtime Environment and Apps

Apps may use the Application Runtime Environment to exchange information with each other. While Section 7.1.4.1 has shown the communication with Backend Services, Figure 82 shows the communication of apps with each other. In this example, App A wants to request data from App B. In this case, the ARE Foundation Provider is invoked which forwards the request to the Message handler. This request contains a UUID for identifying App B. The Message Handler forwards the request to App B and returns the response. For allowing the exchange of data, apps will have to implement an abstract class allowing the Message Handler to invoke a query method with a request that carries payload information for defining the request. The Application Runtime Environment uses the same Message Handler interface to also provide data exchange to the media playback methods (start/stop/play/etc) of the Media Data Streams and Data Prefetching Logic as depicted in Section 5.4. In this case, the Media Playback API will act as replacement of App B.

7.1.4.4 Interaction with the Media Data Streams and Data Prefetching Logic

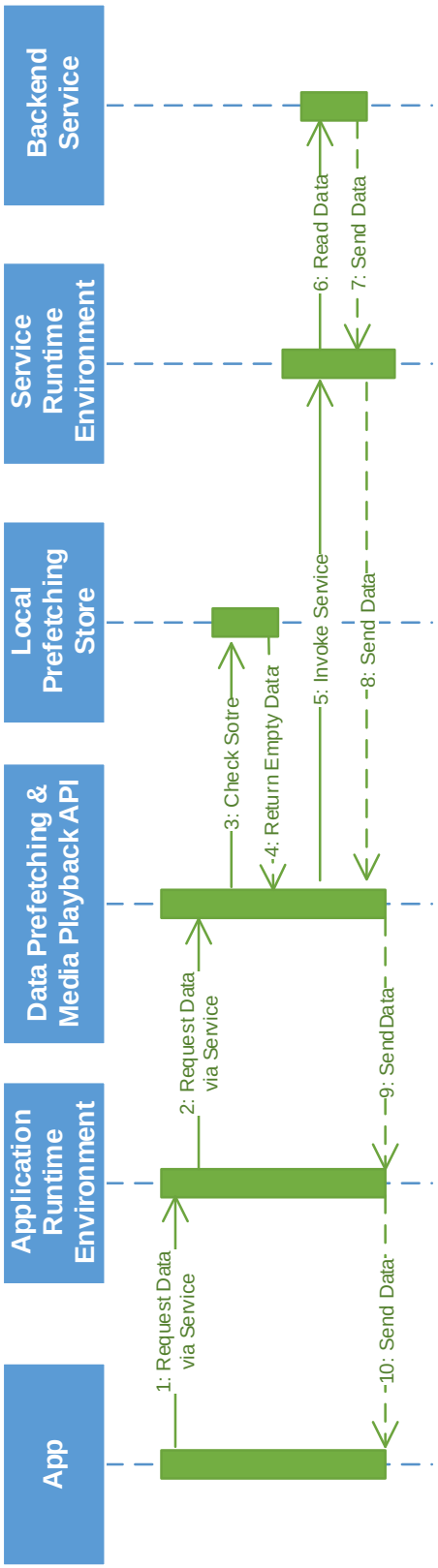

Figure 83: Interaction of the Application Runtime Environment and Data Prefetching & Media Playback API

The Data Prefetching & Media Playback API prefetches certain information by invoking services automatically as described in Section 5.4. From an app perspective this full process is handled invisibly via the Application Runtime Environment. Figure 83 shows this process graphically. Essentially, each app makes a “normal” service request whenever querying data. This data will in some cases be delivered from the Data Prefetching & Media Playback API via the Local Prefetching Store (if it has been prefetched) and in other cases it may come from the backend service via the Service Runtime Environment.

Please note that app developers may make use of an abstract class to mark, which data is prefetchable and which is not. If data is not marked as prefetchable, the Local Prefetching Store is not invoked at all and data is directly received from the Backend Service (see Figure 30).

7.1.4.5 Interaction with the Service and App Marketplaces

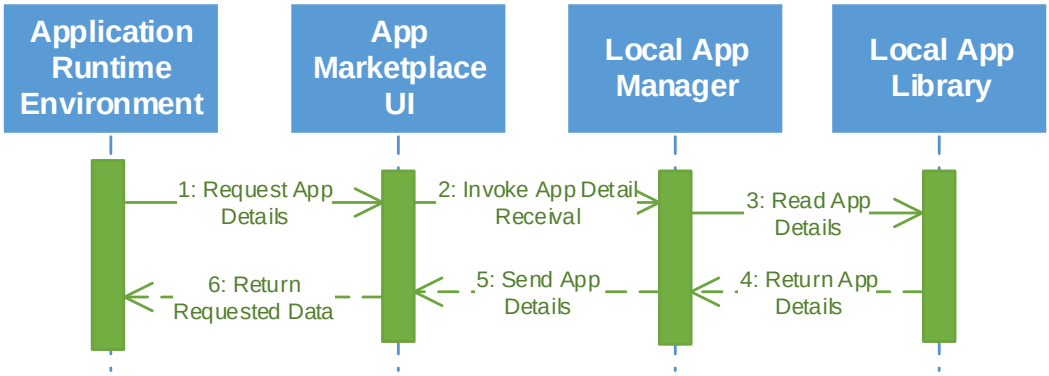

Figure 84: Interaction of the Application Runtime Environment and the App Marketplace UI

The Application Runtime Environment may in some cases need to receive information about a specific app such as the manifest file. This is performed by receiving information from the Local App Library as depicted in Figure 84. To achieve this, the Application Runtime Environment invokes the App Marketplace UI to request details from the Local App Manager, which has access to the Local App Library. The requested information is then forwarded to the Application Runtime Environment via the Local App Manager and the App Marketplace UI. Please note that the App Marketplace UI therefore not only contains a UI itself but also acts as a unique entrance to the PMA part of the Service and App Marketplace component.

7.1.4.6 Interaction with the PMA-based Sensor Abstraction

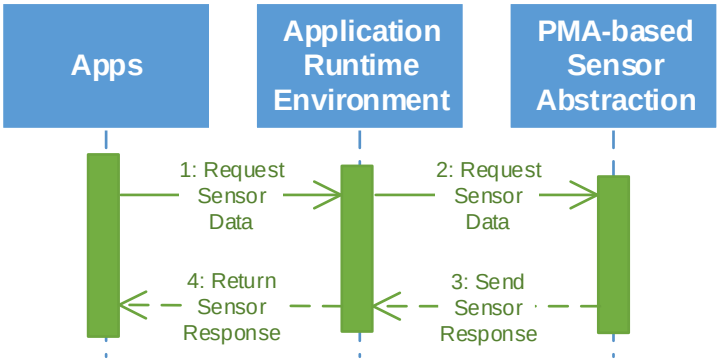

Figure 85: Interaction of the Application Runtime Environment and the PMA-based Sensor Abstraction

The Application Runtime Environment allows apps to access the functionality of the PMA-based Sensor Abstraction. Figure 85 shows this process graphically. To achieve this, the Application Runtime Environment acts as a gateway for apps to access the PMA-based Sensor Abstraction functionality. It provides an interface which apps may invoke and then forwards all requests to the PMA-based Sensor Abstraction which delivers the response to the apps via the Application Runtime Environment. Within this process, the PMA-based Sensor Abstraction queries the local sensors as described in Section 7.3 (not shown in Figure 85).

7.1.4.7 Interaction with the Multimodal Dialogue Interface

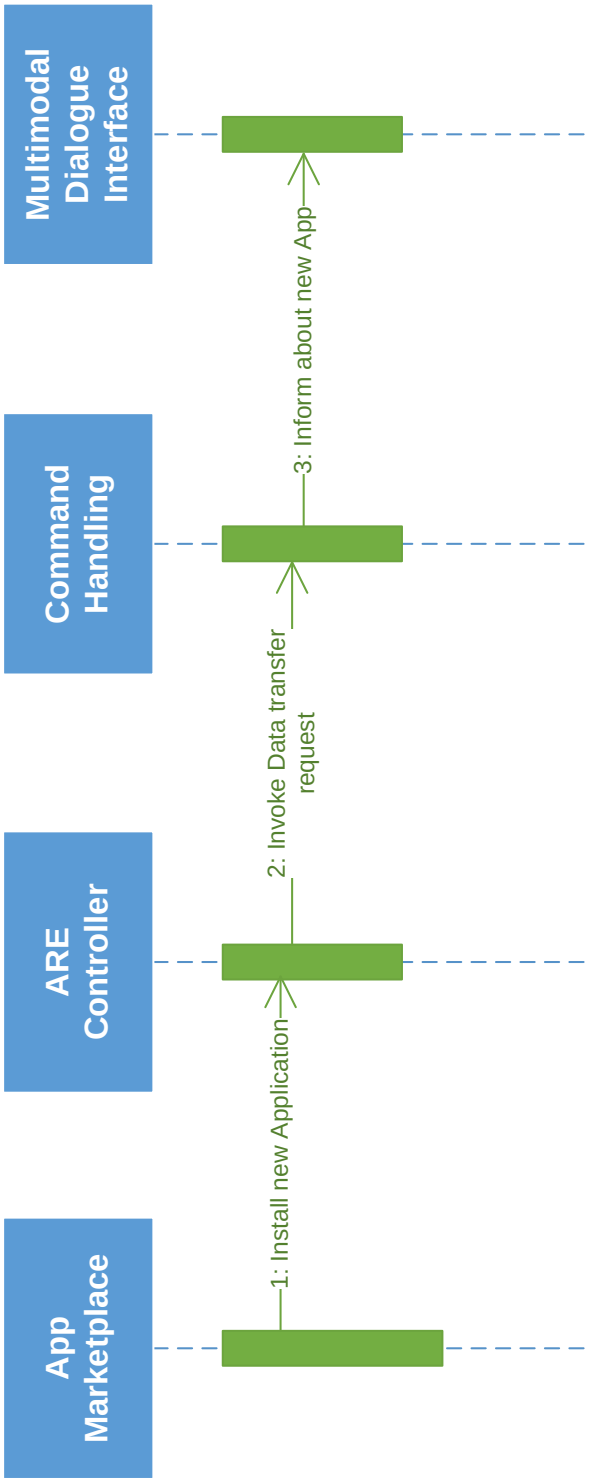

Figure 86: Interaction of the Application Runtime Environment and the Multimodal Dialogue Interface to Inform it about New Apps

As soon as a new App is installed on the PMA the ARE Controller notifies the Multimodal Dialogue Interface (via the Command Handling) about it (see Figure 86). After the Multimodal Dialogue Interface is invoked, further interaction is handled by internal processes that are described in Section 7.2.4.

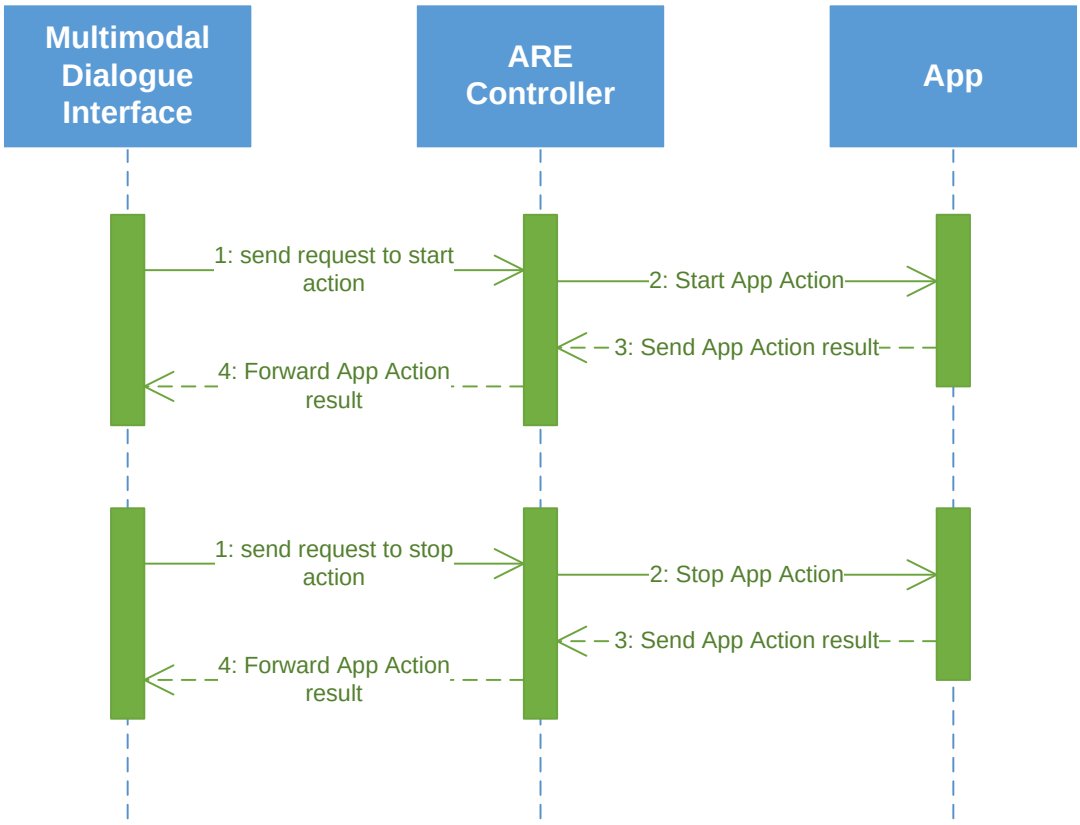

Figure 87: Interaction of the Application Runtime Environment and the Multimodal Dialogue Interface to inform it about New Apps

Figure 87 shows the interaction in order to start or stop an "app action" (see Section 7.2.4.2): The Multimodal Dialogue Interface invokes the Application Runtime Environment as soon as an App Action has to be started or stopped. This request is forwarded by the ARE Controller that handles the runtime management of the Apps. If the Multimodal Dialogue Interface wants an app to stop an action the interaction is the same, since the Application Runtime Environment is invoked for all controlling functionality with apps.

7.1.5 User Interface

The Application Runtime Environment does not feature a UI, as the interaction with users is done through the Multimodal Dialogue Interface (see Section 7.2).

7.1.6 Conceptual Data Model

The Application Runtime Environment heavily relies on the unified message model, which is used to provide a global model for inter-app messaging. This unified message model will contain information about the source of the message, the destination, the type of the message (e.g. error, notice, data update, warning, status updates), the value of the message (e.g. plain text, true/false, numbers), a checksum of the message and a set of dynamic parameters that are defined in the specific implementation of the message. The specific implementation of a message is done by the app and service developer.

The push messages need a certain structure to help the Service Runtime Environment to find the correct receiver. This structure could include the name of the receiving app or sub-component, the ID of the device (if the message is person-specific) and the payload.

7.1.7 Parameters to Take into Account for Technical Specification

Table 21: Criteria for Technical Specification

Parameter	Importance (--, -, -+, +, ++)
Generic Criteria	
Up-to-Datedness	+
Stability	+
Extensibility & Open Source/Standards	-+
Familiarity	+
Performance	-+
Interoperability	++
Specific Criteria	
Inter-App Communication	++
Push Support	+
Pull Support	++
Error Handling Support	++

7.2 Multimodal Dialogue Interface

7.2.1 Overall Functional Specification

This component is the user interface layer of SIMPLI-CITY, taking user input in the form of utterances managing the need for further user input, and transforming them into app calls. The result of the app call is then fed back to the user, using speech. User input is collected using an Automatic Speech Recognition (ASR) subcomponent. The speech input is interpreted to dialogue "moves" (dialogue acts, such as *ask*, *answer*, *and request*) with some semantic content. The dialogue component can also be triggered by app activity, and can fetch input data from the app instead of asking the user if suitable.

The dialogue moves are forwarded to the central subcomponent Dialogue Move Engine (DME). The DME contains a description of the dialogue context, including information about recent utterances, recent questions, active apps etc., and makes decisions about the system's next move based on this information. The next move can be to "do nothing", to "ask a question", "answer a question" or to "carry out an app call". Any resulting dialogue move output from the DME will be sent to a Generate subcomponent, which will generate GUI contents and an utterance, respectively. The final step in the iteration is to do the utterance, which is done using the Text-To-Speech (TTS) subcomponent. A Turn Manager subcomponent is used to manage the right and opportunity to speak during the dialogue (to make an utterance in a dialogue is often referred to as a "turn"). The Turn Manager distributes the turn between the user and the system.

Note: In contrast to all other SIMPLI-CITY components, there will be no technical selection for the Dialogue Interface and the Multimodal User Interface. Naturally, this is a result of the fact that Talkamatic is one of the technologically leading providers of voice-based and multimodal user interfaces for usage in cars in Europe and has therefore been chosen to become a part of the SIMPLI-CITY project consortium. Hence, Talkamatic's own software products will be the foundation for the Dialogue Interface and the Multimodal User Interface, but they will be extended and adapted in order to suit the purposes of SIMPLI-CITY.

As a result, the following definition of the subcomponents is already more final than it is the case for the other components, where due to technical aspects and technology and software selection, changes could be done during the Technical Specification (deliverable D3.2.2).

7.2.2 Subcomponents

As depicted in Figure 88, the subcomponents of the Multimodal Dialogue Interface include both frontend components, which necessarily are running on the client machine (in SIMPLI-CITY: the mobile device), and backend components, running on a server. The server can be located in the cloud, on the client machine or elsewhere.

Frontend components:

- Automatic Speech Recognition (ASR): Wraps an ASR service for speech input, either local or in the cloud.
- Text-to-Speech: Wraps a TTS engine for speech output.

- GUI: Displays menus and feedback messages on the screen and receives GUI input (menu selections and information about pushed buttons). The GUI will mainly be text based, serving as a visual supplement to the TTS.
- ARE Connector: The interface to the apps, running in the Application Runtime Environment.

Backend components:

- Dialogue Move Engine: The central dialogue management component. It will use *dialogue plans* defined in Dialogue Domain Description in order to ask the right questions to the user and requesting the right data from the Application Runtime Environment. It is also responsible for starting and stopping activities in the Application Runtime Environment. All communication with the Application Runtime Environment is done via the ARE Connector.
- Interpret: Semantic interpretation of the user utterance recognised by the ASR
- Generate: Generation of utterances from semantic representations
- InterpretGUI: Semantic interpretation of user GUI input
- GenerateGUI: Generation of XML screen descriptions from semantic representations
- Turn Manager: Distributes the “turn” (the right and opportunity to speak) between user and the system.
- Dialogue Domain Description: Describes the ontology, the plans and the grammar of a particular dialogue domain. The ontology defines concepts, entities and actions that the user and the system may reference in questions, answers and requests. The dialogue plans describe how actions are carried out and how questions are answered – and also describe what information is needed in order to carry out the actions or answer the questions. The grammar defines mappings between linguistic surface forms and semantic entities.
  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 stored in the Application Marketplace.
- Dialogue Domain Manager: Loads and provides access to Dialogue Domain Descriptions.

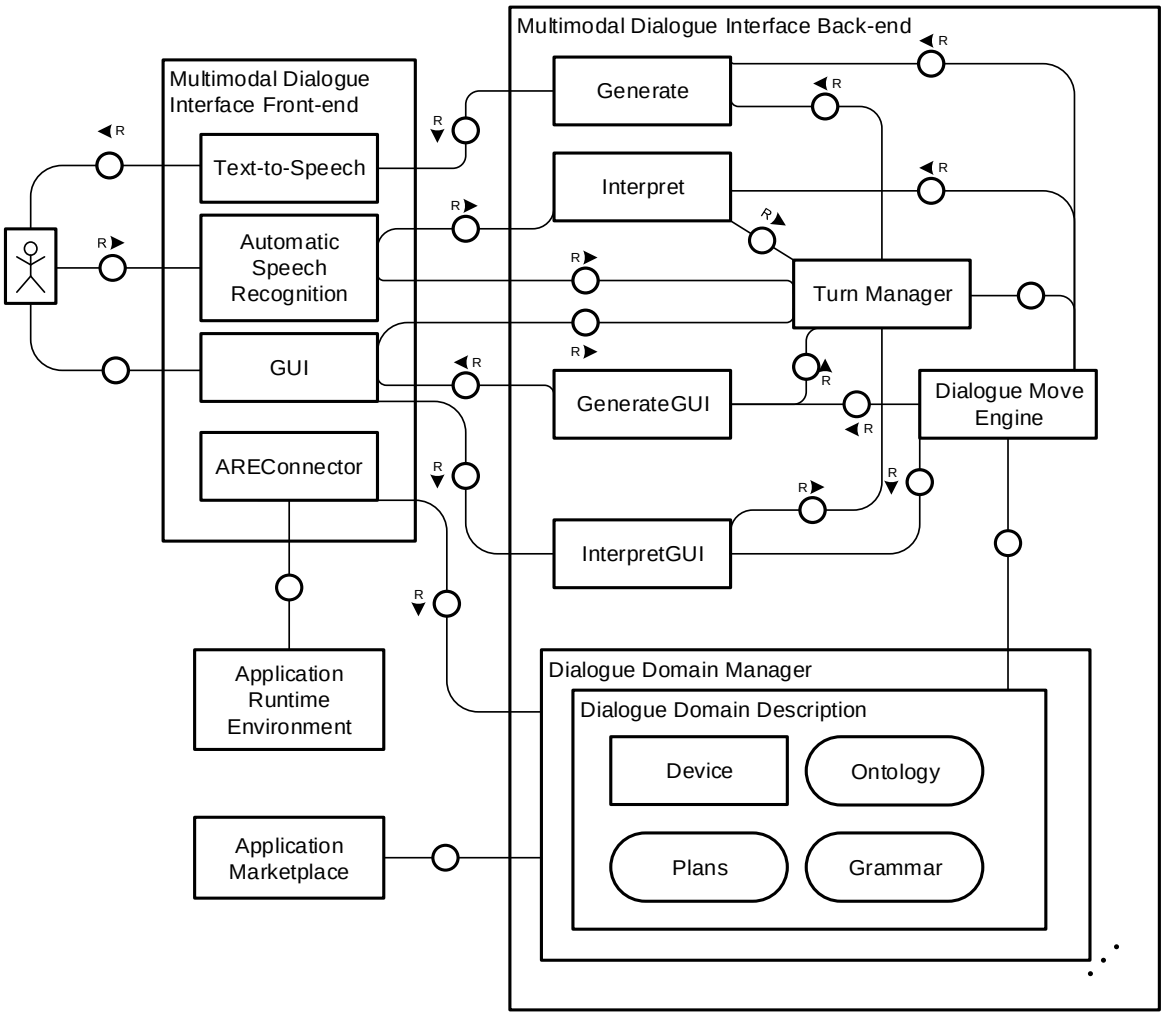

Figure 88: Subcomponents and Interactions of the Multimodal Dialogue Interface

The Multimodal Dialogue Interface interacts with other SIMPLI-CITY components:

- The Application Runtime Environment is accessed from the AREConnector in the frontend. The Application Runtime Environment can also post notifications about started and ended activities via the Device agent.
- The Application Marketplace is accessed from the Dialogue Domain Description in order to collect grammar, ontology and plans for specific apps. This is done every time a new app is added to the Application Runtime Environment.

7.2.2.1 Interaction from User Utterance to DME Processing

Figure 89 is a simplified description of what happens in the Multimodal Dialogue interface when the user has said something.

When the user speaks, the ASR module notifies the Turn Manager and the GUI that a recognition has started (the Turn Manager will notify other subcomponents which need this information). As soon as there is a recognition result, the Turn Manager and the GUI are notified about this. The GUI needs the information in order to signal the listening state to the user, while the Turn Manager needs the information to correctly model the turn state. The recognition result is passed on to the Interpret module, which provides a semantic interpretation for all of the hypotheses in the recognition result. The interpretation is passed back to the Turn Manager, which feeds the interpretation to the DME, which

performs context-dependent disambiguation, interpretation and processing of the user's utterance.

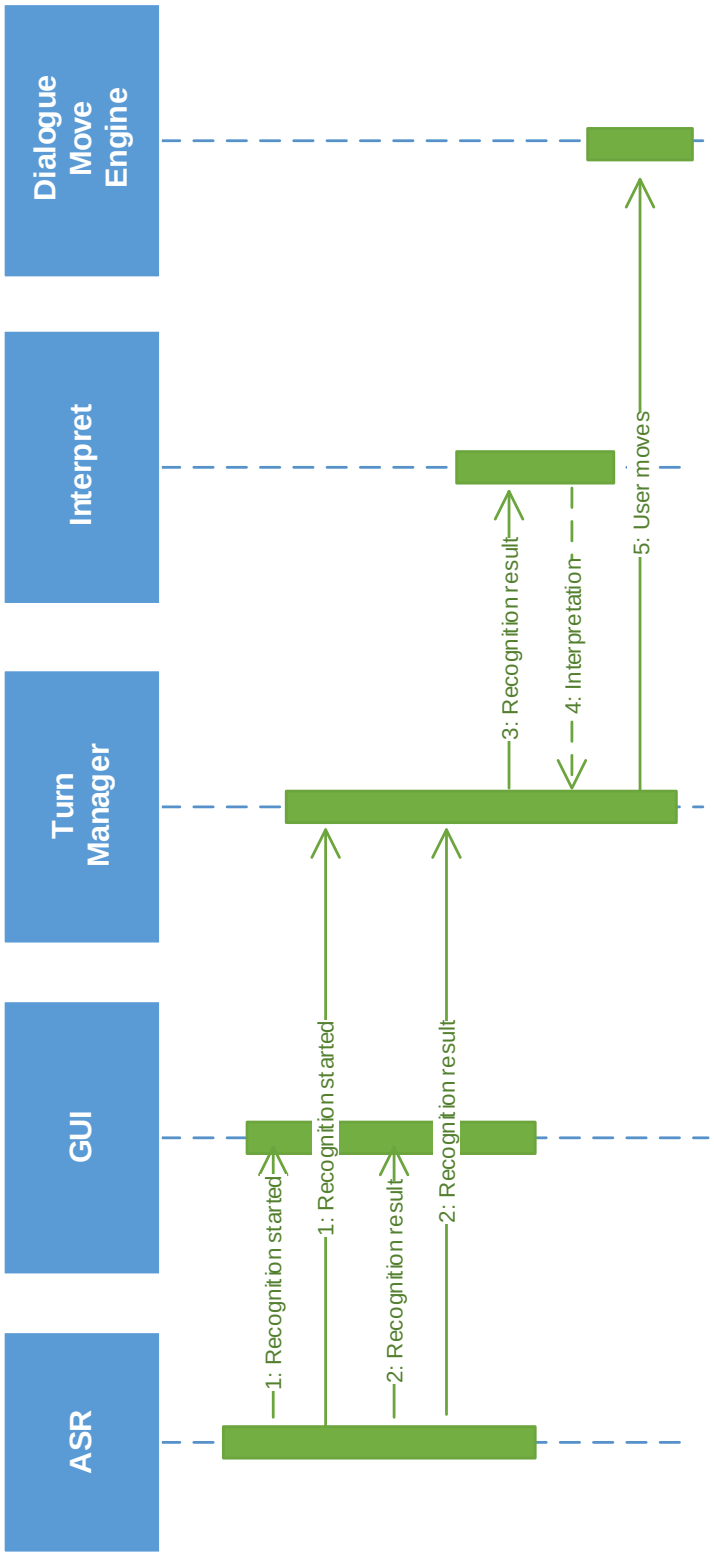

Figure 89: Interaction from User Utterance to DME Processing

7.2.2.2 Interaction from DME Processing to System Utterance

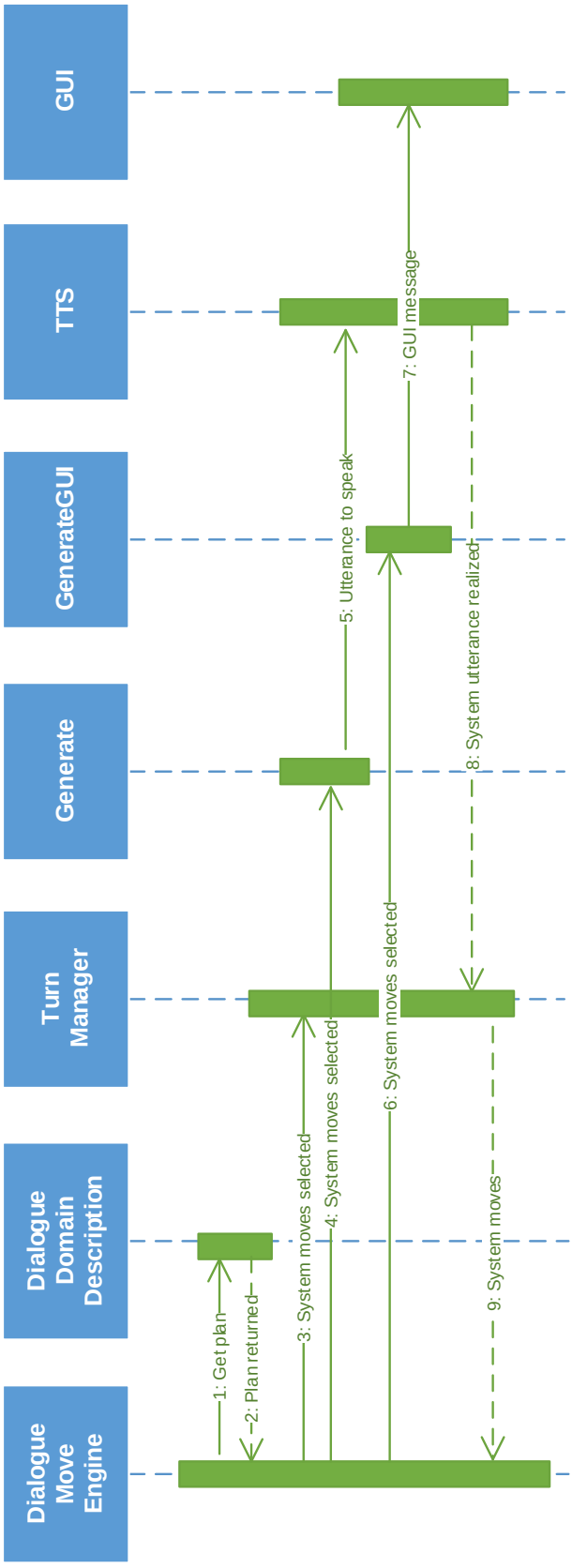

Figure 90: Interaction from DME Processing to System Utterance

Figure 90 shows a simplified description of what happens in the Multimodal Dialogue Interface when the system is processing a dialogue move from the user and responds by requesting more information from the user.

Firstly, the DME identifies a plan from the Dialogue Domain Description which corresponds to the user move. The DME selects a suitable response to the user moves, and signals this to the Turn Manager and to Generate and GenerateGUI using the “System Moves Selected” message. The Generate module transforms the semantic representation of the system moves into an utterance in natural language, which is passed on to the TTS engine. The TTS engine speaks the utterance. The GenerateGUI component passes an XML description of the system moves to the GUI module, which renders the description as a screen and displays it to the user. When the TTS has finished the utterance, it signals to the turn manager that the utterance is completed. The turn manager passes back the system moves to the DME, which integrates them into the dialogue context. This integration may in some contexts lead to a second round of system moves.

7.2.2.3 Interaction from DME Processing to GUI Menu

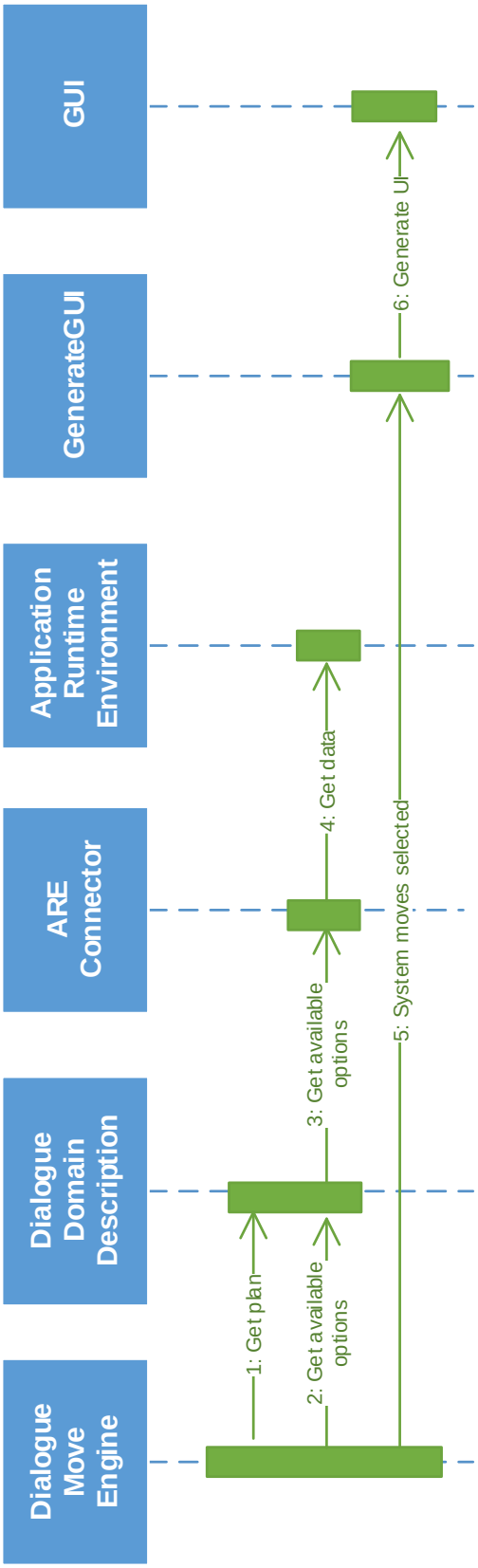

Figure 91: Interaction from DME Processing to GUI Menu

Figure 91 shows a simplified description of what happens in the Multimodal Dialogue interface when the system processes a dialogue move from the user and responds by requesting more information from the user, but focuses on the GUI screen generation.

Firstly, the DME identifies a plan from the Dialogue Domain Description which corresponds to the user move. The DME selects a suitable response to the user moves (in this case a wh-question[3]). It also asks the Dialogue Domain Description for all possible answers to the question. The Dialogue Domain Description forwards this call to the Application Runtime Environment, which in turn collects these possible answers to the question from the app. These answers are sent to the GenerateGUI module along with the selected question using the "System Moves Selected" message. The GenerateGUI module transforms the data into an XML description of the current question, and the possible answer alternatives. The GUI receives this information embedded in a GUI Message. The result of this interaction can be seen in the figure below.

The same functionality (from the DME and down to the "get data" message) will be used when the plan has the goal to answer a user question, although the generation of the GUI information will be somewhat different.

7.2.3 Related Requirements

Table 22: Requirements Related to the Multimodal Dialogue Interface

Requirement	Handled by Subcomponent	Comment
Must Have Requirements		
U36: Natural speech recognition (P1)	Dialogue Move Engine Automatic Speech Recognition	Automatic Speech Recognition coverage should be large enough to allow for a multitude of user expressions. Interpretation should be robust in order to handle variations.
U41: Input interactions with system via multimodal UI: on screen, voice control, and non-voice control (P1)	Automatic Speech Recognition Interpret GUI InterpretGUI Turn Manager Dialogue Move Engine	The system must hear, interpret, understand and act on haptic and spoken user input on the actual platform.
U42: Output interaction from system through UI (P1)	Text-to-Speech Generate Turn Manager Dialogue Move Engine GUI GenerateGUI	The system must be able to come up with dialogue moves to perform, generate utterances from them and do the actual utterances on the actual platform. It must also generate screen information from the moves and display the information on the actual platform.

[3] *Who, where, what, which, how.*

Requirement	Handled by Subcomponent	Comment
Should Have Requirements		
U18: Reasonable response time (P1)	All components	Latency must be kept at a minimum.
U43: Non-distracting interaction (P1)	Dialogue Move Engine Generate GenerateGUI	System output should be planned by the Dialogue Move Engine in order to be non-distracting. Also the generation of the messages should be done in a way that they are not distracting. The generation of the screen content should be done in a way that it is not distracting to the user. The information must be relevant to the user in the current context.
U45: Automotive quality voice recognition (automotive acoustic models for in car use) (P1)	Automatic Speech Recognition	If available, when in the car, an appropriate acoustic model should be used.
U46: High speech recognition rate (P1)	Automatic Speech Recognition Dialogue Move Engine Interpret	The Automatic Speech Recognition and Interpret subcomponents must have enough coverage to correctly identify the key concepts of the utterances.
Could Have Requirements		
U19: Minimum manual configuration (P1) U22: Personalization – Incremental configuration (P1)	Dialogue Move Engine SIMPLI-CITY Dialogue Domain Description	The system must be usable “out of the box”. Potential configuration should be done automatically, if possible, and incrementally.
U37: Result oriented instead of service/app recognition oriented (P1) U38: Disambiguation (P1) U39: Provision of a limited number of alternatives (P1)	Dialogue Move Engine	The Dialogue Move Engine must contain efficient procedures for selecting apps when more than one can be applied to solve the same user problem.
U40: System should have a friendly voice (P1) U44: Voice quality (P1)	Text-to-Speech	The voice quality must be good as this influences the perceived system performance.

Requirement	Handled by Subcomponent	Comment
U47: Voice interaction through the car audio system (microphone & loudspeakers) for hands free in car use (P1)	Automatic Speech Recognition	When used in a car, the system could use the in-car sound input/output resources.
Will not have for now		
U20: Multilingual (P5)	All components	All subcomponents are affected if multiple languages should be supported. While per se the Multimodal Dialogue Interface will allow to make use of different languages, the actual software prototypes will only allow (British) English. This could be extended during the commercialisation of the project results.
U21: Link voice commands to apps (P4)	SIMPLI-CITY Dialogue Domain Description	The system must allow end-users to link certain utterances of their choice to certain app functionalities.

7.2.4 Interaction with other Components

As can be seen in Figure 88, the Multimodal Dialogue Interface interacts with the Application Runtime Environment and the Application Marketplace. The following subsections will briefly explain the different kinds of interaction between the Application Runtime Environment and the Multimodal Dialogue Interface.

7.2.4.1 Interaction to Register an App

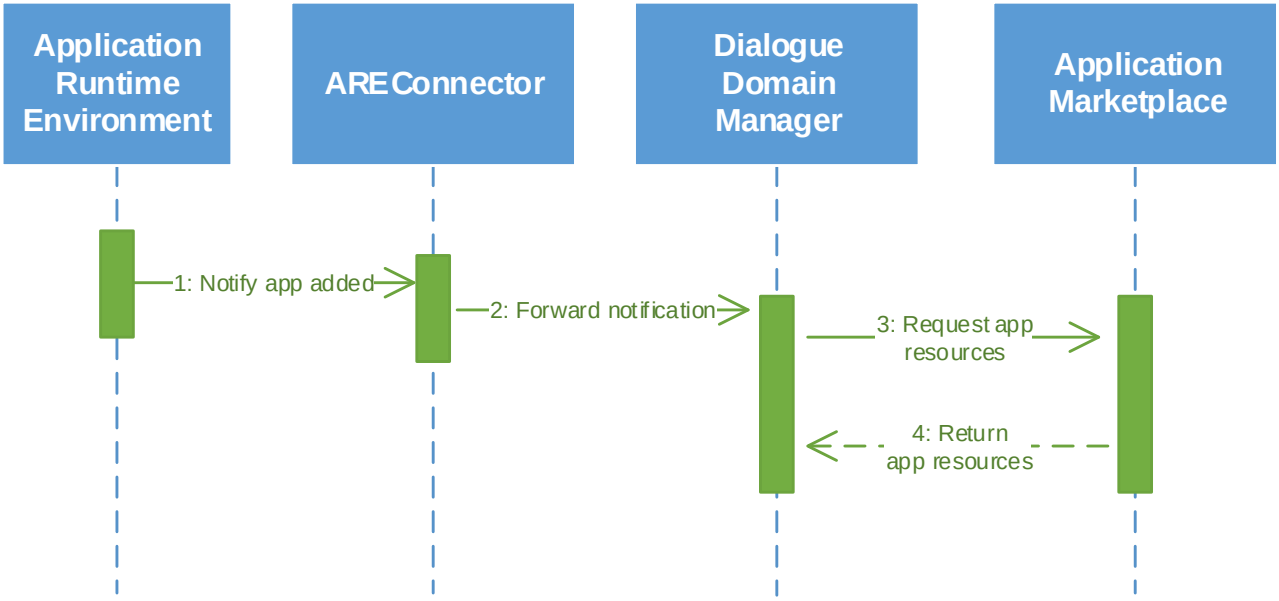

Figure 92: Interaction of Application Runtime Environment and Multimodal Dialogue Interface for the Registration of New Apps

Figure 92 shows the necessary interaction for the registration of a new app at the Multimodal Dialogue Interface. As soon as the user has installed an app from the marketplace, the Multimodal Dialogue Interface should be able to understand input from the user relating to the new app. For this reason, the Application Runtime Environment notifies the AREConnector in the Multimodal Dialogue Interface about the addition of each new app. Upon receiving such a notification, the AREConnector forwards the notification to the Dialogue Domain Manager, which requests the app's plans, grammar and ontology from the App Marketplace.

7.2.4.2 Interaction to Start/Stop Actions in the Application Runtime Environment

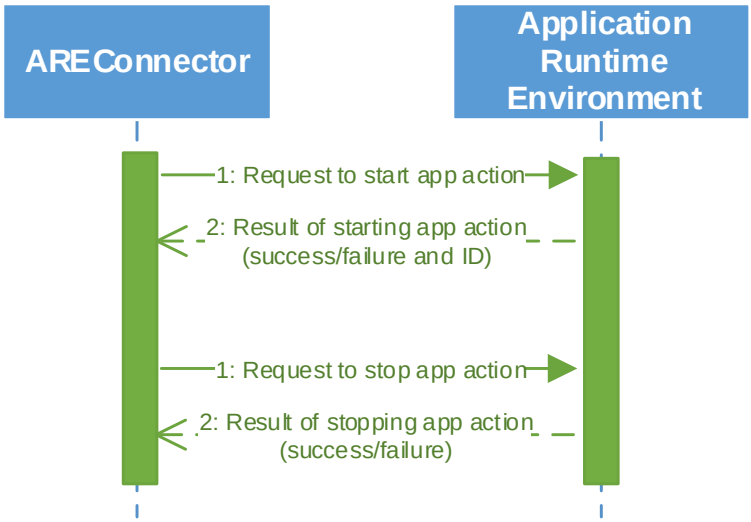

Figure 93: Interaction to Start and Stop Actions in the Application Runtime Environment

As depicted in Figure 93, when the user initiates an action, such as playing music or starting navigation, the ARE Connector in the Multimodal Dialogue Interface forwards the request to the Application Runtime Environment. This request contains the app ID, the name of the action, and parameters. In turn, the Application Runtime Environment returns the result of the request (e.g. success or failure).

Similarly, the ARE Connector forwards requests to stop an action, such as when the user requests to stop music or abort a transaction.

The call to start an action will return an identifier for the action, which can be used to refer to the action when the call to stop the action is issued. Note that the Application Runtime Environment can notify the Multimodal Dialogue Interface about started and stopped actions also in cases where the Multimodal Dialogue Interface did not request the action.

7.2.5 User Interface

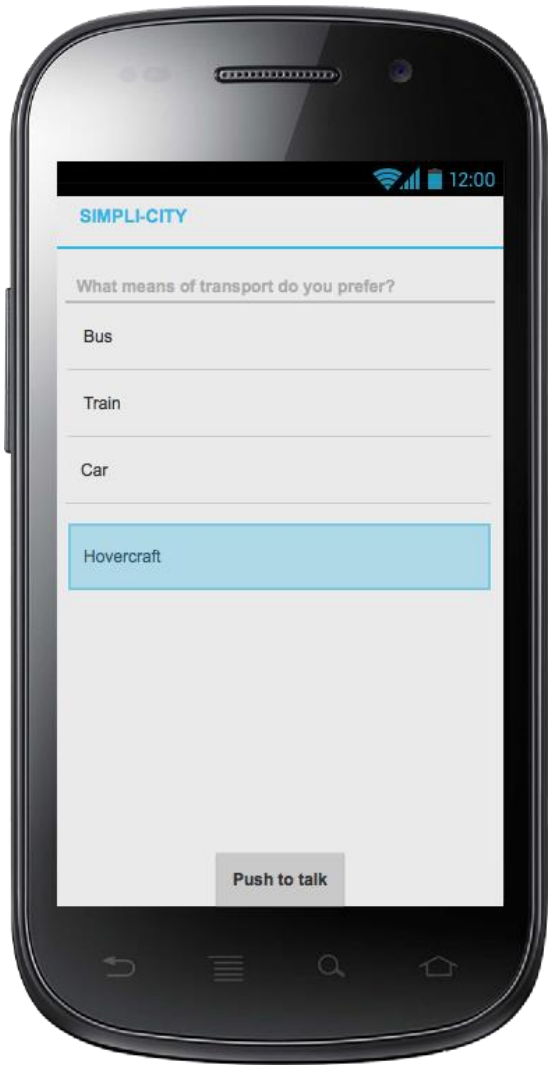

Figure 94: Question and Menu in Multimodal User Interface

The figure above shows the GUI of the Multimodal User Interface. As the system voice asks the question “What means of transport do you prefer?”, the GUI shows the same question as text, and also displays a menu of available answers. At this point in the interaction, the user can either

- Push the PTT button to give a spoken answer to the question (or any other utterance) or
- Select any of the available answers as an answer to the question.

Figure 95 shows the GUI in the situation where the system has done a recognition of the user utterance with a low confidence (either because the actual recognition was of low quality, or because the recognised utterance was unexpected given the current context). The system asks by voice if it has understood the user correctly, at the same time as it displays in the GUI the same question, giving the user the option to select yes or no. In this case the system was so unsure that it heard right that it will interpret the lack of an answer as a “No”, and re-raise the question about what means of transport the user prefers.

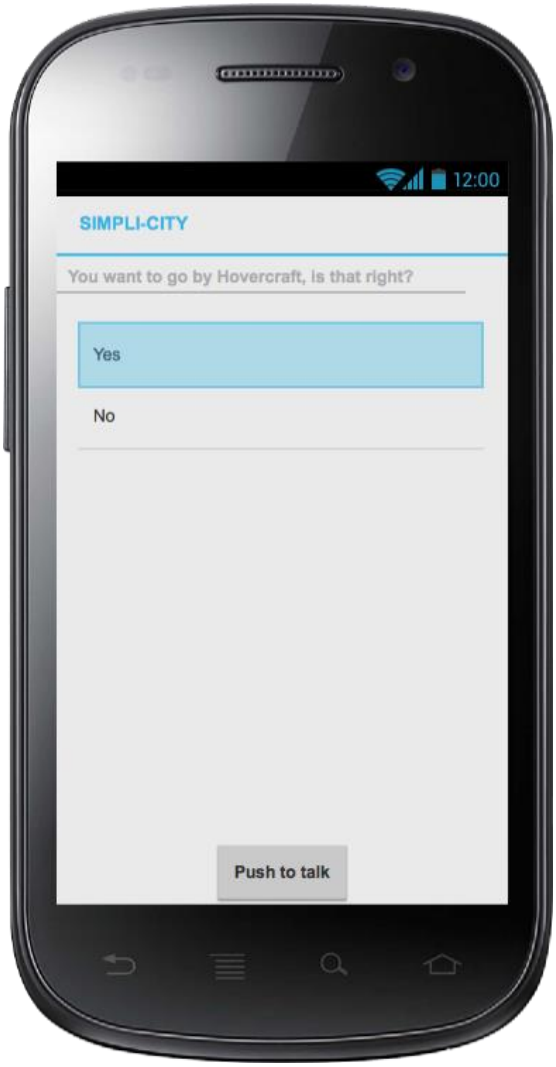

Figure 95: Grounding Sub-Dialogue in Multimodal User Interface

7.2.6 Data Model

This section describes the data received by the Multimodal Dialogue Interface from other components.

7.2.6.1 Grammar

The grammar defines the mapping between linguistic surface forms and semantic entities. It is contained in a text file where each line defines a particular mapping.

Example:

```
play_music = "play music"
stop_music = "stop music"
PlayMusic_started = "playing <answer:artist_to_play> by
<answer:song_to_play>."
PlayMusic_ended = "stopped playing."
StopMusic_failed_not_playing = "no music is playing."
```

7.2.6.2 Ontology

The ontology defines concepts, entities and actions that the user and the system may reference in questions, answers and requests. It is contained in a Python file, divided into separate sections for actions, predicates (concepts), individuals (entities) and sorts (domain-specific data types).

Example:

```
class MusicPlayerOntology:
  actions = set([
    "play_music",
    "stop_music",
  ])

  predicates = {
    "artist_to_play": "string",
    "song_to_play": "string",
  }

  individuals = {}

  sorts = {}
```

7.2.6.3 Dialogue Plans

The dialogue plans are contained in a Python file and describe how actions are carried out and how answers to users' questions are found. They also describe what information is needed in order to carry out the actions or to answer questions.

Example:

```
class MusicPlayer:
  plans = [
    {"goal": "perform(play_music)",
     "plan":
       ["findout(?X.artist_to_play(X))",
        "findout(?X.song_to_play(X))",
        "dev_perform(PlayMusic, MusicPlayerDevice)"]
    },

    {"goal": "perform(stop_music)",
     "plan":
       ["dev_perform(StopMusic, MusicPlayerDevice)"]
    }
  ]
```

7.2.7 Parameters to Take into Account for Technical Specification

This section describes the criteria for selecting the Automatic Speech Recognizer and the Text-To-Speech engine to be used in the SIMPLI-CITY project.

Table 23: Criteria for Technical Specification

Parameter	Importance (--, -, -+, +, ++)
Generic Criteria	
Up-to-Datedness	-+
Stability	++
Extensibility & Open Source/Standards	-+
Familiarity	+
Performance	++
Interoperability	-
Specific Criteria (max. 10)	
Realtime/Speed	++
Large Coverage	++
Cost	++
Quality	++
Platform Availability	++

7.3 PMA-based Sensor Abstraction

7.3.1 Overall Functional Specification

Sensor data, respectively sensor readings, of a variety of different sources provide a major data source within SIMPLI-CITY. These sensors and the corresponding sensor data usually are rather heterogeneous in both access schemes and data formats, for example ranging from built-in sensors like an accelerometer over virtual sensor data from personal calendar events to wirelessly connected sensors in the user's environment, respectively the car sensors.

In contrast to the Sensor Abstraction and Interoperability Interfaces described in Section 5.3, the PMA-based Sensor Abstraction deals with the direct interaction with built-in sensors of the PMA or directly connected sensors in the car. These sensors will also be available to apps if no data connection is available. The PMA-based Sensor Abstraction will also have a connection to the server-based Sensor Abstraction and Interoperability

Interfaces (see Section 5.3), to provide access to local sensors and user centric data to other SIMPLI-CITY server components for further data processing.

To provide this homogeneous access to built-in and directly connected devices in the local environment, e.g., the car, the PMA-based Sensor Abstraction component is developed. This component will provide the seamless integration of heterogeneous sensor sources and sensor readings on the PMA, e.g., by providing corresponding wrappers. In this context, the component will take care of providing pull-based and (event-based) push-based data access.

7.3.2 Subcomponents

To achieve the functionalities described in the previous subsection, the PMA-based Sensor Abstraction provides the following subcomponents as depicted in Figure 96:

- Sensor Wrapper: Includes:
  - Access Handler: Allows the access to the diverse sensor data.
  - Data Wrapper: Provides the wrapping facilities for transforming diverse proprietary sensor data to data (formats) usable by the other SIMPLI-CITY components, respectively SIMPLI-CITY apps.
- Data Subscriber: Takes care of sensor readings that are only available at specific events and buffers the values in the Data Buffer for later use.
- Sensor Abstraction Interface: Exposes the interfaces to other SIMPLI-CITY components, respectively apps in the Application Runtime Environment over which sensors and sensor data are accessed.

The PMA-based Sensor Abstraction is directly used by the apps running in the Application Runtime Environment and provides homogeneous access to all locally available sensors. In the following sequence diagrams as well as in Figure 96, the Application Runtime Environment is named as placeholder for all apps making use of direct access to the PMA-based Sensor Abstraction. Additionally this component provides the interfaces for the Sensor Abstraction and Interoperability Interfaces, described in Section 5.3, to directly access sensors and data connected to the PMA.

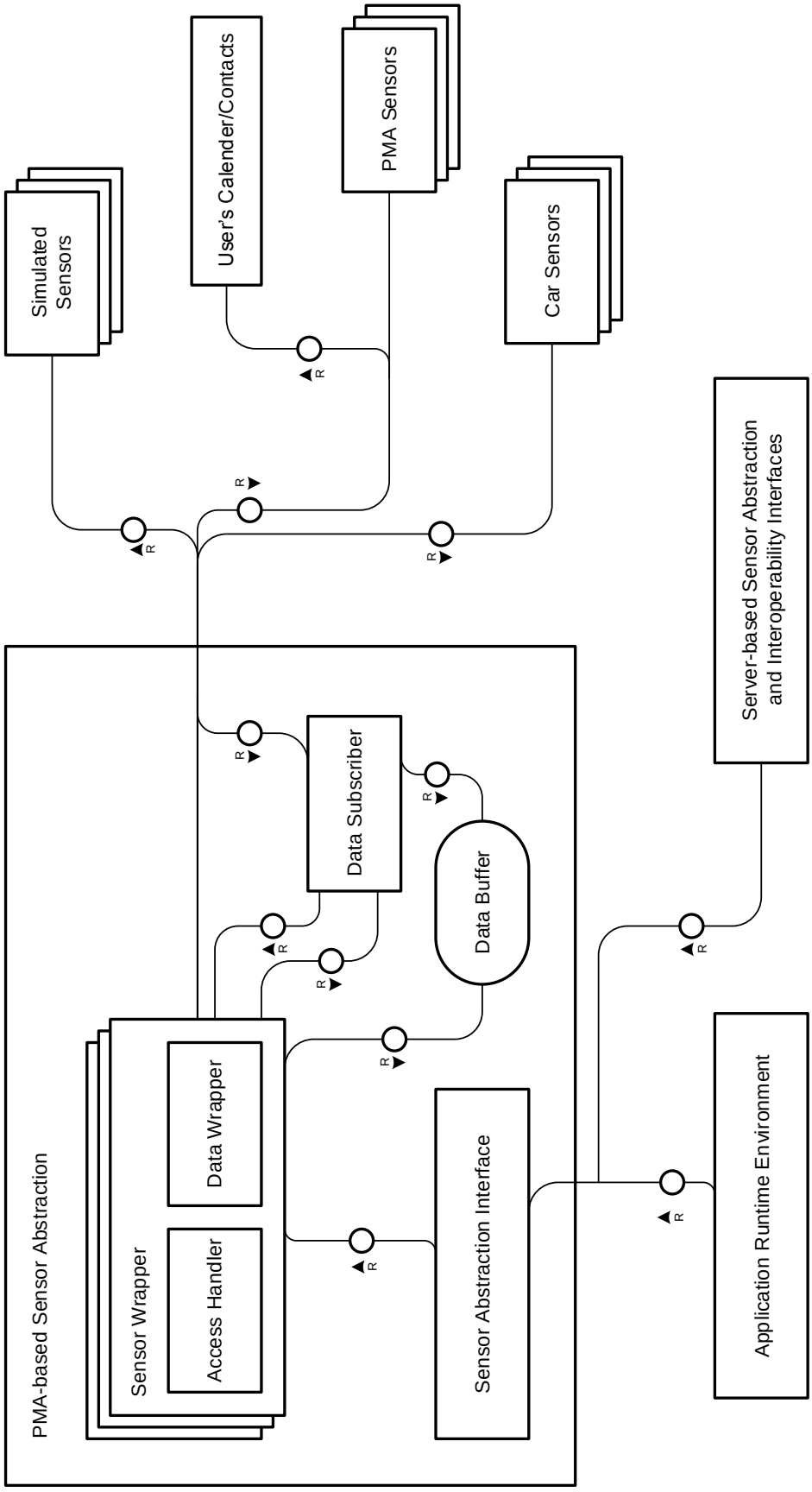

Figure 96: Sensor Abstraction Service on PMA and Interoperability Subcomponents and Interactions

7.3.3 Related Requirements

Table 24: Requirements Related to the PMA-based Sensor Abstraction

Requirement	Handled by Subcomponent	Comment
Must Have Requirements		
U85: Interaction with car sensors U107: Access to sensors of the vehicle U215: Vehicle information available to the system U33: Reaction to KPIs from the car, like level of fuel and kilometres done U202: Diagnosis of abnormal traffic condition in real-time U203: Prediction of abnormal traffic condition U204: Support for querying diagnosis historic U205: Support for querying impact factor on traffic condition	Sensor Wrapper Sensor Abstraction Interface	This requirement describes the functionality to connect to the car information infrastructure and access the car sensors. The sensor readings will be wrapped into the common data format and access will be provided via the Sensor Abstraction Interface running as background service on the PMA.
U108: Access to smart device sensors	Sensor Wrapper Sensor Abstraction Interface	This requirement describes the functionality to access sensor information of sensors built-in the PMA. The sensor readings will be wrapped into the common data format and access will be provided via the Sensor Abstraction Interface running as background service on the PMA.

Requirement	Handled by Subcomponent	Comment
U191: Simulation of sensors (P1)	Sensor Simulation	This requirement describes the need to provide developers with simulated sensors and simulated sensor data. This is necessary to decouple the development process for apps and services from the actual access to real sensors. This would significantly simplify the development, as it is expected that developers not necessarily have easy access to a real PMA connected to a car. Thus, by using the simulated sensors and simulated sensor data they can still develop their solutions without the requirement to access real sensors.
U189: Unified interface for accessing sensors (P1)	Sensor Abstraction Interface	This requirement describes the basic functionality of the PMBA-based Sensor Abstraction component of providing a unified access method to heterogeneous sensors for app and service developers. Thus, building on a unified data description and access model (see requirement U109) an interface has to be provided for app and service developers, which allows a unified access to sensors independent of their (heterogeneous) type. However, it has to be noted, that within this component only car sensors and PMA-based sensors are considered. All other sensor sources are addressed within the server-based Sensor Abstraction and Interoperability Component (see Section 5.3).
U190: Unified interface for accessing user centric data	Sensor Abstraction Interface Sensor Wrapper	This requirement describes the need to provide a unified access to user centric data, e.g., calendar or contact data available on the PMA. The Sensor Wrapper will provide the possibility to request user centric data that is available on the PMA and will provide a unified access via the PMA-based Sensor Abstraction.

Requirement	Handled by Subcomponent	Comment
Should Have Requirements		
U30: Reaction to who is in the car, via sensors	Sensor Abstraction Interface Sensor Wrapper	This requirement describes the need to provide information about the people in the car, e.g., the amount of passengers. By means of the sensors of the vehicle, the PMA-based Sensor Abstraction will provide via the Sensor Wrapper information from car sensors about passengers like the number of passengers or even the ID of the key, used to open the car.
U110: Remote control of car components, e.g., air conditioning, heating, battery charge timing	Sensor Wrapper Sensor Abstraction Interface	This requirement describes the need to remote control of some car components. For example, this can be the remote activation of the cars heating system before the user arrives at the car or the remote controlled charging of the batteries of electric cars depending on the current energy prices. The PMA-based Sensor Abstraction should provide a unified way to send predefined commands via the Sensor Wrapper to the vehicle. This command flow can be achieved by means of car APIs of vehicles.
U114: Configuration of the frequency of update of the data from data sources (P1)	Data Subscriber	This requirement describes the necessity of being able to manage data streams from sensor sources in the sense of listening to sensor reading events and buffering the corresponding sensor reading in the local data buffer. This is because some sensors provide their data only at specific events and are not accessible on demand. This can also be virtual sensor data triggered by a predefined event and generated by calculation out of other current sensor data.

Requirement	Handled by Subcomponent	Comment
U120: Handle data streams (P1)	Sensor Wrapper	This requirement describes the need for being able to handle sensor data coming in as streaming data. In this context, an analysis and interpretation of this streaming data might be necessary to manipulate and transform the incoming data to specific system needs.
Will not have for now		
U83: Interaction with head up display (P3)	Sensor Abstraction Interface	This requirement describes the functionality of exploiting cars' built-in displays to visualize information from the SIMPLI-CITY system, e.g., apps, to the driver. Therefore, access possibilities to post information to these built-in displays are required. However, there might be some restrictions for this active access to a car's built-in systems so that thus active communication might not be possible.

7.3.4 Interaction with other Components

7.3.4.1 Interaction with Sensors

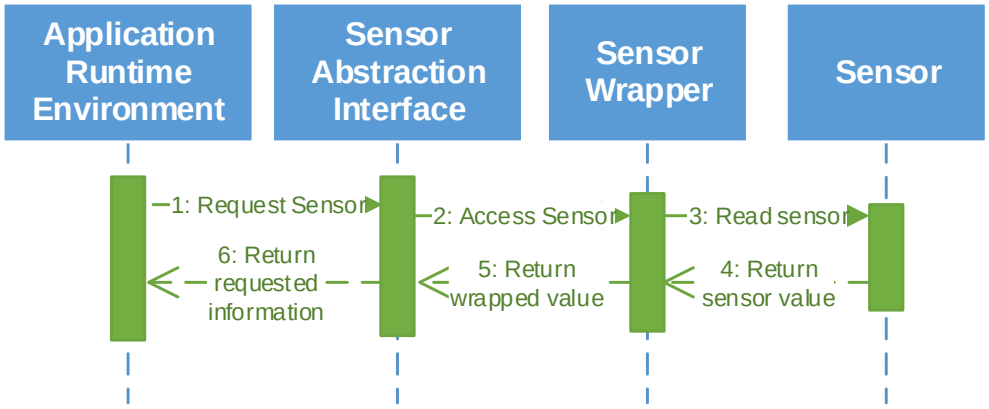

Figure 97: Interaction of Application Runtime Environment, PMA-based Sensor Abstraction and Sensors

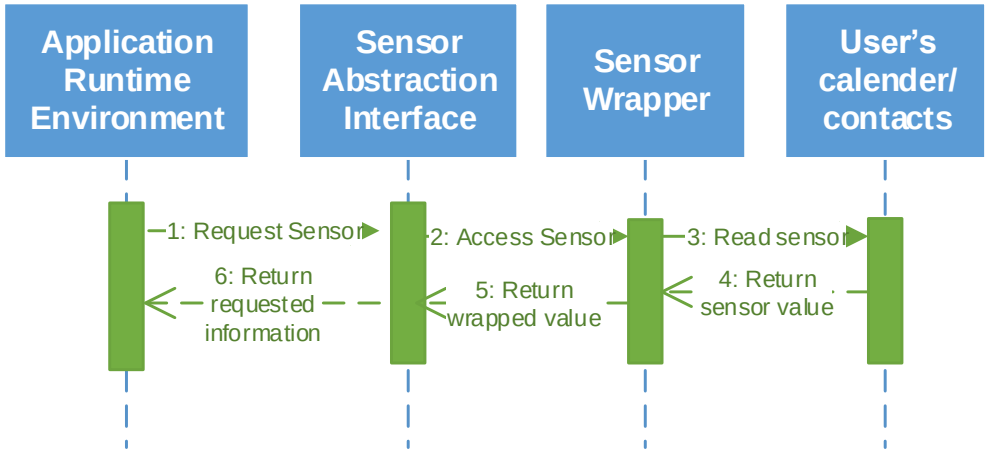

Figure 98: Interaction of Application Runtime Environment, PMA-based Sensor Abstraction and User's Calendar/Contacts

Analogue to the server side data access as discussed in Section 5.3.4.1, Figure 97 shows the interaction between the Application Runtime Environment and the PMA-based Sensor Abstraction. If an app requests sensor data, the Application Runtime Environment redirects the request to the PMA-based Sensor Abstraction that allows accessing the sensor by the use of the Sensor Wrapper. This component contains the Access Handler (not depicted) that is responsible to access the sensor and the Data Wrapper (not depicted) that transforms the returning sensor data into the common data format.

In the next step, the Sensor Wrapper passes the transformed data back to the Sensor Abstraction Interface that is now able to return the requested information. For all sensors that are accessible on demand the data flow is the same, thus the sensor depicted in Figure 97 could be both a local sensor on the PMA or a car sensor or even simulated sensor readings generated by a virtual sensor as software module. As depicted in Figure 98 also the access to the user's calendar and contacts is handled in the same way.

7.3.4.2 Interaction based on Buffered Sensor Data

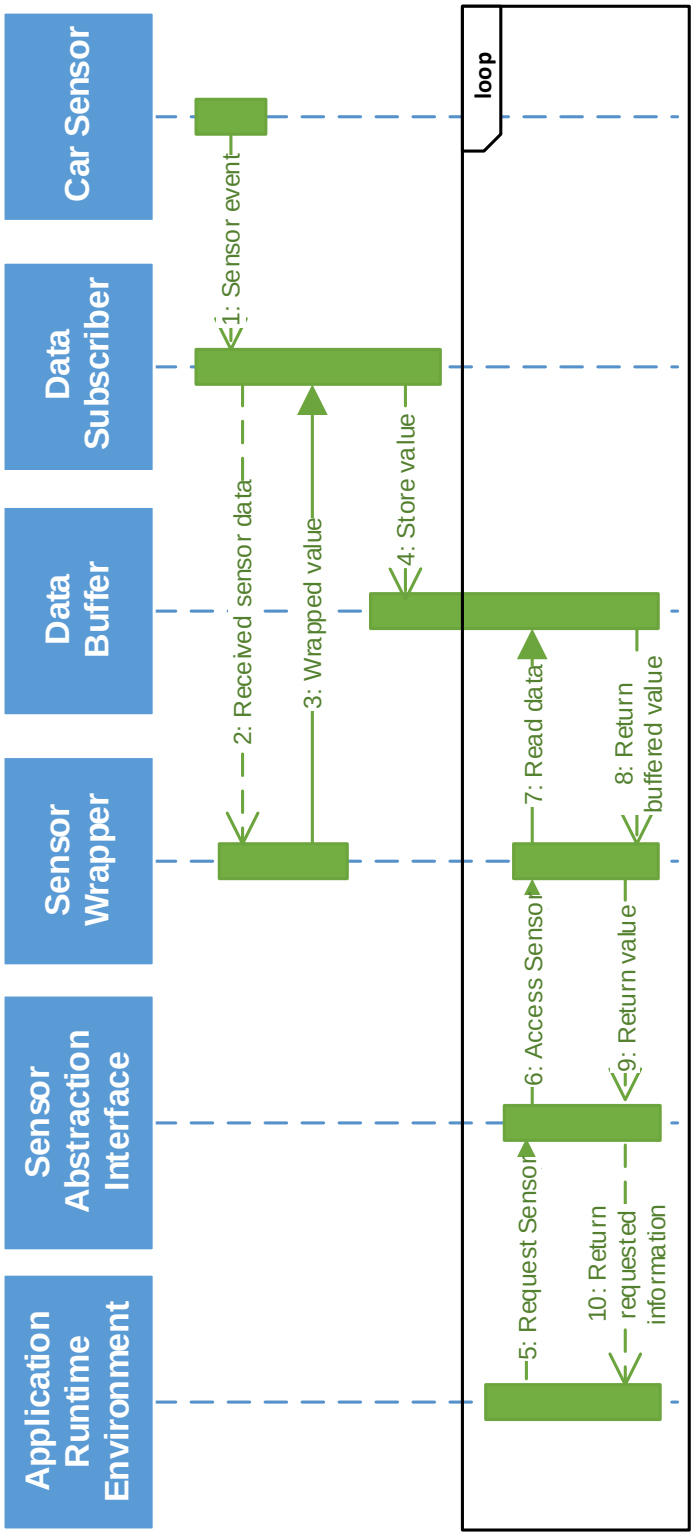

Figure 99: Interaction of PMA based Sensor Abstraction and Buffered Sensor Data

Figure 99 shows the interaction between the Application Runtime Environment and the PMA-based Sensor Abstraction for the case of buffered sensor readings. Some sensor data is only available at specific events. The Data Subscriber is listening for such events and receives the corresponding sensor reading which is then sent to the Sensor Wrapper

and transformed into the common data format. This wrapped value is then transferred back to the Data Subscriber component which stores the value in the Data Buffer for later usage. To enable this functionality, some preparative steps have to be done that are not depicted in the figure above. In the first step, the address of the Data Subscriber component has to be registered at the data source to enable the transmission of the respective sensor events. In the second step, a customized Data Wrapper has to be prepared by a developer to enable the translation of the proprietary external data format into the common SIMPLI-CITY data format.

If an app requests for such a type of sensor data, the Application Runtime Environment redirects the request to the Sensor Abstraction Interface that allows accessing the buffered sensor data by the use of the Sensor Wrapper. This component contains the Access Handler (not depicted) that is responsible to access the sensor and knows that the data of this sensor is buffered in the local Data Buffer. In the next step, the Sensor Wrapper passes the data back to the Sensor Abstraction Interface that is now able to return the requested information.

7.3.5 User Interface

The PMA-based Sensor Abstraction will not feature a specific user interface.

7.3.6 Conceptual Data Model

The conceptual sensor data model is described in Section 9.1.

7.3.7 Parameters to Take into Account for Technical Specification

Table 25: Criteria for Technical Specification

Parameter	Importance (--, -, -+, +, ++)
Generic Criteria	
Up-to-Datedness	-+
Stability	+
Extensibility & Open Source/Standards	+
Familiarity	+
Performance	++
Interoperability	+
Specific Criteria	
Low Energy Consumption	-+
Fast Response Time	++

Parameter	Importance (--, -, -+, +, ++)
Lightweight	+
Extensible Data Model	+

8 Functional Specification: Developer Support

8.1 Application Design Studio

8.1.1 Overall Functional Specification

The Application Design Studio offers an appropriate step-by-step procedure to app development, assisting the app developer during the entire development process. It delivers everything that a developer needs to prepare an app for the usage within SIMPLI-CITY. Among other things, the studio provides a set of guidelines and HowTos to developers.

The studio includes support for the definition of an App Manifest file. A Manifest file is used to specify the runtime environment requirements and to specify properties of apps such as the app icon or the dependencies to other apps. The resulting Manifest file will be included along with the app when it is submitted.

The Application Design Studio itself has no direct relation to components within the Application Runtime Environment (see Section 7.1) or the App Marketplace (see Section 6.5). Instead, interfaces to these components will come preinstalled in the form of JAR files that can be picked up by the IDE with all corresponding functionality like autocompletion and Javadocs within the developer's project.

8.1.2 Subcomponents

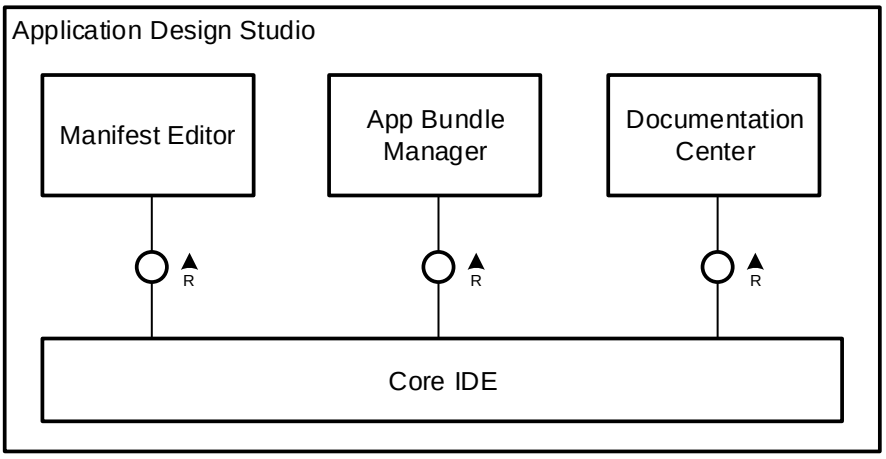

Figure 100: Application Development Studio Subcomponents and Interactions

To achieve the functionalities described in the previous subsection, this component provides the following subcomponents as depicted in the figure above:

- Core IDE: Providescore development facilities.
  - Based on a well-known IDE such as Eclipse or Google Android studio.
  - Has standard look and feel familiar for developers, SIMPLI-CITY functionality (i.e., exposition of functionalities from other components – see Section 8.1.4.2) will be provided by seamlessly integrated plugins.
- Manifest Editor: Provides a UI for creating the App Manifest file.

  - A GUI with predefined fields the developer has to be filled in. An additional tab will expose the underlying XML, which the developer can edit directly.
  - This subcomponent is similar to the built-in Eclipse editors for Ant or Maven configuration files.
- App Bundle Manager: Compiles the app and the manifest into a submittable format.
  - Invoked by clicking the deploy button of the corresponding menu item, this subcomponent packs the software and produces the deployable artefact, that is, a SIMPLI-CITY app.
  - The GUI of the subcomponent is rather minimal, the majority of work is done without interaction with the developer.
  - Bundled apps are stored locally and can then be uploaded manually to the App Marketplace (see Section 6.5.4.2).
- Documentation Center: Provides HowTos, UI templates, examples and other documentation.
  - Integrated into the help system of the IDE, the Documentation Center supplies necessary documents and code samples based on the context upon hitting a predefined help key. The documents are structurally organised via a master table of context document where dependencies and appropriate context are defined in the XML format.
  - The help documents themselves are separate HTML files, editable by the developer of the service and/or component used in that moment. The HTML files are downloaded to the plugin *help* folder when the developer downloads the API of certain component. As an example, if the developer is going to make use of the Cloud-based-storage API to store/retrieve files from there, the HTML files accompanying the API will be locally downloaded to the “plugins/cloud-based-storage/help” folder.

The component indirectly interacts with the App Marketplace because it creates a Manifest file, which users will bundle together with their app when submitting an app to the marketplace. As such, this component and the App Marketplace component need to have the same understanding of the Manifest file.

8.1.3 Related Requirements

Table 26: Requirements Related to the Application Design Studio

Requirement	Handled by Subcomponent	Comment
Must Have Requirements		
U105: Access to the dialog system (P1) U159: Support for Dialogue (P1) U160: Support for Apps (P1)	Core IDE Documentation Center	The Application Design Studio is the component used by the developer when it comes to hands-on development of SIMPLI-CITY apps. As such, the access to the different apps provided by the SIMPLI-CITY Application Runtime Environment has to be possible during that task. However, the Application Design Studio itself has no direct relation to components within the Application Runtime Environment (Push Service, Message Handler and ARE Foundation Provider). Instead interfaces to these components will come preinstalled in the form of JAR files that can be picked up by the IDE with all corresponding functionality like auto-completion and Javadocs within the developer's project. Please note that an app gets support for functionality (U105, U159, U160) from the first column via the Application Runtime Environment. For that reason, the app has to implement Application Runtime Environment interfaces which will be shipped with the Application Design Studio..

Requirement	Handled by Subcomponent	Comment
U106: Access to cloud services (P1) U161: Support for Data services (P1) U162: Support for Backend services (P2)	Core IDE Documentation Center	Services provide the basic functionalities for end user apps. As such, the access to the different services provided by the SIMPLI-CITY Service Runtime Environment has to be possible during that task. However, the Application Design Studio itself has no direct relation to the services offered by the Service Runtime Environment. Instead, functionalities of the Service Runtime Environment, which are necessary for the execution of apps are provided through the Application Runtime Environment. For the integration of backend and data services, app developers need to be aware of their endpoints and can then directly invoke them.
U155: Provision of Java API (P1) U157: Standard programming interface (P1) U158: Easy access to API through the developer studio (P2)	Core IDE Manifest Editor Documentation Center	The Application Design Studio will come based on a well-known IDE such as Eclipse or Google Android studio. These functionalities will be realised through the IDE itself and the Manifest Editor which will ease the development tasks. Additionally, the documentation of the API will make use of the Documentation Center (as Javadocs) for ensuring a correct understanding of the different functions.
U166: Identification of the developer/signature (P1)	App Bundle Manager	The App Bundle Manager will allow signing the bundle with the developer's digital signature to identify the origin of the app.

Requirement	Handled by Subcomponent	Comment
U168: Provision of source code examples (P1) U169: Provision of UI templates (P1) U170: Provision of best practices (P1) U171: Provision of tutorials (P1) U172: Provision of guidelines (P1) U173: Provision of examples (P1)	Documentation Center	The Application Design Studio will provide samples for writing apps for SIMPLI-CITY which will include interactions with all types of PMA-Side components (see Global Architecture in Section 2.1) as well as samples for App GUIs. Additionally, written or even video recorded tutorials on how to use the Application Design Studio may be provided.
U178: Definition of minimum hardware requirements of apps (P4)	Manifest Editor	The developer will be required to specify minimum hardware requirements in the Manifest file. If these are omitted, the editor will report an error.
Should Have Requirements		
U23: Unified look & feel within the project (P2) U24: Unified look & feel for 3rd party developers (P2) U25: Usage of guidelines within the project (P2) U26: Intuitive usability (P2)	Documentation Center	Please note that this mainly depends on the individual preferences of the app developer. Nevertheless, SIMPLI-CITY will provide the means to achieve a unified look and feel by supplying HowTos and manuals describing how to achieve this. Furthermore, UI templates will be provided to the developers.
U102: Backwards compatibility of API (P3)	App Bundle Manager	Please note that this also depends on the interfaces and behaviour of all other SIMPLI-CITY components. All artefacts in SIMPLI-CITY will have standard version notation <major>.<minor>.<patch> where an increase in the major number will indicate backward incompatibility. Based on that information, the App Bundle Manager will identify and highlight possible API incompatibilities.

Requirement	Handled by Subcomponent	Comment
U174:Permit to develop an app in less than a day (P1) U175: Hide complexity from the developer (P2)	Core IDE Documentation Center	Please note that this is not a technical requirement as it highly depends on the pre-knowledge of the developer. However, the Application Design Studio will support this by providing documentation, examples and modern development functionality like code autocompletion and code assist.
Could Have Requirements		
U180: Easy debugging of services (P3) U181: App crash reports (P5) U182: Provision of statistics, e.g., usage, traffic (P3) U183: Provision of bug reports (P4) U184: Provision of crash reports (P5)	Core IDE App Bundle Manager	All this functionality will be realised through usage of specially developed auxiliary libraries that can be embedded into an app and that will deliver required feedback.
Will not have for now		
U156: Provision of C/C++ API (P5)	N/A	The chosen reference device is Android-based. Therefore, Java is the natural choice. C/C++ API might be interesting for developers using the actual base system of Android (Linux), but its hardware/software configuration is not known now.
U177: Provision of a PMA emulator (P4)	N/A	The reference device is a generic Android-based smartphone. For this reason, there is no need for an emulator.
U187: Composition of apps (P5)	N/A	This feature has been marked as hardly relevant in the Requirements Analysis (deliverable D2.3), but implementing functionality supporting it would require disproportionally high effort.

8.1.4 Interaction with other Components

8.1.4.1 Interaction with the App Marketplace

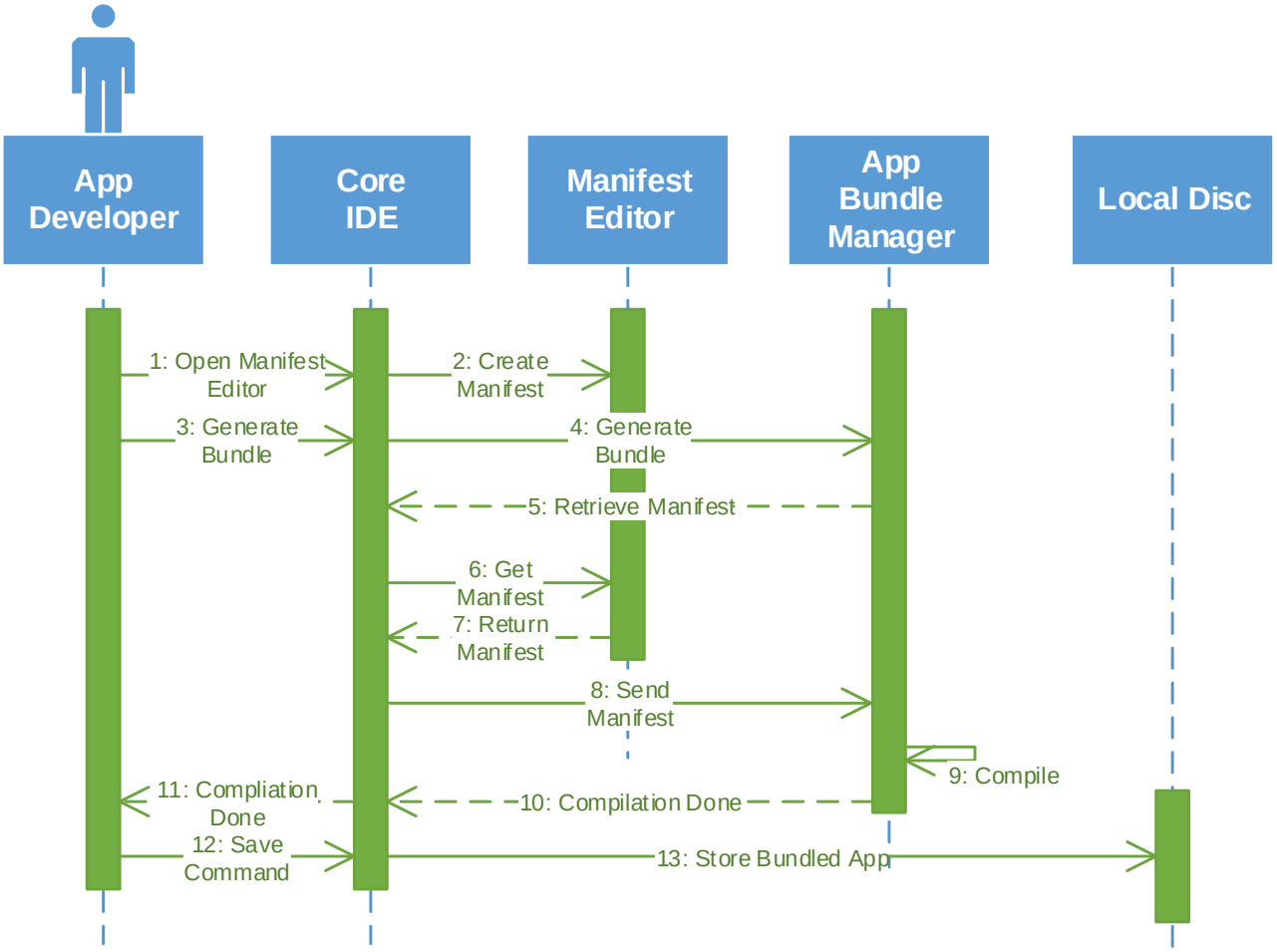

Figure 101: Interaction of the Application Development Studio and App Marketplace

The Application Design Studio includes support for the definition of an App Manifest file, which allows developers to specify properties for their apps including the commands, access rights and requirements of the app. The Manifest Editor is available via the Core IDE and provides a GUI to easily create and compile an App Manifest. Once the Manifest file is completed, developers bundle the app with it and then save the Bundled App on the Local Disc. While it is not a prerequisite for Step 3 to have a new compiled manifest, it is recommended to start command and access rights definitions only once a Manifest file has been generated.

As depicted in Figure 101, the compiled and bundled app is ready to be uploaded to the App Marketplace via its CRUD operations (see Section 6.5.2 and Figure 62).

The process to update an already existing app follows the same process steps. However, before starting the creation of a new, updated Manifest file, the App Developer would load the existing version of the app into the Core IDE.

8.1.4.2 Exposition of Functionalities from Other Components

The Application Design Studio itself is not used by any other components, as it is the main tool to create new apps. To gain full functionality and usability, the Application Design Studio has to provide access to the Application Runtime Environment. This runtime needs to provide the functionality of the SIMPLI-CITY PMA in order to give the developer the opportunity to debug and run the app during development.

The Application Design Studio exposes the functionalities of the following components through according APIs:

- Cloud-based Information Infrastructure (see Section 5.2): Developers have to make use of the Cloud-based Information Infrastructure in order to store app-relevant data. This exposition includes the Local Key Storage (not depicted in Figure 14), which is the main part of this exposition, since the Local Key Storage is the main data storage for SIMPLI-CITY apps that are running on the Personal Mobility Assistant.
- Media Data Streams and Data Prefetching Logic (see Section 5.4): Developers who want to make use of media playback can access the Media Playback API (PMA Side) of the Media Data Streams and Data Prefetching Logic component (see Section 5.4.2). The Data Prefetching API (Server Side) of the Media Data Streams and Data Prefetching Logic will not be considered right now since it is part of the Server Side of the Media Data Streams and Data Prefetching Logic component.
- Application Runtime Environment (see Section 7.1): The Application Runtime Environment is responsible for most of the functionality used on the PMA. This applies to the full Application Runtime Environment stack which includes all interactions of the Application Runtime Environment with other SIMPLI-CITY components through according interfaces or APIs. For this, the Application Design Studio allows to bundle apps. Subsequently, the app developer has to upload the app bundle to a mobile device for testing purposes.
- Multimodal User Interface (see Section 7.2): To create and test voice recognition while developing an app, functionalities of the Multimodal User Interface need to be exposed to the Multimodal User Interface via an API. This includes either microphone abstraction via the Application Runtime Environment or via the soundcard of the developer's machine.
- PMA-based Sensor Abstraction (see Section 7.3): App developers need to access sensor data while developing a SIMPLI-CITY app. This includes all car-sensors and the simulated sensors as well as personal data (e.g., calendars, contact-data), which is depicted in the FMC figure in Section 7.3. This data will be available for developers for full usage in the app that is being developed.

8.1.5 User Interface

The Application Design Studio will most likely be based on top of an existing IDE such as Eclipse or Google Android Studio. As such, the IDE will be oriented towards source code editing facilities. The following figure shows a typical screenshot of those types of IDE using Eclipse as an example:

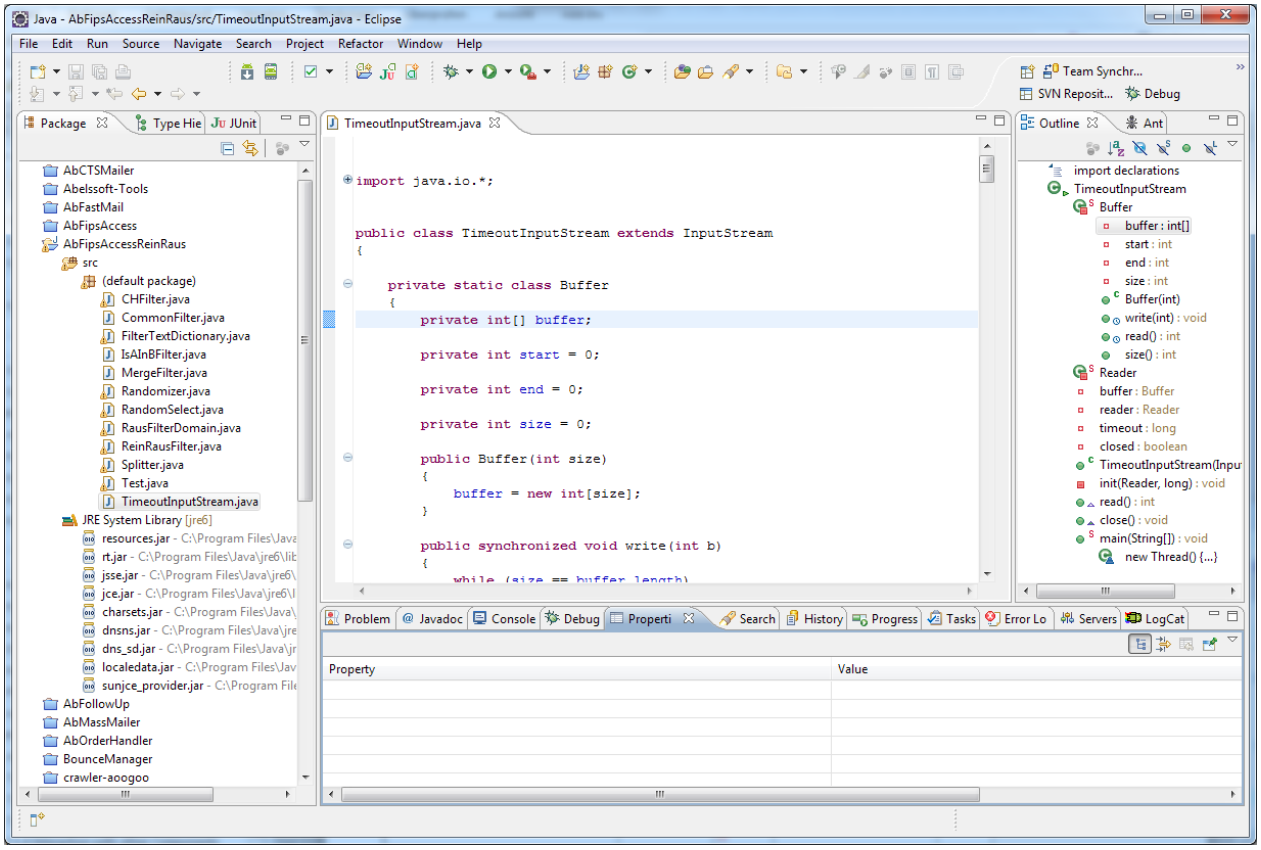

Figure 102: Screenshot of a Typical Development IDE

8.1.6 Conceptual Data Model

The App Manifest Data Model is described in Section 9.4.

8.1.7 Parameters to Take into Account for Technical Specification

Table 27: Criteria for Technical Specification

Parameter	Importance (--, -, -+, +, ++)
Generic Criteria	
Up-to-Datedness	++
Stability	+
Extensibility & Open Source/Standards	++
Familiarity	++

Parameter	Importance (--, -, -+, +, ++)
Performance	-+
Interoperability	+
Specific Criteria	
Android Development Support	++
Plugin Support	++
Ease of Use	++

8.2 Service Development API

8.2.1 Overall Functional Specification

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.

The Service Development API covers the full lifecycle of creation and management of services. It provides the means to easily define, design, and develop services, and it assists developers during the whole process, providing templates and documentation. The templates (e.g., abstract classes) allow the exposition of functionality provided by other SIMPLI-CITY components like the Context-Based Service Personalisation component or the Service Monitoring component into backend services, with the objective that the created services are enabled for being executed within the SIMPLI-CITY Service Runtime Environment.

Furthermore, the Service Development API allows building wrappers/proxies for SIMPLI-CITY-external backend and data services and registering them within the Service Registry for future invocations.

The functionalities provided by the Service Development API are:

- Creation and configuration of services:
  - Creation and registration of new services, including the provision of abstract classes and methods for the creation of SIMPLI-CITY-enabled services.
  - Configuration and update of existing services.
  - Deletion of services.
  - Service discovery/lookup.
- Service metadata management.
- Exposition of functionality of other SIMPLI-CITY components and subsequent inclusion of such functionalities into the proxies and services created by developers.
- Access and exposure of data from data sources through data services and according wrappers.
- Proxy generation for internal and external services.

- Provision of documentation, tutorials, and examples in order to help developers during the service creation process.

The Service Development API provides its functions through a publicly accessible set of methods for programmatic access. These functionalities could be provided through a set of easy-to-use plugins for an IDE.

The created services are stored in the Service Registry and are available for execution within the Service Runtime Environment. Service developers are able to provide their services to other developers through the Service Marketplace, as described in Section 6.5, but these services can also be configured for exclusive use by the service creator.

8.2.2 Subcomponents

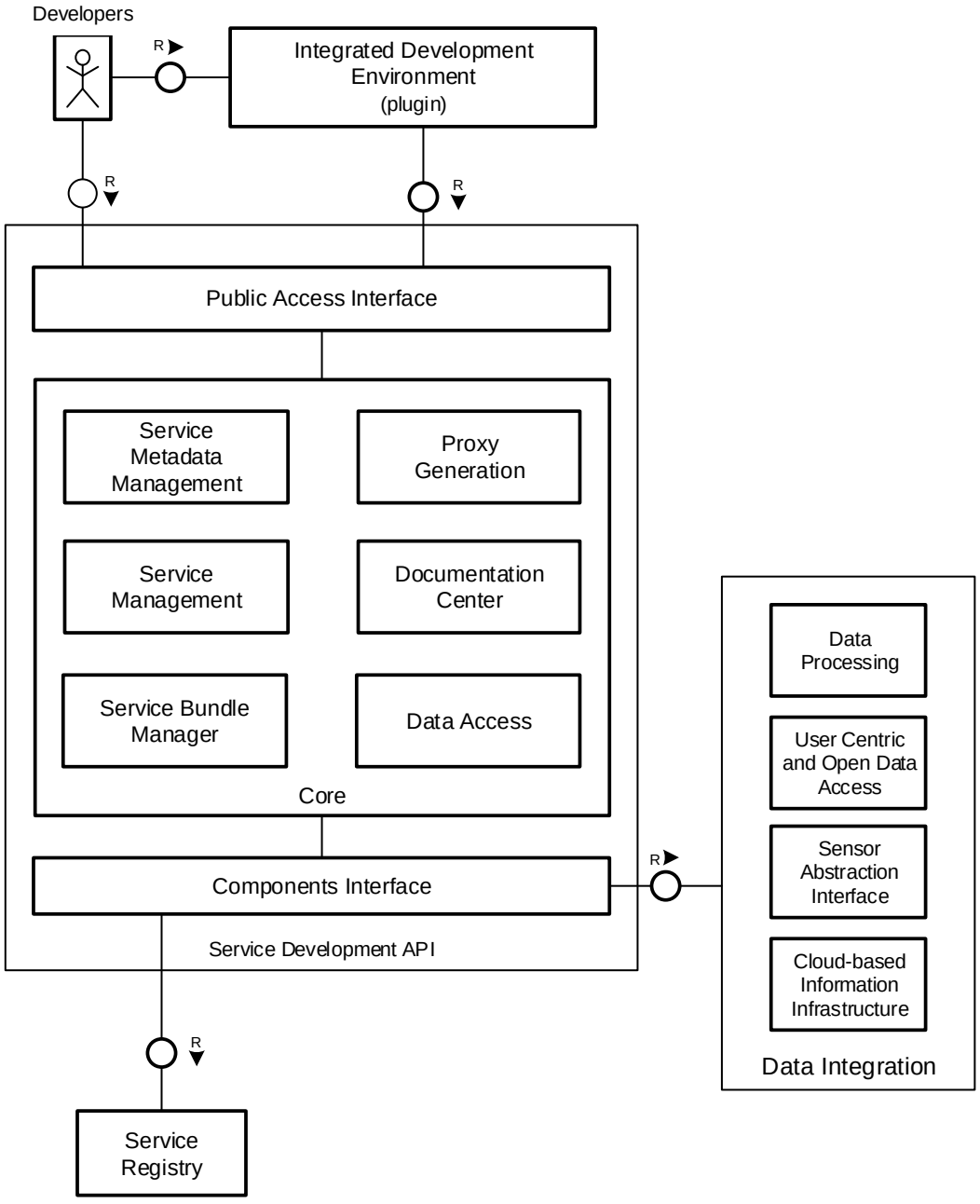

Figure 103: Service Development API Subcomponents and Interactions

Figure 103 shows the different subcomponents composing the Service Development API and it briefly describes the relationship with other components of the SIMPLI-CITY framework.

To achieve the functionalities described in the previous subsection, the Service Development API provides the following subcomponents:

- Public Access Interface: This component provides external access to service developers of all the functionalities provided by the component. It can either be accessed directly or via an IDE plugin.
- Core: This subcomponent provides the core functionality of the Service Development API. It is composed of different modules, each one covering specific functionality:
    - Service Management: Provides the necessary functions to manage a service: creation, modification/update, and deletion of services. The component interacts with the Service Registry to implement this functionality.
    - Service Metadata Management: Allows the management of the metadata information of a service. This metadata information includes basic information to describe a service, classification information, personalization information and the SLA associated with a service. This metadata information is validated before the submission to the Service Registry.
    - Service Bundle Manager: Manages the bundling of the service artefact and associated metadata into the service bundle.
    - Data Access: Provides runtime values of the data sources used within a service. It interacts with the different components providing data access like the User Centric and Open Data Access component and the Data Processing component to obtain this information. Also, it allows to directly access data sources which have been previously registered as data services.
    - Proxy Generation: Allows the generation of proxies for external and internal backend services as well as data services. A proxy is formed by a wrapper over a service which includes augmented functionality for the service, including security, monitoring and accounting support. This augmented functionality is provided by the following components/subcomponents:
        - Context-based Service Personalisation: Usage of methods to integrated context-based personalisation functionalities into internal backend services (see Section 6.3).
        - Monitoring: Usage of monitoring functionality for SLA management and enforcement (see Section 6.2).
        - Service Runtime Environment: Usage of accounting functionalities provided by the subcomponent Accounting (see Section 6.1).
    - Documentation Centre: Provides documentation for the developer about the usage of the component for service creation and management, including HowTos and examples.
- Components Interface: Allows the integration with other components of the SIMPLI-CITY framework and therefore provides all necessary functionalities to communicate and translate information from/to other components.

8.2.3 Related Requirements

Table 28: Requirements Related to the Service Development API

Requirement	Handled by Subcomponent	Comment
Must Have Requirements		
U106: Access to cloud services (P1)	Public Access Interface Core (Data Access) Components Interface	The Service Development API must be able to provide the means to integrate cloud-based data services into backend services.
U148: Service Registration (P1)	Public Access Interface Core (Service Management) Components Interface	The Service Development API interacts with the Service Registry in order to register services.
U149: Service extension and modification (P1) U150: Service management (P1)	Public Access Interface Core (Service Management) Components Interface	The Service Development API provides the means to manage previously created services, including the modification/update and the deletion of existing services.
U152: SLA support (P1)	Public Access Interface Core (Service Metadata Management) Components Interface	The Service Development API permits to define SLAs associated to a service, and to store this information in the Service Registry. Also, this information may be changed or deleted, if necessary.
U161: Support for data services (P1) U162: Support for backend services (P2)	Public Access Interface Core (Service Management) Components Interface	The Service Development API provides the means to create and configure data and backend services. For this, functionalities to build services, provide wrappers and proxies, and register services, will be provided.
U168: Provision of source code examples (P1) U169: Provision of UI templates (P1) U170: Provision of best practices (P1) U171: Provision of tutorials (P1) U172: Provision of guidelines (P1) U173: Provision of examples (P1)	Public Access Interface IDE (plugin)	The Service Development API supports service developers during the service creation process through according documentation and examples. This support should include the provision of source code examples, user interface templates, general guidelines, tutorials and examples of the overall process of creation of services, including the configuration of data services.

Requirement	Handled by Subcomponent	Comment
U179: Service Metadata (P1)	Public Access Interface Core (Service Metadata Management) Components Interface	The Service Development API provides the means to describe services using metadata, store this metadata in the Service Registry, and update/delete the metadata, if necessary.
Should Have Requirements		
U59: User centric data services (P1)	Public Access Interface Core (Service Management) Components Interface	The Service Development API permits to realize and configure the integration of user-related data into services. For this, User centric data services may be registered and building additional information in terms of proxies and wrappers is supported.
U114: Configuration of the frequency of update of the data from data sources (P1)	Public Access Interface Core (Service Metadata Management) Components Interface	In some cases, backend services need to pull data from a particular data source/service on a regular, predefined basis. This is supported by the Service Development API.
U116: Unified data model (P1)	Public Access Interface Core (Data Access) Components Interface	The Service Development API makes use of the Unified Data Model (see Section 5.1) to describe services. Furthermore, it allows integrating arbitrary data services which make use of this data model. Also, it provides the means to build data service proxies/wrappers using the Unified Data Model for the description of service inputs and outputs.
U151: Service versioning (P1)	Public Access Interface Core (Service Management) Components Interface	The Service Development API supports the storage of different versions of the same service in the Service Registry. This includes management of service bundles/artefacts and the according metadata.
U163: Easy to use environment (P2) U164: Low investment required (P3)	Public Access Interface IDE (plugin)	The Service Development API provides an easy to use environment to create and manage services. Moreover, the interface should not require a heavy investment to use it.

Requirement	Handled by Subcomponent	Comment
U166: Identification of the developer / signature (P1)	Public Access Interface Core (Service Management) Components Interface	It should be possible to store information about the developer of a particular service. The Service Development API allows to define signatures and to store them (together with the service information) in the Service Registry.
Could Have Requirements		
U165: User-friendly developer studio interface (P2)	IDE (plugin)	The Service Development API should provide its functionalities in a user-friendly way.
U180: Easy debugging of services (P3) U183: Provision of bug reports (P4)	Public Access Interface Core (Service Management) Components Interface	The Service Development API should provide debugging functionalities that allow developers to debug services and to be informed about the correct operation or about any problem occurred during the process.
Will not have for now		
U167: Service hot updates (P4)	Public Access Interface Core (Service Management) Components Interface	The Service Development API could support to update a service in real time. This means that both service descriptions and service artefacts must be updated on-the-fly. However, this functionality will not be provided for now, as it is not planned for the Service Runtime Environment to allow it.
U181: Service crash reports (P5) U184: Provision of crash reports (P5)	Public Access Interface Core (Service Management) Components Interface	The Service Development API could provide service developers with extensive crash reports. However, this functionality will not be supported for now.
U188: Composition of services (P3)	Public Access Interface Core (Service Management) Components Interface	The Service Development API could allow service compositions, which combine services in order to provide value-added functionality.

Requirement	Handled by Subcomponent	Comment
U197: Community support for service developers (P5)	IDE (plugin)	SIMPLI-CITY could provide the means to create a community of developers. However, this would require providing according software tools for social networking etc., which would lead to large efforts that cannot be justified for such a tertiary functionality.

8.2.4 Interaction with other Components

The Service Development API interacts with different components of the SIMPLI-CITY framework, acting as the entry point for service developers in order to create and manage services. It provides a programmatic interface for using the different functions provided by the component. This programmatic interface can be used directly by the developer or through the plugin provided for an IDE.

8.2.4.1 Interaction with the Service Registry (Service Discovery)

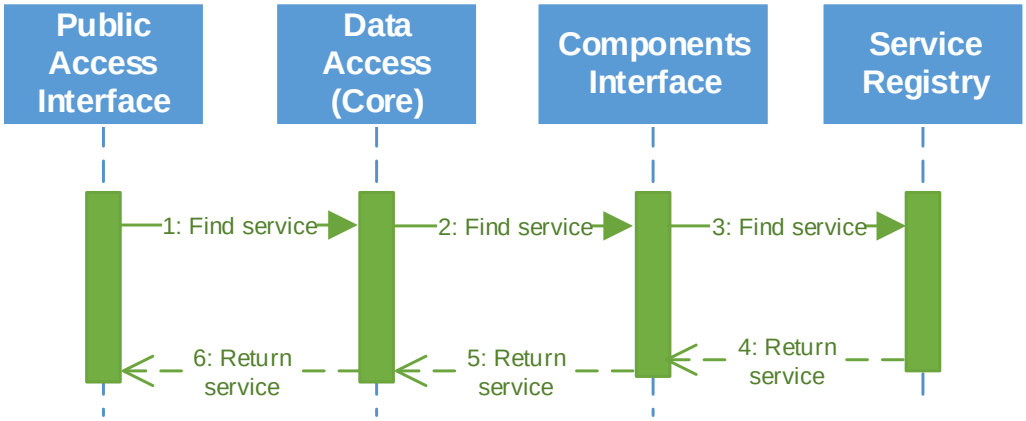

Figure 104: Interaction of Service Development API and the Service Registry for Service Discovery

Figure 104 depicts a sample service lookup/discovery operation, returning an identified service. As it can be seen, the Public Access Interface is used (by a service developer) in order to find services based on some particular service data, e.g., the name of a service. This request will be forwarded to the Service Registry through the Data Access subcomponent and converted in the Components Interface, if necessary. The Components Interface subcomponent eases the integration with the Service Registry.

The technical details of this translation should be defined in D3.2.2. Once the Service Registry has received a valid request, the service information (e.g., service metadata, but also the service manifest) will be returned to the requester.

8.2.4.2 Interaction with the Service Registry (CRUD Functionalities)

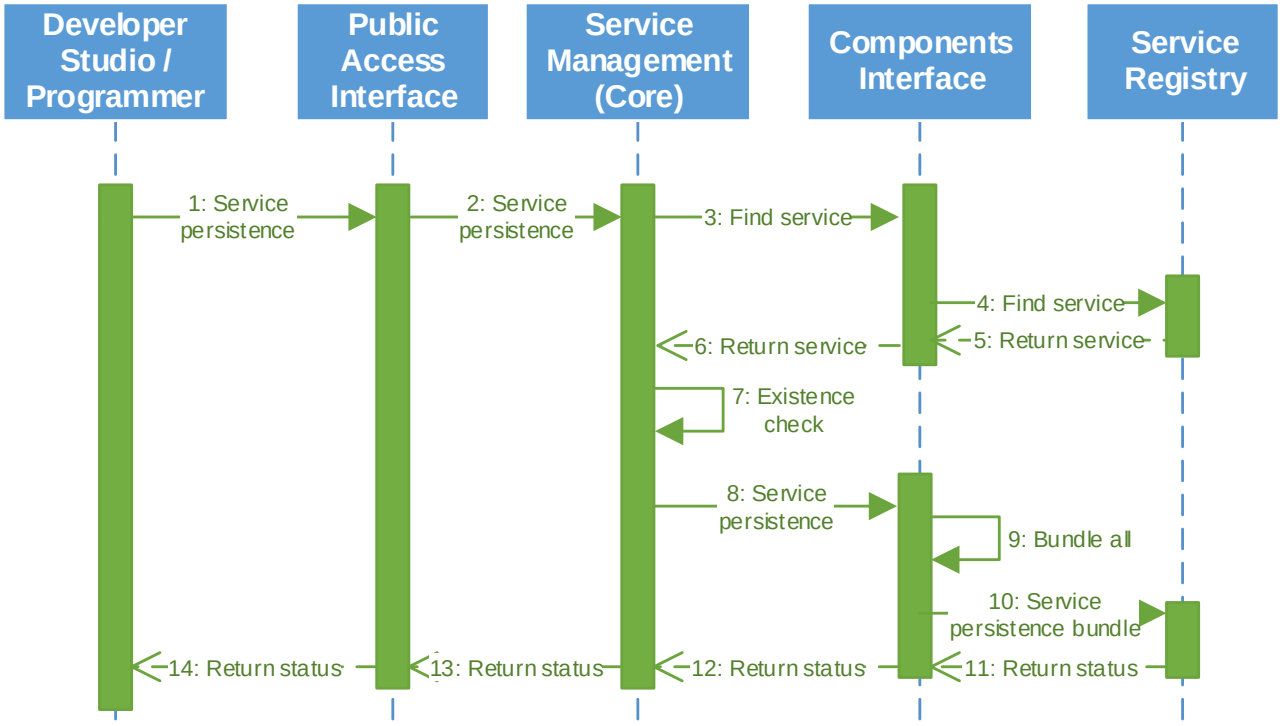

Figure 105: Interaction of Service Development API and the Service Registry for CRUD Operations (Updating an Internal Backend Service)

The usage of the Service Registry's CRUD functionalities by the Service Development API has been discussed in detail in Section 6.4.4.1. In the following descriptions, the same functionalities will be described from the internal perspective of the Service Development API.

These basic service management operations may be classified into two groups, persistence (creation and update) and deletion calls, which affects the semantics of the messages needed to be sent to the Service Registry.

The persistence operations group handles two possible scenarios, according to the nature of the persisted service. The first persistence scenario is for internal Backend Services, grouping its artefact file with the metadata; and the second persistence scenario is for metadata-only services: External services and data services, which are based on a proxy artefact, defined using the Service Development API and stored in the Service Registry. The details about how such proxy artefacts are stored will be defined in D3.2.2.

Each scenario takes into account a different option for delivering the service information and associated metadata. The information of a service is provided by means of a service bundle, which is formed by a service artefact and associated metadata. The service artefact may be provided either as a JAR file or as a WAR file (see Section 6.4.6)

Before the service bundle is delivered to the Service Registry, a validation phase takes place, considering the following aspects:

- Basic service conformance (e.g., checking the presence of the metadata, JAR structure standard adhesion).
- Metadata analysis:

- Service / Metadata cohesion (e.g., name coherence, service version applicability).
- Export section: coherence, SLA support, supported MIME types, etc.
- Import requirements (data sources, etc.) are checked one-by-one for existence (and applicability), SLA provision (e.g., there is enough room for data transmission), etc.

- Problems with the Service Registry.

Import requirements problems should be notified as non-fatal errors, because these requirements could be fully provided at runtime.

Figure 105 depicts the first persistence scenario for an internal backend service, i.e., a service for which the service artefact is available in the Service Registry. The first operation consists in checking if the service is already available in the Service Registry, by means of a service lookup/discovery call between the Components Interface and the Service Registry. Then, the service metadata and artefact are bundled in the Components Interface and provided to the Service Registry for its storage.

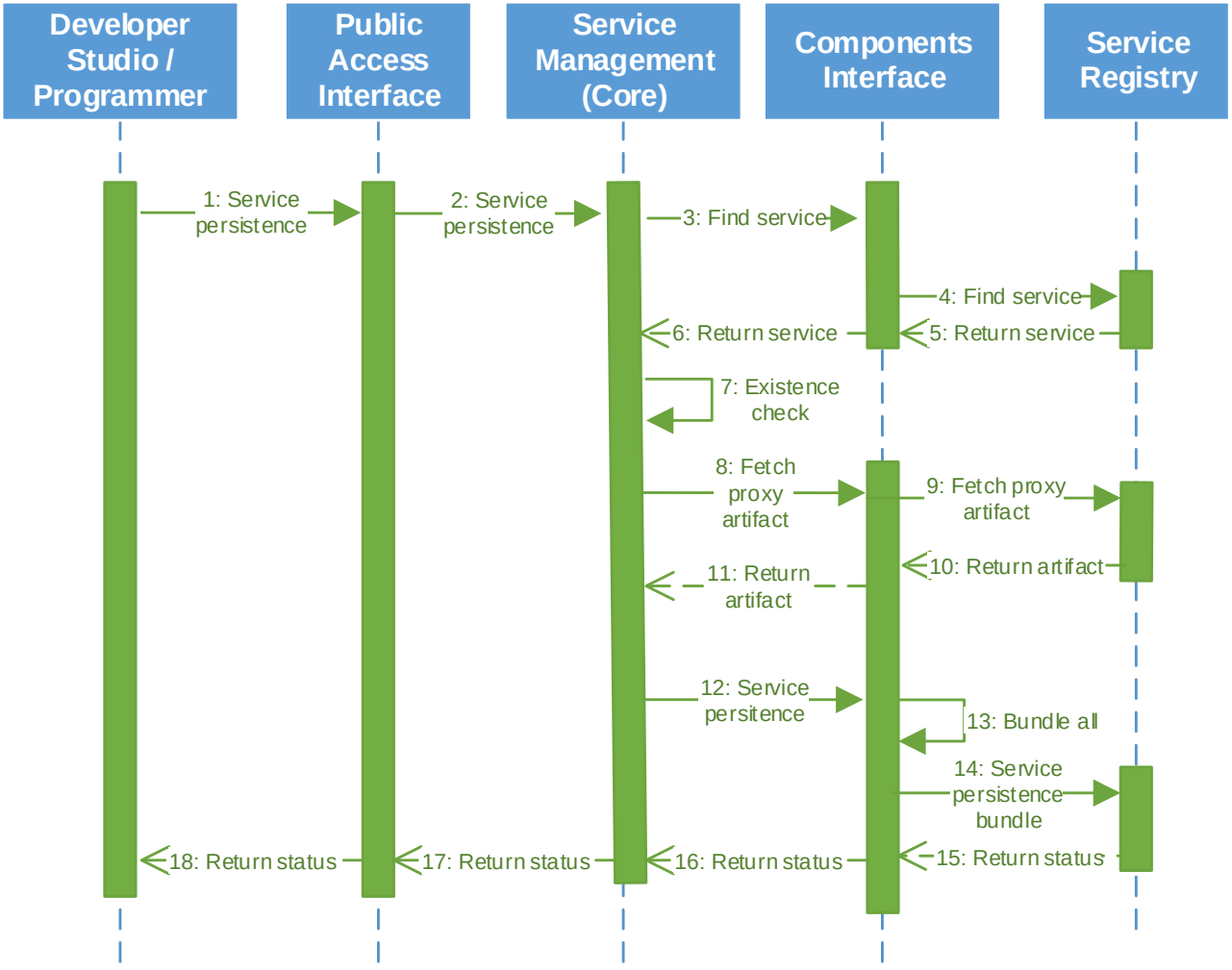

Figure 106: Interaction of Service Development API and the Service Registry for CRUD Operations (Updating an External Backend Service or Data Service)

Figure 106 depicts the second persistence scenario for an external Backend Service. First, a service lookup/discovery call is performed to the Service Registry in order to check if the service is already available in the Service Registry. Then, the proxy artefact is fetched through another call. And after that, the service bundle is persisted into the Service Registry. As it can be seen, the actual service bundle is not stored in the Service Registry and therefore only proxy information and metadata can be updated.

Figure 107 depicts a service deletion call for a service (either internal or external backend service or data service):

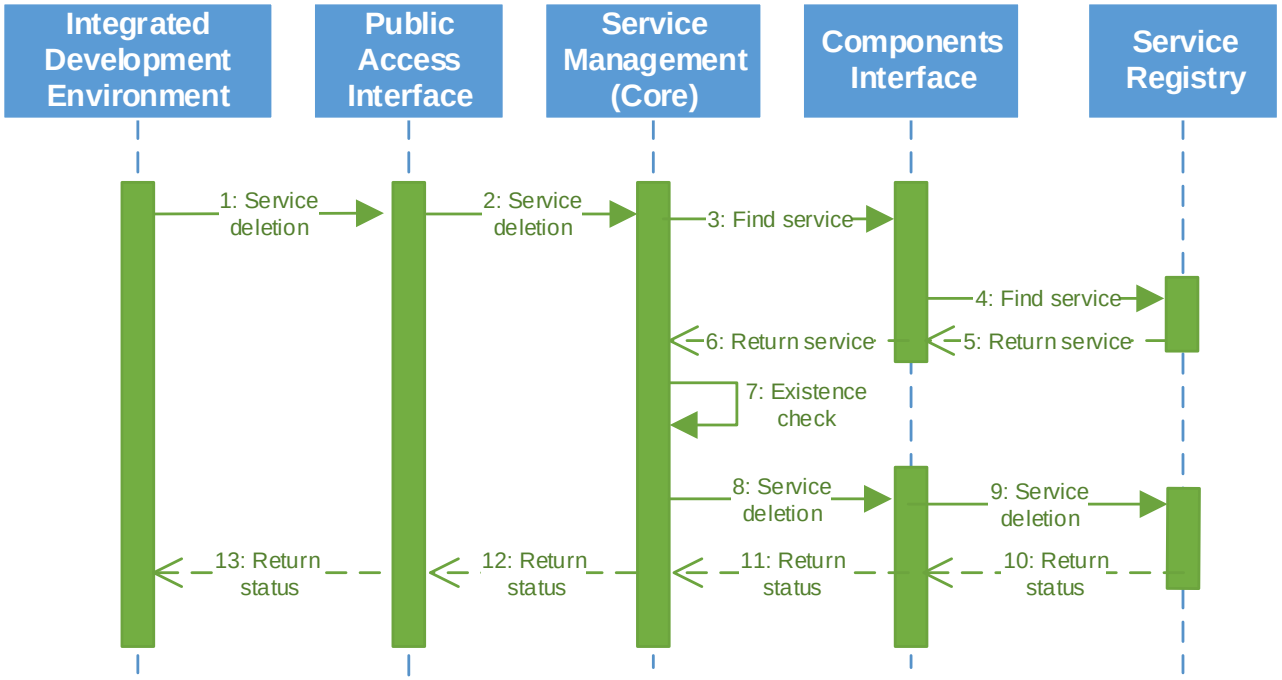

Figure 107: Interaction of Service Development API and the Service Registry for CRUD Operations (Deleting a Service)

8.2.4.3 Interaction with the Data Services

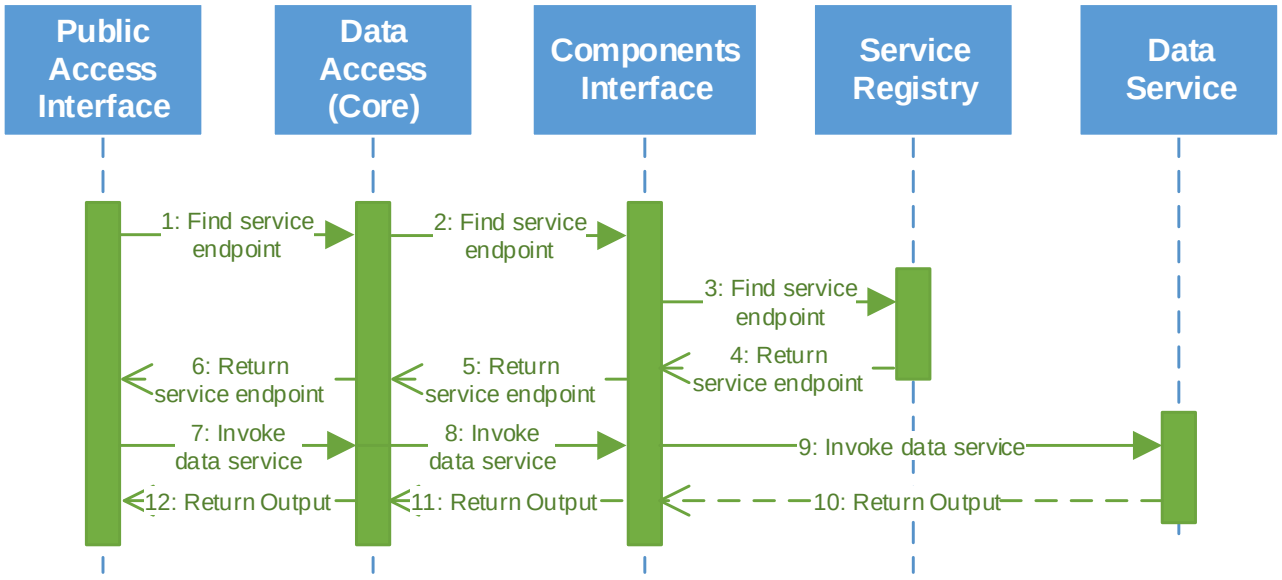

Figure 108: Interaction of Service Development API and a Data Service

During service development, it might be helpful for the service developer to access the Data Services that should be used in a Backend Service. For this, the Service Development API features exactly such functionality. As it can be seen in Figure 108, the required interactions are based on service lookup/discovery as described in Section 8.2.4.1, i.e., it is first necessary to request the information about a service from the Service Registry. However, for Data Services, the service endpoint is necessary. This service endpoint could also be the endpoint of a proxy, i.e., leading to an indirect call of the actual data source (not depicted in the figure). Afterwards, the Data Source (Data Service) can be invoked through the Data Access and Components Interface.

If the service endpoint is already known to the service developer, Steps 1-6 can be omitted and the Data Service can be directly accessed. This is also the case if the Data Service for a particular Data Source has not been defined yet – in this case, a Data Source needs to be invoked directly.

Importantly, the interactions for invoking a particular Data Source are always the same, regardless if it is preprocessed data (see Section 5.1), data stored in the cloud (see Section 5.2), or sensor data (see Section 5.3).

8.2.4.4 Exposition of Functionalities from Other Components

The Service Development API allows exposing functionality provided by other components of the SIMPLI-CITY platform. This way, functionalities from other components can be easily integrated into backend services and therefore exploited by service developers.

This exposed functionality is provided by means of abstract methods/classes (service templates) that developers have to extend within their own services in order to make use of these functionalities. Some of these functionalities are of compulsory use in order to execute the services within SIMPLI-CITY (e.g., Accounting and Monitoring), while other functionalities are optional depending on the needs of the developer (e.g., usage of Context-based Service Personalisation functionalities).

The Service Development API exposes the functionality of the following components:

- Data Processing component (see Section 5.1): Developers are enabled to include into services data available through the Data Processing component, as well as other functionality provided by this component such as the contextualization and reasoning methods. The usage of this functionality is optional and developers are able to make use of it by means of API calls.
- Cloud-based Information Infrastructure (see Section 5.2): Service developers are able to access data from the Cloud-based Information Infrastructure through pre-defined interfaces. These interfaces are exposed by the Service Development API.
- Sensor Abstraction and Interoperability Interfaces (see Section 5.3): Sensors provide individual REST-based interfaces, which can be used by service developers. Following the DaaS approach, the Service Development API allows the integration of such data sources into backend services in a unified way.
- Accounting (see Section 6.1) and Licensing (see Sections 6.1 and 0): For SIMPLI-CITY-internal backend services, SIMPLI-CITY offers the possibility to automatically bill service usage and check if the service consumer actually has got a valid license to make use of a service.
- Monitoring (see Section 6.2): All data and backend services include functionality to be used by the Monitoring component in order to monitor the QoS parameters of a service. Inclusion of this functionality is compulsory in a service.

- Context-Based Service Personalisation (see Section 6.3): Developers are able to include into internal backend services the functionalities provided by the Context-Based Service Personalisation component by implementing specific abstract classes that permit to make use of the functionality.
- Push Service (see Section 7.1): SIMPLI-CITY allows two different kinds of messaging between apps and backend services – either pull-based, i.e., apps request backend services ad hoc, or push-based, i.e., apps subscribe themselves to backend services for (regular or irregular) information updates, which will be automatically pushed to the apps. For push-based interactions, backend services need to implement according functionalities provided by the Push Service subcomponent of the Application Runtime Environment. This functionality is exposed through an abstract class and is only available for internal backend services.

8.2.5 User Interface

The proposed user interfaces are visual example mockup proposals for the plugin of the IDE. This plugin allows to manage services and to edit their information based on the metadata.

8.2.5.1 Creating a New Service

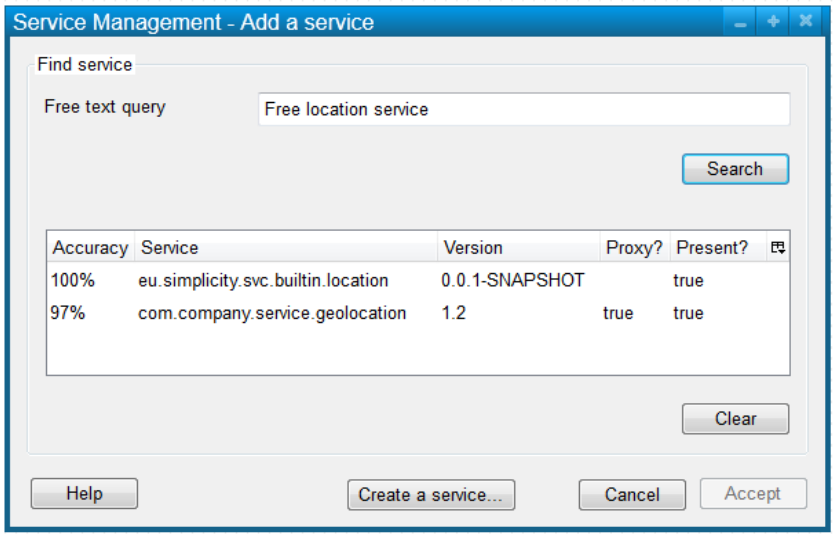

Figure 109: Service Addition Mockup User Interface

Figure 109 shows how to add a service and also how to find a service based on a free text query. The “create a service” button allows the creation of a new service and to add the information (metadata) of a service Furthermore, by clicking on a listed service, it is permitted to update the metadata associated to the service. Both cases lead to the mockups of the next subsection.

8.2.5.2 Service Metadata Management

Figure 110 shows the basic and extended information sections of the metadata based on the preliminary service metamodel as described in Section 9.

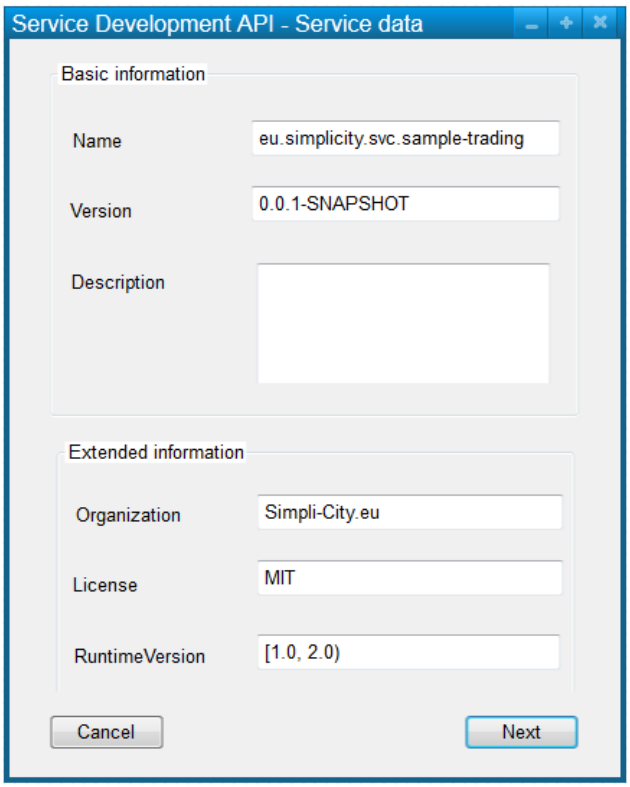

Figure 110: Service Information Mockup User Interface

Figure 111 shows the classification section of service metadata.

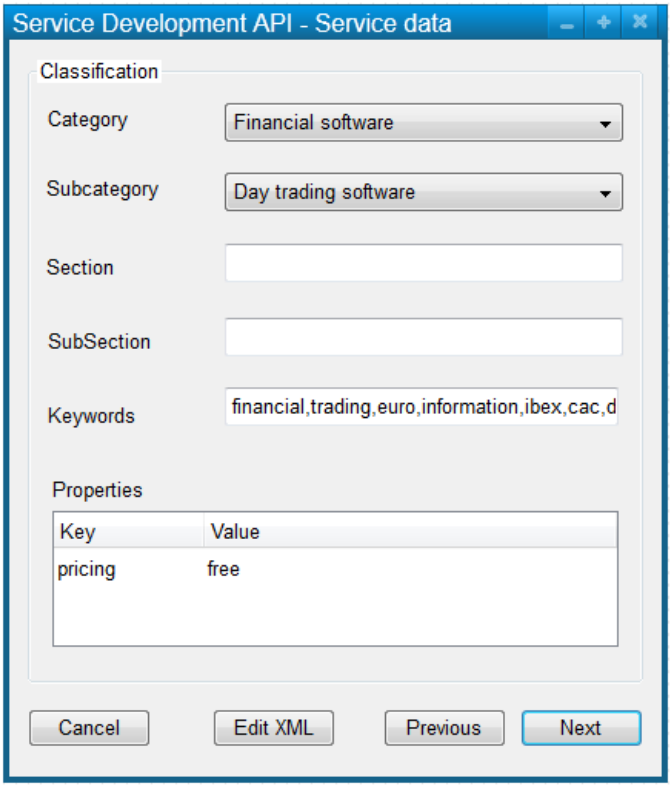

Figure 111: Service Classification Mockup User Interface

8.2.6 Conceptual Data Model

The conceptual data model for backend services is described in Section 9.3.

8.2.7 Parameters to Take into Account for Technical Specification

Table 29: Criteria for Technical Specification

Parameter	Importance (--, -, -+, +, ++)
Generic Criteria	
Up-to-Datedness	++
Stability	++
Extensibility & Open Source/Standards	-+
Familiarity	++
Performance	+
Interoperability	++
Specific Criteria	
Web Service Support	++
Service Metadata Validation	++
Well Documented	++

9 Data Models

9.1 Unified Data Model

The proposed data model for use in the work package WP4 Data Processing component (core to contextualized outcomes) is a Unified Data Model in the form of structured semantic data in order to allow contextualization and reasoning over heterogeneous datasets.

The Unified Data Model has the implication that each of SIMPLI-CITY's heterogeneous data sources undergoes a transformation of its data into a common format whether the data is coming from sensor, personal, user profile or open data. This will ease the integration of technologically heterogeneous data sources into backend services and end user apps and also allow sophisticated data analysis and processing functionalities.

Static and stream data from the different data sources can be combined using vocabularies and ontologies to produce a contextualized common semantic structured tree. This facilitates the project requirements and operations across the combined datasets such as merging, filtering, aggregating, correlation and splitting to fulfil user and service contextualized requests.

One proposed semantic data format that can be used is the Resource Description Framework (RDF). Such a format would allow the necessary level of spatio-temporal querying needed to do reasoning over the SIMPLI-CITY datasets. Thus all relevant datasets coming from the various SIMPLI-CITY data sources as part of a request to the Data Processing component would be transformed to the selected semantic data format.

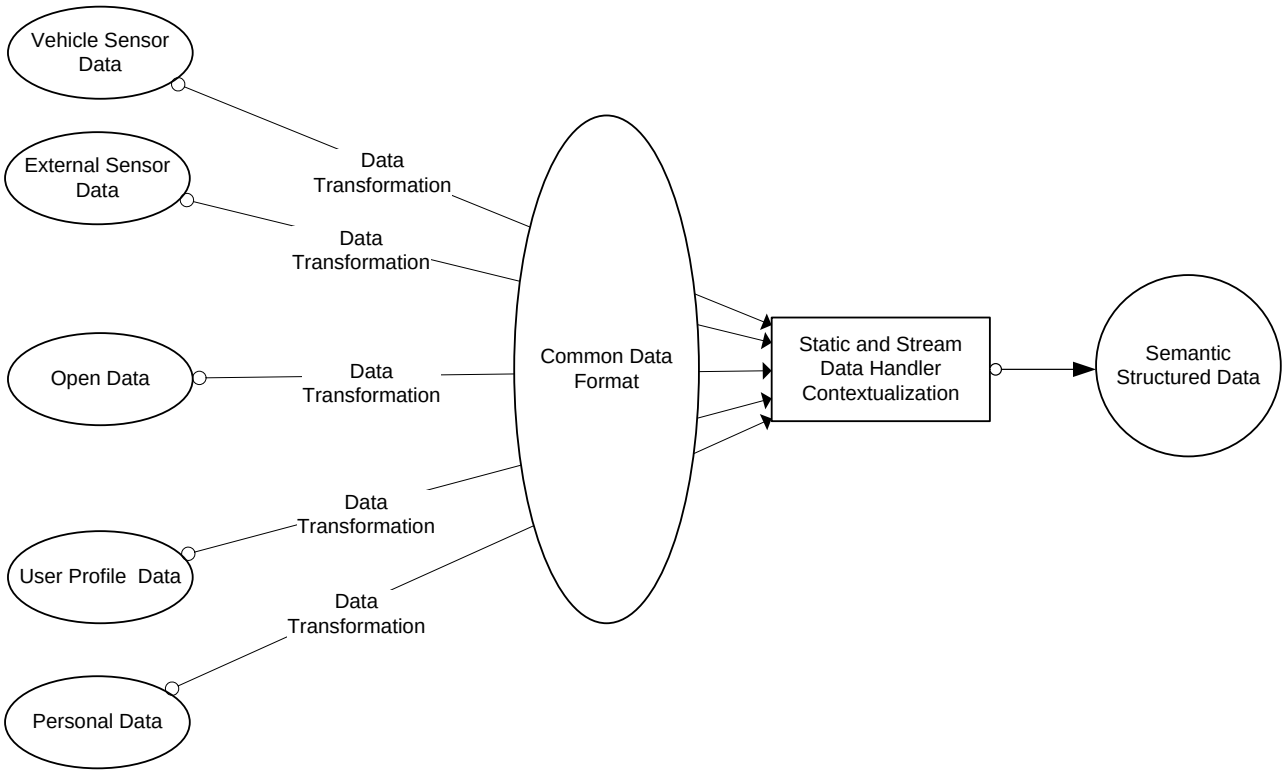

Figure 112: Unified Data Model – Overview

The figure above shows the data path from the various SIMPLI-CITY data-source types resulting in a structured semantic data tree giving a Unified Data Model.

9.2 Data Communication Model

In contrast to the semantic data format used for contextualization and reasoning, the data communication format for transferring data between the distributed components of SIMPI-CITY can be a different more lightweight format. This format needs to be platform-independent in order to be usable between the geographically distributed components of the SIMPLI-CITY systems. One such communication format which would facilitate this is the JavaScript Object Notation (JSON) format used with a RESTFUL service application server. This would facilitate data exchange between distributed components and make use of secure transfer protocols for transfer between, e.g., the Data Processing and Cloud-based Information Infrastructure components or the Service Runtime Environment and Sensor Abstraction Interfaces, amongst others.

9.3 Backend Service Model

In order to manage, configure (and even run) SIMPLI-CITY backend services, some administrative information and service metadata is required. For this, it is necessary to describe the services as well as establish a common format for the actual software artefacts (see Section 6.4.6). The latter heavily depends on the technical specification of the Service Runtime Environment and will therefore be described in deliverable D3.2.2. In Table 30, the different fields of the backend service model will be described. The fields depicted below are inspired by the Maven Project Object Model (POM) file format[4], and are grouped in a semantic manner. Notably, this list is not exhaustive and will be further defined within the Technical Specification deliverable D3.2.2.

Table 30: Preliminary Backend Service Data Model

Field Name	Comments
Basic Data	This mandatory group contains the basic information to identify and describe the service.
Extended Data	This optional group contains the information to identify the developer/organization responsible of this service and the applied license, helping the consumer of the service to choice whether this service suits its needs.
Service Marketplace Integration Data	This optional group contains information related with the publication of the service in the Service Marketplace. It contains information to indicate if the service is publishable in the Service Marketplace or not. It may also include licensing information.
Functional Specification	This optional subgroup addresses the classification aspects for service search. The classification could be done based on the Debian Package Tags[5] or some software classification[6]. Furthermore, an according SIMPLI-CITY taxonomy could be drafted during the course of the project or a fitting taxonomy could be reused.

[4] http://maven.apache.org/pom.html
[5] http://packages.debian.org/about/debtags
[6] https://en.wikipedia.org/wiki/Application_software#Application_software_classification

Field Name	Comments
Export Data	This optional group, inspired from WSDL specification, addresses the technical aspects of the integration of a service as a data and functionality provider. In order to provide any functionality to other services, it must be published through a set of interfaces, exposing the low-level details about how to integrate such services. Note: An engineering task force could define a standard set of service interfaces. Based on Android's Permissions Manifest model[7], these interfaces should enumerate and reference common functionalities provided by SIMPLI-CITY services, like, e.g., geo-location functionalities and secure communications.
Service Level Agreement	This group defines the Service Level Agreement (SLA) to be applied in the context of any group with SLA enforcement.
External Provider Integration Data	This optional group defines information for the proxy-based service integration of backend services as data or functionality providers. Proxy-based services are easily defined by this metadata, but neither binary code nor data is attached. This lack of binary code is the main difference if compared to SIMPLI-CITY internal backend services.
Import Integration Data	This optional group defines a set of requirements (alias dependencies) that should be fulfilled at runtime. By "requirements" it is meant to be references to the abstract big blocks of SIMPLI-CITY: services, sensors and data. The proposed service lookup mechanism should take into consideration the following aspects: • Version: Should match against the newest service that conforms with the specified version range. • SLA: Should match against the service that provides at least the specified SLA. The rationale behind the explicit definition of requirements is mainly ahead-of-time validation, aside of other benefits like performance gains and human-readable requirements, of the presence of such dependencies (meeting the described lookup mechanism) before the service is being run, notifying interested parties (user, service market place, etc.) about any problem.

The following listing shows a preliminary draft for the backend service data model using XML notation:

```
<?xml version="1.0" encoding="UTF-8"?>
<service
      xmlns="urn:simpli-city.eu/Mobility-Services-Framework/Service-Development-
API/0.0.1"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="
      urn:simpli-city.eu/Mobility-Services-Framework/Service-Development-
API/0.0.1 http://simpli-city.eu/documents/xsd/msf-sdapi-0.0.1.xsd
      urn:simpli-city.eu/Mobility-Services-Framework/classification/0.0.1
http://simpli-city.eu/documents/xsd/msf-clan-0.0.1.xsd
```

[7] http://developer.android.com/reference/android/Manifest.permission_group.html

```
      urn:simpli-city.eu/Mobility-Services-Framework/sla/0.0.1 http://simpli-
city.eu/documents/xsd/msf-sla-0.0.1.xsd
      ">

      <!-- Basic data -->
      <name>eu.simplicity.svc.sample-trading</name>
      <version>0.0.1-SNAPSHOT</version>
      <description xml:lang="en">This is a service sample service
            definition that does nothing but simulate a daily trading news
            software</description>
      <!-- Extended data -->
      <organization>Simpli-City.eu</organization>
      <license>MIT</license>
      <runtimeVersion>[1.0, 2.0)</runtimeVersion>

      <classification
            xmlns="urn:simpli-city.eu/Mobility-Services-
Framework/classification/0.0.1">
            <category>Financial software</category>
            <subcategory>Day trading software</subcategory>
            <keywords
xml:lang="en">financial,trading,euro,information,ibex,cac,das,fts</keywords>
            <properties>
                  <pricing>free</pricing>
            </properties>
      </classification>

      <export>
            <interface>
                  <description>Company related information</description>
                  <class>eu.simplicity.sample-trading.interfaces.Company</class>
                  <type>application/xml</type>
                  <frequency>once-a-week</frequency>
            </interface>
            <interface>
                  <class>eu.simplicity.sample-trading.interfaces.Stock</class>
                  <type>application/EDIFACT</type>
                  <!-- new data every 3 minutes -->
                  <frequency>3m</frequency>
                  <sla xmlns="urn:simpli-city.eu/Mobility-Services-
Framework/sla/0.0.1">
                        <class>background</class>
                  </sla>
            </interface>
      </export>

      <import>
            <service>
                  <name>com.company.some-service</name>
                  <version>1.2</version>
            </service>
      </import>
</service>
```

9.4 App Manifest Model

An app is described via the Manifest file stored in the App Registry (see Section 6.5). 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. In the following table and listing, a preliminary draft of the App Manifest data model is presented. Notably, this list is not exhaustive and will be further defined within the Technical Specification deliverable D3.2.2.

Table 31: 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).

The following listing shows a preliminary draft for the app Manifest file data model using XML notation:

```
<?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>
```

10 Conclusion

This deliverable defines the Functional Specification of the SIMPLI-CITY software framework both with regard to the single software components and their interactions. Based on the Requirements Analysis Report (deliverable D2.2) and the Global Architecture Definition (deliverable D3.1), the components and their subcomponents have been further defined and refined where necessary. Furthermore, a check was performed with regard to how the single subcomponents help to meet the demands as defined within the Requirements Analysis Report. As an advancement of the former specifications, the Functional Specification has closed existing gaps and helped to provide a deeper understanding of the interactions between the components. The specification has been done in a technology-agnostic way, i.e., a preselection of software and technologies has not been performed in most cases.

Starting from a system overview from the Global Architecture Definition (D3.1) and a basic explanation of the interaction between the high-level software components of SIMPLI-CITY, the functional specification for the three major Research, Technology, and Development (RTD) work packages and the respective software components has been performed, i.e., for the SIMPLI-CITY Data Integration (“Mobility Data as a Service”), the SIMPLI-CITY Mobility Service Framework, the Personal Mobility Assistant, and the SIMPLI-CITY Developer Support for app and service developers. For this, the single software components have been formally described using the FMC notation and the interactions with other software components have been depicted. Furthermore, the interactions between the subcomponents as well as external components have been described using UML sequence diagrams. The description of these interactions is a very important prerequisite for the identification and specification of component interfaces (in terms of methods used by other components) in the Technical Specification.

Based on the subcomponent specification and depiction of interactions, selection parameters have been identified for every single component, which will be the foundation for the technology selection to be performed as part of the Technical Specification. The list of parameters is made up from generic criteria, which apply to all components, and specific criteria, which define requirements based on the envisioned functionalities of each component. In addition, the necessary data models have been preliminary defined and, if applicable, user interface mockups have been provided.

In detail, the following components have been further discussed:

The Data Processing component provides generic data (pre-)processing functionalities and allows to access Open Data repositories. The Sensor Abstraction and Interoperability Interfaces provide access to sensor data and therefore allow integrating data from these sources in services. In addition, the PMA-based Sensor Abstraction offers a similar, yet more lightweight functionality for sensors on the PMA. The Media Data Streams & Data Prefetching Logic component allows integrating media data streams in services and also facilitates prefetching of data sources and services.

The Service Runtime Environment is the core of the SIMPLI-CITY Mobility Services Framework and allows the execution of backend services and the integration of data services. Monitoring and Service Registry are important helper components for the Service Runtime Environment. Context-based Service Personalisation allows the integration and execution of service personalization functionalities into backend services. Services may be offered on the Service Marketplace.

For the realization of the Personal Mobility Assistant, SIMPLI-CITY defines an Application Runtime Environment, i.e., a framework that makes it possible to run and control apps on a mobile device. The user interface to the PMA is made up from the Dialogue Interface and the Multimodal User Interface.

Finally, SIMPLI-CITY offers service and app developers a rich set of tools and supporting documents in the Service Development API and the Application Design Studio.

The Functional Specification is the foundation for the Technical Specification (deliverable D3.2.2). Naturally, during the Technical Specification, aspects may be changed for technical reasons and due to the technology selection performed. The same will most likely happen during the software engineering for the single software components. However, no major changes regarding the defined functionalities and interactions are foreseen.