simpli·city

The Road User Information System Of The Future

WP4 – Mobility-Related Data as a Service

D4.3.2: Sensor Abstraction and Interoperability Interfaces Prototype II

Deliverable Lead: TUDA

Contributing Partners: TUV, CRF

Delivery Date: 06.07.2015

Dissemination Level: Only for Programme Participants

Version 1.00

This deliverable describes the work carried out during the development of the first prototype of the Sensor Abstraction and Interoperability Interfaces component of the SIMPLI-CITY platform. It specifies the scope of this first version and the degree of fulfillment of the requirements to be covered by the component. It specifies how to install and execute the different subcomponents implemented.

This project has received funding from the European Union's Seventh Framework Programme for research, technological development and demonstration under grant agreement no 318201

Document Status	
Deliverable Lead	Daniel Burgstahler, TUDA
Internal Reviewer 1	Kristof Kipp, ASC
Internal Reviewer 2	Freddy Lecue, IBM
Type	Deliverable
Work Package	WP4: Mobility-Related Data as a Service
ID	D4.3.2: Sensor Abstraction and Interoperability Interfaces Prototype I
Due Date	31.03.2015
Delivery Date	06.07.2015
Status	For Approval

Document History	
Contributions	V0.1, Daniel Burgstahler, TUDA 16.03.2015, Added document structure V0.2, Daniel Burgstahler, TUDA 31.03.2015. Added Sections 1-2 V0.3, Daniel Burgstahler, TUDA 02.04.2015. Added Sections 3-4 V0.4, Daniel Burgstahler, TUDA 08.04.2015. Added Sections 5-7 V0.5, Daniel Burgstahler, TUDA 14.04.2015. Modified Sections 2-7 V0.6, Daniel Burgstahler, TUDA 20.04.2015. Modified Sections 2-7 V0.7, Daniel Burgstahler, TUDA 21.04.2015. Added input from CRF V0.8, Freddy Lecue, IBM 12.05.2015. Internal Review. V0.9, Kristoff Kipp, ASC 13.05.2015. Internal Review. V0.9.1, Daniel Burgstahler, TUDA 17.05.2015. Integrated Internal Review. V0.9.2, Daniel Burgstahler, TUDA 18.05.2015. Integrated Internal Review. V0.9.3, Daniel Burgstahler, TUDA 21.05.2015. Integrated Internal Review. V0.9.4, Daniel Burgstahler, TUDA 22.05.2015. Integrated Internal Review. V0.9.5, Daniel Burgstahler, TUDA 25.05.2015. Integrated Internal Review. V1.0, Daniel Burgstahler, TUDA 27.05.2015. Finalized Document.
Final Version	May 27th, 2015.

Note

This deliverable is subject to final acceptance 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

Atos Worldline, Spain

Centro Ricerche FIAT, Italy

SRM – Reti e Mobilità, Italy

Executive Summary

This deliverable describes the work carried out during the development of the final prototype of the Sensor Abstraction and Interoperability Interfaces component.

The document starts with an introduction of the Sensor Abstraction and Interoperability Interfaces and its subcomponents and describes the scope of the final prototype.

Afterwards, the degree of fulfilment of each requirement to be covered by the component and specified in the Requirements Analysis Report (D2.3) is described.

In the final version, REST interfaces are provided that can be used by service developers as well as the User Centric & Open Data Access components to get access to a user's PMA sensors as well as sensor data of the user's car.

This document describes how potential users (i.e. service developers and administrators) can prepare, install and execute the different parts of the Sensor Abstraction and Interoperability Interfaces component. For this, a step-by-step process to install and make use of the prototype is provided.

This deliverable D4.3.2 is the final prototype of the Sensor Abstraction and Interoperability Interfaces.

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.

Within this document, the final prototype of the Sensor Abstraction and Interoperability Interfaces Component will be presented. The document accompanies the corresponding software prototype, which is the main content of the deliverable.

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 Services Framework: A next-generation European Wide Service Platform (EWSP) allowing the creation of mobility-related services as well as the creation of corresponding apps. This will enable third party providers to produce a wide range of interoperable, value-added services, and apps for drivers and other road users.
- Mobility-related Data as a Service: The integration of various, heterogeneous data sources like sensors, cooperative systems, telematics, open data repositories, people-centric sensing, and media data streams, which can be 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

The purpose of this document is to provide the means to use the final prototype of the Sensor Abstraction and Interoperability Interfaces component and to exploit its functionalities. For this, the scope and requirements of this component and this prototype, the requirements and preparations for users and developers, an installation and usage guide are provided.

The final Sensor Abstraction and Interoperability Interfaces prototype is the outcome of the discussions and implementation work done in project months 9 to 30. It provides the final implementation of the functionalities of the Sensor Abstraction and Interoperability Interfaces as provided with SIMPLI-CITY deliverables D3.2.1 (Functional Specification), and D3.2.2 (Technical Specification). The deliverable D4.3.1 is superseded by this deliverable D4.3.2, which provides the second and therefore final prototype of the Sensor Abstraction and Interoperability Interfaces.

1.3 Document Status and Target Audience

This document is listed in the Description of Work (DoW) as “Public”. This is the final prototype and contains therefore the work of SIMPLI-CITY task T4.3.

1.4 Abbreviations and Glossary

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

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

1.5 Document Structure

This deliverable is broken down into the following sections:

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

Section 2 provides an overview of the scope of the prototype, showing where the Sensor Abstraction and Interoperability Interfaces component fits into the overall SIMPLI-CITY software framework and the outcome of the final prototype. Furthermore, an assessment of the requirements covered by this prototype is given.

Section 3 presents the requirements and preparations to be done by software developers if they want to make use of the Sensor Abstraction and Interoperability Interfaces component.

Section 4 states information about the installation and deployment of the provided software package.

Section 5 describes how software developers can use the provided functionalities.

Section 6 describes the integration of different car sensors into the Sensor Abstraction and Interoperability Interfaces component via OBD-II diagnose interface for cars in general and via Uconnect for specific cars from FIAT.

Section 7 explains in detail the required steps for the usage of the sensor simulation application on the PMA. This allows developers can make use of it and tests the own software with simulated sensors if no connected car is currently available.

Finally, Section 8 provides a summary of the document.

2 Prototype Scope and Requirements Coverage

This section describes the scope of the final prototype and the coverage of the requirements that were defined in deliverable D2.3. To begin with, some general information about the Sensor Abstraction and Interoperability Interfaces is given.

2.1 Sensor Abstraction and Interoperability Interfaces – General Information

The Sensor Abstraction and Interoperability Interfaces provide the means to integrate external sensor data sources, e.g., data sources that are available via an Internet network connection. As these sensors and the corresponding sensor data are usually rather heterogeneous, for example ranging from different parking space data to data from traffic lights, etc., a homogeneous access method has to be provided for this data to be of any use. A possibility to access open data is described in detail in Section 5 of deliverable D4.1.2. Furthermore, an integrative access method eases the efforts for software/service developers who want to exploit these data sources. The function of the Sensor Abstraction and Interoperability Interfaces component is to provide this integrative access. This component will provide the seamless integration of heterogeneous sensor sources and sensor readings within SIMPLI-CITY, e.g. by providing corresponding wrappers.

The Sensor Abstraction and Interoperability Interfaces also provide to other SIMPLI-CITY server components access to sensors connected to the PMA and user-related data, accessible on the PMA, e.g. contacts and calendar data. This functionality is supported through the PMA-based Sensor Abstraction, which is in fact a subcomponent of the Sensor Abstraction and Interoperability Interfaces. This subcomponent is executed as a background service on the PMA and is responsible for allowing access from the server-side Sensor Abstraction and Interoperability Interfaces component to the local data sources on the PMA. This background service is part of the Application Runtime Environment (ARE), described in deliverable D6.3.2. However, the background service is also available as stand alone application within the provided code of this deliverable. The component directly interacts with the server-side Sensor Abstraction and Interoperability Interfaces and provides Java interfaces to the local running apps to access local sensors and sensors of directly connected devices on the PMA, e.g., the car and its sensors. This allows locally running apps to access all connected sensors in a unified way and to receive data in a common data format. Another main functionality of the PMA-based Sensor Abstraction component is the interaction with the previously mentioned server-side Sensor Abstraction and Interoperability Interfaces. The PMA-based Sensor Abstraction provides the complex functionality to integrate the local sensors into the server-based interfaces. Thus, this component provides the functionality to request sensor data of the PMA or connected devices to the server-based components with very low latency, i.e. much less than a second, while being independent of the network and currently used network address of the PMA.

Figure 1 shows the location of the Sensor Abstraction and Interoperability Interfaces in the SIMPLI-CITY Global Architecture. The covered components within the Sensor Abstraction and Interoperability Interfaces are highlighted by vibrant colours. For the full Global Architecture, refer to deliverable D3.1.

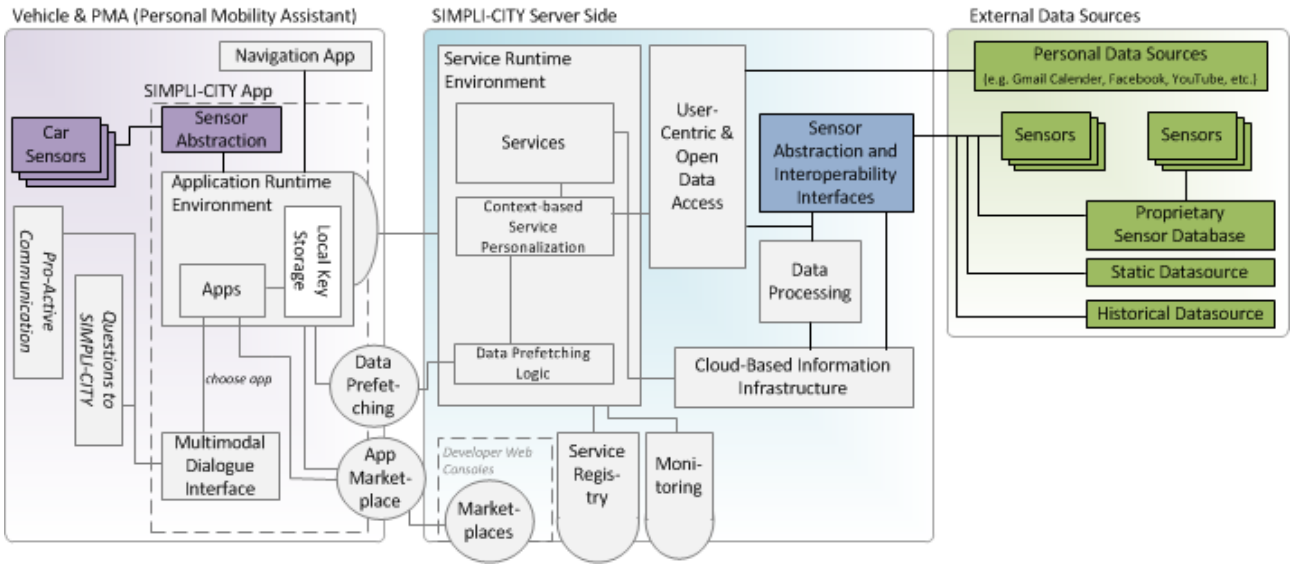


Figure 1: Location of the Sensor Abstraction and Interoperability Interfaces in the SIMPLI-CITY Global Architecture

2.2 Scope of the Final Prototype

The final prototype of the Sensor Abstraction and Interoperability Interfaces provides all functionalities that have been defined and announced in the technical specifications (deliverable D3.2). This includes all interfaces and their functionalities that are provided to other SIMPLI-CITY components. Furthermore, the prototype realizes the interaction between the server side and the PMA side of the Sensor Abstraction and Interoperability Interfaces. Additionally, this prototype provides the server side access to all local PMA sensors, including sensors of a connected car, as well as access to user data from the contacts and the calendar of the user.

Figure 2 depicts the status of development of the final prototype of the Sensor Abstraction and Interoperability Interfaces component, showing the subcomponents that are covered within this prototype.

The status of the implementation is shown using the following colour codes:

- Green: Fully implemented.
- Orange: Partially implemented.
- White: No implementation so far.

As explained in Quarterly Management Report 2013/Q4, partner WORLDLINE had a shortage of resources since the former chief developer for SIMPLI-CITY left the company and it was not possible to replace him in time for this deliverable. Hence, the RDF to JSON Data Wrapper Component has not been covered within the first prototype of T4.3. This shortcoming will be balanced by quick integration of this functionality in the next project-internal prototype for T4.3. It should be noted that postponing this feature has not substantially decreased the overall functionality, envisioned for the first prototype and the prototype is still fulfilling the expected outcome.

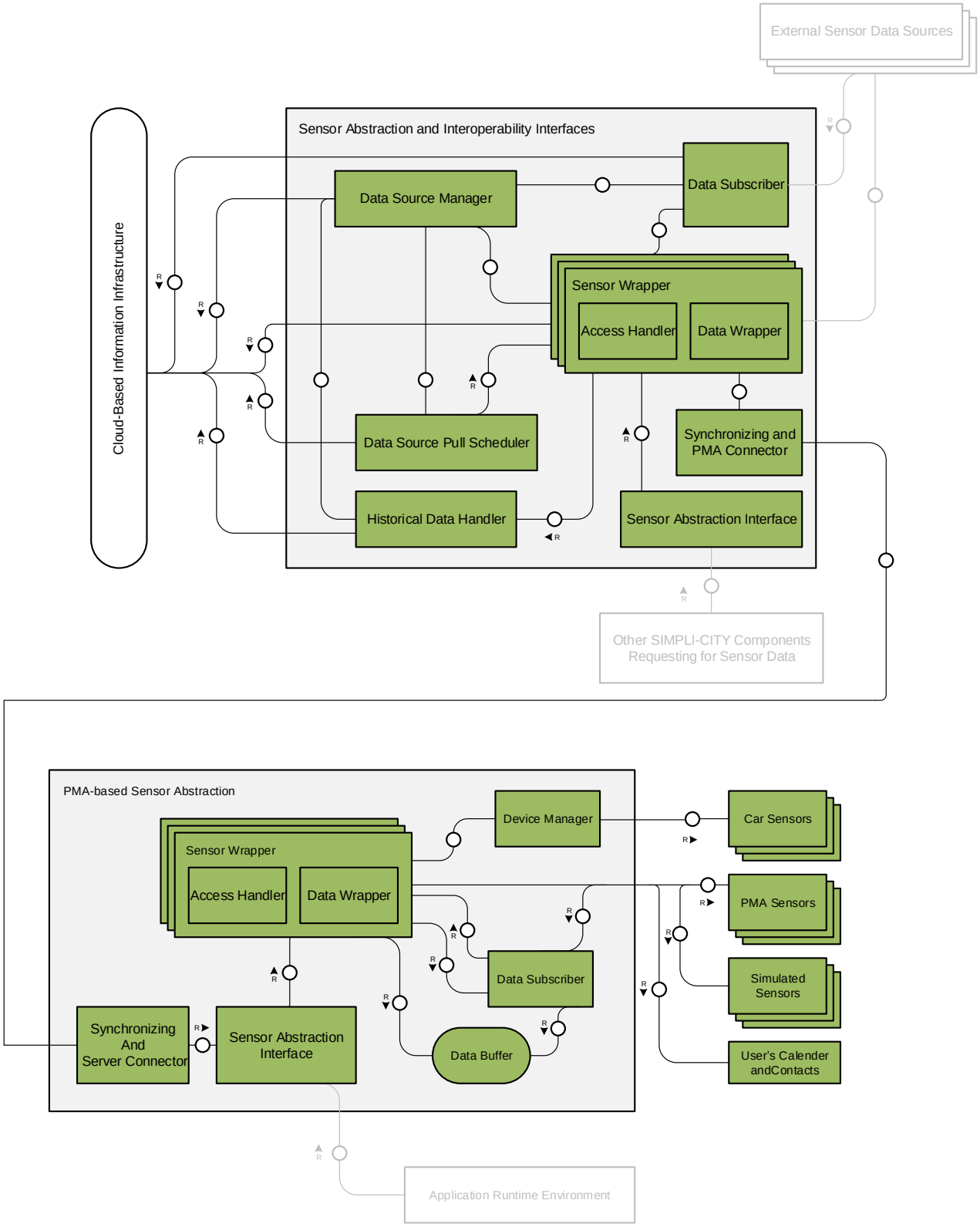


Figure 2: Scope of the Final Prototype of the Sensor Abstraction and Interoperability Interfaces Component

In the following subsections, the scope and status of the single subcomponents of the Sensor Abstraction and Interoperability Interfaces (as depicted in Figure 2) are discussed in more detail. It should be mentioned that the depicted components Cloud-based Information Infrastructure, Application Runtime Environment, External Sensor Data sources and other SIMPLI-CITY Components are not part of the Sensor Abstraction and Interoperability Interfaces. For the Functional Specification and Technical Specification of these subcomponents, refer to SIMPLI-CITY deliverables D3.2.1 and D3.2.2, respectively.

2.2.1 Synchronizing and PMA Connector / Server Connector

For the communication between the server side and the PMA-based components of the Sensor Abstraction and Interoperability Interfaces, two complementary components are responsible: The Synchronizing and PMA Connector on the server side and the Synchronizing and Server Connector on the PMA side. These components have two major functionalities. On the one hand, they are responsible for synchronizing some of the PMA or user-related data with a local buffer on the PMA, to make it available even if the PMA is offline. On the other hand, they are responsible for establishing a connection between the PMA and the server.

These components provide the means to allow the server to access the local sensors of the PMA. This allows the server to send sensor requests to the PMA, i.e., sending a command to the PMA that triggers the event to send the respective requested sensor information to the server. For this, a publish subscribe based push notification with message queuing is used. On the server side a request for a sensor value is encapsulated in a command message and sent to the input queue of the PMA. The response is then sent into a message queue on the server side where the complementary component retrieves this information. To realize the message queuing, the components make use of Apache ActiveMQ. This allows an asynchronous communication approach. To reduce the energy consumption of the notification functionality, the components makes use of the MQTT messaging protocol[1].

In the first prototype almost all functionalities of this sub-component were already implemented. The missing functionality of the synchronization of personal user data and the handling of historical data has been added in the final prototype.

2.2.2 Sensor Abstraction Interface

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

On the PMA side of this component are the APIs that are exposing the interfaces to the Application Runtime Environment and therefore allows apps to exploit data from device-internal data sources. These APIs are provided as Java interfaces.

In the final prototype all functionalities of this component are implemented. All interfaces are available as described in deliverable D3.2.2.

[1] http://mqtt.org

2.2.3 Data Source Pull Scheduler

The Data Source Pull Scheduler takes care of pulling data from sensors according to a pre-defined time interval. This interval can be defined by an administrator by the use of the web configuration interface described in Section 5.2. Pulling for sensor data is optional and will be used for sensor data sources that are only available at certain times or in case of that historical data of a sensor data source should be available. The time intervals for pulling the external sensor data are configured individually for each source via the web interface described in Section 5.2. The retrieved data will be stored in the Cloud-based Information Infrastructure for later usage.

In the final prototype, this component is covered. A web interface is provided which allows selecting sensors that will be pulled in a selectable time interval.

2.2.4 Data Subscriber

This component is responsible for receiving data, triggered by an external event. It provides the functionality to subscribe to external event bus systems and allows receiving event triggered sensor readings. On the PMA side, such data is buffered in the Data Buffer (a local buffer, c.f Figure 2) until the value is requested via the Sensor Wrapper or a new value is provided by a subsequent event.

In the final prototype, this component is covered via the Publish-Subscribe Middleware of D5.2.2. It provides an interface that allows to register sensors. Either the sensor system sends a message on its own to the middleware if a requested special event occurred or the middleware pulls in a defined interval the sensor data and validates if the requested special event took place.

2.2.5 Device Manager

The Device Manager will act as a control subcomponent to interact with external data sources. It is responsible for managing and establishing a wireless connection to the external devices, e.g., the user's car.

In the final prototype this component is covered to handle the access to the car sensors via the on-board diagnose interface and also via Uconnect. Uconnect is the current infotainment platform of FIAT that is supported within this project. The component is able to connect to all local sensors, as well as car sensors that are available via the on-board diagnose interface (OBD-II) of the car. Via Uconnect sensors of specific FIAT cars can be gathered. Here only the CAN node (B-CAN) is read so that no impact on safety is caused.

2.2.6 Historical Data Handler

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

This component is covered in the final prototype and accessible via the RESTful interface at the server.

2.2.7 Data Source Manager

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

This component is implemented within the final prototype. It provides a web user interface so that itself and its subcomponents can be configured.

2.2.8 Sensor Wrapper

The Sensor Wrapper is the component to access sensor data sources and translate the external data format into the SIMPLI-CITY data format. It mainly consists of two subcomponents. The Access Handler is responsible for accessing the external sensor data source. The 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. The Sensor Wrapper is also used by other components of the Sensor Abstraction and Interoperability Interfaces, e.g. the Data Subscriber, to convert the received external data into the common SIMPLI-CITY JSON data format.

In the final prototype, the PMA-based local data sources are invariant implemented. The server side subcomponent is covered in such a way that it can handle free configurable external sensor data sources and also the incoming values from the PMAs.

2.2.9 Simulated Sensors

Simulated sensors are provided for testing purposes.

In the final prototype this component is implemented by providing an App for the PMA with a simulation of sensors values of the car that are accessible over OBD-II and Fiat interfaces and of the PMA. This topic is handled in Section 7.

2.3 Covered Requirements

This section describes the degree of fulfilment of the requirements to be covered by the Sensor Abstraction and Interoperability Interfaces component as specified in the Requirements Analysis Deliverable (D2.3) and the Functional Specification (D3.2.1).

Table 1: Requirements Related to the Sensor Abstraction and Interoperability Interfaces Component and their Degree of Fulfilment

Requirement	Degree of Fulfilment	Comment
Must Have Requirements		
U85: Interaction with car sensors	100%	Fully covered in the final prototype. A set of car sensors is available via the on-board diagnose interface of the car, this is generic for all car types. Another set of sensors is available via Uconnect (c.f. Section 2.2.5) for specific cars of the cooperation partner FIAT.
U86: Transparency, U87: Confidentiality - Do not give away data to third parties, U88: Data encryption, U89: Certification - Only certified apps are allowed to access users data	100%	Communication technology has been chosen that supports SSL transport layer encryption. For debugging reasons these features are not activated in the final prototype implementation, but they are foreseen and the switch to encrypted communication can be done with low to medium effort. Furthermore anonymisation is used to transmit sensor data without referring to a specific user.
U90: Availability, U91: Integrity, U92: Secure access to system, U93: Third party access to the system	100%	These security issues were taken into account for the storage of sensor data, which is saved at a distinct database that does not contain direct information of the corresponding users. This way it is impossible to extract personal user data out of stored information.
U108: Access to smart device sensors	100%	Completely covered in the final prototype. The sensors are available to the ARE in the PMA and the access is forwarded to the server-side.

U109: Access to remote sensors	100%	Covered in the final prototype. PMA, car sensors and external sensor data sources are available via the interfaces on the server side.
U114: Configuration of the frequency of update of the data from data sources	100%	This is covered with the Data Source Pull Scheduler that can be configured with the Data Source Manager over a web interface.
U120: Handle data streams	100%	This is covered in the final prototype by the use of the Publish-Subscribe Middleware of D5.2.2.
U191: Simulation of sensors	100%	The simulation of sensors has been covered on the PMA side for the sensors of the PMA and the car.
U189: Unified interface for accessing sensors	100%	Completely implemented in the final prototype. The server-side interfaces, as well as the PMA side interfaces, are available as specified in deliverable D3.2.
Should Have Requirements		
U107: Access to sensors of the vehicle	100%	Covered in the final prototype. A set of car sensors is available via the on-board diagnose interface of the car, this is generic for all car types. Another set of sensors is available via Uconnect for specific cars of the cooperation partner FIAT.
Could Have Requirements		
U83: Interaction with head up display	0%	Not covered in the prototype.
U110: Remote control of car components, e.g. air conditioning, heating, battery charge timing	0%	Not covered in the prototype.

3 Preparations

This section provides information about what potential users need to prepare in order to use the functionalities of the delivered prototype.

The server side part of the Sensor Abstraction and Interoperability Interfaces component will be installed by administrators of the SIMPLI-CITY Mobility Services Framework.

3.1 Server Side (System Administrators)

In order to make use of the final prototype of the Sensor Abstraction and Interoperability Interfaces the SIMPLI-CITY Service Runtime Environment (SRE) has to be installed on a server since the Sensor Abstraction and Interoperability Interfaces component is integrated into this SRE as a service bundle. Thus the Java SE Development Kit 7 from Oracle and Apache Maven 3.0.5 need to be installed on a Linux system. In addition, port 8080 has to be available for the SRE as this port will be used for the Service Registry. If this port is not available, the Service Runtime Environment will not work properly. The installation of the SRE is described in deliverable D5.3.3.

In addition, a message broker capable of the protocol MQTT is needed to create the link between the server and the PMA. During development the *Mosquitto*[2] message broker was chosen as dedicated application running on the same machine like the SRE. For the final prototype the message broker can be located on any machine that is connected to the internet. The address of the message broker has to be configured at following two files:

```
http://simpli-city.eu:9292/d.burgstahler/sensorabstraction/tree/master/ pma-
connector/src/main/resources/OSGI-INF/blueprint/blueprint.xml
http://simpli-city.eu:9292/d.burgstahler/sensorAbstractionPMA/tree/master/
SensorAbstractionService/res/values/config.xml
```

Furthermore, the SRE requires a valid Internet connection to be able to establish a connection to the PMA. Otherwise the component will not provide any functionality.

The final prototype was tested on a Linux machine running Ubuntu 12.04, Java(TM) SE Runtime Environment (build 1.7.0_51-b13) and Apache Maven 3.0.5. Thus this is the recommended configuration.

In addition, it is required that the environment variable JAVA_HOME is defined. This can be done for example by editing the .bashrc file on the Linux server and extending with the following command:

```
export JAVA_HOME=jdk-install-dir

export PATH=$JAVA_HOME/bin:$PATH
```

After editing this file a new Terminal session has to be started to become these changes working.

[2] http://mosquitto.org

3.2 PMA Side

On the PMA side, an Android device has to be set into the debug mode to be able to deploy the PMA-based component of the Sensor Abstraction and Interoperability Interfaces that consist of an Android App for testing purpose and a background service, which is integrated in the ARE that is further described in D6.3.2.

The final prototype was tested on a Google Nexus 4 Android smartphone with software versions from 4.4.2 to 5.0. Thus these are recommended configurations.

To deploy the software component, an Eclipse installation with working Android SDK integration is needed. Within the Android SDK the complete Android 4.4.2 (API 19) should be installed.

Furthermore, following file (c.f., Listing 1) has to be edited before deployment to configure the address of the SRE and the MQTT message broker:

```
http://simpli-city.eu:9292/d.burgstahler/sensorAbstractionPMA/
tree/master/SensorAbstractionService/res/values/config.xml
```

Listing 1: Example of the config.xml File

```
<?xml version="1.0" encoding="utf-8"?>
<resources>

    <!-- Occurrence of MQTT and Http in ...mqtt.MqttThread and ...auth.LoginActivity -->
    <!-- MQTT Message Broker-->
    <!-- <string name="default_broker_url">tcp://xyz:port</string> -->
    <!-- <string name="default_broker_url">tcp://172.16.16.27:1883</string> -->
       <!-- <string name="default_broker_url">tcp://78.7.100.67:1883</string>  -->
    <string name="default_broker_url">tcp://130.83.245.97:5419</string> <!-- broker@KOM -
->
    <!--  <string  name="default_broker_url">tcp://192.168.0.13:5419</string>  -->  <!--
no_internet -->

    <!-- Http REST interface on server for registration-->
    <!-- <string name="default_broker">http://172.16.16.27:8080</string>  -->
       <!-- <string name="default_broker">http://78.7.100.67:8080</string> -->
    <string name="default_broker">http://130.83.245.97:4321</string>
    <!--  <string  name="default_broker">http://192.168.0.13:8080</string>  -->      <!--
no_internet -->

    <!-- Target of contacts sync operation -->
    <string name="contact_sync_url">%1$s/cxf/pmaconnector/syncContacts</string>

    <!-- MQTT Topics -->
    <string name="topic_request">PMA_REQUEST</string>
    <string name="topic_response">PMA_RESPONSE</string>
    <string name="topic_register">PMA_REGISTER</string>
</resources>
```

4 Installation (Deployment)

This section provides guidelines on how to install and deploy the final prototype of the Sensor Abstraction and Interoperability Interfaces. All necessary files are provided in the SIMPLI-CITY GIT repository.

4.1 Server Side

As mentioned before, the server-side component of the Sensor Abstraction and Interoperability Interfaces is executed as a bundle within the SRE. This bundle has the name as shown in Listing 2. It can be distributed with the final prototype or built with the source code of the final prototype. To create the bundle with the corresponding source code, it is necessary to open a terminal window and navigate to the respective path of the git repository on the local disk. After that the command “mvn clean install” needs to be executed. This specific Apache Maven command removes old build files, builds a new kar-package file based on the current version of the source code and runs the integrated unit tests.

After the kar-file has been created or the location of it is known, the SRE has to be started as described in deliverable D5.3.3. In the next step the necessary components have to be loaded within the SRE. This is done by the command given in Listing 2 that has to be executed within the SRE terminal. Also the “<path-to-kar-file>” has to be substituted with the respective path on the local disk. In the case of a self-build version the path consists of two parts. The first part is the path to the home directory of the respective user. The second part contains the repository of Apache Maven. In the used version 3.0.5 of Maven the second part is as follows:

```
.m2/repository/eu/simpli-city/eu.simpli-city.sensorabstraction/0.0.1-SNAPSHOT
```

All components necessary for data storage are already included within the installation of the SRE.

Listing 2: Installation of the Server-side components of the Sensor Abstraction and Interoperability Interfaces within the SRE

```
kar:install file://<path-to-kar-file>/eu.simpli-city.sensorabstraction-0.0.1-SNAPSHOT.kar
```

4.2 PMA Side

In the final prototype, the PMA side is implemented as a service that provides the functionality of the respective part of Sensor Abstraction and Interoperability Interfaces. Since this service is integrated into the ARE, in this context only the installation process of the App for testing purposes is shown.

In a first step the App has to be deployed to the Android smartphone. For this the debugging mode has to be activated within Android. This can be done within the Android system settings on the smartphone. To deploy the prototype app the Eclipse IDE can be used. To do this the Android developer tools have to be installed in advance[3]. After the project is loaded into Eclipse the option to execute the app on the connected smartphone is offered once the project is selected to be started as Android application. As dependency

[3] http://developer.android.com/sdk/installing/installing-adt.html

the maven project SCSensors, located at the *sensorabstractioncommon* repository, needs to be on the build path. This is shown in Figure 3.

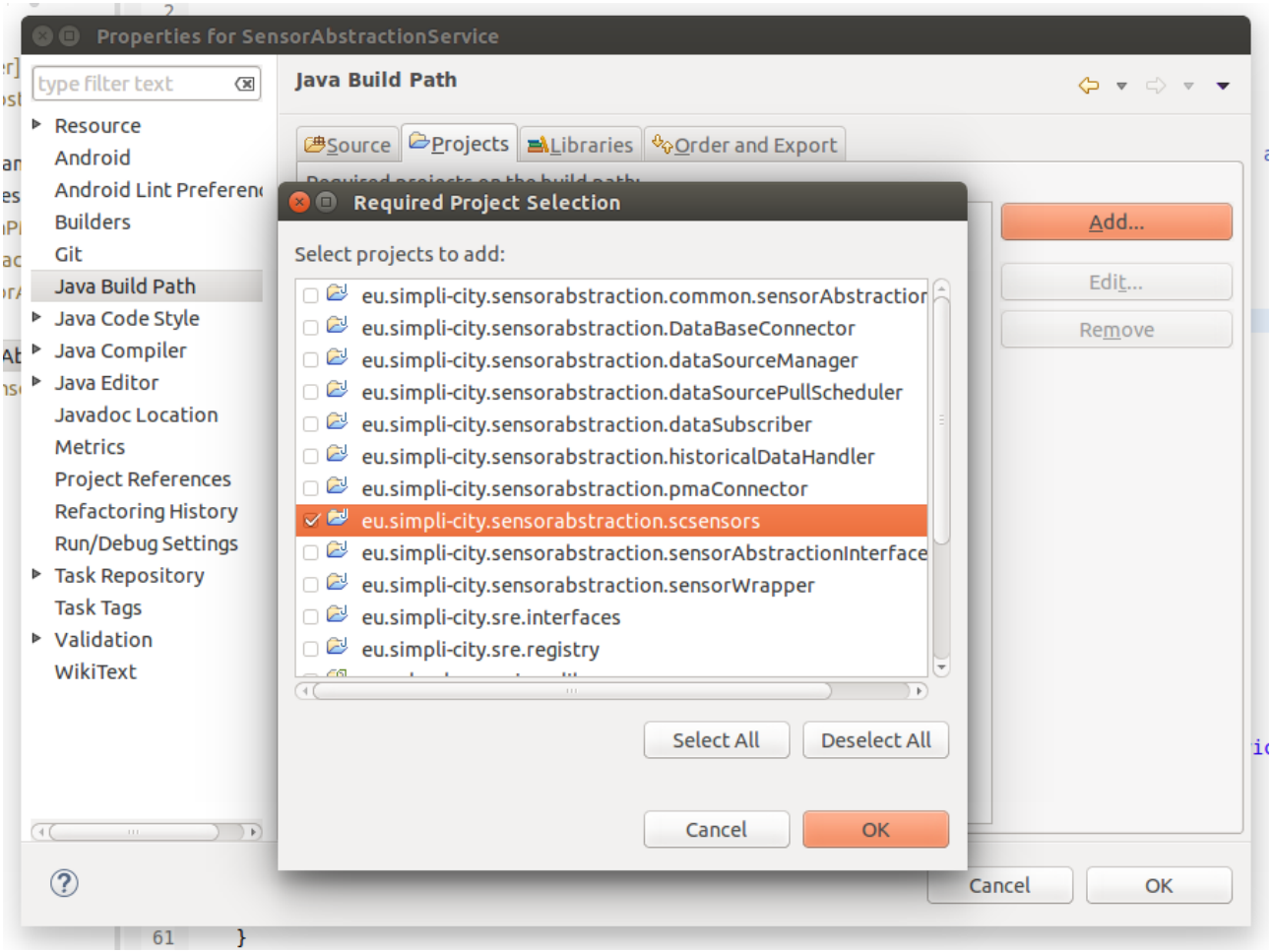
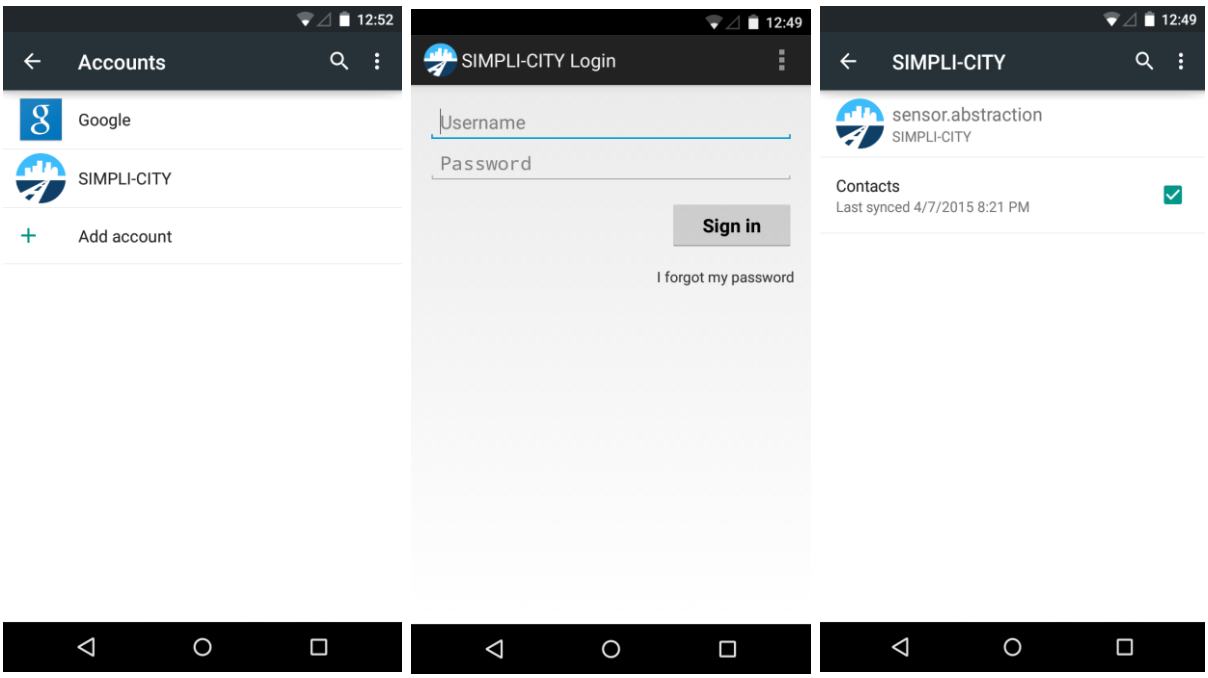


Figure 3: Adding Project Dependency

The SIMPLI-CITY user account has to be added to the Android smartphone. This can be done in the Android settings in the menu accounts as depicted in Figure 4.



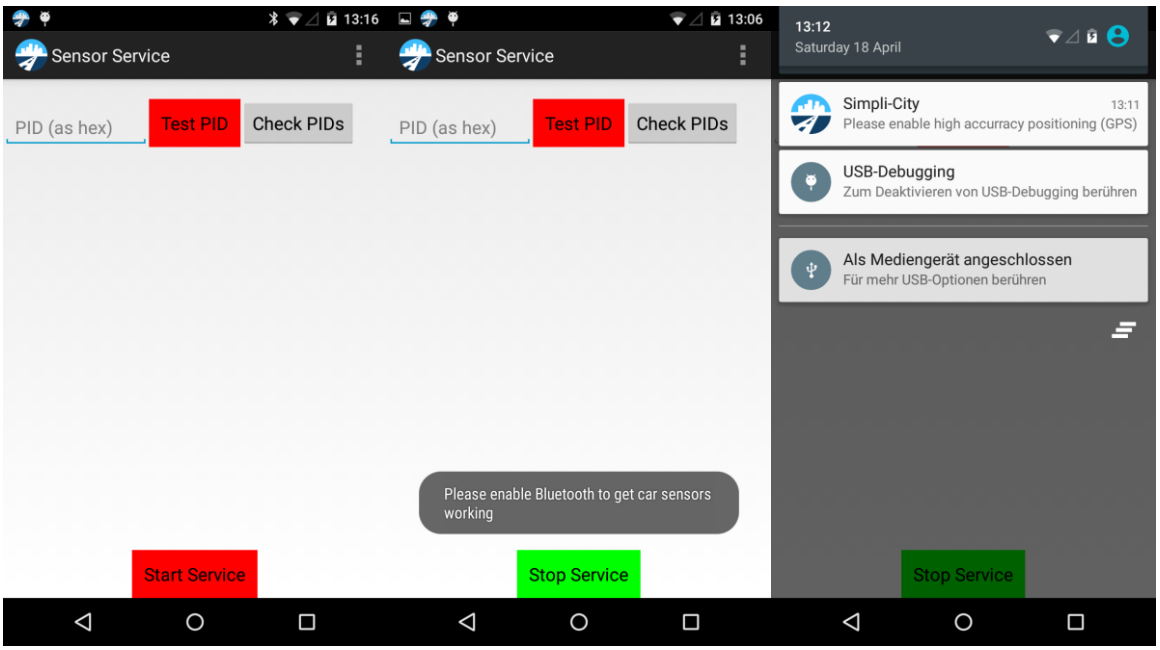
(a) (b) (c)

Figure 4: SIMPLI-CITY User Account Management on the PMA

Figure 4 shows the SIMPLI-CITY user account management on the PMA. To add a new account, the SIMPLI-CITY user name and the respective password has to be entered (b). Afterwards, the user account is listed as in the first picture (a). Clicking on it leads to a description of further account properties and settings (c). This user account has to be created via the provided web interface of the server components (c.f. Section 5.2).

In the next step the App has to be started. The App provides a very simple user interface as can be seen in Figure 5, since the App is only a container to start the PMA-based Sensor Abstraction and Interoperability Interfaces component that consists of a background service for remote requests and a second background service for local requests via Java API. This App provides a *start* button to start the background service, which handles requests from the server component and is therefore a long running operation. The second service is exclusively running on invocation of the local Java API. Additionally, single sensors of a connected car can be requested by their Parameter ID and the results are displayed below the edit field for debugging purposes. A connected car is indicated when the colour of the button “Test PID” turns into green. In the demo App the user also gets informed to activate GPS and Bluetooth functionality (this is necessary to connect to the OBD car connector, c.f. Section 6.1).

After the start of the background service, all sensors are available via the RESTful interfaces provided by the server side component of the Sensor Abstraction and Interoperability Interfaces. Thus PMA sensors can be accessed via the server-side. The prototype App is just for demo purposes, the service is also available via the ARE.



(a) (b) (c)

Figure 5: Simple Demonstrator App of the PMA-based Sensor Abstraction and Interoperability Interfaces Component

Figure 5 shows the simple demonstrator App of the PMA-based Sensor Abstraction and Interoperability Interfaces Component at the start (a), start of the service and notification that Bluetooth needs to be activated (b). A connection to the SRE is only granted when the colour of the button “Stop Service” turns into green. If the PMA is powered via USB a notification appears that advises the user to activate the highest precision location service (GPS) if not already done (c).

5 Execution and Usage of the Software

5.1 Usage of the REST Interfaces

This section describes how to use the different subcomponents of the prototype. The core scope of the final prototype was the completion of the PMA-based component and the server based component of the Sensor Abstraction and Interoperability Interfaces that have not been finished in the first prototype. Once the demo Application on the PMA is started as described in Section 4.2 or the ARE is started (since this service is contained in both), the PMA-based Sensor Abstraction and Interoperability Interfaces component automatically connects to the counterpart component on the server side and automatically maintains the connection. The server provides a RESTful interface to request sensor information from the PMA. For a simple test purpose, any web browser can be used to request this RESTful interface.

In the following, the workflow for a test request of data from the PMA is explained in all necessary steps. This example uses the server address 130.83.245.97 and the port 4321 for the RESTful interfaces.

The demonstration workflow for a test execution is based on a sensor request from the users PMA. Before a sensor can be requested one has to know the respective unique ID of the sensor. All sensors belong to a device that is in case of PMA sensors the PMA itself. Thus the first step is to request the information about this device. To do so a dedicated RESTful interface is provided to request information about a user's PMA. However, this interface needs the unique ID of the user as input parameter. Services and Apps that are using the Sensor Abstraction and Interoperability Interfaces might know this ID but for testing and debugging purpose it might be necessary to request such an ID directly. Therefore the RESTful interface *login*, that is also internally used during the account login process at PMA side can be utilized, which is depicted in Listing 4 (It has to mentioned here that this description is just a demonstration workflow). The input parameters are the user name and the respective password. While in the first prototype users were added manually to the server side database, the final prototype provides a web-based user management is provided which can be accessed as follows in Listing 3.

Listing 3: Login, Registration and Administration of the Users

```
http://130.83.245.97:4321/cxf/users/login
```

Through this webpage an admin can login and create, delete and update users. Also new users can register themselves with a new account and also update their profiule information. Furthermore, an admin can add external sensors and the Data Source Pull Scheduler can be configured through the web-based interface that will be explained in 5.2.

Listing 4: Request of the User ID

```
http://130.83.245.97:4321/cxf/pmaconnector/register?username=simplicityuser1&password=gmsmplcty1
```

The response contains the unique user ID, which is necessary as input parameter to request data from the PMA of a specific user as mentioned before. With this knowledge another RESTful interface, as depicted in Listing 5, provides access to the device information of the users PMA. While making the request and reading the response, no

data is cached no data is cached, all requests are passed to the PMA, and the response is sent back to the server-side. The data structure of devices and sensors, that is the data structure of the response, is described in deliverable D4.1.2. The input parameter is the previously requested user ID.

Listing 5: Request of the User's PMA Information

```
http://130.83.245.97:4321/cxf/sensorabstraction/pma?userID=ba9e2620-aac6-11e3-a5e2-0800200c9a66
```

The return value of the request shown I the listing above is the device description in the SIMPLI-CITY JSON data. In the following Listing 6, the device description of a PMA based on a Nexus 4 is given. The description contains a unique ID of the PMA device. In this example, no sub device (e.g. a car) is connected to the PMA. The available sensors are listed as an array of ConnectedSensor JSON objects that all provide a unique ID. This ID can be used to directly request data from a sensor.

Listing 6: Result to the Request of the User's PMA Information


```
{
   "connectedSensors": [
      {
         "sensorType": {
            "type": "userdata",
            "subtype": "calendar"
         },
         "timeStamp": "2014-05-06T07:49:48Z",
         "objectID": "11b04404-b0b4-4cb3-931a-f94e54cd4004"
      },
      {
         "sensorType": {
            "type": "movement",
            "subtype": "linar_acceleration"
         },
         "timeStamp": "2014-05-06T07:49:48Z",
         "objectID": "141fb8a1-7599-42c0-a287-b9ec06d2899a"
      },
      {
         "sensorType": {
            "type": "movement",
            "subtype": "rotation_vector"
         },
         "timeStamp": "2014-05-06T07:49:48Z",
         "objectID": "69b9e39f-6b48-4769-a9b9-abf5e07b93e0"
      },
      {
         "sensorType": {
            "type": "environment",
            "subtype": "light"
         },
         "timeStamp": "2014-05-06T07:49:48Z",
         "objectID": "13044b4d-9fab-46f1-9154-f087ad702575"
      },
      {
         "sensorType": {
            "type": "environment",
            "subtype": "proximity"
         },
         "timeStamp": "2014-05-06T07:49:48Z",
         "objectID": "793e2009-c220-436e-880f-583fd5201fa1"
      },
      {
         "sensorType": {
```

```
            "type": "environment",
            "subtype": "magnetic_field"
         },
         "timeStamp": "2014-05-06T07:49:48Z",
         "objectID": "1a90842d-872d-45d2-828e-bb5558882d65"
      },
      {
         "sensorType": {
            "type": "environment",
            "subtype": "air_pressure"
         },
         "timeStamp": "2014-05-06T07:49:48Z",
         "objectID": "c9891d78-6c8f-482b-be30-69889b77fba0"
      },
      {
         "sensorType": {
            "type": "userdata",
            "subtype": "contacts"
         },
         "timeStamp": "2014-05-06T07:49:48Z",
         "objectID": "fe550ee4-99ac-4284-bbc9-906c2b572240"
      },
      {
         "sensorType": {
            "type": "environment",
            "subtype": "raw_magnetic_field"
         },
         "timeStamp": "2014-05-06T07:49:48Z",
         "objectID": "69449fde-c9f4-4441-8ab3-e966b4de98b2"
      },
      {
         "sensorType": {
            "type": "movement",
            "subtype": "gyroscope"
         },
         "timeStamp": "2014-05-06T07:49:48Z",
         "objectID": "c74c058b-59e1-4102-aaf8-942ede4a3465"
      },
      {
         "sensorType": {
            "type": "movement",
            "subtype": "game_rotation_vector"
         },
         "timeStamp": "2014-05-06T07:49:48Z",
         "objectID": "ed8291f0-b849-40c9-9c42-de1e287ddd4b"
      },
      {
         "sensorType": {
            "type": "movement",
            "subtype": "gyroscope"
         },
         "timeStamp": "2014-05-06T07:49:48Z",
         "objectID": "f68d8f2b-2ffc-4fe2-8c9a-c29ebc827240"
      },
      {
         "sensorType": {
            "type": "orientation",
            "subtype": "gravity"
         },
         "timeStamp": "2014-05-06T07:49:48Z",
         "objectID": "21146ea4-30e8-4ab6-93c6-05e760e8f6d5"
      },
      {
         "sensorType": {
            "type": "movement",
            "subtype": "accelerometer"
```

```
         },
         "timeStamp": "2014-05-06T07:49:48Z",
         "objectID": "a569726f-7108-4dde-ac42-40474f88c557"
      },
      {
         "sensorType": {
            "type": "orientation",
            "subtype": "device"
         },
         "timeStamp": "2014-05-06T07:49:48Z",
         "objectID": "5133cb4c-fe3c-435d-94ff-ee54c54da0a9"
      },
      {
         "sensorType": {
            "type": "movement",
            "subtype": "significant_motion"
         },
         "timeStamp": "2014-05-06T07:49:48Z",
         "objectID": "831abff5-ff29-4315-9208-a04f69c5d5ca"
      }
   ],
   "connectedDevices": [],
   "deviceType": "PMA",
   "object": "device",
   "timeStamp": "2014-05-06T07:49:48Z",
   "objectID": "4b168d73-3722-43df-896b-482ac4aba724"
}
```


These listed sensors contain all built-in sensors of the PMA device, as well as the virtual sensors *calendar* and *contacts*. A request of these virtual sensors directly allows requesting personal user data. To request one of the listed sensors, the following RESTful interface can be used as depicted in Listing 7. The input parameter is the unique ID of the respective sensor that is the ObjectID of this sensor (c.f. Listing 6).

Listing 7: Request of a Specific Built-In Sensor

```
http://130.83.245.97:4321/cxf/sensorabstraction/sensor?sensorID=4e7224b1-a6bd-49f7-bfc0-15b20b7db268
```

A request for a device has to be performed as in Listing 8. In this example, the requested device is the PMA from above, therefore the result is exactly the same as in Listing 6.

Listing 8: Request of a Device

```
http://130.83.245.97:4321/cxf/sensorabstraction/device?deviceID=4b168d73-3722-43df-896b-482ac4aba724
```

As an example for a sensor request, in the following Listing 9, the result of a request to the PMAs accelerometer sensor is given.

Listing 9: Result of a Request to the PMA's Accelerometer Sensor


```
{
   "sensorType": {
      "type": "movement",
      "subtype": "accelerometer"
   },
   "sensorValue": [
      [
         {
            "name": "x axis",
            "value": "-6.0993295",
            "unit": "m/s^2",
            "accuracy": {
               "value": "none"
```

```
            }
         },
         {
            "name": "y axis",
            "value": "-1.6651437",
            "unit": "m/s^2",
            "accuracy": {
               "value": "none"
            }
         },
         {
            "name": "z axis",
            "value": "-5.469076",
            "unit": "m/s^2",
            "accuracy": {
               "value": "none"
            }
         }
      ]
   ],
   "object": "sensor",
   "timeStamp": "2014-05-06T07:53:34Z",
   "objectID": "a569726f-7108-4dde-ac42-40474f88c557",
   "parent": "4b168d73-3722-43df-896b-482ac4aba724"
}
```


As an example for personal user data, the following Listing 10 depicts the result of a request of the virtual sensor *calendar* without filtering. The following example only contains two calendar events. A request to the RESTful interface will return an array with all events in the respective user calendar.

Listing 10: Result of a Request to the Calendar Data


```
{
   "sensorType": {
      "type": "userdata",
      "subtype": "calendar"
   },
   "sensorValue": [
      [
         {
            "name": "title",
            "value": "May Day"
         },
         {
            "name": "organizer",
            "value": "en-gb.irish#holiday@group.v.calendar.google.com"
         },
         {
            "name": "description",
            "value": ""
         },
         {
            "name": "dtstart",
            "value": "2015-05-04T00:00:00Z"
         },
         {
            "name": "dtend",
            "value": "2015-05-05T00:00:00Z"
         },
         {
            "name": "allDay",
            "value": "1"
         },
         {
```

```
            "name": "duration"
         },
         {
            "name": "eventLocation",
            "value": ""
         },
         {
            "name": "calendar_displayName",
            "value": "Holidays in Ireland"
         }
      ],
      [
         {
            "name": "title",
            "value": "DSS GROUP Meeting"
         },
         {
            "name": "organizer",
            "value": "sensor.abstraction@gmail.com"
         },
         {
            "name": "description",
            "value": ""
         },
         {
            "name": "dtstart",
            "value": "2014-04-30T14:00:00Z"
         },
         {
            "name": "dtend",
            "value": "2014-04-30T16:00:00Z"
         },
         {
            "name": "allDay",
            "value": "0"
         },
         {
            "name": "duration"
         },
         {
            "name": "eventLocation",
            "value": ""
         },
         {
            "name": "calendar_displayName",
            "value": "sensor.abstraction@gmail.com"
         }
      ]
   ],
   "object": "sensor",
   "timeStamp": "2014-05-06T07:54:08Z",
   "objectID": "11b04404-b0b4-4cb3-931a-f94e54cd4004",
   "parent": "4b168d73-3722-43df-896b-482ac4aba724"
}
```


As mentioned, a request for filtered calendar data can be done. Instead of a HTTP GET request a HTTP POST request needs to be send to the RESTful interface in Listing 11. This is because the message body contains the filter properties as JSON that has to be transmitted to the interface.

Listing 11: Request of Filtered Calendar Data

```
http://130.83.245.97:4321/cxf/sensorabstraction/calendarEvents?sensorID=11b04404-b0b4-
4cb3-931a-f94e54cd4004
```

As can be seen, the sensorID is the ID of the calendar sensor in the PMA request above from Listing 6. Additionally to the correct URL, the body type and content must be set properly. The body type or also called content type has to be set to “application/json” else the server will respond with the status code “415”, which stands for “unsupported media type”. The content of the body has to look as in Listing 12, which is a JSON representation of an array of FilterParameter objects. The result of this request is depicted in Listing 13. The example shows that the filtering searches for matching substrings in a case-insensitive manner. Listing 14 shows the list of keywords that are allowed for the filtering. Multiple different key-value pairs are handled as conjunction. Multiple FilterParameter with the same key are not supported.

Listing 12: Content of the Body for the Request of the Filtered Calendar Data

```
[
    {
        "Key": "title",
        "Value": "may"
    },
    {
        "Key": "organizer",
        "Value": "google"
    }
]
```

Listing 13: Result of a Request to the Filtered Calendar Data

```
{
   "sensorType": {
      "type": "userdata",
      "subtype": "calendar"
   },
   "sensorValue": [
      [
         {
            "name": "title",
            "value": "May Day"
         },
         {
            "name": "organizer",
            "value": "en-gb.irish#holiday@group.v.calendar.google.com"
         },
         {
            "name": "description",
            "value": ""
         },
         {
            "name": "dtstart",
            "value": "2015-05-04T00:00:00Z"
         },
         {
            "name": "dtend",
            "value": "2015-05-05T00:00:00Z"
         },
         {
            "name": "allDay",
            "value": "1"
         },
         {
            "name": "duration"
         },
         {
            "name": "eventLocation",
            "value": ""
```

```
          },
          {
             "name": "calendar_displayName",
             "value": "Holidays in Ireland"
          }
       ]
    ],
    "object": "sensor",
    "timeStamp": "2014-05-06T07:54:08Z",
    "objectID": "11b04404-b0b4-4cb3-931a-f94e54cd4004",
    "parent": "4b168d73-3722-43df-896b-482ac4aba724"
}
```


Listing 14: Possible Calendar Filter Keys

```
title
description
eventTimezone
allDay
duration
rrule
rdate
exrule
exdate
eventLocation
organizer
availability

dtstart
dtend
```

The filter keys “dtstart” and “dtend” perform a special filtering as they are not just matching a substring but be processed in a way so that events later than the value of “dtstart” and earlier than the value of “dtend” get matched. Additionally, it is important to make sure that the values for “dtstart” and “dtend” are given as ISO 8601 time. An example for the respective filter is given in Listing 15.

Listing 15: Example for the Time Span Filter


```
[
       {
              "Key": "dtstart",
              "Value": "1414358813000"
       },
    {
              "Key": "dtend",
              "Value": "1414531613000"
       }
]
```


The contact data can also be requested with additional filtering through the RESTful interface in Listing 16. The content type and body have to be set as in the case for calendar data. The result looks similar to Listing 13. The difference to a request of calendar data is the set of possible filter keys, illustrated in Listing 17. Data is gathered from the cloud database, to which the PMA synchronizes its address-book.

Listing 16: Request of Filtered Contact Data

```
http://130.83.245.97:4321/cxf/sensorabstraction/contact?sensorID=fe550ee4-99ac-4284-bbc9-906c2b572240
```

Listing 17: Possible Contact Filter Keys

```
surname
firstName
title
phoneNumber
street
postalCode
country
email
```

The Historical Data Handler allows it to gather data from a specific sensor in a defined time interval of the past. Every time a sensor gets requested the data gets saved into the cloud database. With the RESTful interface in Listing 18 this data can be requested per sensors ID. The begin and end of the time interval has to be given as ISO 8601 time.

Listing 18: Request of Historical Data

```
http://130.83.245.97:4321/cxf/sensorabstraction/sensorInterval?sensorID=13044b4d-9fab-46f1-9154-f087ad702575&begin=1970-01-01T00:00:00Z&end=2015-04-15T12:41:02Z
```

An example result for the light sensor of the PMA above is given in Listing 19.

Listing 19: Result of a Request of Historical Data of the PMA's Light Sensor


```
[
 {
   "sensorType": {
      "type": "environment",
      "subtype": "light"
   },
   "sensorValue": [
      [
         {
            "name": "luminous intensity",
            "value": "44.0",
            "unit": "lx",
            "accuracy": {
              "value": "HIGH"
            }
         }
      ]
   ],
   "object": "sensor",
   "timeStamp": "2015-03-15T08:08:00Z",
   "objectID": "13044b4d-9fab-46f1-9154-f087ad702575",
   "parent": "4b168d73-3722-43df-896b-482ac4aba724"
 },
 {
   "sensorType": {
      "type": "environment",
      "subtype": "light"
   },
   "sensorValue": [
      [
         {
            "name": "luminous intensity",
            "value": "68.0",
            "unit": "lx",
            "accuracy": {
```

```
                "value": "HIGH"
              }
           }
        ]
    ],
    "object": "sensor",
    "timeStamp": "2015-03-15T12:40:56Z",
    "objectID": "13044b4d-9fab-46f1-9154-f087ad702575",
    "parent": "4b168d73-3722-43df-896b-482ac4aba724"
  }
]
```

As it could be quite interesting to gather information about the immediate environment, the available sensors at a specific location and within a certain radius of the location can be obtained by the RESTful interface given in Listing 20. Parameters of the interface are the latitude and longitude of the location, as well as the radius of the desired area. The unit of the radius is meters. Since some sensors, like the ones of parking lots, were composed to a virtual parent device the result of devicesInArea is a list of Device objects as JSON format. An empty list is returned if no sensors were found in the specified area. An example of an answer is given in Listing 21.

Listing 20: Request for Getting Devices in Area

```
http://130.83.245.97:4321/cxf/sensorabstraction/deviceInArea?lat=49.8746692&long=8.6606771&radius=100
```

Listing 21: Result of deviceInArea RESTful Interface

```
[
  {
    "connectedSensors": [
       {
         "sensorType": {
              "type": "environment",
              "subtype": "humidity"
         },
         "timeStamp": "2015-05-08T16:07:58Z",
         "objectID": "aea131d9-4eb5-4d6f-bba6-d62bb388c744"
       }
    ],
    "connectedDevices": [
    ],
    "object": "device",
    "timeStamp": "2015-05-08T016:07:58Z",
    "objectID": "2de5818b-5032-4c98-957d-6671378466e5"
  },
  {
    "connectedSensors": [
       {
         "sensorType": {
              "type": "parking",
              "subtype": "parking_lot"
         },
         "timeStamp": "2015-05-08T16:07:58Z",
         "objectID": "0787fb9e-9c72-4d6f-9f99-c7c76a0868a5"
       }
    ],
    "connectedDevices": [
    ],
    "object": "device",
    "timeStamp": "2015-05-08T016:07:58Z",
    "objectID": "dc0f1319-40e0-4852-8ca7-32473f46ad8e"
  },
```

```
  {
    "connectedSensors": [
       {
         "sensorType": {
              "type": "environment",
              "subtype": "temperature"
         },
         "timeStamp": "2015-05-08T16:07:58Z",
         "objectID": "3e0735bf-5eae-4d38-ba7c-153399fc3b36"
       }
    ],
    "connectedDevices": [
    ],
    "object": "device",
    "timeStamp": "2015-05-08T016:07:58Z",
    "objectID": "a186b5b9-de08-4c8c-9573-beff39efe4b7"
  }
]
```


The RESTful interfaces for getting a device or PMA information return information about connected sensors and devices. However, the number of connected devices and available sensors may vary since they might be disconnected later in time. This is the case for example for a connected car and its sensors. If a PMA is connected to a car and one requests the respective information from the RESTful interface this might be out of date in a later point in time, when e.g. the car is not connected to the PMA anymore. If one requests device information of the car (by RESTful interface *for getting a device*) or even a specific car sensor, then the return value contains no information about connected sensors and devices in case of that the PMA to which this device was previously connected is connected to the server. This result is given in Listing 22. In case of that the PMA as parent device is temporarily not connected to the SRE server the request will result in a timeout.

Listing 22: Return Value of RESTful Interface *getDevice* if the Device is Temporary not Available


```
{
  "connectedSensors": [],
  "connectedDevices": [],
  "deviceType": "CAR",
  "object": "device",
  "timeStamp": "2014-11-18T23:22:41Z",
  "objectID": "072eccf0-f0d9-4c77-bda7-9e3c02ae1acd"
}
```


In case of a specific sensor is requested but temporary not available because it belongs to a device that is temporarily not connected to the PMA, e.g. a car sensor of a not connected car, then the returned JSON will not contain any values for the requested sensor as given in Listing 23. In case of the PMA as parent device is temporary not connected to the SRE server the request will result in a timeout.

Listing 23: Return Value of RESTful interface *getSensor* if the Sensor is Temporary not Available, e.g. Belonging to a Temporary not Connected Car


```
{
  "object": "sensor",
  "timeStamp": "2014-11-18T23:22:03Z"
}
```

5.2 Usage of Data Source Manager and its Subcomponents

The Data Source Manager provides the means to control the data source subcomponents and allows to manage the user database. With the URL of the Listing 24 a new user can register himself or update his information. The login screen for the end-user is given in Figure 6. However, Not only the user management is handled by this web interface, also administrators can log in and make further configurations.

Listing 24: Login Website for the Data Source Manager

```
http://130.83.245.97:4321/cxf/users/login
```

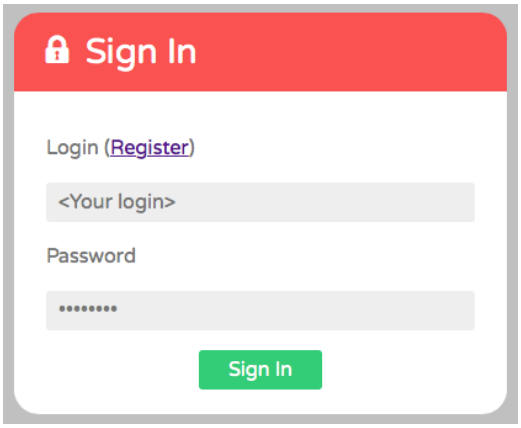


Figure 6: User Login of the Data Source Manager

Figure 7 shows the interface for the external sensor management, which allows adding all kinds of supported sensors. New users can be easily created by the use of the register link within the login screen. The sensor configuration can only be done by the admin user. The configuration tab is hidden for other users. A necessary requirement is that the sensors have a name so that they are clearly recognizable if something needs to be changed. Also the correct sensor type and subtype needs to be selected. The exact location in the format of latitude and longitude are also required required. The final source of this sensor needs to be given and also how the results of the source are formatted so that the correct value can be extracted. If adding an external sensor is successful, the unique UUID of it can be looked up and be requested by the use of the respective interface.

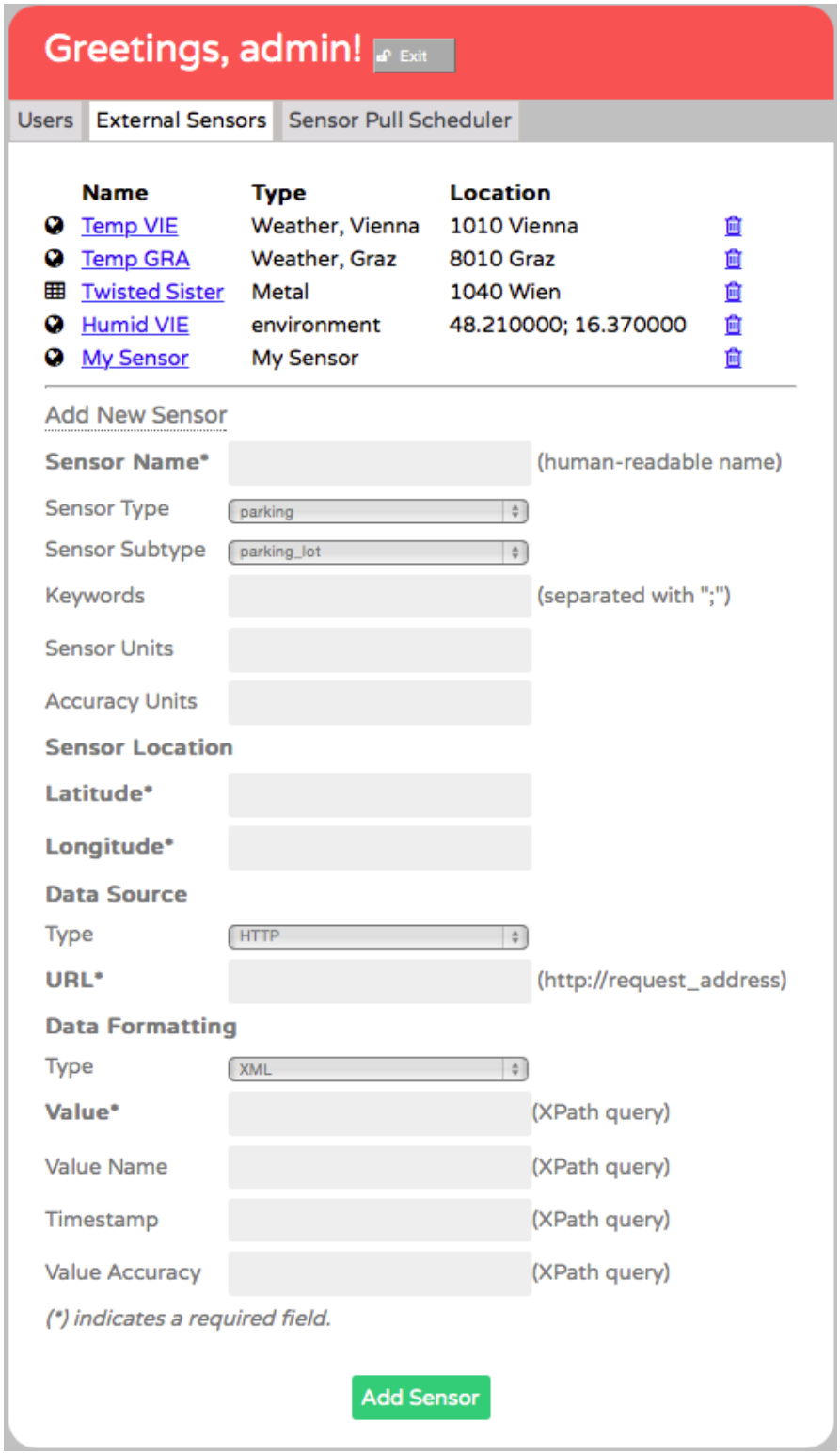


Figure 7: External Sensor Management of the Data Source Manager

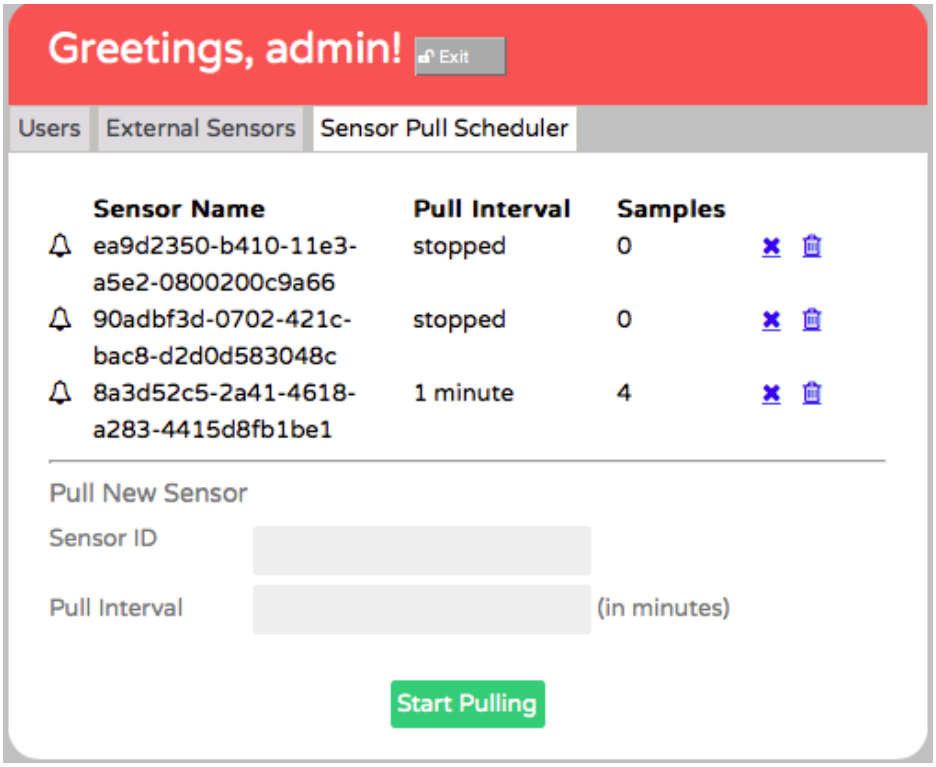


Figure 8: Sensor Pull Scheduler Management of the Data Source Manager

Figure 8 shows the configuration interface of the Sensor Pull Scheduler component. A new task can be created by filling in the sensor ID and the update interval of the sensor to watch. If already a pull task exists for a sensor, then the pull interval gets updated with the new value. A repeated pull task can also be temporary stopped. In the case that the completed task is not needed anymore it can be deleted. The collected data can be obtained through the Historical Data Handler interface (c.f., Section 5.1).

As already mentioned in section 2.2.4 is the Data Subscriber is implemented through the Publish-Subscribe Middleware module in D5.2.2. Therefore a further description of its functionality is redundant in this context. A description of how to install the required software, how to add sensors and how to use the component is described in full detail in D5.2.2.

5.3 Local API

The previously mentioned background service on the PMA enables other SIMPLI-CITY components to access local PMA sensors, car sensors and personal user data in a common way. The provided local API for the ARE is explained in detail in the following. This API is directly available for third-party developers by the provided ARE. To give an early access to the Java API, while integration into the ARE was under construction, a dedicated Android library project is provided. By including the library project into any Android App project access to the backend service and therefore sensor data is given. This allows local applications to make use of the local sensors and car sensors. The provided interface methods are illustrated in Listing 25.

Listing 25: Java Interfaces of the Local API

```
Sensor getSensor(UUID sensorID) throws SensorNotFoundException, ServiceNotAvailableException
Device getDevice(UUID deviceID) throws DeviceNotFoundException, ServiceNotAvailableException
Device getPmaInformation() throws ServiceNotAvailableException
Sensor getContacts(Map<String,String> filterValues) throws ServiceNotAvailableException
Sensor getCalendarEvent(Map<String,String> filterValues) throws ServiceNotAvailableException
```

To make use of these interfaces in an Android App project the first step is to import the library project into Eclipse. This can be done by a right-click in the project explorer, inside the Eclipse IDE. In the context menu the entry “import” has to be selected. Then one has to select *Existing Android Code Into Workspace* as depicted in Figure 9.

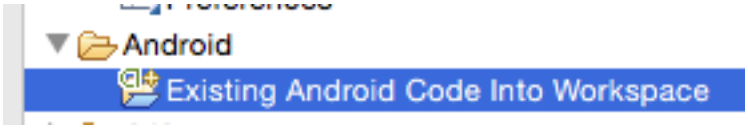


Figure 9: Import Library Project into the Eclipse IDE

Then the project SensorAbstractioLocalAPI has to be chosen, located at the GIT repository sensorAbstractionPMA (git@simpli-city.eu:d.burgstahler/sensorAbstractionPMA.git). Now the local API of the Sensor Abstraction and Interoperability Interfaces can be added to the project properties of an Android project. To do so, one has to right-click the Android project and select “*properties”*. The Android tab on the left pane has to be selected. Inside this pane the add button on the lower right side allows to add libraries. Here the SensorAbstractioLocalAPI has to be added. Afterwards the Android properties of the project should look similar as depicted in Figure 10.

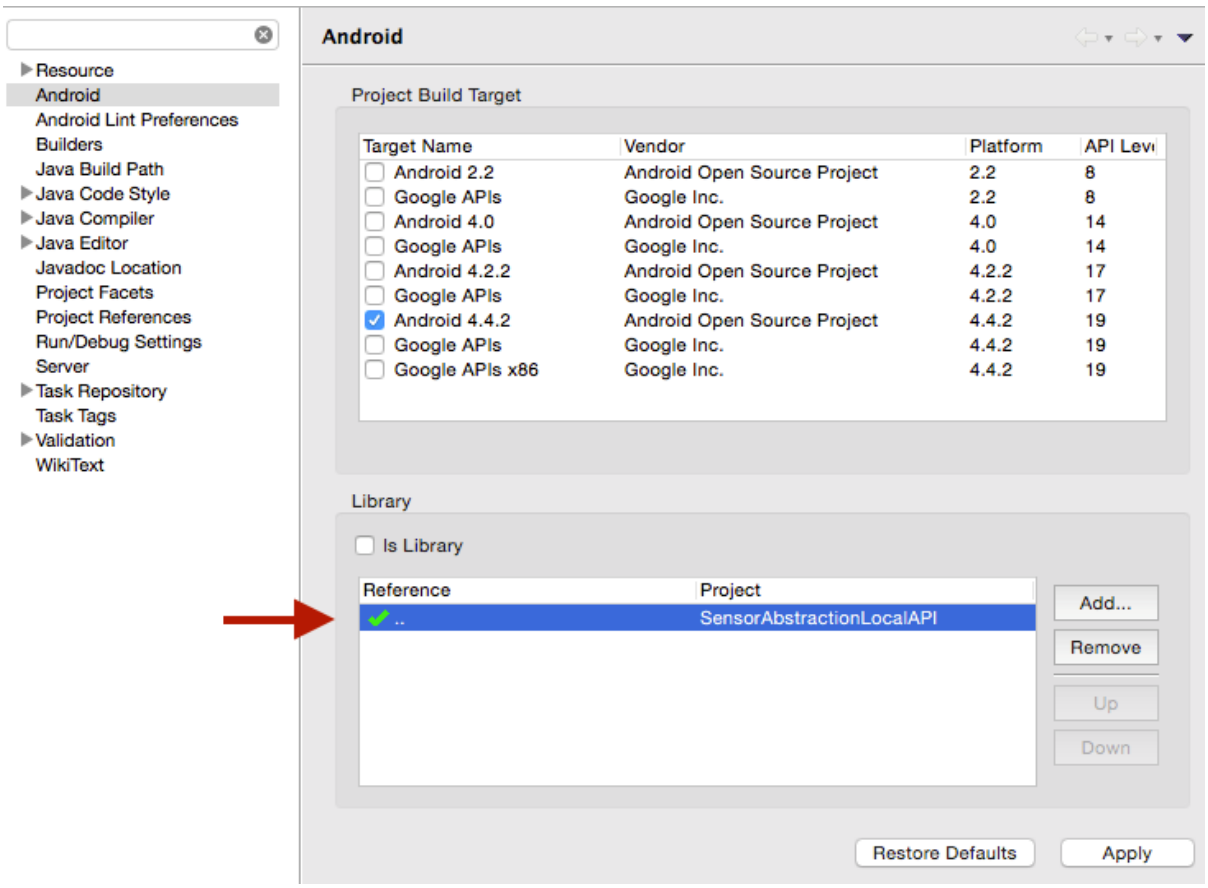


Figure 10: Added SensorAbstractionLocalAPI to the Android Properties

Finally it is possible to use the local API inside the Java code of the Android project. An example of how to use this API is given in the following Listing 26. Ensure the SensorAbstractionService is deployed to the PMA to make the local API work. The service will be deployed automatically with the ARE. In case of using a stand-alone app the Sensor Abstraction Service App has to be deployed in advance by deploying the Sensorabstraction project ass application to the smartphone.

Listing 26: Example How to Use the SensorAbstractionLocalAPI

```
import eu.simplicity.sensorabstraction.SAI;
import eu.simpli_city.sensorabstraction.Device;
import eu.simpli_city.sensorabstraction.Sensor;


…
public void example(){
       SAI con = new SAI(getContext());
       Device pma = con.getPmaInformation();
       Sensor sensor = con.getSensor(s.getID());

//IMPORTANT: Before closeing the app one has to call the close method
       con.close();
}
```

6 Car Sensor Integration

This section describes the two possible ways to access car sensors. These are the OBD-II diagnose interface for almost all cars and Uconnect for specific cars from the FIAT company. To use the OBD-II interface an OBD-Bluetooth adapter is necessary.

6.1 Integration of Car Sensors via OBD-II

OBD is a standardized interface for car diagnostic, in use since the 1980s. It has been evolved to a common used tool. As enhancement the successor, OBD-II has been introduced in the mid 1990s, which is contained in almost all cars nowadays. It is designed for car repair and diagnoses and is also used by car manufacturers to find malfunctioning components in a car. However, since several sensor signals are available by this interface a lot of communities started to use this data for other applications. Nowadays several easy to use adapters are available that use an OBD-II to RS232 chip and provide this way an easy access to this date via Bluetooth or even Wi-Fi.

For the communication between the car and the PMA a Bluetooth adapter is necessary that needs to be attached to the OBD-II port of the car. It gets powered by the OBDII port and starts up automatically the communication with the car. It does also provide itself as a Bluetooth device to which the PMA can be connect. The required steps for the initialization are shown in Figure 11. First one needs to establish the connection via Bluetooth to the adapter and in a second step one has to set the correct protocol and baud rate between the adapter and the OBD-II interface. Within this project an Apos BT OBD 327 OBDII to Bluetooth adapter was used[4]

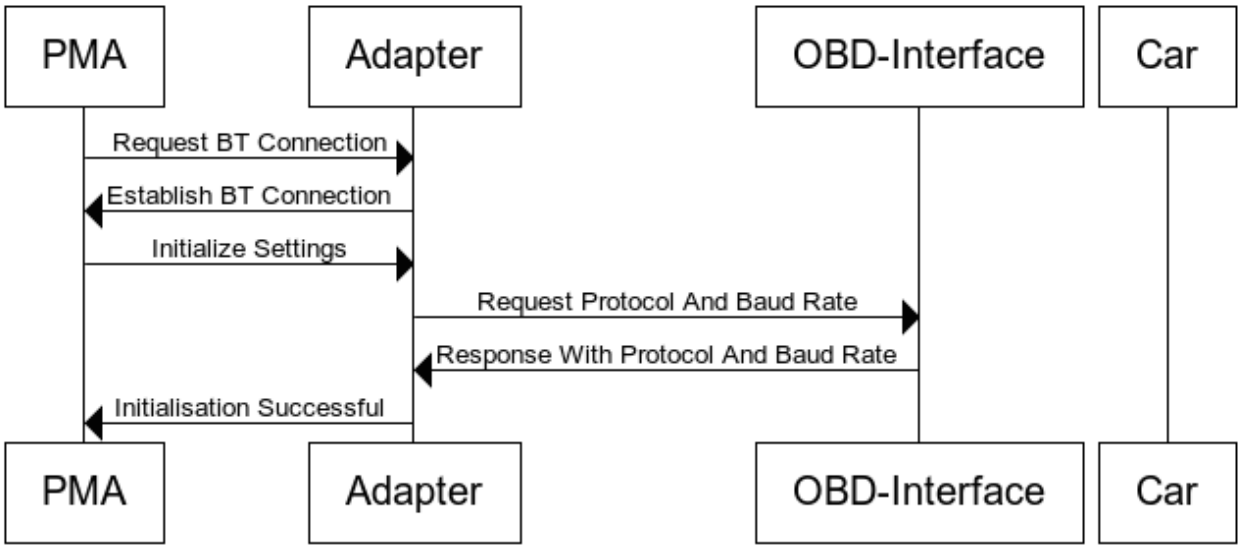


Figure 11: PMA to OBD-II Initialization Data Flow

After the user does manually connect to the Bluetooth device the first time, the service provided by Sensor Abstraction and Interoperability Interfaces does connect automatically to it if it gets detected.

[4] http://www.apos-gmbh.de/OBD/Bluetooth-OBD-Adapter-BT-OBD-327.html?force_sid=e88391d62caee4f1c9c483820c58179f

If the initialization is successful, other applications can see the car as a device of the PMA and therefore also request the available sensors of the car. With the ARE they can actually request for sensors values. In the case of a OBDII-based car sensor data request the process depicted in Figure 12 gets started. Internally the sensor is requested as value of a corresponding parameter ID (PID).

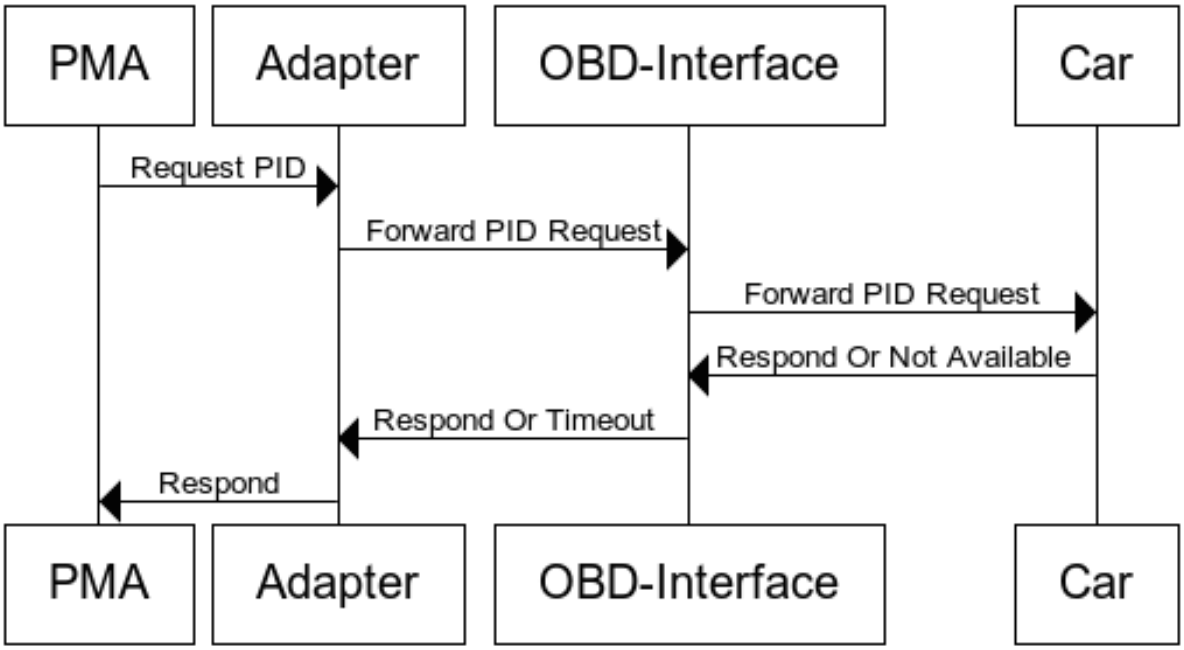


Figure 12: Data Flow for a PID Request

The PID request gets forwarded from the PMA to the car through the components OBD-II adapter and OBD-II interface of the car. Following this each component waits for the next component to answer the request. If the OBD interface does not response to the OBD adapter in time, this will respond to the PMA that no data could be received. Else a response will be send that contains the requested PID and the sensor value.

The following Table 2 shows the PIDs supported by the Sensor Abstraction and Interoperability Interfaces service. This service supports only a subset of all possible PIDs. However, typically a car does not support all possible PIDs and the selected set contains the most supported ones. Anyway, additional PIDs can be easily added as extension. Descriptions of the PIDs are also publically available, e.g., at Wikipedia[5]. The formal specification can be found in SAE J1979.

Table 2: List of Supported PIDs and their Description

PIDS as Hexadecimal	Description
04	CAR_CALCULATED_ENGINE_LOAD_VALUE
05	CAR_ENGINE_COOLANT_TEMPERATURE
06	CAR_SHORT_TERM_FUEL_BANK_1
07	CAR_LONG_TERM_FUEL_BANK_1
08	CAR_SHORT_TERM_FUEL_BANK_2

[5] http://en.wikipedia.org/wiki/OBD-II_PIDs

09	CAR_LONG_TERM_FUEL_BANK_2
0C	CAR_ENGINE_RPM
0D	CAR_VEHICLE_SPEED
0F	CAR_INTAKE_AIR_FLOW_RATE
10	CAR_MAF_AIR_FLOW_RATE
11	CAR_THROTTLE_POSITION
1F	CAR_RUN_TIME_SINCE_ENGINE_START
23	CAR_FUEL_RAIL_PRESSURE_DIESEL_OR_GASOLINE_DIRECT_INJECT
2F	CAR_FUEL_LEVEL_INPUT
31	CAR_DISTANCE_TRAVELED_SINCE_CODES_CLEARED
33	CAR_BAROMETRIC_PRESSURE
45	CAR_RELATIVE_THROTTLE_POSITION
46	CAR_AMBIENT_AIR_TEMPERATURE
47	CAR_ABSOLUTE_THROTTLE_POSITION_B
52	CAR_ETHANOL_FUEL_IN_PERCENT
5A	CAR_RELATIVE_ACCELERATOR_PEDAL_POSITION
5D	CAR_FUEL_INJECTION_TIMING
5F	CAR_EMISSION_REQUIREMENTS_TO_WHICH_VEHICLE_IS_DESIGNED

6.2 Integration of FIAT Car Sensors via Uconnect

The solution provided by CRF allows the car sensors integration for a specific FIAT prototype car equipped with Uconnect, a proprietary telematics platform. With the implemented solution, sensor data gathered from the vehicle CAN bus is serialized and then sent to the mobile device using a Bluetooth connection. Uconnect is indeed the information telematics platform used in all last generation vehicles produced by FCA.

The current version of Uconnect does not support MirrorLink for the communication with mobile devices. MirrorLink is an industry standard developed by the Car Connectivity Consortium (CCC), that currently implements a very limited vehicle interface. Several car manufactures announced their intentions to implement it but FCA currently does not support MirrorLink technology with their products portfolio that is available on the market at the moment.

On the market there are several other proprietary protocols under license; FCA chooses one of these for the current production named Uconnect Access Via Mobile. The advantage of such solutions are that the OEM maintains the design of mobile integration granting safety rules.

Pairing

In order to enable the data exchange, it is necessary to make the pairing between Uconnect and the smartphone.

This operation is performed by means of the On-Board-Unit (OBU) Human-Machine Interface (HMI); after the first time the OBU has started it searches for all devices that are already registered and reconnects to them automatically.

Following Bluetooth profiles are exposed by Uconnect:

- HFP (Hands Free Profile)
- A2DP (Advanced Audio Distribution Profile)
- PBAP (Phonebook Access Profile)

For security reasons, Uconnect does not allow third party applications to use the serial port profile (SPP) to exchange data with the paired devices.

The experimental on board unit used for the SIMPLI-CITY prototype provides only the possibility to create a pipe between the OBU and the paired device. The pipe is used for establishing the communication with the PMA: on the PMA a dedicated gateway application provides the sensor values by the use of an UDP socket on local loopback.

Data flow

The process followed for providing car data from a FIAT vehicle to the PMA on the smartphone involves the following components running on the three connected devices:

- On the vehicle: B-CAN bus
- On the OBU: Sensor collector, JSON formatter, Smartphone notifier
- Smartphone: Android receiver, CRF Car Sensor

The communication between them is shown in Figure 13. Also in the following a part of these single components get further explained.

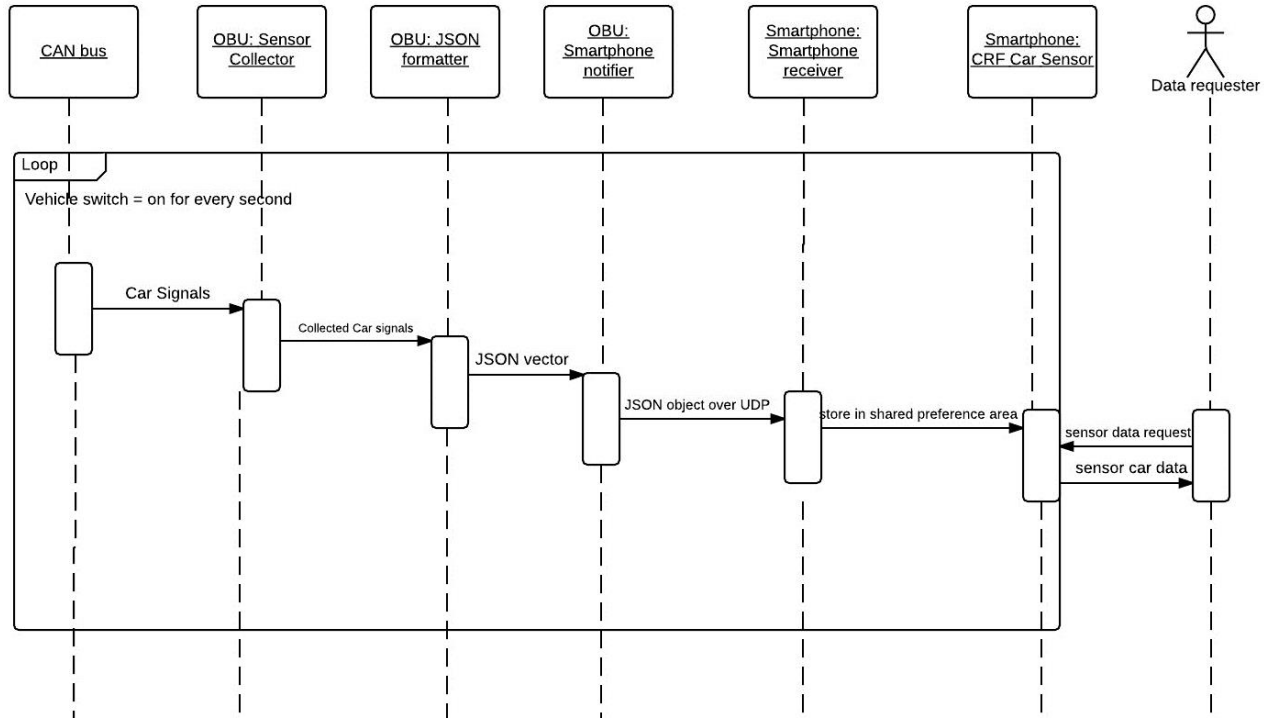


Figure 13: Car Sensors Integration Data Flow

Sensor collector: The sensor collector uses the API so that it reads every second the selected car signals from the CAN bus (bus at 1 Hz).

The Uconnect API provided by the SDK is able to read only messages from the CAN node (B-CAN) that don't have an impact on safety and that don't need a high transmission rate. The corresponding car sensors with their Id and description are shown in Table 3. Furthermore, a reading rate of 1 Hz has been evaluated adequate for the provided signal types.

Table 3: List of Car Sensors and ID Used in the JSON Vector Sent to the PMA

Signal/Id	Description
CAR_AUTONOMY_DISTANCE = 0	Days left before service
CAR_DISTANCE_TO_SERVICE = 1	Distance left before service
CAR_AVERAGE_FUEL_CONSUMPTION_APLUS = 2	Instant fuel consumption for Trip A
CAR_INSTANT_FUEL_CONSUMPTION = 3	Instant fuel consumption
CAR_AVERAGE_SPEED_APLUS = 4	Average speed calculated for Trip A
CAR_SPEED = 5	Indication about the vehicle speed
CAR_BATTERY_VOLTAGE_LEVEL = 6	Value of the voltage of the battery
CAR_DAYS_TO_SERVICE = 7	Days left before service
CAR_ENGINE_SPEED = 8	Indication about the engine speed in rpm
CAR_FUEL_LEVEL = 9	Indication about fuel level
CAR_KEY_STS = 10	Status of the key (Off, On, CrankOn, Stop)
CAR_LOW_FUEL_WARNING_STS = 11	Indication about the emergency fuel level is detected
CAR_DRIVER_DOOR_STS = 12	Indication about the status of the driver's door (open/close)
CAR_PASSENGER_DOOR_STS = 13	Indication about the status of the front passenger's door (open/close)
CAR_RHR_DOOR_STS = 14	Indication about the status of the rear passenger's door (open/close)
CAR_TEMPERATURE_UNIT = 15	Current temperature unit °C/F
CAR_EXTERNAL_TEMPERATURE = 16	External temperature
CAR_TOTAL_ODOMETER = 17	Total of the kilometers/miles performed by the car
CAR_PARTIAL_ODOMETER_APLUS = 18	Total of the kilometers/miles performed by the car for Trip A
CAR_ACCELERATION_ECO_INDEX = 19	Acceleration score (performance) calculated by the

	CRF/FIAT FIAT Eco:Drive algorithm (1 - bad value / 5 - great value)
CAR_DECELERATION_ECO_INDEX = 20	Deceleration score (performance) calculated by the CRF/FIAT FIAT Eco:Drive algorithm (1 - bad value / 5 - great value)
CAR_GEAR_SHIFTING_ECO_INDEX = 21	Gear change score (performance) calculated by the CRF/FIATFIAT Eco:Drive algorithm
CAR_SPEED_ECO_INDEX = 22	Speed score (performance) calculated by the CRF/FIAT algorithm FIAT Eco:Drive algorithm
CAR_TOTAL_ECO_INDEX = 23	Total score (performance) calculated by the CRF/FIAT algorithm FIAT Eco:Drive algorithm based on the performances of other EcoIndex value
CAR_GPS_LATITUDE = 24	GPS latitude
CAR_GPS_LONGITUDE = 25	GPS longitude

JSON formatter: It receives the car signals provided by the sensor collector and builds the JSON vector used by the Android receiver.

The information, contained in the JSON provided on the OBU, is then used by the Android receiver to build the JSON according to the SIMPLI-CITY format.

The initial version used an implementation of the protocol buffers data format. Doing some tests, it was noted that it was possible to use directly JSON, which solved some bugs that occurred in the previous version and made the data flow more efficient. A further advantage is that on the Android side a single common library can be used to de-serialize the data that is received by the car and then serialize it again to supply the SIMPLI-CITY format.

The JSON sent to the smartphone is a vector containing the pair of key/value: (ID_SENSOR, Value). An example of JSON sent to the smartphone is provided in the below.

Listing 27: Example of JSON Sent to the PMA

```
{"sensors":[{"id":"0","value":"250"},{"id":"1","value":"0"},{"id":"2","value":"0"},{"id":
"3","value":"11.2"},{"id":"4","value":"0"},{"id":"5","value":"0"},{"id":"6","value":"0"},
{"id":"7","value":"0"},{"id":"8","value":"0"},{"id":"9","value":"0"},{"id":"10","value":"
4"},{"id":"11","value":"0"},{"id":"12","value":"0"},{"id":"13","value":"0"},{"id":"14","v
alue":"0"},{"id":"15","value":"0"},{"id":"16","value":"12"},{"id":"17","value":"3200"},{"
id":"18","value":"0"},{"id":"19","value":"12.55"},{"id":"20","value":"0.1"},{"id":"21","v
alue":"0.2"},{"id":"22","value":"0"},{"id":"23","value":"0.41"},{"id":"24","value":"45.01
12855"},{"id":"25","value":"7.5665653"}]}
```

Smartphone notifier: This component is the client that is connected to the smartphone. It serializes the JSON provided by the JSON formatter and sends it to the Android receiver using an UDP socket.

Android receiver: This component is the server that is connected to the OBU. It receives the packets and sends them in broadcast to the *CRF Car sensor*.

CRF Car sensor: It de-serializes the received JSON and save the sensor values in the shared preferences area.

When the server requests the car sensors value, the SIMPLI-CITY data format is built using the values saved in the shared preferences area.

This way, a buffering is not needed because the last values are always used.

7 Preparation and Execution of the Simulator

The Sensor Simulator is a separate app for the PMA and is contained in the GIT repository or alternatively in the source code of the final prototype of the PMA-based components of the Sensor Abstraction and Interoperability Interfaces. This simulator App is able to simulate PMA sensors, OBD car sensors as well as CRF specific car sensors. The simulation is based on a virtual journey. Start point and destination have to be entered before the simulation can be started. Then a virtual car drives a calculated route in realistic speed.

7.1 Preparation and Installation

The simulator App is an Android App that makes use of several Google server functionalities, e.g. map visualization and auto completion for destination input.

To deploy this App from an IDE one has to previously set two developer Google API keys to the project configuration file. Also one key set can be used for multiple devices it is recommendable to get extra key sets for the own development project. Because the allowed requests per day to Google are limited which are required in the simulator App. In the following all steps to generate these keys are explained.

The first step is to log into the Google API Console with a valid Google account (create a new Google account if none is given).

Go to: https://code.google.com/apis/console/

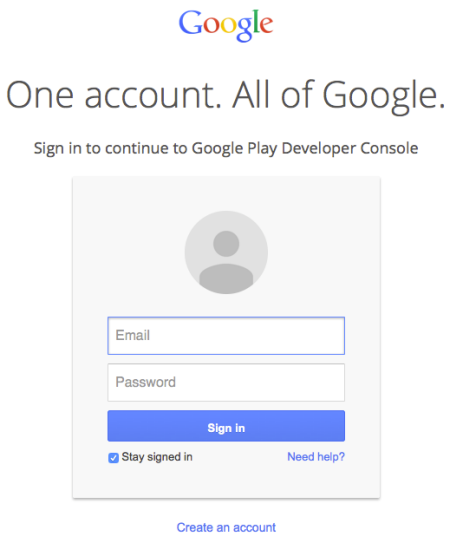


Figure 14: Login Screen of the Google API Console

Login to the Google API Console and then click on “create a project” as depicted in Figure 15.

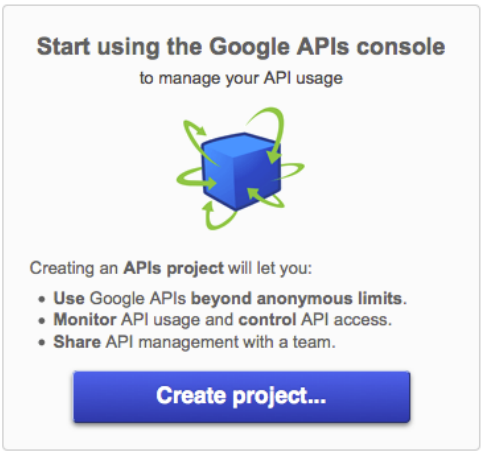


Figure 15: Create New Project

Now *services* has to be selected (if not selected) on the left top menu as depicted in Figure 16.

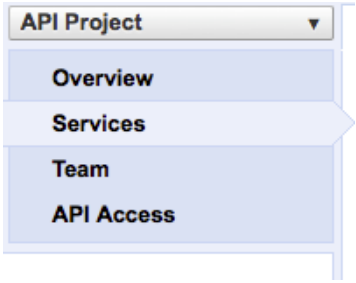


Figure 16: Selection Tab on the Top Left Side of the Google API Console

Now the list of all available Google services is displayed. In the next step the status switch of the Google Maps Android API v2 and the Places API have to be activated. One has to scroll down to *Google Maps Android API v2*, c.f., Figure 17. Here the status switch (middle column) has to be set to “on” for the service.

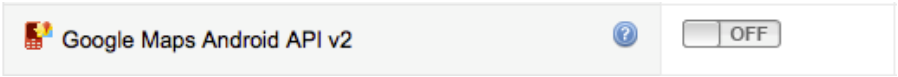


Figure 17: Google Maps Android API v2

After activating the switch button (by clicking on it) the Google Maps API Terms of Service have to be accepted (one has to activate the agree to terms check-box) as depicted in Figure 18.

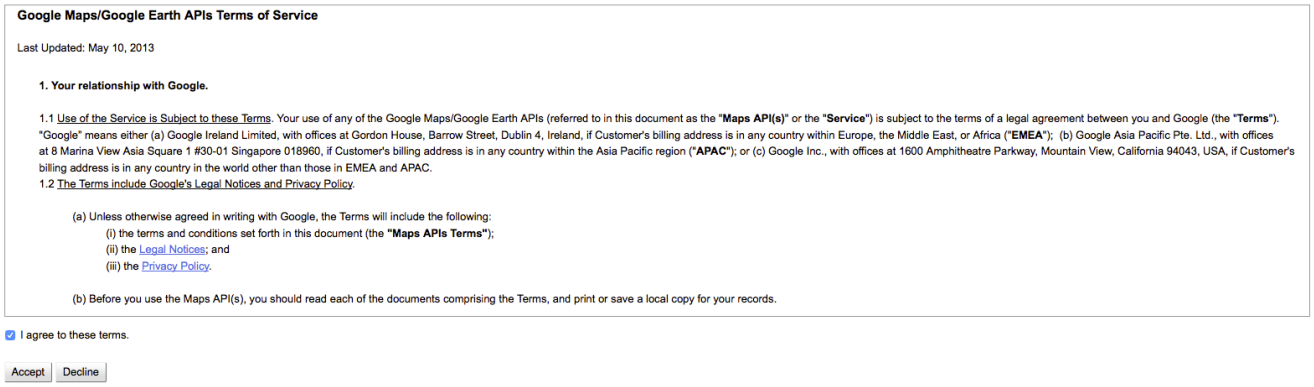


Figure 18: Google Maps API Terms of Service

Now this service should be activated like this:

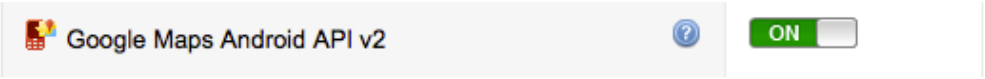


Figure 19: Google Maps Android API v2 with Activated Switch

In the next step the status switch of the Places API has to be activated. Scroll down to *Places API*, see Figure 20, and set the status switch (middle column) to on for the service.

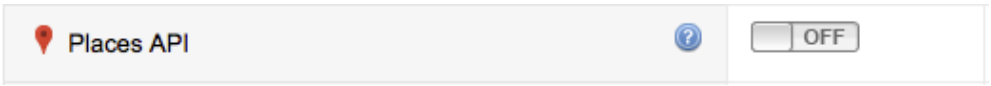


Figure 20: Places API

After activating the switch button the Google Places API Service has to be registered as depicted in Figure 21. This has to be committed by clicking the “Submit” button on the lower left side.

Figure 21: Places API Registration

Now this service should be activated as depicted in Figure 22.

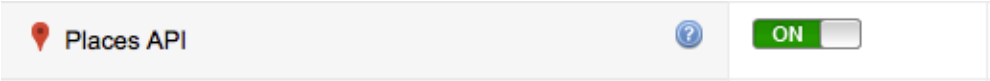


Figure 22: Google Places API with Activated Switch

Then one has to select the API ACCESS section on the top left menu as depicted in Figure 23.

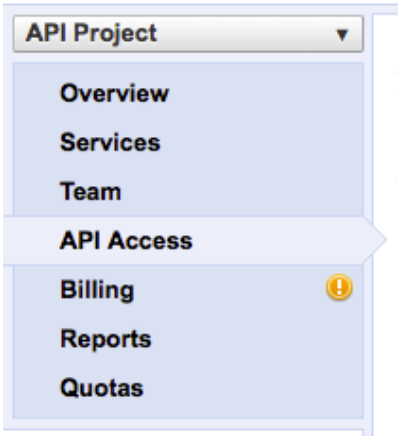


Figure 23: API Access

Here one can see a menu as depicted in Figure 24.

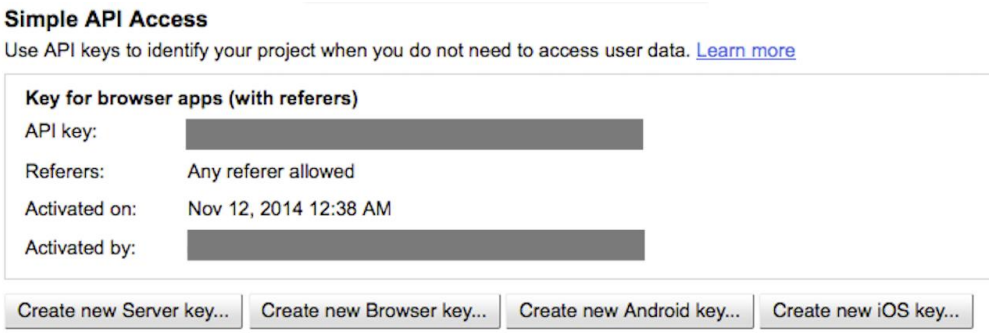


Figure 24: The Simple API Access Menu

Now one has to click on *Create new Android Key* and then finalize the step by clicking the *create* button in the new upcoming window (one does not need to enter anything) as depicted in Figure 25.

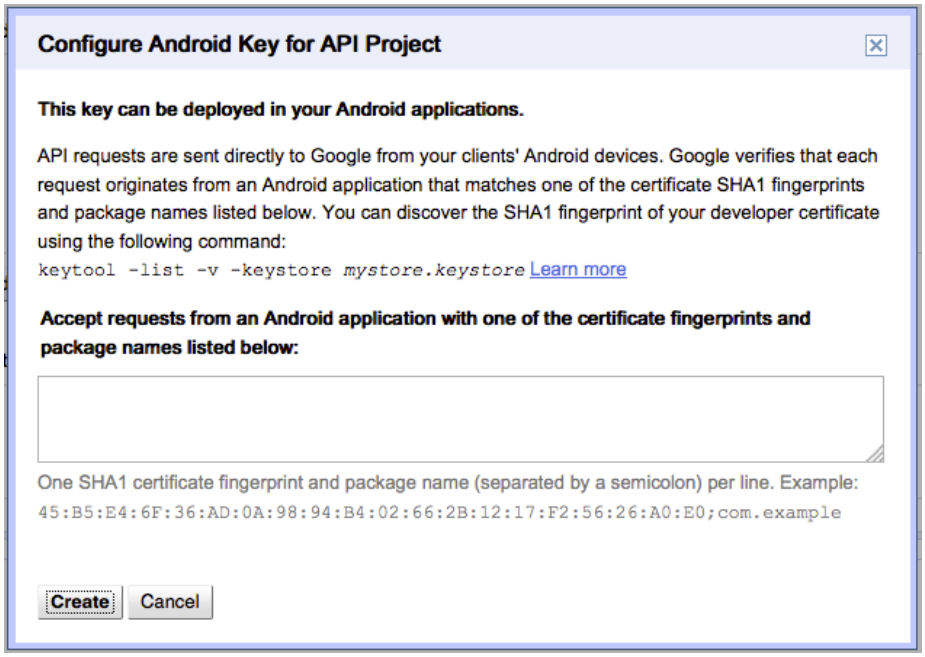


Figure 25: Configuration for the Android Key

Now one should see the overview window with two keys as depicted in Figure 26.

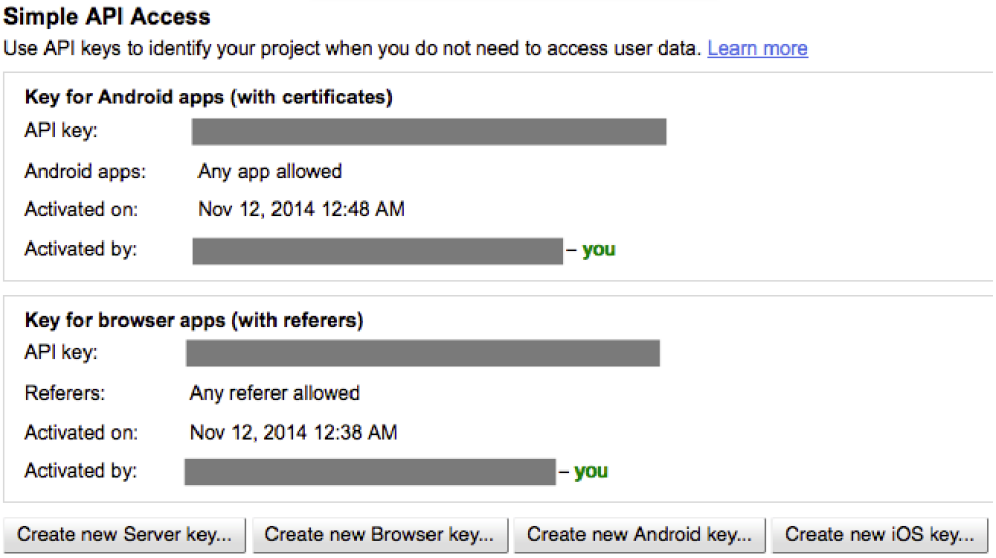


Figure 26: API Keys Overview

In the next step one has to click on *Create new Server key,* the most left button in the API key overview as depicted in Figure 26.

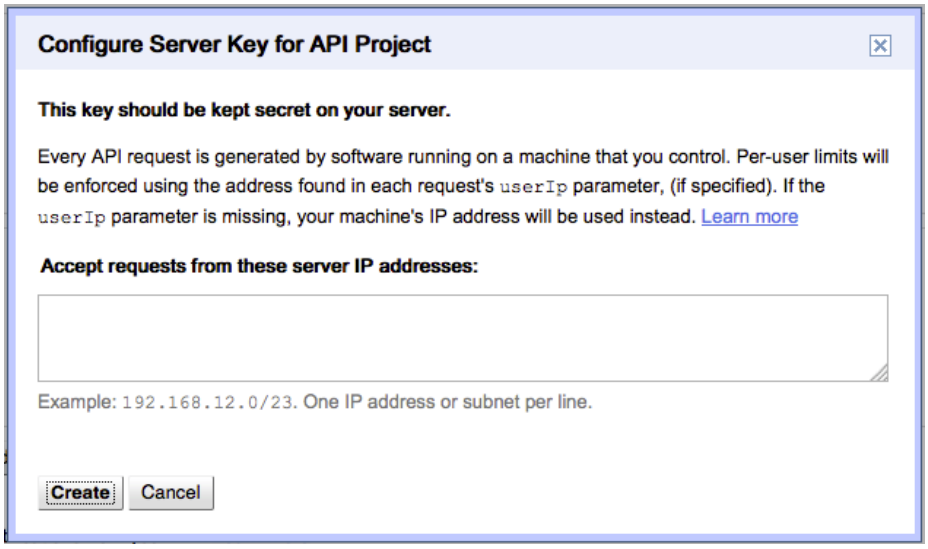


Figure 27: API Key Configuration

Now one has to click on *Create new Android Key* and then click the create button in the new window (don't need to enter anything) as depicted in Figure 27.

Now one can see three keys in the overview window as depicted in Figure 28.

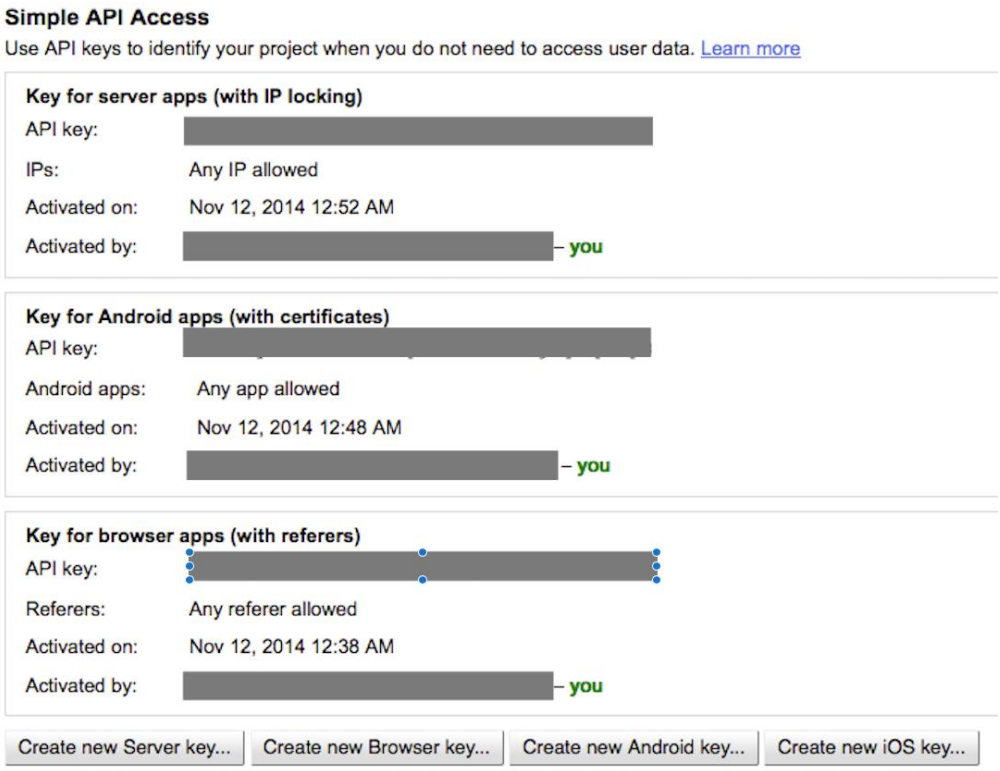


Figure 28: API Keys Overview

Now one has to copy the Android App key and the server apps key to the simulator manifest file. This file can be found in the source code at sensorAbstractionPMA/SensorAbstractionSimulator/AndroidManifest.xml.

One should use the Android App key for: com.google.android.maps.v2.API_KEY and use the server apps key for: android.places.key as depicted in Figure 29.

```
<meta-data android:name="com.google.android.maps.v2.API_KEY" android:value="YOUR_NEW_GENERATED_KEY"/>
<meta-data android:name="android.places.key" android:value="YOUR_NEW_GENERATED_KEY"></meta-data>
```

Figure 29: API Key Settings in AndroidManifest.xml file

7.1.1 Additional Requirements

Possibly it is required to increase the memory of the Java IDE (in the recommended case Eclipse) before one can start to compile the simulator app. To do this one has to navigate in a file browser to the eclipse program folder and edit the eclipse.ini file with a text editor as can be seen in Listing 28.

Listing 28: Required Changes for Increasing Memory Availability in Eclipse

```
…
--launcher.XXMaxPermSize
1024m
…
-Xms512m
-Xmx1024m
…
```

After editing the file must be saved and the Eclipse IDE has to be restarted.

Before deploying the app, one has to check the project properties *Android* tab, google-play-services_lib must have a green check in the Library section as depicted in Figure 30.

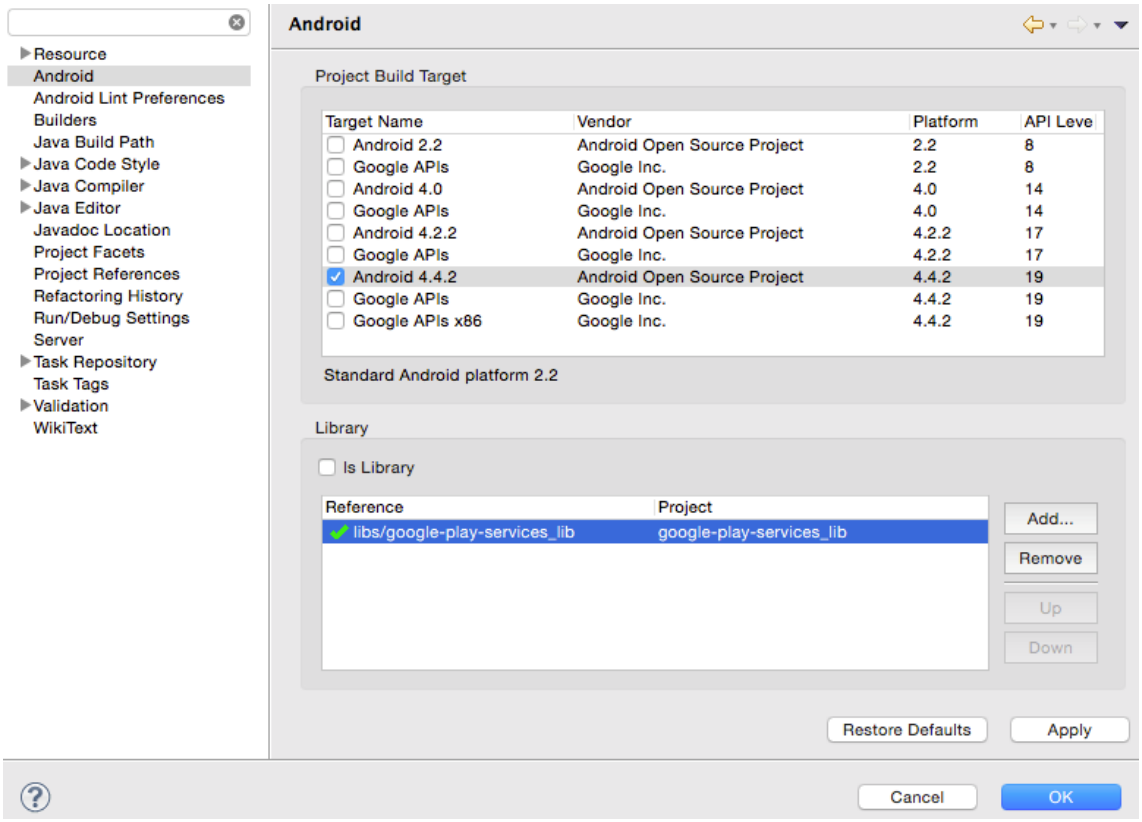


Figure 30: Android Project Properties

If this is missing one has to use the Android SDK manager to install the Google Play Services.

One can install the simulator app by the use of Java IDE Eclipse and running the project as Android Application.

7.2 Further Information about the Sensor Simulator

The Sensor Abstraction and Interoperability Interfaces automatically provide simulated sensor data when the simulator App is started and the simulation is executed. However, the device information contains all supported sensors while the simulation is running. If the simulation is stopped the device information contains only the available sensors. Furthermore the virtual sensors of the simulator have different IDs than the real sensors.

7.2.1 Behavior of the Sensor Simulator

The behaviour of the Sensor Abstraction and Interoperability Interfaces with regard to the PMA-based sensor simulator is depicted in Figure 31. The return value is automatically fetched from the simulation engine while the simulation is running.

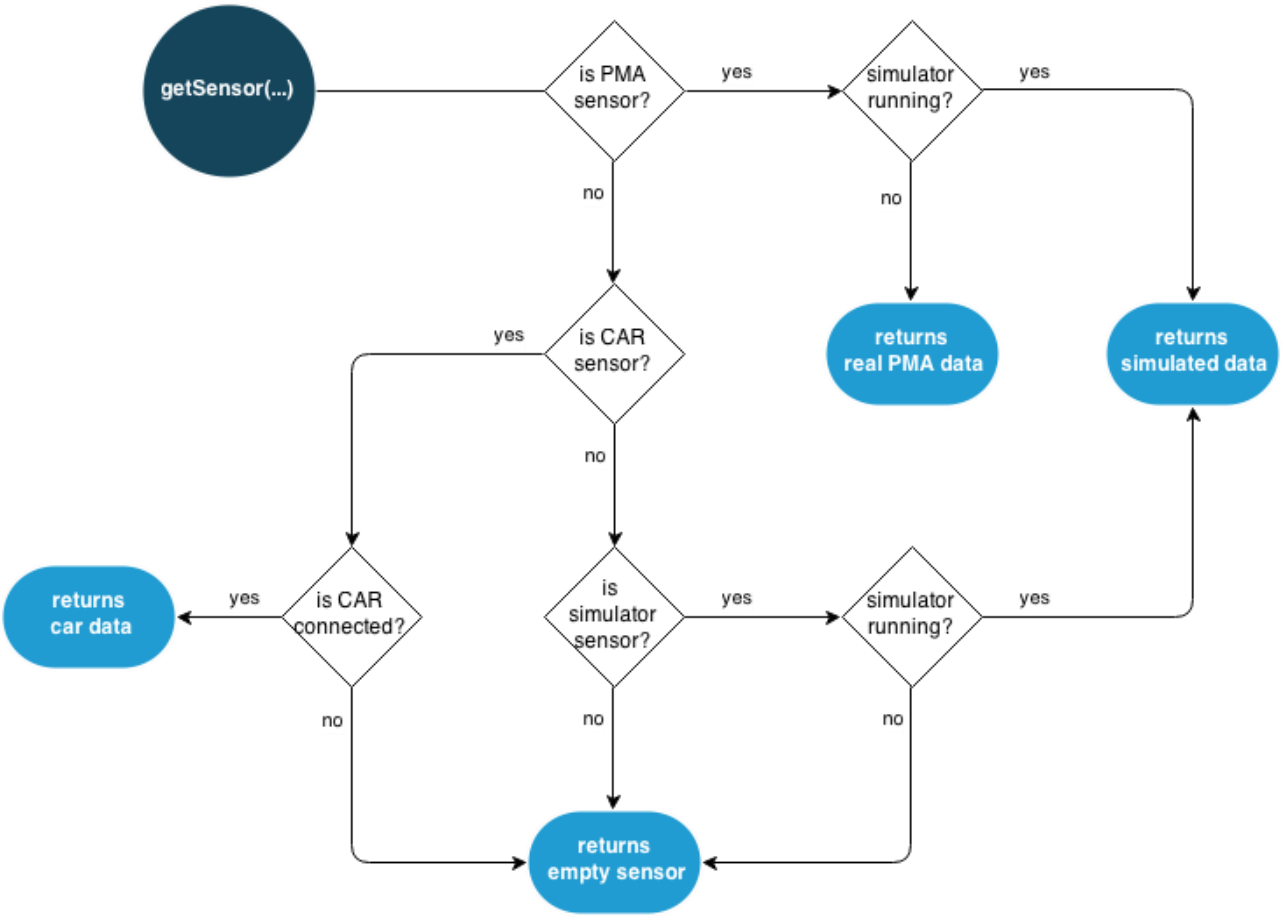


Figure 31: Behavior of the Sensor Abstraction and Interoperability Interfaces with regard to the PMA-based sensor simulator

8 Summary

The deliverable at hand presents and describes the scope of the final prototype of the Sensor Abstraction and Interoperability Interfaces. This is a core component within SIMPLI-CITY to integrate external sensor data sources. As these sensors and the corresponding sensor data is usually rather heterogeneous, for example ranging from different parking lot data to data from smartphones or vehicles, etc., a homogeneous access method has to be provided for this data to be easy to use. This component provides the seamless integration of heterogeneous sensor sources and sensor readings within SIMPLI-CITY, e.g., by providing corresponding wrappers. The Sensor Abstraction and Interoperability Interfaces also provide access to sensors, connected to the PMA, and user-related data accessible on the PMA, e.g. contacts and calendar data to other SIMPLI-CITY components. These are the SIMPLI-CITY server components as well as SIMPLI-CITY Apps, running in the Application Runtime Environment (ARE). This functionality is supported through the PMA-based Sensor Abstraction, which is in fact a subcomponent of the Sensor Abstraction and Interoperability Interfaces. This component is executed as a background service on the PMA and is responsible for allowing the access from the server side of the Sensor Abstraction and Interoperability Interfaces component to the local data sources on the PMA. This background service is also included in the final prototype of the ARE.

Within this prototype, an final prototype version of the Sensor Abstraction and Interoperability Interfaces component was implemented. The component consists of two main parts, the server side Sensor Abstraction and Interoperability Interfaces that are executed as service bundle within the SRE and the PMA-based Sensor Abstraction component that is executed as Android background service on the PMA. Both provide Java interfaces for SIMPLI-CITY services and Apps. The server side component also provides RESTful interfaces for external services.

The focus of the final prototype was to integrate the PMA-based component into the Application Runtime Environment and to complete the features of the server based components of the Sensor Abstraction and Interoperability.

Apart from the prototype description, an analysis of the degree of fulfilment of the requirements that need to be covered by this component (as specified in the Requirements Analysis Report (D2.3)) was presented.

In addition, all required steps to install, deploy, and execute the different components have been described.