simpli·city

The Road User Information System Of The Future

WP5 – Mobility Services Framework

D5.3.3: Service Runtime Environment Prototype II

Deliverable Lead: TUV

Contributing Partners: TIE

Delivery Date: 03/2015

Dissemination Level: Public

Version 1.0.3

This deliverable describes the work carried out during the development of the final prototype of the Service Runtime Environment of the SIMPLI-CITY Mobility Services Framework. It specifies how to install and execute the different implemented subcomponents.

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	Philipp Hoenisch, TUV
Internal Reviewer 1	Xavier Cases Camats, WORLD
Internal Reviewer 2	Kristof Kipp, ASC
Type	Deliverable
Work Package	WP5: Mobility Services Framework
ID	D5.3.3: Service Runtime Environment Prototype II
Due Date	31.03.2015
Delivery Date	30.03.2015
Status	For Approval

Document History	
Contributions	V0.0.1, Philipp Hoenisch, TUV, 02.02.2015, Added document structure. V0.0.2, Rostyslav Zabolotnyi, TUV, 20.02.2015, Added monitoring and reporting description. V0.0.3, Philipp Hoenisch, TUV, 20.02.2015, Updated monitoring descriptions. V0.0.4, Rostyslav Zabolotnyi, TUV, 26.02.2015, Fixed SRM logo, Updated monitoring descriptions. V0.0.4.1, Vadim Petrenko, TIE, 09.03.2015, Minor Updates. V0.0.5, Philipp Hoenisch, TUV, 10.03.2015, Update and finalize. V1.0.0, Philipp Hoenisch, TUV, 11.03.2015, Finalized for internal review. V1.0.2, Philipp Hoenisch, TUV, 11.03.2015, Update according internal review V1.0.3, Philipp Hoenisch, TUV, 25.03.2015, Finalized
Final Version	March 30st, 2014

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 – Retie Mobilità, Italy

Executive Summary

This deliverable describes the work which was carried out during the development of the final prototype of the Service Runtime Environment. For this, this document starts with introducing the Service Runtime Environment 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.

Within this deliverable we specially focus on the REST Proxy, the Monitoring and Reporting component.

The REST Proxy was created with the purpose of providing an extra layer between the caller of a service and the service itself. It can also be described as the single entry point to invoke available backend services via RESTful calls. The REST Proxy invokes the Monitoring component on each service call in order to ensure given SLAs (Service Level Agreements). Furthermore, the response time is measured and stored including the point of time and the caller ID. This information is used for auditing and the generation of statistics and reports. These reports may be displayed in the Service Marketplaces, by service developers or other backend services. In the case that a SLA is violated or a service invocation failed, the service developer will be notified via mail. If this happens several times, the system administrator will be notified. The REST Proxy contains logic to reinvoke the method in question in case of the original invocation failed, thus increasing robustness of the solution. Security is enforced on the service method level, and service requesters can be optionally required to authenticate themselves against user information stored in the Cloud Infrastructure before any service access will be granted. Caller access rights are checked centrally and transparently by the REST Proxy. These authorisation rules can be defined in the service interfaces with per method granularity.

The last sections of this document describe how potential users (i.e. service developers and administrators) can prepare, install and execute the different parts of the Service Runtime Environment component. For this, a step-by-step guideline to install and make use of the prototype is provided.

Notably, although the Service Registry is mentioned below, we will not describe its interfaces or usage. The Service Registry had its own final deliverable i.e., D5.3.2 (Service Registry Prototype).

This deliverable D5.3.3 is the final prototype of the Service Runtime Environment.

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 Service Runtime Environment 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 SIMPLI-CITY Service Runtime Environment and exploit its functionalities. For this, the scope and requirements of the final Service Runtime Environment prototype, the requirements and preparations for administrators and developers, and an installation and usage guide are provided.

The final Service Runtime Environment prototype is the outcome of the discussions and implementation work done in project months 18 to 30. It provides the final implementation of the functionalities of the Service Runtime Environment as provided with SIMPLI-CITY deliverables D3.2.1 (Functional Specification), and D3.2.2 (Technical Specification). This deliverable D5.3.3 is the final prototype of the Service Runtime Environment

1.3 Document Status and Target Audience

This document is listed in the Description of Work (DoW) as “Public”. It provides the means to exploit the functionalities of the SIMPLI-CITY Service Runtime Environment as defined in deliverable D3.2.2 (Technical Specification).

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

1.4 Abbreviations and Glossary

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

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

1.5 Document Structure

This deliverable is broken down into the following sections:

Section 1 provides an introduction 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 and relationship of the prototype, showing where the Service Runtime Environment fits into the overall SIMPLI-CITY Mobility Services Framework. Also, the outcome of the implementation phase is presented. 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 to make use of the Service Runtime Environment final prototype.

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 provides a summary of the document.

2 Prototype Scope and Requirements Coverage

2.1 Service Runtime Environment – General Information

The Service Runtime Environment is the core framework for hosting and invoking internal backend services, i.e. services running within the SIMPLI-CITY Mobility Service Framework. In addition it offers functionalities for the invocation of data services and external services, i.e. services which are not deployed in the SIMPLI-CITY Mobility Services Framework, but nevertheless provide functionalities needed by mobile apps running in the Personal Mobility Assistant (PMA).

For this, the Service Runtime Environment provides the functionalities to execute and bind backend services. Furthermore, it controls the monitoring of services. While backend services are usually invoked by apps running in the PMA through the Application Runtime Environment, in some use cases, backend services may also be invoked by other backend services. For this purpose, the Service Registry and the REST Proxy provide Java as well as RESTful interfaces.

Figure 1 shows the location of the Service Runtime Environment in the SIMPLI-CITY Global Architecture. For the full Global Architecture, please refer to deliverable D3.1.

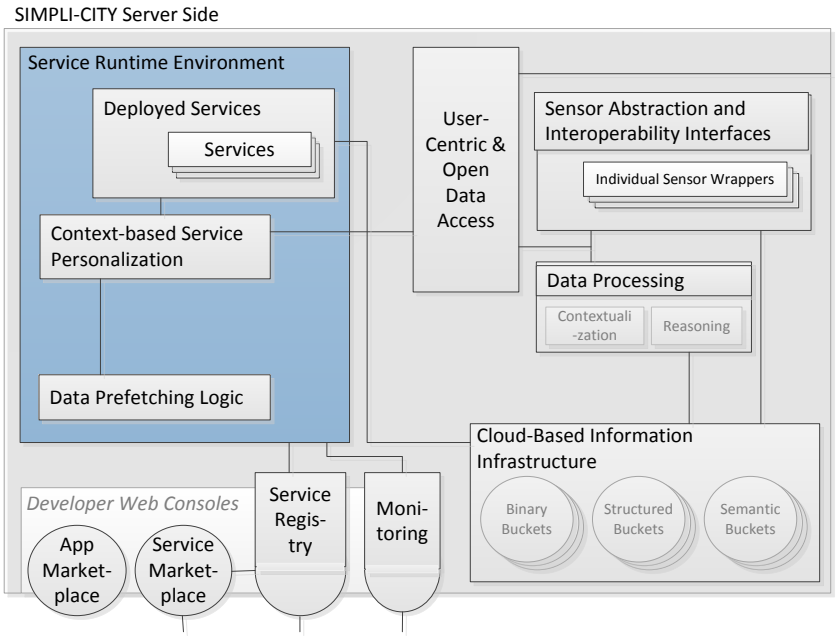

Figure 1: Location of the Service Runtime Environment in the SIMPLI-CITY Global Architecture

2.2 Scope of the Final Prototype

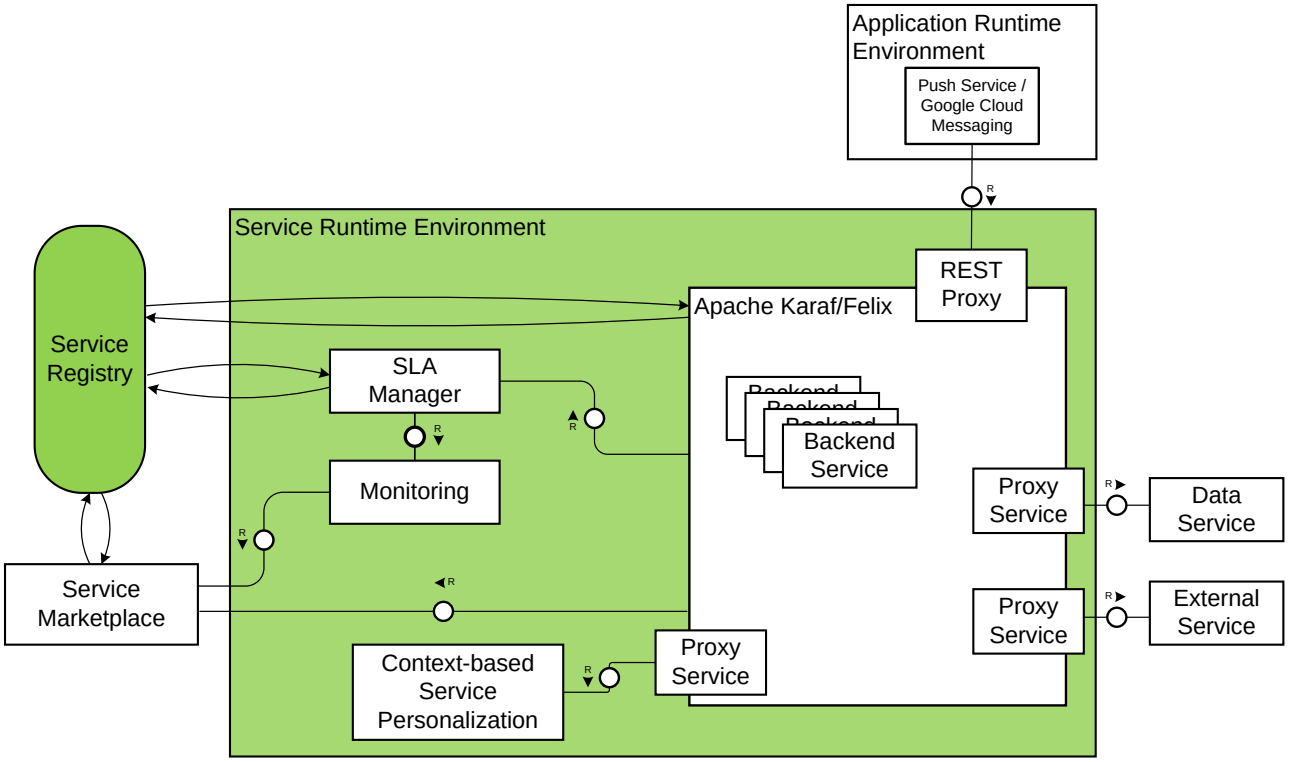

Figure 2: Scope of the Final Prototype of the SIMPLI-CITY Service Runtime Environment

Figure 2 depicts the status of the development of the Service Runtime Environment. For that, the subcomponents that are covered within the final prototype are highlighted. Note that this figure shows the Service Runtime Environment and also other parts of the SIMPLI-CITY Global Architecture, which are not developed within this component (i.e., the Context-based Service Personalization, External and Data Services, the Service Marketplace and the PMA). Beside of that, for the sake of convenience and visibility, the subcomponents SLA Manager, Monitoring, Apache Karaf/Felix and the REST Proxy are not coloured but also fully implemented.

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

- Green: Fully implemented
- Orange: Partially implemented
- White: No implementation, or not part of this deliverable

In the following subsections, the scope and status of the single subcomponents (as depicted in Figure 2) will be discussed in more detail. 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 Apache Karaf/Felix

Apache Karaf is the core of the Service Runtime Environment, which makes use of the OSGi container Apache Felix. It provides the means of deploying services and allows other services to execute them. As this is a crucial functionality for almost every backend service, it is fully covered within the final prototype.

2.2.2 Monitoring

The purpose of the Monitoring component is to ensure Service Level Objectives defined within the SLAs. Doing so, it enables the Service Runtime Environment to provide feedback to end users as well as service developers about the status of the services. This component was fully covered within the final prototype. Further, a reporting service was implemented. It can be used to generate reports for a specific service in a defined time interval in order to notify software developers or system administrators.

2.2.3 Service Registry

The Service Registry stores software service artifacts along with their description in order to be accessible by other services or apps running on the PMA. Since the latter ones are not running inside the Service Runtime Environment, the Service Registry provides RESTful interfaces next to the Java interfaces for internal services.

These interfaces provide the means of querying the Service Registry for services according to a particular filter. Further, the Service Registry also offers Java and RESTful interfaces to create, update or delete services including their description to software developers. This allows them to deploy their services in the Service Runtime Environment, to update the service description or the whole service, or to delete everything if not needed anymore. Within the final prototype, means to create, to update and to delete services was implemented. In addition to that, the functionality for retrieving service descriptions by a filter was implemented.

Beside of this, the Service Registry also provides the means of creating, updating or deleting licenses for a particular service, creating and updating a service watch and the ability to retrieve historical service watch data.

Notably, although the Service Registry is mentioned here, its interfaces are not part of this deliverable as the Service Registry had its own final deliverable, i.e., D5.3.2 (Service Registry Prototype).

2.2.4 REST Proxy

The REST Proxy provides external users with a single point entry for invoking services available in the Service Registry. There is only one *invokeService()* method which calls the desired service based on the given parameter set that includes the serviceID, the invocation method name, additional parameters and the caller credentials (username and password).

Within the final prototype, the secure access (caller authentication and authorisation) was implemented in addition to monitoring and SLA (specified per service in the service descriptor). For achieving the monitoring functionality, the REST Proxy invokes the SLA Manager on every service invocation.

The REST Proxy provides the logic to reinvoke a method in in case of an invocation failure, thus increasing robustness of the solution. Security is enforced on the service method level, and the caller of the service can be optionally required to authenticate themselves against user information stored in the Cloud Infrastructure before any service access will be granted. Caller access rights are checked centrally and transparently by the REST Proxy. These authorisation rules can be defined in the service interfaces with per method granularity.

2.2.5 SLA Manager

The SLA Manager is responsible for constant monitoring and verification of the overall condition of the OSGi container. For that, it verifies that every service invocation abides to its SLA. Further, the SLA Manager monitors the behaviour of the Service Runtime Environment asynchronously, i.e., it gets invoked by the REST Proxy on each service call. Within the final prototype this functionality was fully covered.

2.3 Covered Requirements

This section describes the degree of fulfilment of the requirements to be covered by Service Runtime Environment and specified in the Requirements Analysis document (D2.3) and the Functional Specification (D3.2.1).

Table 1: Requirements Related to Service Runtime Environment and their Degree of Fulfilment

Requirement	Degree of Fulfilment	Comment
Must Have Requirements		
U27: Proactive behaviour U195: Proactive user notifications U196: Prioritization of notifications	100%	This requirement is not directly covered within this deliverable but the needed functionality is provided. Requirements U27, U195 and U196 itself are part of D5.2.2, i.e., the final prototype of the Context-based Service Personalization component.
U106: Access to cloud services	100%	This functionality is crucially for the Service Runtime Environment. Hence, it was fully implemented.
U139: Provision of service statistics U182: Provision of statistics, e.g., usage, traffic	100%	This functionality is fully covered by the final prototype. It is possible to retrieve a statistic over a specific time period.
U192: Service deployment	100%	The deployment of backend services is possible, further it is also possible to undeploy them again, and retrieve service descriptions through the Service Registry.
U193: Exchange of information from apps to server U194: Exchange of information from server to apps	100%	SIMPLI-CITY services can be invoked in the RESTful way through a single entry point, i.e., through the REST Proxy.

U207: Support suggestions for road/trip optimization if conditions change U209: Provision of real-time information about the current route U212: Notification to end users about the proximity of Points of Interest	100%	These requirements are not part of this deliverable but the needed functionality has been provided. The actual implementations of these services have been provided within D5.2.2, i.e., the final prototype of the Context-based Service Personalization component.
U151: Service versioning	100%	This functionality was not fully covered in D5.3.2 the final prototype of the Service Registry but has been finished within the final prototype of the Service Runtime Environment. It is now possible to have different version for a particular service.
Should Have Requirements		
U86: Transparency U87: Confidentiality U88: Data encryption U89: Certification	85%	These functionalities have been partially covered within the REST Proxy, i.e., the REST Proxy is an additional layer which takes care of different security aspects as authentication, i.e., via username and password, authorization, some services may be restricted to be used by special users. The result is a transparent layer between service caller and backend services. Data encryption for data transfer between Service Runtime Environment and Application Runtime Environment has not been covered. However, this functionality is foreseen and prepared for a final implementation in the context of the SIMPLI-CITY use case implementation. Hence, if security aspects emerge during the final use case implementation, a secure transport protocol such as HTTPS will be applied

U90: Availability U91: Integrity U92: Secure access to system U93: Third party access to the system	100%	A high level of availability has been achieved through a stable system and well used exception handling. In case of errors the system will remain functional and a responsible person, e.g., a developer or system administrator will receive a notification. Integrity and secure access has been achieved by applying authentication mechanism to the system via the REST Proxy. This ensures that services can only be accessed by authorized users and each request is logged. Further, third parties can request access to the system by requesting username and password. With these, third parties can provide their services within the Service Runtime Environment or make use of existing once.
U103: Fault tolerance	100%	This functionality is covered within the final prototype. However, a 100% fault tolerant system can never be guaranteed if 3rd party services are used.
U104: Stability	100%	This functionality is covered within the final prototype. However, a 100% stable system can never be guaranteed if 3rd party services are used.
U125: Open interfaces U126: Openness of the system U127: Extensibility	100%	The interfaces of the final prototype are well defined and documented and therefore the SIMPLI-CITY Service Runtime Environment is easily extendible.
U188: Composition of services	0%	This requirement was not covered within the final prototype, since the composition of services happens in apps, not on the level of services.

Could Have Requirements		
U124: Scalability of the service platform	50%	This functionality as such was not covered within the final prototype, however the overall Service Runtime Environment has been designed in a way that allows provisioning of scalability features in the future. REST Proxy being the single entry point for all method calls can be retrofitted to serve as a load balancer and the built-in monitoring and SLA watching modules can spawn new Service Runtime Environment instances should load increase and response times degrade.

3 Preparations

This section provides information about what potential users (both administrators and software developers) need to prepare in order to use the functionalities of the delivered prototype.

The server side part of the Service Runtime Environment will be executed 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 Service Runtime Environment, Oracle's Java SE Development Kit 7 and Apache Maven 3.0.5 need to be installed on the system. In addition, port 8080 has to be available for the Service Runtime Environment as this port will be used for the Service Registry. If this port is not available, the Service Runtime Environment will not work properly. In addition, it is required that the environment variable JAVA_HOME is defined as per Oracle guidelines.[1]

Further, the Service Runtime Environment requires a valid internet connection; otherwise, some of the services may fail.

The final prototype was tested on Linux and Windows:

- The Linux machine was running Ubuntu 12.04, Java(TM) SE Runtime Environment (build 1.7.0_51-b13) and Apache Maven 3.0.5.
- The Windows machine was running Windows 8.1, Java(TM) SE Runtime Environment (build 1.7.0_55-b13) and Apache Maven 3.0.5.

[1] http://docs.oracle.com/cd/E19182-01/820-7851/inst_cli_jdk_javahome_t/index.html

4 Installation (Deployment)

While Section 3 provides information about the technical preparation *before* the installation of the software, this section provides guidelines on how to install and deploy the final prototype of the Service Runtime Environment.

A bash script is provided for Windows and Ubuntu.

1. Unpack the provided archive file and change into the resulting folder, where the following directory structure should appear:
    - services/
    - eu.simpli-city.sre.assembly-1.0.0-SNAPSHOT.tar.gz
    - eu.simpli-city.sre.assembly-1.0.0-SNAPSHOT.zip
    - start.cmd
    - start.sh

On a Windows machine:

2. On Windows, double-click “start.cmd” and wait for the Service Runtime Environment to start.

On a Linux machine:

1. If not already done, start a terminal and navigate into the resulting folder.
2. You can make the file “start.sh” executable by typing
    - chmod a+x start.sh
3. Type into the terminal: “./start.sh”. This will start the Service Runtime Environment and automatically deploy some example services. If the output is similar to Figure 3, the Service Runtime Environment should be up and running.

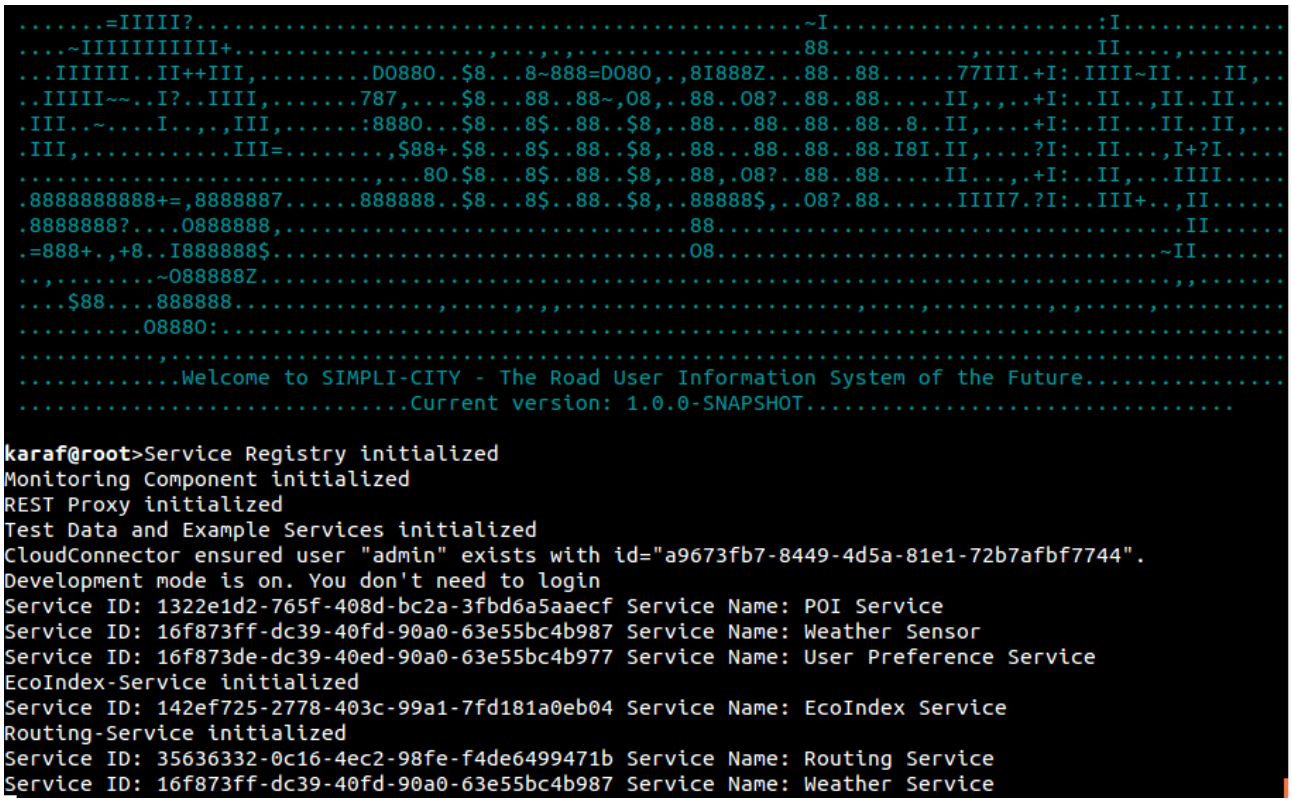

Figure 3: Service Runtime Environment Start-Up Output

If no error was printed, the Service Runtime Environment has been started successfully. In addition, the log shows that the Service Registry was successfully started and 4 example services and 1 example sensor are pre-deployed:

- A Point of Interest (POI) Service
- A User Preference Service
- An EcoIndex Service
- A Routing Service
- A Weather Sensor

4.1 Service Execution Setup

The service execution setup includes the following set of steps:

1. The Service Runtime Environment has to be started as per instructions provided above (refer to Figure 3 for the Service Runtime Environment start up output).
2. Once the Service Runtime Environment is started, the three fundamental modules should be started and correctly initialized:
    a. Service Registry (message “Service Registry initialized” should appear in Karaf output)
    b. REST Proxy (message “REST Proxy initialized”)
    c. Monitoring Component (message “Monitoring Component initialized”)
3. Being started up and initialized properly these component provide the basis for execution of backend services.

An example of a successfully started Karaf with correct output is given in the Figure 3.

5 Execution and Usage of the Software

This section describes how to use the different subcomponents of this final prototype.

5.1 Server Side (System Administrators)

Within this section it is shown how the final prototype of the Service Runtime Environment can be tested and/or used.

5.2 Service Runtime Environment Service Developer Usage Guide

The following subsections shows how the subcomponents covered within the final prototype can be used by service developers.

5.2.1 Monitoring

Service Monitoring has been fully implemented in the final prototype. It provides two methods that allow starting and stopping monitoring of a particular service invocation. Each “start monitoring” call accepts service and caller’s IDs and stores this information within a database, e.g., the point in time when this invocation has been started including the caller’s ID and service ID (see Listing 1). The “Stop monitoring” method uses this information to calculate the service response time and other Quality of Service (QoS) parameters. This information, including the service outcome, e.g., if the invocation was successful, is stored in a database (see Listing 2).

Service Monitoring is used by the REST Proxy to transparently monitor service executions and verify SLAs. Whenever an SLA of a particular service is violated, the service developer gets notified with an e-mail that describes the violation details. However, when a significant amount of services are constantly violating their SLAs, a global system malfunctioning might be detected and the system administrator is notified instead.

SLA violation notification parameters can be configured from the Service Runtime Environmnet using console commands. Command “reporting:config” allows viewing the current configuration as well as changing notification thresholds and delays. Command “reporting:mail” allows changing the configuration of the e-mail account used to send SLA violation notifications. All changes are applied immediately and reset on a Service Runtime Environment restart.

Listing 1: Method Signature – Start Service Monitoring Invocation

```
/**
* Starts monitoring a particular service invocation.
* @param serviceID, defines the ID of the service instance to be invoked.
* @param requesterID, defines the ID of the service requester.
*
* @throws ServiceNotFoundException in case the service ID is invalid.
* @throws MonitoringException in case an error occurred during monitoring
* initiation.
*/
public void startMonitoring(UUID serviceID, UUID requesterID) throws
ServiceNotFoundException, MonitoringException;

…
monitoringManager.startMonitoring(serviceID, userID);
```

Listing 2: Method Signature – Stop Service Monitoring Invocation

```
/**
* Stops monitoring a service and stores the results in the Monitoring Data Bucket
* @param serviceID, defines the ID of the service instance to be invoked.
* @param requesterID, defines the ID of the service requester.
* @param successful, indicates if the service invocation was successful.
*
* @return response time in milliseconds.
*
* @throws ServiceNotFoundException in case the service ID is invalid.
* @throws MonitoringException in case start monitoring was not called for this
particular service invocation.
*/
public long stopMonitoring(UUID serviceID, UUID requesterID, boolean successful)
throws ServiceNotFoundException, MonitoringException;

…
long responseTime = monitoringManager.stopMonitoring(serviceID, userID, success);
```

5.2.2 Reporting

Service Reporting has been fully implemented in the final prototype. The main purpose of the Reporting Service is service usage tracking and accounting. Table 2 shows the final version of the Reporting RESTful interfaces which can be executed outside of the Service Runtime Environment. Figure 4 shows an example output in the browser. For the corresponding API references, please refer to deliverable D3.2.2. Service Reporting consists of two methods that allow generating current reports and retrieving service historical usage information.

Table 2: Reporting RESTful Interface

Functionality	Type	Command	Parameters	Description
Generate Report	HTTP-GET	http://localhost:8080/cxf/monitoring/generateReport?serviceID=<serviceID>&requesterID=<requesterID>&startTime=<startTime>&endTime=<endTime>	<serviceID>: UUID <requesterID>: UUID <startTime>: Long <endTime>: Long	Generates a report with monitoring data of a specified service and defined user between particular dates.
Get monitoring data	HTTP-GET	http://localhost:8080/cxf/monitoring/getMonitorData?serviceID=<serviceID>&requesterID=<requesterId>&startTime=<startTime>&endTime=<endTime>	<serviceID>: UUID <requesterID>: UUID <startTime>: Long <endTime>: Long	Retrieves historical monitoring data from the database.

Figure 4: Service Runtime Environment: Result of Get Monitoring Data via Browser

5.2.3 Service Invocation via the REST Proxy

A backend Service running inside the Service Runtime Environment can be invoked through the REST Proxy in two different ways:

- The REST way, using the HTTP protocol.
- Using direct Java method invocation, if initiated from inside the Service Runtime Environment by another backend service that wants to invoke its peer.

These two ways will be demonstrated in the following paragraphs.

5.2.3.1 Invoking REST Proxy via HTTP protocol (REST)

As described in the Technical Specification (D3.2.2, page 217), the REST Proxy can be called using the HTTP protocol, providing input parameters in the JSON format, in this case it returns a result of the invocation also in the JSON format.

In this document it is assumed that the developer has a tool "cURL"[2] installed on their computer for the purposes of testing. cURL is present in many Linux flavours, and on Windows can be installed as part of the Git distribution (http://git-scm.com/download/win). A Git Bash program can be found in the Git program group after installation. The cURL commands below can be typed directly into the command window of Git Bash.

In the following steps the EcoIndex backend service is used for illustrations. This service is bundled together with the Service Runtime Environment and is started automatically upon Service Runtime Environment start up. To verify the service has started, it is sufficient to find the line: "Service ID: 142ef725-2778-403c-99a1-7fd181a0eb04 Service Name: EcoIndex Service" in the output of Karaf (see Figure 3).

To invoke the service using cURL the following mandatory parameters have to be provided in the request in addition to the service ID:

- Method name to invoke, in this enample: "getEcoIndex"
- Method parameters (which happen to be none in this example, otherwise, normal JSON syntax should be used to enumerate the array of parameter values)
- Username and the password to identify the caller of the method

The username and the password will be matched against the values retrieved from the Cloud Infrastructure. The method "getEcoIndex" is annotated with a corresponding Java annotation in the code, requiring caller authentication prior invoking the method. The default user "admin" is used in the example, having the default password "password". Notably, these username and password are only for presentation and are not be used in the production mode.

Listing 3: Invocation of the REST Proxy Using cURL

```
curl -i -X POST -H 'Content-Type: application/json' -d '{"serviceID": "142ef7
25-2778-403c-99a1-7fd181a0eb04", "methodName": "getEcoIndex", "parameters": [],
 "username": "admin", "password": "password"}' http://localhost:8080/cxf/restPr
oxy/invokeService
```

Upon executing this command, cURL will return the result of service invocation:

[2] http://en.wikipedia.org/wiki/CURL

Listing 4: Result of Invocation of the REST Proxy Using cURL

```
HTTP/1.1 200 OK
Content-Type: application/json
Date: Mon, 09 Mar 2015 13:00:14 GMT
Content-Length: 60
Server: Jetty(8.1.14.v20131031)

{
  "result": {
    "ecoIndex": 46
  },
  "errorMsg": null
}
```

The resulting eco index is returned in JSON format.

5.2.3.2 Invoking REST Proxy from Java

The REST Proxy is a backend service on its own and can be called from any other backend service running inside the Service Runtime Environment. This allows monitoring and other extended features of the REST Proxy to accompany the service call. In Listing 5, an example is provided for invoking the same EcoIndex service, but from within some other service running inside the Service Runtime Environment. In the example, it is assumed that the REST Proxy reference is already obtained from Karaf using either ServiceRegistry mechanisms or using OSGi built-in functionality.

Listing 5: Invocation of the REST Proxy Using Java

```
RestProxyService restProxyService = ...;
try {
      Object result = restProxyService.invokeService(
                  UUID.fromString("142ef725-2778-403c-99a1-7fd181a0eb04"),
                  "getEcoIndex", new Object[] {}, "admin", "password");

} catch (ServiceNotFoundException e) {
      log.error("Service was not found in the ServiceRegistry.", e);

} catch (ServiceInvocationException e) {
      log.error("Service threw an exception during invocation.", e);

} catch (NotAuthorizedException e) {
      log.error("The user is not authorized to invoke the service.", e);
}
```

The invoked EcoIndex service returns a Java object as programmed by its developer (in this case a JsonObject).

6 Summary

6.1 Service Runtime Environment

The Service Runtime Environment delivered in this prototype is fully functional and covers all mentioned requirements from Table 1. It allows invoking arbitrary services by supplying the service ID, the wanted method name including corresponding parameters. In addition to that, the requester has to provide its identification credentials.

The Service Registry which covers the service management part (Create, Read, Update, Delete, …) has its own final prototype, i.e., D5.3.2 (Service Registry Prototype).

6.2 Monitoring Component

Within the final prototype of the Service Runtime Environment, the Monitoring component has been fully implemented. It is fluently integrated in the REST Proxy and will be called on every service invocation. It measures the service response time and stores the generated data, including the information about the service invocation's success, in a database. This data can then be used to generate reports for the Service Marketplaces, service developers or can be used in other backend services.

Further, the monitoring data is used to verify that no SLA has been violated. In the case of a violation, a notification is sent to the service developer. In addition, if several SLA violations happen, it is assumed that the Service Runtime Environment has a malfunction and a system administrator will be notified.

Due to technical limitations of the Java language we experienced some concurrency problems. This means, it is not possible to have multiple service invocations from one user to one service at the same time. While this should be considered in general, it is not crucial, i.e., the system will remain functional if a user sends multiple parallel requests to a particular service but the monitoring data might be corrupted.

6.3 REST Proxy

The REST Proxy has been fully implemented. It allows the invocation of backend services (i.e., services running inside the Service Runtime Environment) from the outside world, i.e., service running outside of the Service Runtime Environment. The REST Proxy is equipped with a basic authentication mechanism which requires a caller to provide username and password upon each call. This on the one hand, ensures a secure access to the system, and on the other hand, service invocations can be accounted to a requester. This is especially needed for auditing purposes, e.g., for creating invoices.

Currently, the REST Proxy in a single server mode. To provide scalability (which is listed under the "could have" section of functional requirements and might be addressed in the future), the REST Proxy could be extended with the functionality of a load balancer, redirecting the requests to a cluster of Service Runtime Environment servers, choosing the least loaded one. In addition, if running in a cloud, the REST Proxy can be programmed to start new server instances as soon as needed. This functionality is not implemented within the framework of this project. However, the REST Proxy was designed with this goal in mind. Should a real production environment require this logic this behaviour can be added smoothly at a later time.