D36.141

Service Development Kit (release 1)

Work package:	**3.6**
Version number:	**Version 0.3**
Dissemination level:	**PU**
Date:	**18/12/2013**

7th RTD Framework Programme
Directorate General for Communications Networks, Content & Technology
Cooperative Systems for energy efficient and sustainable mobility (FP7-ICT-2011-6.7)
Contract Type: Collaborative project
Grant agreement no.: 318485

Version Control

Version history			
Version	**Date**	**Main author**	**Summary of changes**
0.1	04/12/2013	Henning Mosebach	Initial structure and content
0.2	10/12/2013	Tobias Schlauch	Description of REST API
0.3	12/12/2013	Tobias Schlauch, Henning Mosebach	Consolidation of document
0.4	19/12/2013	Henning Mosebach	Integration of reviewers feedback
	Name		**Date**
Prepared	Henning Mosebach		13/12/2013
Reviewed	Igor Passchier		17/12/2013
Authorised	Dorota Boruc		23/12/2013
Circulation			
Recipient		**Date of submission**	
European Commission			
Project partners			

Authors

Henning Mosebach, Tobias Schlauch, Jörg Belz, Dirk Beckmann

Table of contents

Abbreviations and definitions

Abbreviation	Definition
API	Application Programming Interface
JSON	Java Script Object Notification
OS	Operating System
REST	Representational State Transfer
SDK	Service Development KIT
UI	User Interface
UML	Unified Modelling Language
USDL	Unified Service Description Language
URL	Uniform Resource Locator
WS	Web Service
WSDL	Web Service Description Language

Executive Summary

This document represents the basic components of the MOBiNET Service Development KIT. In general the SDK is not a central runtime component to be hosted by the MOBiCENTRE server but it is a bundle of developing tools that helps service developers doing their job as comfortable and reliable as possible.

This deliverable gives the list of tools and supporting documents for the service developers. The tools are classified towards their use in release 1, 2 and 3. It furthermore provides recommendations and best practices for RESTful Web Service design.

1. Introduction

1.1. Document Purpose and Scope

The MOBiNET platform consisting of MOBiCENTRE and MOBiAGENTis not only a central server hosting the applications. There is an additional need to enable a bridge between the MOBiNET platform and the user orientated service provider. Tools and guidelines are required to enable a developer that to develop services and apps that make use of the platform. These tools have to be identified, developed and adapted to make sure that the tools fit in the technical boundaries and functional requirements. The sets of tools will form the SDK, and enables service providers to develop and test their services in an aligned and comfortable way.

The activities regarding the service developer KIT have to be regarded in-line with the Service Developers Facilities (compare to D36.151). The Service developer facilities are providing working environments that help service providers to build, test and deploy services.

The purpose of this document is to give an overview over the Service Development KIT components and their boundaries.

1.2. Intended Audience

The targeted audience of this deliverable are the MOBiNET service developers. It is expected that the reader is already familiar with the MOBiNET platform (see D41.1) and has general technical expertise concerning software development based on a service platform. The future releases 2 and 3 will be more and more addressed to service developers outside of the MOBiNET project.

1.3. Boundaries and requirements of the SDK

Focussing on the need of a service provider a bundle of supporting tools is required to ease the development and deployment procedure of a new created service. While the MOBiNET components themselves and the hosting platform need a high level of expertise to be treated the service providers should have a package of means of support in order to develop their service in a comfortable and ”non-expert” way. In other words the complexity of a service is moved to the platform.

This package is developed within the SDK and will contain tutorials, tools that are required to use MOBiNET as service developing environment, a reference API and procedure description how to make use of the MOBiNET platform. The SDK also contains plug-ins for existing tools and example services that demonstrates the MOBINET functionality.

On a generic level these developer tools are

- An integrated developing environment containing:
    - A source code editor

    - Debugging opportunities for different programming languages e.g. Java) and for the REST API
    - A compiler adapted to the platform runtime environment
- A dummy service that serves as an example for service developers how to create their applications
- Guidelines and tutorials and/or references to existing accompanying documents
- Migration tools that allows to migrate existing services on the MOBiNET platform
- Definitions for data translation and semantics

During the study of this deliverable it is recommended to keep the division between the MOBiAGENT and the MOBiCENTRE in mind as it is represented in D31.2, fig.6.

2. Developer Tools

2.1. SDK tool candidates

To achieve the goal described in chapter 1 several tools concerning development support, testing environments and supporting software are required. The list of proposed components is given in Table 1. The objective of this deliverable is to select a realistic number of supporting tools for the first release that are mandatory to enable the developers creating their services.

Table 1: Tool candidates for supporting service developers and their classification (target releases)

Topic	**Short description**	**Example or source (link)**	**Target release**
Organisational support	Revision control system (SVN), Tracking procedure for developer requests	- Version control support (svn for Eclipse) - Jira-project dedicated to service providers	1
Tutorials for service providers	A complete description on an aligned level of details that enables service developers to develop a service	Containing: How to… - Define a service - Use the MOBiCENTRE and MOBiAGENT functionalities - Use the developing environment - Implement a service - Maintain a service	2,3
Tools for migrating existing services	A complete description on an aligned level of details that enables a service developer to migrate a service	Containing: How to… - Prepare the service before migration - Migrate the service	2,3
Selection of plug-ins for developing environment Eclipse	Eclipse plug-ins	- Editor - Debugger for REST services - Version control plug-ins - Jira plug-in	1,2,3

Supporting development with MOBiAGENT	Tools and documents for MOBiAGENT development	- Manuals - Tutorials - Examples on how to develop and deploy services in the MOBiAGENT	1,2,3
ZIP-Archive with complete bundle	Service developers bundle (for both LINUX and Windows)	Contains all relevant tools for a service developer	2,3
Editor for USDL service description	Editor for USDL service description	MOBiCENTRE services are expected to be described in USDL	1
Migration tools for USDL service description	Migration tools for USDL service description	Services to be integrated in MOBiCENTRE needs supported by changes of the service description	2,3
REST API recommendations	Specification of RESTful Web service API	Recommendations and best practices for RESTful Web Service design.	1
Example services including tutorials	Dummy service as example for service developers	“Greeting service” as guiding example	1

2.2. SDK tools for release 1

2.2.1. Organisational support

A software repository system will be set up and hosted by DLR in order to support the developers with a version control system to provide a central place towards developers.

For collecting and tracking issues specifically from the service developers, another project in our MOBiNET JIRA instance will be set up. Incoming request are collectively handled by the partners.

2.2.2. Tutorials for service providers

To be updated in release 2.

2.2.3. Tutorials for migrating existing services

To be updated in release 2.

2.2.4. Tools for migrating existing services

To be updated in release 2.

2.2.5. Selection of plug-ins for Eclipse environment

A recent Eclipse distribution bundled with plug-ins will be available via a Zip-archive. The target operating system for the developers is both Windows 7 and generic Linux.

As base distribution, we use the current Eclipse Kepler release (edition: Java EE Developers) (http://www.eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/keplersr1)

It contains the components explained in the following subchapters.

2.2.5.1. Plug-in for version control

For Subversion access, we provide the Subversive plug-in:

http://www.eclipse.org/subversive/

In addition, if developers want to use another version control systems, we also provide the EGit plug-in: http://www.eclipse.org/egit/

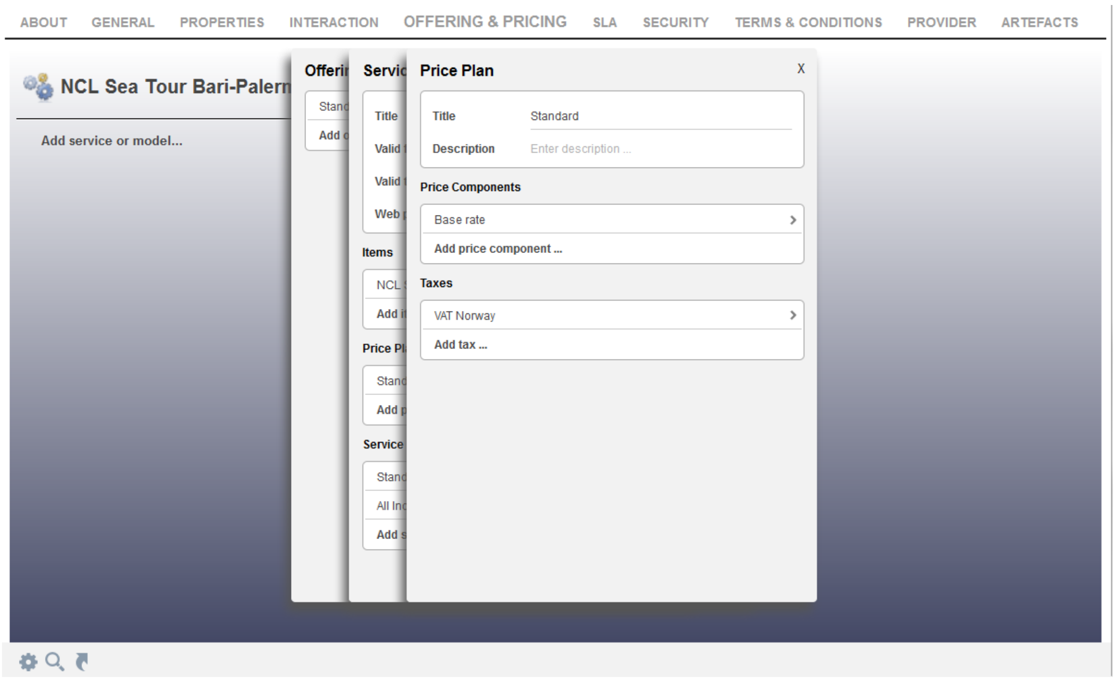
2.2.5.2. JIRA for issue tracker

The Jira tool that is already established in the working methodology (described in D31.2) will be set up for issues concerning the developing of services. The working procedure is described more detailed in MOBiNET D21.1 and can also be found using the following link:

https://confluence.atlassian.com/display/IDEPLUGIN/Working+with+JIRA+Issues+in+Eclipse

2.2.5.3. Editor for USDL service description

An editor for editing services descriptions in USDL will be added to the Eclipse distribution as HTML5 standalone app. An example of the editor is shown in Figure 1.



Title: Standard
Description: Enter description ...

- Figure 1: Example of USDL-Editor (here: http://www.linked-usdl.org/node/229)

2.2.5.4. Plug-ins for debugging REST services

The Eclipse HTTP Client will be used for debugging and testing the REST services. The HTTP Client (HTTP4e) is an Eclipse plug-in for making HTTP and RESTful calls. Build with user experience in mind, it simplifies the developer/QA job of testing Web Services, REST, JSON and HTTP. It is a useful tool for your daily job of HTTP header tampering and hacking. The features of the client are

- Making/Replaying an HTTP call directly from Eclipse IDE
- Visual Editors for HTTP headers, parameters and body
- Tabbed browsing (allowing replaying different RESTful, HTTP calls on separate tabs)
- History support (persisting your valuable REST calls)
- One-click HTTP code generation to Java, PHP, C#, Flex/ActionScript, Cocoa/Objective-C, Ruby, Python and Visual Basic
- One-click JMeter script generation
- Import and export HTTP4e replay script
- Export HTTP sessions as HTML report
- Import raw HTTP packets and Firefox's Live HTTP headers
- Aesthetic UI, Code assist, Headers auto-suggest, Syntax coloring
- “Raw”, “Pretty”, “Hex”, “Browser” and “JSON” views
- Proxy Configuration
- BASIC and DIGEST Authentication
- SSL/HTTPS support
- Unicode UTF8 support
- HTTP tampering
- Tab renaming
- Available on Windows, MacOS X, Linux, Solaris

More features and explanations can be found using: http://www.nextinterfaces.com/

2.2.5.5. Migration tools for USDL service description

To be updated in release 2.

2.2.6. Supporting Development with MOBiAGENT

For release 1 there are two ways under discussion how to develop a MOBiAGENT app:

1.) "Normal" native android apps that used the REST api's of the mobiagent (SDK tools: Android SDK, examples, manuals on how to use the REST API)

2.) OSGI bundles (SDK tools: OSGI SDK, and tools to deploy OSGI bundles on Android based OSGI.

A decision will be made in early 2014 which will be the preferred way for release 1.

2.2.7. ZIP Archive with complete bundle

To be updated in release 2.

2.3. REST API

REST APIs will be available to access the core functionality of both MOBiCENTRE and MOBiAGENt. The API is the crucial interface for service developers. Thus, they directly profit from a clear and well-documented API. In this context, we support the responsible partners with REST API design recommendations and initial specification proposals.

The following sections describe the proposed high-level design process for the creation of a REST API with clear recommendations. In addition, we propose the first version of the REST API specification structure.

2.3.1. Recommendations *for RESTful Web Service Design*

This section provides recommendations and best practices for RESTful Web Service design. One specific aim of this section is to guide a proper translation of the identified (RPC-like) service operations into a well-defined RESTful Web Service API. However, we are not able to cover all relevant aspects in detail and want to refer to the available introductory books ([Richardson2007][7], [Allamaraju2010][1], [Webber2010][2]).

In context of RESTful Web Services, HTTP is used as an application level protocol. Thus, it is important to understand the basic semantics of HTTP like its uniform interface, status codes, and caching

[1] Allamaraju2010: Subbu Allamaraju; RESTful Web Services Cookbook; O'Reilly Media; 2010
[2] Webber2010: Jim Webber, Savas Parastatidis, Ian Robinson; REST in Practice: Hypermedia and Systems Architecture; O'Reilly Media; 2010

characteristics. The above mentioned books cover these aspects as well. For a better understanding, we want to introduce the uniform interface and its specific characteristics.

Table 2 HTTP Methods and Semantics

HTTP Method	Purpose	Characteristics
GET	Retrieve resource representations.	Safe, Idempotent, Cacheable
HEAD	Retrieve resource headers.	Safe, Idempotent
OPTIONS	Lists supported HTTP methods of a resource.	Safe, Idempotent
PUT	Replace / create[3] a resource.	Idempotent
DELETE	Remove a resource.	Idempotent
PATCH	Update the resource state.	-
POST	Primarily used for resource creation. But can be used for any other operation on the resource.	Cacheable (limited support / rarely used, works via HTTP control headers)

In the following, we briefly describe the specific characteristics:

- **Safe:**

  This property implies that an operation is executed without any side-effect to resource state. E.g., searching in the WWW is built upon search indexes which are created via the safe GET method. However, if GET calls changed resource states, the WWW would lose its search capabilities.

- **Idempotent:**

  This property implies that the operation can be performed one or more times with the same result. However, it does not imply that state changes are not allowed. It just assumes that every state change results in the same resource state. E.g., the DELETE operation is idempotent. When the resource is deleted, the DELETE operation can be safely issued several times without any additional changes to the state of the server. Thus, the significant benefit is that an operation can be simply retried if the first attempt seems to have failed.

- **Cacheable:**

  This property implies that representations which are returned by the operation can be used for future request. This property is the main reason for horizontal scalability of the WWW because

[3] PUT should be used for resource creation if the client knows the full new URI. Otherwise POST should be used which creates the new resource below a parent URI.

intermediaries are enabled to fulfil a request directly without going through all hops to the origin server.

Design Methodology

Figure 2 shows the RESTful maturity model, which has been introduced by Leonard Richardson [QConTalk2008][4]. The model shows the evolution from a RPC-oriented Web Service API towards a RESTful API and introduces the RESTful design principles step-by-step.

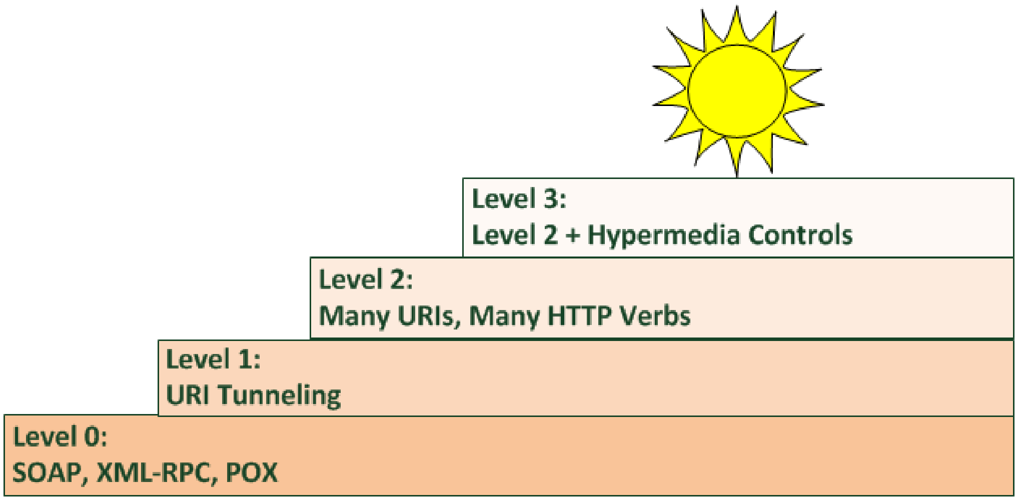


Figure 2: RESTful Maturity Model

We briefly explain the different levels and their semantics:

- **Level 0:**

  On this level HTTP is used as a pure transport protocol. Thus, the remote procedure call is encoded in a specific data format and tunnelled over HTTP. XML-RPC and most SOAP services fit into this level. But this level does not necessarily imply the usage of XML. In fact, formats like JSON, YAML or even a custom format could be used. However, the common synopsis is: “Call a singular endpoint via HTTP POST which responds with a document which contains all required information”. Such a document will also contain all error messages if something does not work as expected.

- **Level 1:**

  At this level we break down a large endpoint into individual resources. I.e., we handle complexity in accordance to the divide and conquer principle. In terms of object-oriented programming, we introduce a kind of object identity. Rather than calling a plain function, we call a method on a respective object (i.e., the resource identified via an URI). However, we still use HTTP for tunnelling requests via POST.

- **Level 2:**

[4] QConTalk2008: http://www.crummy.com/writing/speaking/2008-QCon/act3.html; Access:13.12.2013

At this level requests are no longer tunnelled via HTTP. Instead HTTP methods should be used as close as to their original meaning which enforces a uniform interface and reduces unnecessary variation. E.g., when retrieving the resource state the safe GET method should be used. This method has different advantages over POST because of the different operation characteristics. I.e., the operation does not change the resource state (safe), succeeding operation calls produce the same results (idempotent), and returned resource states can be cached (cacheable). The POST method implies none of these properties (despite the cacheable property in rare cases).

Additionally, further available HTTP semantics should be used like HTTP status codes and HTTP headers. E.g., when a resource has been permanently moved to another location this status change is indicated via an HTTP response 301 and the client can easily redirect to the new URI.

- **Level 3:**

  At this level the last REST attribute is taken into account: Hypermedia as the engine of state (HATEOS). HATEOS implies that returned resource representations contain links to further resources. Thus, the response indicates the next possible actions and the URIs which need to be manipulated for that purpose. In this context, it is important to carefully select media types because they scope the processing semantics and tell the client how to find respective links.

  HAETOS introduces a discovery mechanism and makes the resulting API more self-documenting. Its basic benefits are that the URI scheme may change and new capabilities can be introduced without breaking existing clients.

It is important to point out that only if a Web Service fulfils all properties up to level three, it can be considered as “RESTful”.

Thus, in the following we want to concentrate on resource-oriented design. ROA design can be considered as an “extreme” version of object-oriented design. Basically, you break the system down into its basic parts – its nouns. Each noun is represented by a class and provides means for interacting with other nouns – its methods. In a ROA design based on HTTP these methods are restricted by the uniform HTTP interface (methods and its semantics). Consequently, if you require a more specific method, you have to introduce a resource instead. E.g., if you want to model a subscription to a news feed, you will introduce a specific subscription resource which can be created or queried.

Finally, we briefly summaries the resource-oriented design approach as proposed in [Richardson2007][7]

- **Figure out the data set and split the data set into resources.**

  The service design starts with an idea of the data set which should be exposed. An important aspect is to design the data set at the right level of abstraction. Thus, you should follow a contract-first approach rather than directly exposing existing object models. Then you have to map the different data objects to resources. Basically, we can distinguish three resource categories:

  - Predefined resources which are used to represent top-level directories or the general entry point of the service API.

    - Resources for every data object which is exposed through the service.
    - Resources which represent a read-only view on a data set which results from an applied algorithm (e.g., search algorithm).

  Thus, a RESTful Web Services exposes its data and its algorithms as resources. Typically, the resources are aligned into a hierarchy which starts out small and branches into an infinite number of leaf resources.

  After you have identified your basic resource types, these steps have to be performed for each of them:

- **Name the resources with URIs**.

  The URI contains all scoping information. For naming you should follow these general guidelines:

    - Use path variables to express hierarchy (e.g., “/user/user1/preferences”).
    - Use punctuation characters to avoid expressing hierarchy where none exists, e.g., “/earth/43.8,17.9” to uniquely express a location by its longitude and latitude.
    - Use query parameter to express inputs for algorithms (e.g., “/user?q=age>20”).

  URI templates [RFC6570][5] should be used to document the resource URIs in compact character sequences (e.g., /user/{user-names}).

- **Expose a subset of the uniform interface.**

  In this context, you have to decide which methods of the uniform HTTP interface are applicable for your resources. E.g., for read-only resources, GET and HEAD need to be supported. For other interactions, the respective HTTP methods should be used (creation: PUT or POST, modification: PATCH, removal: DELETE). If you require a more specific method to interact with the resource, you should introduce a new resource rather than (mis)-using the POST method.

- **Design the representations.**

  In this context, you need to define the corresponding input and output representations for the supported HTTP methods. The output representations should contain links which establish the connection with other identified resource types. Thus, they define the next possible application states. The selected media types establish the interface contract by defining the processing model of the representation. Particularly, you can also support multiple representations for different client types. The client can select the required representation by indicating the MIME type in the “Content-Type” HTTP header. E.g., an AJAX application could select a JSON-based representation and human user would request a readable plain HTML representation.

  However, there is no real standard way to express linking semantics. In the following we provide some basic guidelines for proper media type selection:

[5] RFC6570: http://tools.ietf.org/html/rfc6570; Access 12.12.2013

  - Avoid the usage of generic non-hypermedia types like “application/json” or “application/xml” because they do not understand the semantics of links.
  - Avoid usage of URI templates to expose all resource types because this approach leads to tight coupling with clients.
  - Re-use existing hypermedia types ([MediaTypes][6], [Richardson2007][7]). E.g., the usage of the extensible Atom Syndication Format [RFC4287][8] is often encouraged which defines a XML Schema and the semantics for link relations. Another option is to make use of XHTML or microformats [Microformats][9] which extend XHTML with domain-specific semantics (e.g., calendar information, contact information).
  - Another popular option is to define vendor-specific MIME types [VendorMIME][10]. This approach allows you to define a hybrid format which describes both data processing and link semantics. E.g., the MIME type “application/vnd.itinerary+json” could describe an itinerary-specific client protocol. However, you should only use a vendor-specific MIME type if no existing fits your needs. Another benefit of this approach is that the MIME type can encode versioning information. E.g., “application/vnd.itinerary-v2+json” would indicate the second version of this API. The advantage is that you do not clutter the URI design with versioning information. The GitHub API [GitHubMIME][11] gives a very good example of this approach.

  Representation design is the most challenging part of the API design. Further information is presented in [Amundsen2011][12].

- **Consider the communication flow.**

  Typically, when a client requests a resource via GET, the server returns HTTP headers, a representation, and a return code 200. However, HTTP headers fields ([RFC2616][13], section 14) and return codes ([RFC2616]8, section 10) offer additional means to tweak the communication.

  E.g., conditional GET requests avoid unnecessary representation transfers and save bandwidth. To implement conditional GET requests, the response header “Last-Modified” and the request header “If-Modified-Since” can be utilised. I.e., the client can store the provided last modification timestamp. When the client tries to retrieve the representation again, it can issue the timestamp with the request header “If-Modified-Since”. Consequently, if no new version of the representation is available, the server returns a 304 code (“Not Modified”) and will not transfer the representation.

[6] MediaTypes: http://www.iana.org/assignments/media-types; Access: 13.12.2013
[7] Richardson2007: L. Richardson and S. Ruby; RESTful Web Services; O’Reilly; May 2007; p. xi
[8] RFC4287: http://tools.ietf.org/html/rfc4287; Access: 13.12.2013
[9] Microformats: http://microformats.org/; Access: 13.12.2013
[10] VendorMIME: http://tools.ietf.org/html/rfc4288#section-3.2; Access: 13.12.2013
[11] GitHubMIME: http://developer.github.com/v3/mime/; Access 13.12.2013
[12] Amundsen2011: Mike Amundsen; Building Hypermedia APIs With HTML5 and Node; O'Reilly Media; 2011
[13] RFC2616: http://www.ietf.org/rfc/rfc2616.txt; Access: 13.12.2013

For requests which cannot be fulfilled by the server-side implementation, corresponding return codes in the 3xx, 4xx, and 5xx range should be used as close as to their original meaning. Additionally, the response could return a document which describes the specific error conditions.

- **Document it.**

  Finally, you should provide sufficient human-readable documentation. At least you should describe the following aspects of the service API:

  - All resource types with their supported methods.
  - Media types which are used for requests and responses.
  - The predefined URIs which are not reachable via links.
  - Query parameters for algorithmic resource types.
  - Specifically used HTTP error codes and headers.

  The concrete proposal for the interface specification is shown in the next section.

2.3.2. REST API Specification Template

In the following we show the interface specification for a component that provides N REST interfaces.

Component name

Short description of the component

Interface name (1-N)

Description

Short description of this interface of the component

URL Structure

http://a.b.c.d/component/interface

Method

GET or POST or DELETE, or …

Input Parameters

Name	Type	Description

name	Description
name	Description

Output

URL encoded string or json string

Name	Type	Description
Name		Description
Name		Description

Exception

Any exceptions that can occur, and what the return code and content will be.

Example

To request … from component …., issue the REST request:

Get `http://a.b.c.d/component/interface?para=dder4&parb=scft6`

This will results in a description of …. ,encoded as json:

```
{
      asdfasdf : ”sdffft”,
      egvf : [4,5,6],
      tgbh: { a: “b”,
            c: false}
}
```

2.4. Tutorials

Concrete tutorials will be provided during development and are continuously updated. In the following we introduce the small example service which we want to use throughout the initial tutorials.

2.4.1. Example Service

The Greetings App for Android is a simple RESTful mobile client to demonstrate user interaction with an application that uses RESTful web services in background. The following figure shows an overview of the principle structure of the service. Basically the service provides a method to add new users using HTTP PUT operation and a method to retrieve user information by using HTTP GET operation. In our sample we use JSON for exchanging content between client and web service.

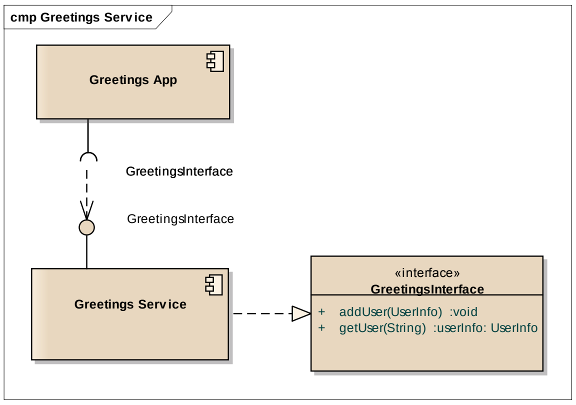
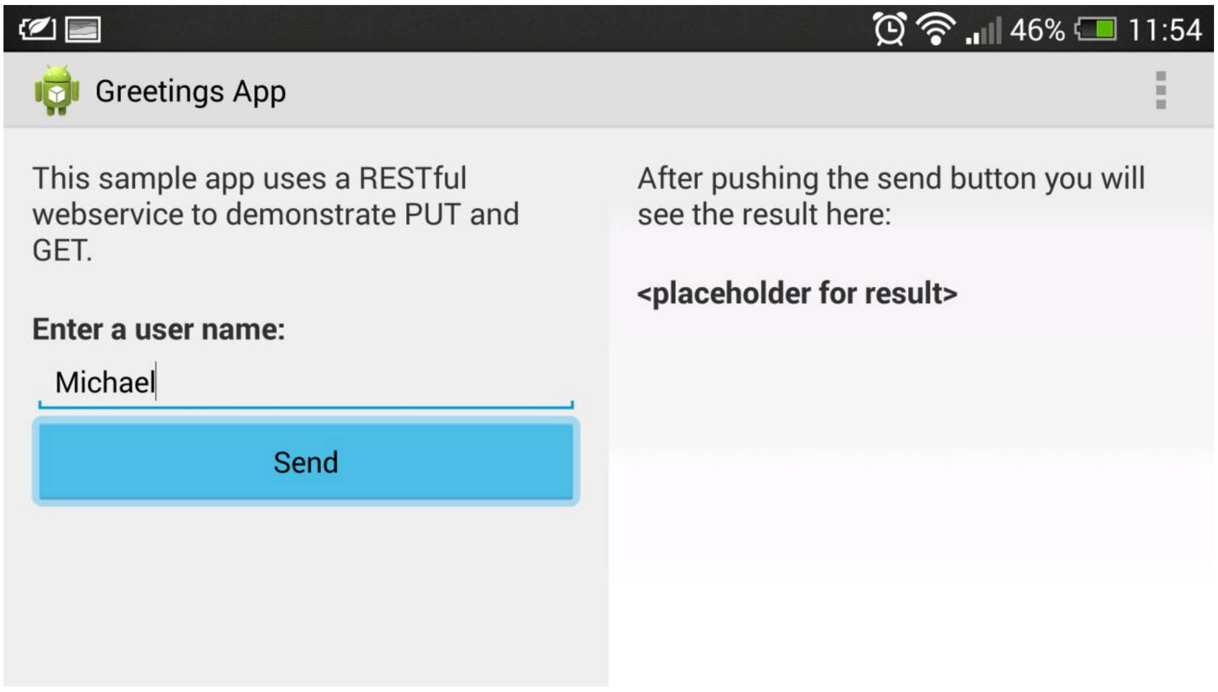


Figure 3: Building block view of the Greetings Service

The App has two separated views. The first view contains a simple input form to enter a user name and send content to the RESTful web service. The second view shows the result provided by the web service. In this case it welcomes the user. The screenshots in Figure Figure 4 and Figure 5 show both views side by side. However, both views can be shown standalone.



This sample app uses a RESTful webservice to demonstrate PUT and GET.

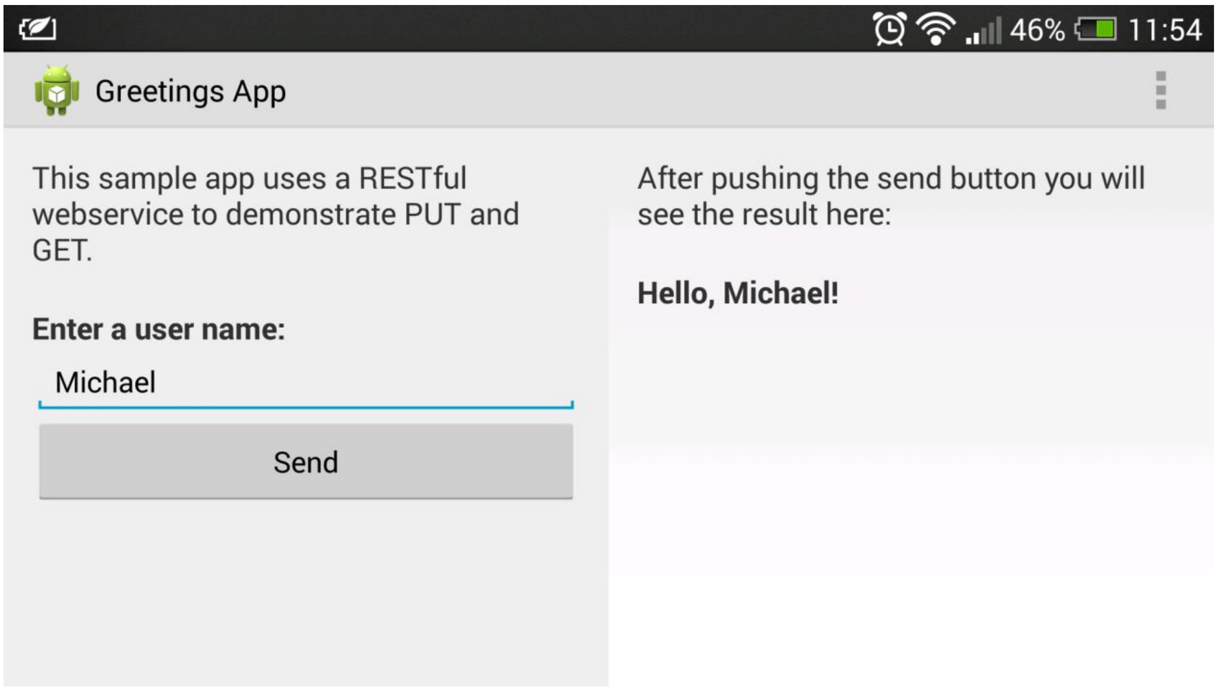
Enter a user name:

Michael

After pushing the send button you will see the result here:

<placeholder for result>

Figure 4: Initial start-up view


46% 11:54

Greetings App

This sample app uses a RESTful webservice to demonstrate PUT and GET.

Enter a user name:

Michael

Send

After pushing the send button you will see the result here:

Hello, Michael!

Figure 5: Result view when service is used

3. Summary

This deliverable explains the components of the Service Directory KIT and the key targets of each of the components.

A selection of tools that will be developed for release 1 is given. They consist of Eclipse plug-ins which are described in detail and with relevant examples and links. The plug-ins will be available as bundle for both Windows and Unix developer platforms.

To access the core functionality of both MOBiCENTRE and MOBiAGENT REST APIs will be available. In this document the REST API design recommendations and initial specification proposals are given.