BONVOYAGE

From Bilbao to Oslo, intermodal mobility solutions, interfaces and applications for people and goods, supported by an innovative communication network

Research and Innovation Action GA 635867

Deliverable D6.1:

Technology dependent interfaces

Deliverable Type:	Report
Deliverable Number:	6.1
Contractual Date of Delivery to the EU:	30.04.2017
Actual Date of Delivery to the EU:	29.04.2017
Title of Deliverable:	Technology dependent interfaces
Work package contributing to the Deliverable:	WP6
Dissemination Level:	Public
Editor:	Stephan Strodl (FLU)
Author(s):	Stephan Strodl, Daniel Skrach, Roman Pickl (FLU), Giuseppe Tropea, Andrea Detti (CNIT),

	Etienne Labyt, Audrey Vidal (CEA), Raffaele Gambuti, Federico Lisi, Silvia Canale (CRAT), Ignacio Gonzalez Fernandez, Guillermo Ibanez Gomez (ATOS)
Internal Reviewer(s):	TRIT
Abstract:	Deliverable D6.1 aims at the documentation of the technology dependent interfaces towards the external actors. It contains the description of design and implementation of the platform services that provide interfaces to both internal and external stakeholders as well as the specification of their APIs, algorithms, protocols and low-level architectures/functions, where applicable. It also contains information about external services used by the platform and the design of the BONVOYAGE mobile application.
Keyword List:	Interfaces, APIs, Architecture

TABLE OF CONTENTS

List of Figures

List of Tables

Abbreviations

ABBREVIATION	DEFINITION
DoA	Description of Action
DS	Data source
ICS	Internames Communication System
MDHT	MetaData Handling Tool
MMMDB	Multi-Modal Mobility Database
OGB	OpenGeoBase
WP	Workpackage

Table 1: Abbreviations

BONVOYAGE Glossary

Table 1 lists and describes the terms that have been considered relevant in this deliverable.

BONVOYAGE GLOSSARY	
TERM	**DEFINITION**
ICS	Internames Communication System
Metadata Handling Tool	The MDHT is the component of the BONVOYAGE architecture that performs monitoring and parsing of the data coming from external Data Sources of Transport Operators
Multi-Modal Mobility Database	The MMMDB is the component of the BONVOYAGE architecture that holds essential information about users' profiles and about multi-modality of the computed solutions
OGB	The OpenGeoBase is the discovery service of the BONVOYAGE platform
Orchestrator	The BONVOYAGE Orchestrator is a decomposition approach to solve the trip planning on a multimodal network by means of soloists. Orchestrators can act as national access points for trip planning.
Soloist	The soloists are distributed trip planning services to handle trip planning tasks for a part of the overall transportation and road network.
SPROUTE	SPROUTE is an open source format to exchange routing information (request and response) as JSON objects.

Table 2: BONVOYAGE Dictionary

1 Introduction

1.1 Deliverable rationale

Work Package 6 (WP6) focuses on designing and developing all the mechanisms needed to seamlessly interact with the heterogeneous external actors of the BONVOYAGE platform. In particular, this WP is in charge of the sensing and actuation functionalities of the platform.

As stated in the Description of Action (DoA), the sensing functionalities are not limited to conventional, technology dependent, adapters to access transport operators' data sources and systems, but also mobile applications and optimized interfaces to gather end user data and feedback (participatory sensing).

The actuation functionalities of the platform interact with the specific Transport Operators' platforms based on the Intelligent Transport Functionalities of the platform (developed in WP4) and the service adaptation functionalities (developed in WP5).

Due to the heterogeneity and technological complexity of the different research fields involved in WP6, the Work Package has been structured in three main tasks:

- Task 6.1: Technology / Operator dependent interfaces
- Task 6.2: Apps
- Task 6.3: Modelling and performance analysis in realistic scenarios

The main objective of this deliverable (Deliverable D6.1) is to document the design and development of the technology dependent interfaces towards the external actors (transport operators' and related systems / data sources, end user applications). It therefore contains the description of the components providing the multimodal integrated interfaces to internal and external stakeholders as well as the specification of the APIs, algorithms, protocols and low-level architectures/functions, where applicable. For further details on the relationship between the use cases defined in WP2 and the APIs specified in this document see Deliverable D7.1 Integration Plan.

1.2 Quality review

The internal reviewer of this deliverable is TRIT.

VERSION CONTROL TABLE			
VERSION N.	**PURPOSE/CHANGES**	**AUTHOR**	**DATE**
0.1	Initial draft	FLU	31.10.2016
0.2	Revised structure	FLU	08.02.2017
0.3	Contributions by partners	CRAT, CNIT, CEA, FLU	16.03.2017
0.4	Consolidation and document structure	FLU	22.03.2017
0.45	Update of car pooling section	CRAT	11.04.2017
0.5	Introduction added	FLU	12.04.2017
0.6	Update of CEA contribution	CEA	18.04.2017
0.7	Contribution for data sources	ATOS	20.04.2017
0.8	First draft	FLU	21.04.2017
0.82	Second draft	CNIT	24.04.2017
0.84	Third draft	FLU	25.04.2017
0.86	Fourth draft	CNIT	26.04.2017
0.9	Quality review	TRIT	27.04.2017
1.0	Final version	FLU, CNIT	28.04.2017

1.3 Executive summary

1.3.1 Deliverable description

The aim of this deliverable is to present the public APIs of BONVOYAGE to external stakeholders. Furthermore, external APIs that are used (e.g. Firebase) are discussed. The deliverable does not repeat the work of WP3, 4, and 5 but provides a guide about how the BV platform can be used by external stakeholders (e.g. app, platform or service provider developers). It is a technical deliverable that focuses on the interfaces implementation of the services developed within the BONVOYAGE project.

This document is organized in the following sections:

- Section 1 describes and summarizes the deliverable organization.
- Section 2 describes how the components providing the multimodal integrated interfaces fit into the BONVOYAGE architecture.
- Sections 3 to 5 deal with the BONVOYAGE components (internal services) at the three layers of the architecture that provide public APIs to external stakeholders.
- Section 6 deals with components that consume the services provided by the BONVOYAGE platform (e.g. the BONVOYAGE app) or are consumed by BONVOYAGE services (e.g. external data sources and services).
- Section 7 summarizes key results and concludes the deliverable.

1.3.2 Summary of results

This deliverable provides extensive descriptions of the components and APIs as a guideline for external integrators to seamlessly harness functionalities developed within the BONVOYAGE project.

In particular, the following objectives where achieved in accordance with the Description of Action (DoA):

- Evaluation and selection of relevant systems of transports operators and related data sources
- Design of the technology dependent interfaces, in compliance with the specification and the architectural design output of WP2.
- Design of optimized interfaces (REST/JSON based) between the BONVOYAGE platform and proper mobile applications.

- Implementation of the interfaces in the chosen programming languages and frameworks.
- Preliminary functional unit tests to verify the correct behaviour of the implemented software components.

2 Technology Dependent Interfaces

In a nutshell, this Technology Dependent Interfaces deliverable provides fundamental technical guidelines for the exploitation of the project results of BONVOYAGE. The deliverable presents the technical aspects of the services developed in WP 3, 4, 5 and 6, providing a practical guideline for usage and integration. The definitions of the various interfaces include the type of interface (e.g. web service or JAVA interface), input and output data and deployment endpoint. The used data formats are specified in detail, and we give practical examples of the data itself. The provided information enables external systems to integrate BONVOYAGE services into existing or new solutions with reasonable effort.

More in details, and following along the lines depicted in Figure 1, this deliverable presents the **internal services provided by** the BONVOYAGE platform in terms of the public APIs they offer to external users. They enable external stakeholders to integrate BONVOYAGE services into their environment. The detailed description of the interfaces, including examples and error codes, ensures a seamless integration with minimal effort into other environments. The deliverable focuses on the technical aspects of the services and on the APIs they provide.

Furthermore, the deliverable discusses the **strategies for integration of external services** and data sources into the BONVOYAGE platform. External services include services for authentication based on reuse of existing user identities and front-ends to travel data sources that are integrated by using a federation approach (see D1.2 Project Vision).

The mobile application of the BONVOYAGE platform is also discussed in this deliverable as it represents a practical example of the usage of BONVOYAGE services by an external entity. The mobile app is easily built on top of the services offered by the BONVOYAGE platform by solely using the aforementioned public APIs, including the Android libraries for User Mode Transportation Recognition and User Stress Level, developed within the project.

Figure 1 presents the high-level architecture of the BONVOYAGE platform in terms of these internal (depicted at the top, in blue colours) and external services (depicted at the bottom, in green colours), and in terms of the internal services' APIs (continuous black arrows), since they provide publicly available interfaces. Dashed lines do not represent APIs but flow of information from external services to the platform, instead. The services are organized into three different Layers: Application, Orchestrator and Infrastructure. Services that provide functionalities for the end user (such as calculating Greenpoints or collecting feedback) are at the Application Layer. Services for computing routes and merging travel solutions are at the Orchestrator Layer. Services that provide functionalities for communication, federation and discovery of data sources, such as Internames Communication System (ICS) and OpenGeoBase (OGB) are at the

Infrastructure Layer. Both ICS and OGB provide an extensive set of functionalities to support infrastructural and communication service that can be easily integrated into other environments. While this deliverable presents their detailed APIs, other deliverables are devoted to the exhaustive description of how they are used to gain access to external data services hosted by Transport Operators, which are more relevant for the Infrastructure Layer, and are made available via a federation of Data Sources (see again Figure 1). Specifically, the reader is redirected to deliverable D5.1 for what concerns the adaptation of data sources to BONVOYAGE, D3.1, D3.2, D3.3 for the technicalities of how the communication infrastructure is used to disseminate travel data efficiently, and to D7.1 on how the various pieces are integrated. A brief high-level overview is given in section 6.3 of this deliverable to serve the purpose of descriptive unity and continuity.

Figure 2 provides a detailed view of the Application Layer, to clarify how external integrators can develop end user constructs based on BONVOYAGE. Services at the Application Layer focus on functionalities such as conveying route information to users, managing user feedback and offering car-pooling. The figure shows that the Application Server, which acts as a single point of contact, provides the hub for these services. The Application Server handles requests from the external users and distributes the request to the corresponding modules and services. The BONVOYAGE mobile application communicates only with the Application Server that provides all required interaction. The mobile application itself is a complex construct that also makes use of other Application Layer services developed in BONVOYAGE, which are libraries for real-time recognition of the User Transport Mode and Stress Level. The APIs of these components and guidelines for reuse are presented in this deliverable, too. The mobile app uses the APIs provided by the modules through the Application Server for feedback, Greenpoints and route calculations, as described later on. The Application Server acts as a proxy for the requests and provides error handling in terms of timeouts and authentication.

Regarding the integration of external services at the Application Layer into the BONVOYAGE ecosystem, Figure 2 shows for example the use of the Firebase service for authentication, as discussed in Section 6.2, by the Application Server, and the usage of standard (for instance, Android) toolkits for mobile apps GUI developing by the Mobile App.

Summing up, this deliverable is meant as a practical guide for reuse and integration of all publicly available services and the relevant components. It allows gaining further technical insight into the BONVOYAGE implementation, and is propaedeutic to reading D7.1.

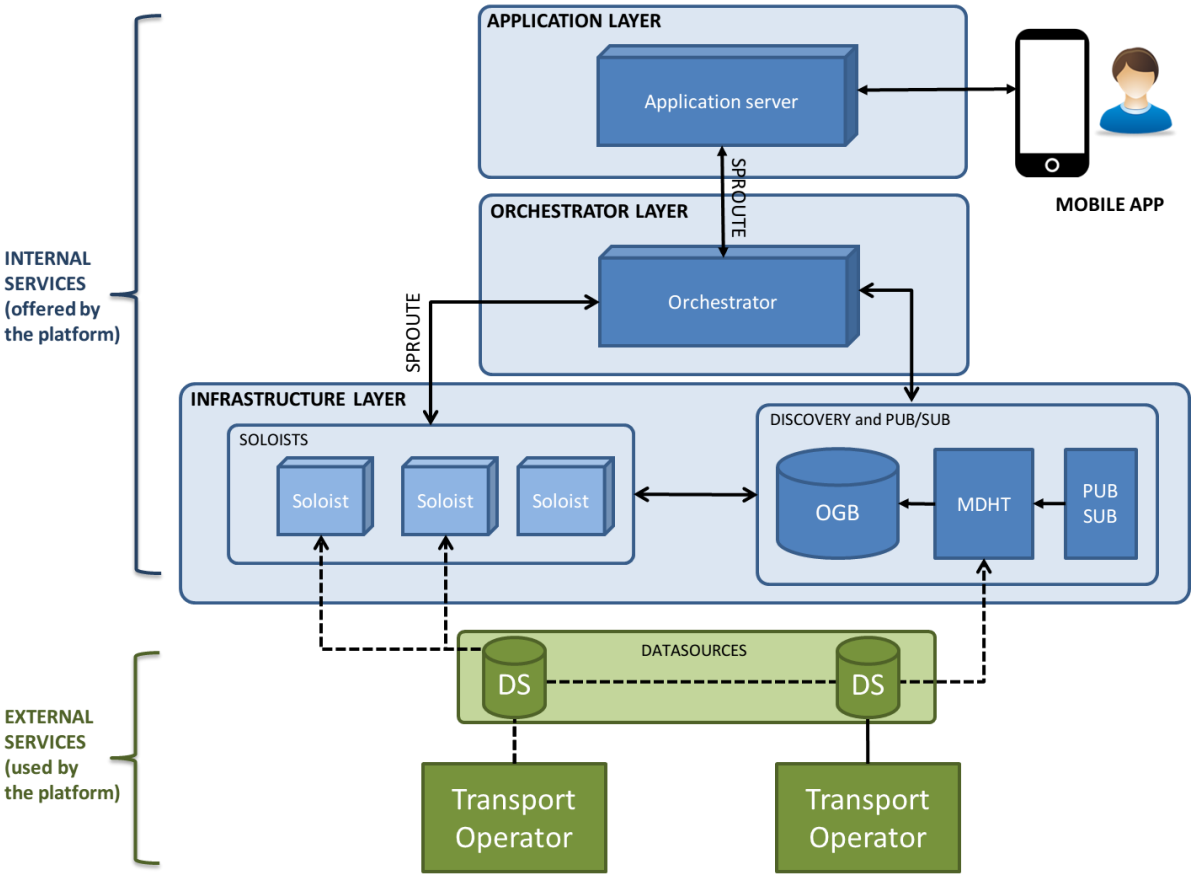

Figure 1 High-level architecture with internal and external services

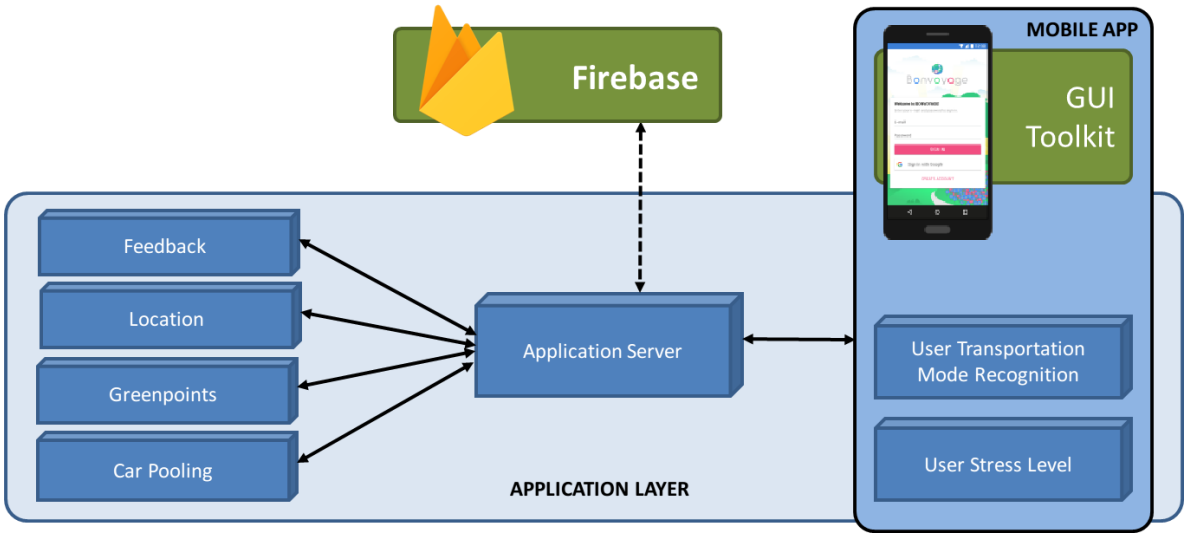

Figure 2 Detail of the Application Layer and Mobile App

3 Internal Services at the Application Layer

This section presents the interfaces' details of the BONVOYAGE Application Layer components (services) that provide public APIs to external stakeholders. Generally speaking, all internal services that are deployed within the BONVOYAGE ecosystem provide public APIs that can be used by end users as well as system integrators.

BONVOYAGE implements a fidelity program that motivates user to use more sustainable travel solutions. The implementation of the Greenpoint module is presented in Section 3.1 including a Greenpoint profile providing user feedback to their travel choices. Feedback from the end users can be reported via the feedback module in Section 3.2. Location based visualisation on end user's devices is supported by the location service as described in Section 3.3. The Android libraries for User Model Transport Recognition and User Stress Level are presented in Section 3.4. Section 3.5 describes the BONVOYAGE car-pooling service and its interfaces.

3.1 Greenpoints module

3.1.1 Description

The Greenpoints module implements the Green Score Policy of BONVOYAGE. The policy evaluates the user travel behaviour in time and categorizes it into four different eco-friendliness profiles. The policy considers the past travel choices of the user. It encourages the user to select more sustainable travel solutions. Based on the profiles, different users will receive different scores for the same travel solution, thus providing incentives/penalties. Detailed description about the Green Score Policy is provided in *D4.1 Design of the Intelligent Transport Functionality*. The Greenpoints module implements services to calculate the individual greenpoints for the user and compute CO2 for travel alternatives. Based on the user's choice the profile is updated. The profile implements a feedback loop to inform the user about the impact of the choices. Moreover, the sum of collected greenpoints and CO2 emissions of all trips are held in the profile.

The Greenpoints module is implemented as two services, the profile information and the calculation service. The information service provides a web service to get the user's greenpoints data. The calculation service adds the greenpoints and CO2 emissions for an individual user to the route alternatives. All route requests to the BONVOYAGE platform are managed by the Application Server, where the greeenpoint information is glued the route alternatives and packaged as SPROUTE format. The extension of the format is in described in *Deliverable 5.1 Design of the adaptation functionality,* while the details of the GetRoutes routing API is described in Section 4.1.

3.1.2 API

GreenPoint Information Profile Service	
URL	https://test-bonvoyage.fluidtime.com/user/profile/score
Input Data:	No input data are needed JWT authorization is required in the HTTP header for authorisation of the user.

Output Data:	UserProfile (JSON)
	
	Scores {
	greenpoints: number
	co2Emission: number
	savedTrips: integer *
	ecoBehaviour: integer *
	}
	
	Response codes:
	200 OK
	403 The provided security credentials did not validate
	500 General error
	503 External web service unavailable or an error occurred while communicating with an external web service.

Communication protocol	GET HTTP/JSON

JAVA API – Application Service

GreenPoint Calculation Service	
URL	public RouteResult transform(RouteResult sourceResult)
Input Data:	Route alternatives (SPROUTE FORMAT) USER-ID (FIREBASE TOKEN)
Output Data:	Route alternatives (SPROUTE FORMAT)
Communication protocol	Java

3.1.3 Data formats

The greenpoint values are added to SPROUTE definition for route and route segment as shown in Figure 3. The greenpoints of a route is a personalised calculation based on the user profile.

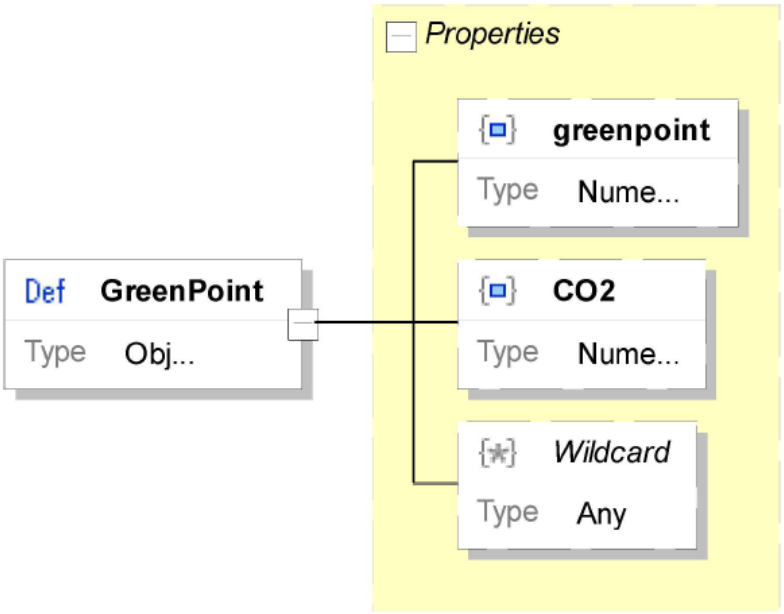

Figure 3 SPROUTE Data object GreenPoints

3.1.4 Usage

Greenpoints and CO2 emissions are shown in the BONVOYAGE mobile application for each travel option and for each leg of the travel option as shown in Figure 4.

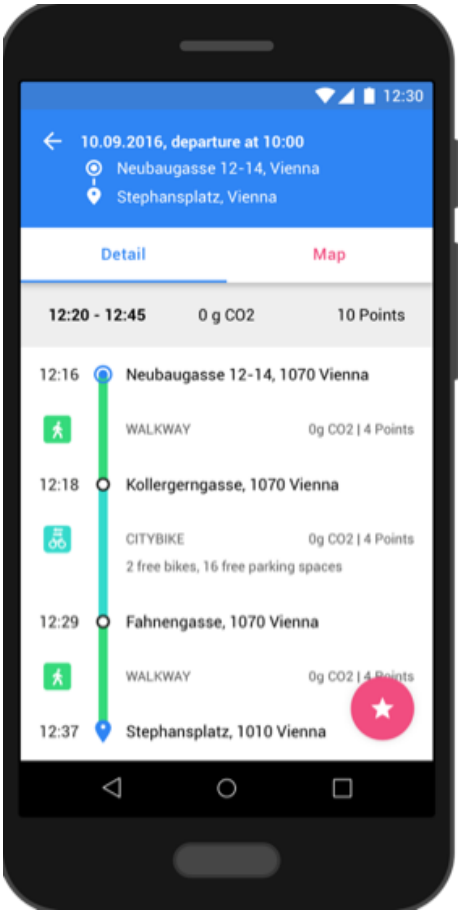

Figure 4 BONVOYAGE App Route Details

Figure 5 shows two examples of the profile screen of the application providing a quick visual feedback to the user about the travel behaviour.

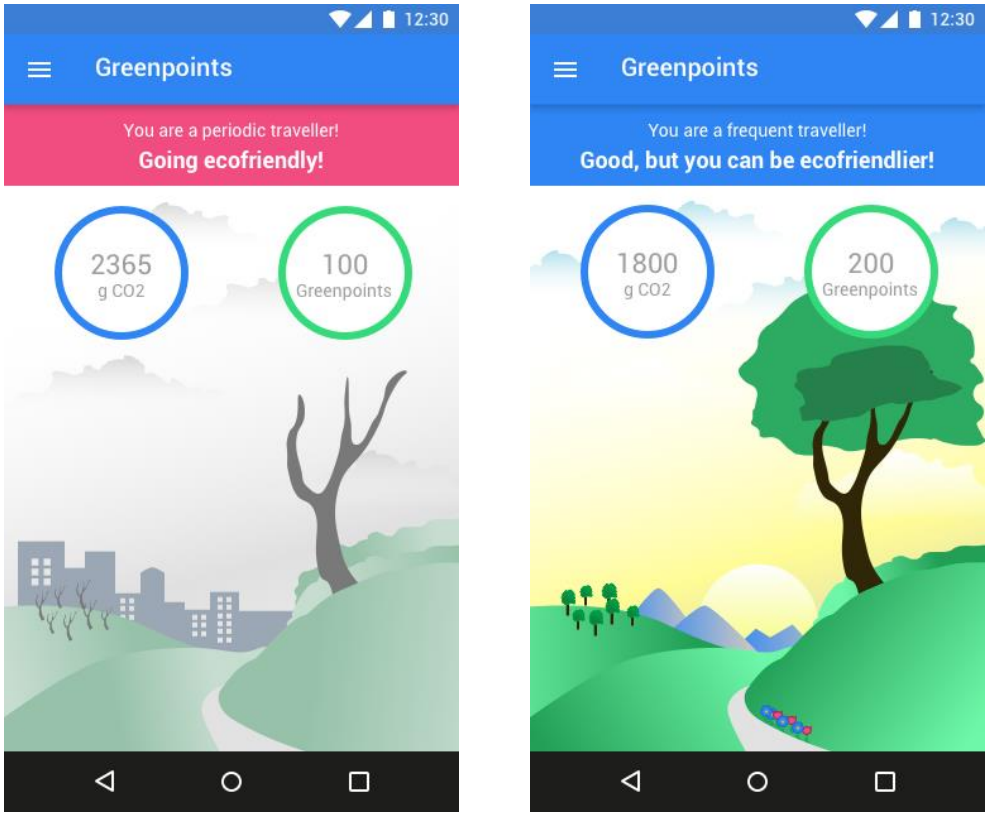

Figure 5 BONVOYAGE App Greenpoint Profiles

3.2 Feedback module

3.2.1 Description

The feedback module provides the mechanism to get the end user's response about the BONVOYAGE service and the actual trip. The module implements a web service to post the feedback to the BONVOYAGE platform. Three types of feedback are currently implemented in the BONVOYAGE mobile application: Feedback to the BONVOYAGE Service, travel solutions and actual trip.

3.2.2 API

Feedback Service	
URL	https://test-bonvoyage.fluidtime.com/user/feedback

Input Data:	Feedback {
	Feedback type: integer
	tripID: string
	rating: integer
	text: string
	}

Output Data:	Response codes: 201 OK 400 The provided input parameters did not validate. 403 The provided security credentials did not validate 500 General error 503 External web service unavailable or an error occurred while communicating with an external web service.
Communication protocol	POST HTTP/JSON

3.2.3 Data formats

The feedback is provided as a JSON object, as content of a POST request. It includes the feedback type, a reference to the trip, a rating and a free text.

```
Feedback {
FeedbackType: integer
tripID: string
rating: integer
text: string
}
```

3.2.4 Usage

The feedback module is used within the mobile application to submit the feedback of the users about the BONVOYAGE Service, travel solutions and actual trip as shown in Figure 6.

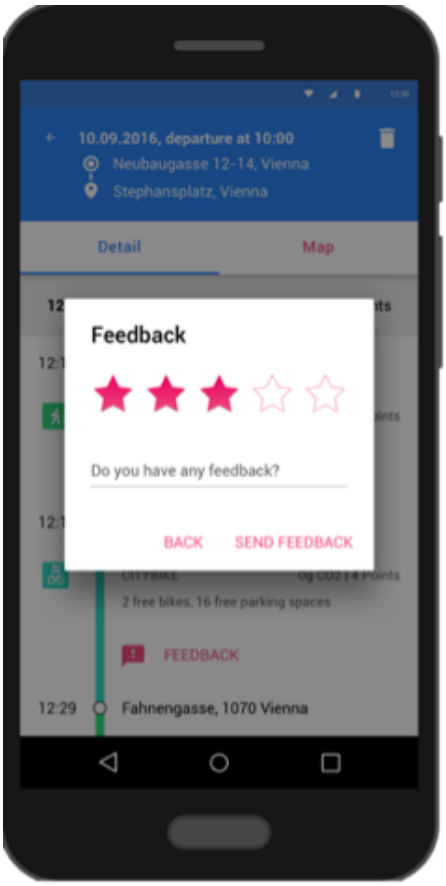

Figure 6 BONVOYAGE Mobile Application Feedback

3.3 Location module

3.3.1 Description

The location module provides geo information and descriptions about POIs for the mobile application. The module is able to aggregate different data sources before presenting the data, so that some POIs can be, for instance, discovered and fetched through OGB and aggregated by the location module for presentation at the client interface with data coming from other mobile-

app-specific channels. The module provides services to request POIs in the surroundings or for a precisely specified area, in order to display them on the mobile device. Thus, two services are deployed: (1) POIs by radius and (2) POIs by bounding box.

3.3.2 API

<table>
<tr><th colspan="2">LocationByRadius</th></tr>
<tr><td>URL</td><td>https://test-bonvoyage.fluidtime.com/locationByRadius</td></tr>
<tr><td>Input Data:</td><td>
<table>
<tr><th>Name</th><th>Description</th><th>Mandatory</th><th>Type</th></tr>
<tr><td>cLon</td><td>The center coordinate's longitudinal value.</td><td>Yes</td><td>number (double)</td></tr>
<tr><td>cLat</td><td>The center coordinate's latitudinal value.</td><td>Yes</td><td>number (double)</td></tr>
<tr><td>limit</td><td>The maximum number of returned locations.</td><td>No</td><td>integer (int32)</td></tr>
</table>
</td></tr>
<tr><td>Output Data:</td><td>Response codes:
201 OK
400 The provided input parameters did not validate.
500 General error

LocationResult:
description: "Result of a request for locations."
properties:
locations:
description: "List of returned locations."
items:
$ref: "#/definitions/Location"
type: "array"
type:"object"

Location {
Represents a geo-coded Location.
address:
{
description: string</td></tr>
</table>

	identifier: string
	name: string *
	uri: string
	}
	coordinate:
	NormalizedWgs84Coordinate {
	A coordinate in the WGS84 coordinate system.
	latitude: number *
	longitude: number *
	}
	description: string
	}

Communication protocol	POST HTTP/JSON

LocationByBoundingbox	
URL	https://test-bonvoyage.fluidtime.com/locationByBoundingbox
Input Data:	

Name	Description	Mandatory	Type
neLon	The north-east boundary coordinate's longitudinal value.	Yes	number (double)
neLat	The north-east boundary coordinate's latitudinal value.	Yes	number (double)
swLon	The south-west boundary coordinate's longitudinal value.	Yes	number (double)
swLat	The south-west boundary coordinate's latitudinal value.	Yes	number (double)
limit	The maximum number of returned locations.	No	integer (int32)

Output Data:

Response codes:

201	OK
400	The provided input parameters did not validate.
500	General error

LocationResult:
description: "Result of a request for locations."
properties:
locations:
description: "List of returned locations."

	items:
	$ref: "#/definitions/Location"
	type: "array"
	type:"object"
	
	Location {
	Represents a geo-coded Location.
	properties:
	address:
	description: "Address related to the enclosing context."
	properties:
	description:
	description: "Human readable description of the enclosing context."
	type: "string"
	name:
	description: "Textual representation of the address."
	type: "string"
	
	coordinate:
	NormalizedWgs84Coordinate {
	A coordinate in the WGS84 coordinate system.
	latitude: number *
	longitude: number *
	}
	categoryId:
	description: "Location category identifier."
	type: "string"
	description:
	description: "Human readable description of the enclosing context."
	type: "string"
	}

Communication protocol	POST HTTP/JSON

3.3.3 Data formats

The result data format is a simple JSON response as defined in Section 3.3.2. It consists of an array of the Location data type. The simple Location format contains the following data objects: an optional address field, coordinates, category id and a description.

3.3.4 Usage

Below an example of a response is shown that can be used to display the information in the mobile application, either as list or as locations in the map.

```
{
  "locations": [
    {
       "address":
         {
         "description": "W07-Lindengasse",
         },
        "coordinate": {
        "longitude": 16.350818280467,
        "latitude": 48.200201102435
      },
      "categoryId": "1",
      "description":"City Zipcar",
   },
   {
      "address":
         {
         "description": "Lindengasse 31-33",
         },
      "coordinate": {
        "longitude": 16.351311666667,
        "latitude": 48.200295
      },
      "categoryId": "102"
      "description": " W-5YOY Sym 50 ccm"
    },
    {
          "address":
         {
         "description": " Kollergerngasse ",
         },
         "coordinate": {
        "longitude": 16.350918,
        "latitude": 48.198527
      },
      "categoryId": "6",
      "description":    "Mariahilferstraße    Ecke    Kollergerngasse<br><br><a
href=\"\"http://www.citybikewien.at\"\">Citybike    Website</a>    18    freie
Stellplätze, 2 freie Räder
    }

   }
```

3.4 User Transportation Mode Recognition Library and User Stress Level Library

Within the BONVOYAGE project two Android libraries (one for detection of stress level and another one for recognition of transport mode) are developed. The libraries are integrated into the BONVOYAGE App (described in Section 6.1), but can also be integrated into other Android applications. These libraries include:

- Java code to connect to smartphone sensors (for transport) or Empatica wristband sensors (for user's stress level) and collect data
- Java code to process sensors data, extract features and perform classification to provide as output either a transport mode (ex: class 1 = road, class 2 – train…) or a stress level (from 0 to 1 by step of 0.1).

Within the BONVOYAGE context, these components are integrated into the BONVOYAGE App. Transport mode and stress output data will be sent from the application to the User Profiler Tool back end (UPT-BE) (please refer to D4.x for the details) for user clustering and refining the user profile, either for pre-trip planning (based on refined profile from data collected during previous travels) or on trip assistance (based on real time data collected during the travel).

The UTMR stands for User Transportation Mode Recognition and is a java/android library which aims at recognizing, in real-time, the current transportation mode of the smartphone user (car walk, bike, …) through the use of smartphone sensors.

3.4.1 Integration of UTMR Library

3.4.1.1 Application manifest file

Service declaration

The usage of the service has to be declared in the main application manifest file.

```
<service android:name="bonvoyage.service.UTMRService" />
```

Required Permissions

The main android application should have at least the following permissions

- Read and write into external storage: recognition models have to be copied and read on the smartphone memory.

- Access coarse or fine location. This is mainly for the GPS and BLE use.

Note: not having those permissions granted may cause the application crash.

The following lines have to be added to the manifest file :

```
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
```

In Android 6 and further, permissions need to be granted explicitly by the smartphone user. Applications integrating the library should check and ask the user before calling the library.

3.4.1.2 Project integration

This part explains how to integrate the lib in Android Studio. The “Bonvoyageservice-release.aar” needs to be copied in the project structure under the “libs” folder. This folder is a child of the “app” folder and sibling of the “src” folder. The following lines need to be added to the application gradle file:

```
repositories
{
        flatDir
        {
                dirs 'libs'
        }
}
dependencies
{
        compile(name:'bonvoyageservice-release', ext:'aar')
}
```

The UTMR library uses third party libraries, which need to be added in the project in the application gradle file:

```
dependencies {
compile 'com.android.support:appcompat-v7:<LOCAL_VERSION>
compile 'nz.ac.waikato.cms.weka:weka-stable:3.6.13'
compile 'commons-io:commons-io:2.5'
compile 'com.google.guava:guava:16.0.1'
}
```

The library should then be ready for usage in the application. For a complete gradle file, please check the UTMRTutoApp's gradle file. The minimal Android SDK version is 17 (JellyBean 4.2).

3.4.2 UTMR Library Usage

3.4.2.1 Start / stop UTMR service

This library is designed as an android service. It should be started with the "startService" command.

```
import bonvoyage.service.UTMRService;

Intent intent = new Intent(MyActivity.this, UTMRService.class);
startService(intent);
```

The service should start.

It first tries to copy the recognition model on the smartphone memory, then loads the model and starts listening to the available sensors of the smartphone.

If something goes wrong in the copying / loading files process, the service stops itself and sends its status via broadcast.

The service is started with the "Start_sticky" flag which means that if something stops for any reason other than it was asked to by the user, the service will try to start again.

To stop the service you should use the command stopService:

```
Intent intent = new Intent(MainActivity.this, UTMRService.class);
stopService(intent);
```

The *INTENT_UTMR_SERVICE_RUNNING* broadcast should warn you if the service has stopped correctly or not. The Boolean should be set to false.

Start and stop service broadcast

Indeed, in order to know if the service has started correctly, you may want to listen to the broadcast the service will send when entirely started or stopped.

To get notified of this broadcast you have to register an intent filter named ***INTENT_UTMR_SERVICE_RUNNING.***

The broadcast will send a Boolean under the name of ***INTENT_SERVICE_RUNNING*** which will be set to true if the service has started correctly, false otherwise.

Adding the Activity name

The service can be still running even though the calling application has been stopped.

An icon on the notification bar indicates that the service is running. You can start again your application by pressing the service notification icon if you pass your activity classname to the intent when starting the service. The intent extra name is ***INTENT_START_SERVICE_CLASSNAME.***

3.4.2.2 The BVPreferences class

The UTMR Service uses parameters, which can be set using the public BVPreferences class.

As the BVPreferences class is recorded on the smartphone using the android shared preferences mechanism, BVPreferences is used in a static way.

This class will help you set the desired classifier and filter. See the JavaDoc and the UTMRTutoApp for more information.

Note:

- Only one classifier named V0401 is available in this release.
- 2 filters are available for this release: VOTE and HMM:
    - ‘VOTE’ is a majority vote over the last p=10 raw classification results. This is the default value.
    - ‘HMM’ is a post classification technique using a discrete hidden markov model (DHMM). This option has not been yet tested and should not be use.

See the UTMRTutoApp for more information about how to set classifier and filter.

3.4.2.3 Communication with the service

Start/stop broadcast

See chapter about the starting and stopping of the service.

Classification Result

For the classifier named V0401 (the only one available in this release), there are 8 classes: 'still', 'walk', 'run', 'bike', 'road', 'rail', 'plane' and 'undefined'.

Road regroups transportation modes like car and bus.

Rail regroups transportation modes like tramway, train, subway.

The class 'undefined' is returned when, for example, some sensor data are missing (e.g., the accelerometer or the magnetometer is not running).

UTMR Service broadcast every classification result that is computed through the INTENT_CHANGE_RESULT broadcast.

This broadcast contains the following information:

(long) ***INTENT_CHANGE_RESULT_NAME_START_TIME*** **:** is the UTC time at which the service was started in msec.
(long) ***INTENT_CHANGE_RESULT_NAME_RESULT_TIME*****:** is the time at which the classification was computed
(String [])***INTENT_CHANGE_RESULT_NAME_MODE_TYPE*****:** is the array of the classes the model uses.

The classification score is the posterior probability of being in one the different classes (8 in our case). It is a vector whose sum equal to 1 (probability). The raw classification result (see function below) is the maximum of these values.

(int) ***INTENT_CHANGE_RESULT_RESULT_VALUE*****:** the index of the winning class before the filtering.
(int) ***INTENT_CHANGE_RESULT_FILTER_VALUE*** **:** the index of the winning class after the filtering.

You can check the associated JavaDoc about for more information about those constants.

3.4.3 User's Stress Level (USL) Library Integration

This library, by using data provided by the Empatica E4 watch (ppg, acceleration, EDA, skin temperature), is able to estimate the stress level of the user. StressLevelService is started via the Andorid startService command. Since the initialization may take some time or fail if the Empatica watch is not responding, intermediate initialization status is sent via the Android Broadcast mechanism. This manual is a concrete guide for integrating and using the RecoStressLib correctly.

RecoStressLib is a java/android library, which aims at recognizing user stress with the help of the Empatica E4 watch.

3.4.3.1 Application manifest file

Service declaration

The usage of the service has to be declared in the main application manifest file.

```
<service android:name=" bv.leti.cea.recostresslib.StressDetectionService" />
```

Required Permissions

The main android application should have at least the following permissions:

- Read and write into external storage: recognition models have to be copied and read on the smartphone memory.
- Access coarse or fine location. This is for BLE use.
- Data service since the phone needs to check for connection permission via a web service to the E4 watch, having an internet connection is mandatory.

Note: not having granted those permissions may cause the application to crash.

The following lines have to be added to the manifest file :

```
<uses-permission android:name="android.permission.BLUETOOTH" /> <uses-
permission android:name="android.permission.BLUETOOTH_ADMIN" /> <uses-
permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" /> <uses-
permission android:name="android.permission.READ_EXTERNAL_STORAGE"/> <uses-
permission android:name="android.permission.INTERNET" /> <uses-permission
android:name="android.permission.ACCESS_COARSE_LOCATION" /> <uses-permission
android:name="android.permission.ACCESS_FINE_LOCATION" /> <uses-permission
android:name="android.permission.WAKE_LOCK" /> <uses-permission
android:name="android.permission.ACCESS_NETWORK_STATE" />
```

3.4.3.2 Project integration

This part explains how to integrate the lib in Android Studio. The “ecostresslib-release.aar” needs to be copied in the project structure under the “libs” folder. This folder is a child of the “app” folder and sibling of the “src” folder. The following lines need to be added to the application gradle file:

```
repositories
{
        flatDir
        {
                dirs 'libs'
        }
}
dependencies
{
        compile(name: 'recostresslib-release', ext:'aar')
}
```

The USL library uses third party libraries, which need to be added in the project in the application gradle file:

```
dependencies {
compile fileTree(dir: 'libs', include: ['*.jar'])
compile 'com.android.support:appcompat-v7:YOUR_LOCAL_VERSION'
compile(name:'recostresslib-release', ext:'aar')
compile 'nz.ac.waikato.cms.weka:weka-stable:3.6.13'
compile 'com.google.guava:guava:16.0.1'
compile 'com.squareup.okhttp:okhttp:2.5.0'
compile 'com.empatica.empalink:empalink:2.1@aar'
}
```

For an example of a complete app gradle file, please check the file distributed with StressLibTutoApp sample project. The minimal Android SDK version is 19 (Android4.4).

3.4.4 USL Library Usage

3.4.4.1 Start/ Stop USL Service

This library is designed as an android service. It should be started with the “startService” command.

```
import bv.leti.cea.recostresslib.StressDetectionService;
final Intent serviceIntent=new Intent(this,StressDetectionService.class);
//add Intent extras here
startService(serviceIntent);
```

The service tries to copy recognition model on the smartphone memory, then loads the model If something goes wrong in the copying / loading files process, the service stops itself.

Raw measure as well as classification results and features are logged into a file under the CEA/BV/Stress/Data folder.

The service is started with the "Start_sticky" flag which means that if something stops for any reason other than it was asked to by the user, the service will try to start again.

To stop the service you should use the command stopService :

```
Intent intent = new Intent(Context c , StressDetectionService.class);
stopService(intent);
```

The ***STRESS_SERVICE_STATUS*** broadcast should warn you if the service has stopped correctly or not. The Boolean should be set to false.

Empatica developer API Key

You need to send your Empatica API KEY number in order for Empatica to check if you are authorized to connect to the available E4 watches. API_KEY is sent via Intent Extra before starting the service.

```
import bv.leti.cea.recostresslib.StressServiceConstants; import
bv.leti.cea.recostresslib.StressDetectionService;

Intent serviceIntent=new Intent(Context c , StressDetectionService.class);
serviceIntent.putExtra(StressServiceConstants.EMPATICA_API_KEY, MY_API_KEY);
startService(serviceIntent);
```

You can get your Empatica API_KEY on the Empatica web site, after purchasing an Empatica E4 watch and becoming a developer[1]. See more information at[2].

[1] http://developer.empatica.com/ (Libraries for Android and iOS are available)
[2] https://www.empatica.com

3.4.4.2 Connecting to the Empatica E4 Device

Right after the StressDetectionService starts, it will be looking for an E4 watch to connect to. It is required that the user presses its E4 button at this moment.

The Empatica library checks if the user, identified by the API_KEY, is allowed, or not, to connect to the scanned Empatica device via a web service. If no Internet connection is available, this connection will fail and throw an Exception, sometimes causing the service to crash.

If the E4 was already started, it is possible that the connection process does not success, especially if the E4 was already working in logging mode. If this occurs, user should stop the watch and the service and start the process over again.

3.4.4.3 Communication with the service

In order to know if the service has started correctly, you need to listen the broadcast the service will send when entirely started.

The broadcast Action Name is ***STRESS_SERVICE_STATUS***.

It has 4 parameters, which are all Booleans:

- ***STRESS_SERVICE_IS_RUNNING***: is set to true if the service is running.
- ***STRESS_CLASSIFIER_IS_LOADED***: is set to true if the classifier s loaded and ready to be used.
- ***STRESS_SENSOR_IS_CONNECTED***: is set to true if the E4 sensor is connected
- ***STRESS_SERVICE_IS_PROCESSING***: is set to true if the bufferization and classification process has started.

This broadcast is sent every time one of this parameters value changes.

Classification Result broadcast

Stress Detection Service broadcasts every classification result that is computed through the *STRESS_SERVICE_RESULT* intent.

The parameter called ***STRESS_SERVICE_RESULT_NAME_RESULT*** is float between 0 and 1 where 0 is the minimal stress note and 1 the maximum.

More parameters are sent with this broadcast, you can find more information about them in the JavaDoc and the DemoStressTutoApp sample project.

IBI Measure received broadcast

See the ***STRESS_SERVICE_MEASURE_IBI*** constant in the JavaDoc for information about how to use this broadcast. IBI stands for Inter Beat Interval and is the base information on which heart rate can be computed by dividing 60 by this value.

EDA Measure received broadcast

See the ***STRESS_SERVICE_MEASURE_EDA*** constant in the Javadoc. EDA stands for electro dermic answer. It is also known as Galvanic Skin Response.

Binder

You can also bind to the service to get its current status via the Android binding process. The binder class name is StressDetectionService.SDServiceBinder.

The service gives access to 3 functions to know the service status:

```
Boolean getClassifierStatus()
Boolean getProcessStatus()
Boolean getSensorStatus()
```

3.5 Car Pooling

3.5.1 Description

The APIs exposed by the Car Pooling service allow managing the interaction between different users that want to offer and search rides in Europe. The service allows offering a ride, searching a ride in a specific area satisfying specific needs and booking a ride in order to reserve the requested number of seats. Finally, users can delete the requests and ride offers at every moment.

3.5.2 API

Add a new ride	
URL	http://82.223.67.189/carpoolingbe/OfferRide
HTTP Request Type	POST
Input Data:	Ride object (JSON)

Output Data:	transferID (INTEGER)
Communication protocol	HTTP/JSON

Get rides offered by the user	
URL	http://82.223.67.189/carpoolingbe/OfferRide
HTTP Request Type	GET
Input Data:	userID (INTEGER)
Output Data:	Rides offered by the user (JSON)
Communication protocol	HTTP/JSON

Get rides offered in a specific area	
URL	http://82.223.67.189/carpoolingbe/OfferRide
HTTP Request Type	GET
Input Data:	latStart/lonStart/latEnd/lonEnd (DOUBLE)
Output Data:	Rides offered in a specific area (JSON)
Communication protocol	HTTP/JSON

Delete a ride offered by the user	
URL	http://82.223.67.189/carpoolingbe/OfferRide
HTTP Request Type	DELETE
Input Data:	transferID (INTEGER)
Output Data:	ack (BOOLEAN)

Communication protocol	HTTP/JSON

Post a search ride request	
URL	http://82.223.67.189/carpoolingbe/SearchRide
HTTP Request Type	POST
Input Data:	Ride object (JSON)
Output Data:	List of solution objects (JSON)
Communication protocol	HTTP/JSON

Get solutions for a specific user request	
URL	http://82.223.67.189/carpoolingbe/SearchRide
HTTP Request Type	GET
Input Data:	userID/transferID (INTEGERS)
Output Data:	List of solution objects (JSON)
Communication protocol	HTTP/JSON

Book a ride	
URL	http://82.223.67.189/carpoolingbe/BookRide
HTTP Request Type	POST
Input Data:	solutionID (INTEGER)
Output Data:	poolID (INTEGER)
Communication protocol	HTTP/JSON

Get booked rides	
URL	http://82.223.67.189/carpoolingbe/BookRide
HTTP Request Type	GET
Input Data:	userID (INTEGER)
Output Data:	List of solution objects (JSON)
Communication protocol	HTTP/JSON

Delete ride reservation	
URL	http://82.223.67.189/carpoolingbe/BookRide
HTTP Request Type	DELETE
Input Data:	userID /solutionID (INTEGERS)
Output Data:	ack (BOOLEAN)
Communication protocol	HTTP/JSON

3.5.3 Data formats

Ride object: JSON schema

```
{
      "driver_id" : "Integer representing the user identifier",
      "pool_id" : "Integer value representing the identifier of the pool. If
the ride is not aggregated yet, the default value is -1",
      "dep_addr" : "Text representing the departure address",
      "arr_addr" : "Text representing the destination address",
      "dep_gps" : {
            "latitude" : "Double precision value representing the latitude of
the departure address",
            "longitude" : "Double precision value representing the longitudeof
the departure address"
      },
      "arr_gps" : {
            "latitude" : "Double precision value representing the latitude of
the destination address",
```

```
            "longitude" : "Double precision value representing the longitudeof
the destination address"
      },
      "dep_time" : "Timestamp representing the departure time",
      "occupied_seats" : "Integer representing the number of available seats if
      user_role='driver' or the number of requested                seats if
      user_role='passenger' ",
      "available_seats" : "Integer representing the number of available seats
      if user_role='driver' or the number of requested seats if
      user_role='passenger' ",
      "special_needs" : {
            "animal" : "Boolean value representing the request/availability
            (accordingly with the user_role) to get an animal on the vehicle",
            "handicap" : "Boolean value representing the request/availability
            (accordingly with the user_role) to accomodate people with handicap
            on the vehicle",
            "smoke" : "Boolean value representing the request/availability
            (accordingly with the user_role) to accomodate smokers on the
            vehicle",
            "luggage" : "Boolean value representing the request/availability
            (accordingly with the user_role) to get a luggage on the vehicle"
      },
      "status" : "Enum value representing the status of the ride. Accepted
values are: 'planned', 'ongoing','closed' ",
      "total_distance" : "dist",
      "total_duration" : "dur",
      "total_cost" : "cost",
      "passengers" : [{
                  "user_id" : "user_id",
                  "transfer_id" : "t",
                  "quantity" : 1,
                  "from" : "f",
                  "to" : "t"
            }, {
        1.1.1.1.              "user_id" : "user_id",

                  "transfer_id" : "t",
                  "quantity" : 1,
                  "from" : "f",
                  "to" : "t"
            }
      ],
      "path" : [{
                  "latitude" : "Double precision value representing the
                  latitude of the first point of the path crossed by the
                  driver",
                  "longitude" : "Double precision value representing the
                  longitude of the first point of the path crossed by the
                  driver",
                  "touchTime" : "Timestamp representing the time when it is
                  supposed that the node is crossed by the driver"
            }, {
                  "latitude" : "Double precision value representing the
                  latitude of the second point of the path crossed by the
                  driver",
                  "longitude" : "Double precision value representing the
                  longitude of the second point of the path crossed by the
                  driver",
```

```
                    "touchTime" : "Timestamp representing the time when it is
                    supposed that the node is crossed by the driver"
            }, {
                    "latitude" : "Double precision value representing the
                    latitude of the last point of the path crossed by the
                    driver",
                    "longitude" : "Double precision value representing the
                    longitude of the last point of the path crossed by the
                    driver",
                    "touchTime" : "Timestamp representing the time when it is
                    supposed that the node is crossed by the driver"
            }
      ]
}
```

Ride object: examples

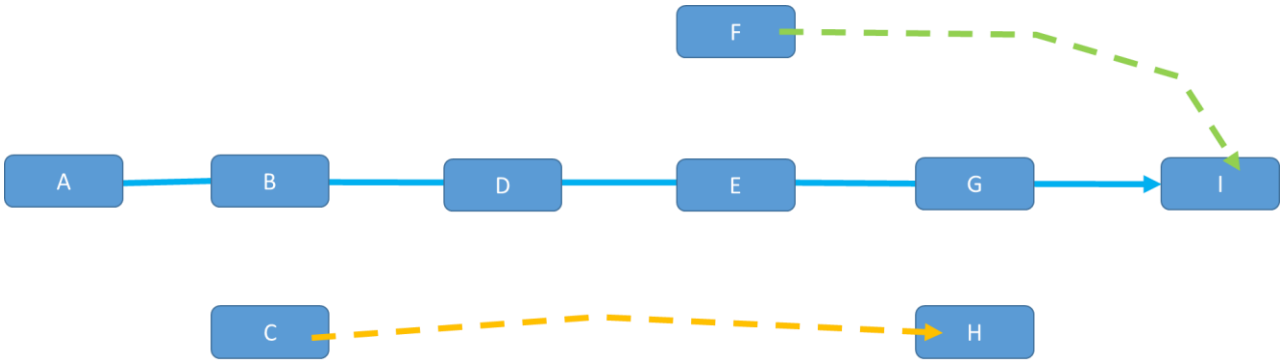

Figure 7 Ride examples: Blue line represents the path of a driver, yellow and green dashed line represent passenger's requests

The following JSON represents the ride object posted by the driver (blue line in Figure 7) to offer the ride:

```
{
  "user_id": 89,
  "pool_id": 0,
  "user_role": "driver"
  "dep_addr": "A",
  "arr_addr": "I",
  "dep_gps": {
    "latitude": latitude(A),
    "longitude": longitude(A)
  },
  "arr_gps": {
"latitude": latitude(I),
```

```
    "longitude": longitude(I)


  },
  "dep_time": 10,
  "seats": 3,
  "special_needs": {
    "animal": true,
    "handicap": true,
    "smoke": true,
    "luggage": true
  },
  "status": "planned",
  "cost": 0.35,
  "det_range": 300.0,
  "ride_details": "Tesla model S",
  "path": [
    {
"latitude": latitude(A),
    "longitude": longitude(A)
      "touchTime": 10
    },
    {
"latitude": latitude(B),
    "longitude": longitude(B)
      "touchTime": 20
    },
    {
    "latitude": latitude(D),
    "longitude": longitude(D)
      "touchTime": 30
    },
    {
    "latitude": latitude(E),
    "longitude": longitude(E)
      "touchTime": 40
    },
    {
    "latitude": latitude(G),
    "longitude": longitude(G)
      "touchTime": 50
    },
    {
    "latitude": latitude(I),
    "longitude": longitude(I)
      "touchTime": 60
    }
  ]
}
```


The following two JSON objects represent the ride objects posted by the passenger to add the ride request (yellow and green line in Figure 7).

```
{
  "user_id": 12,
  "pool_id": 0,
  "user_role": "passenger"
  "dep_addr": "C",
  "arr_addr": "H",
  "dep_gps": {
    "latitude": latitude(C),
    "longitude": longitude(C)
  },
  "arr_gps": {
"latitude": latitude(H),
    "longitude": longitude(H)
  },
  "dep_time": 25,
  "seats": 2,
  "special_needs": {
    "animal": false,
    "handicap": false,
    "smoke": false,
    "luggage": true
  },
  "status": "planned",
  "cost": 0.40,
  "det_range": 400.0,
  "ride_details": "from station to home",
  "path": [
  ]
}


{
  "user_id": 34,
  "pool_id": 0,
  "user_role": "passenger"
  "dep_addr": "F",
  "arr_addr": "I",
  "dep_gps": {
    "latitude": latitude(F),
    "longitude": longitude(F)
  },
  "arr_gps": {
"latitude": latitude(I),
    "longitude": longitude(I)


  },
  "dep_time": 45,
  "seats": 1,
  "special_needs": {
    "animal": false,
    "handicap": false,
    "smoke": false,
    "luggage": false
  },
  "status": "planned",
```

```
  "cost": 0.40,
  "det_range": 200.0,
  "ride_details": "from home to work",
  "path": [
  ]
}
```

Solution object: JSON schema

```
{
 "driver_id":"Integer representing the user identifier",
 "pool_id":"Integer value representing the identifier of the pool. If the ride
is not aggregated yet, the default value is -1",
 "dep_addr":"Text representing the departure address",
 "arr_addr":"Text representing the destination address",
 "dep_gps": {
   "latitude":"Double precision value representing the latitude of the
departure address",
   "longitude":"Double precision value representing the longitudeof the
departure address"
  },
 "arr_gps": {
   "latitude":"Double precision value representing the latitude of the
destination address",
   "longitude":"Double precision value representing the longitudeof the
destination address"
  },
 "dep_time":"Timestamp representing the departure time",
 "occupied_seats":"Integer representing the number of available seats if
user_role='driver' or the number of requested seats if user_role='passenger'",
 "available_seats":"Integer representing the number of available seats if
user_role='driver' or the number of requested seats if user_role='passenger'",
 "special_needs": {
   "animal":"Boolean value representing the request/availability (accordingly
with the user_role) to get an animal on the vehicle",
   "handicap":"Boolean value representing the request/availability (accordingly
with the user_role) to accomodate people with handicap on the vehicle",
   "smoke":"Boolean value representing the request/availability (accordingly
with the user_role) to accomodate smokers on the vehicle",
   "luggage":"Boolean value representing the request/availability (accordingly
with the user_role) to get a luggage on the vehicle"
  },
 "status":"Enum value representing the status of the ride. Accepted values are:
'planned', 'ongoing','closed'",
 "total_distance":"Double precision representing the total distance of the path
in meters",
 "total_duration":"Double precision representing the total duration of the path
in seconds",
 "total_cost":"Double precision representing the total cost of the path
(euros/scores)",
 "passengers": [
    {
     "user_id":"Integer representing the user identifier",
     "transfer_id":"Integer value representing the ride identifier",
```

```
     "quantity":"integer value representing the number of seats occupied by the
ride indentifyied by transfer_id",
     "from":"Text value representing the departure address",
     "to":"Text value representing the arrival address",
     "routes": [
        {
         "geometryGeoJson": {
           "type":"Feature",
           "geometry": {
             "type":"LineString",
             "coordinates": [
                [
                 "Double precision value representing the longitude of the
first node crossed by the user_id",
                 "Double precision value representing the latitudeof the first
node crossed by the user_id",
                 "timestamp representing the time when the user_id visits the
first node"
                ],
                [
                 "Double precision value representing the longitude of the
second node crossed by the user_id",
                 "Double precision value representing the latitudeof the second
node crossed by the user_id",
                 "timestamp representing the time when the user_id visits the
second node"
                ],
                [
                 "Double precision value representing the longitude of the last
node crossed by the user_id",
                 "Double precision value representing the latitudeof the last
node crossed by the user_id",
                 "timestamp representing the time when the user_id visits the
last node"
                ]
              ]
            },
           "durationSeconds":"Double precision representing the
duration(seconds)of the path covered by the user_id",
           "lengthMeters":"Double precision representing the distance(meters)of
the path covered by the user_id",
           "cost":"Double precision value representing the cost of the path
covered by the user_id",
           "segments": [
              {
               "nr": 1,
               "durationSeconds":"Double precision representing the
duration(seconds)of the segment covered by the user_id",
               "lengthMeters":"Double precision representing the
distance(meters)of the segment covered by the user_id",
               "cost":"Double precision value representing the cost of the
segment covered by the user_id",
               "modeOfTransport": {
                 "generalizedType":"CAR_POOLING"
                },
               "from": {
                 "coordinate": {
                   "type":"Feature",
```

```
                    "geometry": {
                      "type":"Point",
                      "coordinates": [
                        "Double precision value representing the longitude of
the departure address",
                        "Double precision value representing the latitude of the
departure address"
                      ]
                    }
                  }
                },
               "to": {
                 "coordinate": {
                   "type":"Feature",
                   "geometry": {
                     "type":"Point",
                     "coordinates": [
                       "Double precision value representing the longitude of
the arrival address",
                       "Double precision value representing the latitude of the
arrival address"
                     ]
                   }
                 }
               }
             }
           ]
         }
       }
     ]
   }
 ]
}
```

Solution object: example

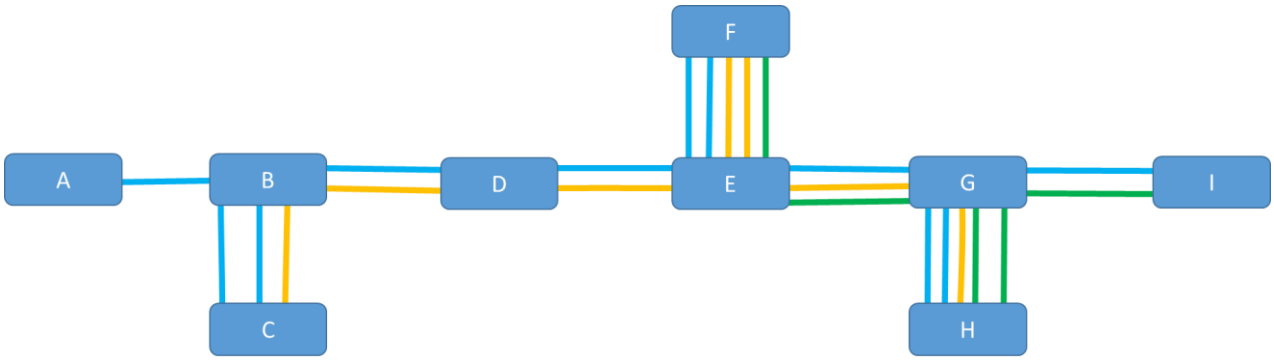

Figure 8 Solution object example: blue line represents the path of the driver that picks the first passenger up in node C (yellow line), then drives to node F to pick also the second passenger up (green line). Then the three users go to node H to leave the first passenger and finally the driver and the second passenger go to destination node I.

The JSON object able to represent the solution discussed in Figure 8 contains the information about the driver and three passenger entities, each one containing the information about the route that the user will share.

3.5.4 Usage

The APIs exposed by the Car Pooling service will allow BONVOYAGE to explore and add innovative travel alternatives that include car-pooling as a standard transport service. In particular, the application will use the POST service to attach a new Car Pooling ride offer to an itinerary (among the alternatives returned by the Orchestrator) that included a path by car, whenever the user is willing to do so. The user can also manage his/her ride offers deleting them and obviously search for a ride among the offers. When an offered ride matches with the user's preferences, the user can book the ride.

4 Internal Services at the Orchestrator Layer

This section presents the interfaces' details of the BONVOYAGE Orchestrator component, which can be used by external stakeholders to easily develop powerful inter-modal and multi-modal travel applications, by offering a routing service. The real-time intermodal routing service, i.e. the API exported by the Orchestrator to the Application Server, is presented in Section 4.1. It includes a description of the GetRoutes API and the SPROUTE data format, which is actually used for route requests and responses throughout the entire project.

4.1 Real-time intermodal routing service

4.1.1 Description

The routing service provides a route planning service for goods and persons. It implements a set of personalisation functionalities providing router recommendations optimised for the individual settings and preferences of the user. The Orchestrator handles routing request from the Application Server. The BONVOYAGE implements a distributed, decomposition approach for the routing request. A federated community of Soloists is used to calculate one or more personalised routes for the request. The details of the approach are described in *Deliverable 4.1 Design of the Intelligent Transport Functionality* and *Deliverable 5.1 Design of the adaptation functionality*. The data format for route requests and response is SPROUTE[3]. Adaptations and extensions have been applied to the format, in order to support the full functionality of the BONVOYAGE platform. These backwards backwards-compatible changes are described in detail in *Deliverable 5.1 Design of the adaptation functionality.*

The Orchestrator is implemented as a web service allowing HTTP POST requests. At input, the route request is sent as JSON in SPROUTE format. The User-ID is set within the request as HTTP-Header parameter. The response contains the route alternatives as JSON in SPROUTE format.

4.1.2 API

GETROUTES	
URL	http://bonvoyage.sintef.no/Routing/json/GetRoutes
Input Data:	Route alternatives (SPROUTE FORMAT) USER-ID (FIREBASE TOKEN)
Output Data:	Route alternatives (SPROUTE FORMAT)

[3] https://github.com/dts-ait/ariadne-json-route-format

Communication protocol	POST HTTP/JSON

4.1.3 Data formats

The SPROUTE data format is used by the BONVOYAGE routing service. The base format is specified at[Error! Bookmark not defined.], while its adaptation and extensions for BONVOYAGE are s pecified in *Deliverable 5.1 Design of the adaptation functionality.* The following Figure 8 shows the structure and the concepts of the format.

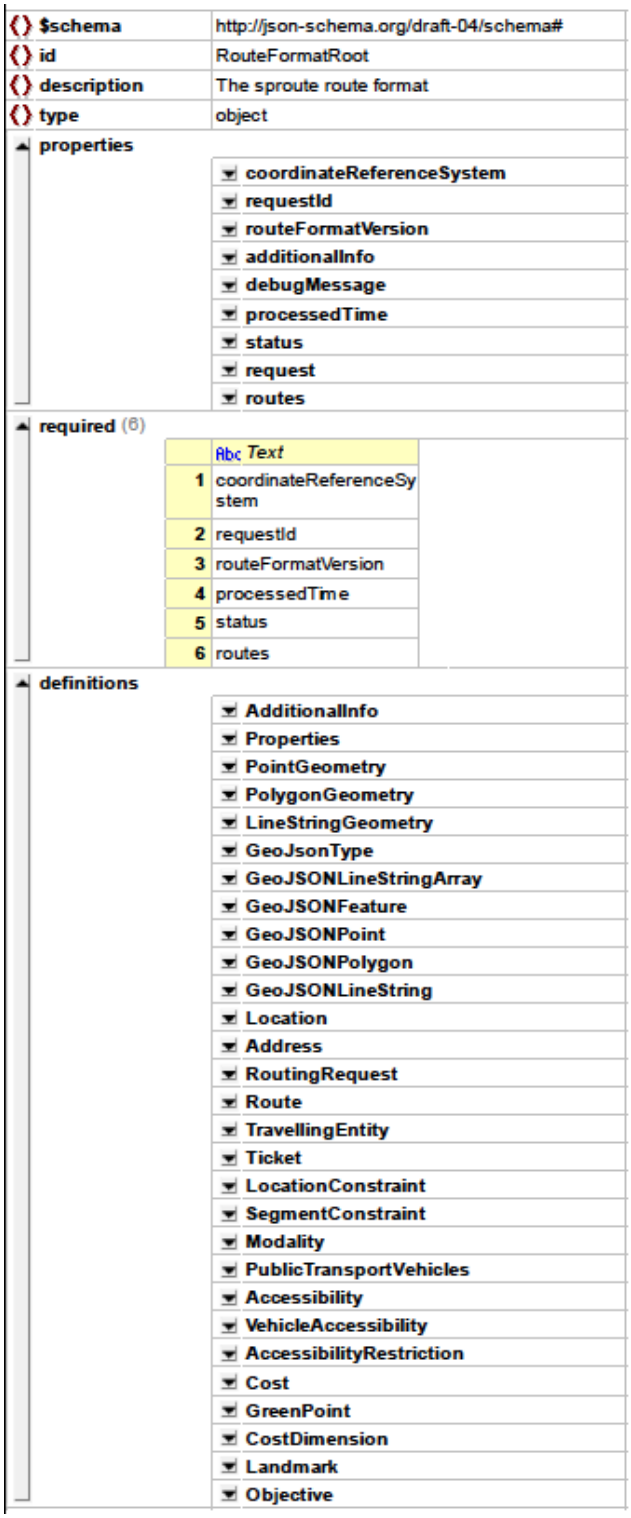

Figure 9 Structure and concepts of the SPROUTE format

The following example shows the JSON schema of the request specifying departure and target location (including optional “via points”). Note that, locations have to be defined using geographic coordinates. Therefore, an external geocoding service (or functionalities of the mobile operating system e.g. google services) has to be used in order to enable the users to enter addresses rather than only directly or indirectly specifying coordinates (e.g. current position, location picked from a map etc.)

Additionally, constrains for the route can be set such as time and mode of transport.

```
            "RoutingRequest": {
                  "javaType": "sproute.RoutingRequest",
                  "type": "object",
                  "properties": {
                        "departureTime": {
                              "type": "string"
                        },
                        "accessibilityRestrictions": {
                              "type": "array",
                              "items": {
                                    "$ref":
"#/definitions/AccessibilityRestriction"
                              }
                        },
                        "maximumPublicTransportRoutes": {
                              "type": "integer"
                        },
                        "acceptedDelayMinutes": {
                              "type": "integer"
                        },
                        "modesOfTransport": {
                              "type": "array",
                              "items": {
                                    "$ref": "#/definitions/Modality"
                              }
                        },
                        "language": {
                              "type": "string"
                        },
                        "optimizedFor": {
                              "type": "string"
                        },
                        "excludedPublicTransport": {
                              "type": "array",
                              "items": {
                                    "$ref":
"#/definitions/PublicTransportVehicles"
                              }
                        },
                        "arrivalTime": {
                              "type": "string"
                        },
                        "additionalInfo": {
                              "$ref": "#/definitions/AdditionalInfo"
```

```
                        },
                        "maximumTransfers": {
                              "type": "integer"
                        },
                        "from": {
                              "$ref": "#/definitions/LocationConstraint"
                        },
                        "via": {
                              "type": "array",
                              "items": {
                                    "$ref": "#/definitions/LocationConstraint"
                              }
                        },
                        "to": {
                              "$ref": "#/definitions/LocationConstraint"
                        },
                        "serviceId": {
                              "type": "string"
                        },
                        "privateVehicleLocations": {
                              "type": "object",
                              "additionalProperties": {
                                    "type": "array",
                                    "items": {
                                          "$ref": "#/definitions/Location"
                                    }
                              }
                        },
                        "travellingEntities": {
                              "type": "array",
                              "items": {
                                    "$ref": "#/definitions/TravellingEntity"
                              }
                        },
                        "currentRoute": {
                              "$ref": "#/definitions/Route"
                        },
                        "segmentConstraints": {
                              "description": "Constraints applying to segments.
If segment constraints are provided, the number of segment constraints must
always be the amount of via points + 1",
                              "type": "array",
                              "items": {
                                    "$ref": "#/definitions/SegmentConstraint"
                              }
                        }
                  },
                  "required": [
                        "modesOfTransport",
                        "from",
                        "to",
                        "serviceId"
                  ]
            }
```

4.1.4 Usage

The following example shows the excerpt of a route response. It shows the first to legs of the route with the id “4”. The route begins with a journey on foot for the starting point “CEA” to the train station “Gare de Grenoble”. From there a train is taken to “Gare de Lyon Saint-Exupery”. The route contains information about duration, distance and the mode of transport.


```
…
 "id": "4",
      "distanceMeters": 1101606,
      "segments": [
          {
              "accessibility": null,
              "additionalInfo": null,
              "alightingSeconds": 0,
              "endTime": "2016-12-05T15:05:00.0000000+01:00",
              "boardingSeconds": 0,
              "boundingBox": null,
              "costs": null,
              "startTime": "2016-12-05T14:53:35.5920000+01:00",
              "durationSeconds": 684,
              "from": {
                  "additionalInfo": null,
                  "address": {
                      "streetName" : "CEA"
            },
                  "coordinate": {
                      "geometry": {
                          "coordinates": [
                              5.7080675287106395,
                              45.19517455763722
                          ],
                          "type": "Point"
                      },
                      "properties": {},
                      "type": "Feature"
                  }
              },
              "geometryEncodedPolyLine":
"ydzrGmzya@N}D??fAmB??lAuB??DK??AC??i@o@??c@Y??BU??BS??@E??B_@??@E???E??@K??@K?
??I???E??BY??FK??JG??nCaA??XK??XK??~@]??JE??JE??lAc@??JE??`Bo@??tAg@??CQ??OmA??
?M??FK??h@Q??NG??FC??PG??AW???A??JE???C??C@???C??Ge@??AA??DC??@A??JE??D???|By@?
?bA_@??D@??DC??BA??DG??HC??AE",
              "geometryGeoJson": null,
              "geometryGeoJsonEdges": null,
              "intermediateStops": null,
              "distanceMeters": 953,
              "modeOfTransport": {
                  "accessibility": null,
                  "additionalInfo": null,
                  "detailedType": "FOOT",
                  "generalizedType": "FOOT",
                  "id": null,
```

```
            "operator": null,
            "service": null,
            "sharingType": ""
        },
        "navigationInstructions": null,
        "nr": 1,
        "to": {
            "additionalInfo": null,
            "address": {
                "streetName" : "Gare de Grenoble"
      },
            "coordinate": {
                "geometry": {
                    "coordinates": [
                        5.71462,
                        45.19011
                    ],
                    "type": "Point"
                },
                "properties": {},
                "type": "Feature"
            }
        }
    },
    {
        "accessibility": null,
        "additionalInfo": null,
        "alightingSeconds": 0,
        "endTime": "2016-12-05T15:31:00.0000000+01:00",
        "boardingSeconds": 0,
        "boundingBox": null,
        "costs": null,
        "startTime": "2016-12-05T15:07:00.0000000+01:00",
        "durationSeconds": 1440,
        "from": {
            "additionalInfo": null,
            "address": {
                "streetName" : "Gare de Grenoble"
      },
            "coordinate": {
                "geometry": {
                    "coordinates": [
                        5.71462,
                        45.19011
                    ],
                    "type": "Point"
                },
                "properties": {},
                "type": "Feature"
            }
        },
        "geometryEncodedPolyLine": "eeyrGkc{a@cufBjw{B",
        "geometryGeoJson": null,
        "geometryGeoJsonEdges": null,
        "intermediateStops": null,
        "distanceMeters": 77316,
        "modeOfTransport": {
            "accessibility": null,
```

```
                        "additionalInfo": null,
                        "detailedType": "RAILWAY",
                        "generalizedType": "PUBLIC_TRANSPORT",
                        "id": "MISSING_ID",
                        "operator": null,
                        "service": null,
                        "sharingType": "RIDE_SHARING"
                    },
                    "navigationInstructions": null,
                    "nr": 2,
                    "to": {
                        "additionalInfo": null,
                        "address": {
                            "streetName" : "Gare de Lyon Saint-Exupery"
                  },
                        "coordinate": {
                            "geometry": {
                                "coordinates": [
                                    5.075841,
                                    45.721006
                                ],
                                "type": "Point"
                            },
                            "properties": {},
                            "type": "Feature"
                        }
                    }
                },
…
```

5 Internal Services at the Infrastructure Layer

This section presents the interfaces' details of the BONVOYAGE Infrastructure Layer components that provide public APIs to external stakeholders. The Soloist routing service is introduced in Section 5.1. The Publish and Subscribe services based on Internames and an example usage both in BONVOYAGE as well as in other contexts is discussed in Section 5.2.

The functionality provided by OGB is described in Section 5.3 (when using the provided JAVA interface). OGB also provides HTTP interfaces as described in Section 5.3.3.

5.1 Soloist routing service

Between Orchestrator and Soloists there exists a GetRoutes API exactly mirroring the functionality of the GetRoutes we presented in 4.1, but mediated through an extended SPROUTE, due to the fact that we need to specify in more details the start and end constraints of the sub-route calculated by an individual Soloist. We also allow the Orchestrator to request "many-to-many" routes from a Soloist as well as allowing the 'from' and 'to' to be specified as polygons and not only coordinates. These extensions are on-going work and will be documented in upcoming deliverable D5.2.

5.2 Publish – Subscribe

5.2.1 Description

The BONVOYAGE Communication System was designed in order to support both request-response and publish-subscribe communication schemes. These services are offered to users (for instance soloists willing to receive updates about real-time travel data) through simplified APIs. This section focuses on publish-subscribe services and provides a technical description of the API.

Figure 10 depicts our reference scenario. It integrates a *Data Producer* (for instance, a bus operator in Rome), a *Data Consumer* (which can be either private, e.g. a single user or tour operator, or a soloist software used for journey planning) and the communication network based on Internames [see Deliverable D3.1]. In general, the publish-subscribe communication scheme (also referred to as *topic-based* communication scheme) can be used for asynchronous data dissemination. It is extremely useful whenever it is difficult to synchronize the generation of requests at the Data Consumer side, mainly when dealing with *real-time contents*. In these contexts, in fact, synchronous interactions may generate issues. For instance, the user may lose

intermediate updates of a given data or may retrieve the same version of the content. Another major problem is related to the possible increment of number of messages exchanged in the network. In fact, the Data Consumer may request the same copy of the content even if it the information did not change. When the asynchronous approach is used, instead, as a first step the Data Consumer issues a subscription request for a given content of interest. Then, every time a new content is generated, the Data Producer has to notify the presence of a new content to all subscribed Data Consumers, allowing them to retrieve updated version of real-time contents.

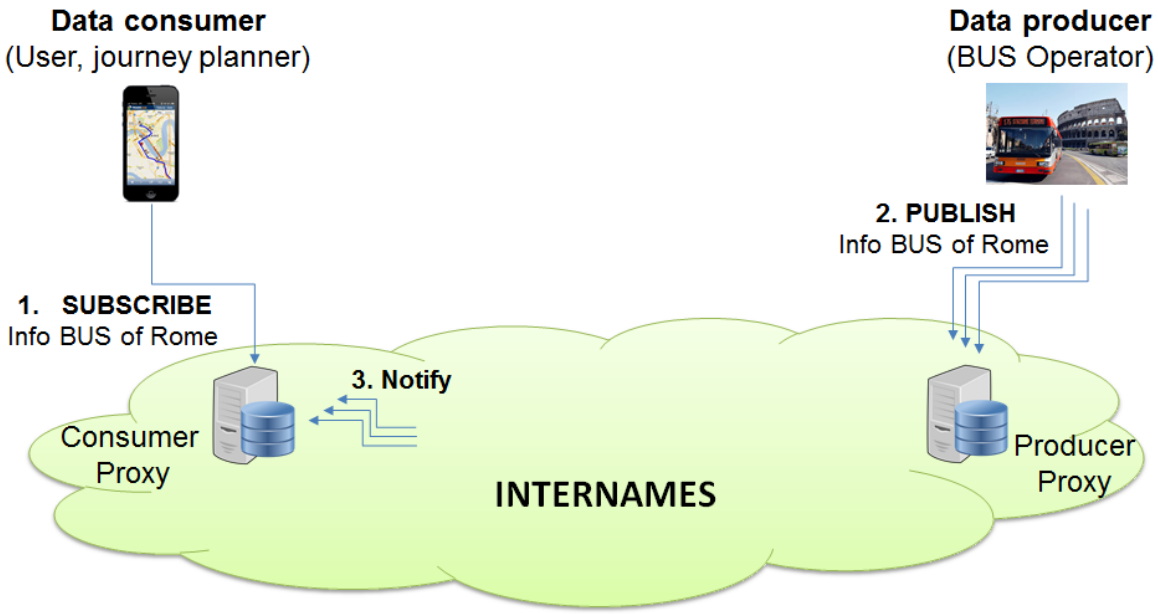

Figure 10 Publish-subscribe reference architecture

In Internames each piece of content is identified through a unique name, for instance "Info_BUS_of_Rome" in the above example. When using the publish-subscribe model implemented by our BONVOYAGE Communication System, content names serve the purpose of "topic channels" for the pub/sub architecture and they identify the topic under which updates are to be regularly published.

Data Consumer and Data Producer do not interact directly with Internames. They, instead, establish a connection with dedicated nodes (Consumer Proxy and Producer Proxy, respectively) that act as an interface between the users of the BONVOYAGE system and the underlying network infrastructure.

Data Consumer and Data Producer are implemented by importing a Java library that offers the relevant APIs. Publishing APIs at the Data Produces side are: **`publish_init`** and

publish_update. Specifically, **publish_init** is used to publish the initial content of a generally dynamic publication. It accepts the name of the publication and the publication itself, returning true is the publishing was well done, or false if something went wrong. **publish_update** is used to publish an update of existing publication, it accepts the name of the publication and the new publication, returning true if the publishing was well done, or false if something went wrong. It is important to remark that data are locally stored at the producer side (this operation is transparent for the end user, as it is implemented by the APIs). Furthermore, both APIs must be called sequentially every time the content is generated, to properly trigger the notification process. Therefore, when the websocket connection between Data Producer and Producer Proxy is closed, data will be no more available in the entire system. However, if in-network caching is enabled in the intermediate border routers, a local copy of data can be temporarily available for Data Consumers, even if the Data Producer is not connected.

The Subscriber uses the **subscribe** API of the library at the Data Consumer side.

Now, by focusing the attention on the whole Data Consumer, Data Producer, Consumer Proxy, and Producer Proxy architecture, the following depicts the typical flow of operations:

Data Producer initiates a websocket connection with the Producer Proxy;

- Data Producer produces a publication message (via **publish_init** or **pubish_update**) announcing the availability of real-time data to the Producer Proxy and becomes a source for these data, till the websocket connection is closed. According to what previously described, announced data refer to one or more name/topic. For each name/topic, two contents must be available: initial contents (namely INIT) and updated contents (namely UPDATE). Initial contents are those that need to be delivered to the consumer as soon it joins the platform. Update contents include, instead, any modification to data currently produced by the Data Producer.
- Once the Producer Proxy receives a publication message, it announces the availability of the (new version of the) content to the BONVOYAGE Communication System;
- Data Consumer initiates a WebSocket connection with the Consumer Proxy;
- Data Consumer communicates to the Consumer Proxy its interest to collect real-time data belonging to a set of names/topics. The list of names/topics was obtained by the Data Consumer somewhere else, for instance via discovery procedures done by querying the

OGB Discovery Service. It uses the **subscribe** API, which receives as input the list of names/topics and triggers the subscription process developed by the Consumer Proxy.

- Once the Consumer Proxy receives the list of names sent by the consumer, it retrieves the corresponding initial contents and makes a subscription request to the BONVOYAGE Communication System (through which it will be able to receive future updates).

5.2.2 API

Publish-subscribe services are exposed to end users by means of three Java APIs, which are **`publish_init`**, **`publish_update`** and **`subscribe`**.

publish_init (name, content)	
End-point	it.telematics.isl.Producer Java class
Input Data:	**name:** the name of the content to be published **content:** the content to be published
Output Data:	**True** if content is successfully announced to the Producer Proxy
Communication protocol	Direct method call of a Java library

publish_update (name, content)	
End-point	it.telematics.isl.Producer Java class
Input Data:	**name:** the name of the content to be published **content:** the content to be published
Output Data:	**True** if content is successfully announced to the Producer Proxy
Communication protocol	Direct method call of a Java library

subscribe (namelist, handler)	
End-point	it.telematics.isl.Consumer Java class
Input Data:	**namelist:** List of names or a single name string. **handler:** Object of class that implements MessageHandler interface. Is used to retrieve matching contents
Output Data:	**True** if list of name or single name is successfully sent to the Consumer Proxy
Communication protocol	Direct method call of a Java library

5.2.3 Data formats

Data Consumers and Data Producers exchange JSON messages with proxies of the Internames infrastructure. A total of three fields compose each message (see Figure 11):

1. *Header*. It contains general information about the message (e.g., the name of the publication the message is carrying). This part of the message is mandatory in order to disambiguate communications and properly address the message to the user/consumer/application which asked for it;
2. *Type*. This field specifies the type of message (which can either contain logging information or data). It provides information about the usage of the retrieved data.
3. *Content*. In it, the real content of the message is communicated.

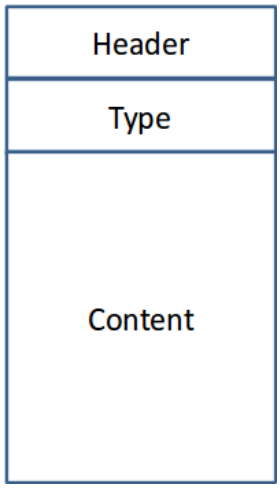

Figure 11 JSON message structure used in the BONVOYAGE Communication System

Usage of this format, together with the JSON data notation, represents an optimal solution for message exchange between network nodes to grant efficient asynchronous and decoupled communications. First, JSON data format lowers the overhead connected to processing activities, which are mandatory to elaborate XML files. Furthermore, messages can be of different nature and the core network does not discriminate between them: control information and data messages dispatched do not imply any differences in delivery throughout the network.

5.2.4 Usage

Publishing

An example code that depicts the behavior of the designed APIs is provided in what follows.

It is assumed that a producer generates contents under the name/topic bv/nametest. These example contents include a timestamp and a random number in the range [0-100]. The dedicated `ProducerExample` class is used for creating the Producer. Every time a new content is generated, the pair `<timestamp, random_number>` is stored within the updated content in a JSON object (see lines 6-10). The latest 3 generated contents, instead, are always stored within the initial content. To this end, every time a new content is generated, both INIT and UPDATE contents are generated and announced to the system through the `publish_init` and `publish_update` APIs. In the provided example, the updates and the related announcements are scheduled for periodic execution with one second pacing.

```
1: import it.telematics.isl.Producer.Producer;
2: public class ProducerExample {
3:    public static void main(String[] args) {
4:          String _defWebsocketServer =
"ws://telematics.poliba.it:8888/internames";
5:          new Producer(_defWebsocketServer);
6:          JsonArrayBuilder initArray = Json.createArrayBuilder();
7:                initArray.add(Json.createObjectBuilder()
8:                      .add("timestamp", System.currentTimeMillis())
9:                      .add("number", ThreadLocalRandom.current()
10:                           .nextInt(minNumber, maxNumber + 1)));
11:         JsonObject initContent = Json.createObjectBuilder()
12:         .add("data", initArray).build();
13:         try {
14:               Producer.publish_init("/nametest", initContent.toString());
15:                   Producer.publish_update("/nametest",
initContent.toString());
16:         } catch (IOException e1) {
17:               e1.printStackTrace();
18:         }
19:         executor = Executors.newScheduledThreadPool(2);
20:         executor.scheduleAtFixedRate(
21:               new updateContent("/nametest"),0, 1, TimeUnit.SECONDS);
```

Subscribing

We provide here a more realistic example of how a Consumer can be created for receiving real-time updates from the Norway NPRA server, which is a Producer of periodic DatexII information about roads in the whole Norway country. For a detailed explanation of this architecture, please see D5.1. Here we will focus on the more technical aspects of the API usage.

In the BONVOYAGE Communication System, we have designed an ad-hoc namespace where each piece of travel centric content is identified through a unique name. It is designed according to a hierarchical and geo-referenced structure [see also Deliverable D5.1], that is:

[NAME] = /bv/[coords]/GPS-ID/[std]/[provider]/[service]/[… other fields …]

where: (i) [coords] provides GPS coordinates of the geographical area which the content refers to, (ii) [std] field is related to the file format used by the Data Producer to deliver data (e.g., datexii), (iii) [provider] indicates who is providing the information, (iii) [service] field identifies the specific service the files is related to. We say that the name is geo-referenced because the [coords] field is filled with a selected geographical area. Starting from the GPS coordinates, in fact, a specific area of approx. 1 km x 1 km is selected. For example, with reference to coordinates (61.56, 8.43) the area is defined with latitude ranging from 61.56 to 61.57 and longitude ranging from 8.43 to 8.44. Thus a geographical area can be described by a set of tiles (see Figure 12), each one identified by a unique name.

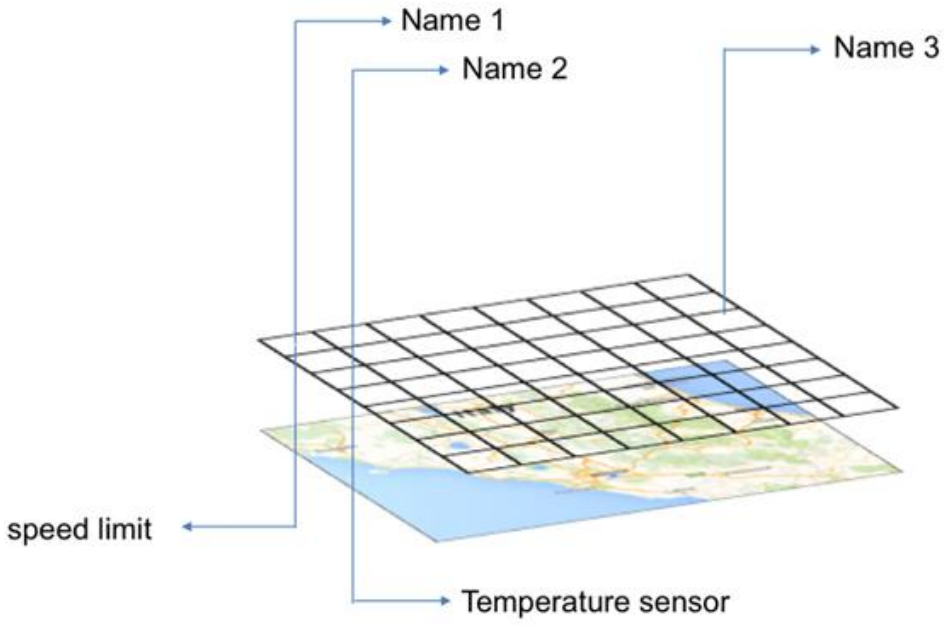

Figure 12 A set of tiles for a geographical area

It is assumed that the user wants to get DatexII contents within an area of interest identified by the two GPS points (expressed in decimal notation):

- Longitude 6, latitude 63,
- Longitude 12, latitude 58,

which identify the following area of interest reported in Figure 13. The Figure reports data coming from the Norway NPRA server, and specifically DatexII dynamic info regarding CCTV, road situations and weather. Each piece of dynamic info is shown in the map, and falls within the scope of a square tile [see Deliverable D5.1].

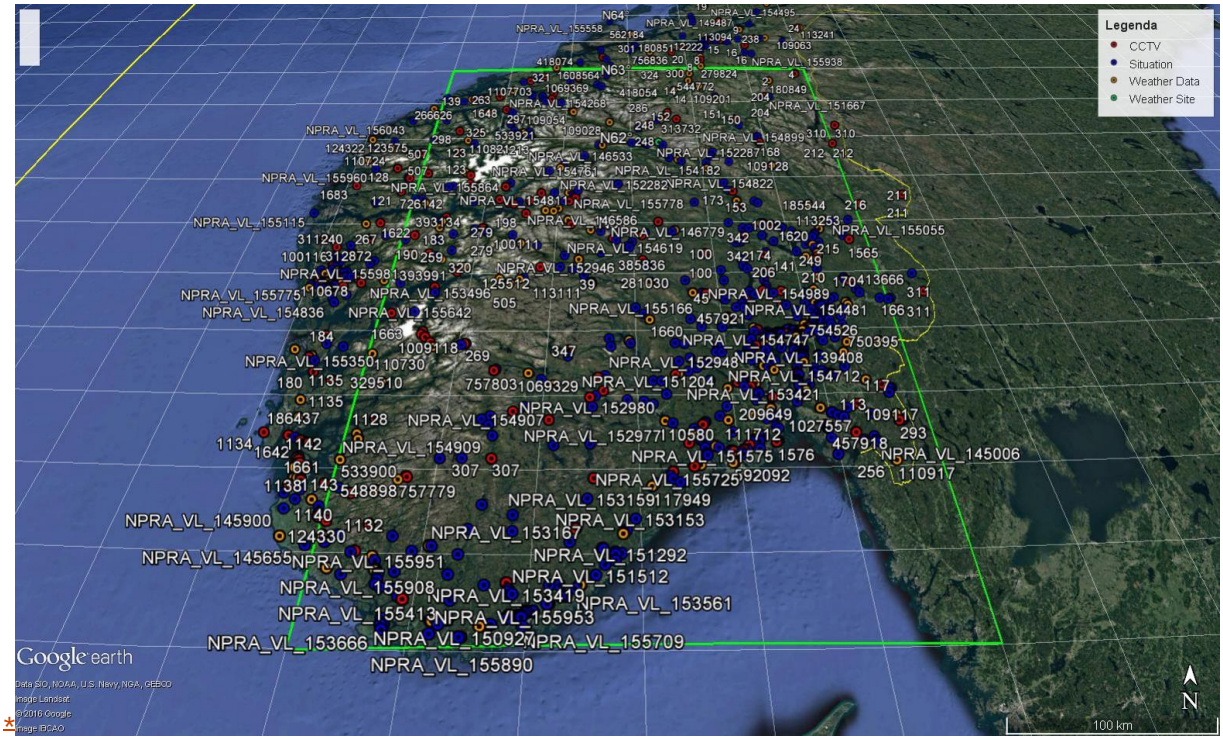

Figure 13 Data from NPRA DatexII server with area of interest

Prior to subscribing, the consumer probes the OGB Discovery Service to retrieve all the contents matching this area of interest. The OGB Discovery Service returns a list of names pointing to every tile that has had publications inside its area. A possible list of tiles' names is:

```
n2n://poliba_datexii/bv/06/63/00/GPS-ID/datexII/npra/cctv
n2n://poliba_datexii/bv/06/63/00/GPS-ID/datexII/npra/weathersite
n2n://poliba_datexii/bv/06/63/00/GPS-ID/datexII/npra/weatherdata
n2n://poliba_datexii/bv/06/63/00/GPS-ID/datexII/npra/situation
n2n://poliba_datexii/bv/06/63/10/GPS-ID/datexII/npra/cctv
……
……
n2n://poliba_datexii/bv/11/57/89/GPS-ID/datexII/npra/situation
n2n://poliba_datexii/bv/11/57/99/GPS-ID/datexII/npra/cctv
n2n://poliba_datexii/bv/11/57/99/GPS-ID/datexII/npra/weathersite
n2n://poliba_datexii/bv/11/57/99/GPS-ID/datexII/npra/weatherdata
n2n://poliba_datexii/bv/11/57/99/GPS-ID/datexII/npra/situation
```

At this step, the consumer sends the list of names to the Consumer Proxy by using **`subscribe`**, basically subscribing to one or more tiles, signaling it wants to receive each updated (for instance

CCTV) information that is going to be published within the scope of the specified tiles. It provides a callback to manage the incoming publications. The Consumer can then organize retrieved contents in a specific structure of folders. Each piece of content is stored in a single file; INIT and UPDATEs versions are stored in different files. The example folder structure is described below, in Figure 14.

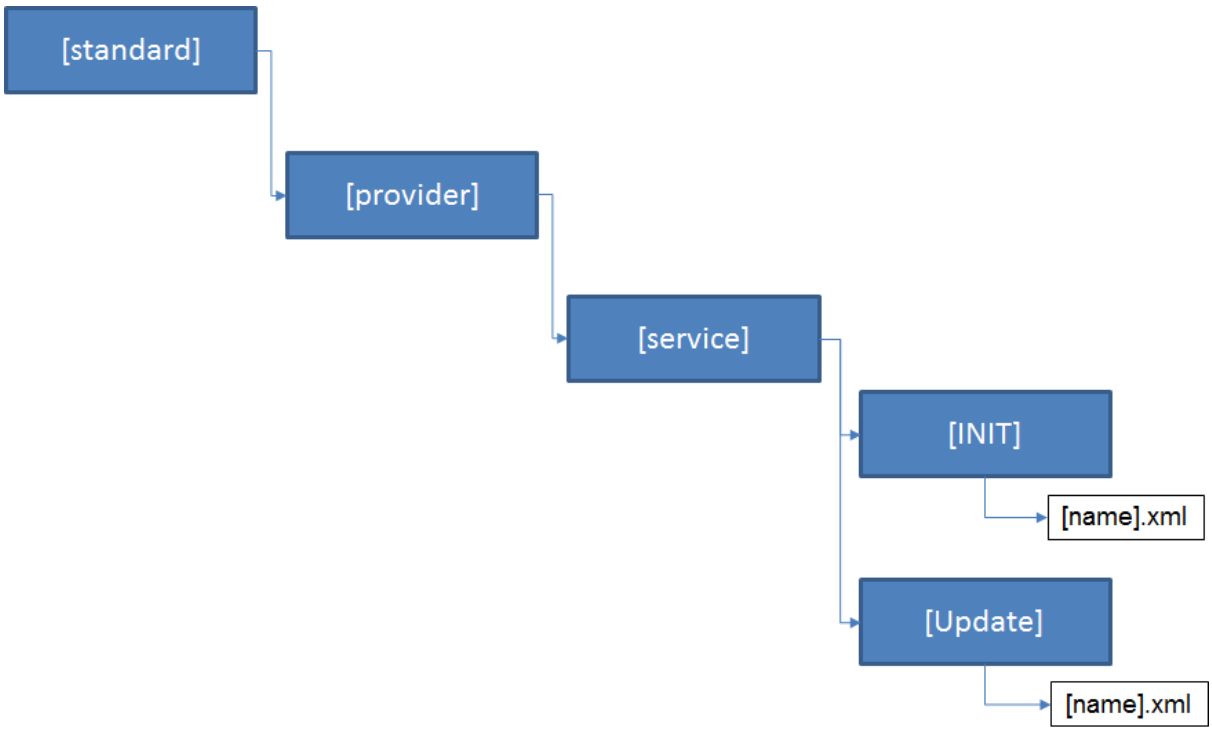

Figure 14 Hierarchical scheme of names (and folders)

The usage of the API for the subscription is described in the following code snippet.

```
1: import it.telematics.isl.Consumer.Consumer;
2: public class ConsumerExample {
3:   public static void main(String[] args) {
4:     String path            = "./datexii_names.txt";
5:     DatexIIHandler handler = new DatexIIHandler();
6:     File names_file        = new File(path);
7:     String lines[]         = null;
8:     boolean isOK           = false;
9:     try {
10:       lines = readFromFile(names_file)
11:         .toString().split("\\n");
12:
13:     ArrayList<String> namelist = new ArrayList<String>(
14:     Arrays.asList(lines));
16:     String _defWebsocketServer = "ws://telematics.poliba.it:8000/internames";
17:     new Consumer(_defWebsocketServer);
18:     isOK = Consumer.subscribe(namelist,
19:           handler);
20:     } catch (URISyntaxException e) {
21:       e.printStackTrace();
22:       isOK = false;
23:     } catch (IOException e) {
```

```
24:       if (e instanceof FileNotFoundException) {
25:         System.out.println("File doesn't exist");
26:       }
27:       else {
28:         e.printStackTrace();
29:         isOK = false;
30:       }
31:     }
```

The code reported above reads from a simple plain text file (which goes under the name of "**datexii_names.txt**"). It contains the list of names, which is stored within an array list object (for instance, **ArrayList<String>**). In order to manage DatexII publications, a dedicated class has been conceived, **DatexIIHandler**. It is meant for managing incoming publications. After the list of names has been created, the **subscribe** API exposed by the Consumer class is called. Two variables are passed to it: the list of names and the dedicated handler. The API is in charge of sending aforementioned list of names to the Consumer Proxy.

The example of the callback that implements the aforementioned folder hierarchy for DatexII publications is provided in the following code snippet. It shows how to receive and correctly parse the JSON messages.

DatexIIHandler.java

```
1: public class DatexIIHandler implements MessageHandler {
2:   private static final String    _pubDir      = "DatexII/npra";
3:   private static final Pattern  _pubTime_rgx  = Pattern.compile(
4:     "<publicationTime>(.+)</publicationTime>");
5:
6:   public DatexIIHandler() {
7:     new File(DatexIIHandler._pubDir).mkdirs();
8:   }
9:
10:   @Override
11:   public void handleMessage(String message) {
12:     File publication;
13:     long endDownloadTime  = System.currentTimeMillis();
14:
15:     JsonReader reader    = Json.createReader(new StringReader(
16:       message.replaceAll("\r?\n", "")));
17:
18:     JsonObject jsonMessage  = reader.readObject();
19:     reader.close();
20:
21:     String header  = jsonMessage.getString("header");
22:     String type    = jsonMessage.getString("type");
23:     String content = jsonMessage.getString("content");
24:
25:     switch (type) {
26:     case "log":
27:       System.out.println("Log :"+ header + ":" + content);
28:       break;
```

```
29:     case "data":
30:       try {
31:         StringBuilder workingDir = new StringBuilder(
32:           DatexIIHandler._pubDir);
33:
34:         String resource_name = header;
35:
36:         if (resource_name.contains("cctv")) {
37:           workingDir.append("/GetCCTVSiteTable");
38:         } else if (resource_name.contains("situation")) {
39:           workingDir.append("/GetSituation");
40:         } else if (resource_name.contains("weatherdata")) {
41:           workingDir.append("/GetMeasuredWeatherData");
42:         } else if (resource_name.contains("weathersite")) {
43:         workingDir.append("/GetMeasurementWeatherSiteTable");
44:         } else {
45:           System.err.println("Unknown service");
46:           System.err.println(content);
47:           return;
48:         }
49:
50:         Matcher pubTime_mtc = DatexIIHandler._pubTime_rgx.matcher(content);
51:         String pubTime = "";
52:         if (pubTime_mtc.find()) {
53:           pubTime = pubTime_mtc.group(1);
54:
55:           workingDir.append("/");
56:           workingDir.append(pubTime);
57:
58:           if (resource_name.endsWith("full")) {
59:             workingDir.append("-snapshot");
60:           }
61:
62:           new File(workingDir.toString()).mkdirs();
63:           publication = new File(workingDir.toString()
64:               + "/"
65:               + resource_name
66:               .replace(":", "\uFF1A")
67:               .replace("/", "\u2215")
68:               + ".xml");
69:           FileUtils.printOnFile(content, publication);
70:
71:           System.out.println("Content stored:" + publication.getCanonicalPath());
72:         } else {
73:           // invalid xml
74:           System.err.println("Invalid xml");
75:           return;
76:         }
77:       } catch (IOException e) {
78:         e.printStackTrace();
79:       }
80:       break;
81:     default:
82:       System.out.println(message);
83:       break;
84:     }
```

During the initialization phase of **`DatexIIHandler`** we create the folder structure previously described in Figure 14. A dedicated function (namely **`handleMessage()`**) is set for JSON

messages parsing; it is used for filing each incoming message according to the service it belongs to. An example of the folder hierarchy produced as output is reported in what follows:

```
DatexII
        NPRA
                GetSituation
                        2017-02-16T16:34:23.743+01:00-snapshot
                                n2n://poliba_datexii/bv/06/58/00/GPS-
                                ID/datexII/npra/situation/full.xml
                                ……
                        2017-02-16T16:52:09.007+01:00
                                n2n://poliba_datexii/bv/06/58/00/GPS-ID/datexII/npra/situation.xml
                                ……
                GetCCTVSiteTable
                        2017-02-16T16:34:24.090+01:00-snapshot
                                n2n://poliba_datexii/bv/06/58/00/GPS-ID/datexII/npra/cctv/full.xml
                                ……
                        2017-02-16T16:58:44.239+01:00
                                n2n://poliba_datexii/bv/06/58/00/GPS-ID/datexII/npra/cctv.xml
                                ……
                GetWeatherData
                        2017-02-16T16:34:23.553+01:00-snapshot
                                n2n://poliba_datexii/bv/06/58/00/GPS-
                                ID/datexII/npra/weatherdata/full.xml
                                ……
                        2017-02-16T16:49:01.102+01:00
                                n2n://poliba_datexii/bv/06/58/00/GPS-
                                ID/datexII/npra/weatherdata.xml
                                ……
                GetWeatherSiteTable
                        2017-02-16T16:34:23.743+01:00-snapshot
                                n2n://poliba_datexii/bv/06/58/00/GPS-
                        ID/datexII/npra/weathersite/full.xml
                                ……
                        2017-02-16T16:37:29.000+01:00
n2n://poliba_datexii/bv/06/58/00/GPS-ID/datexII/npra/weathersite.xml
……
```

5.3 DISCOVERY SERVICES

5.3.1 Description

BONVOYAGE discovery services for Data Sources and Soloists are supported by the OpenGeoBase federated spatial database whose JAVA and HTTP APIs are hereafter described.

5.3.2 OGB JAVA API

This is a java library developed for OpenGeoBase frontend[4]. To use this library in your project, export OgbJavaLibrary/src/com/bonvoyage/ogb/clientlib/OgbClientLib.java as runnable jar file and add it to the build path of your project. Then use

```
import com.bonvoyage.ogb.clientlib.*;
```

5.3.2.1 Library object constructor

Create a new OgbClientLib with optional settings to manage communication with the OGB FrontEndServer.

OgbClientLib(String serverURL)

- **Parameters:**
  - String serverURL : URL of the FrontEndServer (ip:port)

The following code shows an example of object allocation:

```
OgbClientLib ogbTestClient = new OgbClientLib(serverURL);
```

5.3.2.2 Login

This method allows the user to log in.
String **login**(String userId, String tenant, String password)

- **Parameters:**
  - *String* userId : the username of the user that attempts to login
  - *String* tenant : the tenant responsible for the user

[4] http://bonvoyage2020.eu/travelcentricservices

- *String* password : the password of the user

- **Response:**
  - (*String*) secure token to be used for next operations with the FrontEndServer. This token identify the user. keep it secret.

The following code shows an example of login:

```
String uid = "myUserID";
String tid = "myTenantID";
String pwd = "myPassword";
String token = ogbTestClient.login(uid, tid, pwd);
```

5.3.2.3 Point insertion

This method allows the user to insert a **Point** GeoJSON object.

String **addPoint** (String token, String cid, HashMap < String, String > propertiesMap, final double[ ] location)

- **Parameters:**
  - *String* token : the authorization token of the user
  - *String* cid : collection identifier
  - *HashMap < String, String >* propertiesMap : set of geo-json properties, hashmap < string properties, string property_value >
  - *double[ ]* location : [latitude longitude] double array

- **Response:**
  - (*String*) object identifier (OID) of inserted GeoJSON

- **Example**

The following code shows an example of **Point** GeoJSON object with additional properties insertion:

```
// point coordinates
double lat = 0.1;
double lon = 0.2;
double [] coordinates =  {lat, lon};
```

```
// point properties
HashMap<String,String> prop = new HashMap<String,String>();
prop.put("train-name", "ice-374");
prop.put("train-speed", "170 km/h");
// db insertion, response is the object identifier (oid)
String oid = ogbTestClient.addPoint(token,cid, prop, coordinates);
```

5.3.2.4 MultiPoint insertion

This method allows the user to insert a **MultiPoint** GeoJSON object.

String **addMultiPoint**(String token, String cid, HashMap < String, String > propertiesMap, ArrayList < double[ ] > coordinates)

- **Parameters:**
  - *String* token : the authorization token of the user
  - *String* cid : collection identifier
  - *HashMap < String, String >* propertiesMap : set of geo-json properties, hashmap
  - *ArrayList < double[ ] >* location : array list of [latitude longitude] double arrays
- **Response:**
  - (*String*) object identifier (OID) of inserted GeoJSON

The following code shows an example of **MultiPoint** GeoJSON object with additional properties insertion:

```
// multipoint coordinates
ArrayList<double[]> mcoordinates = new ArrayList<double[]>();
double lat_point1 = 0.01;   //point 1 latitude
double lon_point1 = 0.01;   //point 1 longitude
double lat_point2 = 0.02;   //point 2 latitude
double lon_point2 = 0.02;   //point 2 longitude
mcoordinates.add(new double[] { lat_point1, lon_point1 }); //point 1
mcoordinates.add(new double[]{lat_point2, lon_point2 }); // point 2
// point properties
HashMap<String,String> mprop = new HashMap<String,String>();
mprop.put("prop100", "value100");
mprop.put("prop200", "value200");

// DB insertion, response is the object identifier (oid)
String moid = ogbTestClient.addMultiPoint(token,cid, mprop, mcoordinates);
```

5.3.2.5 Polygon insertion

This method allows the user to insert a **Polygon** GeoJSON object.

String **addPolygon**(String token, String cid, HashMap < String, String > propertiesMap, ArrayList < double[ ] > coordinates)

- **Parameters:**
  - *String* token : the authorization token of the user
  - *String* cid : collection identifier
  - *HashMap < String, String >* propertiesMap : set of geo-json properties, hashmap
  - *ArrayList < double[ ] >* location : array list of [latitude longitude] double arrays
- **Response:**
  - (*String*) object identifier (OID) of inserted GeoJSON

The following code shows an example of **Polygon** GeoJSON object with additional properties insertion:

```
    ArrayList<double[]> pcoordinates = new ArrayList<double[]>();
    double polygon_lat_point1 = 0.00;   //point 1 latitude
    double polygon_lon_point1 = 0.00;   //point 1 longitude
    double polygon_lat_point2 = 0.11;   //point 2 latitude
    double polygon_lon_point2 = 0.11;   //point 2 longitude
    double polygon_lat_point3 = 0.11;   //point 3 latitude
    double polygon_lon_point3 = 0.00;   //point 3 longitude
    pcoordinates.add(new double[] { polygon_lat_point1, polygon_lon_point1 });
    pcoordinates.add(new double[] { polygon_lat_point2, polygon_lon_point2 });
    pcoordinates.add(new double[] { polygon_lat_point3, polygon_lon_point3});
    // point properties
    HashMap<String,String> polygonProp = new HashMap<String,String>();
    polygonProp.put("URL", "http:/myurl.it/gtfz.zip");
    polygonProp.put("Type", "GTFS");
    polygonProp.put("Provider", "Bonvoyage Project");

    // db insertion, response is the object identifier (oid)
    String poid = ogbTestClient.addPolygon(token,cid, polygonProp, pcoordinates
);
```

5.3.2.6 GeoJSON insertion

This method allows the user to insert a GeoJSON object. Supported GeoJSON shape are Point, MultiPoint and Polygon.

String **addGeoJSON**(String token, String cid, String geoJSON)

- **Parameters:**

- *String* token : the authorization token of the user
- *String* cid : collection identifier
- *String* geoJSON : string representing geoJSON object

The following text shows an example of **Point** GeoJSON object with additional properties:

```
{
    "geometry": {
        "coordinates": [12.28,41.63],       // coordinates in format longitude,
latitude
        "type": "Point"                     // supported geometry types are
Point, Polygon, MultiPoint
    },
    "type": "Feature",
    "properties": {                         // additional properties
        "train_speed" : "10"
        "delay" : "0"
    }
}
```

Following text shows an example of **MultiPoint** GeoJSON object with additional properties:

```
{
    "geometry": {
        "coordinates": [            //For type "MultiPoint", the "coordinates"
member must be an array of positions.
            [12.536976,41.950308],
            [12.459226,41.937373],
            [12.363410,41.914593],
        ],
        "type": "MultiPoint"
    },
    "type": "Feature",
    "properties": {
        "Provider", "Bonvoyage Project",
        "Type", "GTFS",
        "URL" : "http:/myurl.it/gtfs.zip"
    }
}
```

Following text shows an example of **Polygon** GeoJSON object with additional properties:

```
{
    "geometry" : {
        "type" : "Polygon",
        "coordinates" :
            [
                [
                    [0.0, 0.0],         //Polygon point 1 coordinate [latitude,
```

```
longitude]
                    [0.11, 0.11],       //Polygon point 2 coordinate
                    [0.0, 0.11]         //Polygon point 3: if needed, the API
method closes Polygon automatically connecting last and first points
                ]
            ]
        },
    "type":"Feature",
    "properties" : {            // object properties
        "name": "null island"
    }
}
```

- **Response:**
  - (*String*) object identifier (OID) of inserted GeoJSON

5.3.2.7 Object query

This method allows the user to retrieve a GeoJSON by its ObjectID.
String **queryObject**(String token, String cid, String oid)

- **Parameters:**
  - *String* token : the authorization token of the user
  - *String* cid : collection identifier
  - *String* oid : unique identifier of the geoJSON object
- **Response:**
  - (*String*) the requested GeoJSON object as string

The following code shows an example of object query:

```
    String oid =
"/OGB/000/000/21/00/GPS_id/GEOJSON/bonvoyage/testCID/test/q37871900e8w1r69";
    System.out.println("\n\n**** Object Query ****");
    String response1 = ogbTestClient.queryObject(token, cid, oid);
    System.out.println("query response: " + response1);
```

5.3.2.8 Range query

This method allows the user to find GeoJSON objects within a spcified **square** area. This method perform GeoJSON spatial query using square **box** shape and $geoIntersects geospatial operator. For more information visit [5].

String **rangeQuery**(String token, String cid, double sw_lat, double sw_lon, double boxSize)

- **Parameters:**
  - *String* token : the authorization token of the user
  - *String* cid : collection identifier
  - *double* sw_lat : south west latitude
  - *double* sw_lon : south west longitude
  - *double* boxSize : box edge size in degree
- **Response:**
  - (*String*) a JSON array of GeoJSON objects within the specified area (if no GeoJSON is present in the area the method returns an empty array).

The following code shows an example of query:

```
    double sw_lat = 0.0; // south west latitude in degree
    double sw_lon = 0.0; // south west longitude in degree
    double boxSize = 0.5; // box size in degree
    String response = ogbTestClient.rangeQuery(token, cid, sw_lat, sw_lon,
boxSize);
    System.out.println("query response: " + response);
```

5.3.2.9 Range query Box

This method allows the user to find GeoJSON objects within a spcified area. This method perform GeoJSON spatial query using **box** rectangular shape and $geoIntersects geospatial operator. For more information visit [6].

String **rangeQueryBox**(String token, String cid, double sw_lat, double sw_lon, double ne_lat, double ne_lon)

- **Parameters:**

[5] https://docs.mongodb.com/manual/reference/operator/query/geoIntersects
[6] https://docs.mongodb.com/manual/reference/operator/query/geoIntersects/

- *String* token : the authorization token of the user
- *String* cid : collection identifier
- *double* sw_lat : south west latitude
- *double* sw_lon : south west longitude
- *double* ne_lat : north east latitude
- *double* ne_lon : north east longitude

- **Response:**
  - (String) a JSON array of GeoJSON objects within the specified area (if no GeoJSON is present in the area the method returns an empty array).

The following code shows an example of box query:

```
    double sw_lat = 0.0; // south west latitude in degree
    double sw_lon = 0.0; // south west longitude in degree
    double ne_lat = 0.5; // north east latitude in degree
    double ne_lon = 0.5; // north east longitude in degree
    String response = ogbTestClient.rangeQueryBox(token, cid, sw_lat, sw_lon,
ne_lat, ne_lon);
    System.out.println("query response: " + response);
```

5.3.2.10 Range query Polygon

This method allows the user to find GeoJSON objects within a spcified area. This method performs GeoJSON spatial query using **Polygon** shape and $geoIntersects geospatial operator. For more information visit[7].

String **rangeQueryPolygon**(String token, String cid, ArrayList < ArrayList < double [ ] > > coordinates)

- **Parameters:**
  - *String* token : the authorization token of the user
  - *String* cid : collection identifier
  - *ArrayList < ArrayList < double [ ] > >* coordinates : ArrayList of ArrayList<[latitude longitude]> double arrays, at the moment supporting only polygon without holes (for more information visit http://geojson.org/geojson-spec.html#id4)

- **Response:**

[7] https://docs.mongodb.com/manual/reference/operator/query/geoIntersects/

– (String) a JSON array of GeoJSON objects within the specified area (if no GeoJSON is present in the area the method returns an empty array).

The following code shows an example of Polygon query:

```
ArrayList<ArrayList<double[]>> polygon = new ArrayList<>();
ArrayList<double[]> subPolygon = new ArrayList<>();
double queryPol_lat_point1 = 0.00;  //point 1 latitude in degree
double queryPol_lon_point1 = 0.00;  //point 1 longitude in degree
double queryPol_lat_point2 = 0.15;  //point 2 latitude in degree
double queryPol_lon_point2 = 0.15;  //point 2 longitude in degree
double queryPol_lat_point3 = 0.15;  //point 3 latitude in degree
double queryPol_lon_point3 = 0.00;  //point 3 longitude in degree
subPolygon.add(new double[] {queryPol_lat_point1,queryPol_lon_point1});
subPolygon.add(new double[] {queryPol_lat_point2,queryPol_lon_point2});
subPolygon.add(new double[] {queryPol_lat_point3,queryPol_lon_point3});
polygon.add(subPolygon);
String response = ogbTestClient.rangeQueryPolygon(token, cid, polygon);
System.out.println("query response: " + response);
```

5.3.2.11 Object removal

This method allows the user to delete a GeoJSON object

boolean **deleteObject**(String token, String oid)

Parameters:

- *String* token : the authorization token of the user
- *String* cid : collection identifier
- *String* oid : the object identifier (OID) of the GeoJSON to remove

Response:

- (*Boolean*) true/false if success/failure

The following code shows an example of object removal:

```
    String oid =
"/OGB/000/000/21/00/GPS_id/GEOJSON/bonvoyage/testCID/test/q37871900e8w1r69";
    boolean removalStatus = ogbTestClient.deleteObject(token, oid);
```

5.3.3 OGB HTTP API

The OpenGeoBase[8] provides a HTTP interface.

5.3.3.1 Register

This method allows to register a new user into the OpenGeoBase system.

- **URL:** /OGB/user/register
- **Method:** POST
- **consumes:** application/json
- **produces:** application/json
- **POST body:** a JSON object with the following keys:values :

  userName : [the username of the new user]

  password : [the password of the new user]

  tenantName : [the tenant responsible for the new user]

  permission : [the permission type for the new user ("r" or "rw")].
- **Response:** an empty response with status code 200 in case of success, an error otherwise with following code:
  - 407, "Register Failed"

5.3.3.2 Login

This method allows the user to log in.

- **URL:** /OGB/user/login
- **Method:** POST
- **consumes:** application/json
- **produces:** application/json
- **POST body:** a JSON object with the following keys:values :

  userName : [the username of the user that attempts to login]

  password : [the password of the user]

  tenantName : [the tenant responsible for the user]

[8] http://bonvoyage2020.eu/travelcentricservices

- **Response:** the users authentication token in case of success, an error otherwise with following code:
  - 407,"Error on login: Wrong credential provided!"

5.3.3.3 Object query

This method allows the user to retrieve a GeoJSON by its ObjectID. Collection id must be the last path component in the method entry point

- **URL:** /OGB/query-service/element/{cid}
- **Method:** POST
- **consumes:** application/json
- **produces:** application/json
- **POST header:** an HashMap with the following key:

  Authorization : [the authorization token of the user retrieved by login procedure]
- **POST body:** a JSON object with the following key:value :

  oid : [the object identifier (OID) of the requested GeoJSON]
- **Response:** the requested GeoJSON object as string if it exists into database, an error otherwise with following codes:
  - 420, "Invalid authorization token"
  - 431, "Invalid oid in GeoJSON!"

5.3.3.4 Range query

This method allows the user to find GeoJSON objects within a specified area. Collection id must be the last path component in the method entry point

- **URL:** /OGB/query-service/{cid}
- **Method:** POST
- **consumes:** application/json
- **produces:** application/json
- **POST header:** an HashMap with the following key:value :

  Authorization : [the authorization token of the user]
- **POST body:** a JSON object that describes the geospatial query. Currently only "$geoIntersects" geospatial operator and $geometry.type {Box, Polygon} are supported.

More information at
https://docs.mongodb.com/manual/reference/operator/query/geoIntersects/

The following text shows an example of range query:

```
{
    "geometry": {
        "$geoIntersects": {
            "$geometry": {
                "type": "Box" ,              // supported geometry types are
"Box" and "Polygon"
                "coordinates":[
                    [12.28,41.63],          // bottom left coordinates in format
longitude, latitude
                    [12.73,42.02]           // upper right coordinates in format
longitude, latitude
                ]
             }
        }
    }
}
```

- **Response:** an array of GeoJSON objects (as string) within the specified area (if no GeoJSON is present in the area the method returns an empty array). In case of failure the following error codes are returned:
  - 420, "Invalid authorization token"
  - 430, "Invalid query params"
  - 440, "Requested area size exceeds the maximum limit"

5.3.3.5 Object insertion

This method allows the user to insert a GeoJSON object. Collection id must be the last path component in the method entry point

- **URL:** /OGB/content/insert/{cid}
- **Method:** POST
- **consumes:** application/json
- **produces:** application/json
- **POST header:** an HashMap with the following key:value :
- Authorization : [the authorization token of the user]

- **POST body:** a JSON object that describes the geospatial structure to be inserted in OGB. Supported geometry types are the following: Point, Polygon, MultiPoint. Geometric objects with additional properties are Feature objects.

Following text shows an example of **Point** GeoJSON object with additional properties:

```
{
    "geometry": {
        "coordinates": [12.28,41.63],       // coordinates in format longitude,
latitude
        "type": "Point"                     // supported geometry types are
Point, Polygon, MultiPoint
    },
    "type": "Feature",
    "properties": {                         // additional properties
        "train_speed" : "10"
        "delay" : "0"
    }
}
```

Following text shows an example of **MultiPoint** GeoJSON object with additional properties:

```
{
    "geometry": {
        "coordinates": [            //For type "MultiPoint", the "coordinates"
member must be an array of positions.
            [12.536976,41.950308],
            [12.459226,41.937373],
            [12.363410,41.914593],
        ],
        "type": "MultiPoint"
    },
    "type": "Feature",
    "properties": {
        "Provider", "Bonvoyage Project",
        "Type", "GTFS",
        "URL" : "http:/myurl.it/gtfs.zip"
    }
}
```

Following text shows an example of **Polygon** GeoJSON object with additional properties:

```
{
    "geometry" : {
        "type" : "Polygon",
        "coordinates" :
            [
                [
```

```
                    [0.0, 0.0],         //Polygon point 1 coordinate [latitude,
longitude]
                    [0.11, 0.11],       //Polygon point 2 coordinate
                    [0.0, 0.11]         //Polygon point 3: if needed, the API
method closes Polygon automatically connecting last and first points
                ]
            ]
        },
    "type":"Feature",
    "properties" : {            // object properties
        "name": "null island"
    }
}
```

- **Response:** the method returns a string representing the Object Identifier (OID) of the inserted GeoJSON object, an error message otherwise with following codes:
  - 420, "Invalid authorization token"
  - 421, "Invalid permission type "
  - 407, "Failed to upload Content!"
  - 407, "Error on upload Content! Empty Content!"

5.3.3.6 Object removal

This method allows the user to delete a GeoJSON object

- **URL:** /OGB/content/delete
- **Method:** POST
- **consumes:** application/json
- **produces:** application/json
- **POST header:** an HashMap with the following key:value :

  Authorization : [the authorization token of the user]
- **POST body:** a JSON object with the following key:value :

  oid : [the object identifier (OID) of the GeoJSON to remove]
- **Response:** the method returns an empty response with status code 200 in case of success, an error otherwise with following codes:
  - 420, "Invalid authorization token"
  - 431, "Invalid oid in GeoJSON!"
  - 403, "User unauthorized!"
  - 421, "Security issues: User grant not retrieved!"

6 External Services

This section describes external services that are consumed and integrated into the BONVOYAGE platform as well as the BONVOYAGE app that uses the internal services provided by the platform.

The mobile application is seen from an architectural perspective as a construct which is external but is developed by also integrating libraries and components that are part of the BONVOYGE Application Layer, of course making use of all BONVYAGE services via their public APIs. External integrators can use the same approach and integrate innovative functionality into existing or new services. Section 6.1 briefly describes the integration of the services. Firebase is used as an authentication service for the platform, and its integration is described in Section 6.2.

The integration of external data sources and the MetaData Handling Tool are discussed in Section 6.3.

6.1 BONVOYAGE Mobile Application

The BONVOYAGE Mobile App implements an end-user interface developed as Android application. It represents an exemplary external usage of the BONVOYAGE platform. Similarly, an application for Apple's iOS could be provided, but is out of scope for this research project.

The mobile application uses the public available interfaces provided by platform. The application is described in the intermediate deliverable I6.2 App and the upcoming deliverable D6.2 in detail. The latest version of the app is available at[9]. All communication between the app and the BONVOYAGE platform is via the Application Server. Figure 15 shows the architectural view of the communication between the BONVOYAGE mobile application and the application server. The communication interfaces are described in Chapters 3 and 4. In order to convert addresses to geographic coordinates when consuming the inter-modal routing service, the Google Places API for android is used[10].

[9] https://download.fluidtime.com/ec/bonvoyage/android
[10] https://developers.google.com/places/android-api/autocomplete?hl=en

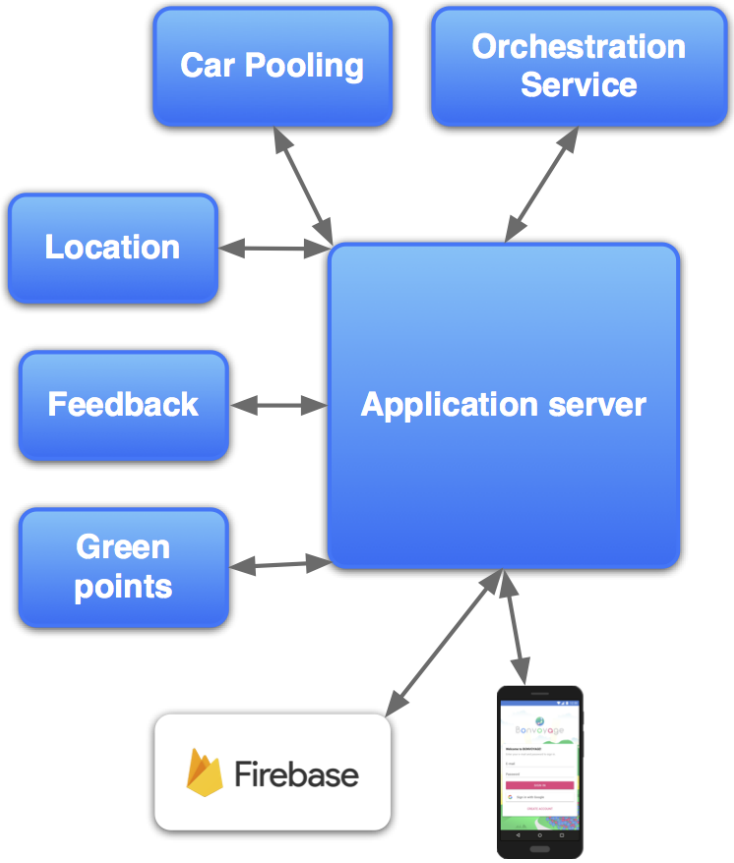

Figure 15 BONVOYAGE Mobile Application interaction

The authentication of the users is done via an external service. Firebase is used as authentication service as described in Section 6.2. It allows the reuse of exiting accounts such as Google or Facebook for the BONVOYAGE platform. Figure 16 shows the login screen of the BONVYAGE mobile application allowing Google accounts or the creation of a new account for the platform. The Firebase framework is also used to provide push-messages to the mobile clients.

The openness of the APIs and the authentication framework lowers the entry barrier for an external developer to integrate BONVOYAGE services into their systems or platforms.

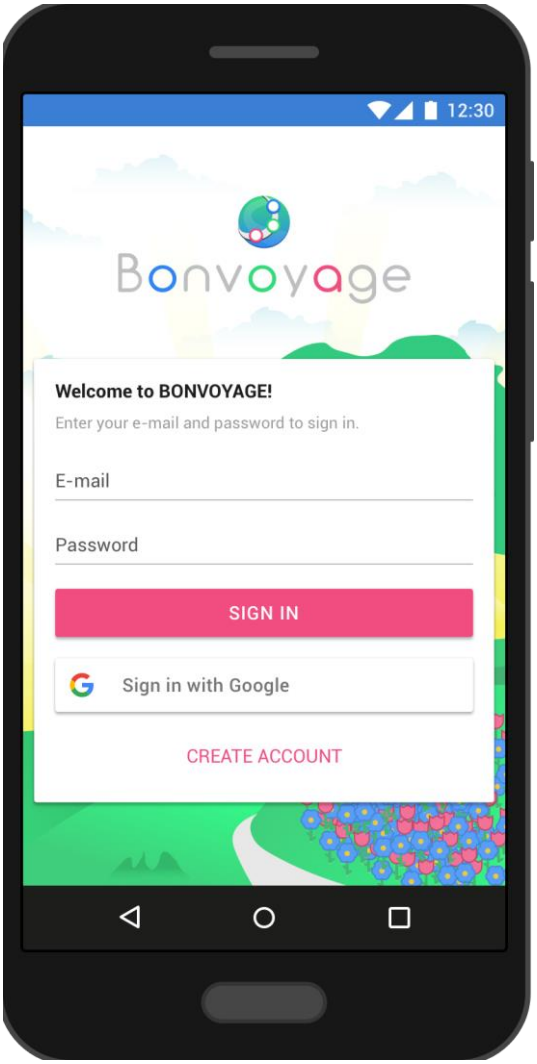

Figure 16 Login screen of the BONVOYAGE Mobile Application

6.2 Firebase

Firebase[11] is a platform operated by Google providing different service for cloud and mobile services (Android, iOS, and Web). It offers a free tier with essential services and hence is very well suited for developing apps and scaling the infrastructure and features based on the adoption by users[12]. Two free services of the platform are used for the BONVOYAGE mobile application, Cloud Messaging and authentication. The service was used to guarantee the reuse and extensibility of the used services. Firebase allows the easy reuse for other client implementation or integration into exiting solutions.

[11] https://firebase.google.com
[12] https://firebase.google.com/pricing/

The Firebase Cloud Messaging is a framework that can be used for push-notification of mobile applications. The advantage of the framework is the cross -platform support. For BONVOYAGE it used to notify users of the Android mobile application, but it can easily be extended to support iOS or Web Clients as well. The detail technical specification of the service is provided at [13].

Authentication is needed in terms of personalisation of the BONVOYAGE services. For authentication the main requirements were a low entry barrier and easy integration of existing identity services. Firebase provides an implementation of the OAuth standard (published as RFC6749[14]). It allows the integration of existing identify services such as Google, Facebook, Twitter and GitHub that can be used for login at the BONVOYAGE platform. The service allows the creation and management of new account. The technical description of the Firebase Authentication service is available at [15].

6.3 Data Sources

For what concerns the Data Sources, we do not consider them just as “simple” external services to connect to. We have developed a distributed and federated approach (see D1.2 Project Vision) to aggregate travel data and services under the umbrella of a federation that supports the view of linked services and National Access Points advocated by the European Union directives on Intelligent Transport Systems.

Of course there are dozens different technologies and end-points where travel data can be fetched from (for instance, some of the services we have integrated in BONVOYAGE include car2go car-sharing APIs, GTFS files, manually compiled excel sheets, the Bilbao CoCities platform, the Norwegian NPRA server, etc…). So we must distinguish:

- An adaptor to the specific Data Source technology. We do not describe all off-the-shelf technologies used to fetch/parse the different sources/services, because they consist of standard interfacing technologies and they are dependent on the specific provider of the data. Adapters can be either distributed (i.e. deployed at the Transport Operator, for instance within the DS blocks of Figure 1), or centralized within the MetaData Handling Tool (MDHT block).
- A module that uses such different adaptors to look for the information we need and brings it inside our platform. This is the MetaData Handling Tool.

[13] https://firebase.google.com/docs/cloud-messaging
[14] https://tools.ietf.org/html/rfc6749
[15] https://firebase.google.com/docs/auth

Generally speaking, the integration of the information provided by external data sources is done under the scope of the "*Discovery and Pub/Sub*" functional block of the BONVOYAGE high-level architecture (see Figure 1).

This process is widely described in Deliverable 5.1, where the technical functionalities of the MetaData Handling Tool are detailed, and in Deliverable 7.1, where a data workflow is developed and the flow of information across the different components (including external data sources) is described in a use case and realistic scenario. Deliverable D3.3, which is towards the end of the project, gives an account of all Travel-centric services developed in BONVOYAGE for what concerns the capability of the platform to intercept data coming from heterogeneous Data Sources and efficiently disseminate it in those cases when information is massive and highly dynamic.

6.3.1 Handling of travel data

Periodic scanning of the different data sources is done via the MetaData Handling Tool (MDHT). This component, which is functionally described in Deliverable 5.1, consists of a distributed set of adaptors and monitors, which are deployed at the Travel Operator premises (or are able to directly talk to the Travel Operator or data relay infrastructure) and perform several tasks within the BONVOYAGE architecture.

- Make sources "*discoverable*" for the platform.
- Continuously track changes and evolution of the data flow coming from the data source.
- Provide a feasible and seamless communication channel between the data sources and the processing core of the platform.

We have designed a simple, effective and optimized process for making data sources discoverable and usable, by means of three different components we have in the "Discovery and Pub/Sub" functional block:

- Internames Communication System
- MetaData Handling Tool
- OpenGeoBase

Once a new data source appears, the first step is granting the access to its information. Deploying an adaptor "in front of" the data source performs this first step, and a monitor is connected to it so that the MetaData Handling Tool scans the information provided by the data source via standard protocols and off-the-shelf technology (we have develop, for instance, parsers for protocols such as NeTeX, DATEX II, GTFS).

After that basic linking, the MDHT periodically parses this information, and processes it by extracting any geographical coordinates and as much meta-information about how to connect, authenticate and about the kind of data available. These geographical data-points (plus meta information) will be the input for an OGB georeferenced database entry: after this processing is performed, the BONVOYAGE platform knows that for a specific travel planning problem, concerning a confined geographical area, a concrete data source can be used for solving it. Such a discovery service is where the different components will later on ask for finding out whether relevant data exists for solving their particular travel problems while a travel is being planned.

Furthermore, once a data source is made "*discoverable*" by inputting it in OGB, its (typically dynamic) raw information can then be received for solving of particular travel problems, for example in a segment of a long trip. The different soloists attached to a specific geographical region access the information, provided by these data sources. They go to the data sources of those specific areas that are indicated in OGB. Specific soloists and specific data sources are used for different areas.

In case the data source inherently deals with massive and highly changing data streams, it is assigned to a distribution channel under a well-specified name (topic-based publish/subscribe paradigm, see D3.2 and D3.3) so that it acts as a publisher and the soloists can act as subscribers through a connection via the Internames Communication System. The data source is then fully linked and only incrementally small differences in the data stream are propagated to the subscribing soloists.

7 Summary

D6.1 presented the design and development of technology dependent interfaces towards the external actors (transport operators, data sources and end user applications). Furthermore, it contains the description of all BONVOYAGE components providing interfaces to internal and external stakeholders. The technical descriptions of this document provide interface specifications and the minimum level of implementation details in order to enable seamless integration of the components.

This deliverable provides fundamental technical guidelines for the exploitation of the project results of BONVOYAGE. The development and the integration process of the components within the BONVOYAGE platform will be further detailed in the upcoming deliverables D7.1 and D7.2.

The deliverable provides specification of the internal services that provide publicly available APIs for external users. These services were developed in WP 3, 4, 5 and 6 of the project. This document enables external stakeholders to integrate BONVOYAGE services into their environment, services or apps. The detailed description includes examples and error codes and the used data formats. This helps ensuring a seamless integration and uptake of the developed services.

The services discussed in this deliverable include the real-time intermodal routing service providing intermodal route results for cross-national door-to-door travels. All routes are enriched by Greenpoint information. This individualised fidelity program aims at changing the travel behaviour of the user towards more sustainable behaviour. The Greenpoint module implements the required functionalities to provide all needed information. The feedback module provides the capabilities to gather feedback from users about the BONVOYAGE services and the suggested travel recommendations. The BONVYAGE platform implements a car-pooling service allowing to offer, query and book car-pooling services. The interfaces of Android library to determine the user mode of transport and the user stress level are presented in this deliverable. The libraries use smartphone sensors to determine the mode of transport and Empatica wristband sensors for the stress level. The APIs of communication services based on Internames, namely the publish-subscribe service and the OGB, are presented in detail, although for tight and large-scale integrations of these services WP3 deliverables provide more details about the underlying Internames Information-Centric Network.

While the deliverable provides an exhaustive description of the public available APIs provided by the BONVOYAGE platform, the deliverable also discusses the gathering of external data services hosted by Transport Operates that are needed for the platform. The data sources are integrated into a federation of Data Sources of the BONVOYAGE Infrastructure Layer. Other services of

BONVOYAGE are using the information provided by the communication service of the Infrastructure Layer that is based on Internames. With respect to the data source integration D5.1 deals with the adaption of the data sources to BONVOYAGE using the Metadata Handling Tool. The technicalities of the communication infrastructure are discussed in deliverables of WP3. Still, the complete API specification for the publication and subscription is provided in this deliverable. The efficient integration of the Data Sources Federation will be analysed in detail in the upcoming deliverable D7.1.

The BONVOYAGE mobile application uses the publicly available APIs of the BONVOYAGE platform, the same interfaces potential external integrators will use. The external service Firebase is integrated into the BONVOYAGE platform for authentication of end users.

Summing up, this deliverable provides a practical guide for reuse and integration of the BONVOYAGE services and the relevant components to all stakeholders. It details the interface implementation of the services. More details about integration and deployments of the BONVOYAGE platform will be provided in D7.1.