WP3 - DEVELOPMENT OF ENHANCED TRAVEL COMPANION AND RIDE-SHARING TSP

D3.3 CROWD-BASED TRAVEL EXPERT SERVICE

This project has received funding from the Shift2Rail Joint Undertaking under the European Union's Horizon 2020 research and innovation programme under grant agreement no. 881825

Project Acronym	RIDE2RAIL
Starting date	01/12/2019
Duration (in months)	36
Deliverable number	D3.3
Call Identifier	S2R-OC-IP4-01-2019
GRANT Agreement no	881825
Due date of the Deliverable	31/05/2022
Actual submission date	27/05/2022
Resposible/Author	FST
Dissemination level	PU
Work package	WP3
Main editor	Luca Mariorenzi
Reviewer(s)	OLTIS, INLECOM
Status of document (draft/issued)	Issued

Reviewed: yes

Consortium of partners

PARTNER	COUNTRY
UNION INTERNATIONALE DES TRANSPORTS PUBLICS (UITP)	Belgium
FIT CONSULTING	Italy
OLTIS GROUP	Czech Republic
FST	Italy
CEFRIEL	Italy
CERTH	Greece
EURNEX	Germany
EURECAT	Spain
POLIMI	Italy
UNIVERSITY OF NEWCASTLE UPON TYNE	United Kingdom
UNIFE	Belgium
UIC	France
UNIZA	Slovakia
ATTIKO METRO	Greece
INLECOM	Greece
FV-Helsinki	Finland
METROPOLIA	Finland

Contract No. 881825

DOCUMENT HISTORY		
Revision	Date	Description
1	01/04/2022	Table of content
2	12/05/2022	First complete draft
3	24/05/2022	Pre-final version addressing reviewers comments
4	27/05/2022	Final version

REPORT CONTRIBUTORS		
Name	Beneficiary Short Name	Details of contribution
Luca Mariorenzi	FST	Lead author
Simone Salviati	FST	Author
Diego Lisi	FST	Author
Enrica Caiello	FST	Author
Alexander Nemirovskiy	PoliMI	Contributor
Matteo Rossi	PoliMi	Contributor
Alessio Carenini	Cefriel	Contributor
Harris Niavis	INLECOM	Reviewer
Petra Juránková	OLTIS	Reviewer

Disclaimer

Contents

Contract No. 881825

List of figures

Contract No. 881825

1. EXECUTIVE SUMMARY

This deliverable describes the results of Task 3.3 – CbTSP formally known as Crowd Based Travel Service Provider, developed in the scope of the RIDE2RAIL (R2R) project. The main objective is to create an innovative framework for smart multimodal mobility by facilitating the efficient combination of flexible and crowdsourced transport services, such as ridesharing, with scheduled transport. This framework will be integrated into existing collective and on-demand transport services, connecting and reinforcing the mobility offer with ridesharing services, especially in rural and low-demand areas.

The goal of Ride2Rail project is to enable access to high-capacity services (public transport services) thanks to easy-to-use multimodal and integrated travel planning, booking, ticketing, and payment features. The Ride2Rail framework will integrate real-time and various information about rail, public transport, and shared mobility in a social ecosystem.

The CbTSP is a Hybrid TSP in the sense that is a software solution with an intrinsic duality; it has to act similarly to a standard TSP, in the sense that has to provide “travellers” with travel solutions based on the properties of their requests (i.e, departure coordinate, arrival coordinate, date and time). The core difference is that the planned routes that are used to compute the solutions for the “travellers” are not based on a fixed schedule as may be found in a standard TSP (i.e the web application of a bus public transportation service, where bus lines and bus stops are fixed over long periods of time) but those “routes” are generated dynamically starting from specific interactions that “drivers” entities have with the CbTSP.

A “driver” may decide to offer a seat in his car for any traveller to reserve, in that case it interacts with the CbTSP sending useful information (i.e his departure coordinates and arrival coordinates, starting date and time), the CbTSP computes for him a route which includes in a “Trip Plan” that gets inserted into the CbTSP data sets. From that point, this “planned trip” may appear to a traveller as a travel solution if it matches his search criteria among other similar solutions. If that solution satisfies his needs, he can then reserve a seat for that specific “ride” (lift reservation).

The core offered functionalities are:

- Ability to register as a driver
- Ability for a driver to plan, modify or delete a ride in a specific day and time
- Ability for an unregistered user / external system to search for available lifts matching criteria
- Ability for an unregistered user / external system to reserve a seat on a ride (lift reservation)
- Ability for an unregistered user / external system to cancel a reservation

In short terms, the CbTSP merges the functionality of a TSP with the one of a Social carsharing solution.

The CbTSP solution has been developed as a collection of specific submodules, when each of them is responsible for a specific aspect of the ecosystem:

Contract No. 881825

- a planner component which is responsible for route planning and rides;
- a backend for anagraphics, rides and lifts logics and general data storage such as the rides planned on the planner;
- an orchestrator that manages those submodules and offers an external interface (API).

This Deliverable is intended to be a companion to the software documentation available on GitHub under the RIDE2RAIL organization at https://github.com/Ride2Rail.

1.1. Abbreviations and acronyms

AL	Agreement Ledger
API	Application Programming Interface
CFM	Calls for Members
CbTSP	Crowd-based transport service provider
CRUD	Create, read, update, delete
DL	Dissemination and exploitation leader
DoA	Description of the Action
EL	Ethical leader
EU	European Union
FIFO	First in - First out
FS	Financial Statement
GA	Grant Agreement
H2020	Horizon 2020
IP4	Innovation Programme 4
OC	Open Call
PC	Project coordinator
PM	Project manager
PMO	Project Management Office
PMT	Project Management Team
PO	Project Officer
pTT	Partial Trip Tracker
QAC	Quality Assurance Committee
RP	Ride progress module

Contract No. 881825

RT	Ride tracking component
S2R JU	Shift2Rail Joint Undertaking
TC	Travel companion
TL	Technical leader
TO	Tracking Orchestrator
TT	Trip Tracking Service
TSP	Transport service provider
WP	Work Package
WPL	Work package leader

2. BACKGROUND

This document constitutes the Deliverable D3.3 "Crowd-based Travel Expert Service" in the framework of WP3 "Development of enhanced Travel Companion and Ride-sharing TSP" and describes what has been developed within Task 3.3 "Crowd-based Travel Expert Service" of Ride2Rail project.

Contract No. 881825

3. OBJECTIVES/AIM

The general goal of this deliverable is to provide relevant information about the implementation of the Crowd-based Travel Service Provider, focusing on the components that allow users and communities to build their own TSPs to offer shared rides by their own private car.
The task in the Grant Agreement was described as follows:

"In order to give individuals and communities the ability to act as self-organized TSPs using their own vehicles, this task will develop software tooling allowing them to build, test and publish Travel Expert web services to the TSP ecosystem through an Asset Manager-based lifecycle process. Built on the Interoperability Framework's Asset Manager, the tool is designed to instantiate a Travel Expert web service out of generic template stubs configured by the users with the specifics of their offering capabilities. The testing phase in the lifecycle includes the application of syntactic, semantic and ecosystem governance validation rules. On publication, the Travel Expert is registered in the Travel Expert registry and available to the Interoperability Framework Services. This task will take advantage from the SocialCar project in reusing and engineering algorithms and modules of the Travel Expert for trip planning combining Ride Sharing and scheduled transport services."

To this end, all relevant components will be described in details:

- CbTSP components
    - Trip planner
    - Back-end
    - Orchestrator
- Trip tracking
- Asset manager

4. CONTENT OF DELIVERABLE

4.1.Introduction

Crowd Based Travel Service Provider, also known as Crowd Based Travel Expert (from now on simply referred as CbTSP) is a collection of specifically developed components that plays at unison to achieve a very specific functionality, which is to offer a TSP alike service to general people, where potentially the same people can act both as “traveller” and as “drivers” intercheangeably. If someone wants to offer a Ride, the CbTSP offers facilities to plans ahead the future route, and add that plan to it’s collections of available “rides” for a potential traveller to look for, while if someone is looking for a “lift”, the same collection of software offer facilities which allows him to look for available offered rides, the possibility to book them, cancel them and so on.
As said before all the funcionalities are achieved by more than one project of which the specific functionalities gets orchestrated in order to achieve all the final goals (plan a ride, offer a ride, cancel a ride, search for an available lift, reserve a lift etc). For this purpose, SocialCar project’s[1] algorithms and modules of the Travel Expert for trip planning combining Ride Sharing and scheduled transport services have been reused.

[1] Alquiler de coches particulares - SocialCar

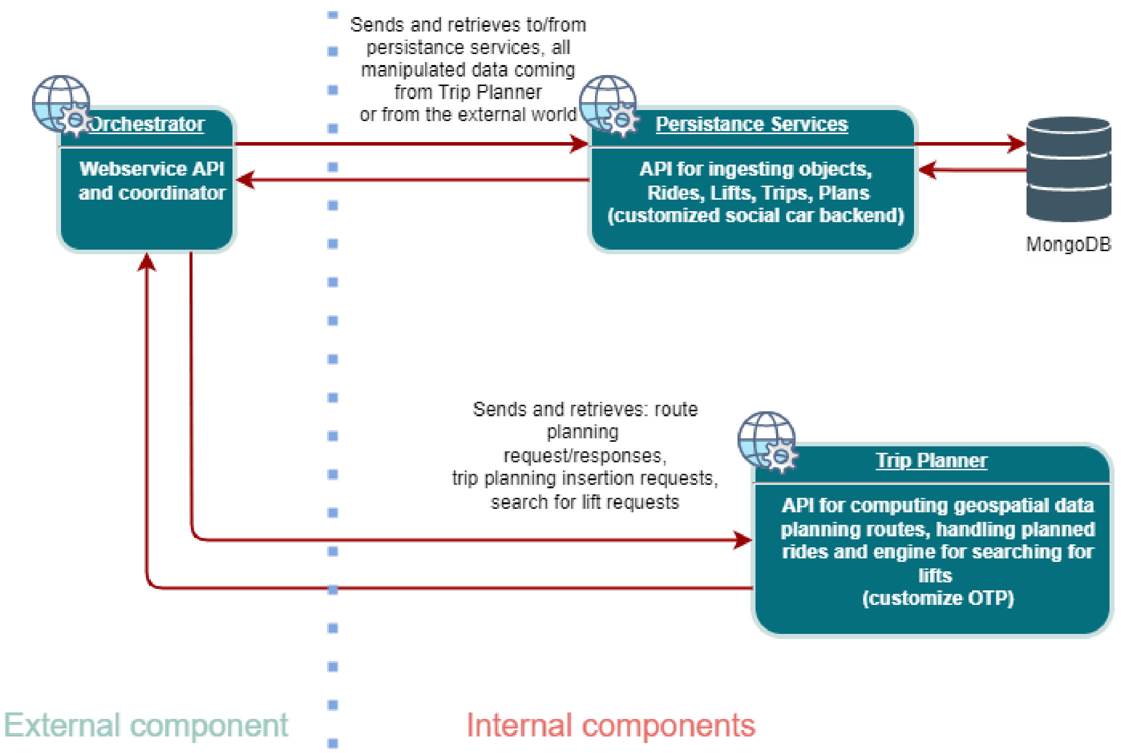

Sends and retrieves to/from persistance services, all manipulated data coming from Trip Planner or from the external world
Orchestrator
Webservice API and coordinator
Persistance Services
API for ingesting objects, Rides, Lifts, Trips, Plans (customized social car backend)
MongoDB
Sends and retrieves: route planning request/responses, trip planning insertion requests, search for lift requests
Trip Planner
API for computing geospatial data planning routes, handling planned rides and engine for searching for lifts (customize OTP)
External component
Internal components

Figure 1 General Architecture

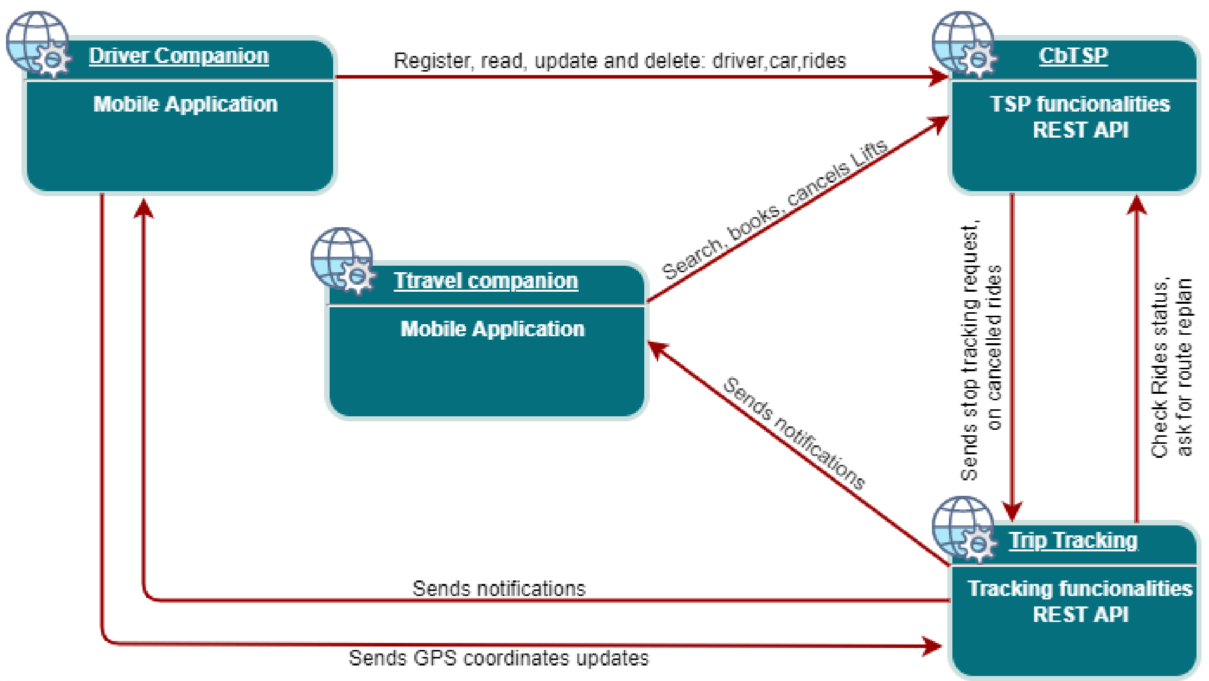

Driver Companion
Mobile Application
Register, read, update and delete: driver,car,rides
CbTSP
TSP funcionalities REST API
Ttravel companion
Mobile Application
Search, books, cancels Lifts
Sends notifications
Sends stop tracking request, on cancelled rides
Check Rides status, ask for route replan
Trip Tracking
Tracking funcionalities REST API
Sends notifications
Sends GPS coordinates updates

Figure 2 CbTSP's interactions with other Ride2Rail components

4.2. Trip Planner

4.2.1. High level component description

Trip planner component is the piece of software responsible for the functionalities related to route evaluation, route planning and scheduling, its main purpose is to expose facilities to allow from external interaction to request a route plan from a source point to a destination point, in this case it acts similarly to a standalone GPS navigator, or as a service like Google Maps. Once the route has been planned, it can be asked to store that route and to include it in its data structure as a planned ride, in this case if an external request comes asking for a lift (so an available ride from nearby a certain source point and destination point) the planner will scan among its internal data structure of beforhand inserted planned rides, for all the rides compliant with the request.

4.2.2. Main Functionalities

Trip Planner main functionalities are:

1. Computing a route from point A to point B and sending this route as a response like a planned car route (generally speaking, similar to a standard gps navigator or Google Maps service).
2. Plan a Ride in a specific date/time with an associated route between two endpoints and insert it into its datastructures such that it can be searched as a travel service (this basically means that the planned ride poses itself like public transportation ride, but it's offered by a person, by car).
3. Search for possible Rides in a specific date/time betwen two endpoints. This functionality allows some degree of slack with respect to the endpoints and the time of departure (i.e if someone asks for departure at a 9:00 am from a certain geospatial point, he will get more than one results starting from all those rides that depart at 9:00 am nearby his departure point of interest.
4. Delete a ride from the planned ones.

Strictly speaking, the Trip Planner does not hold information about drivers or passengers, aside from some unique ids representing the driver, as it's an engine responsible for route planning and rides definition.

4.2.3. Technical and implementation details

Contract No. 881825

The trip planner implementation is based on a customization of the OpenTripPlanner project[2], which is developed in Java, customizations involved enabling the software to take and store dynamically planning requests in response to an input from the outside world, based on the geospatial data configured internally.

Requests are handled in a REST fashion for ease of integration and includes the possibilities to plan a route from two points (source and destination), to store that route as a planned ride for a specific date and time (offered ride) and so on.

The more store requests it gets, the more its "available ride set" grow accordingly. This dynamically maintained data structure is built by users requests and becomes available as a constantly changing set of rides, when an external system posing as traveller sends a request: are there any "public transport like" rides available from point A to point B? The systems answers with a collection of compatible "offered rides".

4.3. Back-end

4.3.1. High level component description

The general backend is based on SocialCar backend component with some modifications to adapt it to its new purpose, this component is used to store all the anagraphics and in general, related objects informations. It's responsible for holding informations about the drivers (like usernamame, passowrd etc) but also other object of interests like the car associated to a driver, the rides he decides to publish, but also the Lifts (which are the reservation that passengers requests on a specific Driver's offered Ride). Rides and Lifts also encompass a simplyfied version of the planned route produced by the Trip Tracker (basically stripped from useless, debug or redundant informations).

This component saves also the planned Ride from the context of the Trip Planner. Since Trips in the Trip Planner are generated "on the fly" whenever a request is formulated, after being inserted in the Trip Planner datastructure which is not persistent for performance reasons, they are also stored into the backend database, just in case there is the need for a restart of the trip planner component, so that those already planned Rides can be restored into the Trip Planner component without much hassle. Backend acts basicaly as a "library of objects", aside from the logic to ensure consistency a and correctness of the data stored, it does not perform other core business logic which are mainly handled by the Orchestrator component.

[2] OpenTripPlanner

Contract No. 881825

4.3.2. Main Functionalities

Back-End main functionalities are basically CRUD operations, more specifically:

- Managing driver entity
- Managing associated objects to driver entity like cars
- Managing offered Ride objects, (i.e a Driver wants to offer a Ride with is Car, with a specific planned route on a specific day, his car has some properties, like the number of seats available which determines the maximum number of passengers he can transport). Some of the information in the ride objects comes from other components like the Trip Planner, and the Orchestrator is responsible for the correct processing, asssembling and storing onto the backend
- Managing Lifts. Similarly to the Ride objects, the heavylifiting of the computation is done by the Orchestrator which ultimately saves the reserved lift onto the backend which is basically responsible for updating and correctly linking the objects togheter (i.e a reserved lift refers to a specifi ride, so once a request for storing a lift is made, the appropriate ride gets updated with the lift information)

4.3.3. Technical and implementation details

The back-end implementation is based on a slightly modified version of the SocialCar backend module which is implemented in python language which stands as a backend with REST API capabilities interfacing with an instance of mongoDB.

Of all the functionalities offered by this module, only those useful for anagraphics are used, basically CbTSP uses the SocialCar backend to persist informations about the business objects (drivers, rides, lift etc.) using internally the already strong and well developed backend REST API plus it stores directly onto the mongoDB instances the route planning object processed by the trip-planner module so that they can be restored automaticaly in case there is the need to restart the module, since those objects are not persisted by the trip-planner.

CbTSP backend basically acts as an interface to a mongoDB instance offering simple REST API to insert syntactically correct objects in the interest of the specific context, it's an internal module that is used only by the CbTSP director module (the Orchestrator).

4.4. Orchestrator

The Orchestrator is the focal point of interest of the CbTSP, albeit all other components are far from being of lesser importance, this is basically the entry point for external requests and the component responsible for manipulating all the objects that come from the other two submodules, in a way to enforce coherence and meaning. The Orchestrator offers external REST API's for interact with the CbTSP and "de facto" implements calls that maps the use cases requested for this component. It is then responsible to make so that, for example there is a non ambigous relation from the representation of a trip object of the

Contract No. 881825

Trip Planner component, and a ride object in the context of the backend so that the two objects are uniquely correlated, also since the sets of trips in the context of the Trip Planner is a volatile and ever changing ensamble, it is also responsible for storing the trip plans into the backend so that if there are needs for restarts or reconfiguration of the Trip Planner component, the orchestrator can detect that and reinsert the still valid, previously computed trips again into the Trip Planner components.

4.4.1. Main Functionalities

Main functionalities regarding the Orchestrator are to pose as an entry point for every service that would like to interact to the CbTSP. This means that it is the Orchestrator that offers the API that is exposed to the world in order to do any kind of operations. The entrypoints offerd with the API covers te functions below:

- Driver Functionalities:
  - Driver creation
  - Driver retrieval
  - Driver information update
  - Driver deletion
- Cars (of a driver) Functionalities
  - Car creation
  - Car retrieval
  - Car information update
  - Car deletion
- Ride Functionalities
  - Ride creation
  - Ride retrieval
  - Ride information update
  - Ride deletion
- Booking Functionalities (a.k.a Lift Functionalities)
  - Search for any available lift based on condition
  - Book a Lift (reservation)
  - Delete a Lift (cancellation)

Driver and Car Functionalities are basically standard CRUD operations, while Rides and Lifts involves a more complex logic, as stated before the Orchestrator offers the external endpoints for the requests, but then it processes those requests in a way to ensure coherence and non ambiguity and forwards those processed requests to the submodules of interest (Planner, Anagraphics back end). For example in case of a driver or a car creation, update or deletion it might not be necessary to interact with all the submodule, but only with the anagraphics backend, While when creating a ride, for example the steps are more complex such as:

1. Retrieving from backend unique informations about the Driver and his car (id's)
2. Forwarding to the planner module the parameters useful for planning (such as the starting and ending points, departure date etc), associated with thouse unique ids

Contract No. 881825

3. Planners compute a route based on the maps its configured with, create a trip plan in its internal structures referencing those unique id's at point 1, and respond to the Orchestrator with the computed plan
4. Orchestrator stores the plan in the backend for safety and backup purposes
5. Orchestrator processe the plan, extracting the necessary informations computes a Ride object and stores it into the backend

At this point a ride has been created, it has been associated to a plan in the trip planner and it is available for the search algorithms

When searching for a lift:

1. Orchestrator receives the request parameters, computes them and create a specific request for the Trip Planner
2. Trip Planner searches into his data structures available plans that suits the requests and send them back to the Orchestrator
3. Orchestrator receives these plans, processes them and formats them as Lift objects (at this point every plan is associated with a temporary lift object, uniquely associated to a Ride, and consequentely to a Driver and a car)
4. Orchestrator responds to the external request with a list of available possible lifts

If then an external system uses one of those object received, in the booking reservation endpoint, an association will be created between that lift and the relative ride, and it's persisted onto the backed.

At that point there will be:

- A Ride with a lift associated to it
- A Lift associated with a Ride
- A plan in the Planner associated with the Ride

NOTE: The CbTSP interacts with the AL, so during Lift reservation or Lift cancellation, the correspondent objects on the AL gets updated consequently

More information about the CbTSP API can be found in the specific documentation hosted in the project Github repository at this provided link: Ride2Rail/CbTSP-r2r-Docker: Repository for CbTSP docker stack (github.com)

4.4.2. Technical and implementation details

The whole component has been designed in house, using Java and Spring Framework functionalities, it communicates internally with the other components of CbTSP using REST API and exposes REST API to the external systems. The Orchestrator and all the other components are available as a stack of docker containers with all the endpoints already configured and editable as needed, more information can be retrieved at the official CbTSP docker repository: Ride2Rail/CbTSP-r2r-Docker: Repository for CbTSP docker stack (github.com)

Contract No. 881825

4.5. Trip Tracking

4.5.1. General overview of the service

Purpose

The Trip Tracking Service (TT) is a software responsible for providing a way to track a ride during its execution and inform interested third-party components in its progress. After a route has been planned through other components in the Ride2Rail ecosystem, the driver can activate its tracking from the Driver Companion Application which will then communicate with the Trip Tracking Service in the background to complete this request.

Once this phase has been dealt with, the Driver Companion can now send periodical position updates to the TT for the entire duration of the ride or until its cancellation. These updates will be used to compute an estimation of how much time the driver has left to reach their destination, and inform the Tracking Orchestrator in case a signiﬁcant diﬀerence is measured between the time to get to a destination and the time estimated during the planning phase of the ride, which likely means a delay or disruption of the ride has been observed.

This tight communication structure between diﬀerent components of the R2R ecosystem can be seen in Figure 2: here the main actors of the above scenario are shown. As an side note, the picture also introduces the internal structure of the TT itself by depicting the service divided into two separate modules based on their purpose: the Ride Progress and the Subscription module.

The approach is to maintain those modules separate because of their not overlapping goals which facilitates horizontal scaling of the software as each service can be replicated to increase the overall throughput of the requests directed towards the TT. To support this, a number of diﬀerent decisions have been made and will be presented in details in the following sections.

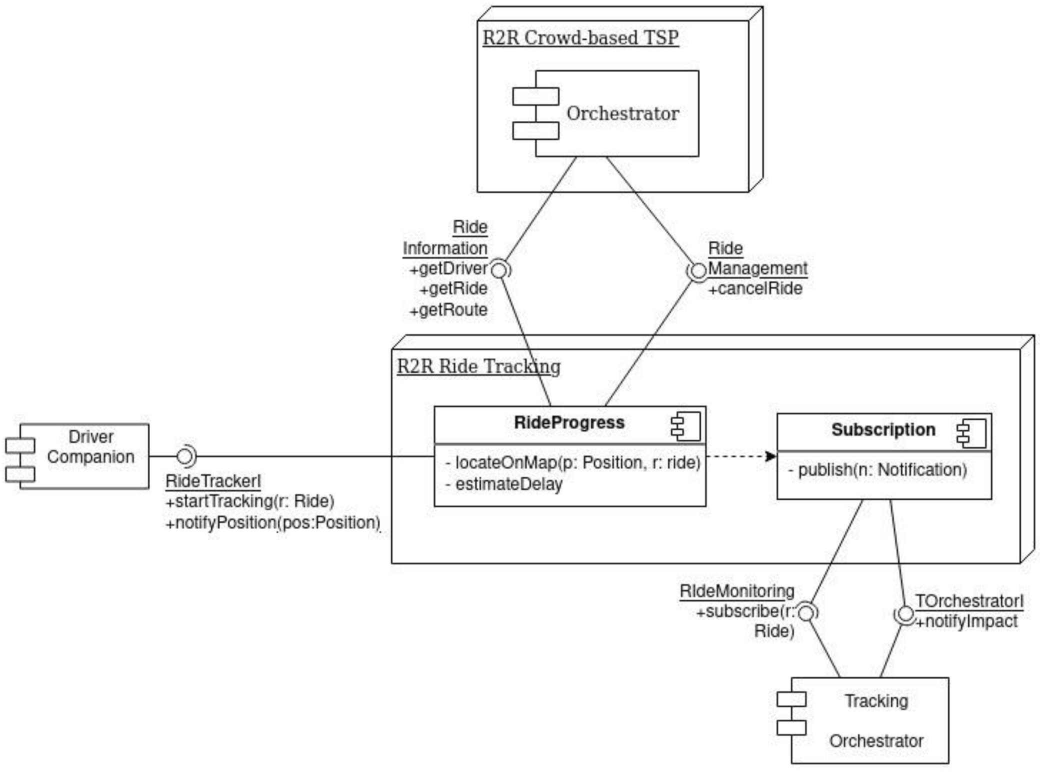


Figure 3 Component diagram in Ride2Rail ecosystem

Interaction within the ecosystem

Before describing the individual components the TT consists of though, let us show how the service in its entirety is supposed to interact within the existing R2R ecosystem of components. Figure 3 encompasses this interaction in the flow of requests made by the individual pieces shown earlier. As displayed here, the TT makes use of both direct and indirect communication exchanges with the other elements of the ecosystem. This is crucial to detect possible bottlenecks that might slow the application down during peak service times. To better accommodate an increase of incoming request, the communication within the service has been made asynchronous, but this is not possible for some outgoing requests to the R2R ecosystem components, like the CbTSP, since their response is needed before process can go further.

4.5.2. High level component description

Ride progress component

The ride progress component deals with everything related to a trip monitoring. It registers requests to track a specific ride, incoming position updates for them that will be sent in by a Travel Companion during a ride execution. The component relies heavily on

Contract No. 881825

the communication with the CBTSP to obtain information about routes drivers choose to follow, and all of their intermediate stops and checkpoints. This is crucial for the correctness of delay estimation and its subsequent delivery to the TO. Without it, the Ride tracker cannot carry on its purpose.

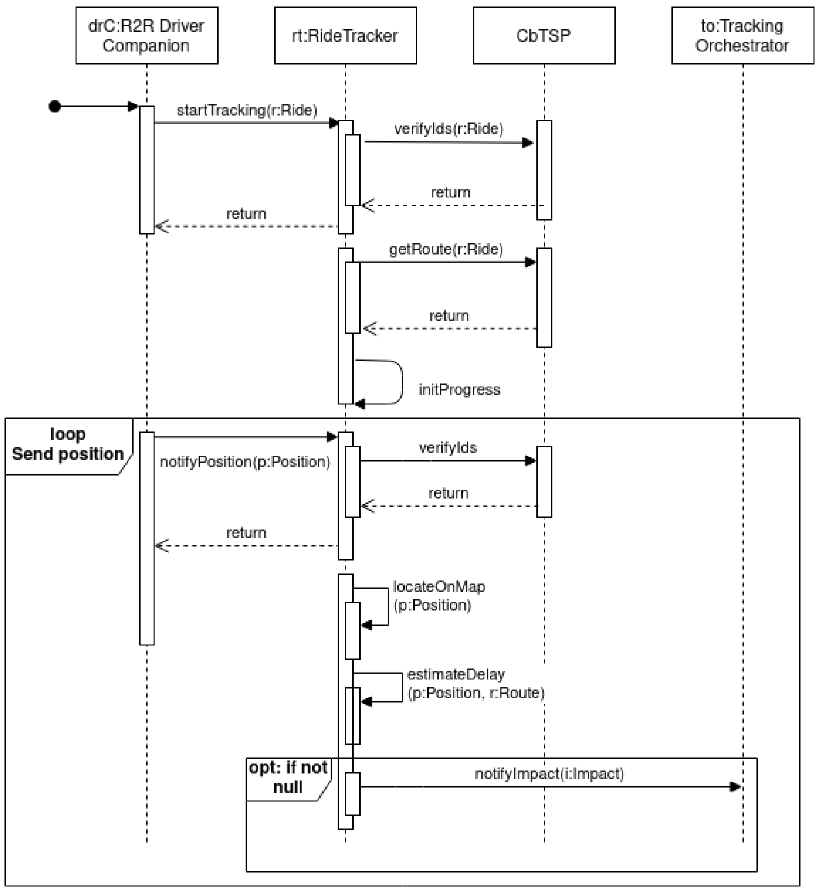


Figure 4 Detection system diagram

Main Functionalities

- Managing requests to begin the tracking activity for a ride. When such a request is made to the RT, this module verifies with the CBTSP that the information necessary to identify the requester, i.e. DriverId and RideId, are related to a registered entity, it saves them in a transient storage, and asks the CbTSP for a ride object associated with them. This object contains a planned route with all the intermediate stops the driver decided to make on the way, and a time estimate for when every stop will be reached.
- Accept and process a position update from the driver's travel companion. Once the tracking process has been requested, the RP is ready to accept positional information related to a driver. This information contains but a pair of geographical

Contract No. 881825

coordinates, a timestamp to attest when the position has been registered, and the driver’s and ride’s identification numbers.

• The request is then used to locate the driver’s vehicle on the route, and subsequently estimate a possible delay by comparing its last known position with the time estimates received for this route from the CbTSP.

Subscription Component

The Subscription component gathers information about all actors interested in receiving notifications when a delay occurs during a ride execution. It fills the role of a partial Trip Tracker (pTT) for the TO. Whenever a delay on a particular ride is measured by the Ride Progress module, it is communicated through the internal message bus to this component that takes care of dispatching an HTTP message to all the interested parties. In this context, the notification is wrapped in a structure compliant with the TRIAS specification required by the current TO implementation, but changing it to a different format is also possible in the future.

Main Functionalities

- Managing subscriptions to a particular ride execution. The information is directly coming from the TO, and it must include an identifier for the component interested in delay notifications, an identifier for the ride said component is interested in, and an address in URL format that represents the destination to which the message must be sent to. This iteration of the software only deals with HTTP/s protocol for the publishing endpoint required by this component, but later modifications to augment its capabilities are possible.
  These subscriptions are saved in the local storage of this component for further use, modification or deletion.
- Publish the delay notification estimated by the RP. Whenever the RP determines a delay in the progress of a particular ride, it sends a message to this component that wraps the information in TRIAS-compliant notification, retrieves all the addresses of subscribers interested in the aforementioned ride, and sends the message to them via an HTTP request.
  Restrictions and limitations imposed by the TO implementation made it so that this is a best-effort process: the notification is sent by what can be described as a ”fire and forget” mechanism, as the Subscription component will not track unsuccessful dispatches.

4.5.3. Technical implementation of the service

This section aims to showcase the Ride tracking service in technical detail to better present its internal workflow and choices made during its design. Figure 4 depicts the current situation of TT architecture. A quick summarization of the different parts comprising the software is as follows:

- Two ingress APIs
    - Subscription API

    - Trip Tracking API
- A message queue for incoming request
- RideProgress module with a persistent storage
- Subscription module with a transient and a persistent storage.

Each of these main items is discussed in details in the upcoming sections.

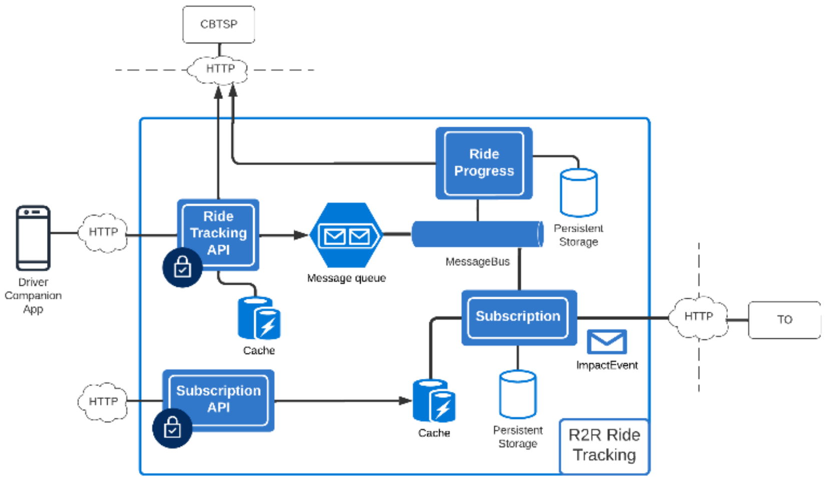


Figure 5 Current implementation of Ride tracking service

4.5.4. Ingress API

This is the initial layer of the service. It takes care of the incoming messages from an external source, the Driver Companion for instance, to perform basic filtering, and authentication functions. It also provides a first caching structure for faster response times and message de-duplication. In the diagram shown in Figure 4 messaging is limited to HTTP communications only, but adapters can be put in place later to accept events from a variety of sources. Although, as apparent from the aforementioned implementation the purpose of this layer is slightly different depending on the requested interaction.

In the case of the Ride Tracking API, in addition to the previous points, its purpose is also twofold: it acts as a validation mechanism for the incoming information by asking the CbTSP, which owns them, about rides, and provides an initial checkpoint for the input message flow. After being verified and validated, incoming requests are put in a processing FIFO queue so that a higher demand, during rush hours for instance, will be more manageable without slowing the application's consumption throughput.

This, however, was deemed unnecessary for the Subscription API which is supposed to provide much lower throughput of messages, since it is dealing only with requests before the ride tracking process starts, or after it is completed, thus allowing it to have a direct communication channel with its core module without the need of a processing queue in front of it.

Moreover, the presence of a message ingestion queue and the interaction between modules makes the process guaranteed but asynchronous, which means the Driver Companion, or other components using this service will only know that their requests have been registered in the system but will not know when they will be taken care of.

4.5.5. Ride progress module

This is one of the two core modules of the service. It is made of multiple components and relies on the communication structure provided by the Message Bus. It is interested in all those events that have something to do with the ride management: its creation, a position update or a cancellation request. While the other two are trivial, a position update event triggers a chain of computations that require some more explanation.
Once the process of tracking has been requested for a particular ride, the Ride Progress module creates a persistent entity to associate all the following position updates that the Driver Companion application would send regarding this ride. Additionally, as said earlier, it needs to ask the CbTSP for details regarding the route the driver wants to follow, including all the intermediate stops, if any, the car will make along the path. This information is the bare minimum to provide a delay estimate for the ride. Although, the more intermediate points there are, the more precise the estimation can be. In case none were selected by the driver, only the starting point and the destination will be considered.

Each time a new position for a particular ride comes in, the RP computes a rough estimate of the distance traveled between the two currently considered checkpoints: the one the driver has already passed and the immediate next they have not reached yet.Once this is done, the RP estimates when the driver should have reached the current location based on the information included within the route request it made earlier to the CbTSP, as it must contain estimated time of arrivals too. If this estimate is greater than a pre-configured threshold, a delay event is triggered signaling a possible disruption of the ride.
There also are scenarios in which the computation cannot be made. If the cause is the impossibility to correctly locate the driver’s vehicle on the map, the RP will send an additional request to the CbTSP asking to recompute the route for the ride at issue. In the extreme case of the total impossibility to complete the evaluation, the update event is discarded completely and the following one will be used as the new base case for comparisons.

4.5.6. Subscription module

This is the second core module of the service. Compared to the previous one, it is more straightforward as its main purpose is to manage subscriptions requests and dispatchevents to third parties that are interested in them.
The API in front of it takes care of handling the requests to add or remove subscriptions of those components that are interested in receiving notifications about a particular trip disruption. A write-through cache has been added on top of the existing persistent storage since the number of requests to read the contents of this component are greater than the possible requests trying to modify it. Also, fewer requests with respect to the RP are

expected in general, thus limiting the need of putting a message queue in front of it, as mentioned earlier.
A consumer process works in background listening to disruption events delivered by the Message Bus. Once such an event is caught, the process will retrieve the necessary information about third parties interested in it from the persistent storage, and will send the notification in the pre-configured TRIAS format the TO requires. The Subscription module has no information about third parties other than the URL address and some identification number it received at the moment in which a subscription was first created. This step is thus a best effort process mainly due to the TO constraints. Any failures or undelivered notifications are thus ignored by this component.

4.6. Asset Manager

The Shift2Rail IP4 ecosystem is a distributed environment, developed and maintained by several companies. Such companies, while integrating TSPs into the ecosystem, need to share consistent and structured descriptions of such services. This is the main requirement which led to the development of the so-called "Asset Manager", an architectural element inside the IP4 ecosystem playing the role of a shared catalogue of digital artifacts. The Asset Manager has been first implemented during the IT2Rail[3] project, then the SPRINT[4] project explored how to exploit such catalogue to improve automation, providing support for data converters and service mediators and supporting complex publication processes involving staff permissions to publish and mail notifications.

The Asset Manager has been evolved by Cefriel into KCONG (Knowledge Catalogue aNd Governance) as a prototype to support digital ecosystems and data fabric architectures, and a new version of the Asset Manager has been developed to support the specific needs of Shift2Rail IP4 members.

The Asset Manager deployed in the context of Ride2Rail project fulfills the main requirement of providing a reference environment to store service documentations coming from different open-call projects and transport service providers, reducing the needs of sharing information via emails and therefore reducing the risk of a partner not having up-to-date information while doing an integration task.

The main interface of the Asset Manager (displayed in Figure 5) allows to gain access to structured descriptions belonging to two asset types:

- Ontologies: reference conceptual data models developed in the context of Shift2Rail IP4

[3] https://cordis.europa.eu/project/id/636078/it
[4] Home (sprint-h2020.eu)

- IP4 Services: services provided by IP4 members and partners as well as services provided by Transport Service Providers

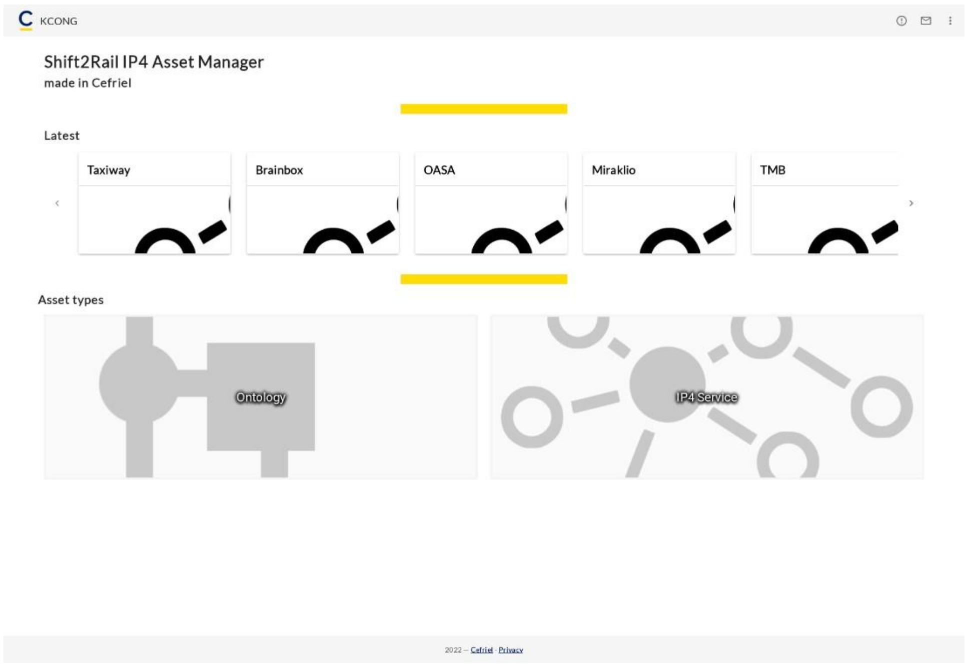


Figure 6 Asset Manager main screen

When accessing an asset type, the interface shows the list of available assets and a short list of the latest published assets, as shown in Figure 6.

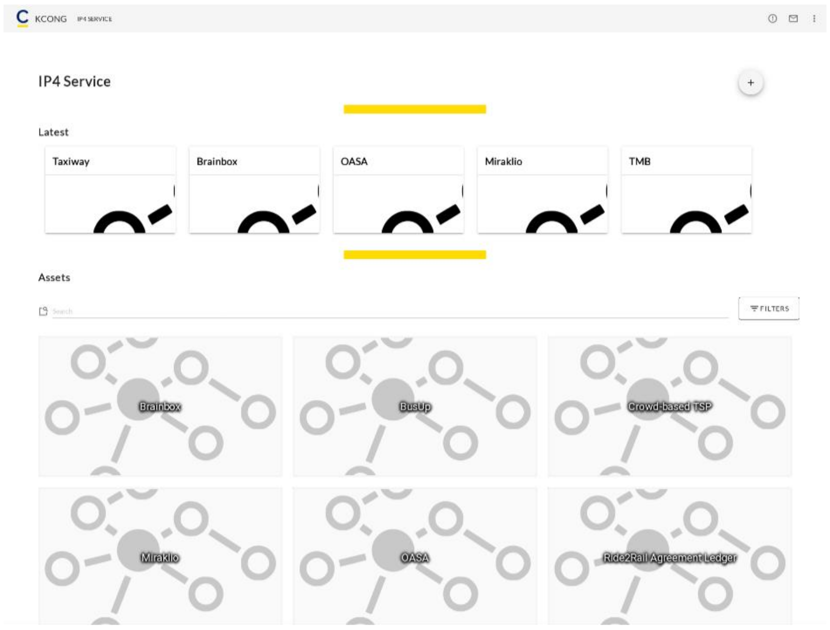
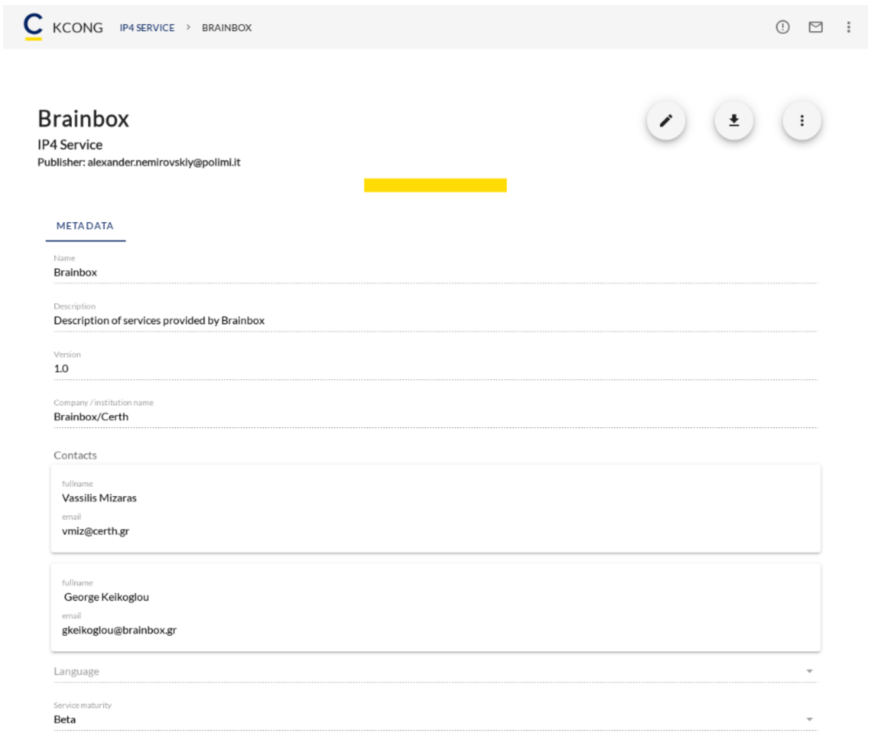


Figure 7 Asset Manager: listing IP4 Services

Contract No. 881825

Pressing the "+" button allows adding a new asset, while clicking on an item allows inspecting the description of an asset, as shown in Figure 7.


KCONG IP4 SERVICE > BRAINBOX

Brainbox
IP4 Service
Publisher: alexander.nemirovskiy@polimi.it

METADATA

Name
Brainbox

Description
Description of services provided by Brainbox

Version
1.0

Company / institution name
Brainbox/Certh

Contacts

fullname
Vassilis Mizaras
email
vmiz@certh.gr

fullname
George Keikoglou
email
gkeikoglou@brainbox.gr

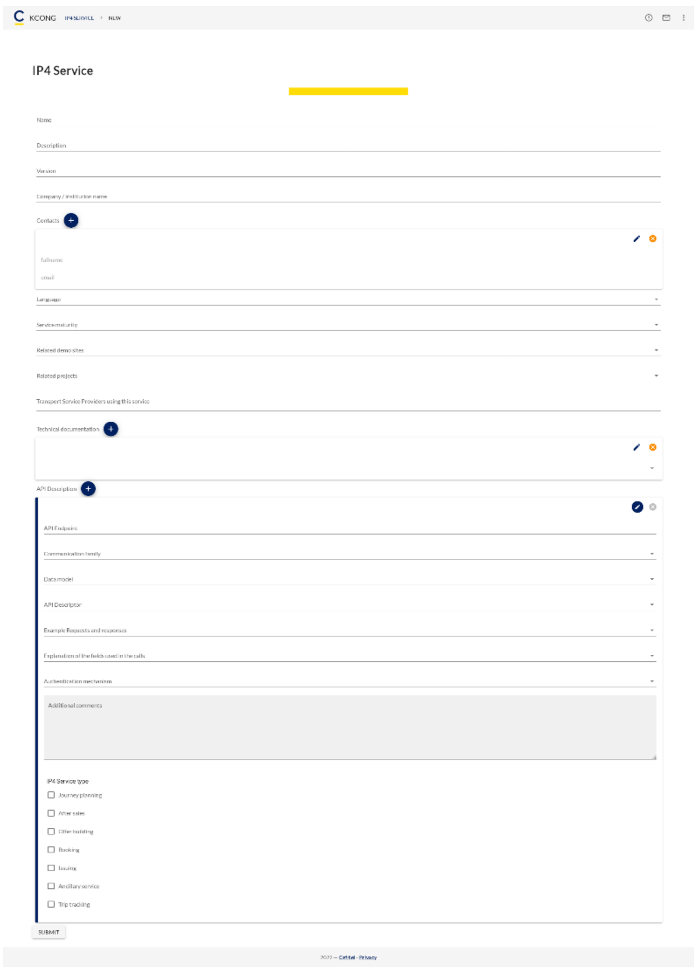
Language

Service maturity
Beta

Figure 8 Asset manager: IP4 Service description

4.6.1. IP4 Service descriptor

Users can insert descriptions of IP4 Services by filling a dynamic form provided by the Asset Manager and shown in Figure 8.



KCONG IP4 SERVICE › NEW

IP4 Service

Name

Description

Version

Company / institution name

Contacts

fullname

email

Language

Service maturity

Related demo sites

Related projects

Transport Service Providers using this service

Technical documentation

API Description

API Endpoint

Communication family

Data model

API Descriptor

Example Requests and responses

Explanation of the fields used in the calls

Authentication mechanism

Additional comments

IP4 Service type

- ☐ Journey planning
- ☐ After sales
- ☐ Offer building
- ☐ Booking
- ☐ Issuing
- ☐ Ancillary service
- ☐ Trip tracking

SUBMIT

2022 — Cefriel - Privacy

Figure 9 Asset Manager: capturing structured description of an IP4 Service

A brief explanation of the meaning and expected values of the fields in the form is explained below:

- Name: name of the service
- Description: textual description of the service providing hints about its functionalities
- Version: service version or version number of the description of the service
- Company/institution name: company owning the service (it may be different from the TSP)
- Contacts
    - Full name: full name of the contact person
    - Email: email address of the contact person
- Language: language used in the service
- Service maturity:
    - Planned: the service is only planned, so all the provided documentation is intended only for future integration
    - Beta: the service is being tested and improved, it may not be reliable
    - In production: the service is ready, stable and can be accessed reliably

Contract No. 881825

    - Deprecated: the service is being dismissed
    - Dismissed: the service is not active anymore
- Related demo sites: list of demo sites where the service is being used
- Related projects: list of projects where the service is being used
- Transport service providers using this service: list of Transport service providers using the service.
- Technical documentation: a list of files providing a general documentation for the service
- API Descriptions
    - API Endpoint: URL of the API endpoint
    - Communication family: technology used to exchange data with the endpoint
        - SOAP
        - REST
        - Web API
        - FTP
        - GraphQL
        - MQTT
        - AMQP
        - Kafka
        - Other
    - Data model:
        - Standard: specification/standard whose data model is used by the service
        - Local ontology: ontology already listed in the catalogue providing the data model for the service
        - Remote ontology: URL of an ontology (not already listed in the catalogue) providing the data model for the service
        - Custom/proprietary: the data model is custom/proprietary and doesn't refer to any existing standard or ontology
    - API Descriptor: URL of the Swagger/OpenAPI or WSDL or any other structured description of the API
    - Example requests and responses: this field allows stating where examples of the requests and responses can be found. The options are:
        - Requests and responses are already described in the technical documentation
        - Requests and responses are already described in the API descriptor
        - Upload a document containing the examples
    - Explanation of the fields used in the calls: this field allows providing an explanation of the parameters to be used when invoking the API. The options are:
        - API parameters are already described in the technical documentation
        - API parameters are already described in the API descriptor
        - Upload a document containing the parameters description
    - Authentication mechanism: this field allows describing the authentication process to be followed to access the API functionalities. The options are:
        - Authentication instructions are already provided in the technical documentation
        - Authentication aspects are already described in the API descriptor

        - Upload a document containing authentication instructions
    - Additional comments: a free-text field which allows documenting other aspects of the interaction with the API not already covered by the other fields
    - IP4-Related information: Shift2Rail IP4 identified a list of functionalities to be provided by the TSP services to be integrated. For specific functionalities IP4 must obtain additional information.
        - Journey planning
            - Supported transport modes: list of transport modes supported by the TSP via this API
            - GTFS URL: URL of the GTFS file providing the TSP's timetables
            - GTFS Upload: as an alternative, the user can directly upload the GTFS dataset
            - GTFS validity: temporal validity of the GTFS dataset
            - GeoJSON coverage area: coverage area of the service being provided by the TSP
        - After sales
        - Offer building
        - Booking
            - Booking expiration time: this field allows stating for how many seconds a booking offer will remain valid
        - Issuing
            - Ticket type
                - QR code
                - Image
                - PDF
                - Other
            - Validation comments: additional information concerning how the passenger is expected to validate the ticket
        - Ancillary service
        - Trip tracking

4.6.2. IP4 Service lifecycle process

The Asset Manager, as introduced in the IT2Rail project, can manage the entire lifecycle of the description of an asset. Starting from the SPRINT project such functionality is implemented by specifying an executable BPMN process, which orchestrates the internal services provided by the Asset Manager and potential external services, and allows stating when a user is expected to provide decisions in order for the publication process to be completed.

The publication process required by the Shift2Rail IP4 members, shown in Figure 9, is centered around the concept of notifying via email a group of people (the *IP4 Staff* members) whenever an asset is added to the catalogue or modified. The *IP4 Staff* also has the permission to modify any asset description.

Contract No. 881825

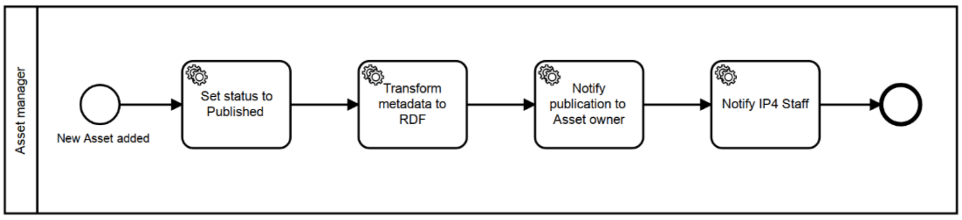


Figure 10 Asset Manager: IP4 Service publication process

CONCLUSIONS

In this deliverable, the results of Task 3.3 (T3.3) – CbTSP formally known as Crowd Based Travel Service Provider, in the context of WP3 of the RIDE2RAIL (R2R) project, have been described.

A set of components that work together in order to reach the goal of acting as a Travel Service Provider when seen from the outside has been developed; the behaviour is similar to a general TSP, there are functionalities exposed that allows whichever entity that integrates with it that acts as a "traveller" to search for travel solutions inside the context of this specific TSP and there are functionalities that offer whichever entity that acts as a "driver" to plan routes and publish travel solutions for the travellers to search.

There is a core difference between a classical TSP and the CbTSP: on "HOW" the available travel solutions are produced. In a standard TSP (i.e. the service of a city bus line), travel solutions are severely planned over a medium/long period of time. Meaning that for example at the start of the year, all the rides of all the lines, at every given time are planned, it's then only a matter of matching an incoming request including departure, arrival and time, in order to answer with a set of available rides. In the CbTSP does not work in this way: there is not any option to know in advance if there will be an available ride matching some criteria at a given time. People that decide to offer a ride are responsible for dinamically inserting rides into the system based on their willingness.

Every driver entity can insert, delete or modify new rides whenever they want, and so the traveller entities may find very different sets of solutions, depending on the type and quantity of rides offered at the specific moment in time they do the search. In short, the set of possible travel solutions that the CbTSP can offer to a general "traveller" is dynamically created and modified in correlation to the actions done by their "driver" counterparts.

The task was meant to implement a piece of software that presents itself as a TSP, and acts like one. i.e anyone can ask the CbTSP to give all the travel solutions that are availeble, that can allow a passenger to depart from a point A, arrive at a point B starting from a said departure time on, just like any general TSP. Those travel solutions however don't come from an "a priori" planning, but are evaluated based on offerings of rides given by specific users of the system (identified as drivers) who decides to share their car in a planned route of their own (carsharing logic). Those offered rides are "born" dynamically into the system based on the interaction of the drivers

The CbTSP can be seen as a special hybrid type of TSP which merges together the functionality of a common TSP, plus the functionality of a carsharing service in order to achieve this goal.

When a "driver" entity decides to insert a ride, it asks the system to plan a route between a starting point and a destination point, the planner component computes the solution and adds it to its set of available travel solution, at this point a new ride has been created with its associated route plan, which includes also the starting date and time. If subsequently the system is asked to find a solution for a traveller, and the search criteria matches the properties of the ride above among others, then that ride will be included in the set of travel solutions proposed to the traveller to choose from; the traveller then choses the

Contract No. 881825

wished solution and asks the system for reservation of a lift on that ride, at this point the association is completed and a traveller, a driver and a ride are correctly associated (basically the passenger has reserved a seat in a car offered by a driver).

Due to its dynamic nature, travel solutions are held in memory, and also the geographical data to speed up computations, this implies that the memory footprint of the CbTSP, especially of the planner component is directly correlated to the size of the geographical area on which it operates.

At the current state the components, and in general the solution provided covers all the objectives required by the WP3 - task 3.3.

It would be pleasant in the future to consider the possibility to research and develop a distributed version of the planner component in such a way that every instance takes care of a specific geographical area in a way that when there is a need to compute a travel solution, it can be composed by the sub-solutions of every instance involved in the computation.

Contract No. 881825

5. REFERENCES

- OpenTripPlanner
- Ride2Rail/CbTSP-r2r-Docker: Repository for CbTSP docker stack (github.com)
- Alquiler de coches particulares - SocialCar
- https://spring.io/