Working with the SynchroNET Planner

This document will give you useful instructions to work with the *Synchro-NET Planner* (SNETP) and is organized as follows. Section 1 gives a brief overview of the Synchro-net project and the main architecture in which the planner is included. Section 2 presents the OpenTripPlanner (OTP), a collection of libraries on which SNETP is based. In particular, the main data formats used (OSM and GTFS) are presented, and instructions to work with the source code of the OTP are provided together with the explanation of the most useful packages concerning the routing algorithms and the user interface. Section 3 presents the main components of the SNETP and provides the details on the data collected and the customization of the code already done. It also explain how the planner communicates with the other optimization modules (real-time optimizer, risk evaluator, ...). Section 4 gives the instructions to connect to the server on which the planner runs (at least in its debugging version) and explicit the content of its storage. Section 5 lists some future works needed to accomplish the project requirements.

**Sommario**

[1 Project overview 2](#_Toc476577056)

[1.1 Project architecture 3](#_Toc476577057)

[1.2 SynchroNET Planner (SNETP) 5](#_Toc476577058)

[2 OpenTripPlanner (OTP) 7](#_Toc476577059)

[2.1 Data standards 7](#_Toc476577060)

[2.1.1 Open Street Map (OSM) 7](#_Toc476577061)

[2.1.2 General Transit Feed Specification (GTFS) 7](#_Toc476577062)

[2.2 Working with the OTP 7](#_Toc476577063)

[2.2.1 Running the OTP 8](#_Toc476577064)

[2.2.2 Working with the OTP source code (by using Eclipse) 9](#_Toc476577065)

[2.2.2.1 Importing the OTP project 9](#_Toc476577066)

[2.2.2.2 Build the .jar file 10](#_Toc476577067)

[2.2.2.3 Build and run OTP in Debug Mode 10](#_Toc476577068)

[2.3 Main OTP packages 12](#_Toc476577069)

[2.3.1 Graph generation and routing 12](#_Toc476577070)

[2.4 Web user-interface 12](#_Toc476577071)

[2.4.1 Web services 14](#_Toc476577072)

[2.4.1.1 Simulating a request from the user-interface 14](#_Toc476577073)

[3 The Synchro-net Planner implementation 15](#_Toc476577074)

[3.1 Data gathering, generation, and filtering 15](#_Toc476577075)

[3.1.1 GTFS\_Batch: a GTFS routes generator 16](#_Toc476577076)

[3.1.2 Obtaining the file of the locations 17](#_Toc476577077)

[3.1.3 Obtaining the OSM data for trucks and trains 18](#_Toc476577078)

[3.2 Software customization and development 18](#_Toc476577079)

[3.2.1 GTFS input feeds 18](#_Toc476577080)

[3.2.2 Routing optimization 19](#_Toc476577081)

[3.2.3 Working with schedules 20](#_Toc476577082)

[3.2.4 Some important parameters 21](#_Toc476577083)

[3.2.5 User interface options and visualizations 21](#_Toc476577084)

[3.2.5.1 Set origin-destination coordinates 21](#_Toc476577085)

[3.2.5.2 New input parameters 22](#_Toc476577086)

[3.3 Interaction with other software modules 23](#_Toc476577087)

[3.3.1 Interaction with the Risk Analysis and Optimization (RAO) module 23](#_Toc476577088)

[3.3.2 Interaction with the Real-Time Optimization module 23](#_Toc476577089)

[4 The SNETP storage on the ICE server 23](#_Toc476577090)

[4.1 How to connect to the server 23](#_Toc476577091)

[4.2 The storage of SNETP on ICEsrv 23](#_Toc476577092)

[5 Future developments 24](#_Toc476577093)

[5.1 Include Risk into the list of KPIs 24](#_Toc476577094)

[5.2 User preferences 24](#_Toc476577095)

[5.3 Accounting and authentication 25](#_Toc476577096)

[5.4 Database and search engine for historical decisions 25](#_Toc476577097)

[5.5 GTFS\_Batch for generating synchronous routes 25](#_Toc476577098)

# Project overview

SYNCHRO-NET (Synchro-modal Supply Chain Eco-Net) is an ongoing project funded by the HORIZON 2020 Programme of the European Commission under the Grant Agreement No. 636354. The project aim is to demonstrate how a powerful and innovative SYNCHRO-modal supply chain eco-NET can catalyze the uptake of the slow steaming concept and synchro-modality, guaranteeing cost-effective robust solutions that de-stress the supply chain to reduce emissions and costs for logistics operations while simultaneously increasing reliability and service levels for logistics users. The core of the SYNCHRO-NET solution will be an integrated optimization and simulation eco-net, incorporating: real-time synchro-modal logistics optimization (e-Freight-enabled), slow steaming ship simulation & control systems, synchro-modal risk/benefit analysis statistical modeling, dynamic stakeholder impact assessment solution, and a synchro-operability communications and governance architecture. Perhaps the most important output of SYNCHRO-NET will be the demonstration that slow steaming, coupled with synchro-modal logistics optimization delivers amazing benefits to all stakeholders in the supply chain: massive reduction in emissions for shipping and land-based transport due to modal shift to greener modes AND optimized planning processes leading to reduced empty KMs for trucks and fewer wasted repositioning movements.

## Project architecture

To achieve the above requirements, SYNCHRO-NET relies on service-oriented and workflow technology to implement an integral platform (see D2.4). This platform allows logistic networks to communicate with the platform as needed in a highly granular fashion and to easily interact with other logistics stakeholders. The platform employs the cooperation and the coordination of different logistic modules, allowing integration with external services using standardized and open APIs. The aim is to develop or enhance all the e-Freight tools and smart coordination mechanisms that allow a complex synchro-modal supply chain to be planned and optimized effectively, both at the strategic level and in real-time. Figure 1 shows a simplified architecture of the logistic modules involved and their connections.

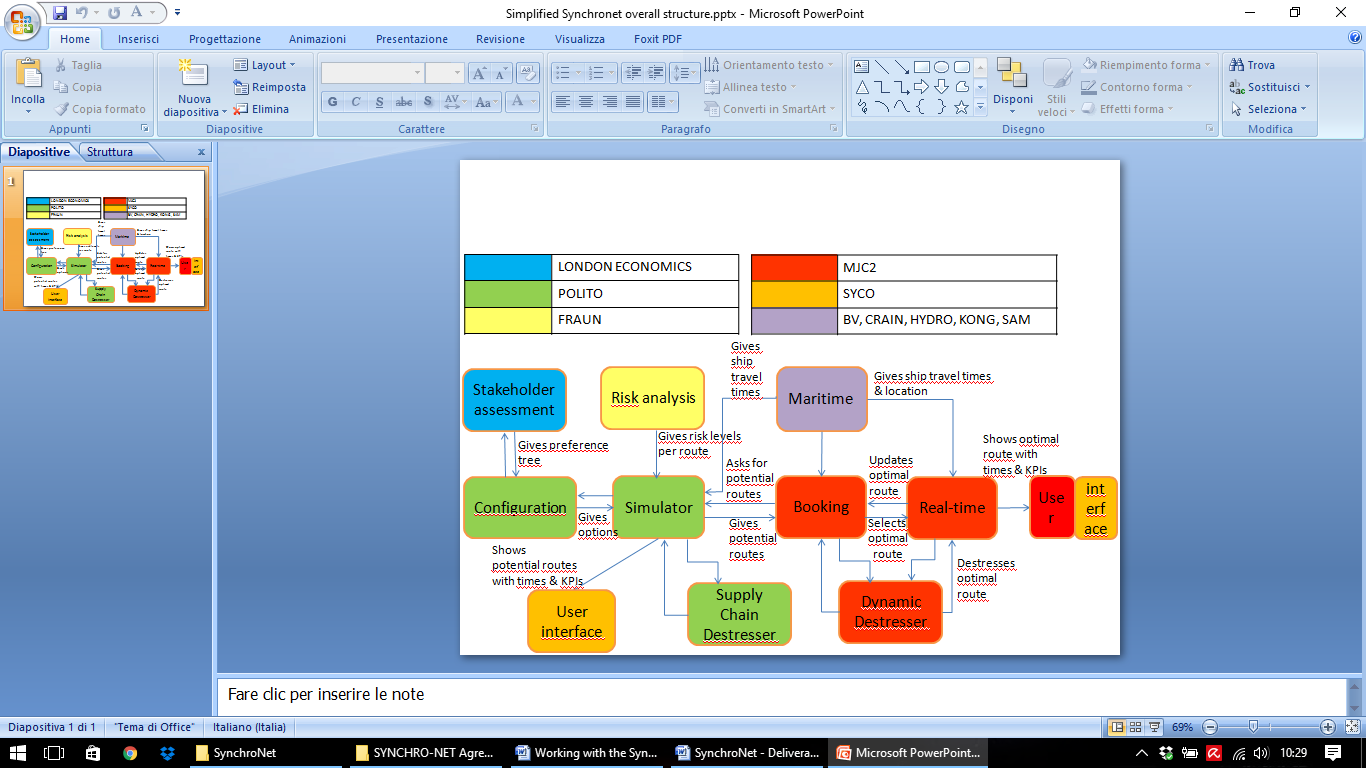


Figure Simplified architecture and logistic modules interaction

More technically, the optimization process involves three main modules working in a cooperative manner: Real-Time Synchro-Modal Optimisation (RTO), dynamic Supply Chain De-stressing (SCD) and Risk analysis and optimization (RAO). Figure 2 illustrates the main interactions among the three optimization modules.

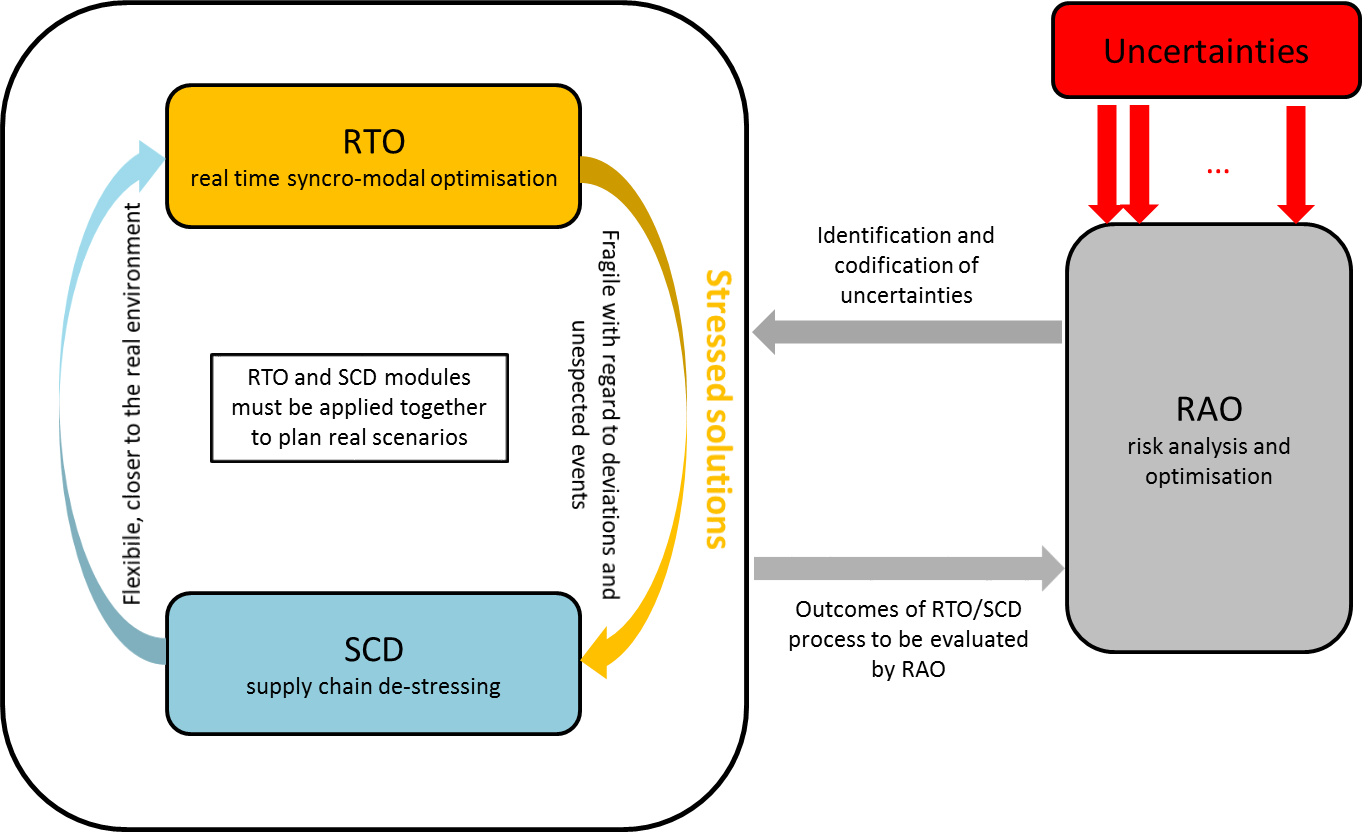


Figure Main interaction between SCD, RAO and RTO modules

The RTO defines stressed solutions for the entire system and passes them to the SCD. The SCD then cooperates with the RAO to de-stress solutions using different objectives based on Key Performance Indicators (KPIs) and Key Risk Indicators (KRIs) evaluation. Some examples are minimum delivery time, minimum total cost, minimum total emission, trade-off between risk of delay and cost saving where real time information and unexpected events are taken into account.

The aim of the SCD module is to de-stress the synchro-modal transport scheduling solutions by cooperating with RTO and RAO. The SCD module consists of two components: 1) De-stresser database and search engine (database storing information on historical decisions, e.g., routes, modes, companies, so that it could be used as references for future requests by the users), and 2) Supply chain de-stresser (provide de-stressed solutions, taking into account the KPI and risk evaluation). Figure 3 shows the data flow between the modules.

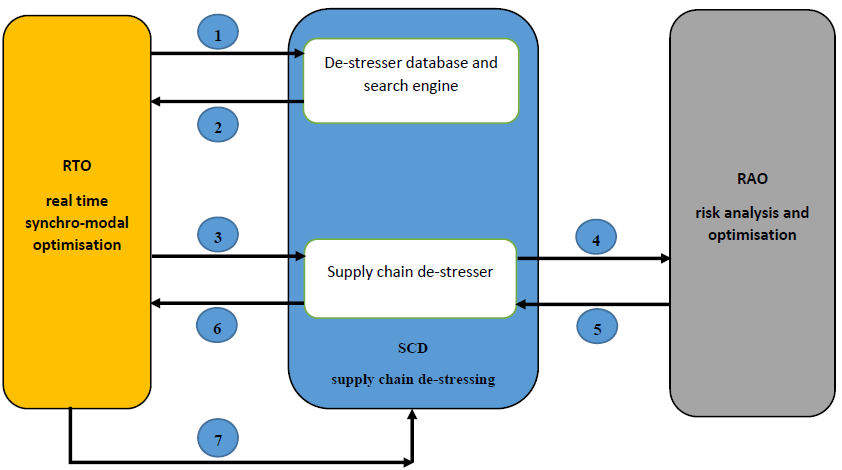


Figure Data flow between SCD, RTO and RO modules

More precisely:

1. The user creates a request to the system (e.g., sending freight from location A to location E and other specifications such as user preferences) and the RTO sends this request to “De-stresser database and search engine” component.
2. The “De-stresser database and search engine” component responds with a list of *n* possible routes A-B-C-D-E. For each link of each route, it will include the actual haulier/service used previously.
3. The RTO elaborates these *n* routes and reduces them to *m* routes which are feasible, taking into account the actual service time/schedules. It then passes these *m* routes (including details about timings of each leg) to “Supply chain de-stresser” component.
4. The “supply chain de-stresser” component calculates the “scores” for each *m* routes based on the key performance indicators (KPIs) (e.g., minimum of the delivery time, minimum of total cost, minimum of total emission). It also sends them to the RAO for risk analysis.
5. The RAO determines key risk indicators (KRI) (e.g., expected time deviation (number of additional hours), expected cost deviation, probability for damage/loss (%), flexibility rate) for each relevant route provided by the “supply chain de-stresser”. Each route is rated on its behavior regarding various disruptions during the execution of a route. The RAO will then pass back KRIs for each of the relevant routes.
6. The “supply chain de-stresser” component sends the “scores” defined by KPIs and KRIs for each of the relevant route back to the RTO.
7. The system (the RTO) shows a list of possible multi-modal routes, along with KPIs and KRIs (cost, risk, environmental), and the user decides on the preferred route. They then will confirm the actual haulier company or rail service or ship company/voyage to be used and agreed rates. The RTO will pass this confidential information to the SCD module so it could be used for future suggestions to this user, making the quoting and booking process faster.

## SynchroNET Planner (SNETP)

The **SynchroNET Planner** (**SNETP** in the following) is the software platform on which the SYNCHRO-NET project is based on. Basically, SNETP implements the SCD optimization functionalities, together with the API interfaces for interacting with the RAO module. Moreover, the presentation layer of SNETP (the user interface) will provide the basic interacting system for the RTO.

SNETP is structured as a client-server application. Figure 4 depicts the main components and the basic architecture of SNETP. As shown, the core of SNETP consists of 3 basic components:

* a **graph builder**: elaborates geo-spatial and transit input data and creates a "logical" and more general graph upon which quantitative analysis are possible;
* a **routing engine**: able to find one (or more) possible feasible route from a node to another node in the graph and to evaluate it in terms of quantitative parameters;
* a **user interface**: gives to the user the possibility to interact with the planner, in order to find the best solution for its needs.

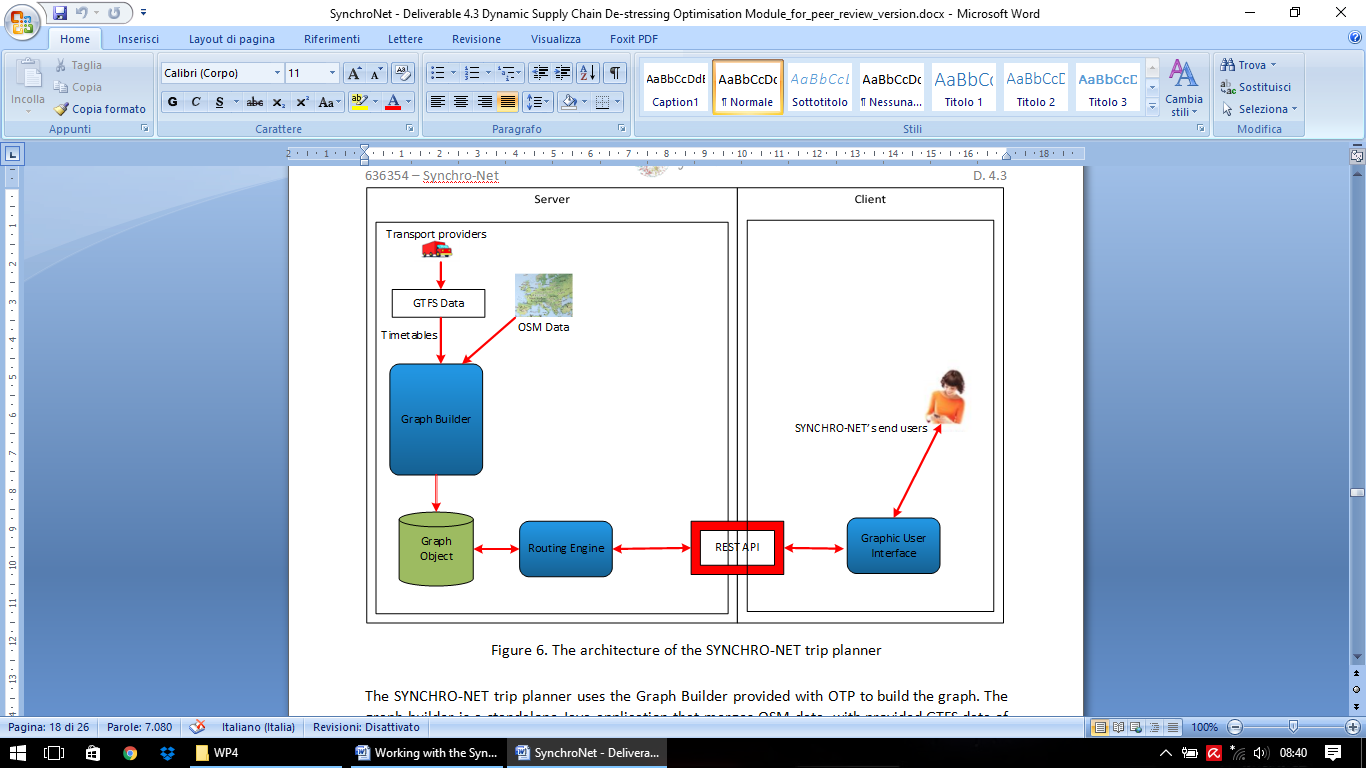


Figure Components of the SNETP

More precisely, the Graph Builder is a standalone Java application that merges *OSM data* with the provided GTFS data of the transportation system (see the next Section for details) and generates a *Graph Object*. This Graph Object is an *eXtensible Markup Language* (XML)-based text file and it is created once the server is set up. The Graph Object is the core data object on which all route planning algorithms implemented in the *Routing Engine* operate on. The Routing Engine is responsible for processing requests from the *Graphic User Interface* (GUI), extracting the requested data from the Graph Object, creating the response and sending it back to the GUI. The GUI is a map-based web interface (see the current version of it in Figure 5).

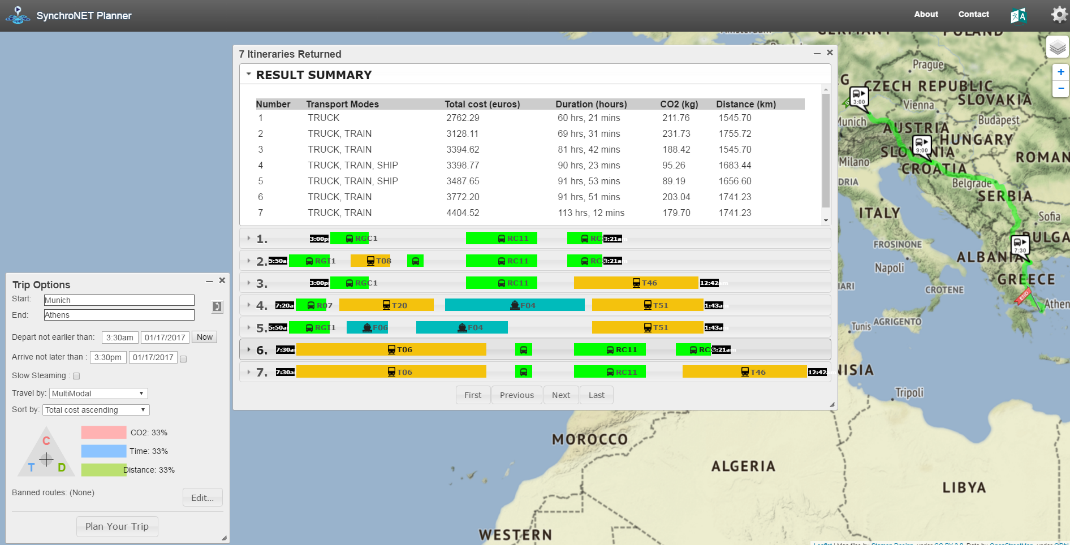


Figure Map-based web GUI of SNETP

# OpenTripPlanner (OTP)

The development of the SNETP platform is made upon an existing open-source collection of libraries called **OpenTripPlanner** (OTP). This package, written in Java, follows a client-server architecture and provides a map-based web interfaces as well as REST API for use by third-party applications. More info at: http://www.opentriplanner.org.

## Data standards

OTP mainly works on two open data standards, namely General Transit Feed Specification (GTFS) for transits and Open Street Map (OSM) for street networks.

### Open Street Map (OSM)

**Open Street Map** (OSM) is a world-wide [collaborative mapping project](https://en.wikipedia.org/wiki/Virtual_community) that aims at creating a [free](https://en.wikipedia.org/wiki/Free_content) editable [map](https://en.wikipedia.org/wiki/Map) of the entire world (https://www.openstreetmap.org/).  OTP reads OSM data (from one or more files) as input in order to create the graph/network on which the routes are constructed.

OSM geo-spatial data can be downloaded from the official website or from other repositories (e.g. http://download.geofabrik.de). Data are encoded in **.osm** files with different formats, such as *OSM XML* or *PBF* (http://wiki.openstreetmap.org/wiki/OSM\_file\_formats). OTP can consume both formats, but it's better to work with PBF files since they are smaller and faster to load. Due to the enormous number of information stored in OSM files, in fact, some formats (such as PBF) are highly compressed but hardly human-readable.

### General Transit Feed Specification (GTFS)

The **General Transit Feed Specification** (GTFS) is used to generate the schedule for a list of routes, where each route is defined by some characteristics such as origin, destination, transport mode, departure time and arrival time at each stop passing through the route, etc.

You can find a general overview of GTFS format at <https://developers.google.com/transit/gtfs/>, and a book entitled "The definitive guide to GTFS" by Q. Zervaas is also available in the project documentation. At <https://github.com/google/transitfeed/wiki/FeedValidator> you can find the tool to validate GTFS feeds.

## Working with the OTP

In this Section we will give some technical information to start using the OTP application and to modify the source code in order to build a custom planner. We refer here, for the sake of simplicity, to the basic version of the OTP (as the one you can download from the official website), but similar procedures should apply also to the already customized OTP versions.

To run correctly, OTP needs a *Java Virtual Machine* (JVM) version 1.8 (or newer), which is provided as part of the *Java Development Kit* (JDK) or of the *Java Runtime Environment* (JRE).

We strongly suggest to install the latest version of the JDK, in order to be able to also customize the OTP application, that can be found at:

www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

### Running the OTP

OTP can be downloaded from: http://maven.conveyal.com/org/opentripplanner/otp/1.1.0-SNAPSHOT/ as a single runnable *.jar* file named **otp-*version*-shaded.jar**. Up to day, the latest release is *1.1.0-20170202.172814-23*. To simplify the instructions, we rename it as **OTP.jar**.

Once downloaded, the .jar file can be executed from a command line by using:

java -Xmx8G -jar OTP.jar <options>

where the -Xmx parameter sets the limit on how much memory the application is allowed to consume (in this case 8 GigaByte) and -jar sets the type of executable file that follows.

The <options> are those specific for the OTP application and can be listed by using --help parameter, as follows: java -Xmx8G -jar OTP.jar --help

The list of possible options, and their meaning (from the --help output), are the following:

--analyst Enable OTP Analyst extensions.

Default: false

--autoReload Auto-reload registered graphs when source data is modified. Default: false

--autoScan Auto-scan for graphs to register in graph directory.

Default: false

--basePath Set the path under which graphs, caches, etc. are stored by default. Default: /var/otp

--bindAddress Specify which network interface to bind to by address.

0.0.0.0 means all interfaces. Default: 0.0.0.0

--build Build graphs at specified paths.

--cache The directory under which to cache OSM and NED tiles. Default is BASE\_PATH/cache

--clientFiles Path to directory containing local client files to serve.

--disableFileCache Disable http server static file cache. Handy for

development. Default: false

--enableScriptingWebService Enable scripting through a web-service

(Warning! Very unsafe for public facing servers)

Default: false

--graphs Path to directory containing graphs.

Defaults to BASE\_PATH/graphs.

--help Print this help message and exit.

--inMemory Pass the graph to the server in-memory after building it,

without saving to disk. Default: false

--insecure Allow unauthenticated access to sensitive API resources,

e.g. /routers. Default: false

--pointSets Path to directory containing PointSets.

Defaults to BASE\_PATH/pointsets.

--port Server port for plain HTTP. Default: 8080

--preFlight Pass the graph to the server in-memory after building it,

and saving to disk. Default: false

--router One or more router IDs to build and/or serve, first one

being the default.

--script run the specified OTP script (groovy, python)

--securePort Server port for HTTPS.

--server Run an OTP API server. Default: false

--verbose Verbose output. Default: false

--version Print the version, and then exit. Default: false

--visualize Open a graph visualizer window for debugging. Default: false

For example, a common call should look like:

java -Xmx8G -jar OTP.jar --build *data\_path* --inMemory

where *data\_path* corresponds to the folder containing GTFS files in .zip format and OSM files (for example, C:\\data\). The time to complete the OTP setup depends on the size of the data input and may takes several minutes. At the end of the process you should see a message saying Grizzly server running. At this point, by opening [http://localhost:8080/](http://icesrv.polito.it:8080/) in a web browser, you should be able to use the OTP application.

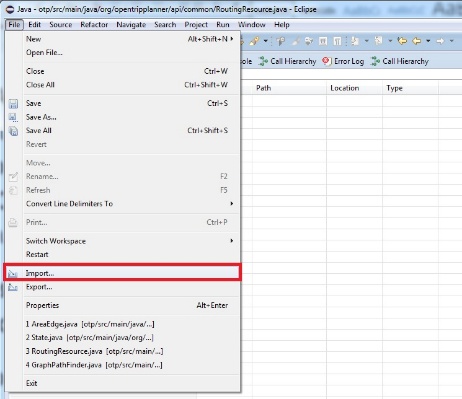
For more details, please refer to <http://docs.opentripplanner.org/en/latest/Basic-Usage/>.

### Working with the OTP source code (by using Eclipse)

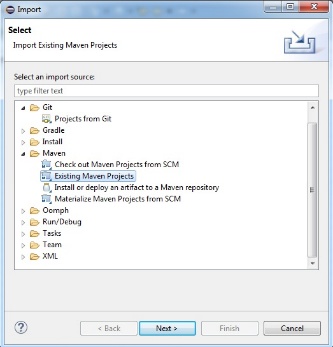
First, from https://github.com/opentripplanner/OpenTripPlanner, the OTP source code must be downloaded. OTP is provided as a *Maven* project. Maven is basically a build and dependency management system that can be easily managed by professional IDEs. The IDE *Eclipse* for JAVA (freely downloadable from https://www.eclipse.org/) has been chosen for this development. The following steps will show you how to import the OTP project into Eclipse and how to build it into an executable file.

#### Importing the OTP project

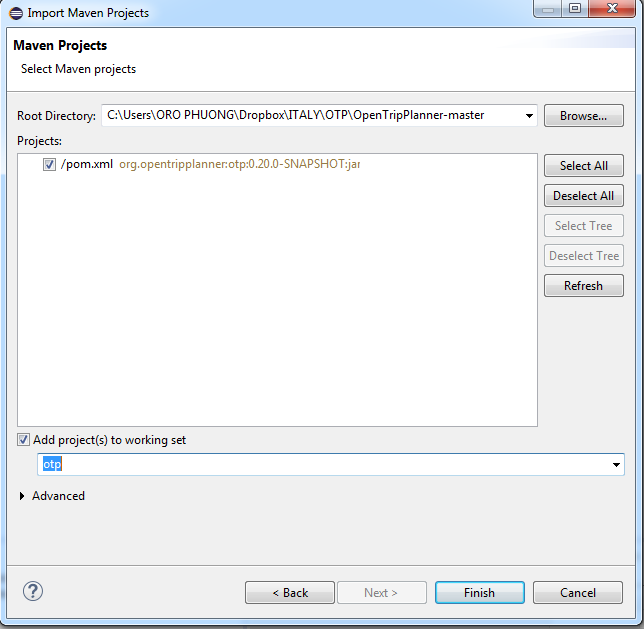
* Once opened Eclipse, go to the menu **File**/**Import**



* In the **Import** window, select **Existing Maven Project** and click **Next**



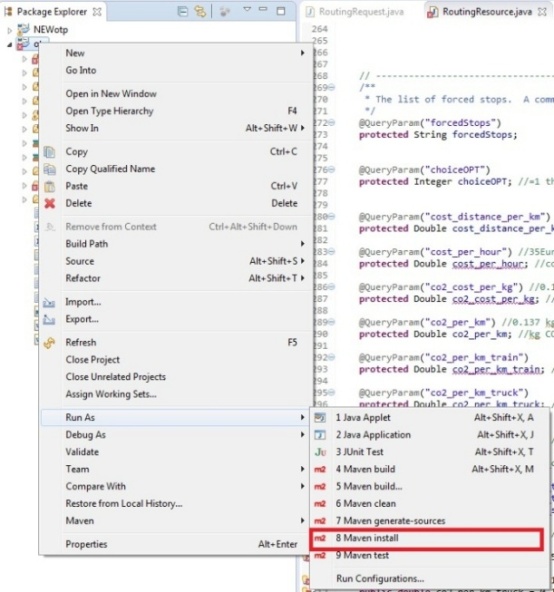
* In **Import Maven Projects** window: 1) in **Root directory** input field select the downloaded folder containing the OTP; 2) in the **Projects** field, choose the **/pom.xml** item; 3) in **Add project(s) to working set** section, write "otp”; 4) click **Finish**



If all went right, we will get the OTP project imported into Eclipse and we can start modifying the source code.

#### Build the .jar file

After modifying the source code, we need to build the jar file. This can be done by right-clicking on the project name (in the **Package Explorer** panel), and by choosing **Run As/Maven install**:



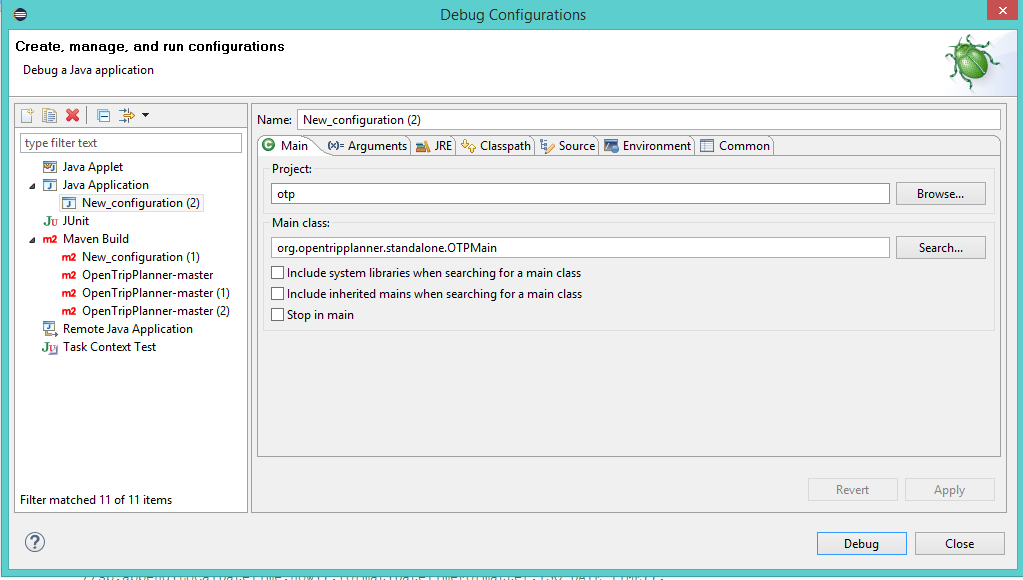
The executable .jar file, whose name ends with "shaded", will be created in the /**target** folder within the main OTP directory.

Troubleshooting: If the building gives some errors related to the JRE (look at the console output), try (even several times) to right-click on the project name, select **Maven/Update Project**, and then click **Ok**.

#### Build and run OTP in Debug Mode

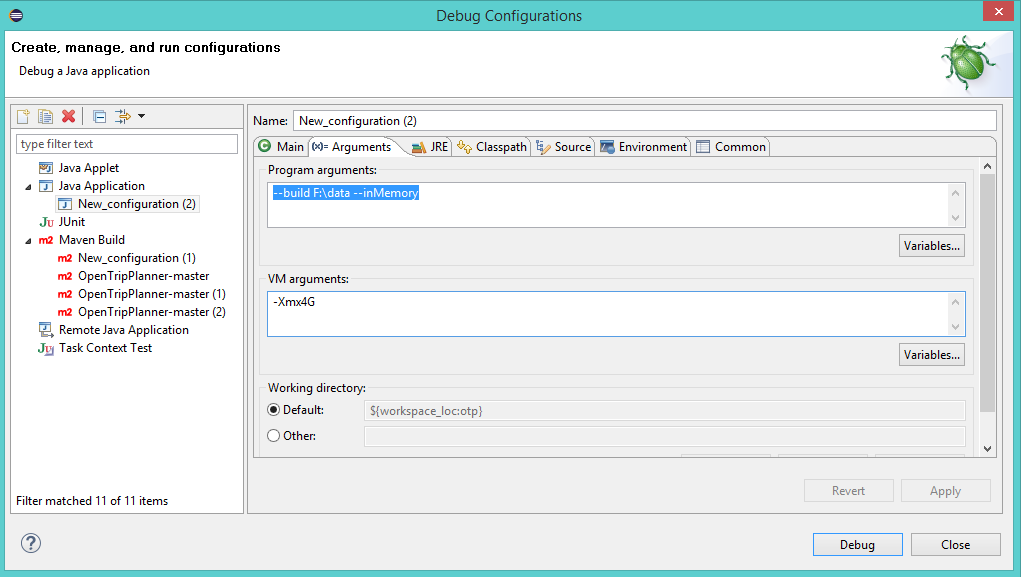
In order to run OTP in Debug Mode, open the **Debug Configurations** window from the **Run** menu. Here, create a new configuration of the **Java Application** type. This can be done by selecting **Java Application** from the list and clicking the **New Launch Configuration** button. The configuration has to know which project to launch and which is the "main" class of the project. Hence in the **Main** tab, select the following data:

* Project: **otp**
* Main class: **org.opentripplanner.standalone.OTPMain**



Moreover, if you want to launch the program with some particular parameters, these ones should be indicated in the **Arguments** tab. For example, if we want to run OTP with the same parameters used in Section 2.2.1, we must enter the following:

* Program arguments: **--build *data\_path* --inMemory**
* VM arguments: **-Xmx8G**



## Main OTP packages

Basically, all the source code that we need to work with can be found in **src/main/java**. In the following, we list the main files, packages, class and methods involved in the actual trip planning process. To deeply understanding the system implementation details, please refer to:

* Project documentation: http://docs.opentripplanner.org/en/latest/
* Javadoc documentation of the project: http://dev.opentripplanner.org/javadoc/1.0.0/
* Webservice API documentation: http://dev.opentripplanner.org/apidoc/1.0.0/index.html
* A discussion forum: https://groups.google.com/forum/#!forum/opentripplanner-users

### Graph generation and routing

In the following, there are some main packages that deal with routing part:

* Org. opentripplanner.routing
  + Org. opentripplanner.routing.graph
  + Org.opentripplanner.routing.algorithm
    - Org.opentripplanner.routing.algorithm.strategies
  + Org.opentripplanner.routing.impl
  + Org.opentripplanner.routing.core
  + Org.opentripplanner.routing.spt
  + Org.opentripplanner.routing.vertextype
  + Org.opentripplanner.routing.edgetype

More precisely, with respect to the main OTP tasks:

* When the graph object is created by using information based on the OSM and GTFS data, its vertices and edges are defined in the packages **org.opentripplanner.routing.vertextype** and **org.opentripplanner.routing.edgetype**, respectively. In particular, concerning the GTFS generation, **GTFSPatternHopFactory** class generates a set of edges from GTFS data and **GtfsLibrary** class defines the route type. The OSM and GTFS data are basically two separate layers in the graph. The **TransitToStreetNetworkGraphBuilderImpl** connects the two together with **StreetTransitLink** edges.
* When a routing request is done by the user, the value for input parameters are taken from the **RoutingResource** and **RoutingRequest** classes. Then **GraphPathFinder** call the selected routing algorithm. The routing algorithms (such as *A\**, *Dijkstra*, etc.) can be found in the package **org.opentripplanner.routing.algorithm**.

## Web user-interface

To deal with the user interface, we need to work with the following packages

* **Org.opentripplanner.api.model**
* **Org.opentripplanner.api.resource**

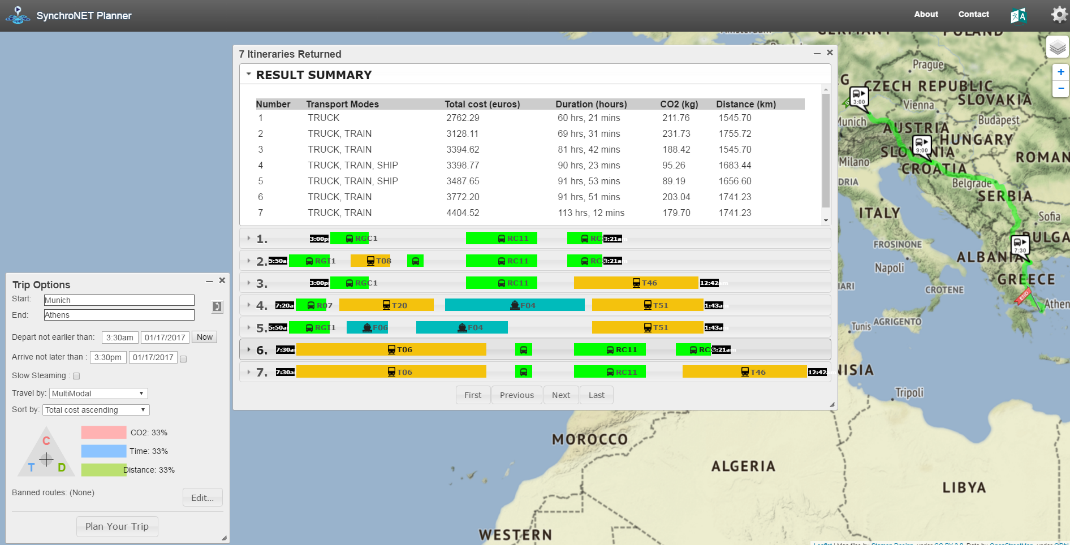
and with the *javascript* files contained in the folder **/src/client/js/otp**.

In general, the routing algorithm presents the result (a list of alternative routes/paths connecting a pair of nodes) as list of **GraphPath** objects. While for the user interface, the result is represented by a list of **TripPlan** objects. Therefore, the class **GraphPathToTripPlanConverter** is used to convert internal **GraphPath**s (found by routing algorithm, e.g. A\*, in the GraphPathFinder.java) to **TripPlan**s, which are returned by the OTP web services. Each **TripPlan** (defined in TripPlan.java) is made up of list of **Itinerary** (defined in Itinerary.java), and each **Itinerary** is made up of list of **Leg** (defined in Leg.java).

The user interface is composed by two panels (as shown in the figure):

1) **Trip Options** is used to get input parameters from the users

2) **Itineraries Returned** is used to show the result (list of alternative routes/itineraries)



The following table shows, for each panel, the list of the main related files :

|  |  |
| --- | --- |
| **Panel** | **Files to work with** |
| Trip options | * TripOptionsWidget.js * tripoptions-templates.html * tripoptions-style.css * PlannerModule.js (to define the queryParam) * MultimodalPlannerModule.js (to add widget) |
| Itineraries returned | * ItinerariesWidget.js * Itinerary.js * TripPlan.js * TripPlan.java * Itinerary.java * Leg.java |

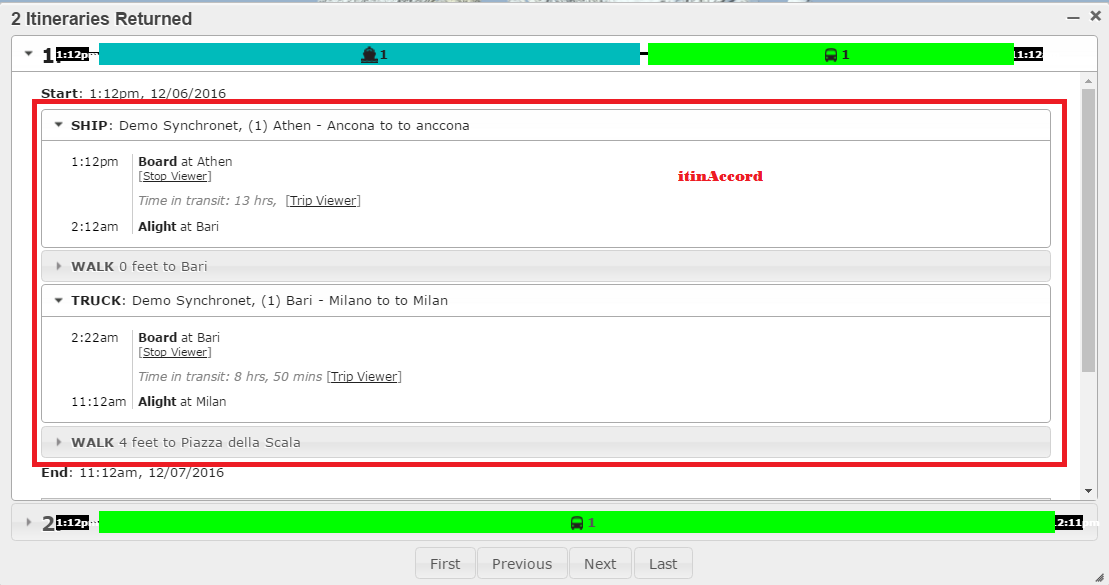
When an itinerary is returned, the format of a leg title is as follows:

***mode***: *agency\_name,* (*route\_short\_name*) *route\_long\_name* to *trip\_headsign*

where:

* *agency\_name* is defined in **agency.txt** of GTFS
* *route\_short\_name* and *route\_long\_name* are defined in the **route.txt** of GTFS
* *trip\_headsign* is defined in the **trip.txt** of GTFS

For example, in the leg showed in the figure, ***mode*** = TRUCK*,**agency\_name* = Demo Synchronet, *route\_short\_name* = 1, *route\_long\_name*= Bari - Milano, and *trip\_headsign* = to Milan.



Note that:

* when hovering on a leg of an itinerary, the leg on the map will be highlighted by calling **HighlightLegdefined** in **PlannerModule.js**
* **config.js** converts miles to km by using metric parameter
* **renderHearderContent** in **ItinerariesWidget.js** draws the black line with time in the header of itinerary window
* the color representing a transportation mode can be changed in **PlannerModule.js**.

### Web services

The packages:

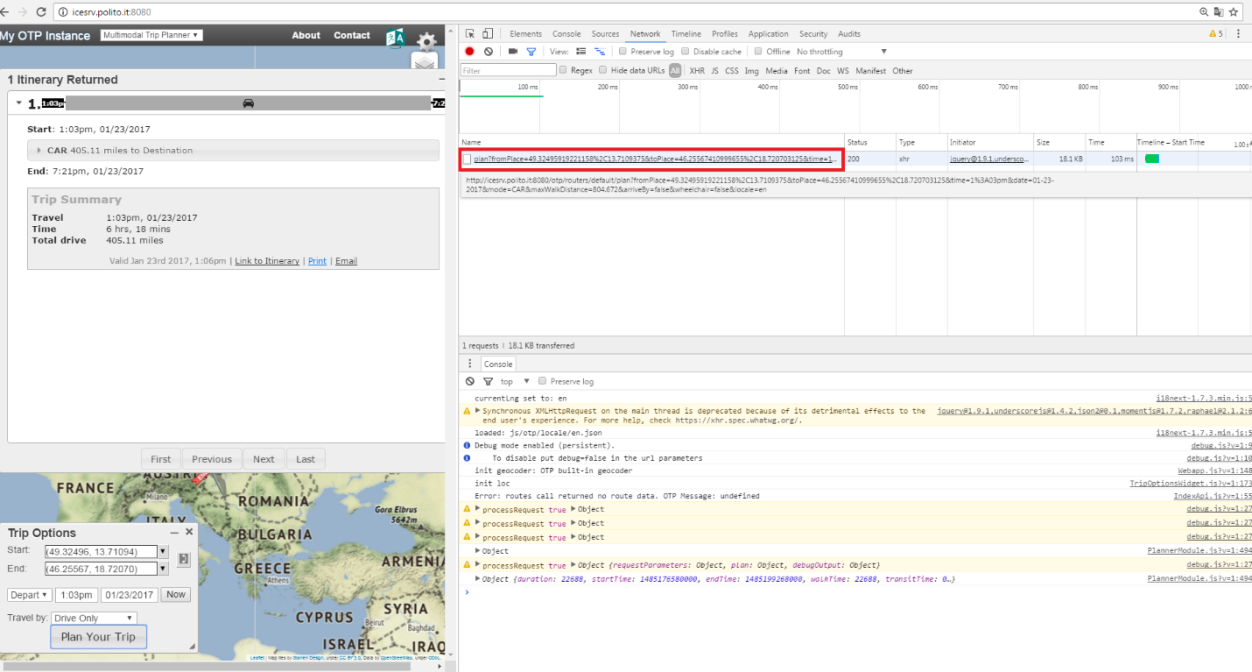
* Org.opentripplanner.api
  + Org.opentripplanner.api.common
  + Org.opentripplanner.api.resource

contain the code which exposes OTP services to the outside world as a REST API. OTP provide the REST API as both a WAR-packaged servlet (via module **otp-rest-servlet**) and a standalone Grizzly-based command-line invoked server (via module **otp-core**).

The class **PlannerResource** is the primary entry point of the OTP web service.

#### Simulating a request from the user-interface

Each request from the user interface is basically translated into an HTTP string (containing all the parameters set by the user) and sent to the server API. An example of such a string can be found by inspecting the HTTP request (consequent to a click of the "Plan your trip" button) through the use of your web browser. E.g., *Google Chrome* web browser will show this information in the "Network" tab, accessible by Ctrl+Maiusc+I (or right-click and then "Inspect"), as shown in the figure:



However, if you want to simulate a client request (for example for debug or batch purposes) by creating a similar string and passing it to the server, you should explicit (as an input of the OTP call) the argument --enableScriptingWebService to deactivate the relative restriction. Otherwise, the system will reply with a non-interpreted XML response.

# The Synchro-net Planner implementation

This section is devoted to describe the decisions taken and the particular implementation details of the SNETP, with a particular emphasis on the classes, methods and parameters involved. In order to respect the project objectives and to create suitable demonstrators (this part of the project is obviously still ongoing). This activity can be divided in three main tasks:

1. Data gathering
2. Software customization and development
3. Interaction with other software modules

## Data gathering, generation, and filtering

The SNETP can creates the routes from OSM and GTFS files (or from each of them separately). In general, OSM data are used to generate the topology of the network, while GTFS are used to define the schedule of the possible trips. Moreover, since the granularity of the OSM data is too high for the Synchro-Net objectives, **we have decided to use only the GTFS data in the final version of the planner**. To generate all the routes, with their schedules, in GTFS format, we will use a modified version of the OTP (*GTFS\_Batch*) taking .osm data and a series of interesting locations as input. In the following, we explain how gather, filter and put in the correct format the data for the SNETP.

The following tools are needed (easily findable for Linux and Windows operating systems):

* osmconvert: [http://wiki.openstreetmap.org/wiki/Osmconvert](http://wiki.openstreetmap.org/wiki/Osmfilter)
* osmfilter: <http://wiki.openstreetmap.org/wiki/Osmfilter>
* osmosis: <https://wiki.openstreetmap.org/wiki/Osmosis>

### GTFS\_Batch: a GTFS routes generator

The aim of SYNCHRO-NET is to create itinerary consisting of multi-modal modes: truck, train and ship. This is totally not provided by the original OTP, which only supports the public transportation for passengers and therefore it uses GTFS for bus schedule and OSM to create itinerary for driving. However, the OTP can create itineraries by using the data either from both OSM and GTFS or from each of them separately. Hence, for our purpose, it is possible to incorporate all the route information for all the modes (truck, rail, ship) into the GTFS data format, without the need of using osm data for creating the logical graph on which the routing algorithm will search. The basic structure of the feeds must follows the standard GTFS rules, however, some data may be used/interpreted differently if necessary. For example, we will see that the name of the locations in *stops.txt* file is currently a list of (possible) many locations corresponding to a cluster identified by unique coordinates.

The developed version *GTFS\_Batch* (that can be found on the ICE server, see Section 4) is an OTP modified version able to generate GTFS feeds for truck or for trains. Given a set of locations, it basically creates all the possible routes for each origin-destination combination. The application takes some parameters as input arguments (in addition to the OTP basic ones):

* --GTFSpath *file.txt* the path of the text file containing the locations. Currently, this file has the same format of a common GTFS stops.txt file. There are 6 fields separated by a comma, in a .cvs style, where the third, the fifth and the sixth fields are the only ones parsed and correspond to the name of the location and to its latitude and longitude, respectively. The file must not contains any headers, only data. The fields are in order of:

stop\_id, stop\_code, stop\_name, stop\_desc, stop\_lat, stop\_lon

Example:

xxx, xxx, Berlin, xxx, 52.5170365, 13.3888599

xxx, xxx, Madrid, xxx, 40.4167047, -3.7035825

xxx, xxx, Athens, xxx, 37.9841493, 23.7279843

xxx, xxx, Kyiv, xxx, 50.4501071, 30.5240501

xxx, xxx, Rome, xxx, 41.8988173, 12.47311

* --GTFSmode [truck,train] the mode (truck or train) for which the GTFS routes are created. Depending on the mode chosen, the osm input file should contains roads or railways information. In the following, we will see how we created these files.

For instance, to generate GTFS for trains we use the following command:

java -Xmx32G -jar GTFS\_Batch.jar --build *data\_path* --GTFSpath locations.txt --GTFSmode train –inMemory

where *data\_path* contains the osm file with railways data.

GTFS\_Batch has been recently improved by Giuseppe. This new version (as well as the internal procedure of the basic one) should be analyzed and documented here.

Currently, the ship routes and schedules (for example from China ports to Spain) are added manually, or given by the maritime partners.

### Obtaining the file of the locations

DEPRECATED We have decided to consider the main cities in Europe (for example cities with more than a certain number of inhabitants), the main European ports and inter-ports (that may not correspond to a big city but are important for commercial trades, e.g. interport of Orbassano or the port of Ancona in Italy). Main cities can be found automatically by using osm data, the other locations are currently added manually or suggested by the trading companies. In the following we explain the procedure to found the main cities from osm data and create the *locations.txt* file:

1) Download the European OSM, go to: [http://download.geofabrik.de/europe.html#](http://download.geofabrik.de/europe.html) (then click 'raw directory index'). Select the latest version and the preferred format, for example the compressed *europe-161217.osm.pbf*.

2) Since *.pbf* compressed files cannot be read by all the applications (as **osmfilter** that we need to use), the program [**osmconvert**](http://wiki.openstreetmap.org/wiki/Osmconvert) will help you converting to other formats (e.g., to .o5m).

$ osmconvert europe-161217.osm.pbf -o=file.o5m

3) Filter OSM file to leave only nodes objects corresponding to cities with more than 100000 inhabitants. We use **osmfilter** with the following command:

$ osmfilter europe-161217.o5m --keep=" ( place=city or place=town ) and population>=100000 " --drop-ways --drop-relations > europe100K.osm

4) Convert the data in a .csv format using **osmconvert.** The following command creates a locations file in a correct format for being read by *GTFS\_Batch* (note that the id is used to fill the non-necessary fields):

$ osmconvert europe100K.osm --csv-separator="," --csv="@id @id name @id @lat @lon" > locations.txt

However, some problems may arise using this basic procedure:

* generally the name of a city in a osm file is written in the national language. This may cause problem in reading non-common characters (cyrillic alphabet, or scandinavian accents marks and symbols) but, more important, makes hard to find the desired origin-destination location for a user.

Solution: It seems that the most important cities have an international (or English) name, corresponding to the key **name:en** in OSM specification, but other cities (the most) may not have one. This is still an open issue...

* the number of generated locations may be too high taking into account that the generated GTFS approximately contains N x N possible routes, where N is the number of locations. For example, the number of location generated by the above filter, is 677. This number doubles is the lower bound on population is changed to 50000. This seems an intractable number of generated routes, causing too long response times or no response at all from the planner when searching for a route. Consider that some ports and inter-ports may be added, further increasing the number of locations to consider.

Solution: Basically, the number of locations has to be reduced. This is still an open issue... One may think to filter more restrictively on the cities' population, or to decide more "qualitative" filtering (e.g., eliminating or considering only few main cities for all the non-EU countries). For the moment, we have decided to apply a clustering of the 677 locations, based on their closeness. Cities within a radius of 50km of geodetic distance are grouped together, and the coordinates of the most populated city among each cluster is chosen as the centre (the automatic procedure applying the clustering, developed by Daniele, needs to be better explained as soon as we decide how to filter). The locations can be reduced to 431 this way. The clustered location.txt file looks like this:

xxx, xxx, Berlin||Potsdam, xxx, 52.5170365, 13.3888599

xxx, xxx, Madrid||Móstoles||Alcobendas, xxx, 40.4167047, -3.7035825

xxx, xxx, Athens||Piraeus||Kallithea, xxx, 37.9841493, 23.7279843

xxx, xxx, Rome, xxx, 41.8988173, 12.4731

### Obtaining the OSM data for trucks and trains

To obtain these data we start by the *europe-161217.o5m* file created after step 2 of the procedure in the previous section.

**OSM for trucks**

We need to filter the OSM by keeping also the main road network. For the moment, we have decided to consider only some types of streets (primary, secondary, motorway, trunk and the relative links), but this may be changed to further reduce the computational effort needed to create the routes and their complexity. We use **osmfilter** with the following command:

$ osmfilter europe-161217.o5m

--keep=" ( place=city or place=town ) and population>=100000 "

--keep-ways=" ( highway=primary or highway=secondary or highway=motorway or highway=trunk or highway=motorway\_link or highway=trunk\_link or highway=primary\_link or highway=secondary\_link )" > EUroads.osm

**OSM for trains**

We need to filter OSM file by keeping only the railway. We use **osmosis**:

$ osmosis--read-pbf file=europe-161217.osm.pbf --tf reject-relations --tf accept-ways railway=rail,station --used-node --write-xml rail.osm

To make sure remove all highway, we use **osmfilter**:

$ osmfilter rail.osm --drop-tags="highway=" -o=EUrail.osm

Modify *EUrail.osm* by replacing: <tag k="railway" v="rail"/> with <tag k="highway" v="motorway"/>. This let us to use GTFS\_Batch, pretending to work with roads.

## Software customization and development

Here we summarize some of the main customizations implemented in the software in order to meet the requisites for the SNETP. A particular attention is devoted to explicit the "places" (files, class, methods) in which the code have been modified.

### GTFS input feeds

When working with different GTFS feeds at the same time (i.e., different zip containing for example routes for the different transportation modes), our OTP version wasn't able to combine the trips from different feeds in a unique route. Details of the problem and how it was solved at: https://github.com/opentripplanner/OpenTripPlanner/issues/2371

### Routing optimization

This section describe how we decided to search for all the possible routes between a selected origin to a selected destination, and respecting the constraints imposed by the user.

Originally, the OTP used only plain **Dijkstra**’s algorithm and **A\*** algorithm (Mandow et De la Cruz., 2005, 2010) with a distance metric to compute trips for all modes of transportation. Those algorithms performed very slowly on large graphs. A new technique called **Contraction Hierarchies** (Geisberger et al., 2008) performs better and more quickly on large graphs. The idea behind Contraction Hierarchies is that a large graph can be contracted by removing vertices one at a time and replacing any paths through the removed vertex with a shortcut representing that path. Based on the implemented routing algorithms, the SNETP adds some new functions in order to address multiple objectives of (e.g., minimizing the pollution/CO2 emissions, improving the safety of trip, etc). It thus can provide more various solutions to the decision makers.

The A\* search algorithm, already included in the OTP, has been customized to accomplish with the user specification. In particular, the objective function that the algorithm try to optimize (minimize in this case) takes into account three different factors, namely the distance, the CO2 emission, and the time. For each route, we calculate a *generalized cost* given by a weighted combination of these three factors:

generalized cost = [distance\_w \* (cost/km) \* Distance (km)] +

[co2\_w \* (cost/kg) \* CO2(kg)] +

[time\_w \* (cost/hour) \* Time(hours)]

= [distance\_w \* (cost/km) \* Distance (km)] +

[co2\_w \* (cost/kg) \* CO2(kg/km) \* Distance(km)] +

[time\_w \* (cost/hour) \* Time(hours)]

This cost is only a quality measure (e.g., can be normalized) and does not represent necessarily the real cost one should pay to follow that route for his freight transportations.

Weights are basically chosen by the user through the option panel interface. Defaults are:

**public double** distance\_w = 1.0; //weight on the distance in generalized cost

**public double** co2\_w = 1.0; //weight on the CO2 in generalized cost

**public double** time\_w = 1.0; //weight on the travel time in generalized cost

The other parameters for the three factors are defined in the **RoutingRequest** class. Note that, concerning distance and CO2 emission, we can have different parameters depending on the transport mode.

**public double** cost\_distance\_per\_km\_train = 0.0324; //cost in euro per km for train

**public double** cost\_distance\_per\_km\_truck = 0.4; //cost in euro per km for truck

**public double** cost\_distance\_per\_km\_ship = 0.108; //cost in euro per km for ship

**public double** cost\_per\_hour = 35; //cost in euro per 1hour of driving/waiting

**public double** co2\_cost\_per\_kg = 0.15; //cost in euro per 1 kg CO2

**public double** co2\_per\_km\_train = 0.06; //kg CO2 per km train

**public double** co2\_per\_km\_truck = 0.137; //kg CO2 per km truck

**public double** co2\_per\_km\_ship = 0.02; //kg CO2 per km ship (medium speed)

**public double** co2\_per\_km\_ship\_Slow = 0.01; //kg CO2 per km ship (slow speed)

**public double** co2\_per\_km\_ship\_Medium = 0.02; //kg CO2 per km ship (medium speed)

**public double** co2\_per\_km\_ship\_Fast = 0.04; //kg CO2 per km ship (fast speed)

Note also that, for the ship mode, we create three types of speed:

* slow speed = 10 knots
* medium speed = 15 knots
* fast speed = 20 knots

Since the GTFS format does not allow to explicitly define a particular transportation mode (apart from the common ones), in order to work with CO2 emissions for different velocities, we assume that the **routeID** ends with 1, 2, or 3 if it corresponds to a route running with slow, medium,or fast velocity, respectively (see **OnBoardDepartPatternHop.java** for an example).

**Astar** calculates the objective function by using **traverse** method for edges. The following is the list of edges whose **traverse** method need to be modified for multi-objective:

|  |  |  |  |
| --- | --- | --- | --- |
| **Edges** | **CO2** | **Time** | **Distance** |
| OnBoardDepartPatternHop | X | X | X |
| PatternDwell |  | X |  |
| PatternHop | X | X | X |
| PatternInterlineDwell |  | X |  |
| PreBoardEdge |  | X[waiting] |  |
| TransferEdge |  | X[waiting] |  |
| TransitBoardAlight |  | X[waiting: don’t need]  Time for boarding to a new vehicle |  |
| PreAlightEdge |  |  |  |
| SimpleTransfer |  | ?? walkReluctance |  |
| StreetTransitLink |  | X[waiting: don’t need] |  |

The X on each column CO2/Time/Distance means that the corresponding edge has an effect on the factor CO2/Time/Distance. For example, for the edge **PatternInterlineDwell**, the (X) only appears in the column Time, it means that this type of edge only affects the Time factor.

### Working with schedules

We have identified two possibilities to specify time information for a route using the GTFS:

* **SYNCHRONOUS ROUTE** mode: the exact departure and arrival time are defined for each stop of the route (e.g. 9am, 18pm, etc). It needs only the file stop\_times.txt
* **ASYNCHRONOUS ROUTE** mode: the relative departure and arrival time are defined for each stop of the route AND the frequency of the operation of the route. It needs both files frequencies.txt and stop\_times.txt

The OTP can handle both synchronous and asynchronous routes at the same time. However, the original OTP (created for bus train schedules) can only manage itineraries with a duration less than 72 hours (3 days) when working with synchronous route and less than 24 hours (1 day) when working with asynchronous route.

Therefore, we have modified the code in order to handle longer durations (e.g., for ship trips):

* for Synchronous route: we changed the value of the parameter **dayCount** in the method **setServiceDays()** of the **RoutingContext.java** (currently, we have set *dayCount = 6*, hence the maximum duration of an itinerary is 144 hours);
* for Asynchronous route: see the method **nextDepartureTime** and **preArrivalTime** of the **FrequencyEntry.java**.

### Some important parameters

**The default speed of vehicles**

If itinerary are generated by using OSM data for the trucks, then we need to define the speed of vehicles in **RoutingRequest**. For example: carSpeed = 40;

**The calculation of approximated position for isolated location**

There may exists locations not connected to the roads in our filtered OSM data. OTP tries to find the closet location connecting to the roads from the isolated location, then will find the path from this approximated location rather than the isolated location. The OTP gives parameter to define the radius to search this closet location in **SimpleStreetSplitter**:

* MAX\_SEARCH\_RADIUS\_METERS
* DUPLICATE\_WAY\_EPSILON\_METERS

**Asynchronous routes**

When frequencies.txt in GTFS is used: if the OTP creates an itinerary consisting of a trip (let say trip A) built from frequencies.txt, then the arrival time at the first stop of trip A = “departure time” at the last stop in previous\_trip of A in the itinerary + “headway” defined in the frequencies.txt of trip A. If A is the first trip of the itinerary, then the arrival time at the first stop of trip A = departure time defined in the user interface + “headway” defined in the frequencies.txt of trip A. See method **nextDepartureTime(int stop, int time)** in the **FrequencyEntry.java.**

**Internal time at stop**

File *transfer.txt* of GTFS defines the time to wait for transfer between 2 stops of 2 routes. **TransferEdge.java** reads the data from transfer.txt.

The method public int getBoardCost (TraverseMode mode) defines the cost we need to pay when boarding to a new vehicle. It is defined in the **RoutingRequest.java**, and used in the **TransitBoardAlight.java**.

**Boarding time and Alight time**

The **BoardingTime** and the **alightingTime** give information on how much time a vehicle takes for boarding/alighting operations. Can be significant e.g. for airplanes or ferries. For each transportation mode, they are defines in **Graph.java**:

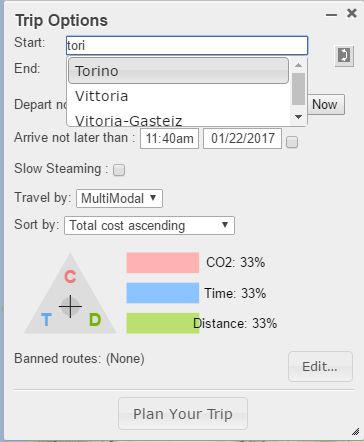
**public** Map<TraverseMode, Integer>boardTimes = Collections.***EMPTY\_MAP***;

**public** Map<TraverseMode, Integer>alightTimes = Collections.***EMPTY\_MAP***;

### User interface options and visualizations

#### Set origin-destination coordinates

With the original OTP, the user can only click the mouse on a point of the map to set it as the start/end location, then the coordinate of the selected point will appear in the Start/End combo boxes of the *Trip options* panel. Coordinates cannot be inserted into the combo boxes directly. Hence, **we modified the Start/End combo boxes so that the user now can give the input from keyboard**. Obviously, rather than asking the user remember the coordinates of the locations, the user now can give just the name of the locations and the system will automatically get the coordinates for these locations (auto-complete mode), as shown in the figure:



The list of locations from which a user can search is stored in the **TripOptionsWidget.js** (currently, it contains all the cities in Europe having population lager or equal to 50000 habitants and some harbors in China).

#### New input parameters

In general, to create new input from the user, we need to modify the relative API call (declare the name of the parameter) in

* **RoutingRequest**.java (declaration and initial assignment, **hash table**)
* **RoutingResource**.java (declaration and initial assignment in **buildRequest** method)

and the web user interface (modify the *Trip Options* panel) in **PlannerModule.js**. In particular, we have to declare the parameter in **otp.modules.planner.defaultQueryParams** and assign it a value in **planTrip** method.

Some new input parameters have been implemented, such as the "Travel by" option, the multi-objective function equalizer, the "Sort by" option, the "Banned route" option and so on.

For instance, the "Sort by" option let the user to sort the list of itineraries found in increasing order of certain metrics (generalizedCost, CO2 emissions, duration, distance, etc). To implement this option, we have created the parameter sortResultType in **RoutingRequest.java:**

**public int** sortResultType = 0;

and in **RoutingResource.java**:

@QueryParam("sortResultType")

**protected** Integer sortResultType;

and we also have to create sortResultType in **PlannerModule.js** for the web-user interface.

The corresponding values are: 0 - totalcost ascending; 1 - C02 ascending; 2 - duration ascending; 3 - distance ascending; 4 - departure ascending; 5 - arrival ascending.

The "Travel by" option, instead, is created in **config.js** file.

## Interaction with other software modules

In this section we have to clarify how the main modules of the whole SynchroNET system will interact with our planner.

### Interaction with the Risk Analysis and Optimization (RAO) module

TO DO

### Interaction with the Real-Time Optimization module

TO DO

# The SNETP storage on the ICE server

POLITO and the ICT for City Logistics and Enterprise (ICE) laboratory have provided a server machine to host, temporarily during the development phase, the SNETP. This server, called *ICEsrv*, is used to share the system outside the POLITO intranet, mainly for demonstrating and testing purposes.

## How to connect to the server

The server can be found at: http://icesrv.polito.it. Currently, it runs an *Ubuntu 14.04.4 LTS* operating system upon a *GNU/Linux 3.13.0-32-generic x86\_64* kernel version. The server can be accessed remotely via **ssh** only by using a username/password authentication (these credentials must be provided by the server administrators). Once logged in, you should be located in your personal home folder: /**home/*your\_user\_name***.

In order to connect via ssh to the server from a client running a Windows operating system, you can use PuTTy (http://www.putty.org/). Moreover, WinSCP (http://winscp.com) can be used to easily download/upload files.

## The storage of SNETP on ICEsrv

The information of this section have to be updated as soon as many decision are taken.

The OTP is stored in the folder /home/ALL\_ABOUT\_OTP. In this folder, there is an executable OTP file (OTP.jar) and four sub-folders:

Folder **OSM** contains all the OSM files that have been used/generated so far. More precisely:

**europe-161217.osm.pbf**: the original european osm file in pbf format

**cityLink50K.osm**: the filtered european file with cities >= 50000 inhabitants and all highway = primary/secondary/motorway/trunk and relative links

**europe\_highway.osm:** as same as cityLink50K.osm, but ignore secondary, so the size is smaller

**rail.osm**: the filtered european file with railway only

**railGTFS.osm:** is the file we get when modifying the rail.osm by replacing

<tag k="railway" v="rail"/> with  <tag k="highway" v="motorway" />

We will use this file to create GTFS for rail transport mode

**railGTFS1.osm:** as same as railGTFS.osm, but use osmosis rather than osmfilter to filter the data. (I keep both these osm files, you can check if there is difference between them when creating GTFS).

Folder **SOURCE\_CODE:**

* **Synchronet**: the last version of the SNETP
* **GTFS\_Batch**: the batch procedure to generate GTFS
  + **GTFS\_Batch1**: call Astar for each pair of locations; for example, there are total 1279 cities in stops.txt -->call Astar1279\*1278 times --> require large amount of computation time
  + **GTFS**\_**Batch2**: call Astar for each origin (let say city A taken from 1279 cities from stops.txt) for amount of computation time (e.g. 5 seconds) and the destination is not taken into account (in other word, it is one-to-many calculation). Then from the shortest path tree built by Astar, for each destination (let say B taken from 1279 cities from stops.txt, and not A), we extract the path and consider it as the path from A to B. It is not guaranteed that the path found is the shortest, but we just need to create GTFS and it helps to reduce the computation time. (this version is not completed yet, we still need to do some experiments for this version)

Folder **software:** contains the software to filter the OSM file (osmconvert, osmfilter, osmosis) and the software to validate the GTFS (transitfeed-windows-binary-1.2.15)

Folder **Data**:

* stops.txt consists of all cities having population >= 50000 getting from cityLink50K.osm. This file is used to create GTFS for truck and train (see Section 5.1.1Generate Itinerary) and also for generate Start/End location in “Trip options” panel (see Section 4.1). --> 1279 cities
* stops\_small.txt: reduce list of cities from stops.txt (remove cities from some countries: north Europe (Norway, Sweden, Denmark), UK, Turkey, Albania, Bulgaria, Croatia, Romania, Kosovo, Estonia etc. --> 893 cities

# Future developments

## Include Risk into the list of KPIs

Currently, we consider 3 KPIs (distance, CO2, and time) in the multi-objective function when generating itinerary. We need also to consider Risk as a KPI. The Risk is provided by RAO module (Risk Analysis and Optimization) responsible by Fraunhofer (see Deliverable 4.3 for details). The Risk is calculated based on itineraries already found by the planner. In other word, the Risk is not included in the computation of the weighted objective function guiding the search. Therefore, when we the planner get the list of itineraries, it will send these completed itineraries to the RAO module. The RAO module will calculate the risk for each itinerary and send them back to us. We will either present risk KPI in the “Itinerary Summary” together with Distance/CO2/Time on the user interface.

The interaction with the RAO module has to be described in section 3.3.1.

## User preferences

Some common preferences are already available in the OTP framework. They have to be enabled, if needed, and added to the user interface. Go to the RoutingRequest.java, we can see this list

* bannedRoutes
* bannedTrips
* bannedAgencies
* bannedStops
* bannedStopsHard
* transferPenalty
* nonpreferredTransferPenaly
* preferredRoutes
* preferredAgencies

For example, if there are total available 5 trips: {1,2,3,4,5}, and we want to allow just 2 trips {2,3}, it is not available in the OTP. But we can make it available by setting the bannedTrips={1,4,5}. Again, if we want to find the itinerary from location A to C going through location B, then we need to apply Astar twice, first from A to B, and then from B to C.

## Accounting and authentication

We need to create accounts for users to access to the system. The users are divided into different types such as carriers, shippers, etc. Each type of users may have different requirements and preferences, the system therefore need to give different corresponding functionalities to each type of user. The accounting will enable the possibility to store past decisions and favorite settings, in order to facilitate a future search.

## Database and search engine for historical decisions

The ***database*** stores information on historical decisions (e.g., routes, modes, companies, etc), aiming to help the users when planning future journeys and also the system “learning” from previous bookings. It thus consists of four main entities:

* “Nodes” database: stores all the nodes that the user has previously entered into the system.
* “Links” database: stores all the individual A-B connection previously entered by the user, tagged with the mode and duration only.
* “Combinations” database: stores all the combinations of links that the user has previously used. For example, if the user has previously booked a journey A-B-C-D-E, then the database will record that “A-to-E has been done by A-B-C-D-E”
* “Actual bookings” database: for each A-B connection booked by the user, the information of the leg on the haulier/ship, cost, C02 (if available), etc. will be stored in this database.

The ***search engine*** uses the above database and current ship/train schedules to list possible alternative bookings that have been used previously and also satisfy the required specifications about the journey given by the users (e.g., do not use train, must go via terminal B). For example, when the user requests a movement from A to E, the search engine queries the combinations database and actual bookings database to list all A-B-C-D-E combinations and corresponding cost, hauliers, etc that have been used previously for A to E.

## GTFS\_Batch for generating synchronous routes

Currently, the GTFS\_Batch can create GTFS only for ASYNCHRONOUS ROUTES. As a future development, we need to generate GTFS also for SYNCHRONOUS ROUTES.