Software Engineering 2: myTaxiService

Design Document

Chitti Eleonora, De Nicolao Pietro, Delbono Alex
Politecnico di Milano

December 4, 2015

Contents

Chapter 1

Introduction

1.1 Purpose

This is the Design Document for the myTaxiService application. Its aim is to provide a functional description of the main architectural components, their interfaces and their interactions, together with the algorithms to implement and the User Interface design. Using UML standards, this document will show the structure of the system and the relationships between the modules.

This document is written for project managers, developers, testers and Quality Assurance. It can be used for a structural overview to help maintenance and further development.

1.2 Scope

The system aims to offer a simple and reliable taxi calling and reservation service for a large city.

The software system is divided into four layers, which will be presented in the document. The architecture has to be easily extensible and maintainable in order to provide new functionalities.

Every component must be conveniently thin and must encapsulate a single functionality (high cohesion). The dependency between components has to be unidirectional and coupling must be avoided in order to increase the reusability of the modules.

Design patterns and architectural styles will be used for solving architectural problems in order to simplify the system comprehension and avoid misunderstanding during the implementation phase.

1.3 Definitions, Acronyms, Abbreviations

RASD: Requirements Analysis and Specification Document.

DD: Design Document (this document).

RDBMS: Relational Data Base Management System.

DB: the database layer, handled by a RDBMS.

UI: User Interface.

Application server: the layer which provides the application logic and interacts with the DB and with the front-ends.

Back-end: term used to identify the Application server.

Front-end: the components which use the application server services, namely the web front-end and the mobile applications.

Web server: the component that implements the web-based front-end. It interacts with the application server and with the users' browsers.

SOA: Service-oriented Architecture.

MVC: Model-View-Controller.

JDBC: Java DataBase Connectivity.

JPA: Java Persistence API.

EJB: Enterprise JavaBean.

ACID: Atomicity, Consistency, Integrity and Durability.

1.4 Reference Documents

This document refers to the following documents:

- *Project rules of the Software Engineering 2 project* [1]
- *RASD and Design Document assignment* - Software Engineering 2 project [2]
- *Template for the design document* - Software Engineering 2 project [5]
- *Requirement Analysis and Specification Document* - the previous deliverable [4]

1.5 Document Structure

This document is structured in five parts:

Chapter 1: Introduction. This section provides general information about the DD document and the system to be developed.

Chapter 2: Architectural Design. This section shows the main components of the systems with their sub-components and their relationships, along with their static and dynamic design. This section will also focus on design choices, styles, patterns and paradigms.

Chapter 3: Algorithm Design. This section will present and discuss in detail the algorithms designed for the system functionalities, independently from their concrete implementation.

Chapter 4: User Interface Design. This section shows how the user interface will look like and behave, by means of concept graphics and UX modeling.

Chapter 5: Requirements Traceability. This section shows how the requirements in the RASD [4] are satisfied by the design choices of the DD.

Chapter 2

Architectural Design

2.1 Overview

This chapter provides a comprehensive view over the system components, both at a physical and at a logical level.

The system will be described starting with high-level components (section 2.2). This high-level design will be thoroughly dissected and detailed in section 2.3. Section 2.4 will put some attention on the deployment of the system on physical tiers, and section 2.5 will describe the dynamic behaviour of the software. Section 2.6 will focus on the interface between different components of the system.

The design choices and patterns used in the aforementioned sections will be presented and discussed in section 2.7.

2.2 High level components and their interaction

The main high level components of the system are the following:

Database: the data layer is responsible for the data storage and retrieval. It does not implement any application logic. This layer must guarantee ACID properties.

Application server: this layer contains all the application logic of the system. All the policies, the algorithms and the computation are performed here. This layer offers a service-oriented interface.

Server side plug-ins: these are the plug-ins which can be used to extend the application server layer. In this document, two plugins will be

designed:

- ride sharing
- ride reservation

Web server: this presentation layer provides a web interface. This layer does not contain any application logic.

Mobile application: this presentation layer consists in the mobile client. It communicates directly with the application server.

User's browser: this presentation layer represents the user's browser; it is not under the control of the system and it accesses the web server.

These high-level components are structured into four layers, shown in figure 2.1.

This design choice makes it possible to deploy the application server and the web server on different tiers. It also improves scalability, since there may be many web servers talking to a single application server. Further implementations may include the use of caching at the web server level.

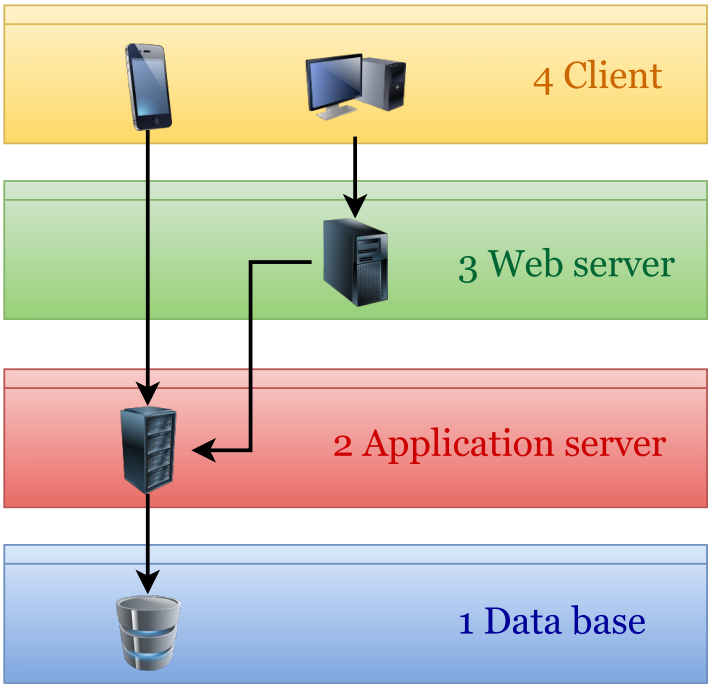


Figure 2.1: Layers of the system.

The interactions between the main components are shown in the figure 2.2. They are all synchronous (obviously, the web server and the application server are multi-threaded).

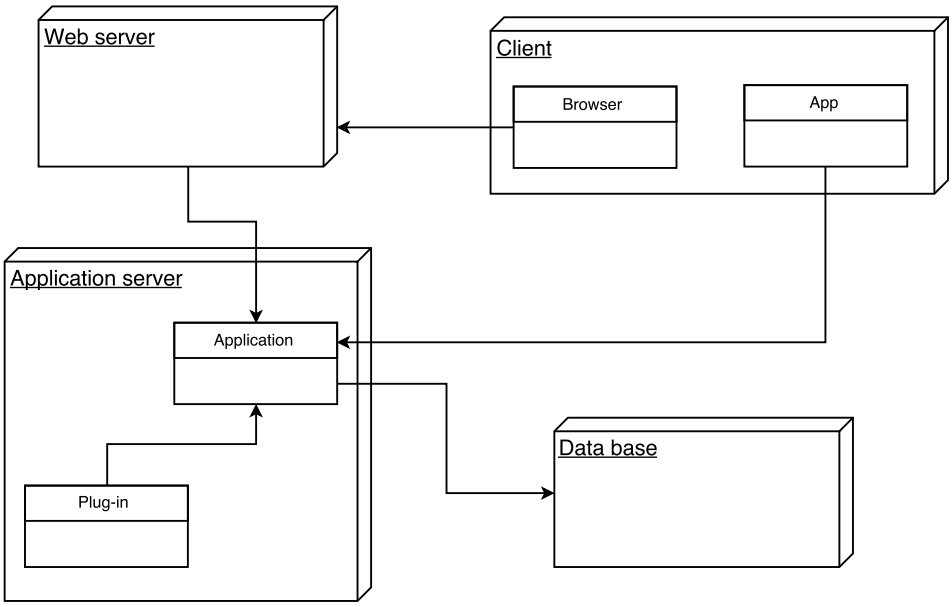


Figure 2.2: High level components of the system.

2.3 Component view

2.3.1 Database

The database tier runs MySQL Community Edition and uses InnoDB as the database engine: the DBMS has to support transactions and ensure ACID properties. The DBMS will not be internally designed because it is an external component used as a "black box" offering some services: it only needs to be configured and tuned in the implementation phase.

The database can communicate only with the business logic tier using the standard network interface, described in section 2.6. Security restrictions will be implemented to protect the data from unauthorized access: the database must be physically protected and the communication has to be encrypted. Access to the data must be granted only to authorized users possessing the right credentials. Every software component that needs to access the DBMS

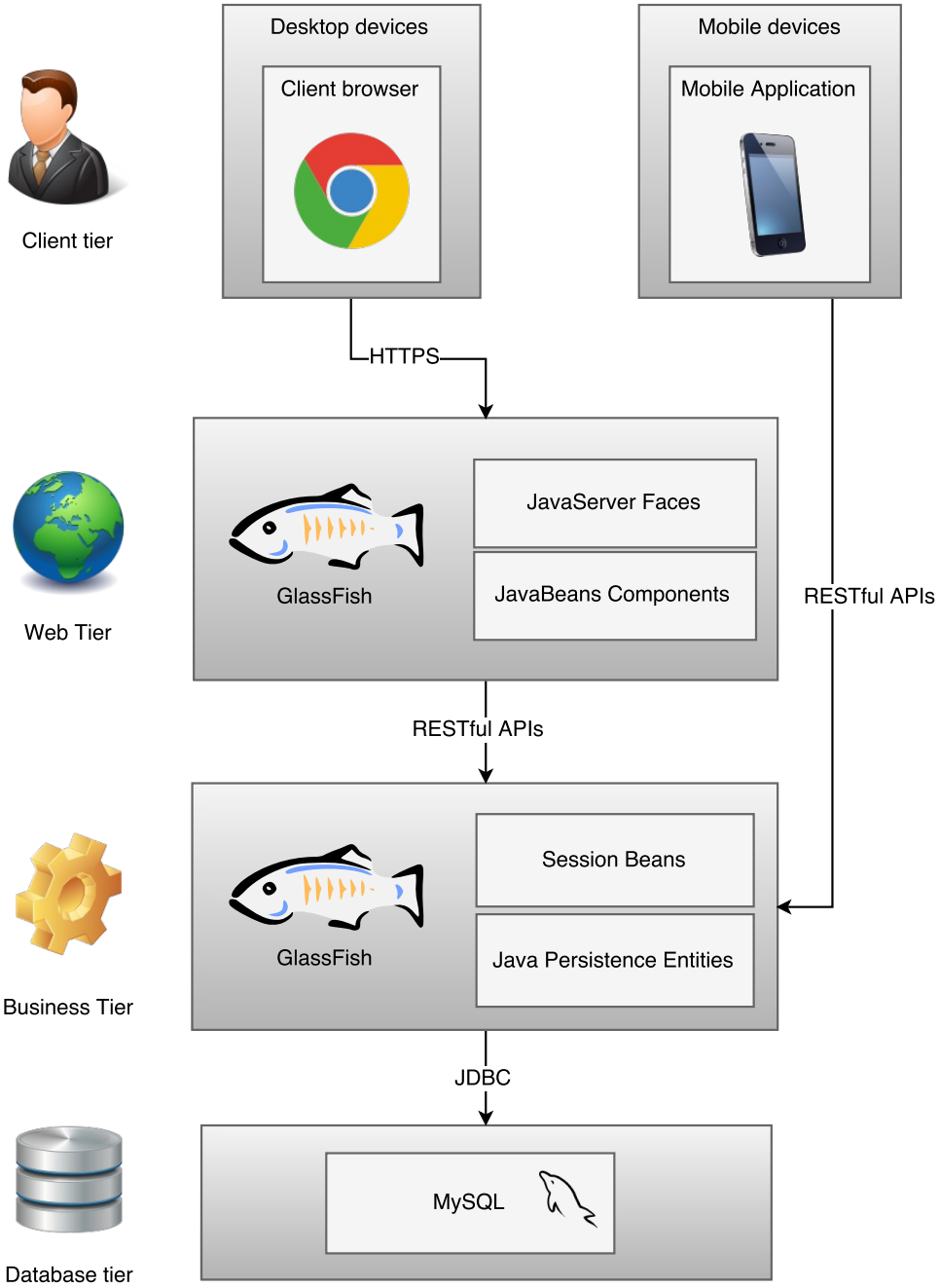


Figure 2.3: The detailed description of tiers, detailed with JEE components.

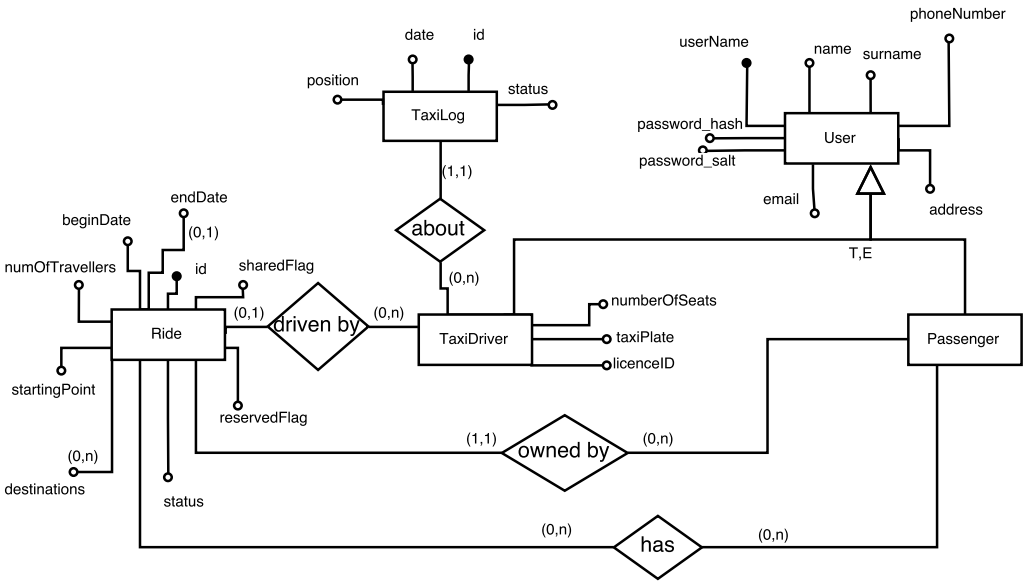


Figure 2.4: The Entity-Relationship diagram of the database schema.

must do so with the minimum level of privilege needed to perform the operations.

All the persistent application data is stored in the database. The conceptual design of the database is illustrated by the E-R diagram in Figure 2.4.

Foreign key constraints and triggers are not used: the dynamic behaviour of the data is handled entirely by the Java Persistence API in the Business Application tier.

2.3.2 Application server

The application server is implemented in the business logic tier using Java EE; it runs on GlassFish Server.

The access to the DBMS is not implemented with direct SQL queries: instead, it is completely wrapped by the **Java Persistence API (JPA)**. The object-relation mapping is done by entity beans.

The Entity Beans representing the database entities (Figure 2.5) are strictly related to the entities of the ER diagram (Figure 2.4).

The business logic is implemented by custom-built **stateless Enterprise JavaBeans (EJB)**. Our application indeed is rather simple and message-driven: the state of the users is stored in the DB, so we do not generally need stateful EJBs which can be more expensive. Concurrency management and performance are fundamental, so the reuse of EJBs for many requests is

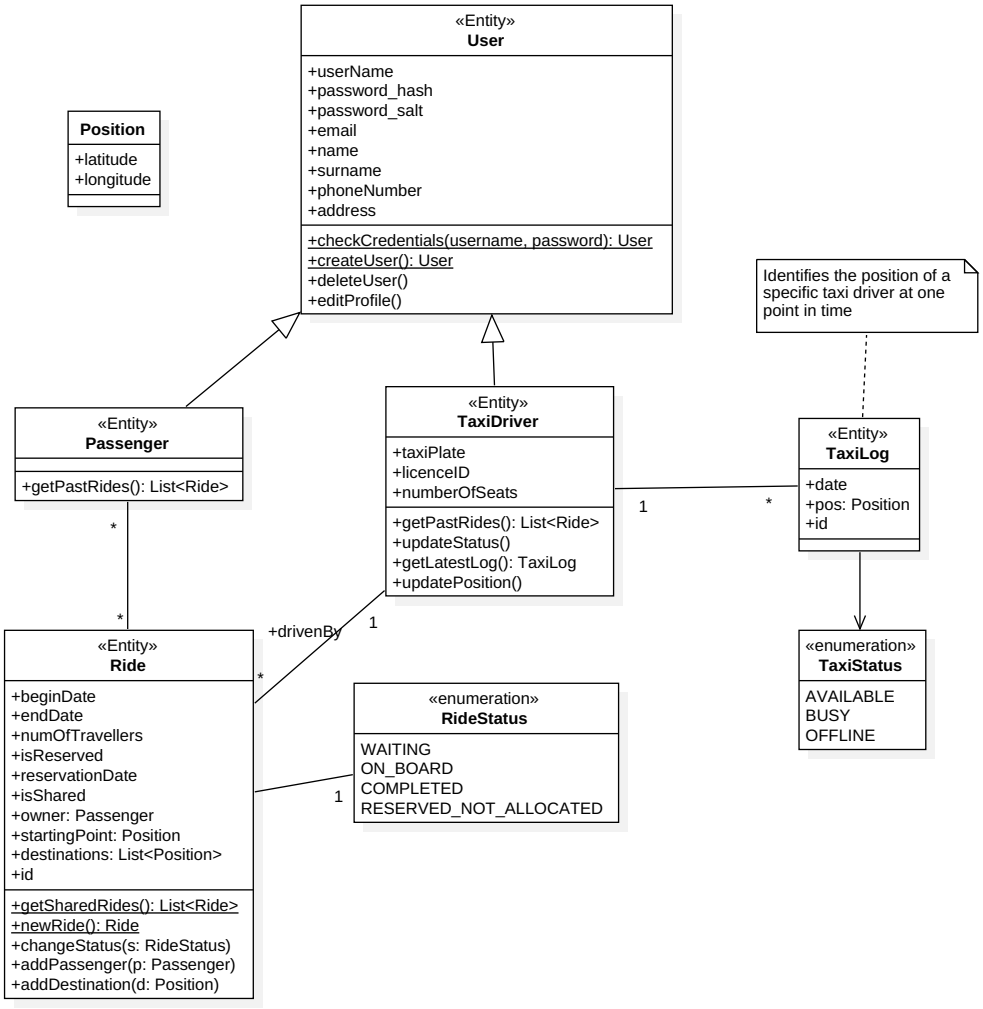


Figure 2.5: Java Entity Beans used to represent the entities in the database.

a desirable behaviour.

The application server implements a RESTful API using JAX-RS to allow the clients (web tier and mobile client) to use the services offered by the EJBs: this interface will be thoroughly discussed in subsection 2.6.2.

Session Beans used for the application server are shown in Figure 2.6.

UserManager

This bean manages all the user management features, namely: user login, user registration, user deletion, user profile editing. It also provides a function to confirm the email address provided by the user with the token sent by email.

RideManager

This bean creates new rides, assigns passengers and taxi driver to existing rides and fetches information about rides. It also allows taxi drivers to update the status of a ride, signaling when the passengers are on board and when the ride is finished.

TaxiManager

This bean handles the availability status and the position of each taxi driver. Taxi drivers can mark themselves as available or unavailable, and they can change their position as well. This component also has a method to find the first taxi in the queue for a specific taxi zone.

This component is stateless and can be instantiated multiple times. It calls TaxiQueueManager for data access.

TaxiQueueManager

This bean is a singleton and is in charge of keeping track of the taxi queue for each taxi zone. Taxi queues are not persistent data, so they are not stored in the database: they are only present in main memory. This allows a quicker access since this data is changed often.

The TaxiQueueManager keeps references to TaxiQueues and updates them when a taxi driver changes his status or his position. This component also has a method to find the first taxi in the queue for a specific taxi zone.

This component takes responsibility to ensure a correct concurrent access to the queues.

HistoryManager

This bean allows users (passengers and taxi drivers) to fetch the history of their past rides. This component is accessed by the RideManager and is able to store the rides which have ended.

EmailSender

This bean is in charge of sending emails to users when needed. For now its only functionality is to send the e-mail confirmation token to the user e-mail address after the registration, but in future implementations it could be extended to implement e-mail notifications.

SharedRidePlugin

This plugin implements all the functions related to the taxi sharing feature. It allows the passengers to find feasible shared rides and join them.

ReservedRidePlugin

This plugin implements all the functions related to the ride reservation feature. This component allows passengers to reserve a taxi for a future time.

ConfiguratorBean

This bean is a singleton and its only duty is to read the server configuration file and provide the value of configuration options to other components. Settings are stored as a set of key-value pairs. Other beans can ask for configuration options using the same keys used to define the settings in the configuration file.

2.3.3 Web server

The web server is implemented using Java EE web components, namely JavaServer Faces (JSF), which is a server-side framework based on MVC. The web server is run by GlassFish Server.

The web tier only implements the presentation layer: all the business logic is handled by the application server tier. The web tier uses the RESTful interface of the application tier.

Using JSF, the view is written as XML files and is completely separated from the logic of the web server. This enables us to write a modular web service.

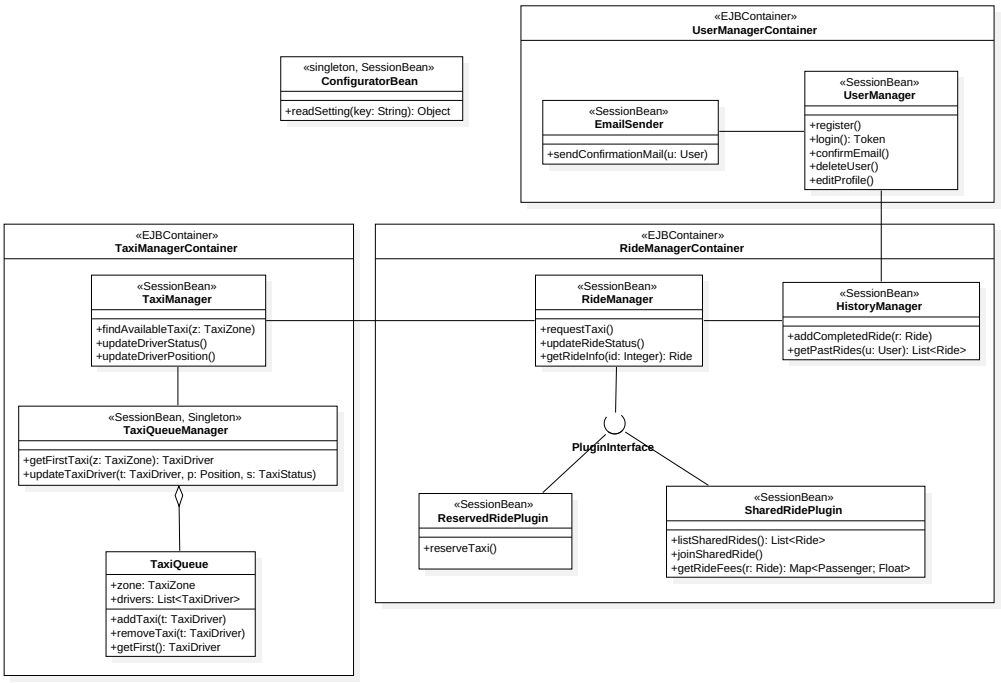


Figure 2.6: The session beans used in the implementation of the business logic. The parameters of the methods are not written here: the detailed interface of each bean is described in detail in subsection 2.6.2

.

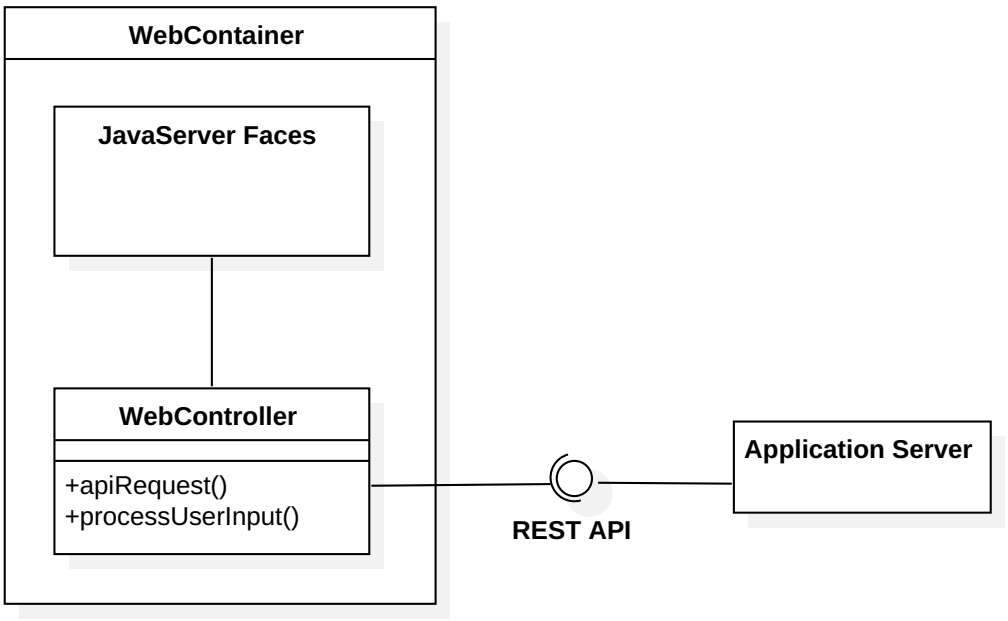


Figure 2.7: The components of the web tier.

The web server architecture is composed simply by JSF (as the view) and by a controller class that takes the user input and translates it in API requests (Figure 2.7).

2.3.4 Mobile client

The mobile client implementation depends on the specific platform. The iOS application is implemented in Swift and mainly uses **UIKit** framework to manage the UI interface. Instead, the Android application is implemented in Java and mainly uses **android.view** package for graphical management.

The application core is composed by a controller which translates the inputs from the UI into remote functions calls via RESTful APIs. The controller also manages the interaction with the GPS component using **CoreLocation** framework in iOS app and **LocationListener** interface in the Android one.

The main structure of the mobile application is shown in figure Figure 2.8.

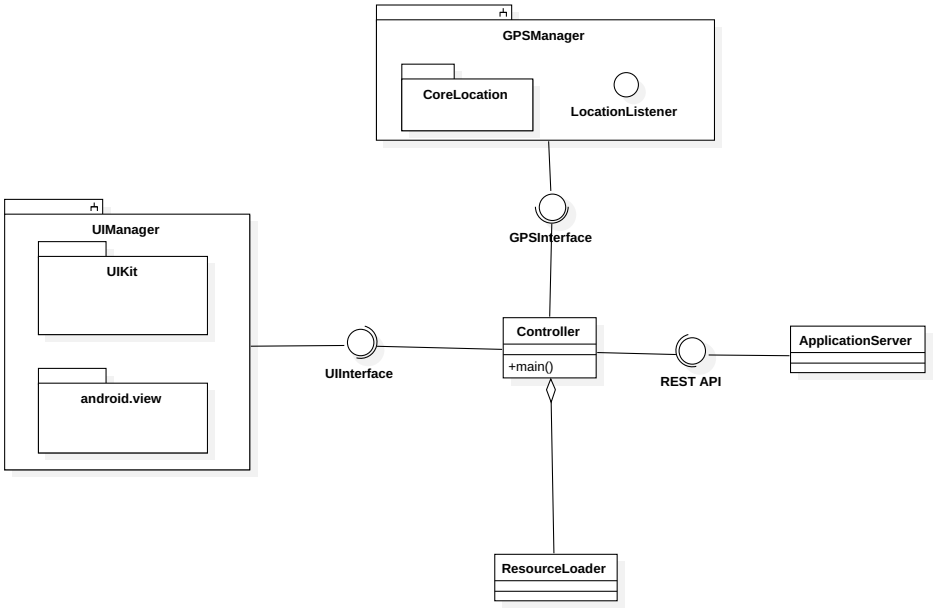


Figure 2.8: The components of the mobile app.

2.4 Deployment view

The deployment diagram for the system is shown in Figure 2.9. The web server and the business logic layers can be allocated to the same physical machine (tier), but in the general case they are on different hosts.

GlassFish Server is used as the Java EE application server for both the business logic and the web tier.

2.5 Runtime view

In this section we will describe the dynamic behaviour of the system. In particular, it will be shown how the software and logical components defined in section 2.3 interact one with another, using *sequence diagrams* for the more meaningful functionalities of the system. We decided not to represent the database in the sequence diagram, because the interaction with the database is totally abstracted by the entities via the Java Persistence API.

We decided not to include *runtime unit diagrams* because we work at a higher abstraction level and do not know how the application server (GlassFish) will instantiate and handle its processes and threads.

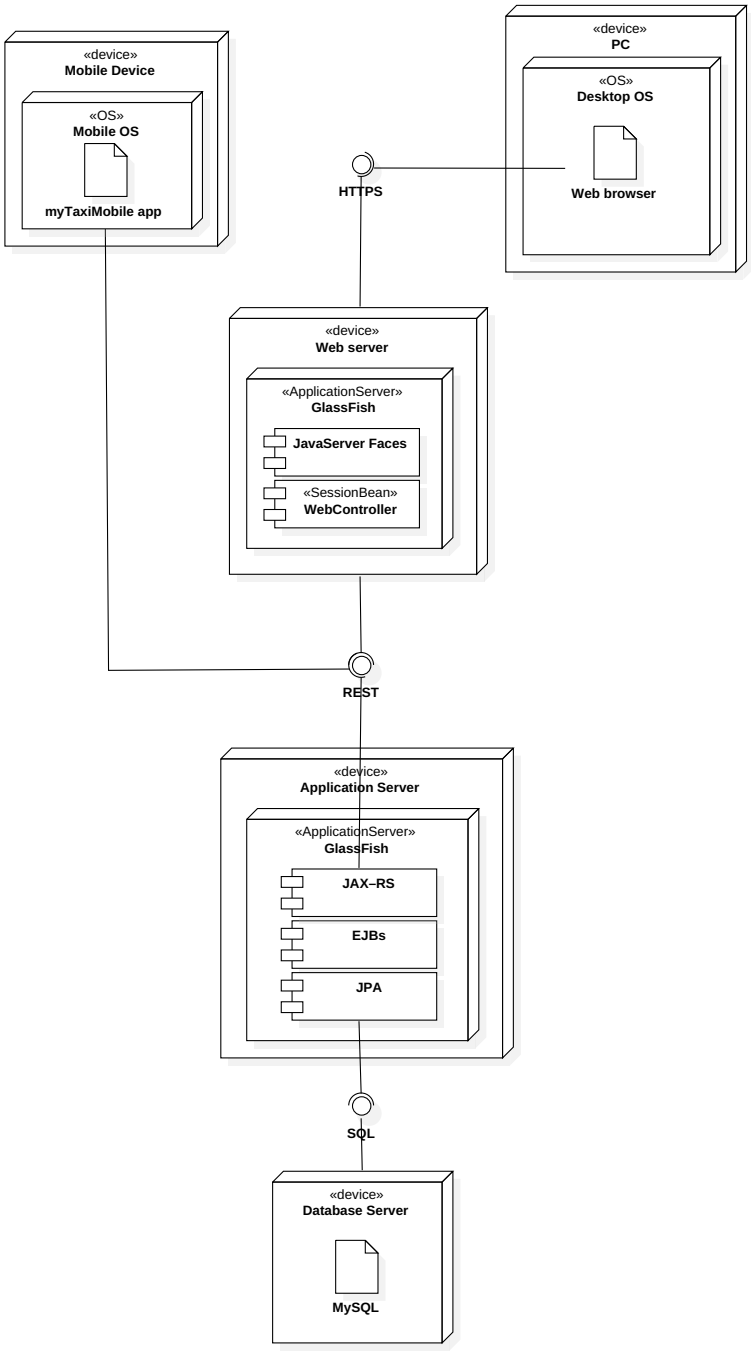


Figure 2.9: The deployment diagram for our application.

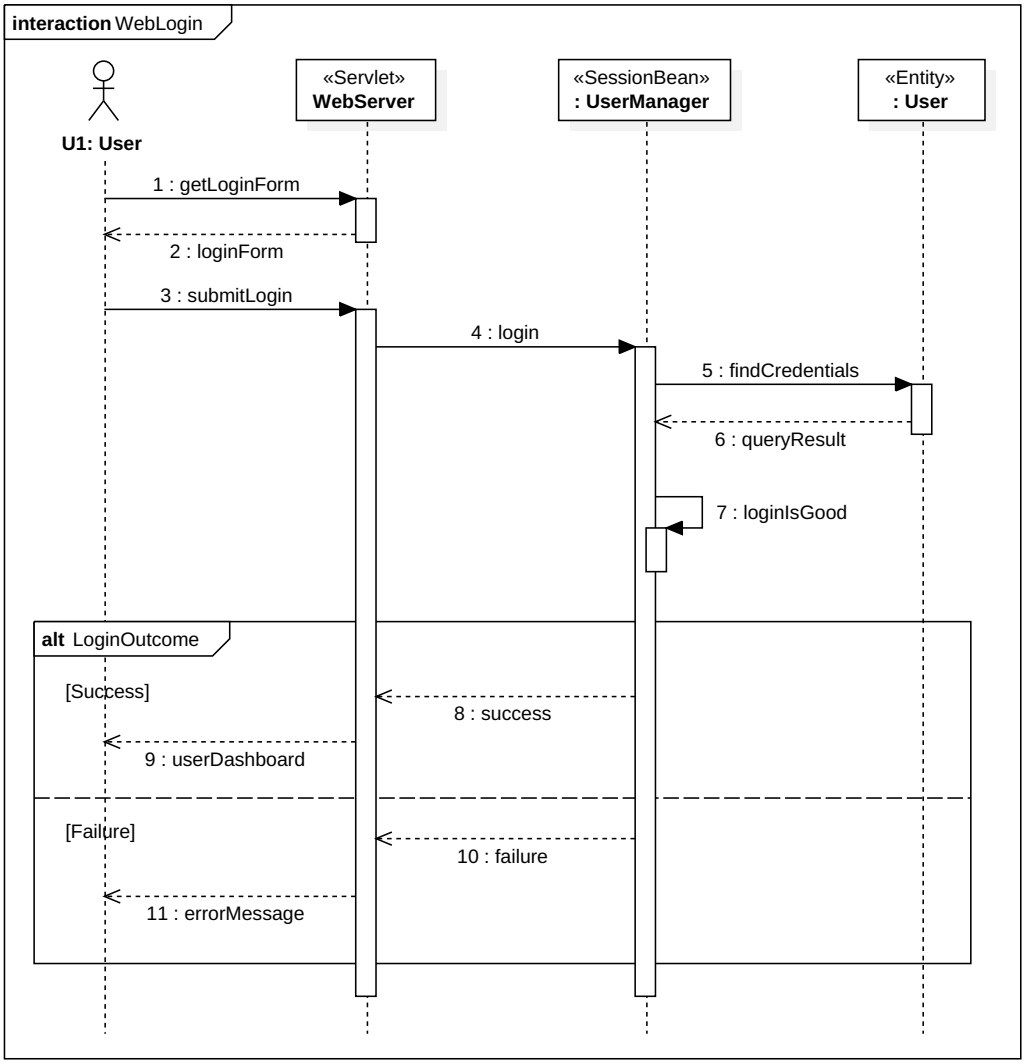


Figure 2.10: Sequence diagram of the login with the web interface.

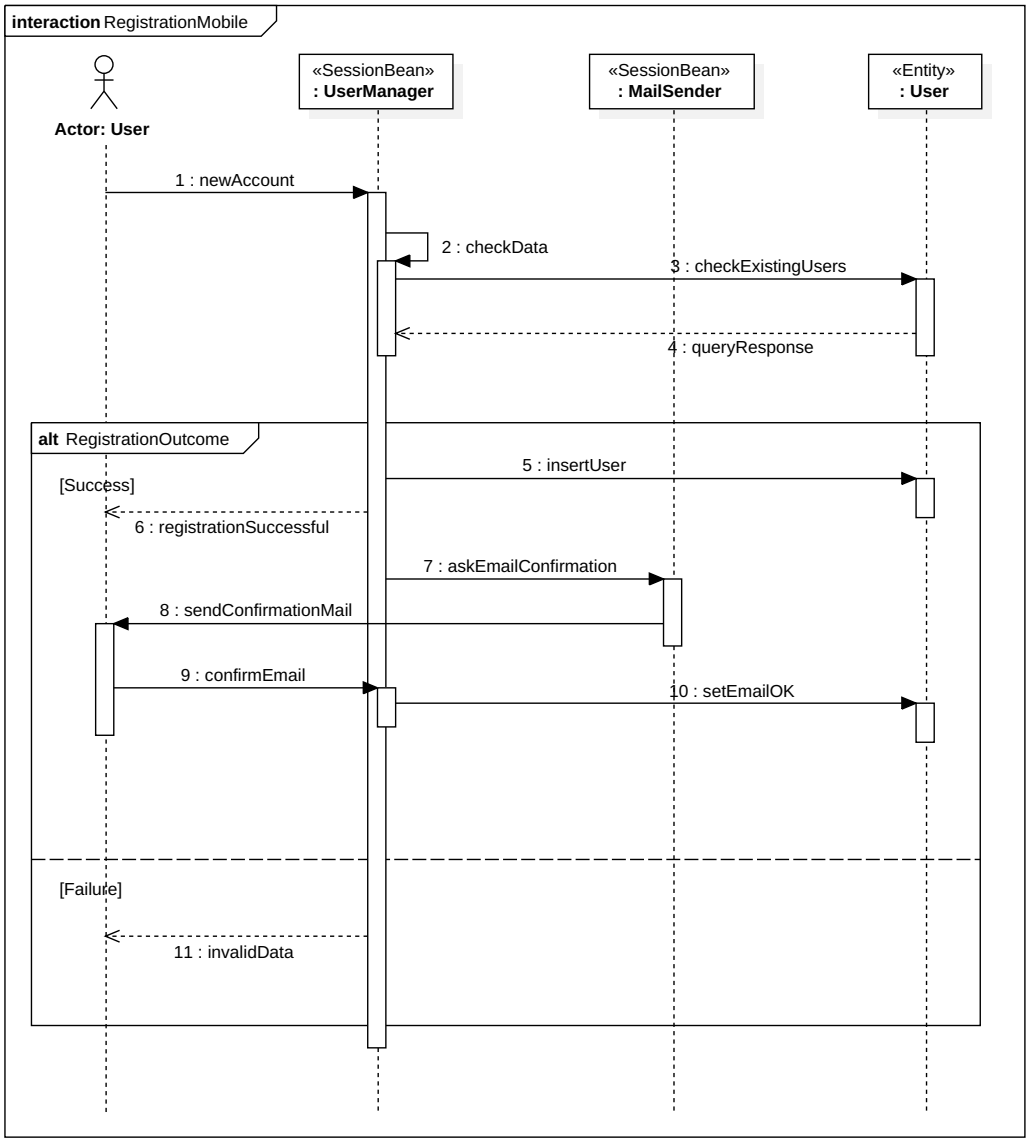


Figure 2.11: Sequence diagram of the registration from a mobile client.

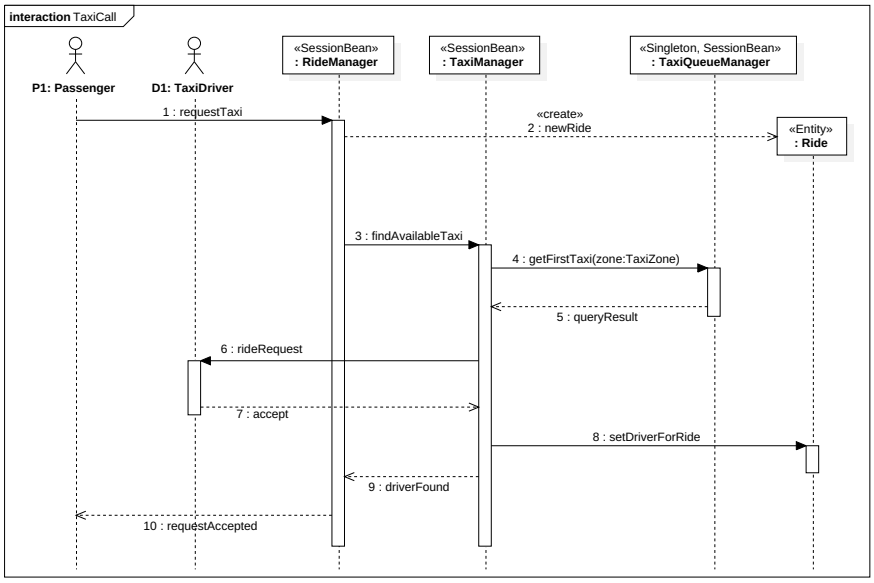


Figure 2.12: Sequence diagram of a taxi call.

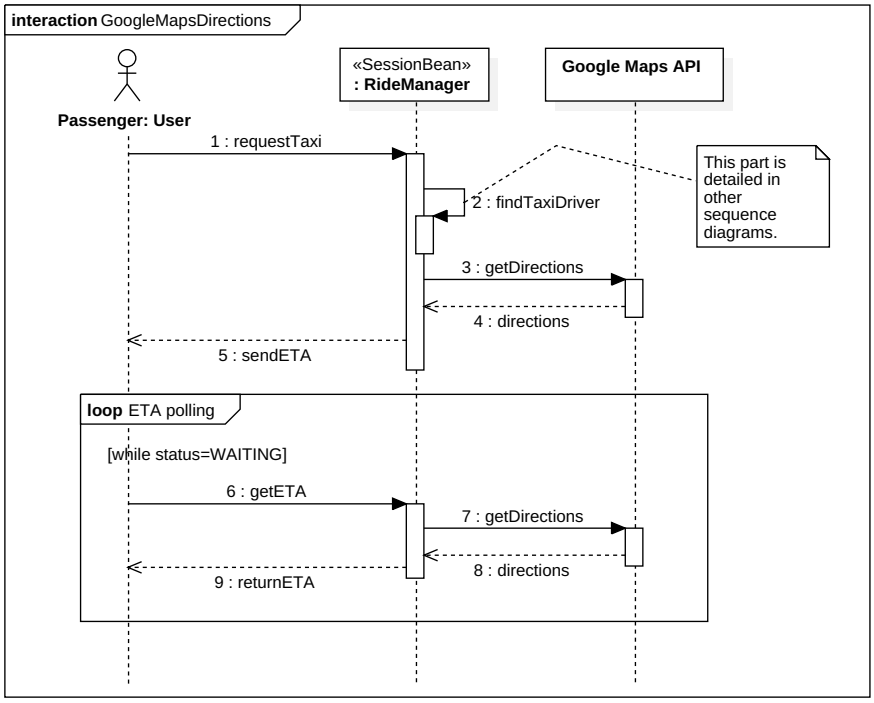


Figure 2.13: Sequence diagram of the waiting time service, that uses the Google Maps Directions API detailed in section 3.4.

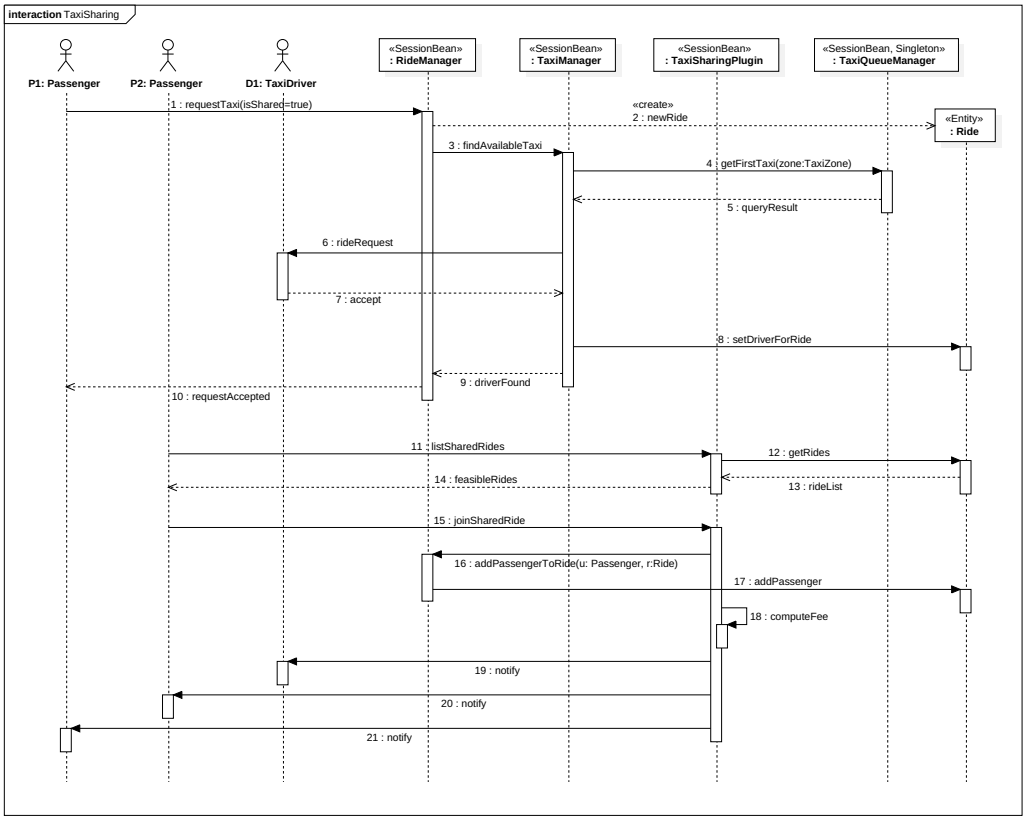


Figure 2.14: Sequence diagram of a shared ride (2 passengers).

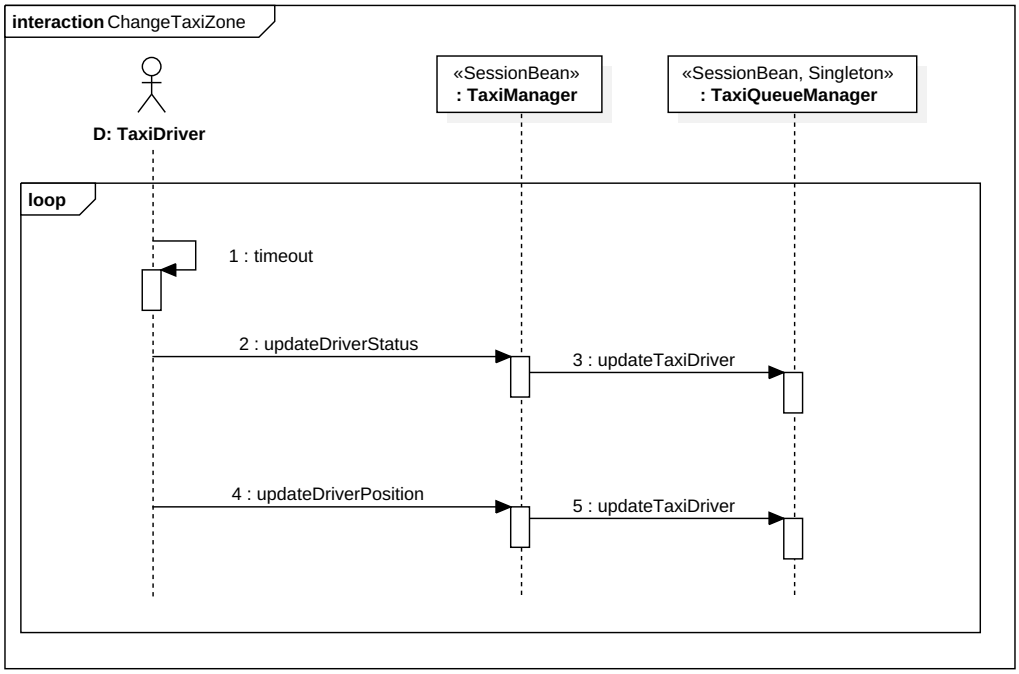


Figure 2.15: Sequence diagram of a taxi driver's position update.

2.6 Component interfaces

2.6.1 Application server to database

The application server communicates with the DBMS via the Java Persistence API over standard network protocols. Thus, the DB and the application server layers can be deployed on different tiers, as well on the same one.

The low-level technicalities about the specific dialect of SQL for the selected DBMS are abstracted by the Java Persistence API, which also deals with the O/R mapping.

2.6.2 Application server to front-ends (REST API)

The front-ends of the system (the web application and the mobile app) shall communicate with the application server using the back-end programmatic interface described in the RASD [4] and implemented as a RESTful interface over the HTTPS protocol. The RESTful interface is implemented in the application server using JAX-RS and uses XML as the data representation language.

The detailed REST API implemented by the core system and by the plugins is described in the following pages.

The conventions used for defining the REST API are the following:

1. Each Session Bean can offer some functionalities as part of the public API.
2. Plugins can add other functionalities and extend the existing ones.
3. Functions can return values and one or more errors.
4. Return values are highlighted in red.
5. Extensions by plugins are highlighted in blue.
6. Position data consists in an ordered tuple containing two floating point values (latitude and longitude).
7. Features can be available to all users (**A**), logged in users (**U**), passengers (**P**) or taxi drivers (**D**) only.
8. The features only available to logged in users require a login token, provided by the `login` function.

UserManager

Functions implemented by **UserManager** (Table 2.1):

`login` This function allows any registered user to log into the system using his username and password. If the credentials are correct, the function returns a token to be used in the future requests to identify the user. Otherwise, an error is returned.

`register` This function creates a new user in the system with the provided data. This function can create both passengers and taxi drivers. If the entered data is correct, an email is sent to the user address to confirm the email address.

`confirm_email` This function confirms the email address of a newly registered user using the token sent by email after the registration.

`delete_user` This function deletes the user account with all the related data. As an extra security measure, username and password are required even if the user is logged in.

`edit_profile` This function allows users to edit their profile information. If a new email is entered, the same confirmation process of `register` will be followed.

RideManager

Functions implemented by **RideManager** (Table 2.2):

`get_ride_info` This function returns all the information about a ride. A user can obtain information about a ride if and only if one of these conditions is met:

1. the user is one of the passengers of the ride;
2. the user is the taxi driver whom the ride is assigned;
3. the ride is shared and new passengers can join.

An error is returned if the ride ID is invalid or if the user is not authorized to access the data.

The function fetches the following information for a ride: the number of passengers, the number of travellers, the origin and destination of the ride (if available) and the ride status. The status of the ride can

Service	Users	Parameters and return values	
All services	A	`token`	Authentication token.
		`errors`	List of errors.
`login`	A	`user_name`	Username of the registered user.
		`password`	Password of the user.
		`token`	Authentication token to be used in future requests.
`register`	A	`user_name`	Username to register.
		`password`	Password chosen by the user.
		`email`	E-mail address chosen by the user.
		`type`	`passenger | driver`
`confirm_email`	U	`user_name`	Username of the registered user.
		`email_token`	Confirmation token sent by e-mail.
`delete_user`	U	`user_name`	Username of the registered user
		`password`	Password of the user.
`edit_profile`	U	`user_name`	New username.
		`password`	New password.
		`email`	New email.
		one parameter for each profile information.	

Table 2.1: REST API implemented by the UserManager bean.

be: reserved (the ride has been reserved but there is not yet any taxi driver allocated to it), waiting (the taxi driver is going to meet the passenger), running (passengers on board), done (the ride ended).

`request_taxi` This function allows a passenger to call a taxi. The passenger has to specify its position along with the number of travellers, and if he wants to share the ride. If the sharing feature is enabled, the passenger has to provide a destination. If the taxi call is successful, the function returns the ID of the newly created ride. An error is returned if the data is not valid or if the passenger is currently in a ride or waiting for a taxi.

The taxi sharing feature is added by the ride sharing plugin.

`update_ride_status` This function is used by the taxi driver client to send messages about a ride. The taxi manager can signal that he has picked up all the passenger or that the ride is ended. If the message if inconsistent with the current status of the ride, an error is returned.

This function can be extended with other types of messages or events that can be sent by the taxi driver.

TaxiManager

Functions implemented by **TaxiManager** (Table 2.3):

`update_status` This function is used by the mobile client of taxi drivers to update the status of the driver. The taxi driver can become unavailable only if he is not currently assigned to a ride.

`update_position` This function is used by the mobile client of taxi drivers to periodically update the position of the driver.

HistoryManager

Functions implemented by **HistoryManager** (Table 2.4):

`get_past_rides` This function returns the information (in the same format of `get_ride_info`) about the rides in which the user was involved (as a taxi driver or a passenger).

Service	Users	Parameters and return values	
All services	A	`token`	Authentication token.
		`errors`	List of errors.
`update_ride_status`	D	`ride_id`	ID of the ride.
		`ride_event`	`passengers_on_board` | `ride_finished`
		`ride_info`	The output of `get_ride_info` for the ride `ride_id`.
`get_ride_info`	U	`ride_id`	ID of the ride returned by `request_taxi`.
		`origin`	Position of the starting point.
		`destinations`	Positions of the destinations.
		`num_travellers`	Total number of travellers.
		`num_passengers`	Total number of passengers.
		`status`	`reserved` | `waiting` | `running` | `done`
		`wait_time`	Estimated waiting time in seconds. Returns -1 if there is no taxi to wait for.
`request_taxi`	P	`origin`	Position of the passenger.
		`travellers`	The number of travellers.
		`destination`	Destination of the ride. Only required if `sharing_enabled=true`.
		`sharing_enabled`	`true` | `false`
		`ride_id`	An ID number identifying the ride.

Table 2.2: REST API implemented by the RideManager bean.

Service	Users	Parameters and return values	
All services	A	`token`	Authentication token.
		`errors`	List of errors.
`update_status`	D	`status`	`unavailable` | `available` | `busy`
`update_position`	D	`position`	The position of the taxi driver.

Table 2.3: REST API implemented by the TaxiManager bean.

Service	Users	Parameters and return values	
All services	A	`token`	Authentication token.
		`errors`	List of errors.
`get_past_rides`	U	`rides`	List of rides in the `get_ride_info` format.

Table 2.4: REST API implemented by the HistoryManager bean.

Plugins

Functions implemented by the **plugins** *ride sharing* and *ride reservation* (Table 2.5):

`reserve_taxi` This function is implemented by the *ride reservation* plugin and allows any passenger to reserve a ride for a future time. The submitted timestamp must be between 2h and 48h after the current time. If the sharing feature is enabled, the passenger has to provide a destination.

`list_shared_rides` This function is implemented by the *ride sharing* plugin and lists all the shared rides compatible with the chosen destination, date and time, and number of travellers. If the data is valid, the function outputs a list with the feasible ride information in the same format of the `get_ride_info` request. If no feasible ride is found, the function returns the empty list. If the data is invalid, an error is returned.

`join_shared_ride` This function is implemented by the *ride sharing* plugin and allows a passenger to join a shared ride, adding him to the list of passengers of the ride and adding his destination to the ride destinations. A passenger can join a ride only if the ride is "feasible" as defined by the algorithm described in section 3.2. The passenger has to provide a destination, along with the number of passengers and the ID of the ride he wants to join. Feasible rides can be obtained by calling `list_shared_rides`. If the provided data is not valid or the ride is not feasible, an error is returned. If the request is successful, the function returns the ride information in the same format of `get_ride_info`.

`get_ride_fee` This function is implemented by the *ride sharing* plugin and allows the taxi driver and the passengers of the ride to know the percentage of the taxi fee that each passenger has to pay, according to the algorithm described in section 3.3.

The function returns a dictionary containing as keys the passengers' usernames and as values the percentage of the fee that each one has to

pay, expressed as a floating point number from 0 to 1. If the function is called on a ride that has only one passenger, the fee percentage for that passenger will be 1.

This information is only available to the taxi driver and to the passengers of the ride.

2.6.3 Configuration file (application server)

The application server is configurable by means of a XML configuration file. The configuration file contains the following settings:

- the boundaries of the taxi zones, expressed as polygons (list of coordinates);
- the credentials of the user that can access the database;
- the host, port and name of the database;
- the network settings of the application server (listening port, host, ...);
- any other settings that will be useful in the implementation phase.

2.6.4 Web server to browser

The users' browsers communicate with the web server via HTTPS requests. Any unencrypted request will be denied, as stated in the RASD [4].

2.6.5 Plug-in interface

The application server exposes extension points to be used by plug-ins: they are the only points in which the plug-ins can access the system. For example, they may not query the database layer directly. Plug-ins may add new services to the REST API of the application server.

Plug-in declaration, discovery and activation are made explicit by means of manifest files and plug-in registry.

2.7 Selected architectural styles and patterns

The following architectural styles have been used:

Client&server The client & server style is used, in our design, at multiple levels:

Service	Users	Parameters and return values	
All services	A	`token`	Authentication token.
		`errors`	List of errors.
`reserve_taxi`	P	`origin`	The position of the starting point of the ride.
		`destination`	The position of the passenger's destination.
		`travellers`	The number of travellers.
		`time`	Time of arrival of the taxi (ISO 8601 format).
		`sharing_enabled`	`true | false`
		`ride_id`	An ID number identifying the ride.
`list_shared_rides`	P	`origin`	The position of the starting point.
		`destination`	The position of the passenger's destination.
		`travellers`	The number of travellers.
		`time`	Time of arrival of the taxi (ISO 8601 format).
		`rides`	A dictionary containing a list of feasible rides, with the same format of `get_ride_info`.
`join_shared_ride`	P	`ride_id`	The ID of the ride.
		`destination`	The position of the passenger's destination.
		`ride_info`	The output of `get_ride_info` for the ride `ride_id`.
`get_ride_fee`	U	`ride_id`	The ID of the ride.
		`fees`	Fee percentage for each user, as a dictionary `{username` ⇒ `Float}`.

Table 2.5: REST API implemented by the plugins *taxi reservation* and *taxi sharing*.

- the application server (client) queries the DB (server);
- the web front-end (client) communicates with the application server;
- the user's browser (client) communicates with the web server.

The separation of the application server from the business tier means that if the web server fails for any reason, the back-end interface will be accessible by the users by means of the mobile application.

Service-oriented Architecture The SOA is used by the system for the communication between the application server and the front-ends.

The SOA allows to think at a higher level of *abstraction*, by looking at the component interfaces and not at their specific implementation.

SOA style also improves *modularity*: by making service description, discovery and binding explicit, it is easier to build new plugins and test single modules independently.

Also, SOA makes it easier to document and maintain the APIs, and simplifies the development of clients.

Plug-ins The plug-in style is used to add functionalities to the application server.

The advantage of this choice is modularity: the core application server can be stripped down to the bare minimum, and then new functionalities can be developed, tested and added separately, with no need to touch the core.

Also, the plug-in architecture allows the system to conform to different needs in different installations: not all the features are always needed nor wanted.

By not building additional features directly into the core, performance is improved when those features are not needed.

The stability of the system is preserved by allowing the plug-ins to interact with the system only in specific extension points.

Thin client The thin client paradigm is implemented with relation to the interaction between user's machine and the system. Having a thin client in our case is an advantage because all the application logic is on the application server, which has sufficient computing power and is able to manage concurrency issue efficiently. Also, updates to the software are easier.

This architectural choice makes it possible for users with devices with limited processing power (i.e. mobile phones) to use our service with an acceptable performance.

Distributed presentation Distributed presentation is the design choice for the web front-end. As mentioned before, the *thin client* approach has been selected, so all the data and the application logic are on the side of the system.

The presentation layer is split among two tiers:

1. the web server generates the web pages and serves all the needed resources (images, styles, scripts);
2. the user's browser interprets and renders the web page, also executing some client-side code (which does not implement application logic).

Model-View-Controller The clients (web front-end + mobile application) are built following the Model-View-Controller design pattern. MVC is the design pattern of choice because it is the most common and the most convenient pattern used with object-oriented languages (including Java) dealing with complex application, and allows to design software with the *separation of concerns* principle in mind.

2.8 Other design decisions

2.8.1 Storage of passwords

Users' password are not stored in plain-text, but they are hashed and salted with cryptographic hash functions. This provides a last line of defense in case of data theft.

2.8.2 Maps

The system uses an external service, *Google Maps*, to offload all the geolocalization, distance calculation and map visualization processes. The reasons of this choice are the following:

- manually developing maps for each city is not a viable solution due to the tremendous effort of coding and data collection required;
- Google Maps is a well-established, tested and reliable software component already used by millions of people around the world;

- Google Maps offers APIs, enabling programmatic access to its features (SOA approach);
- Google Maps can be used both on the server side (calculation, shortest paths, traffic, incident reporting) and on the client side (map visualization);
- the users feel comfortable with a software component they know and use everyday.

2.8.3 Availability and redundancy

In the RASD (section 3.4.2) we stated that we had to ensure an availability of 98%, i.e. 7,3 days of downtime a year. We did not specify a greater availability because the architecture we designed is not redundant: the tiers are not replicated at any level. In future implementations, replication can be added (for example, at the database and web tiers), improving the availability.

Chapter 3

Algorithm Design

3.1 Taxi queue management

As stated in the RASD [4], the taxi drivers are divided in city zones according to their current location. Each zone is served by a local queue of taxi drivers who are enabled to accept ride requests and serve the passengers. This section elaborates in detail how the queues are managed by the system.

The algorithm is run every time there is a new event which changes the order or the number of the elements in one of the taxi queues.

According to the type of the event, the algorithm updates the queues moving, deleting or creating elements. There are several types of events:

1. new request incomes;
2. taxi driver accepts a request;
3. taxi driver refuses a request;
4. taxi driver changes zone;
5. taxi driver changes his/her status.

The input data are:

- Q_i: list of FIFO queues, where i is the zone of the correspondent queue.
- e: event which will change the queues.
- OOS: list of taxi drivers out of service.
- B: list of busy taxi drivers.
- P: list of taxi drivers with a pending ride.

The algorithm works differently accordingly with the input event. The alternatives are explained in the following subsections.

3.1.1 Type 1, new request incomes

The algorithm extracts the zone z of the new incoming request, pops the first element of the Q_z and inserts this element in the P list.

3.1.2 Type 2, taxi driver accepts a request

The algorithm extracts the taxi driver t from the event and removes him/her from P. After that it inserts t into B.

3.1.3 Type 3, taxi driver refuses a request

The algorithm extracts the taxi driver t from the event and it removes him/her from P. After that it retrieves the taxi driver's actual zone z and it pushes t in Q_z.

3.1.4 Type 4, taxi driver changes zone

The algorithm extracts the taxi driver t, the previous zone pz and the next zone nz from the event and removes t from Q_{pz} and pushes t into Q_{nz}.

3.1.5 Type 5, taxi driver changes status

If the taxi driver t changes his/her status from "not in service" to "in service", the algorithm extracts the zone z from the event, it removes t from OOS and pushes t into Q_z.

If the taxi driver t changes his/her status from "in service" to "not in service", the algorithm extracts the zone z from the event, it removes t from Q_z and pushes t into OOS.

3.2 Taxi sharing matching

The algorithm is run by the back-end when a new shared ride is requested and it computes the compatible shared rides with the new one.

A ride can be compatible with a new one if its starting point is not further than $MaxBeginDistance$. $MaxBeginDistance$ is the parameter which contains the value expressed in meters of the max accepted distance between

the actual position of the passenger of a new ride and the starting point of another ride.

Furthermore, the eventual new passenger needs to have the time to reach the starting point.

The last requirement, in order to be a compatible ride, is that once the rides are matched, the estimated travel time of the two rides does not increase of $ExtraPercentage$ of the previous estimated travel time, where $ExtraPercentage$ is the parameter which contains the percentage of the max admissible increasing.

The input data:

- $MaxBeginDistance$: is the max distance which the passenger has to walk in order to reach the starting point of the shared ride.
- $ExtraPercentage$: is the max increase of time travel for every ride, expressed in percentage.
- SR: is the set containing only the shared rides not started yet.
- r: is the new ride.
- wv: is the feasible walking speed of a person.
- $currentTime$: is the current time of the system.

Let cr_i be an element of SR. $cr_i.startingPoint$ is the position at the beginning of the ride, $cr_i.duration$ is the estimated time needed for the ride and $cr_i.startingTime$ is the starting time of the ride.

A new ride, formed by two matched rides cr_1 and cr_2 is denoted by $cr_1 \star cr_2$. The starting point of the matched rides is the starting point of the first allocated ride, and the duration of the matched rides is computed from the starting point to the destination of the last passenger which will get off the taxi. $(cr_1 \star cr_2).startingTime$ is equal to $cr_1.startingTime$.

A ride cr_i is compatible if it fulfills every following condition:

- $|cr_i.startingPoint - r.startingPoint| \leq MaxBeginDistance$
- $(cr_i \star r).duration \leq cr_i.duration(1 + ExtraPercentage)$
- $(cr_i \star r).duration \leq r.duration(1 + ExtraPercentage)$
- $|cr_i.startingPoint - r.startingPoint|wv + currentTime \leq cr_i.startingTime$

Passenger	Traveller	Distance		Fee
	1	90		
Alice	2	20	115	64%
	3	5		
	1	25		
Bob	2	30	65	36%
	3	10		
Total		**180**	**180**	**100%**

Table 3.1: An example of the taxi fee splitting algorithm.

3.3 Taxi fee splitting

This algorithm is run by the back-end in the case of shared rides to compute the percentages of the taxi fee that each passenger has to pay.

The algorithm computes the fee percentages proportionally to the overall distance traveled by all the people (travellers) that each passenger on the taxi brings with himself.

The input data are:

- P: list of passengers;
- T_i: list of travellers associated with passenger i;
- $d(t)$: distance traveled by traveller t.

The algorithm computes f_i, that is the percentage of the fee that the passenger i has to pay. f_i is expressed as a decimal number, i.e. $0 \leq f_i \leq 1$.

The algorithm works as follows:

$$f_i = \frac{\sum_{t \in T_i} d(t)}{\sum_{p \in P} \sum_{t \in T_p} d(t)} \tag{3.1}$$

An example of an application of this algorithm is shown in Table 3.1.

3.4 Taxi waiting calculation

When a passenger requests a taxi and his request is assigned to a taxi driver, the system tells the user the estimated time of arrival (ETA) of the taxi.

URL	`http://maps.googleapis.com/maps/api/directions/xml`
`origin`	GPS coordinates of the taxi driver
`destination`	GPS coordinates of the passenger
`key`	API key
`mode`	`driving`
`departure_time`	`now`
`traffic_model`	`pessimistic`

Table 3.2: Parameters of the Google Maps API call for the estimation of the waiting time.

This delay is computed by using the Google Maps Directions API [6], called as a service from the business logic layer.

The flow of events is as follows:

1. The passenger requests a taxi and sends his current position to the system.
2. The system looks for an available taxi driver in the same zone of the passenger.
3. The request is accepted by one taxi driver, who sends his current position to the application server.
4. The application server computes the ETA by sending a request to Google Maps; the format of the request is specified in Table 3.2.
5. The application server sends to the passenger the ETA, when required by the client.
6. Points 4, 5 are repeated every 90 seconds until the taxi notifies that it has picked up the passenger.

A detailed view of the sequential interaction among the components was shown in the runtime views, in Figure 2.13.

The detailed API call sent to the Google Maps Directions API is shown in Table 3.2. We chose to adopt the pessimistic traffic model because, although taxi drivers often have reserved lanes, it's best to leave some safety margin of time.

Chapter 4

User Interface Design

4.1 UX diagram

The UX diagram shows the different screens of the User Interface of the clients and the interaction between them.

This diagram follows the UI requirements in RASD [4]. In order to access this interface, the user has to be logged to use myTaxiService.

The services common for all the users, like profile management, and passenger's ones, like taxi call, are accessible both via web or mobile app; however taxi driver's services are accessible only via mobile phone, indeed the taxi driver has to use the service while working in the taxi. This is why the UX is divided in two parts:

- the **white** one: represents the model common for web and mobile application
- the **colored** one: highlights the part of the diagram only for mobile application; in case of web application the TaxiDriver Home has not more operations than the User Home.

Figure 4.1 shows the UX diagram for the application.

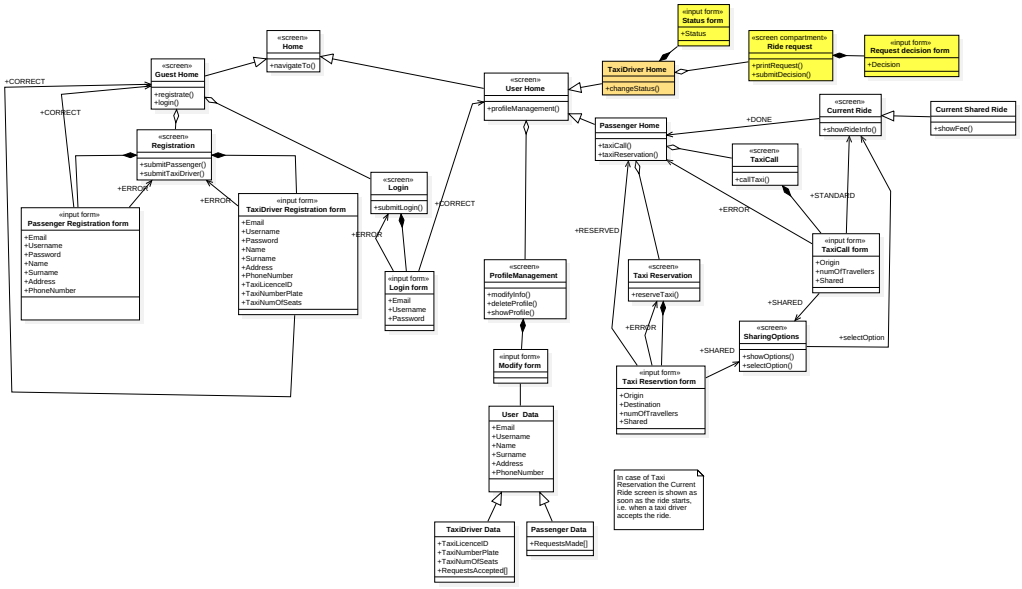


Figure 4.1: The UX diagram.

4.2 User Interface concept

4.2.1 Web interface

These mock-ups show how the user interface for the web application should look like on web browser, also with the possibility to choose the language in every screen.

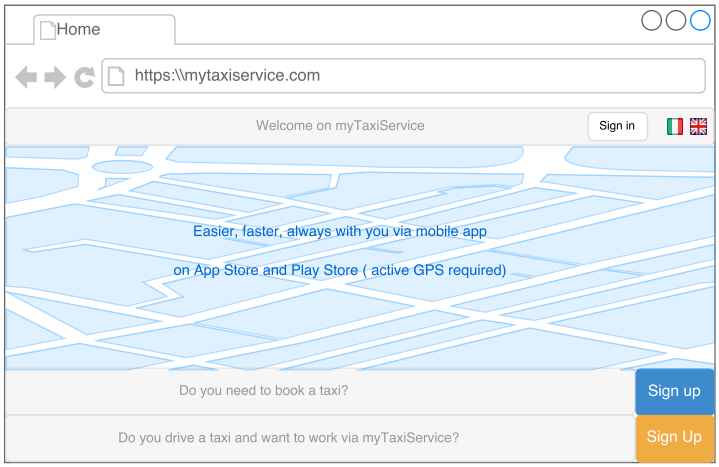


Figure 4.2: The home page of myTaxiService before log-in or registration.

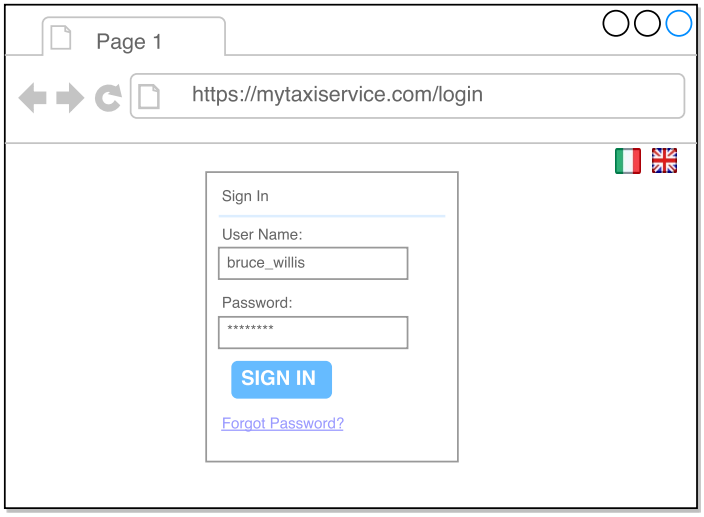


Figure 4.3: Login screen for the web application.

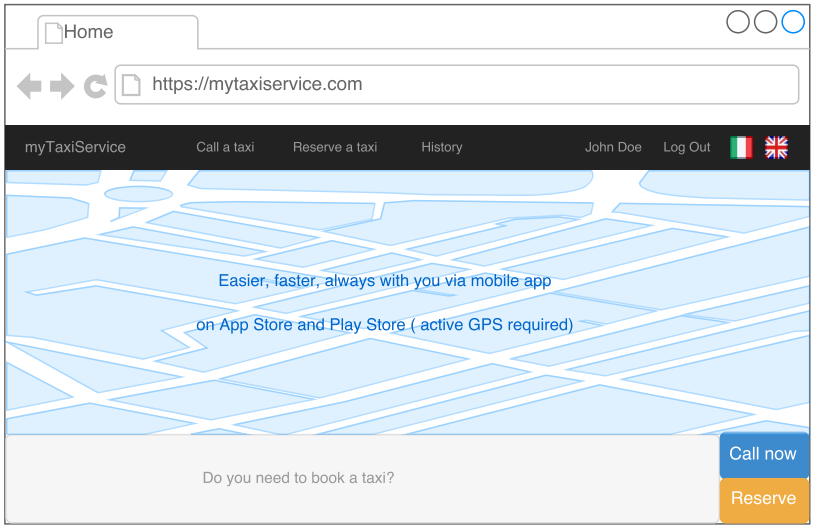


Figure 4.4: Home page for the logged in passengers linking to all the functionalities.

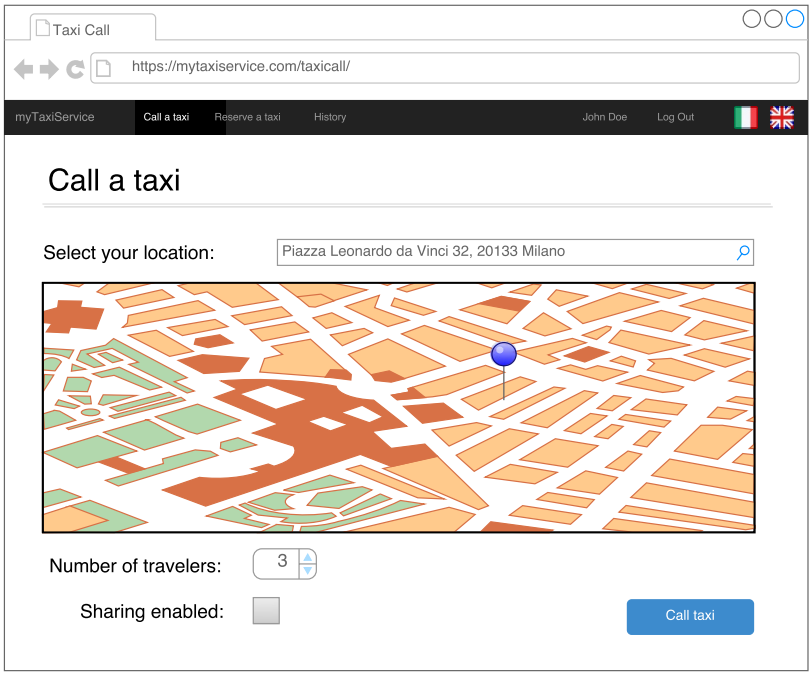


Figure 4.5: Taxi call screen in the web application.

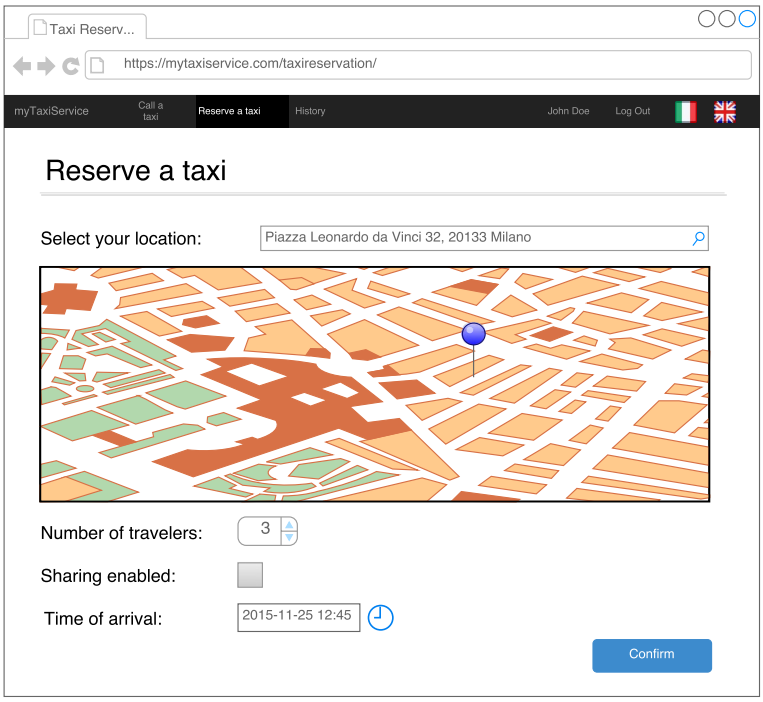


Figure 4.6: Taxi reservation screen in the web application.

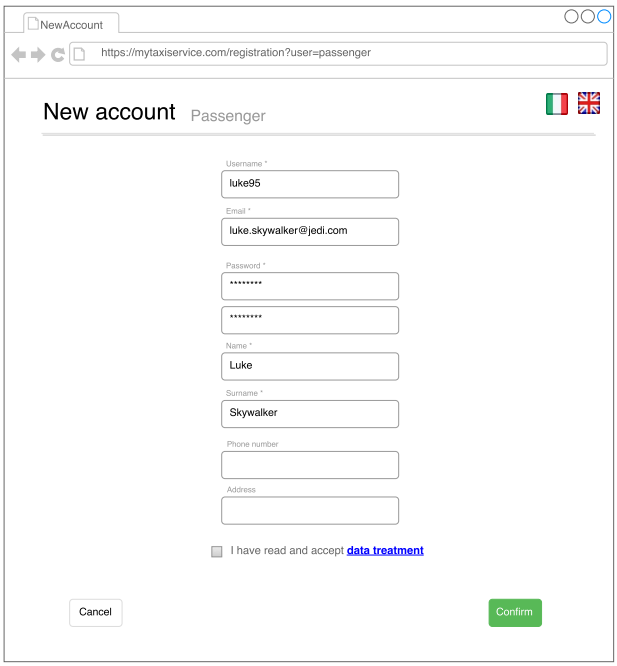
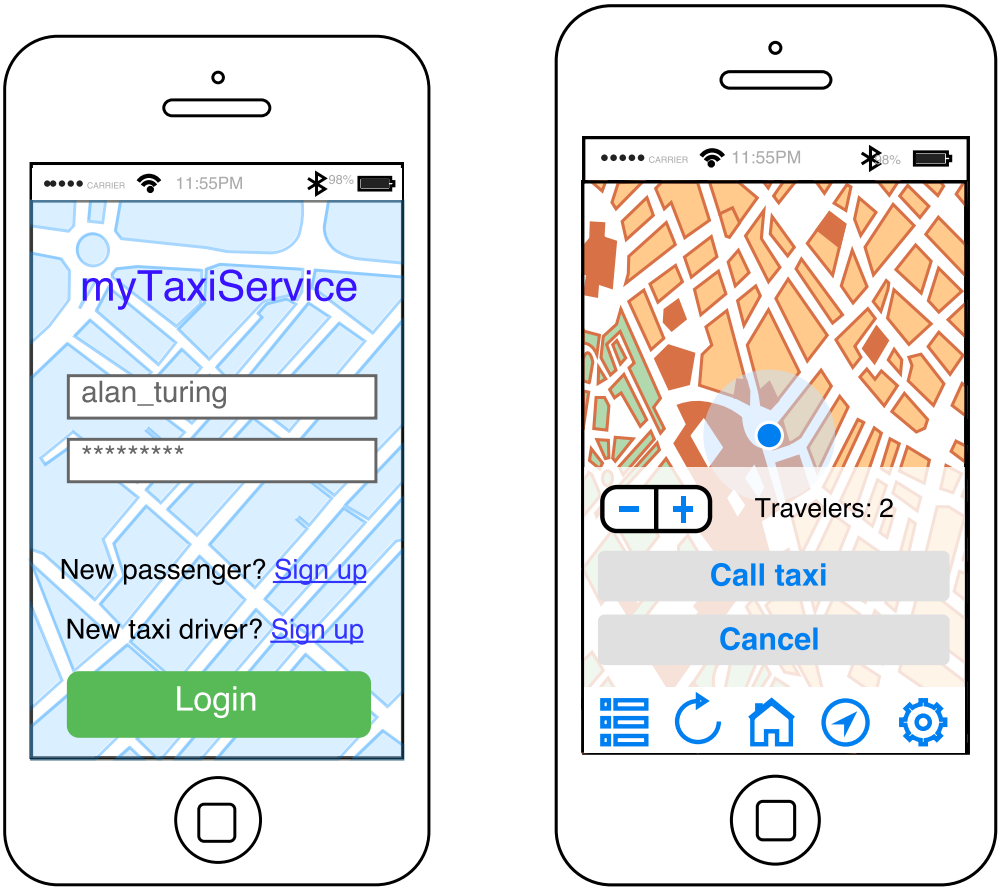


Figure 4.7: Registration screen for new passengers in the web application.

4.2.2 Mobile interface

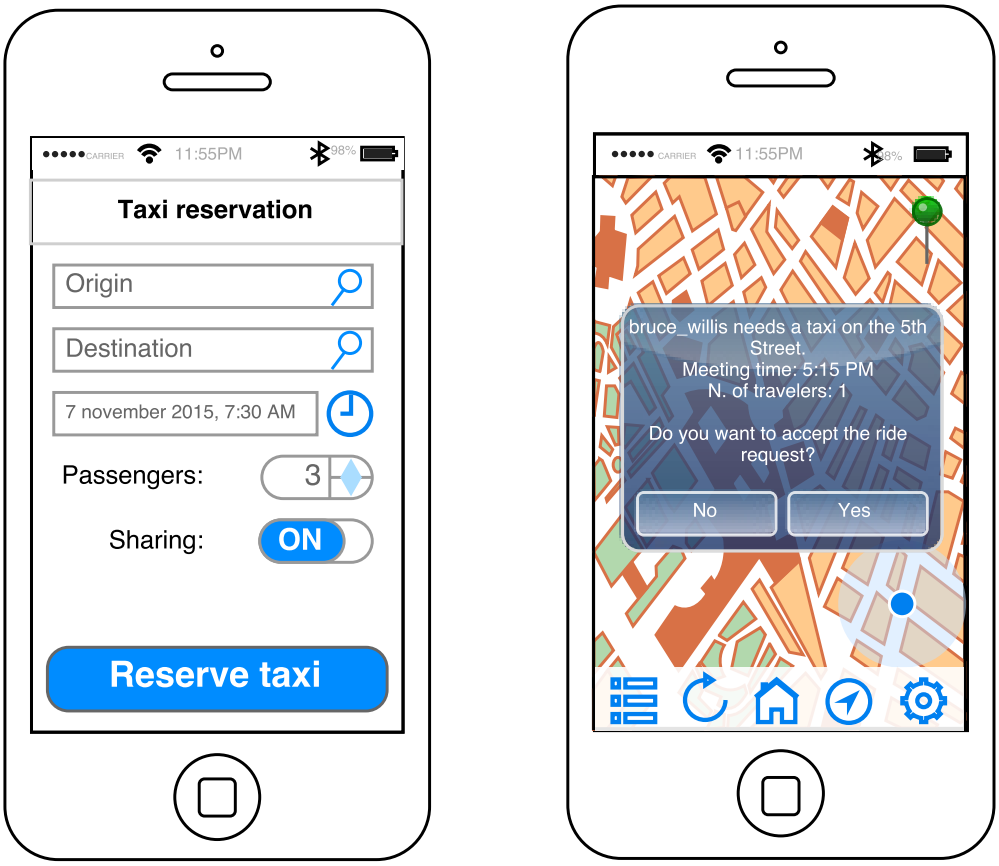
These mock-ups show how the interface of the myTaxiService mobile application will look like.



(a) Initial login screen on the mobile application. This screen also presents the option for signing up if the user does not have an account.

(b) Taxi call interface for the mobile application. The passenger may check if his location is correct and call a taxi specifying the number of travelers.

Figure 4.8: Login and taxi call screens for the mobile application.



(a) Taxi reservation screen for the mobile application.

(b) Ride request notification for taxi drivers in the mobile application.

Figure 4.9: Taxi reservation and ride request screens for the mobile application.

Chapter 5

Requirements traceability

All the decisions in the DD have been taken following functional and non-functional requirements written in the RASD. In the making of the Design Document, some changes were made to the RASD. They are explained in section 5.3.

5.1 Functional requirements and components

Table 5.1 maps the functional requirements contained in **specific sections** of the RASD to the **components** in the Design Document that fulfill them.

Component (DD)	**Requirements (RASD)**
EmailSender	3.2.1 e-mail confirmation process in User registration
UserManager	3.2.1 User registration 3.2.2 User login 3.2.8 User profile management.
HistoryManager	3.2.8 rides history in User profile management.
TaxiManager	3.2.4 Ride request notification to the taxi driver 3.2.5 Taxi availability handling.
TaxiQueueManager	3.2.5 Taxi availability handling (taxi queue management)
TaxiQueue	3.2.5 Taxi availability handling (taxi queue)
RideManager	3.2.3 Standard taxi call.
ReservedRidePlugin	3.2.6 Taxi reservation.
SharedRidePlugin	3.2.7 Ride sharing.

Table 5.1: Table that links components to functional requirements.

5.2 Non-functional requirements

The non-functional UI requirements that are common to all client interfaces will be mainly followed during the implementation part. Some of them, though, are shown in Chapter 4 (UI Design):

- The first screen must ask the user to login.
- The dashboard with links to every function shall be displayed in the home page.
- The top bar must show the last taxi service called.
- The possibility to choose the language at all times.

The other functional (programmatic interface) and non-functional **requirements in RASD** fulfilled by the Design Document are shown in Table 5.2.

5.3 Modifications to the RASD

The RASD incorrectly stated that the back-end would be written using the Java SE 8 framework; it has been changed into Java EE 7 as can be seen in sections 2.3, 2.4.

The following requirements have been **removed from the RASD** as they are deemed unnecessary.

- Web application point 2 in 3.1.1 User interfaces: the web pages must be accessible also by text-only browsers.
- Point 4 in 3.4.3 Security: The system must log all login attempts with IP addresses for at least 7 days.
- Point 5 in 3.4.3 Security: The modules must state clearly which data they will need to read and write.

Section (DD)	**Requirements (RASD)**
Component view: Database (2.3.1)	3.1.3 Software interfaces: MySQL
Web server (2.3.3)	3.1.3 Software interfaces 3.4.4 Maintainability: MVC pattern 3.4.5 Portability, see section 5.3
Mobile client (2.3.4)	3.1.2 Hardware interfaces
Selected architectural styles and patterns (2.7)	3.4.4 Maintainability: MVC pattern 3.1.3 Software interfaces: modularity
Application server to front-ends (REST API) (2.6.2)	3.2.9 Programmatic interface 3.1.4 Communications interfaces 3.4.3 Security: HTTPS
Configuration file (application server) (2.6.3)	3.1.1 User interfaces: Server back-end
Web server to browser (2.6.4)	3.1.4 Communications interfaces 3.4.3 Security: HTTPS
Plug-in interface (2.6.5)	3.2.9 Programmatic interface 3.1.3 Software interfaces: modularity
Other design decisions (2.8)	3.4.2 Availability 3.4.3 Security: user's password in DB hashed and salted
Application server: ER diagram (2.4)	3.4.3 Security: user's password in DB hashed and salted

Table 5.2: Table that links sections to functional and non-functional requirements, the functional requirement is written in blue.

Appendix A

Appendix

A.1 Software and tools used

- LaTeX for typesetting this document.
- GitHub[1] for version control and distributed work.
- Draw.io[2] for mock-ups.
- StarUML[3] for UML diagrams.

A.2 Hours of work

The statistics about commits and code contribution are available on GitHub [4]. Please keep in mind that many commits are actually group work (when this is the case, it is stated in the commit message).

- Eleonora Chitti: 17 hours
- Alex Delbono: 19 hours
- Pietro De Nicolao: 19 hours

[1] `https://github.com`
[2] `https://www.draw.io/`
[3] `http://staruml.io/`
[4] `https://github.com/pietrodn/se2-mytaxiservice`

Bibliography

[1] Software Engineering 2 Project, AA 2015/2016 - *Project goal, schedule and rules*

[2] Software Engineering 2 Project, AA 2015/2016 - *Assignments 1 and 2*

[3] IEEE Standard 830-1998: *IEEE Recommended Practice for Software Requirements Specifications*

[4] *Software Engineering 2: myTaxiService – Requirements Analysis and Specification Document* Chitti Eleonora, De Nicolao Pietro, Delbono Alex – Politecnico di Milano

[5] Software Engineering 2 Project, AA 2015/2016 - *Template for the Design Document*

[6] The Google Maps Directions API - `https://developers.google.com/maps/documentation/directions/intro#traffic-model`