**Overview**

1. **High level components and their interaction**

This application will be developed and implemented following the wide spread **client-server** pattern. In the designing phase, we privileged a **four-tiered** architecture (as shown in the figures below).

The reasons of this choice rely on the fact that the application must be as reactive as possible, therefore we instantiated the DBMS and the application-layer on the same tier, to reduce latency.

In fact, we decided to delegate to the server both the handling of the application logic and the managing of the database, since the latter is neither heavy nor complex and it is required to be as fast as possible, allowing an high efficiency.

Furthermore, keeping the database as near as possible to the application-server leads to a strong security level, since the amount of data crossing the network decreases.

As far as the distribution of the logic is concerned, we observed that it would be useful and smart to leave a little part of the logic client-side, to lower the load of the server, reduce useless interactions and minimizing the delay.

Due to the notification functionality offered by the system (related to unpredictable delays, weather changes, strikes…), a **publish-subscribe** approach will be adopted.



//img architettura

The registration messages are exchanged in an asynchronous way, since the client sends to the server the form filled with all the data of the user, and the server replies with a confirmation email to the address indicated in the form (obviously asynchronously).

Then the other messages between the client and the server are exchanged synchronously, since the client waits an acknowledge for each message sent to the server.

All the messages are made private through cryptography, since they contains personal information of the user.

The client-server interactions are handled through REST paradigm, to make it as flexible as possible, whereas external services interactions are carried out through proper APIs.

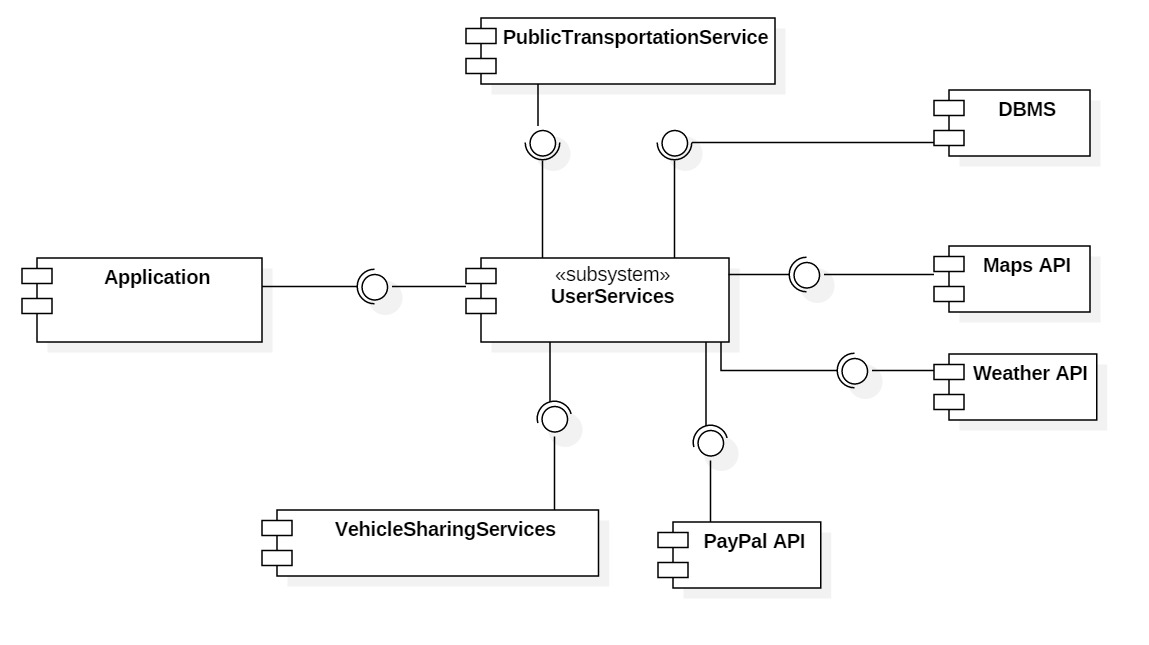
**b. Component View**

**High-level component view**

This diagram aims to explain how the different components of the system interact.

On the left we find the client-side (Application), intended both as installed application and as web accessible.

On the right we find the server-side (UserServices), which interacts with our DBMS and all the external services necessary to accomplish the tasks.



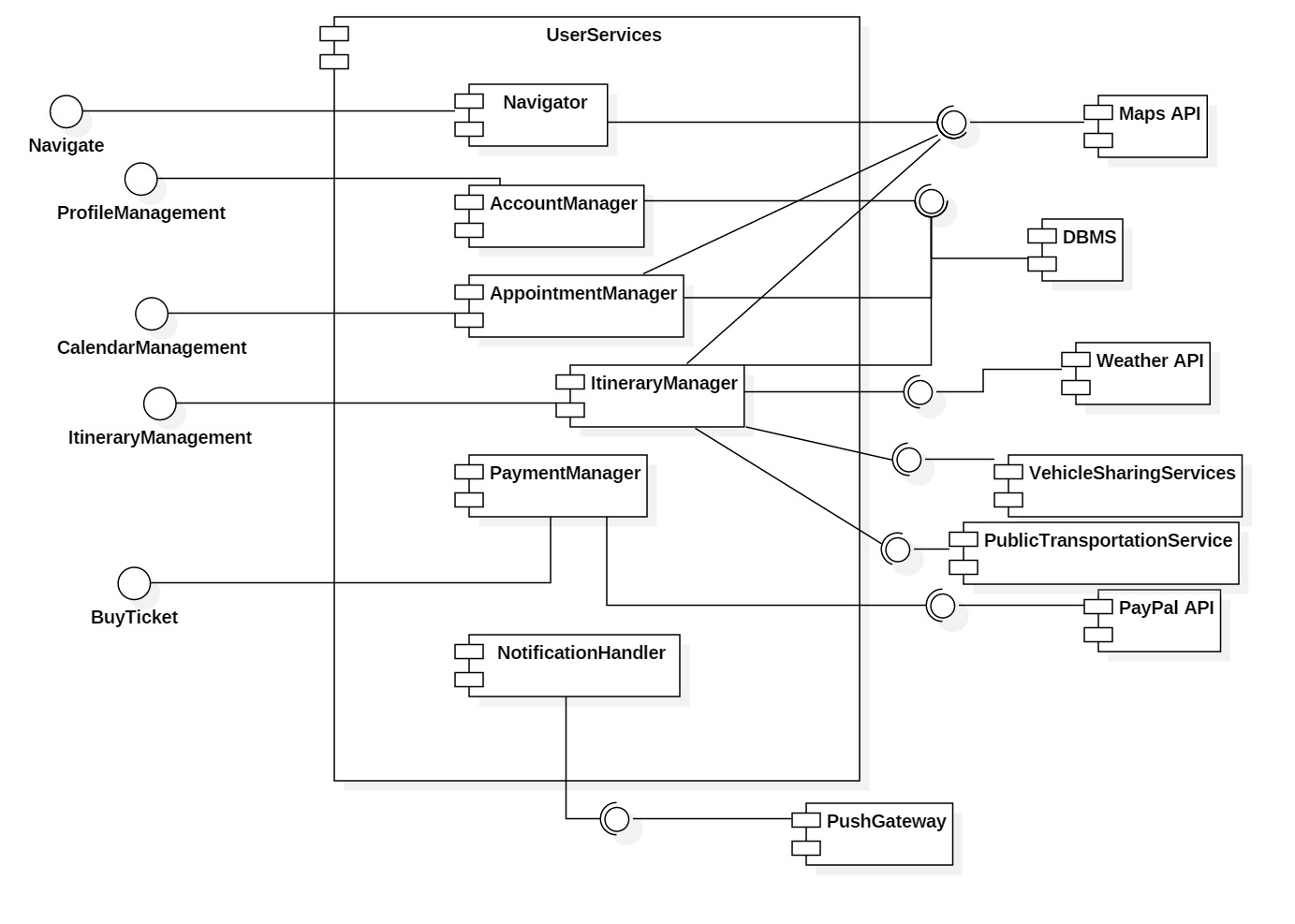
**UserServices component view**

This diagram explodes the UserServices subsystem, to clarify the various modules and the exposed interfaces.

This component is composed by 6 main modules:

* Navigator: it guides the user toward its destination, following the indications already computed.
* AccountManager: it handles all the user’s preferences.
* AppointmentManager: it manages all the schedules of the users, checking the general consistency.
* ItineraryManager: it organizes all the trips among appointments, computing paths using both information provided by external services and users’ preferences.
* PaymentManager: it manages the payments to buy public transportation tickets.
* NotificationHandler (**Event Dispatcher**): its aim is to warn user of incoming appointment and of itinerary variations, through push notifications.

It is the main component of our publish-subscribe pattern.



**Entity-Relation Diagram**

The following is the model to follow in the implementation of the database.

As mentioned above, we aim to a system as flexible and light as possible, therefore we adopted a skinny approach for our database, in order to let the interactions be fast:

for instance, entities like DailySchedule (see UML) are not stored, but they are computed by the system by querying about all the appointments on the same date.

This approach also prevents from update anomalies, since redundancy is avoided.



**c. Deployment view**

**e. Component interfaces**

The UML diagram shows the high level interface exposed from the server and used by the client in the client-server paradigm:

-*ProfileManagement* is used by the application to register a new user, to check the credential in the logging phase and to manage the user preference.

-*Calendar* and *ItineraryManagement* expose the methods to fill and edit the calendar with the appointments and the related itineraries.

-*Navigate* is necessary to have access to the commands of the navigator during the trips.

*-BuyTicket* behaves as a broker between the client which wants to buy a public transportation ticket and the PayPal service that is in charge of manage the trade.



**Further interfaces**

**Application Server & External Systems**

The application server is expected to connect with other external systems:

* PayPal: it provides API to which the server itself must adapt in order to perform payments.
* Weather Forecast, Public Transportation Services, Maps Provider and Vehicle Sharing Services: these services are supposed to expose some interfaces in order to be queried. The information collected will be used to arrange itinerary and appointments.

**Application Server & Web Server**

The communication between Application Server and clients, both direct and via the Web Server, will be performed via RESTful APIs, provided by the Application Server itself and implemented using JAX-RS.

**Database & Application Server**

Access to the database is carried out through the Java Persistence API, mapping between objects and actual relations.

**Web Server - Web Browsers**

As shown above, this interaction is completely based on HTTP protocols.