Power EnJoy

An electric car sharing system

Presented By:

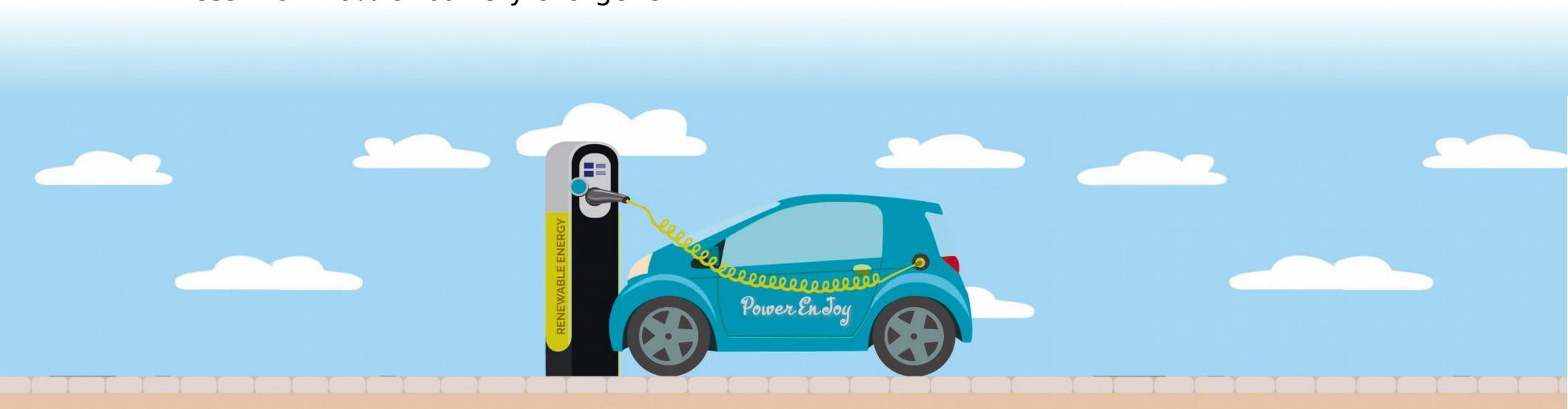
Simone Mosciatti
Sara Zanzottera

Overview

PowerEnJoy is a car sharing platform focused exclusively on electric cars. It aims to improve the mobility inside the city in a eco-friendly way. To achieve its goal, it encourages people to be environment aware drivers.

The system provides all the functionalities normally provided by a car sharing service. In addition, the system gives bonuses and penalties in term of discounts or over-prices depending on the behavior of the user, like:

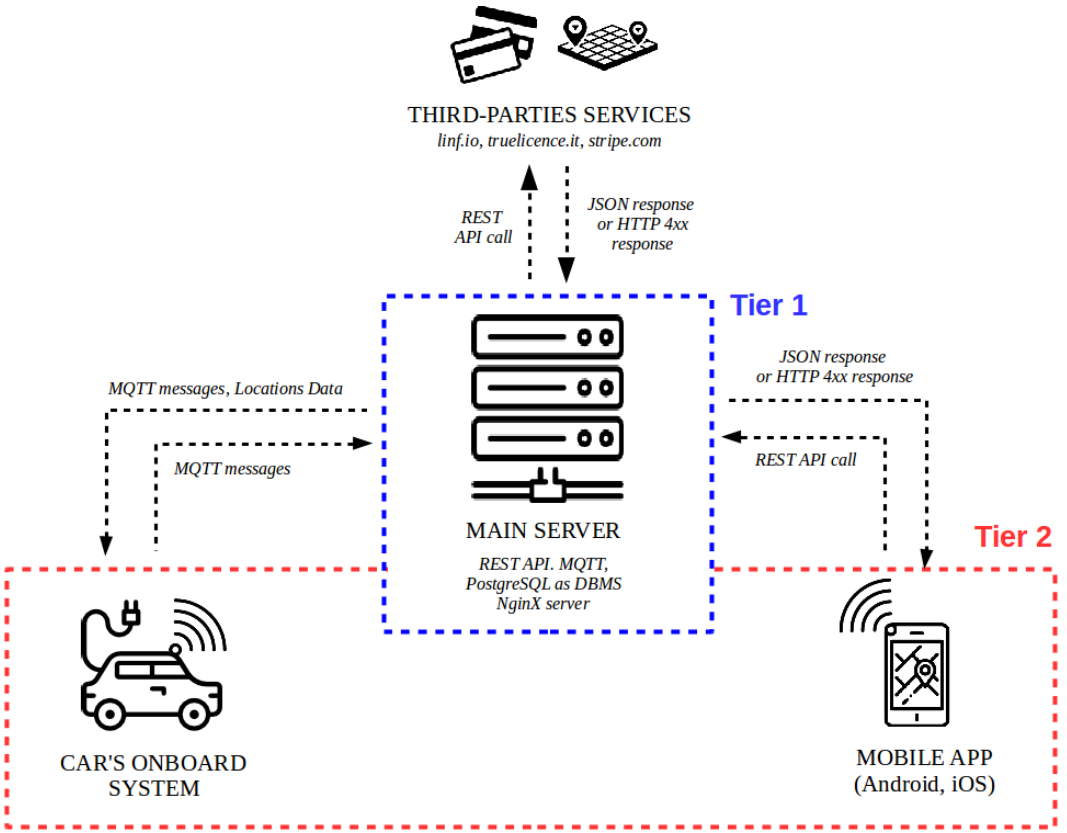
- 10% discount for users who brings at least two other passengers with him during the ride.
- 20% discount for users who leaves the car with at least 50% of the battery still full.
- 30% discount if the car is plugged to a charging station at the end of the ride.
- 30% overprice if the car is left more than 3km from the nearest charging station or with less than 20% of battery charge left.



Overview

PowerEnJoy is an inherently distributed system, based on a central server interactions with many distributed nodes. In detail the system can be divided into four main parts:

- a **public app**, used by customers to access the service
- a **centralized backend** that provides the service
- the **cars' onboard system**, that communicates only with the centralized backend
- a **reserved fronted**, used exclusively by the staff members to better organize their job



Part One

REQUIREMENTS AND SPECIFICATIONS

Early Design Choices

The client server architecture
for the kind of application we are developing, a client server architecture is one of the most basic choice we could made.

App
A mobile app to provide the service to customers is a reasonable choice.
Using the term “App” we do not mean an application for a specific mobile vendor, but a broader interface to the PowerEnJoy services.

Fines
Fines management seems to us not directly related to the service we are providing: we decided to not include this functionality into the system.
Considering that local police officers, when issuing a fine on a PowerEnJoy car, are likely to send the fine to the company instead of the user who drove the car in that moment, the system is responsible only for allowing the company to find out which user was driving which car in a certain moment. Fines are then managed by the company the way they prefer.

Text Assumptions

Assumptions on the final user

1. The final user has reached the legal age and has a driving licence.
2. The final user has a smartphone with the PowerEnJoy app installed, a GPS and Internet connectivity.

Assumptions on the car system

3. Each car is provided with internet connectivity and capable to send and receive data to the main server.
4. Each car is provided of a GPS of reasonable accuracy.
5. Each car can capture events related to the engine, passengers and lock correctly
6. Each car is able to monitor the residual charge of its battery.
7. Each car is provided with a device that can scan and read a driving licence.
8. The system is able to prevent the ignition of each car's engine remotely.
9. The system is able to switch off every car's engine remotely.
10. The system is able to unlock each car remotely.
11. The system is able to lock each car remotely.
12. The total running time of an ride is counted from either when the user switches the engine on or five minutes after the car is unlocked.
13. The total running time of an ride ends when the user locks the car.

Statuses Assumptions

The cars are assumed to have some internal statuses describing their conditions. These are:

- **Availability Status:** describe if the car is interacting with some users of if it is available for rent. (Available, Booked, Unlocked, Running, Parked, Not Available)
- **Charging Status:** describe if the car is connected to the power grid or not. Boolean values (True or False)
- **Exception Status:** describe if the car has some issues. (No Issue, Out Of Power, Unsafely Parked, etc…)

We defined a list of text assumptions based on the definition of these statuses that eventually model the expected behavior of the system with respect of cars, and the behavior of cars themselves.

Domain Properties

- **Each user** has only one **unique Identity ID** related to them.
- Each user has only one **unique driving licence** related to them.
- A driving license will **not expire** nor be revoked for the whole time the user is registered.
- Each account uniquely identify a physical person: **account cannot be shared** or used by different people, even in different moments.
- It is possible for the staff to bring back cars from not-safe areas into safe areas.
- A car cannot change its position while the engine is switched off.
- A car's engine cannot be switched on while the car is locked (the contrary may happen).
- A car's residual charge does not increase if it is not plugged to the power grid.
- A car's residual charge does not decrease if it is not running, i.e. the engine is off.
- Users are able to plug and unplug the car from the charging stations.
- It is **possible** to determine whether the **car is parked in a safe area or not**, at any moment.
- It is possible to determinate the actual level of charge of the battery with negligible error.
- It is possible to determinate how **many people are in each car** at any moment.
- Each and any message that the car sends to the main server arrives and is processed.
- It is possible to **read a driving license** informations.
- Once the car is plugged to the charging station, the battery starts to increase its charge.
- Once the car's engine is switched on, the battery starts to decrease its charge.

External Services

The system depends on external services that are served by third-parties. These services
are used to:

- Charging fees to the users.
- Determinate if the car is been parked in a safe area.
- Determinate the distance from the closest charging station.
- Determinate the distance between two different GPS signals.
- Scan and read driving licences' informations.
- Ensure validity of Identity IDs and if they match with personal informations provided by the user and the provided driving license data.
- Regularly check and maintain the cars.

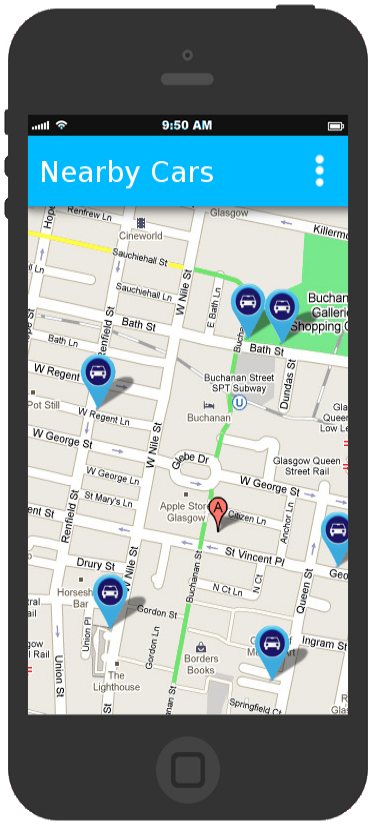
The decision to rely on external partners to run these functions has been made since
those services are not the core business for PowerEnJoy, and other external companies
has already found very efficient and cost-effective solution to them.

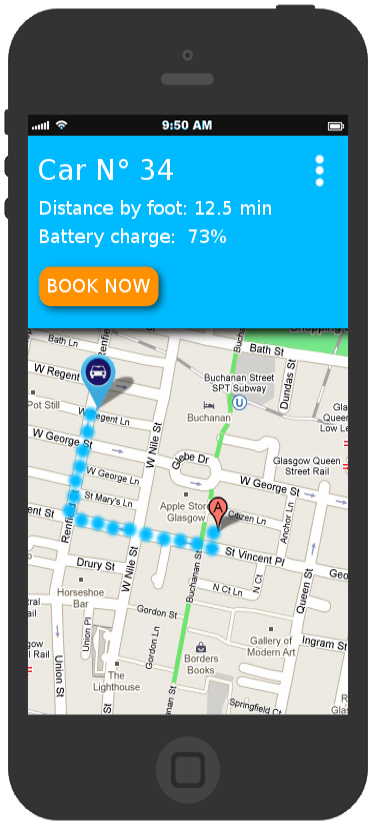
Goals of the System

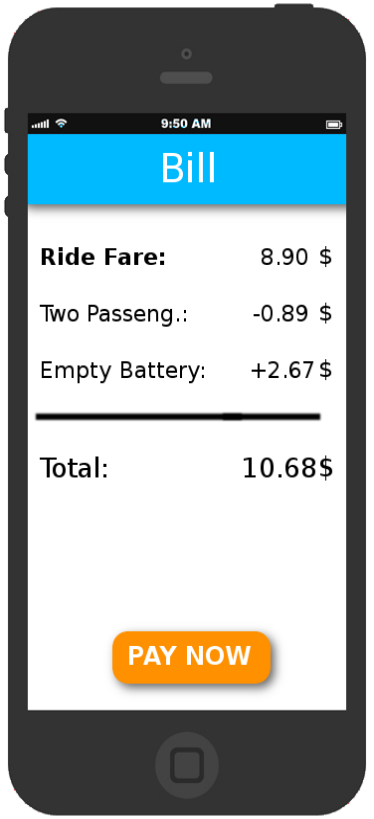
1. **REGISTRATION** Users can register to PowerEnJoy .
2. **LOGIN** Users can login to PowerEnJoy .
3. **LOOKUP** Users can find cars nearby a given position, according to their search settings.
4. **BOOK** Users can book a car for a short amount of time.
5. **UNBOOK** Users can decide to cancel a booking made before the expiration.
6. **UNLOCK** Users can unlock the car they booked once they are near it.
7. **RIDE** Users can drive to their destination.
8. **SAFE AREAS** Users can locate safe parking areas.
9. **UNSAFE PARKING** The system must react to an unsafe parking.
10. **POWER STATIONS** Users can locate charging stations.
11. **CHARGE** At the end of the ride, users are charged a fee.
12. **PAYMENTS** Users can pay bills through the app.
13. **REPORT ISSUES** Users can report issues to the system.
14. **FIND ISSUES** The staff can locate cars that need their intervention.
15. **SUPPORT** The staff can identify and solve car’s issues.
16. **FINES** The system can provide enough details for the staff to manage correctly the fines.

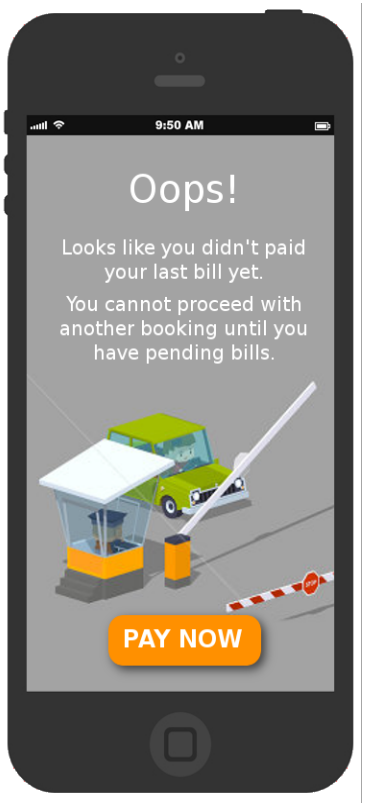
Interface Mockups

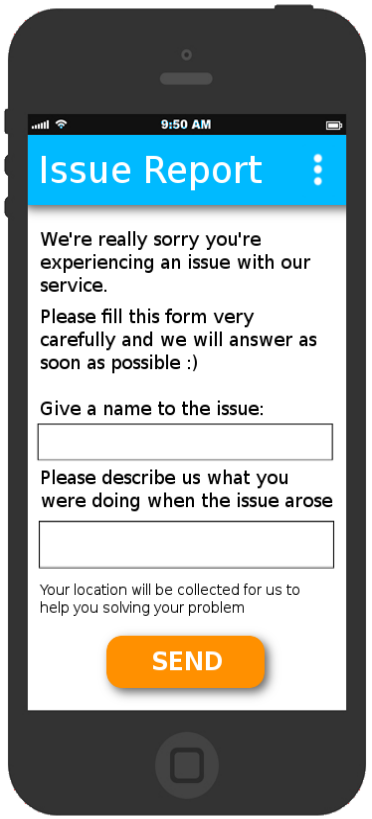
Public App





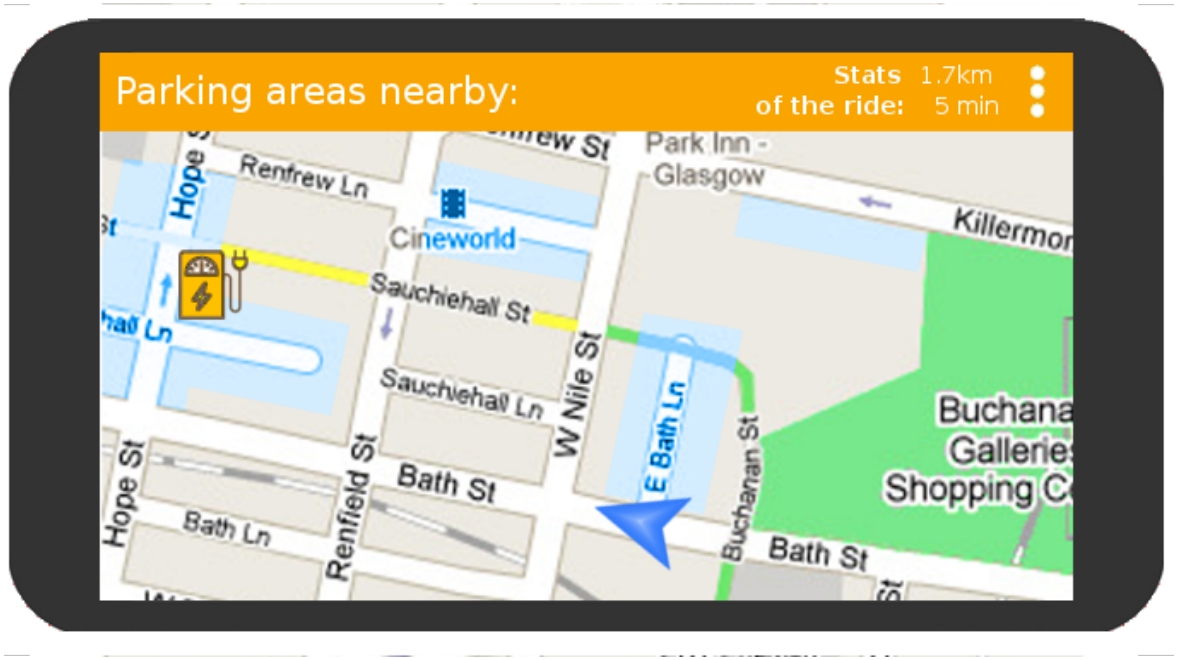






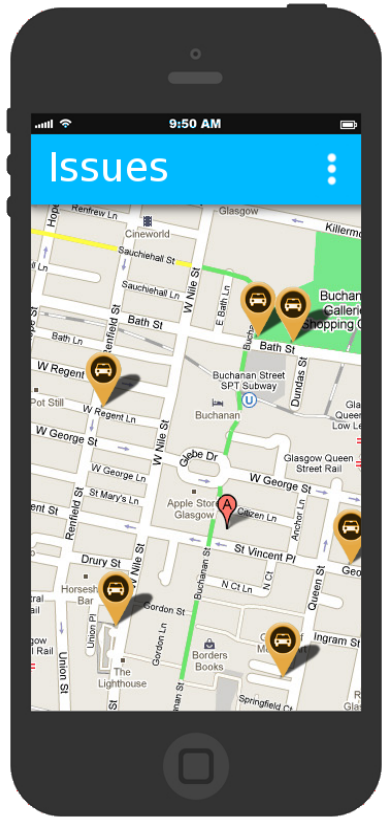
Interface Mockups

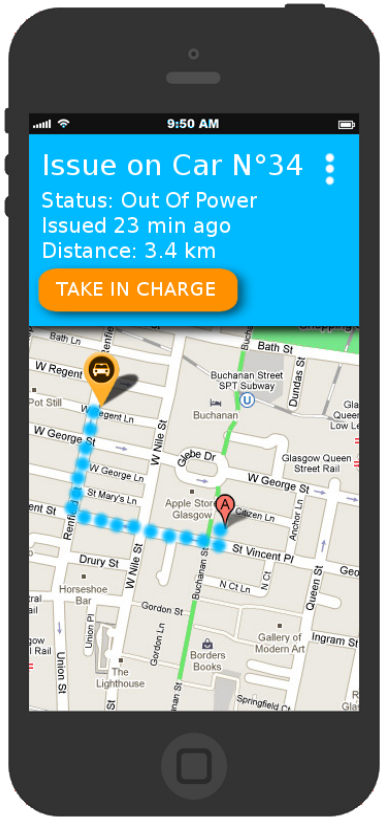
Car’s Onboard System

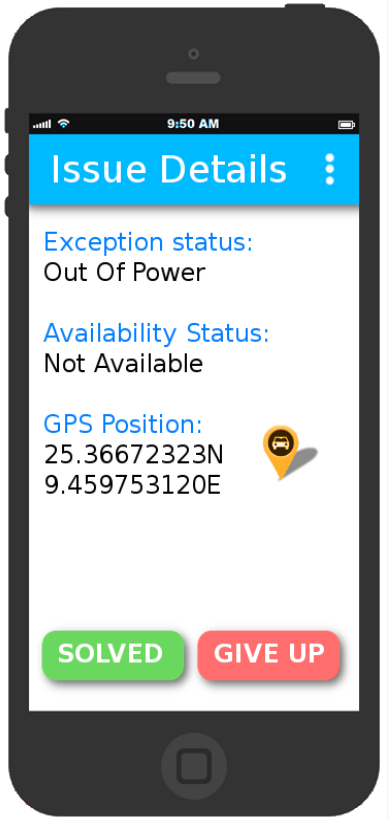


Interface Mockups

Staff’s Reserved App

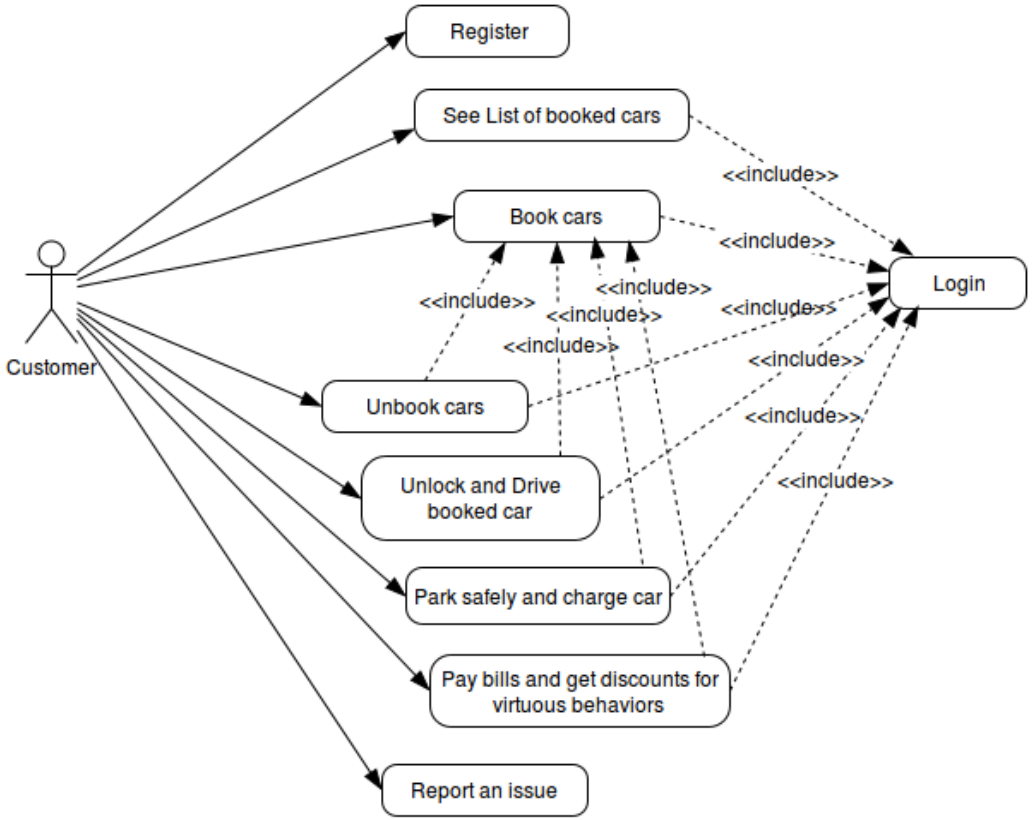




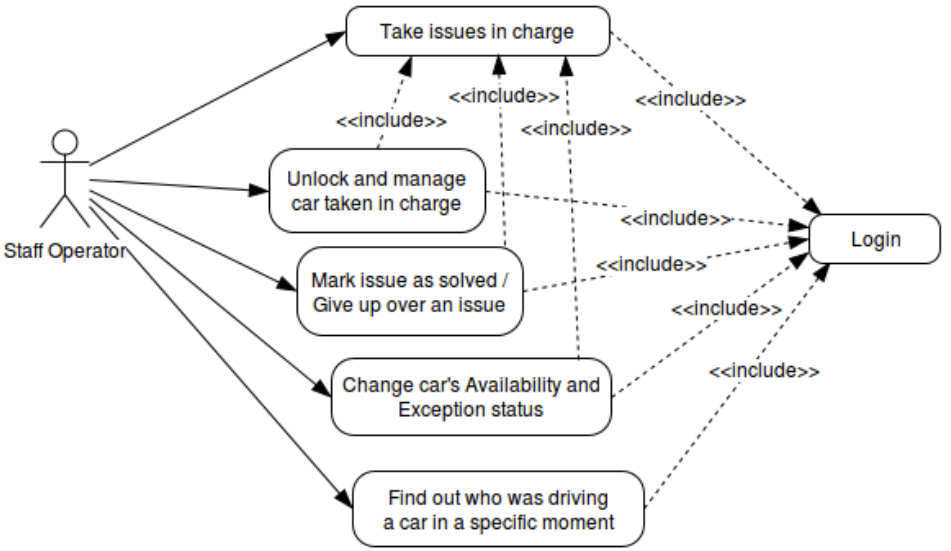


Use Cases

Customer Use Cases



Staff Operator Use Cases

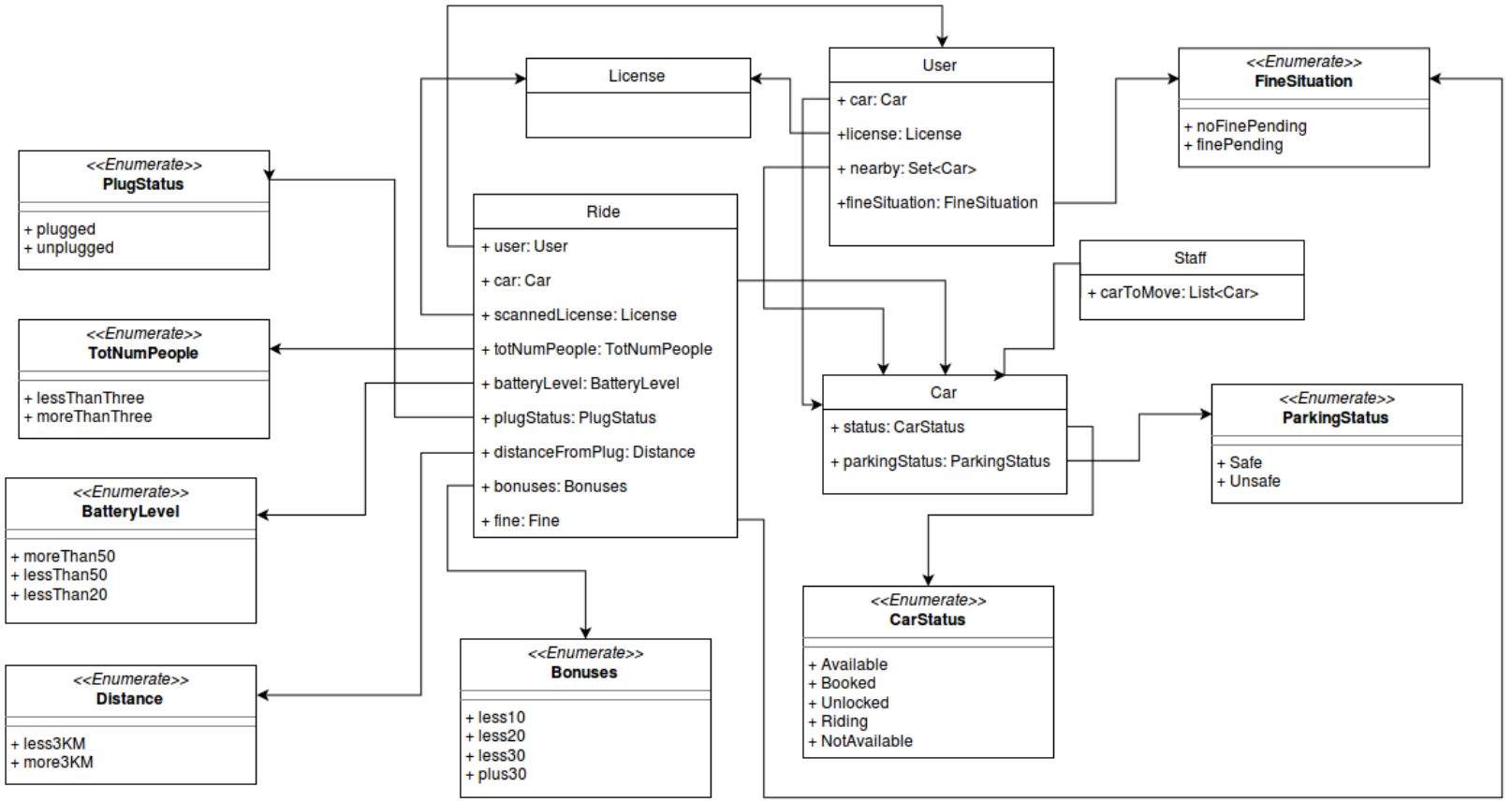


Alloy Models

We developed **two** separate Alloy models:

- The main one provided a static description of the system. First developed and tested in chucks, then merged into one single model.

- The second one, smaller, provided a dynamic description of the system.



Part Two

SYSTEM DESIGN

Design Process Description

The overall design process has been carried in a **bottom-up** approach, starting from the analysis of goals and requirements moving upwards to the definition of the higher level components of the system.
The process followed these fundamental steps:

1. Analysis of **goals**.
2. Analysis of the **interface**: given the list of goals and requirements, we identify what interfaces the system need to provide to the users.
3. Analysis of **functionalities** required for the interfaces.
4. Functionalities are organized into **high-level components** taking care to respect the “Single Responsibility” principle, in order to provide highly decoupled and reusable components.
5. Identification of the **physical nodes** where our components should be deployed in order to fulfill their functionalities.
6. Analysis of the **communication protocol** among different nodes.
7. **Deployment** of components into the physical nodes.

To double check the correctness of the overall design we produce several sequence diagrams, showing for the uses cases what functionalities are involved and what functionalities are called.

Goals Analysis (1/2)

Goals from the RASD are named **SB/Name** in order to highlight that these functions are related to the system's boundaries.

We know already that some goals are specific for customers, some are specific for the staff operators, and some are shared:

- **SB/ALL/FunctionalityName**: shared functionalities
- **SB/CUST/FunctionalityName**: customer's reserved functionalities
- **SB/STAFF/FunctionalityName**: staff's reserved functionalities

Specific functionalities provided by the backend are named **Sy/FunctionalityName**, where "Sy" indicate an abstract system.

SB/CUST/Ride

Description Users can ride a reserved car.

Requires

- Sy/StartRide
- Sy/EndRide
- Sy/ValidateLicense
- Sy/GeoLocationAreas
- Sy/Lock

SB/STAFF/Support

Description The staff can identify and solve car's issues.

Requires

- Sy/Lock
- Sy/Unlock
- Sy/TakeChargeIssue
- Sy/SolveIssue
- Sy/GiveUpIssue

SB/ALL/PowerStation

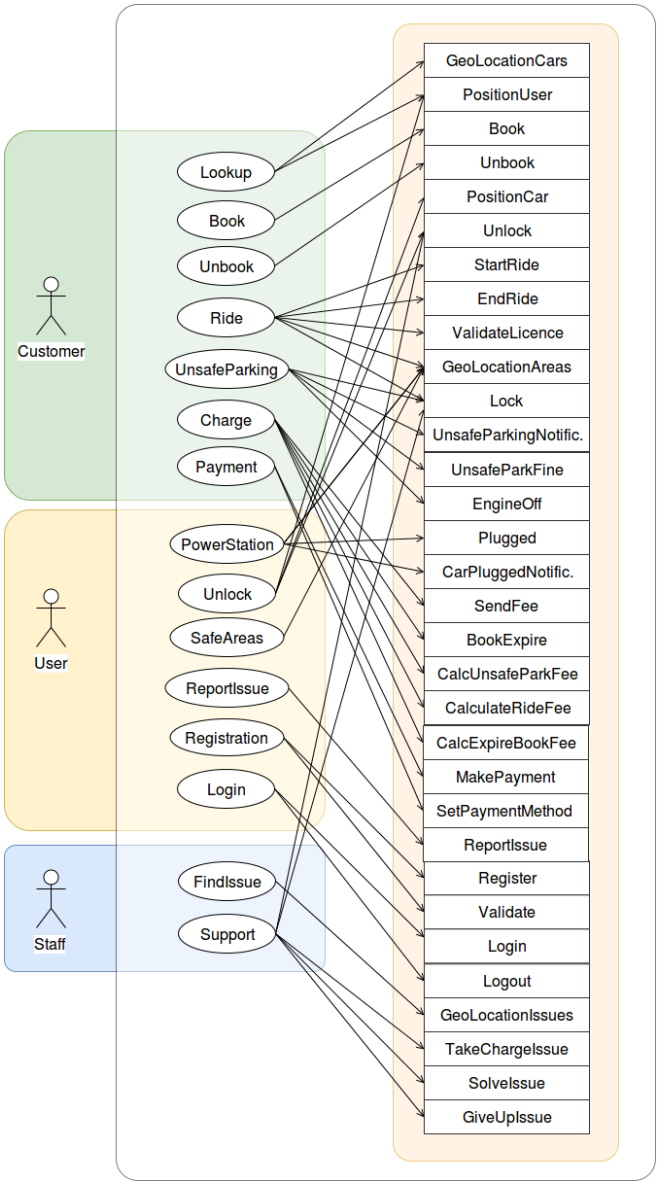
Description Users can locate and use charging station.

Requires

- Sy/GeoLocationAreas
- Sy/Plugged
- Sy/CarPluggedNotification

Goals Analysis (2/2)

Graphical visualization of goals and system's required functions identified above,
divided by type of user.



Interface Design

Now we organize all functions into **higher-level interfaces**, being careful at respecting the responsibilities given to each one of them.

In order to clarify how previously defined “Sy/FunctionalityName“ maps with the interface’s specific functions, we used a notation **INTERFACE/FunctionalityName ⇔ Sy/FunctionalityName.**

BOOKING_MANAGER

Responsability Manages reservations.

BOOKING/Book ⇔ Sy/Book

Responsability Books one available car.

Input The ID of the car and the ID of the user.

Output The car is booked and the ID of the reservation is provided.

BOOKING/Unbook ⇔ Sy/Unbook

Responsability Removes a reservation.

Input The ID of the user and the ID of the reservation.

Output The reservation is cancelled.

BOOKING/Expire ⇔ Sy/BookExpire

Responsability Removes an expired reservation and fines the related user.

Input ID of the reservation.

Output The reservation is cancelled and the user is fined.

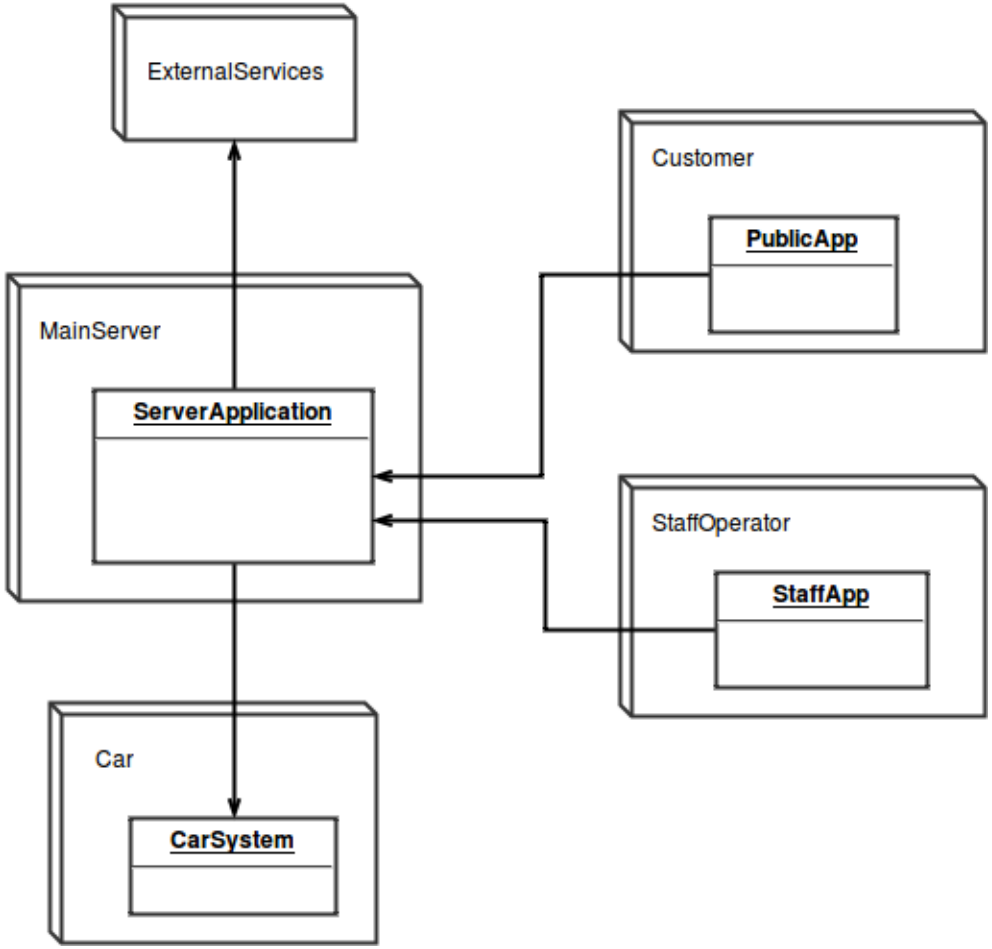
Physical Structure

As for the RASD, the system is divided into four elements:

- the **customer's app**, used by customers to access the service.
- the **staff's app**, used exclusively by the staff members to better organize their job.
- the **main server**, a centralized backend that provides the service.
- the **cars' onboard system**, that communicates only with the centralized backend.

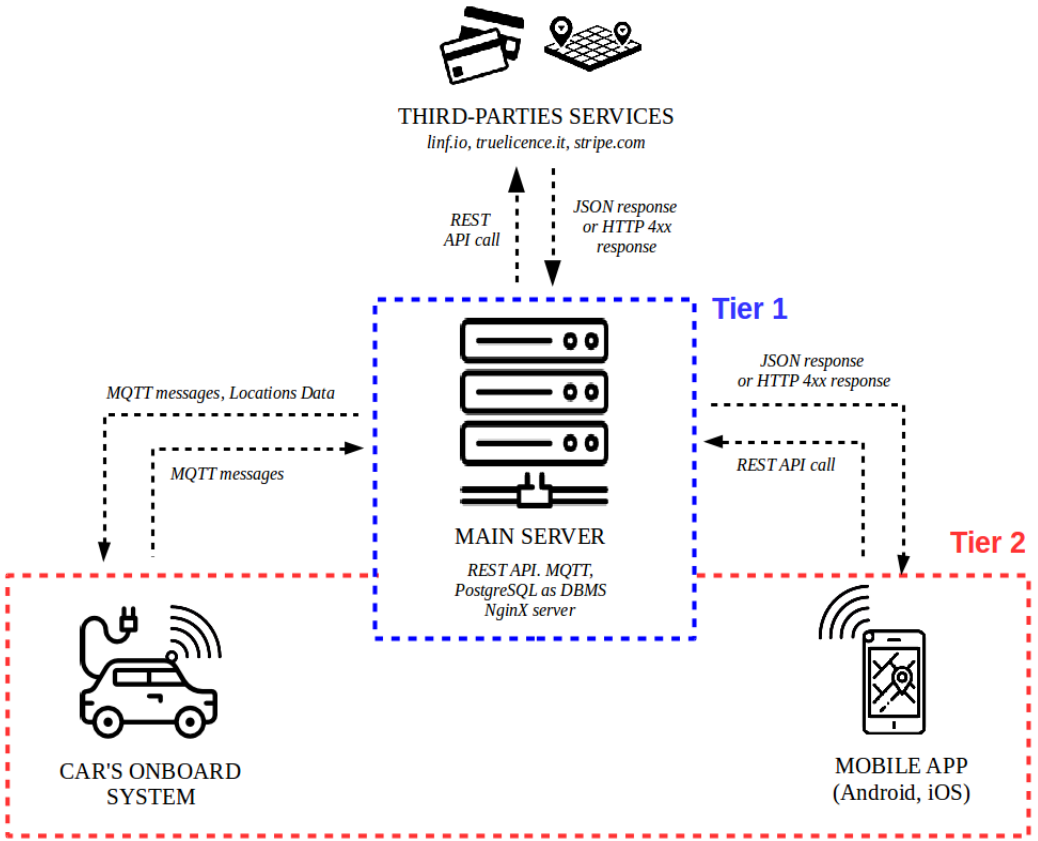
The above elements can be organized in a **two-tier Client-Server** architecture as follows:

- Tier 1, the main server, which handles Application Logic and Data Management.
- Tier 2, comprising mobile apps and cars, hosts the User Interface.



Communication Design

We designed a system based on only two communication channels: **Main Server ⇔ Apps** and **Main Server ⇔ Cars**.
We completely avoided all kinds of communication between the cars and the apps: the main server always acts as an intermediary.



The server acts as an intermediary also for communications from Tier 2 to 3rd parties.

Three different approaches for three channels:

- Main Server ⇔ Apps: simple **Client-Server** approach (well known standard)
- Main Server ⇔ Cars: a **Publisher-Subscriber** pattern (high throughput and low latency)
- Main Server ⇔ External Services: **Service-Oriented** approach (external services are Web Services)

Components Design (1/4)

Up to this point we defined the logical components of the system in terms of interfaces and communication protocols.

We now deploy the system on its physical nodes, in order to end up with a complete, high-level logical architecture for our system.

BOOKING_MANAGER

Considering that all its functionalities are exposed as API by the server, the component is deployed on the server entirely.

CAR_MANAGER

This component requires a deeper analysis, as its interfaces are exposed by different elements. In details:

- CAR/Unlock is exposed by Server to Apps
- CAR/GetDetails is exposed by Cars to Server
- CAR/ValidateLicense is exposed by Server to Cars
- CAR/Lock is exposed by Cars to Server
- CAR/TurnOff is exposed by Cars to Server
- CAR/Telemetry is exposed by Cars to Server
- CAR/SetStatus is exposed by Cars to Server

Thus we create two CAR_MANAGER subcomponents:

REMOTE_CAR_MANAGER

Comprises all the functions exposed by the server:

- CAR/Unlock
- CAR/GetDetails
- CAR/ValidateLicense

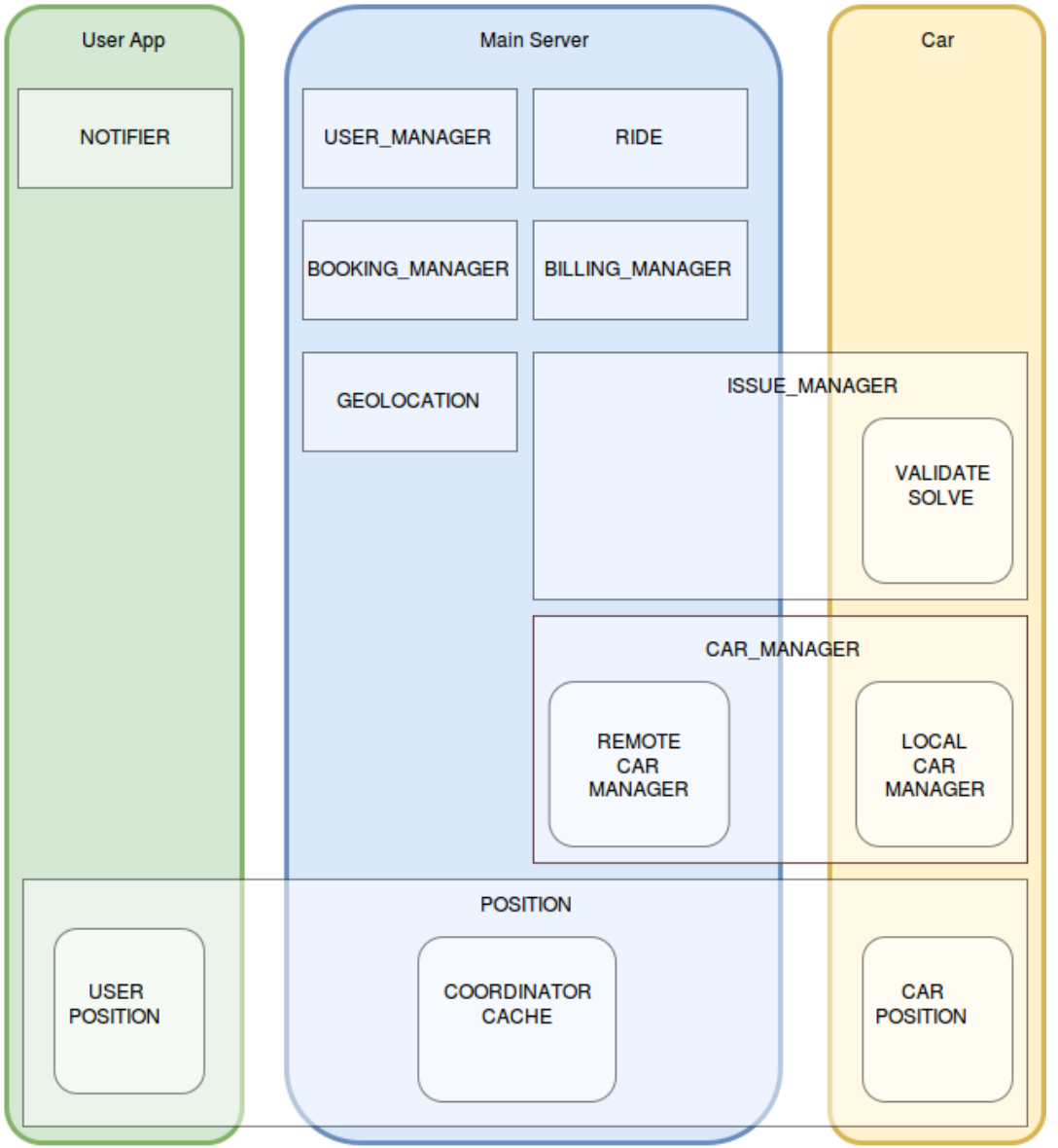
LOCAL_CAR_MANAGER

Comprises all the functions exposed by the cars:

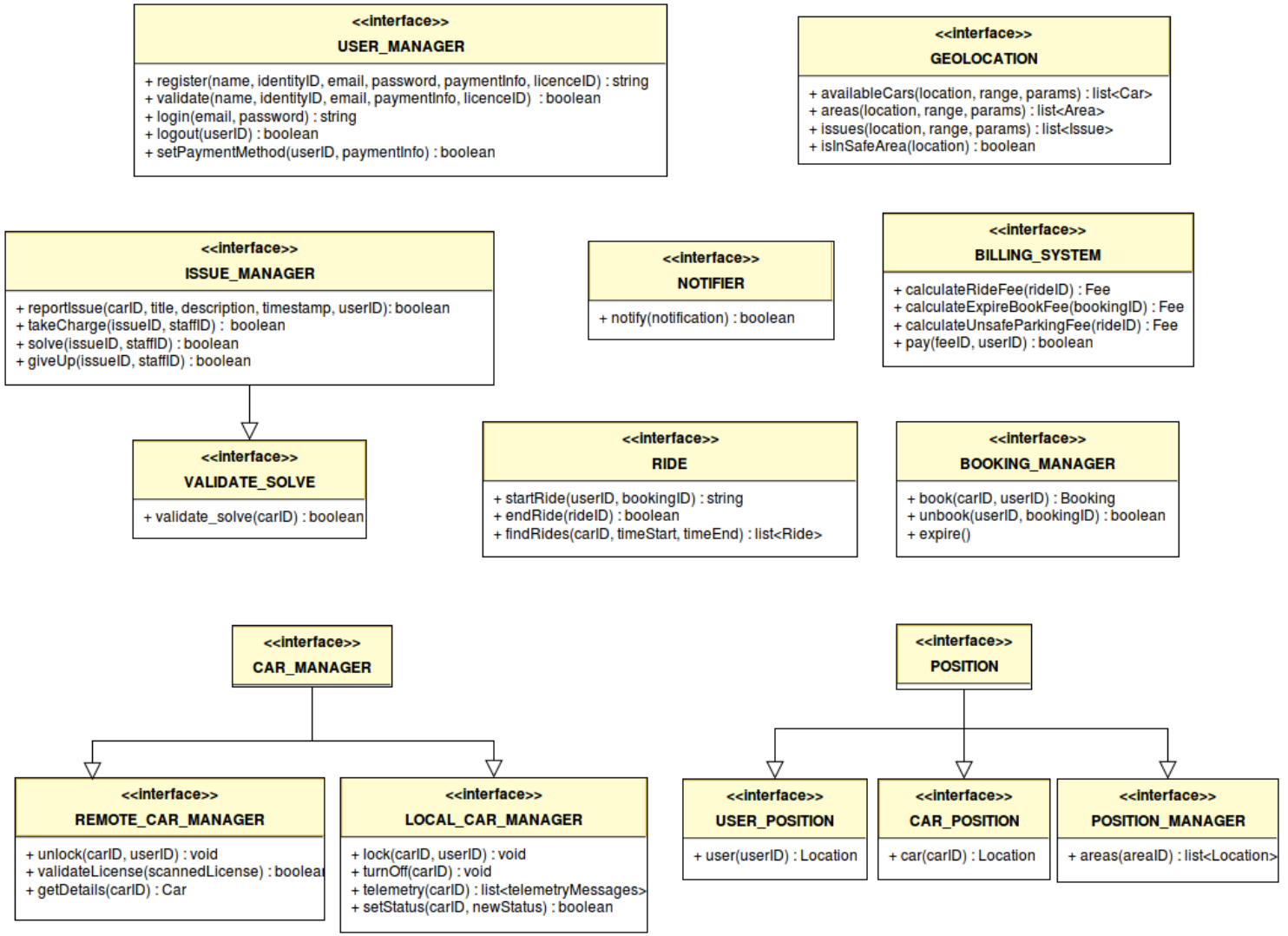
- CAR/Lock
- CAR/TurnOff
- CAR/Telemetry
- CAR/SetStatus

Components Design (2/4)

Graphical visualization of the abstract components identified.

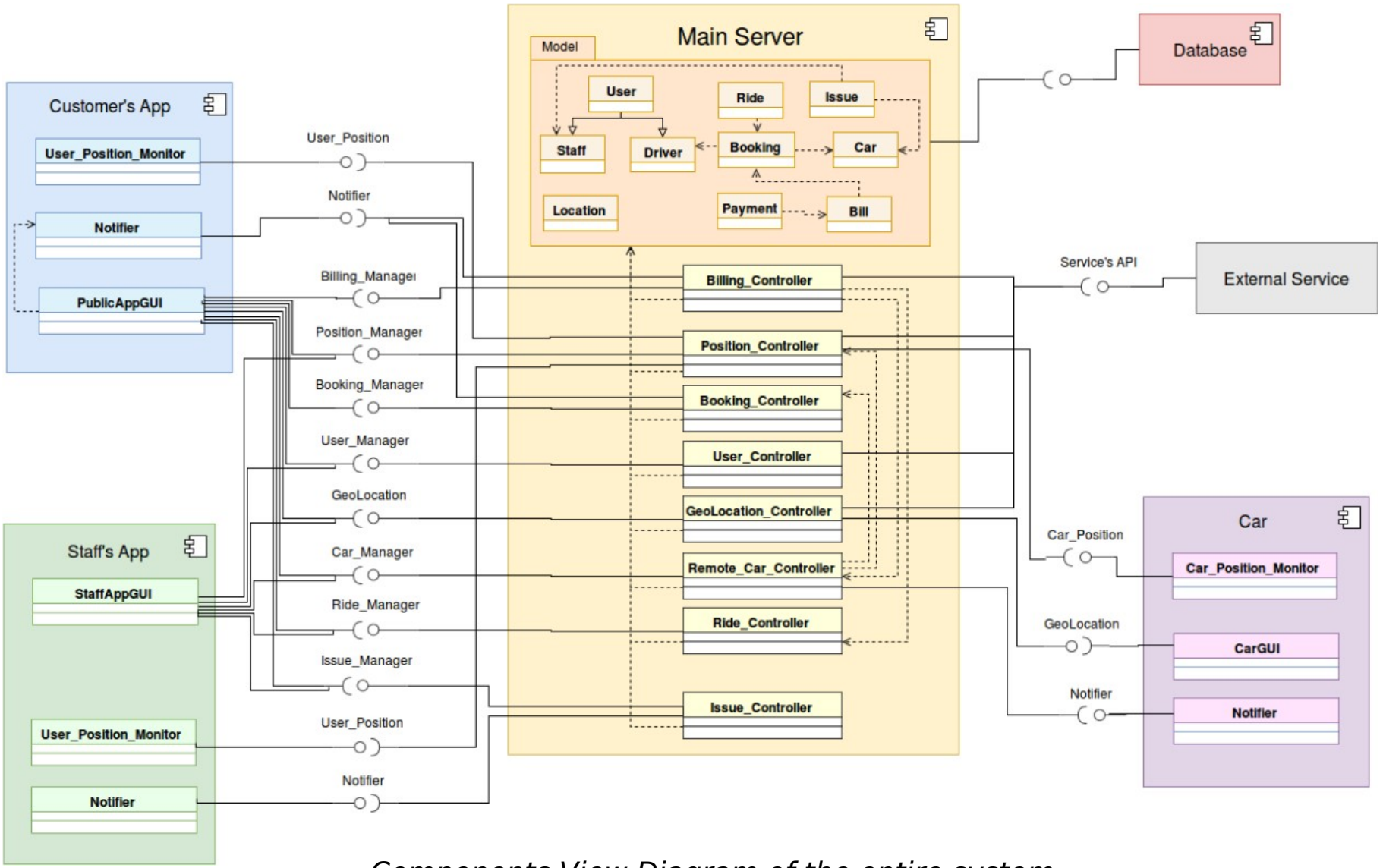


Components Design (3/4)



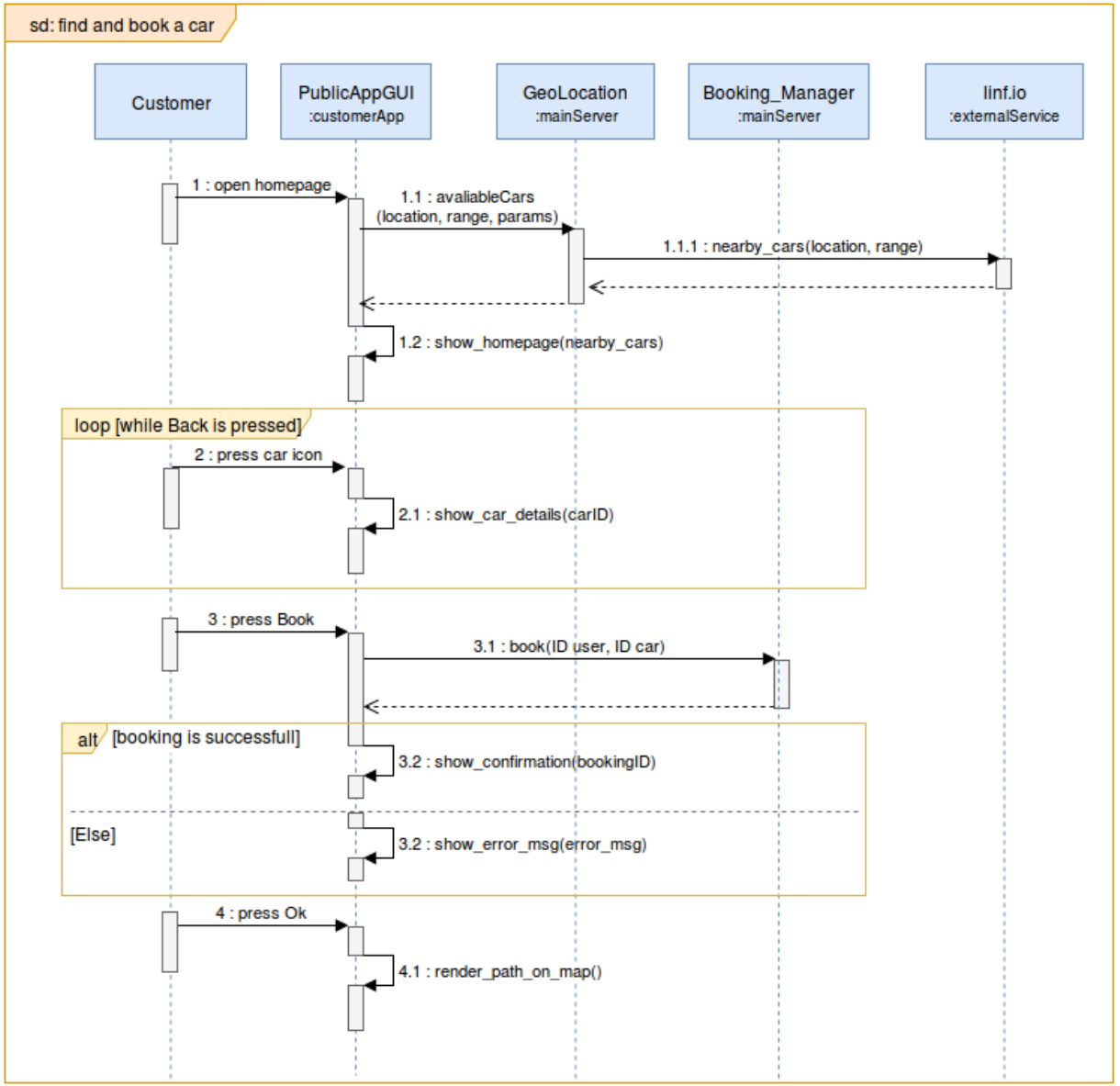
Description of the interfaces of the components identified above.

Components Design (3/4)



Components View Diagram of the entire system.

Sequence Diagrams



The high level of **decoupling** becomes evident here.

The first goal is achieved with the sole need of GEOLOCATION, deployed server-side, that hides to the final user the interaction with linf.io, which is an external service.

The app is only responsible of rendering the informations into an interactive map.

Once on the map, users can navigate cars details until they decide to book a car, when a call to BOOKING MANAGER is issued. The module handles the full procedure internally.

The app is responsible only to provide feedback to the user.

Selected Technologies

RESTful API
The main server exposes its API in the most conventional way, using the classical HTTP/TCP/IP stack. In particular we aim to provide RESTful interfaces.

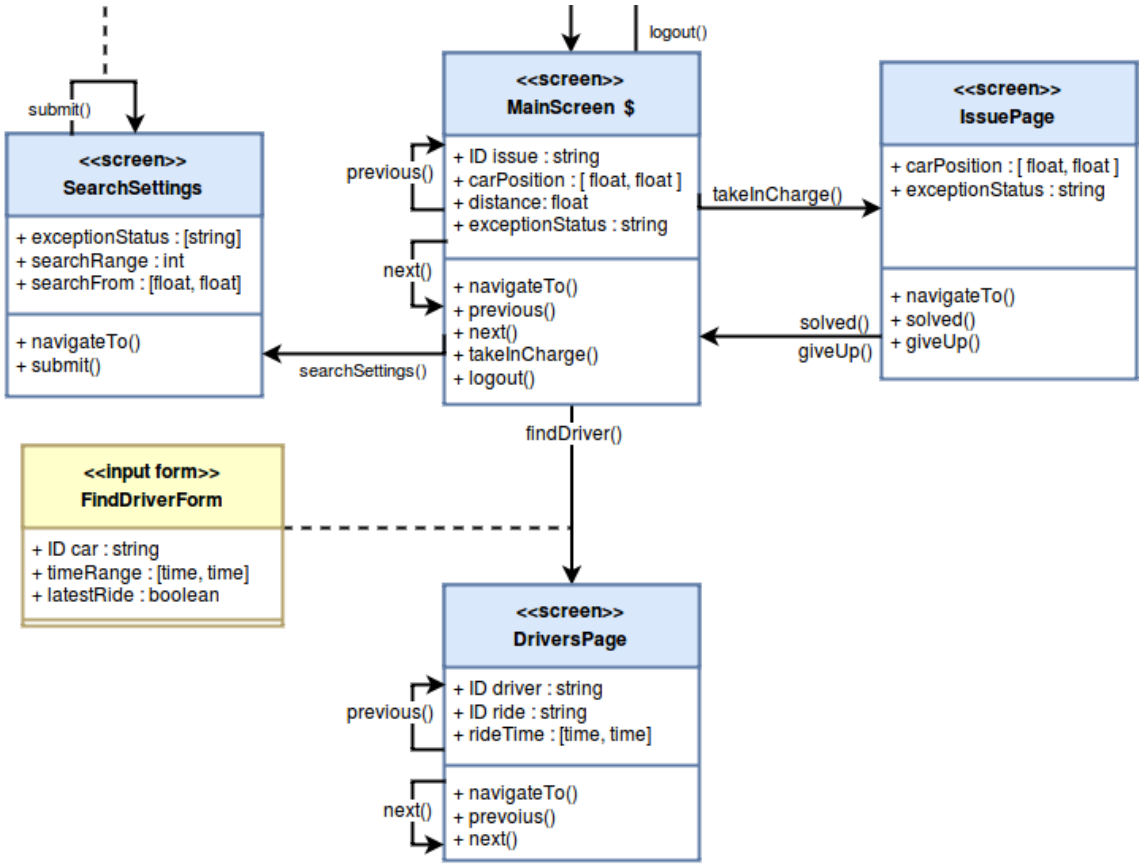
MQTT
Among all the implementation of the PubSub protocol we chose to use MQTT for its low overhead, its QoS and because it is widely used in the industry.

Nginx
This choice has been made with care, because Apache is surely a more widely used standard for servers, whereas Nginx is a younger, less used solution. Indeed, considering our server is going to handle a lot of parallel operations, we chose a software designed exactly to tackle this issue better than standard Apache-based solutions.

PostgreSQL
As DBMS for our system we chose PostgreSQL because it is an enterprise-level, open-source software, widely known in the industry for its reliability, integrity and correctness. it supports all major operating systems, it is very standard compliant and at the same time highly customizable.
In addition, PostgreSQL features some spatial database extenders, like **PostGIS**, that adds geographical objects support and makes easier running location queries on the database.

User Interface Design (1/2)

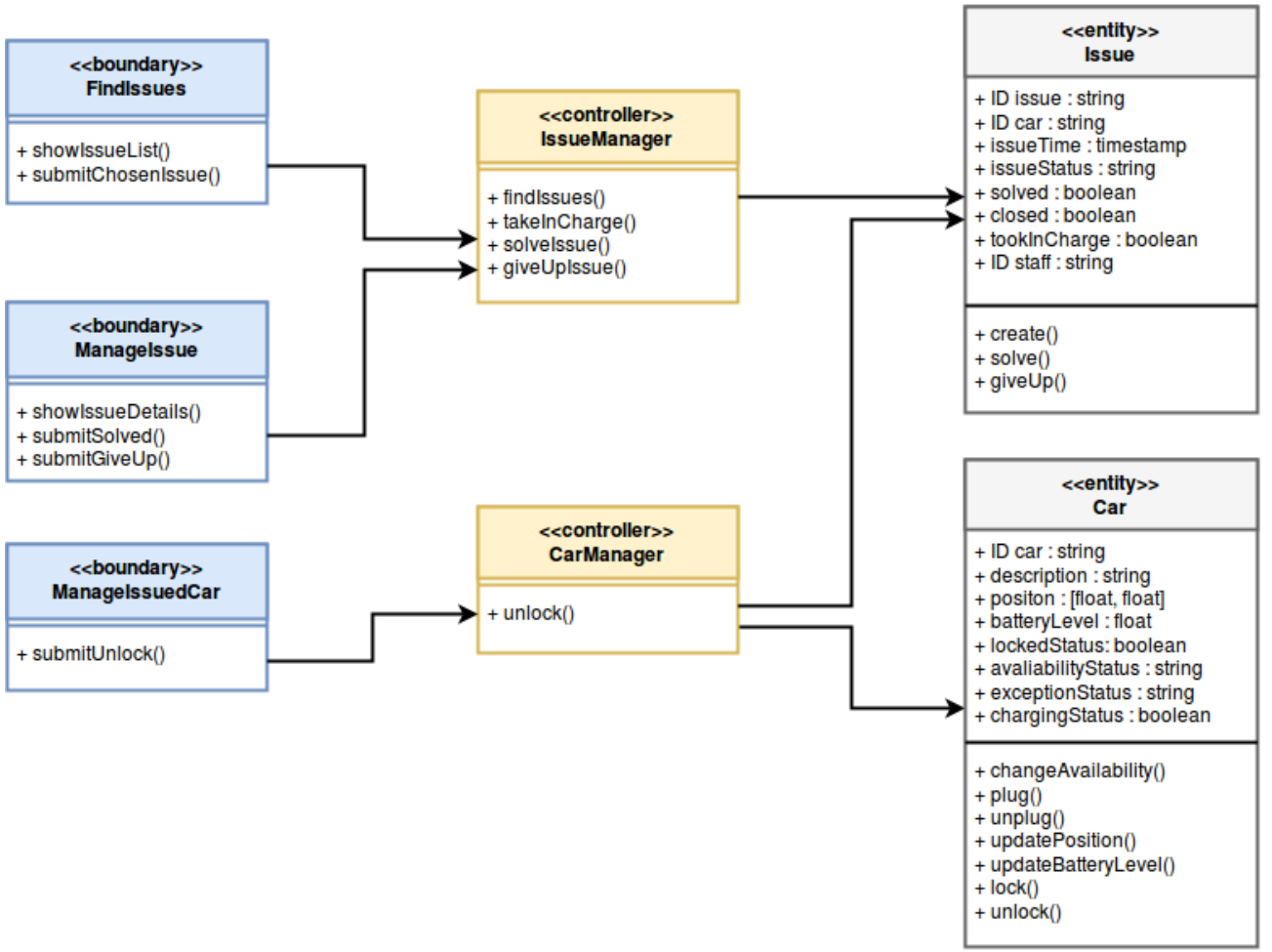
UX Diagrams



Section of the UX Diagram for the Staff's App Interface

User Interface Design (2/2)

BCE Diagrams



Section of the BCE Diagram for the Staff's App Interface

Part Three

TESTING STRATEGY

Entry Criteria & Integrations

Entry Criteria
As long as the **integration tests are automatic and fast** => no strong or specific precondition.
Run the integration test **after each commit** to catch **regressions as early as possible**.
Unit tests should be used to guarantee the interfaces are respected, that the component runs in isolation and fulfills its functionalities.

Elements to be integrated
All the components that interact with or make up the apps must be integrated with the Frontend.
The GEOLOCATION component needs to be integrated with the CarGUI.
BILLING SYSTEM, POSITION, GEOLOCATION, USER MANAGER should be integrated with external services.
Apart fror **IT04**

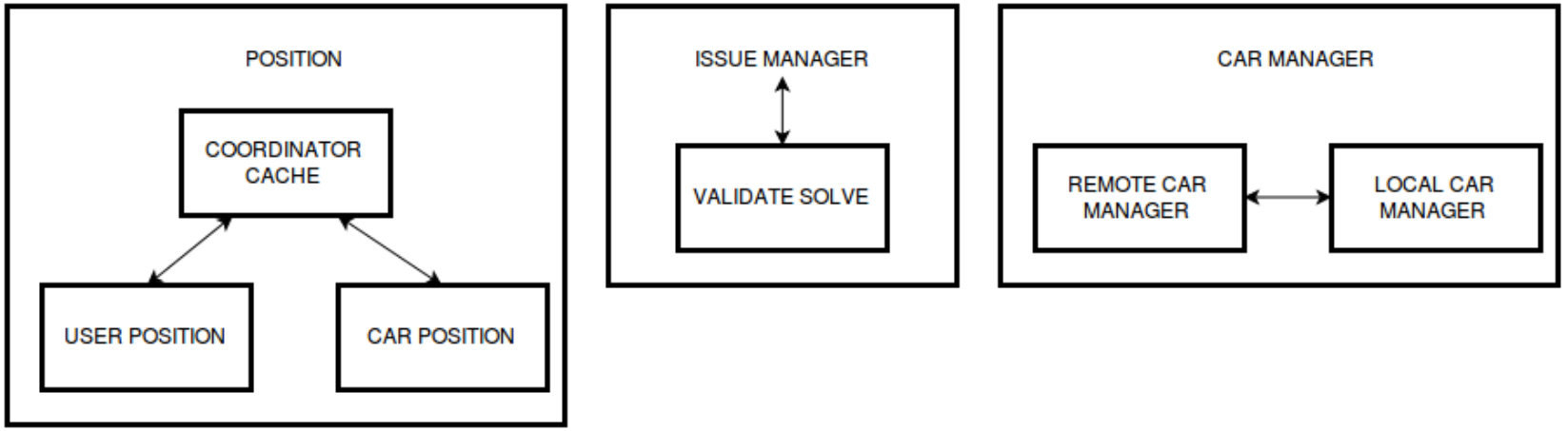
Component1 : BILLING_MANAGER
Component2 : CAR_MANAGER
Functionality : BILL/Pay
Description : When the car is locked, the payment procedure of the user starts.

Integration Sequence (1/2)

We chose to implement a **bottom-up** approach.

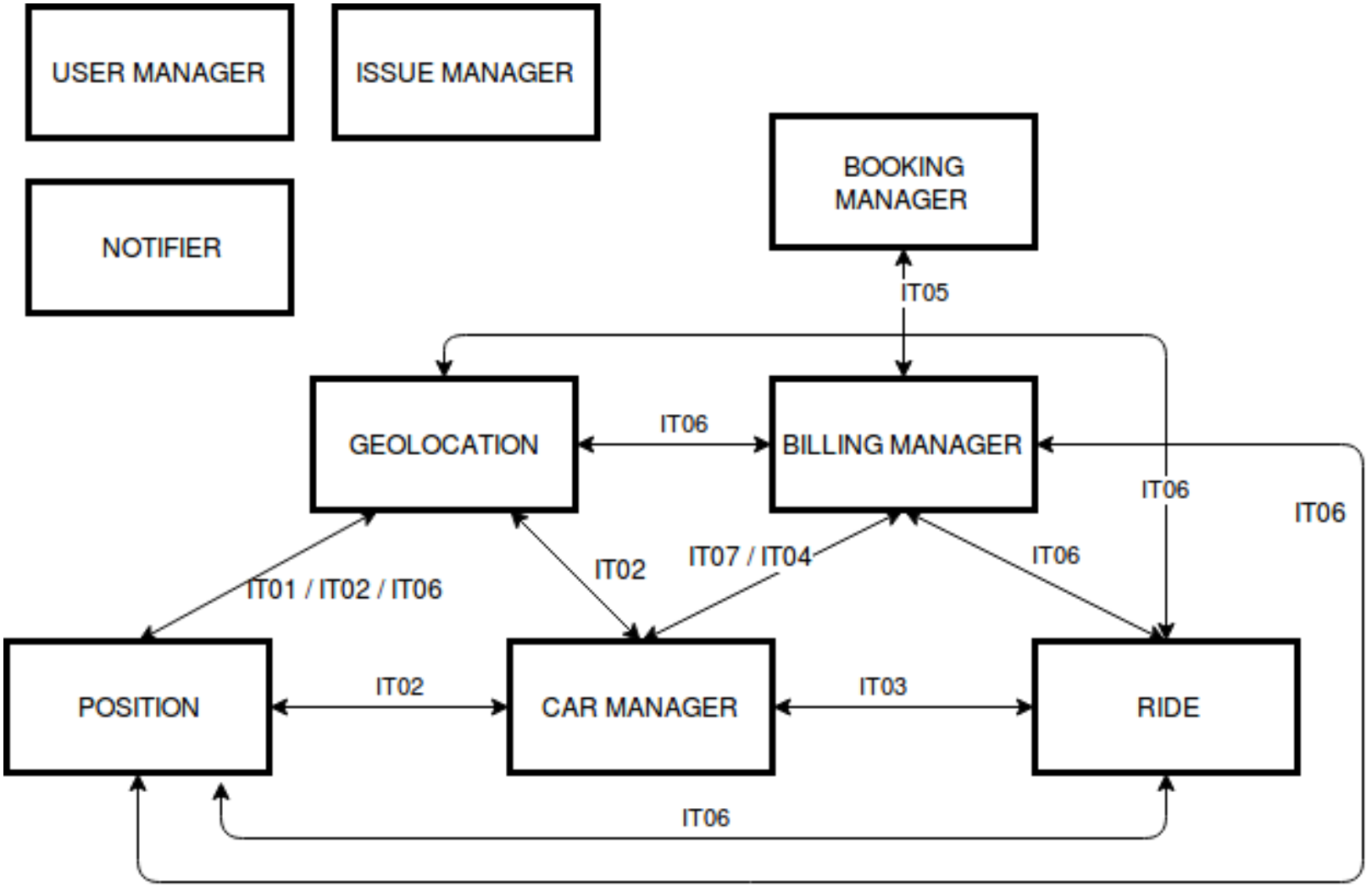
The only components that requires a sub-components integration are the three services that are physically distributed: CAR MANAGER, ISSUE MANAGER POSITION.

In parallel we can proceed integrating external services with our own high-level components, namely BILLING SYSTEM , POSITION , GEOLOCATION, and USER MANAGER.



Sub-components integration regarding physically splitted components into a single, high-level logical component.

Integration Sequence (2/2)



Integration net

Individual Steps and Descriptions

These tests are to be considered the bare minimum required to guarantee a satisfactory level of coverage.

We described only the most straightforward test that must be implemented; however, other tests can be deployed too.

Usually it is a good idea to test not only that a method is invoked and used, but also that a method is not being invoked. We do not specifically describe those tests because they are extremely implementation dependent.

Integration IT 04

1. **Test Case ID: IT04C1**

   Test Items: CAR_MANAGER → BILLING_MANAGER
   Input Specification: The locking signal is received by CAR_MANAGER
   Environment State: A ride for that car exist and is closed.
   Output Specification: The payment procedure starts.
   Purpose: Verify that after the locking signal has been sent, the payment procedure starts immediately.
   Dependencies: IT03C2 and unit tests for CAR_MANAGER and BILLING_MANAGER

Tools and Test Equipment

A very reasonable choice is JUnit4, which is the default testing framework for Java application.

Continuous Integration and Unit Testing
A server of continuous integration is necessary in order to execute the integration tests and the unit tests after each commit to the main code base.
The unit test suite is expected to be extremely fast.
For the integration test, we don’t require them to be fast but, at least, completely automatic.

Testing Strategy
Test should be produced in order to avoid regression.
After fixing a bug, a test relative to that same bug should be put in place.

Deploying
The deploy of the whole architecture should require the least amount of steps possible, ideally just one.
Considering the application is serving live traffic, a deploy of a new version should follow the incremental GREEN / BLUE methodology:

1. Start running the new software only a small fraction of the overall traffic to the new version
2. If everything on the control metrics still looks fine, increase the amount of traffic redirected to the new version until reaching the totality.
3. Otherwise, immediately roll back to the previous version at the first sign of problems.

Program Stubs and Test Data

Stubs
At this time, no stubs are strictly required to run our integration test suite.
All our external services expose some specific testing APIs that can be used to test their integration.

Test Data
The necessary test data will be generated using ad-hoc software.
There should be two different databases: one for tests and another for production.

Hardware Emulation
An emulation environment for the car’s onboard system should be enough to run all our integration tests.
Relying on the actual hardware for running integration test may lead to delays or to skip the run of fundamental integration tests.

Part Four

PROJECT MANAGEMENT

Project Size Estimation

We use the Function Point Approach to determine the size of the project.

Provided that all the components have an intrinsic, similar complexity, it is reasonable to assume that components that interact with few other components are quicker and simpler to develop. In our opinion the real complexity lies in the connection of components.

Estimation Outcomes
Internal Logical Files = 81
External Interface Files = 17
External Input = 102
External Inquiry = 3
External Output = 18

The overall total is: 81 + 17 + 102 + 3 + 18 = 221
Lower bound of: 221 * 46 = 10166 lines of code
Upper bound of: 221 * 67 = 14807 lines of code.

Cost Estimation (COCOMO II)

RELY	Precedentedness	Nominal	1.00
DATA	Development Flexibility	Very Low	n/a
CPLX	Product Complexity	Nominal	1.00
RUSE	Developed for Reusability	Nominal	1.00
DOCU	Documentation Match to Life-Cycle Needs	Nominal	1.00
TIME	Execution Time Constraint	Nominal	1.00
STOR	Main Storage Constraint	Low	n/a
PVOL	Platform Volatility	Nominal	1.00
ACAP	Analyst Capability	High	0.85
PCAP	Programmer Capability	High	0.88
PCON	Personnel Continuity	Extra High	n/a
APEX	Applications Experience	High	0.88
PLEX	Platform Experience	High	0.91
LTEX	Language and Tool Experience	High	0.91
TOOL	Use of Software Tools	High	0.90
SITE	Multisite Development	Very High	0.86
SCED	Required Development Schedule	Nominal	1.00
Product			0,422

Effort Estimation

$$Effort = A * EAF * KSLOC^{E}$$

$$A = 2.94$$
$$EAF = 0.422$$
$$E = B + 0.01 * sum(scaledriver)$$

$$E = 0.91 + 0.01 * 11.56 = 0.91 + 0.1156 = 1, 0256$$

$$F = 0.28 + 0.2 * (E - B) = 0.28 + 0.2 * (1.0256 - 0.91) = 0.303$$

Lower Bound	2.94 * 0.422 * 10.166 ** 1.0256	13.38
Upper Bound	2.94 * 0.422 * 14.807 ** 1.0256	19.68
Duration	3.67 * Effort ** F	?
Duration Lower Bound	3.67 * 13.38 ** 0.303	8.05
Duration Upper Bound	3.67 * 19.68 ** 0.303	9.05

Schedule

Timespan
Our risk mitigation strategy requires us to validate the product as early as possible.
Slower development cycles to ensures the quality of the final result.
We account for an additional month of time: roughly 10 person-months.

Deployment stages

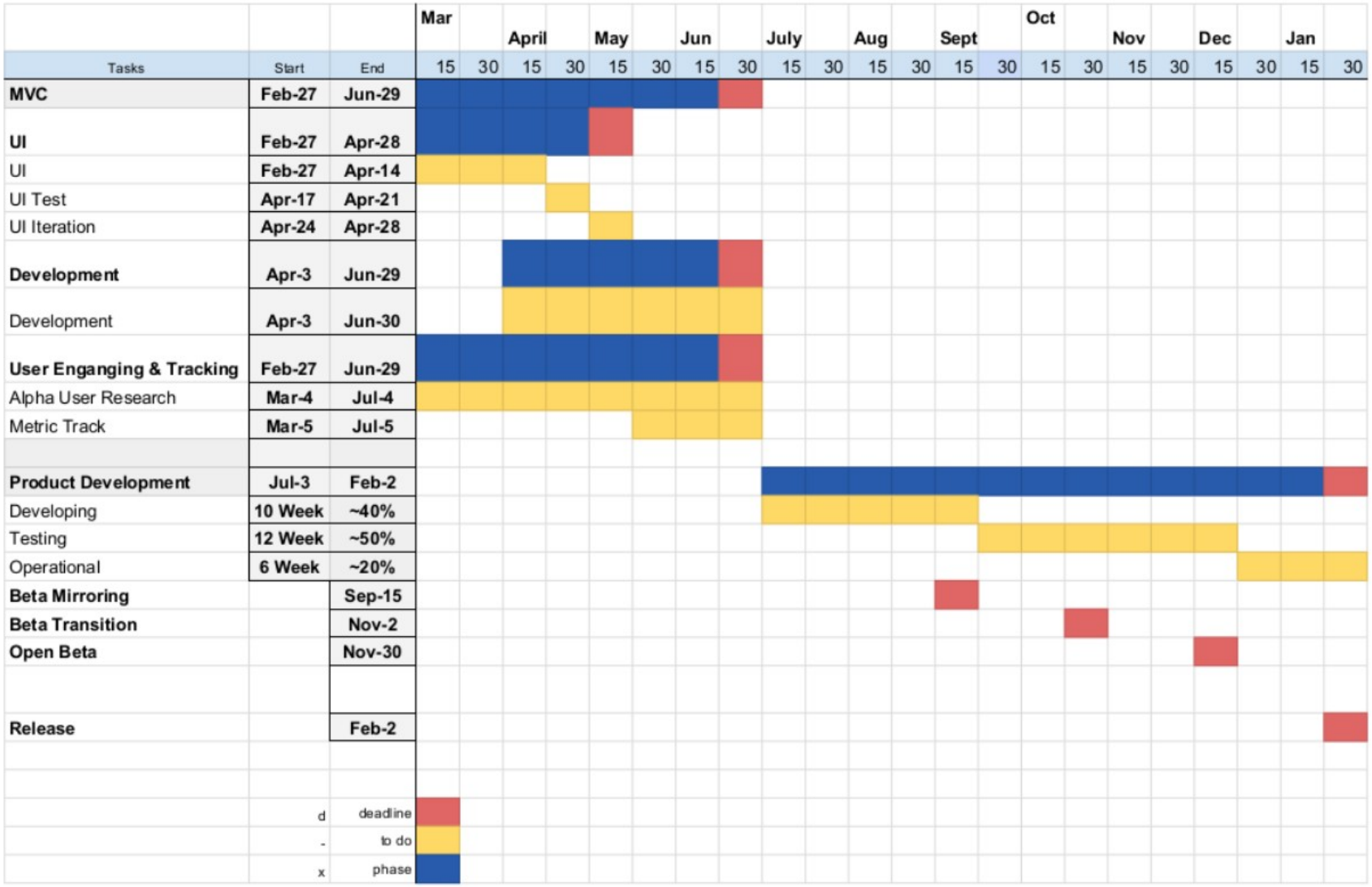
1. Develop a very simple MVP for selected alpha users.
2. Creation of the real platform.
3. Develop unit and integration tests (in parallel with the above one).

Deploy pipeline
Keep a clean building/deploy pipeline to be always able to deploy: expensive, but worth. In this way we can:

- Mirror the alpha traffic to the new service.
- Move alpha users to the new service and effectively make them beta users.
- Add more beta user to the service.

Longer development timings, but we lower risk of a later failure.

GANTT Diagram



Resource Allocation

A single developer will be responsible for every single component.

Sara is more experienced in UI / UX and in clearly communicating with the stakeholders:

1. USER
2. BOOKING
3. ISSUE
4. BILL
5. NOTIFY
6. User Interface

Simone is more experience in backend developing and M2M communication:

7. GEOLOCATION
8. POSITION
9. CAR
10. RIDE

Risk Management

- **Users don’t understand the User Interface**
  - Adopt standard symbols and icon
  - Test first mock up of the UI with target demography

- **Software more complex than our estimation**
  - Set several milestones along the development process, so to catch as early as possible if the project is going to run late.

- **Users don’t like the product**
  - Provide a MVP.
  - Get feedback from the market as soon as possible.

- **Low Quality software**
  - Automatic test regarding both performance and correctness.

Thank you

For you attention

Apache OFBiz

Code Review

Presented By:

Simone Mosciatti
Sara Zanzottera

Assigned Class

The class assigned to our group is:

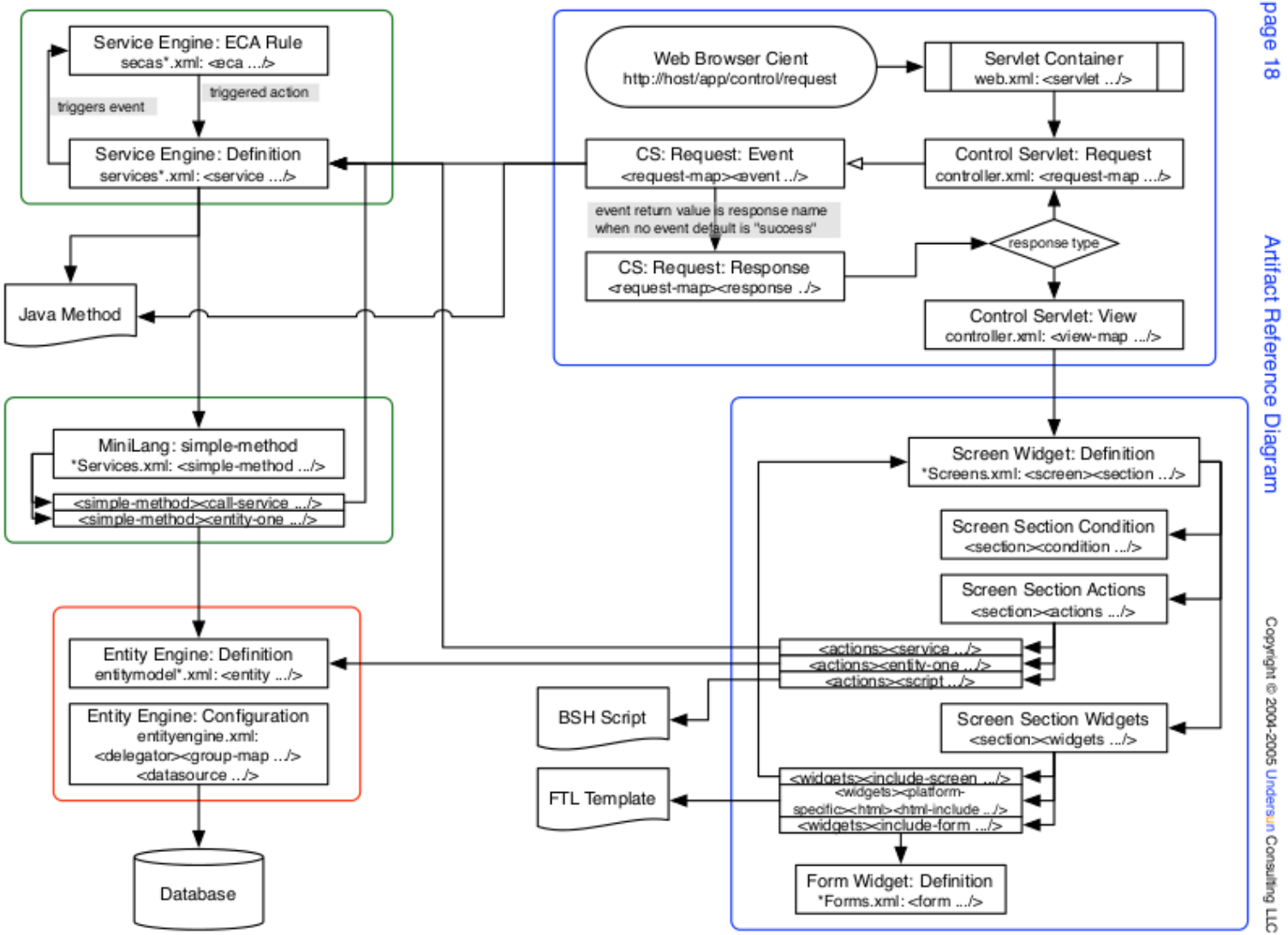
MiniLangUtil.java

(apache-ofbiz-16.11.01/framework/minilang/src/main/java/org/apache/ofbiz/minilang/MiniLangUtil.java)

Role
This class provides some utility functions for the MiniLang Engine.
Mini-Language is described on the official documentation as follows:
The Mini-Language concept in Open For Business is [...] to create simple languages that simplify complexor fre quently performed tasks.
The documentation does not provide in-depht informations about this section of the OFBiz framework: it is impossible to understand the actual role of the class without a detailed inspection of all the classes of the minilang package.

The results of a deeper analysis on the source code done with grep supports our previous hypotesis: MiniLangUtil is an internal class of the minilang package. It is never used outside the package and provides support functions for the processes of the minilang package.

Artifact Reference Diagram



Issues Found

MiniLangUtil class
The most issues found are usually of little impact on the source code quality, even if almost no function is found without any issue at all.

The most critical issue is the general lack of Javadoc or, when present, its low quality.
We highlight the presence of an hardcoded path into the source (.writeMiniLangDocument(), line 289: the path "component://minilang/config/MiniLang.xslt")

PlainString

MiniLangUtil include a public, inner class called PlainString.

This class, which can be found at line 323, is completely empty and shows no JavaDoc.
A public class in general has no reason for being internal, especially into a class like MiniLangUtil that could hold a default modifier instead of the public one. This solution simply makes difficult for developerd to locate PlainString into the project.

Being the class empty, we wondered if PlainString is even used into the project, or it is a leftover. So we performed a lookup into the code base to locate eventual usages.
We realized that PlainString is user thoughout the whole project, also outside the minilang package. Seems that PlainString is an entity, probably a field type or something similar,

In addition, the documentation shows no references about PlainString. Also
the online version of the Javadoc is of no use, as the class itself has no Javadoc.
While it remains unclear why developers should feel the need to use such an emptyclass to (probably) identify a plain string, and if PlainString is actually the way OFBiz deals with plain strings, the location of this class looks completely wrong. PlainString should have been located with other basic field types, and into a dedicated Java source file, instead of being a nested class of a randomly chosen class of the project..

Thank you

For you attention