POLITECNICO DI MILANO

Software engineering 2 project

TAXI SERVICE

DESIGN DOCUMENT

AUTHOR

Pallotta Simone

PROFESSOR

Raffaela Mirandola

Version 1.0

ACADEMIC YEAR 2015/2016

intentionally left blank

Introduction

1.1 Architecture Overview

The architecture cam be fundamentally split on four main tier:

- Client tier
- Web/Server tier
- Messaging Tier
- Persistence Tier

Client tier: It is mobile applications (iOS, Android) that interacts directly with the two type of user.

User can received information and updates but also interacts sending request, geo position and confirmation.

Mobile application will be implementing using a native approach so users will experience the native UX trough the native UI

Web/Server tier: Consists of standard Linux Server running apache with HTTP services written in ruby.

The server will accept request from client and will deliver confirmation and updates to the client tier.

Messaging Tier: It is in charge of delivering messaging to push notification cloud based on apple push notification servers for iOS platform.

This tier will not be implementing but we rely on their services.

Persistence Tier: It is the data source tier based on mysql application server that stores all user information , all the request and all the confirmation.

The queue sub system abstraction is implemented using DDL and DML available on this tier.

1.2 Identifying sub-system

We decide to adopt a top-down approach at least at this point of the project because allowed separation of function and separate development of the different sub system.
We divided the system into other sub-systems, in order to make it easy to easily implement the different functionalities separated logically into groups.

This separation allows independent implementation of sigle behaviour allowing better management of software development cycle and maintenance.
We separate our systems into these sub-systems:

- Signup/Sign in sub-system
- Travel request sub-system
- Queues sub-system
- Travel dispatcher sub-system
- Messaging sub-system
- Network REST sub-system

1.3 Design Language

In the following document will be used specific language for each development environment:

- Datebase design: E/R diagram
- Logic database design: logic scheme
- MVC design: MVC diagram

1.4 Summary

This document exposes all the main aspects close to software design of the proposed solution that showed the following points:

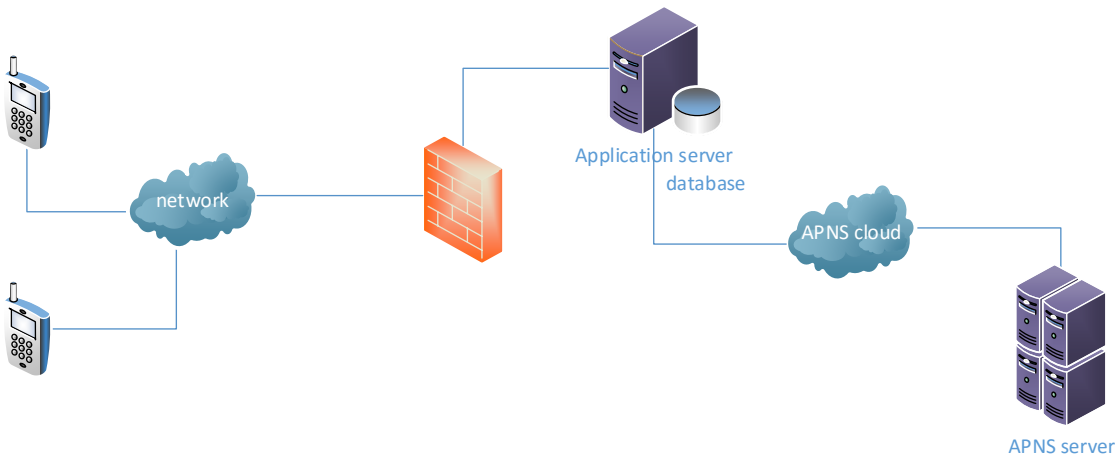
1. system architecture made up by different tiers and layers and the interaction among themselves
2. database conceptual structure and logic design
3. Class design
4. MVC design

System Architecture

2.1 Architecture, layers and tiers

As described in RASD document, the architecture is based in there main components.

- The mobile application that contains: client layer and presentation layer.
- Application Server that contains: service layer, business layer, hosting layer, but also physically contains the database layer. The application server manages the request from client via REST services, manages the queues via DML commands and deliveries messages to mobile app but also to the external messaging system.
- (A)PNS infrastructure is not part of our development effort but is essential for the final solution to deliver messages using standard mobile channels.



In the final deployment out solution will implement some sort of security / filtering of http/https traffic to prevent exploits and DDOS.

This is achieved using a Firewall.

The application server can connect to the database app in the internal zone, or better, as described above, using local loopback if installed on the same machine enhancing security on the Database.

Its important also set correctly the firewall's ports to allows the access to services exposed by application server.

During the development phase this will be kept down to better debug the services.

For our app the need only standard http/https ports, so only port 80 / 443 should be set on firewall.

2.2 Queues and queues management

The business model is based on the concept of queues.

Every zone has one independent queue that contains all the available taxi for the zone.

The algorithm will scan the queue from the first slot for an 'available' taxi, sends the request for confirmation to the taxi.

The state of taxi in the queue is now suspending until ACK or NACK from the cab.

The next scan must skip 'suspended' taxi while searching for a free slot.

When receiving an ACK the system changes the state of taxi in queue to 'busy' and sends a notification to the customer.

When receiving an NACK the system push the taxi to the bottom of the queue and the status is reset to available.

When receiving 'end of trip message' the system changes the state of taxi to 'available' and moves it to the bottom of the queue.

In case there is no available slot the system refuses the request without storing it, but simply sends a refused message to the customer.

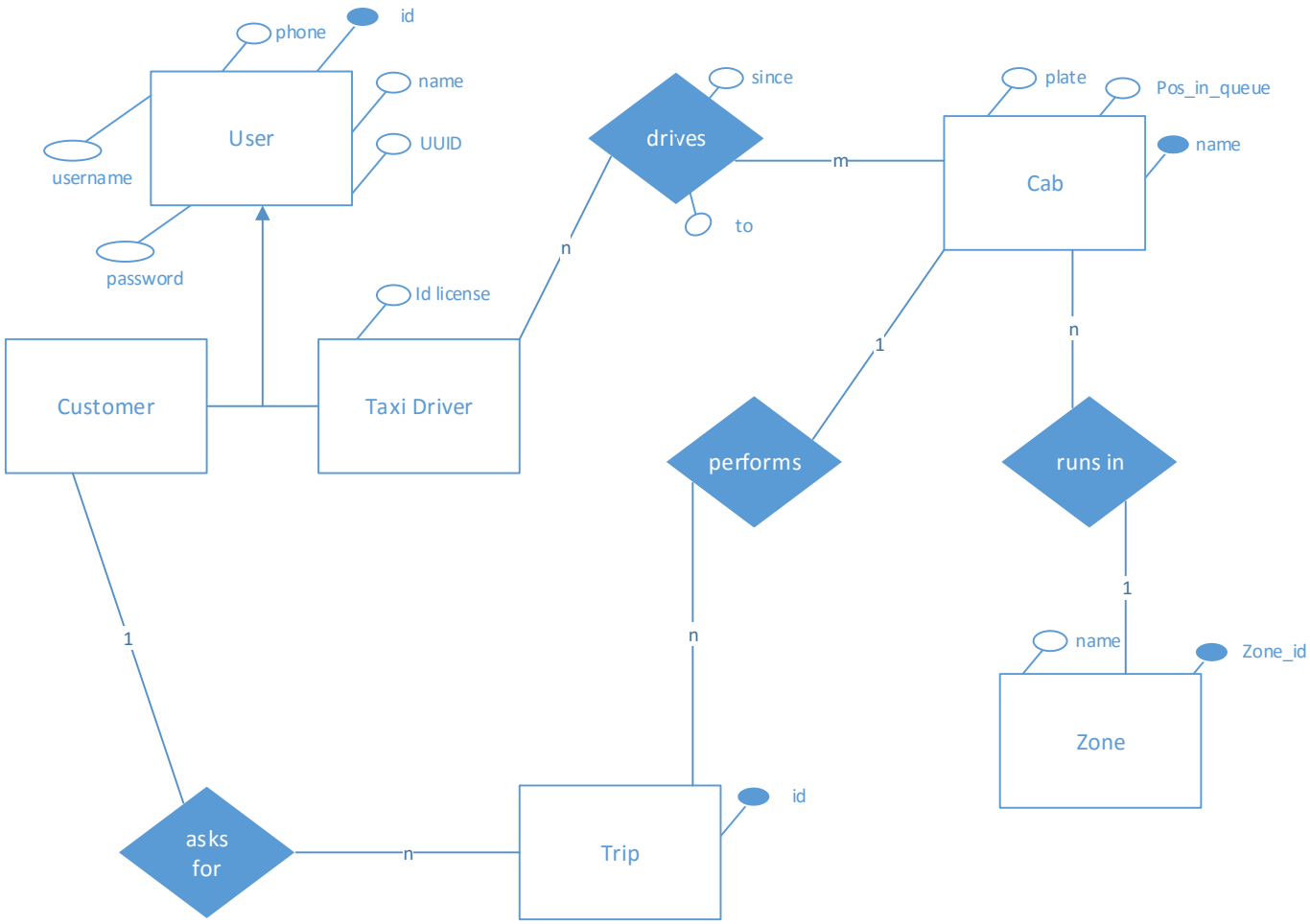
All these requests are executed in parallel as they are run as ruby script on the server, so we must put care in the concurrency model.

The queue will be implementing neither a arrays nor its a tables: the queue is an abstract concept we implement using attributes and foreign keys of the entities.

Persistent Data Management

3.1 Conceptual design

In the following E/R scheme we show the conceptual model of our software solution with a little description of each relation with its own cardinality.



3.1.1 Relationship

- 1 customer *asks for* N trips but 1 trip ca be *asked* by only 1 customer
- 1 cab *performs* N trips but 1 trip *is performed by* only 1 cab
- 1 cab *runs in* 1 zone but in 1 zone *runs* N cabs
- 1 taxi driver *drives* N cabs and M cabs *are driven by* N taxi drivers in different periods

3.1.2 ER Restructuration

In order to implement the real database we must apply some transformations.

In our system we have two types of user: customer and taxi driver.

We need store different information about entities so we can split the generalisation in two different entities, even if some fields are similar.

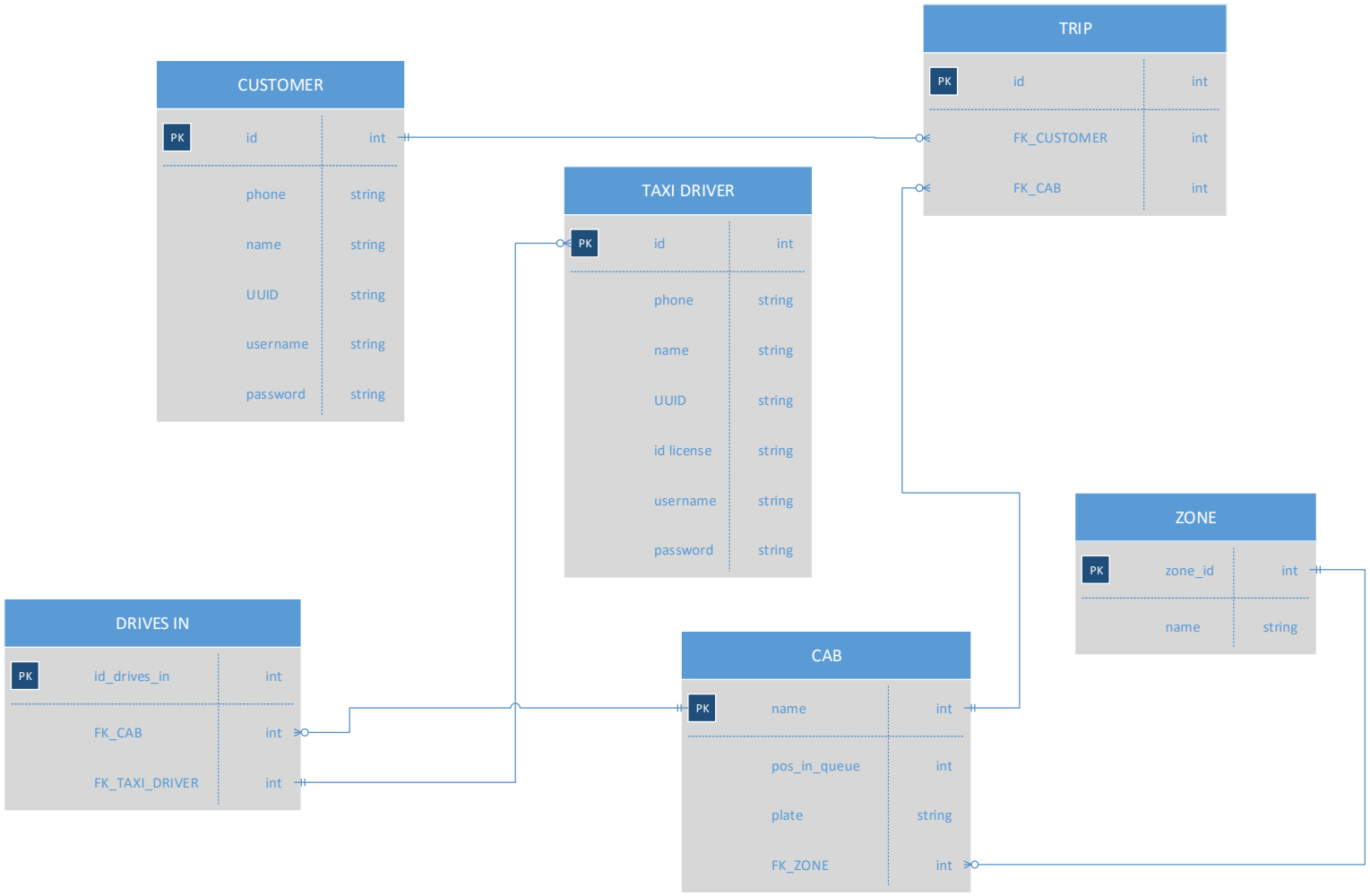
An other feasible approach can be to add an attribute 'type' that can assume the values **'d'** for driver or **'c'** for customer.

We have chosen the first approach.

Another case concern the relationship *drives*, is an m-n relationship so in database we need to create an intermediate table *drives* where we use two attributes that contains the foreign keys to *cabs* and *taxi drivers* respectively.

3.1.3. Translation to Logical Model

- relation *ask for* between customer and trip is translated using a foreign key into trip entity
- relation *performs* between trip and cab is translated using a foreign key into trip entity
- relation *runs in* between cab and zone is translated using a foreign key into cab entity
- relation *drives* is translated into a real table contained the foreign keys of taxi driver and cab


CUSTOMER
PK
id
int
phone
string
name
string
UUID
string
username
string
password
string
TRIP
PK
id
int
FK_CUSTOMER
int
FK_CAB
int
TAXI DRIVER
PK
id
int
phone
string
name
string
UUID
string
id license
string
username
string
password
string
ZONE
PK
zone_id
int
name
string
DRIVES IN
PK
id_drives_in
int
FK_CAB
int
FK_TAXI_DRIVER
int
CAB
PK
name
int
pos_in_queue
int
plate
string
FK_ZONE
int

3.2 Logic design

The final model has the following physical structure:

User (ID_user, username, name, email, password,UUID, phone)

Taxi_driver (*ID_taxi_driver*, username, name, email, password, UUID, phone, id_license)

Cab (*NAME*,pos_in_queue, plate,*FK_ZONE*)

Zone (*ID_Zone*; name)

Trip(*ID_trip*,*FK_CUSTOMER*,*FK_CAB*)

Drives(since,to,*FK_TAXI_DRIVER*,*FK_CAB*)

Class Design

On the server we basically implement only the M part of the pattern as we use *Ruby on rails Active records* that allow to map entities to classes and tuples to objects

3.1 Active Record

Active Record is the M(odel) in MVC responsible for representing business data and logic facilitating the use of business objects whose data requires persistent storage to a database.

Active Record use the following concept:

- Represent models and their data.
- Represent associations between these models.
- Represent inheritance hierarchies through related models.
- Validate models before they get persisted to the database.
- Perform database operations in an object-oriented fashion.

3.1.1 Name mapping

Active record uses naming conventions to map between models and database tables;

in our solution we have the following mappings on the server side:

Model / Class	**Table / Schema**
Customer	customers
TaxiDriver	taxi_drivers
Trip	trips
Zone	zones
Cab	cabs

On the client side we map between JSON and classes, both in upload and in download

“user":

{

"username": "John",

"password": "Doe"

"name": "John",

"mail": “John@Doe.com”,

"phone": "+892384768"

"UUID": “sns83hdfk392jh4pd39047"

}

```
{
“taxi_driver”:
{
    "username": "John",
    "password": "Doe"
    "name": "John",
    "mail": “John@Doe.com”,
    "phone": “+892384768",
    "UUID": “sns83hdfk392jh4pd39047”,
   “car_license”:”34543fgdg”
}
}
```

```
{
“verification_code”:
{
    "username": "John",
    "password": "Doe"
    "UUID": "sns83hdfk392jh4pd39047",
    “code”:”23432342”
}
}
```

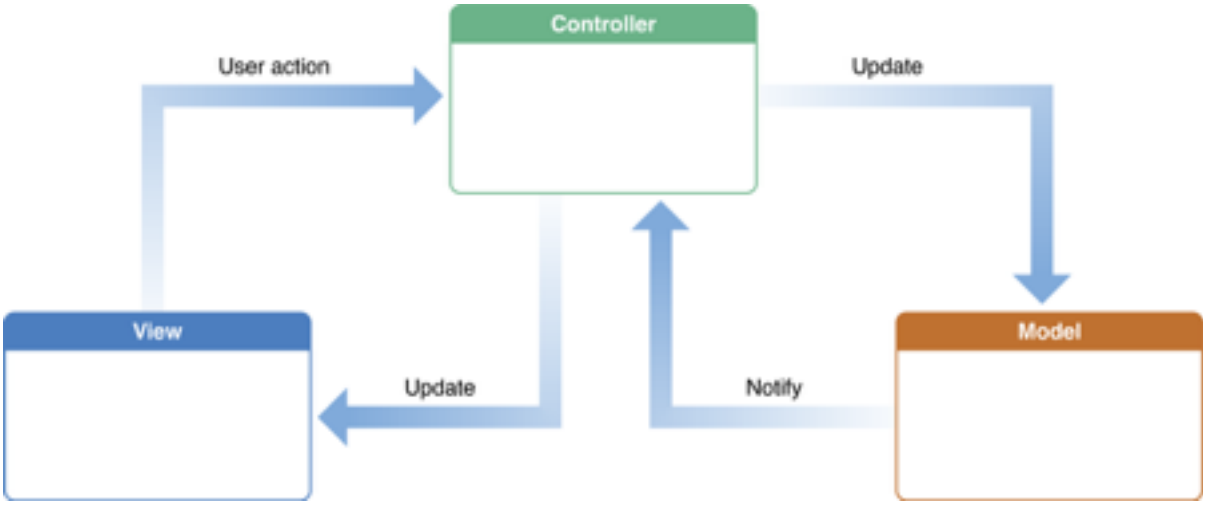
```
{
“asking_ride”:
{
    "username": "John",
    "password": "Doe"
    "UUID": "sns83hdfk392jh4pd39047",
   "position":
   {
     "lat": "34534.54",
     "long": "5345.435"
   }
}
}

{
  "request_ride":
{
   "firstName": "John",
   "id_ride": "3",
   "lastName": "Doe",
   "position": {
     "lat": 34534.54,
     "long": 5345.435
   },
   "status": "accepted"
  }
}
```

MVC Design

We can distinguish two different MVC patterns in our solution:

On the mobile application we have a complete MVC schema implemented in software where the model will be implemented using swift/java classes, views using standard object oriented UI elements and controllers will be classes contains the business logic.



On the server we basically implement only the M as written before about Class design.

4.1 MVC Diagram