CITYFIX - THE WORLD YOU KNOW. QUITE BETTER.

---

1. VISION

CityFix is a social app that makes people feel and live more their own city. Using this application users will be able to help fixing some of their city problems. For example, a person walks through the same streets everyday and detects the same problems without being fixed (broken water pipes, missing traffic signs, etc.). By using the application, the user can upload a picture of that problem, using our webapp or when it gets home on its personal computer, and it will be redirected to the responsible authorities. CityFix, the world you know. Quite better.

2. STATE OF THE ART

2.1 Technologies

Node JS
The main idea of Node.js is using non-blocking, event-driven I/O to remain lightweight and efficient in the face of data-intensive real-time applications that run across distributed devices.

Advantages

- Building fast, scalable network applications;
- Capable of handling a huge number of simultaneous connections with high throughput;
- Asynchronous I/O.

Disadvantages

- Bad concurrency primitives;
- Single-Threaded;
- Lack of maturity;
- Reliance on stringly-typed programming;
- Hard to make things fault-tolerant;
- JavaScript's semantics and culture.

Summary

Good Use Cases

- JSON APIs;
- Single page apps;
- Shelling out to unix tools;
- Streaming data;
- Soft Realtime Applications.

Bad Use Cases

- CPU heavy apps;
- Simple CRUD / HTML apps.

AngularJS

AngularJS is a structural framework for dynamic web apps. It lets using HTML as template language and extend HTML's syntax to express the application's components clearly and succintly. Also, Angular's data binding and dependency injection eliminate much of the code it has to be written.

Advantages

- Provides capability to create Single Page Application in a very clean and maintainable way;
- Provides data binding capability to HTML thus giving user a rich and responsive experience;
- Code is unit testable;
- Uses dependency injection and make use of separation of concerns;
- Provides reusable components;
- Less code and more functionality;

Disadvantages

- Not secure - being JavaScript only framework, applications written are not safe. Server side authentication and authorization is must to keep an application secure;
- Not degradable - if the user disables JavaScript then the user will just see the basic page and nothing more.

3. ARCHITECTURAL REQUIREMENTS

CityFix is a social app that makes people feel and live more their own city. Using this application, users will be able to help fixing some of their city problems. For example, a person walks through the same streets everyday and detects the same problems without being fixed (broken water pipes, missing traffic signs, etc). By using the application the user can upload a picture of that problem and it will be redirected to the responsible authorities.

The person can upload the picture using our web app or when it gets home on its personal computer. This report aims to give the reader an idea of the system and its architecture, as well as the interactions between the components. The document includes the logical and technological architecture, the main design decisions and technologies to be used.

3.1 Technological Architecture

The technologies in Table 1 were chosen for two main reasons: product and development efficiency, by choosing the newest technologies in their stable versions.

Technology	Version
Bootstrap	3.3.5
AngularJS	1.4.7
SQLite	3.1.3
Ruby on Rails	4.2

Table 1: Technologies

- **Bootstrap** - free and open-source collection of tools for creating websites and web applications. It contains HTML and CSS-based design templates for typography, forms, buttons, navigation and other interface components, as well as optional JavaScript extensions. It aims to ease the development of dynamic websites and web applications.

- **AngularJS** - open-source web application framework maintained by Google and by a community of individual developers and corporations to address many of the challenges encountered in developing single-page applications. It aims to simplify both the development and the testing of such applications by providing a framework for client-side model–view–controller (MVC) and model–view–viewmodel (MVVM) architectures, along with components commonly used in rich Internet applications.

- **PostgreSQL** - an object-relational database management system with an emphasis on extensibility and on standards-compliance. As a database server, its primary function is to store data securely, supporting best practices, and to allow for retrieval at the request of other software applications. It can handle workloads ranging from small single-machine applications to large Internet-facing applications with many concurrent users.

- **Ruby On Rails** - model–view–controller (MVC) framework, providing default structures for a database, a web service, and web pages. It encourages and facilitates the use of web standards such as JSON or XML for data transfer, and HTML, CSS and JavaScript for display and user interfacing. In addition to MVC, Rails emphasizes the use of other well-known software engineering patterns and paradigms, including convention over configuration (CoC), don't repeat yourself (DRY), and the active record pattern.

3.2 Logical Architecture

Our systems is built on the Ruby on Rails framework. The data connection to the PostgreSQL database is handled by this framework. Upon it we have the business model with the the features grouped by function and by model. We'll use AngularJS to develop the controllers and also JQuery to handle the views.

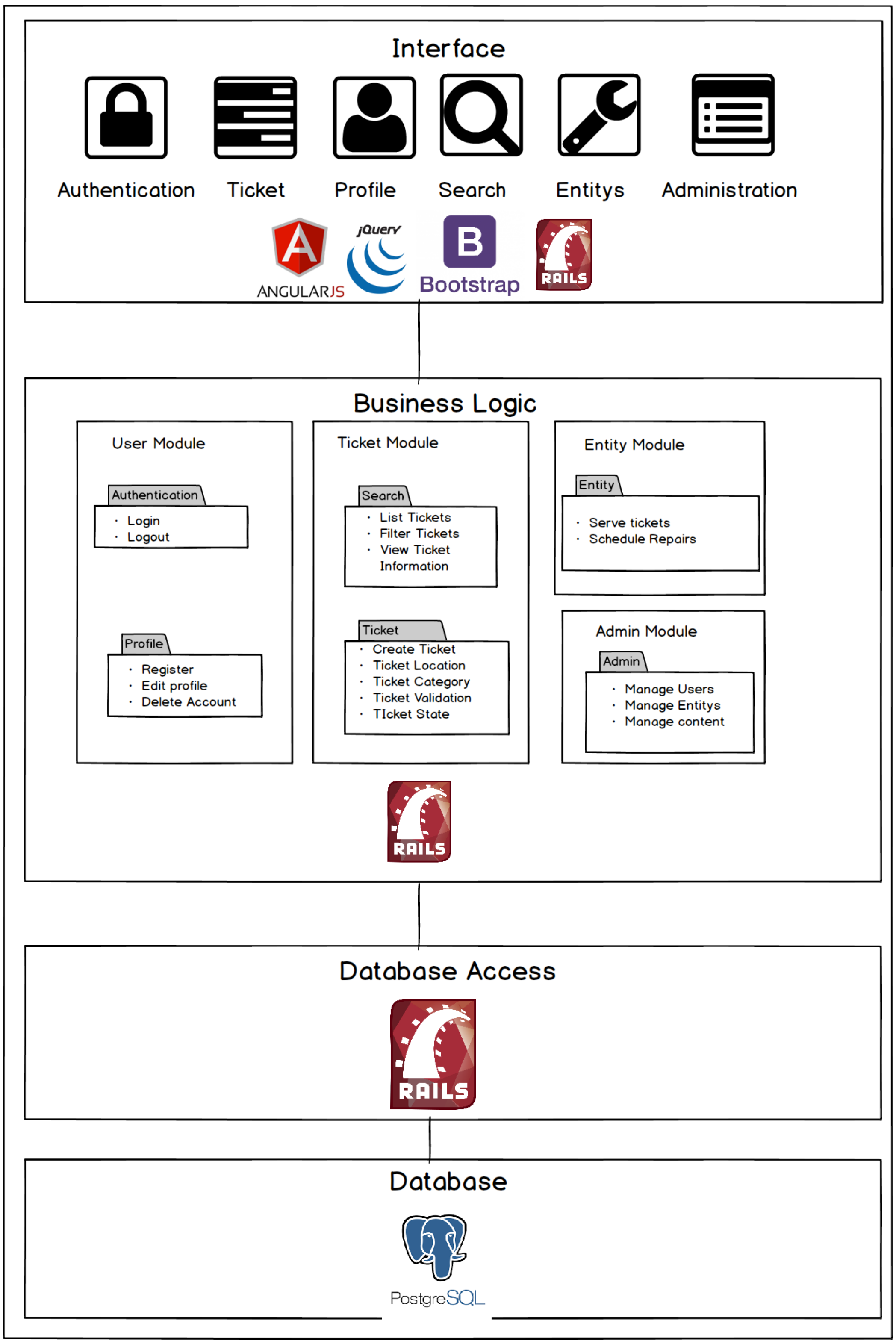


Figure 1: Logical Architecture

3.3 Physical Architecture

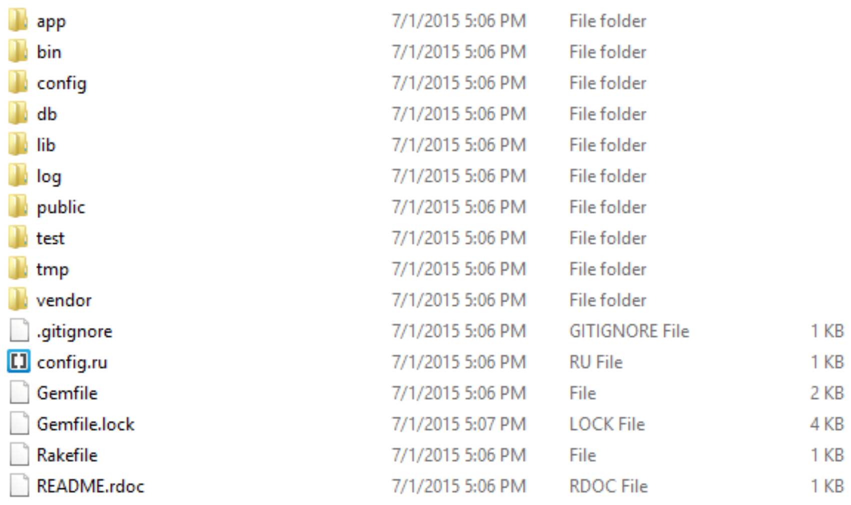
Our initial design decisions were made upon choosing the right technologies. For a quick, responsive interface development, we chose Bootstrap. Enforcing this decision is the fact that we will not use a native mobile app - instead we will use a Web app.

We chose Rails over Grails, since we want a MVC based framework, upon which we will stack other technologies like AngularJS. To support this decision is the fact that other than server requests we don't have many asynchronous actions, and the requests itself should be simple and fast. To counter the need for heavy requests and asynchronous requests we will use promises and web workers.

The database of our preference is PostgreSQL rather than MongoDB since we are more comfortable with relational databases and since our database needs fits PostgreSQL better.

We will have an authentication system to control security violations and other abuses on the system.

We decided upon a logging system on administrative actions to better manage potential conflicts on administrative action. We'll also have input verification with error handling on JSON (server response).



app	7/1/2015 5:06 PM	File folder	
bin	7/1/2015 5:06 PM	File folder	
config	7/1/2015 5:06 PM	File folder	
db	7/1/2015 5:06 PM	File folder	
lib	7/1/2015 5:06 PM	File folder	
log	7/1/2015 5:06 PM	File folder	
public	7/1/2015 5:06 PM	File folder	
test	7/1/2015 5:06 PM	File folder	
tmp	7/1/2015 5:06 PM	File folder	
vendor	7/1/2015 5:06 PM	File folder	
.gitignore	7/1/2015 5:06 PM	GITIGNORE File	1 KB
config.ru	7/1/2015 5:06 PM	RU File	1 KB
Gemfile	7/1/2015 5:06 PM	File	2 KB
Gemfile.lock	7/1/2015 5:07 PM	LOCK File	4 KB
Rakefile	7/1/2015 5:06 PM	File	1 KB
README.rdoc	7/1/2015 5:06 PM	RDOC File	1 KB

Figure 2: Directory Structure

Now let's explain the purpose of each directory:

app - It organizes your application components. It’s got subdirectories that hold the view (views and helpers), controller (controllers), and the backend business logic (models).

app/controllers - The controllers subdirectory is where Rails looks to find the controller classes. A controller handles a web request from the user.

- **app/helpers** - The helpers subdirectory holds any helper classes used to assist the model, view, and controller classes. This helps to keep the model, view, and controller code small, focused, and uncluttered.

- **app/models** - The models subdirectory holds the classes that model and wrap the data stored in our application’s database. In most frameworks, this part of the application can grow pretty messy, tedious, verbose, and error-prone. Rails makes it dead simple!

- **app/view** - The views subdirectory holds the display templates to fill in with data from our application, convert to HTML, and return to the user’s browser.

- **app/view/layouts** - Holds the template files for layouts to be used with views. This models the common header/footer method of wrapping views. In your views, define a layout using the `layout:default` and create a file named default.html.erb. Inside default.html.erb, call <% yield %> to render the view using this layout.

- **components** - This directory holds components, tiny self-contained applications that bundle model, view, and controller.

- **config** - This directory contains the small amount of configuration code that your application will need, including your database configuration (in database.yml), your Rails environment structure (environment.rb), and routing of incoming web requests (routes.rb). You can also tailor the behavior of the three Rails environments for test, development, and deployment with files found in the environments directory.

- **db** - Usually, your Rails application will have model objects that access relational database tables. You can manage the relational database with scripts you create and place in this directory.

doc - Ruby has a framework, called RubyDoc, that can automatically generate documentation for code you create. You can assist RubyDoc with comments in your code. This directory holds all the RubyDoc-generated Rails and application documentation.

- **lib** - You'll put libraries here, unless they explicitly belong elsewhere (such as vendor libraries).

- **log** - Error logs go here. Rails creates scripts that help you manage various error logs. You'll find separate logs for the server (server.log) and each Rails environment (development.log, test.log, and production.log).

- **public** - Like the public directory for a web server, this directory has web files that don't change, such a s JavaScript files (public/javascripts), graphics (public/images), stylesheets (public/stylesheets), and HTML files (public).

- **test** - The tests you write and those that Rails creates for you, all goes here. You'll see a subdirectory for mocks (mocks), unit tests (unit), fixtures (fixtures), and functional tests (functional).

- **tmp** - Rails uses this directory to hold temporary files for intermediate processing.

vendor - Libraries provided by third-party vendors (such as security libraries or database utilities beyond the basic Rails distribution) go here.

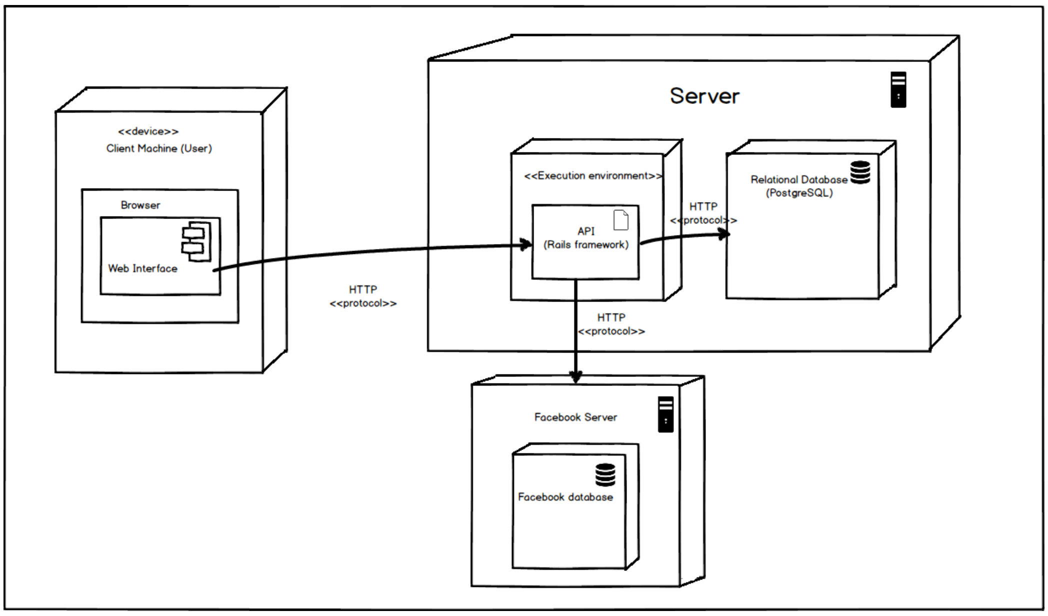


Figure 3: Physical Architecture

3.4 Key Design Decisions

A key design decision that was made while planning the implementation of this project was to base the architecture on a proven web development framework. We chose Ruby on Rails for a number of reasons:

- It is a Model-View-Controller (MVC) framework, with clear separation of concerns between the presentation (web) layer, web services and database
- It emphasizes well-known software engineering paradigms by focusing on convention over configuration and don't-repeat-yourself (DRY) principles, among others
- Because it brings a high level of functionality out-of-the-box, it accelerates the development of software applications and, thus, is considered agile-oriented

Still concerning technological choices, AngularJS was chosen to fulfill the gap between server-side generated pages and the need to have a high degree of interactivity on the project's pages, running on the browser. In order to better separate the DOM structure of the generated HTML pages from the functionality and interactivity provided by the Javascript code that will run on those pages, we decided a framework like AngularJS would be very relevant. As such, this framework will allow us to implement an MVC pattern on the client-side, with major benefits in terms of separation of concerns.

We should also add that Security and Logging are also important parts of the architecture, and as such, will be considered in all layers of the implementation.

4. HORIZONTAL PROTOTYPE

This section, the Horizontal Prototype, has two main goals:

- help identify and describe the user requirements;
- give a perspective of how the product user interface will look like and evoke new requirements;

First, it's described the general interface principles and common characteristics to all pages. Then it's presented the system overview on user's perspective (sitemap), followed by the main interactions with the system in order to illustrate the steps associated to each identified scenario. Lastly, it's presented a sequence of screenshots representing the interfaces, called storyboards.

The interfaces are presented and described at the end of this section.

4.1 Interface and General Principles

The user interface of **CityFix** is a set of Web pages implemented using the last standards: HTML5, Javascript, CSS3 and AngularJS.

On Figure 4 it's presented the Web pages layout that will be part of **CityFix**.

4.2 Overview

A Sitemap is visual representation of the information space in order to help users understand where they can go. Sitemaps can provide such a visualization, offering a useful supplement to the primary navigation features on a website [Nielsen08].

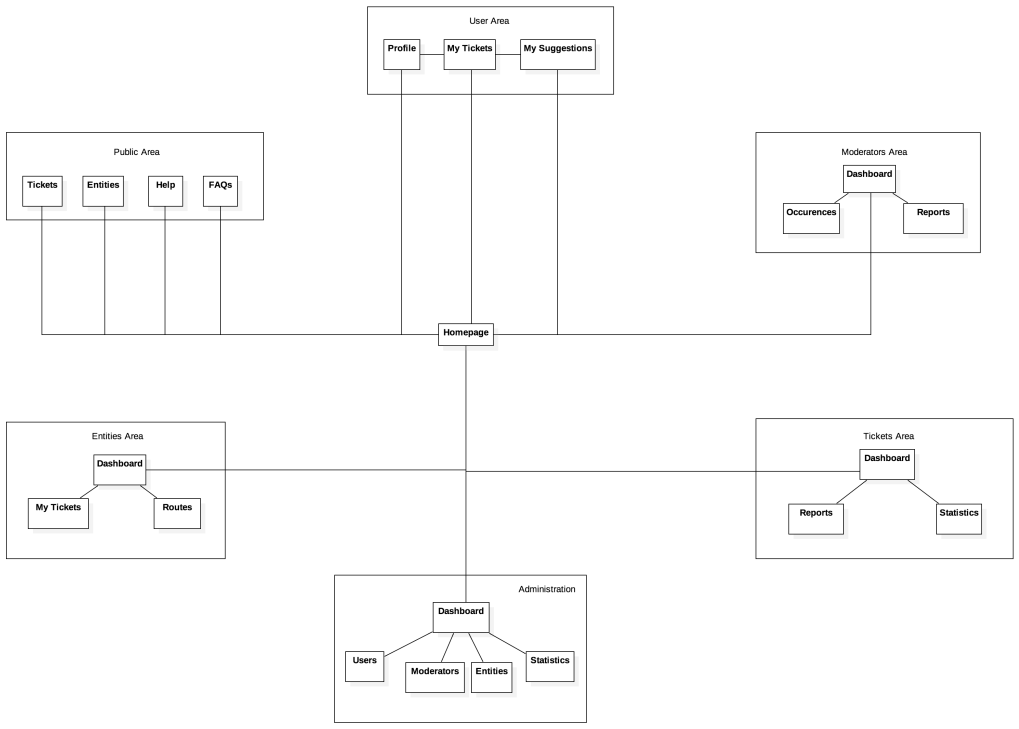


Figure 5: Sitemap

4.3 Interactions

On this section it's described the key interactions with the system to illustrate the sequence of steps associated with each of the identified usage scenarios.

The presented Activity UML Diagrams (or flowcharts [Brown10]) describe how people complete their tasks, through a series of screens which collect or provide information to the user.

These diagrams may not reveal all the interaction details but they should provide a complete experience oversight of its use to reach a particular goal.

Scenario 1: Sign In

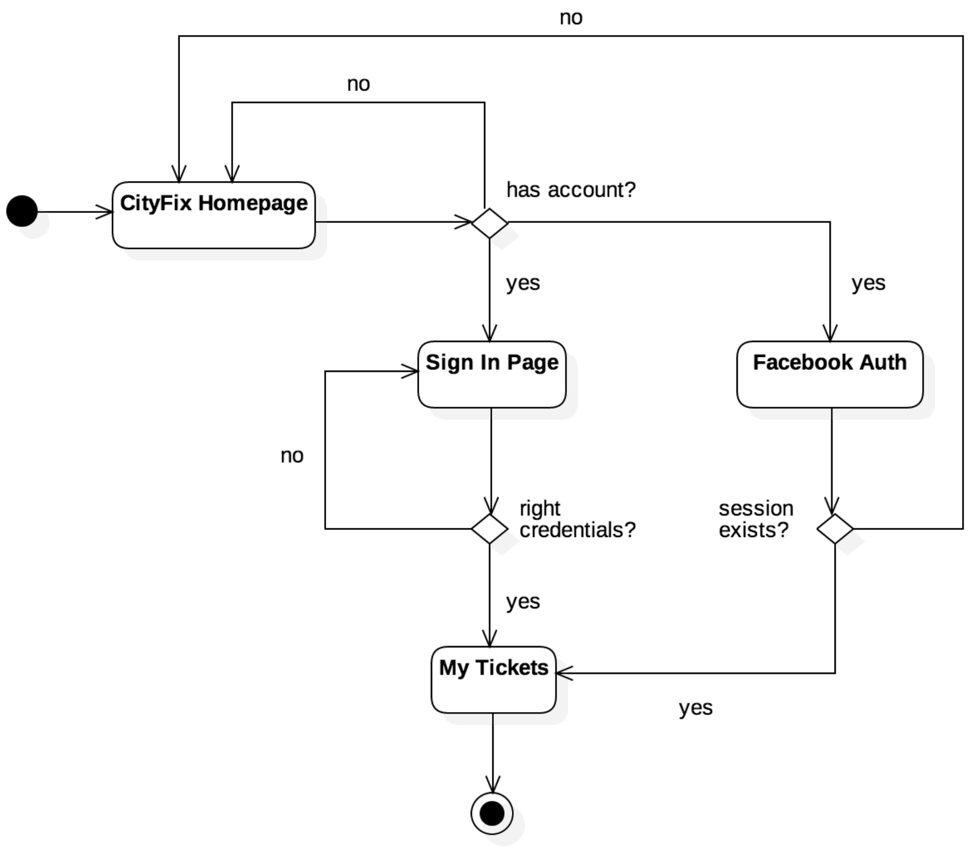


Figure 6: Sign In Scenario

Scenario 2: Sign Up

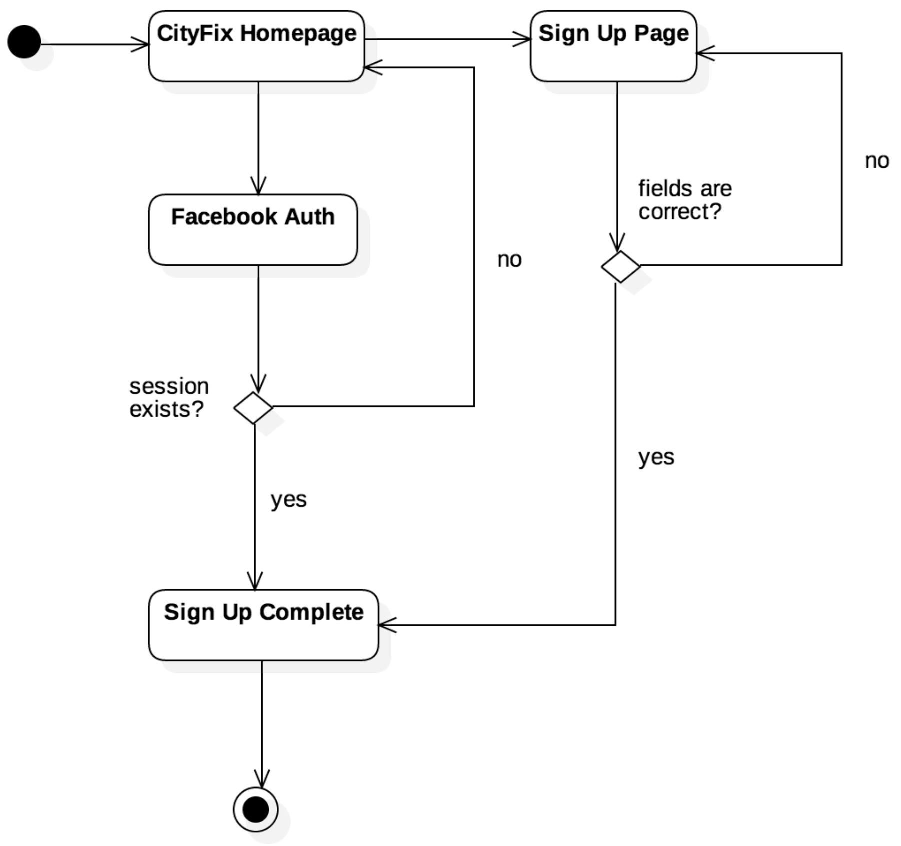


Figure 7: Sign Up Scenario

Scenario 3: Add Ticket

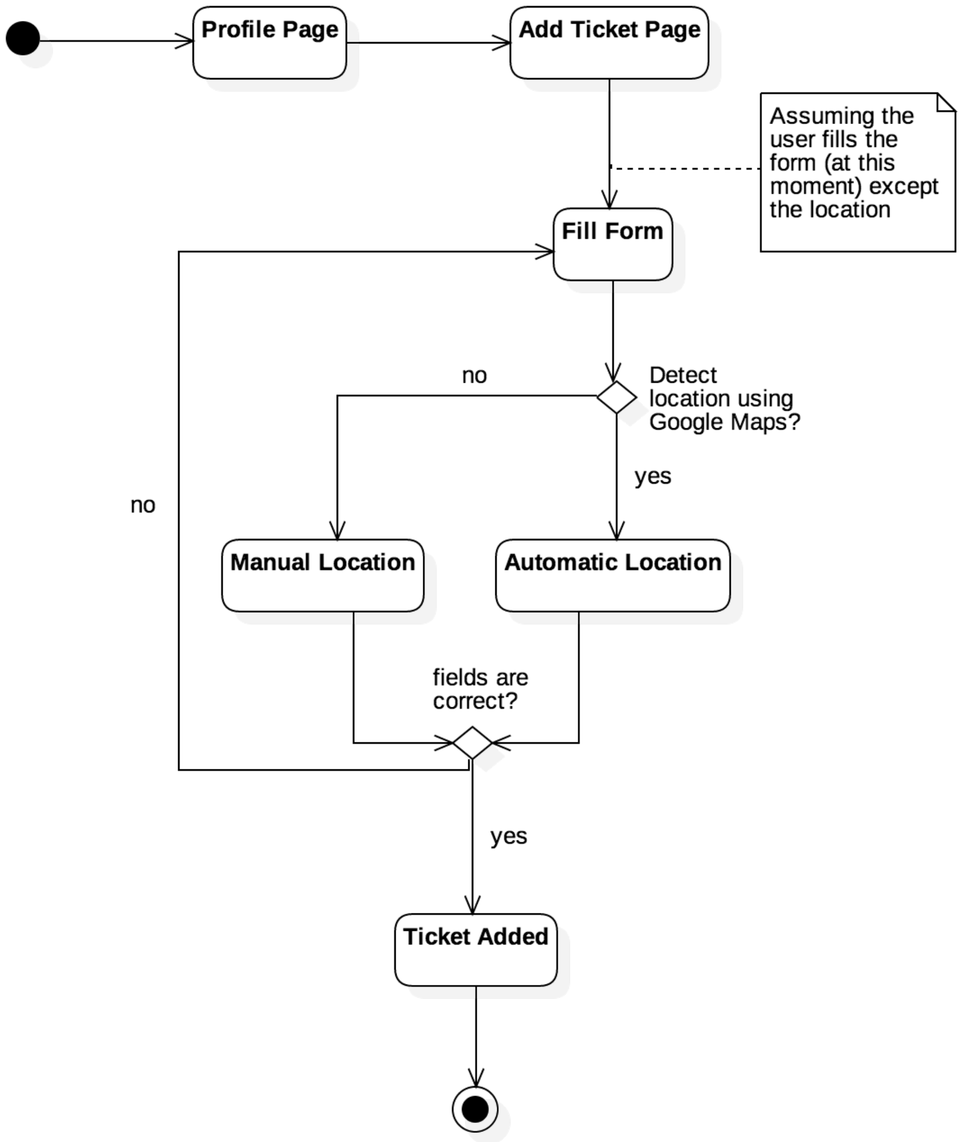


Figure 8: Add Ticket Scenario

Scenario 4: Edit Ticket

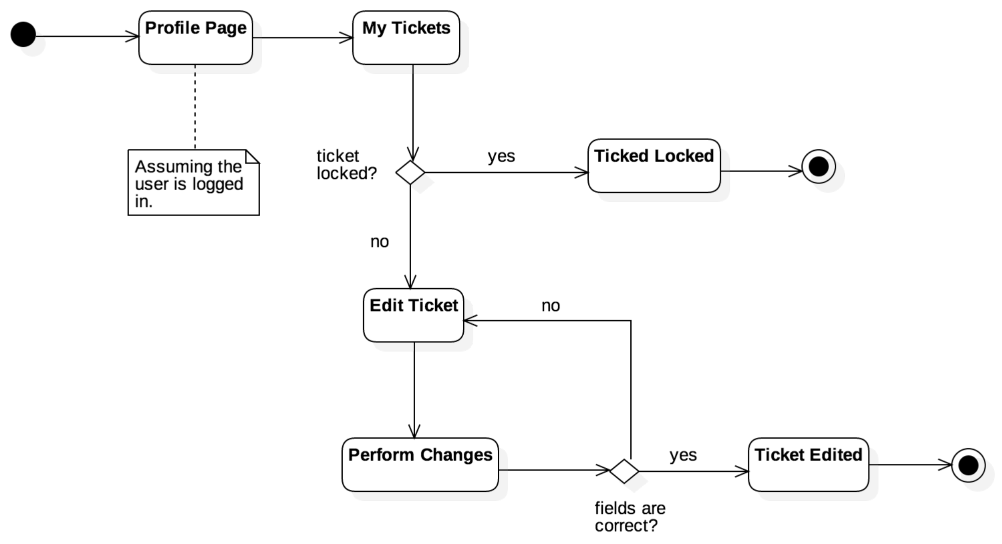


Figure 9: Edit Ticket Scenario

Scenario 5: Password Recovery

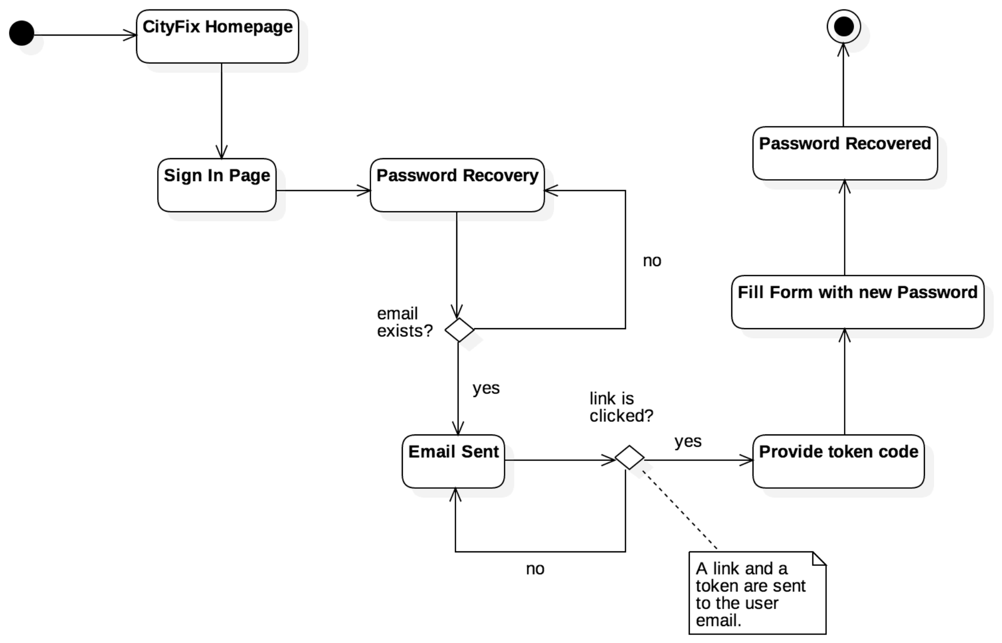


Figure 10: Password Recovery Scenario

Scenario 6: Suggest Ticket

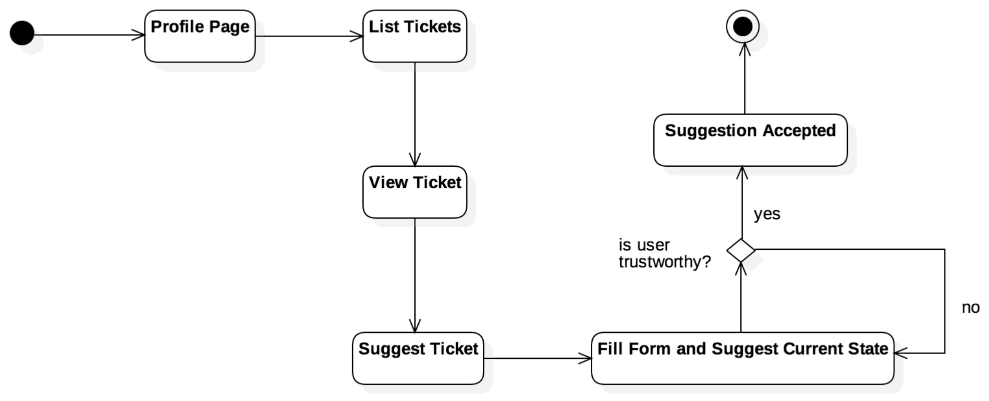


Figure 11: Suggest Ticket Scenario

Scenario 7: Vote

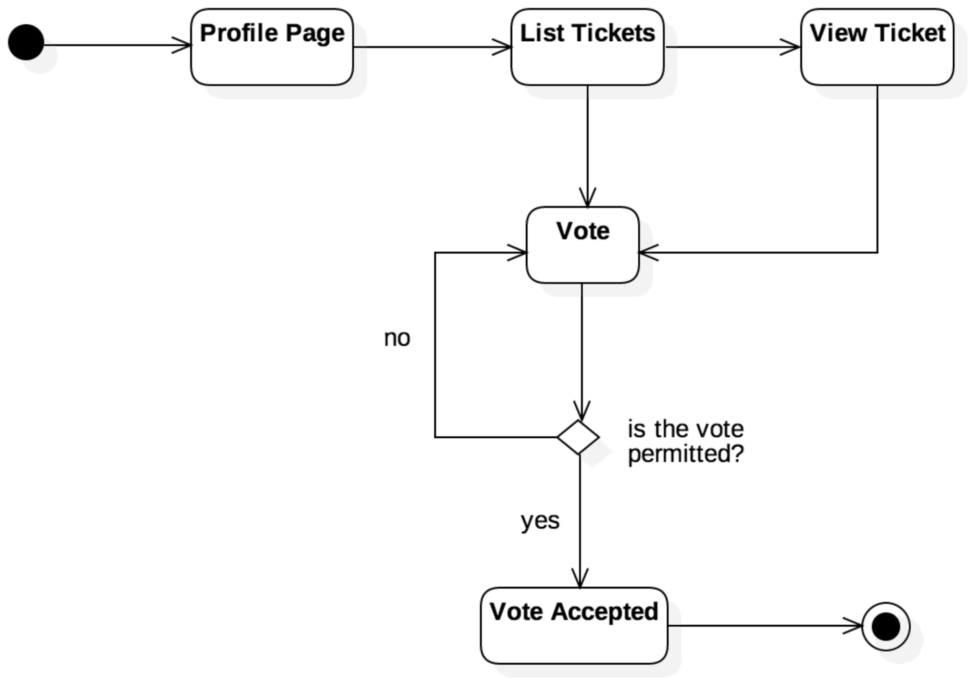


Figure 12: Vote Scenario

Scenario 8: Remove Ticket

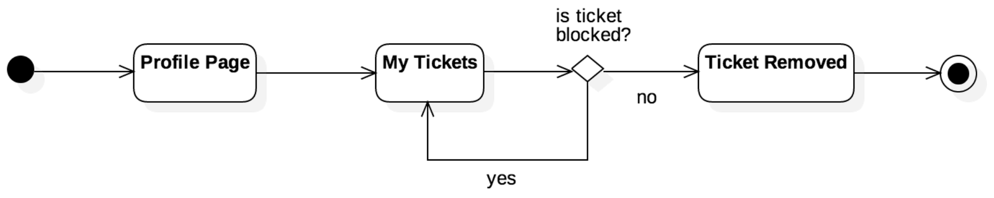


Figure 13: Vote Scenario

Scenario 9: Search

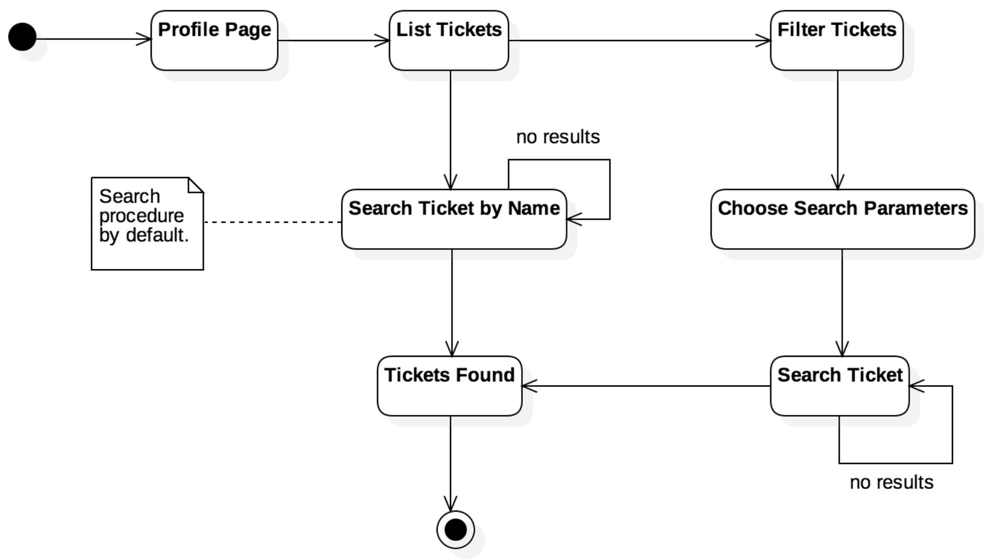


Figure 14: Search Scenario

Scenario 10: Entity Tickets Management

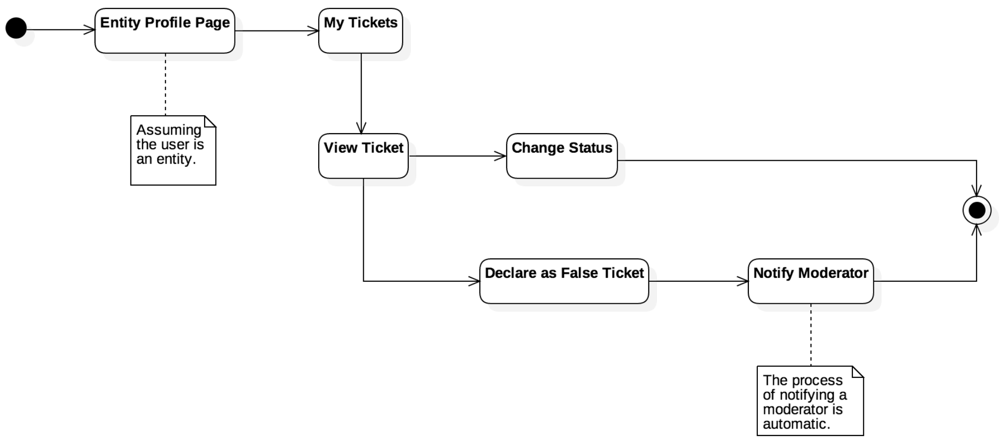


Figure 15: Entity Tickets Management Scenario

Scenario 11: Routes

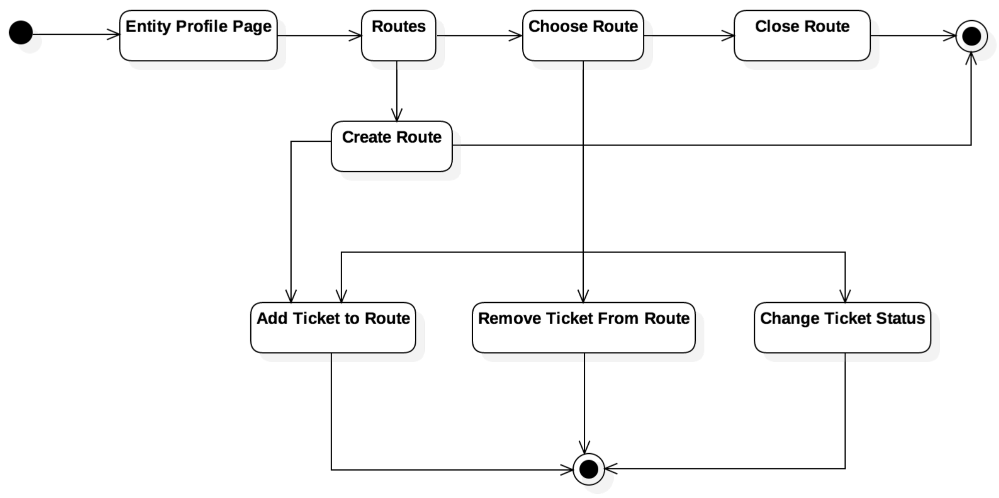


Figure 16: Routes Scenario

5. CONCEPTUAL MODEL

On this section it's presented the conceptual model for the project CityFix, as well as additional notes and constraints concerning the diagram.

5.1 UML Class Diagram

Conceptual modeling is the abstraction of a simulation model from the part of the real world it is representing ("the real system") [Robinson08]. After collecting the User Stories and all the necessary requirements, we achieved the following conceptual model, represented by Figure 17.

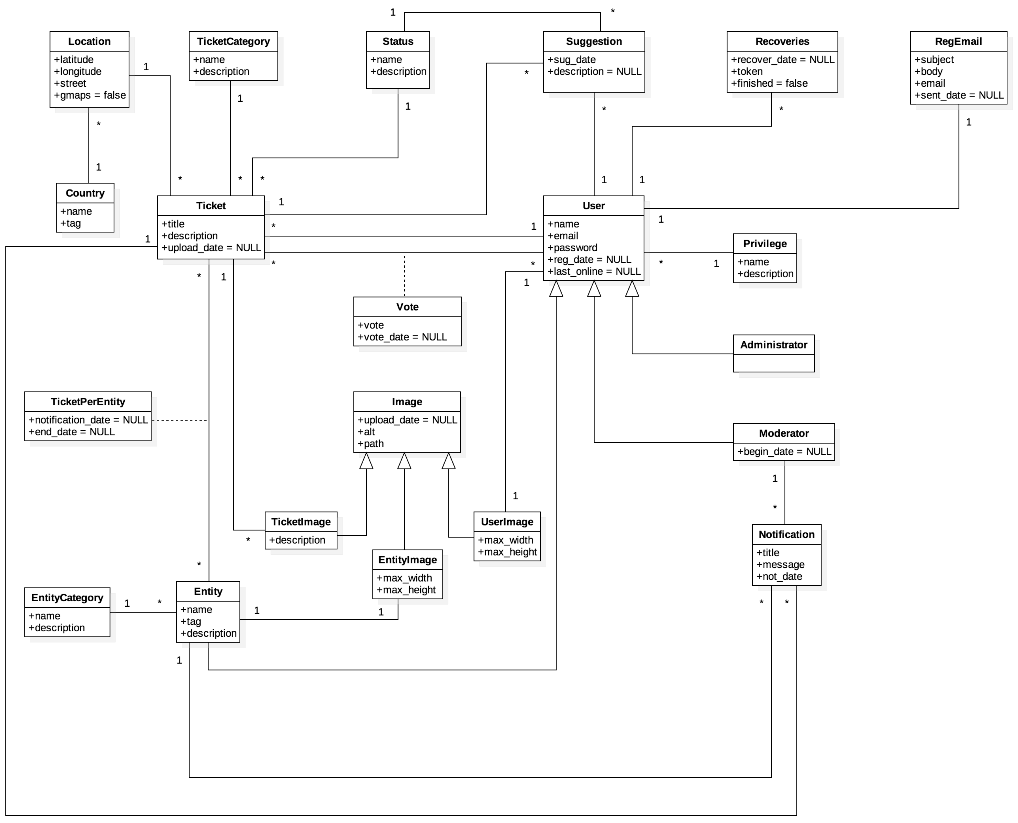


Figure 17: Conceptual Model

6. RELATIONAL SCHEMA

On this section it's presented the Relational Schema obtained from the Conceptual Model designed on the previous section, the normalization study for all relation schemas and the schema tuning.

The relational schema may be presented through a Physical Data Model (PDM or Physical Data Model) using UML notation [Ambler13], or otherwise (e.g. SQL), to make clear the attributes' domains and the tuple CHECKs. The relational schema includes the relation schemas, attributes, domains, primary keys, foreign keys and other integrity rules: UNIQUE, DEFAULT, NOT NULL, CHECK.

Relational schemas can also be specified using the following compact notation:

Notation
Table1(<u>id</u>, attribute NOT NULL)
[illegible]

Table2(<u>id</u>, attribute → Table1 NOT NULL)
Table3(<u>id1</u>, id2 → Table2, attribute UNIQUE NOT NULL)
Table4((<u>id1</u>, <u>id2</u>) → Table3, id3, attribute)

Table 2: Relational Schema compact notation

6.1 Physical Data Model

On Figure 18, it's presented the Relational Schema through a Physical Data Model obtained from the conceptual model.

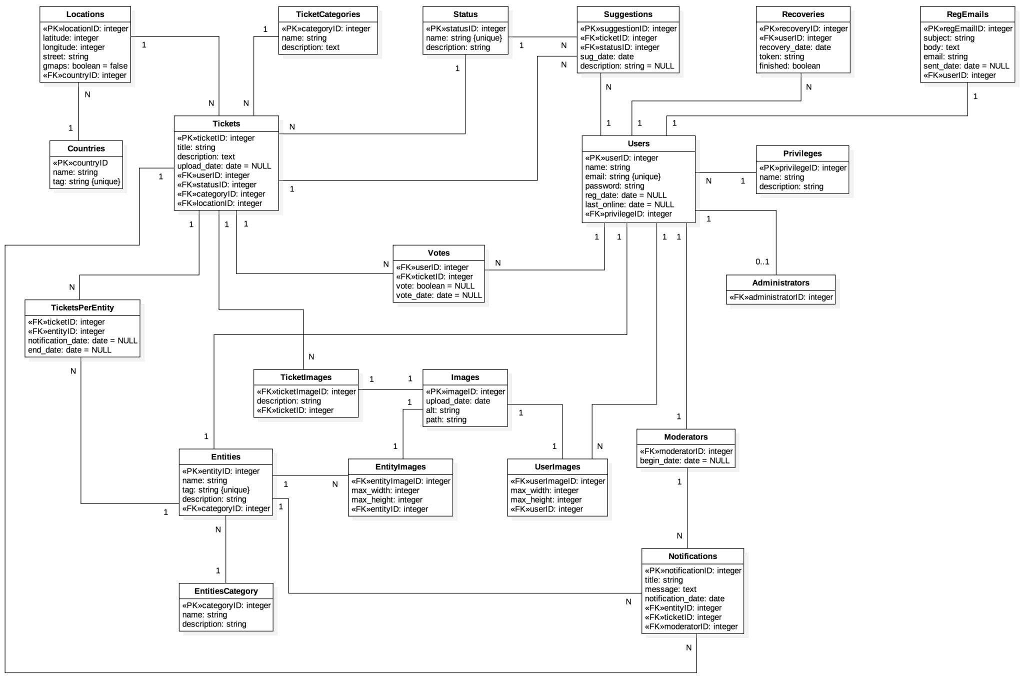


Figure 18: Physical Data Model

Below it's presented Relational Schema according to compact notation used in Table 1 (with the respective domains).

Relations
Administrators(<u>administratorID</u>: INTEGER → Users)
Countries(<u>countryID</u>: INTEGER, name: STRING, tag: STRING)
Entities(<u>entityID</u>: INTEGER, name: STRING, tag: STRING, description: STRING, categoryID: INTEGER → EntitiesCategories)
EntitiesCategories(<u>categoryID</u>: INTEGER, name: STRING, description: STRING)

EntitiesImages(entityImageID: INTEGER, max_width: INTEGER, max_height: INTEGER, entityID: INTEGER → Entities)
Images(imageID: INTEGER, upload_date: DATE, alt: STRING, path: STRING)
Locations(locationID: INTEGER, latitude: INTEGER, longitude: INTEGER, street: STRING, gmaps: FALSE, countryID: INTEGER → Countries)
Moderators(moderatorID: INTEGER, begin_date: DATE)
Notifications(notificationID: INTEGER, title: STRING, message: STRING, notification_date: DATE, entityID: INTEGER → Entities, ticketID: INTEGER → Tickets, moderatorID: INTEGER → Moderators)
Privileges(privilegeID: INTEGER, name: STRING, description: STRING)
Recoveries(recoveryID: INTEGER, userID: INTEGER → Users, recovery_date: DATE, token: STRING, finished: BOOLEAN)
RegEmails(regEmailID: INTEGER, subject: STRING, body: TEXT, email: STRING, sent_date: date, userID: INTEGER → Users)
Status(statusID: INTEGER, name: STRING, description: STRING)
Suggestions(suggestionID, ticketID: INTEGER → Tickets, statusID: INTEGER → Status, sug_date: DATE, description: STRING)
Tickets(ticketID: INTEGER, title: STRING, description: TEXT, upload_date: DATE, userID: INTEGER → Users, statusID: INTEGER → Status, categoryID: INTEGER → TicketCategories, locationID: INTEGER → Locations)
TicketsCategories(categoryID: INTEGER, name: STRING, description: STRING)
TicketsImages(ticketImageID: INTEGER → Images, description: STRING, ticketID: INTEGER → Tickets)
TicketsPerEntity(ticketID: INTEGER → Tickets, entityID: INTEGER → Entities, notification_date: DATE, end_date: DATE)
Users(userID: INTEGER, name: STRING, username: USERNAME, email: STRING, password: PASSWORD, reg_date: DATE, last_online: DATE, privilegeID: INTEGER → Privileges)
UsersImages(userImageID: INTEGER, max_width: INTEGER, max_height: INTEGER, userID: INTEGER → Users)
Votes(userID: INTEGER → Users, ticketID: INTEGER → Tickets, vote: BOOLEAN, vote_date: DATE)

Table 3: CityFix Relational Schema compact notation

6.2 Attributes Domains

Name	**Description**
FALSE	BOOLEAN DEFAULT false
GENDER	CHAR ‘F’ OR ‘M’
PASSWORD	STRING with at most 60 characters

TRUE	BOOLEAN DEFAULT true
USERNAME	STRING between 5 and 20 characters

Table 4: Attributes Domains

6.3 Tuple CHECKs

Images

```
CREATE TABLE Images(
    ...
    upload_date: DATE NOT NULL,
    alt: STRING,
    ...
    CHECK (upload_date = CURRENT_DATE)
);
```

6.4 Normalization

Data normalization is a process in which data attributes within a data model are organized to increase the cohesion of entity types. In other words, the goal of data normalization is to reduce and even eliminate data redundancy [Ambler13].

The relational schema is in BCNF (Boyce-Codd Normal Form).

7. DATABASE POPULATION

This section includes the complete script to create the database, including all the necessary SQL to the definition of all constraints, indexes, and triggers. It also includes the SQL for populating and testing the database with plausible values for the attributes.

7.1 Database and Relations Creation

7.1.1 Tables

Administrators

```
CREATE TABLE Administrators (
    administratorID INTEGER,
    CONSTRAINT admin_pk PRIMARY KEY (administratorID),
    CONSTRAINT admin_user_fk FOREIGN KEY (administratorID)
    REFERENCES Users (userID) ON UPDATE CASCADE ON DELETE CASCADE
);
```

Countries

```
CREATE TABLE Countries (
    countryID INTEGER,
    name STRING,
    tag STRING,
    CONSTRAINT country_pk PRIMARY KEY (countryID),
    CONSTRAINT tag_u UNIQUE (tag)
);
```

Entities

```
CREATE TABLE Entities (
    entityID INTEGER,
    name STRING,
    tag STRING,
    description TEXT,
    categoryID INTEGER,
    CONSTRAINT entity_pk PRIMARY KEY (entityID),
    CONSTRAINT category_fk FOREIGN KEY (categoryID)
    REFERENCES Categories (categoryID) ON UPDATE CASCADE ON DELETE
CASCADE
);
```

EntitiesCategories

```
CREATE TABLE EntitiesCategories (
    categoryID INTEGER,
    name STRING,
    description STRING,
    CONSTRAINT category_pk PRIMARY KEY (categoryID)
);
```

EntitiesImages

```
CREATE TABLE EntitiesImages (
   entityImageID INTEGER,
   max_width INTEGER,
   max_height INTEGER,
   entityID INTEGER,
   CONSTRAINT entity_image_pk PRIMARY KEY (entityImageID),
   CONSTRAINT entity_image_fk FOREIGN KEY (entityImageID)
   REFERENCES Images (imageID) ON UPDATE CASCADE ON DELETE CASCADE,
   CONSTRAINT entity_fk FOREIGN KEY (entityID)
   REFERENCES Entities (entityID) ON UPDATE CASCADE ON DELETE
CASCADE
);
```

Images

```
CREATE TABLE Images (
   imageID INTEGER,
   upload_date DATE,
   alt STRING,
   url STRING,
   CONSTRAINT image_pk PRIMARY KEY (imageID)
   REFERENCES Images (imageID) ON UPDATE CASCADE ON DELETE CASCADE
);
```

Locations

```
CREATE TABLE Locations (
   locationID INTEGER,
   latitude INTEGER,
   longitude INTEGER,
   street STRING,
   gmaps FALSE,
   countryID INTEGER,
   CONSTRAINT location_pk PRIMARY KEY (locationID),
   CONSTRAINT country_fk FOREGIN KEY (countryID)
   REFERENCES Countries (countryID) ON UPDATE CASCADE ON DELETE
CASCADE
);
```

Moderators

```
CREATE TABLE Moderators (
   moderatorID INTEGER,
   begin_date DATE,
   CONSTRAINT moderator_pk PRIMARY KEY (moderatorID),
   CONSTRAINT moderator_fk FOREIGN KEY (moderatorID)
   REFERENCES Users (userID) ON UPDATE CASCADE ON DELETE CASCADE
);
```

Notifications

```
CREATE TABLE Notifications (
   notificationID INTEGER,
   title STRING,
   message STRING,
   notification_date DATE,
   entityID INTEGER,
   ticketID INTEGER,
   moderatorID INTEGER,
   CONSTRAINT notification_pk PRIMARY KEY (notificationID),
   CONSTRAINT entity_fk FOREIGN KEY (entityID)
   REFERENCES Entities (entityID) ON UPDATE CASCADE ON DELETE
CASCADE,
   CONSTRAINT ticket_fk FOREIGN KEY (ticketID)
   REFERENCES Tickets (ticketID) ON UPDATE CASCADE ON DELETE
CASCADE,
   CONSTRAINT moderator_fk FOREIGN KEY (moderatorID)
   REFERENCES Moderators (moderatorID) ON UPDATE CASCADE ON DELETE
CASCADE
);
```

Privileges

```
CREATE TABLE Privileges (
   privilegeID INTEGER,
   name STRING,
   description STRING,
   CONSTRAINT privilege_pk PRIMARY KEY (privilegeID)
);
```

Recoveries

```
CREATE TABLE Recoveries (
   recoveryID INTEGER,
   userID INTEGER,
   recovery_date DATE,
   token STRING,
   finished BOOLEAN,
   CONSTRAINT recovery_pk PRIMARY KEY (recoveryID),
   CONSTRAINT user_fk FOREIGN KEY (userID)
   REFERENCES Users (userID) ON UPDATE CASCADE ON DELETE CASCADE
);
```

RegEmails

```
CREATE TABLE RegEmails (
   regEmailID INTEGER,
   subject STRING,
   body TEXT,
   email STRING,
   send_date DATE,
   userID INTEGER,
   CONSTRAINT reg_email_pk PRIMARY KEY (regEmailID),
   CONSTRAINT user_fk FOREIGN KEY (userID)
   REFERENCES Users (userID) ON UPDATE CASCADE ON DELETE CASCADE
);
```

Status

```
CREATE TABLE Status (
   statusID INTEGER,
   name STRING,
   description STRING,
   CONSTRAINT status_pk PRIMARY KEY (statusID)
);
```

Suggestions

```
CREATE TABLE Suggestions (
   suggestionID INTEGER,
   ticketID INTEGER,
   statusID INTEGER,
   sug_date DATE,
   description STRING,
   CONSTRAINT suggestion_pk PRIMARY KEY (suggestionID),
   CONSTRAINT ticket_fk FOREIGN KEY (ticketID)
   REFERENCES Tickets (ticketID) ON UPDATE CASCADE ON DELETE
CASCADE,
   CONSTRAINT status_fk FOREIGN KEY (statusID)
   REFERENCES Status (statusID) ON UPDATE CASCADE ON DELETE SET
NULL
);
```

Tickets

```
CREATE TABLE Tickets (
   ticketID INTEGER,
   title STRING,
   description TEXT,
   upload_date DATE,
   userID INTEGER,
   statusID INTEGER,
   categoryID INTEGER,
   locationID INTEGER,
   CONSTRAINT ticket_pk PRIMARY KEY (ticketID),
   CONSTRAINT user_fk FOREIGN KEY (userID)
   REFERENCES Users (userID) ON UPDATE CASCADE ON DELETE CASCADE,
   CONSTRAINT status_fk FOREIGN KEY (statusID)
   REFERENCES Status (statusID) ON UPDATE CASCADE ON DELETE SET
NULL,
   CONSTRAINT category_fk FOREIGN KEY (categoryID)
   REFERENCES TicketsCategories (categoryID) ON UPDATE CASCADE ON
DELETE SET NULL,
   CONSTRAINT location_fk FOREIGN KEY (locationID)
   REFERENCES Locations (locationID) ON UPDATE CASCADE ON DELETE
SET NULL
);
```

TicketsCategories

```
CREATE TABLE TicketsCategories (
   categoryID INTEGER,
   name STRING,
   description STRING,
   CONSTRAINT category_pk PRIMARY KEY (categoryID)
);
```

TicketsImages

```
CREATE TABLE TicketsImages (
   ticketImageID INTEGER,
   description STRING,
   ticketID INTEGER,
   CONSTRAINT ticket_image_pk PRIMARY KEY (ticketImageID),
   CONSTRAINT ticket_image_fk FOREIGN KEY (ticketImageID)
   REFERENCES Images (imageID) ON UPDATE CASCADE ON DELETE CASCADE,
   CONSTRAINT ticket_fk FOREIGN KEY (ticketID)
   REFERENCES Tickets (ticketID) ON UPDATE CASCADE ON DELETE
CASCADE
);
```

TicketsPerEntity

```
CREATE TABLE TicketsPerEntity (
   ticketID INTEGER,
   entityID INTEGER,
   notification_date DATE,
   end_date DATE,
   CONSTRAINT ticket_entity_pk PRIMARY KEY (ticketID, entityID),
   CONSTRAINT ticket_fk FOREIGN KEY (tickerID)
   REFERENCES Tickets (ticketID) ON UPDATE CASCADE ON DELETE
CASCADE,
   CONSTRAINT entity_fk FOREIGN KEY (entityID)
   REFERENCES Entities (entityID) ON UPDATE CASCADE ON DELETE
CASCADE
);
```

Users

```
CREATE TABLE Users (
   userID INTEGER,
   name STRING,
   username STRING,
   email STRING,
   password PASSWORD,
   reg_date DATE,
   last_online DATE,
   privilegeID INTEGER,
   CONSTRAINT user_pk PRIMARY KEY (userID),
   CONSTRAINT privilege_fk FOREIGN KEY (privilegeID)
   REFERENCES Privileges (privilegeID) ON UPDATE CASCADE ON DELETE
CASCADE
);
```

UsersImages

```
CREATE TABLE UsersImages (
   userImageID INTEGER,
   max_width INTEGER,
   max_height INTEGER,
   userID INTEGER,
   CONSTRAINT user_image_pk PRIMARY KEY (userImageID),
   CONSTRAINT user_image_fk FOREIGN KEY (userImageID)
   REFERENCES Images (imageID) ON UPDATE CASCADE ON DELETE CASCADE,
   CONSTRAINT user_fk FOREIGN KEY (userID)
   REFERENCES Users (userID) ON UPDATE CASCADE ON DELETE CASCADE
);
```

Votes

```
CREATE TABLE Votes (
   userID INTEGER,
   ticketID INTEGER,
   vote BOOLEAN,
   vote_date DATE,
   CONSTRAINT user_pk PRIMARY KEY (userID),
   CONSTRAINT ticket_fk FOREIGN KEY (ticketID)
   REFERENCES Tickets (ticketID) ON UPDATE CASCADE ON DELETE
CASCADE
);
```

8. High Level Architecture and Web Resources Specification

On this section it's presented an overview of the web resources which are organized in modules. The access permissions to those modules (user privileges) and the supported HTTP methods and corresponding JSON responses (if exist) are also described.

8.1 Architecture Overview

On this section it's provided a list of the necessary modules and a description for each one of them.

Module	Description
M01: Authentication and Profile	Web resources associated with authentication and profile management, which includes the following system features: login / logout, recovery credentials, consult and edition of profile information.
M02: Registration	Web resources associated with the users registration via Facebook (using features from **Module M08**) or by creating an account on CityFix.
M03: Ticketing Management	Web resources associated with ticketing management (tickets uploading, updating and deletion, change current state, etc.).
M04: Ticketing Voting and Validation	Web resources associated with the tickets voting system and validation, such as: forwarding the ticket to the corresponding entities, increase or decrease the ticket's owner reputation depending on the ticket's content (appropriate or inappropriate).
M05: Users Administration	Web resources associated with the users administration, including: consultation and search of users, removal or blocking accounts and access details.
M06: Entities	Web resources associated with problems solving by entities, such as: receiving notifications of new tickets and manage them (change their current state, close the ticket, etc.).
M07: Google Maps API Integration	Web resources associated with the detection of the user's current location when uploading a ticket.
M08: Facebook API Integration	Web resources associated with the authentication (**Module M01**) or users

	registration (**Module M02**) via Facebook.
M09: Search	Web resources associated with searching public information, such as users, entities or tickets.

Table 5: Modules

8.2 Privileges

On this section is defined the permissions used in the modules to establish the conditions of access to resources.

Identifier	Name	Description
PUB	Public	Unprivileged user group
OWN	Owner	Own user (owner or same user)
MOD	Moderator	Moderator's group
ENT	Entity	Entity's group
ADM	Administrator	Administrators group

Table 6: Privileges

8.3 Modules

On this section is presented each of the features for all modules (the URL, HTTP methods, possible parameters, user interfaces - or JSON responses in a usage of APIs scenarios).

M01: Authentication and Profile

R01: LOGIN

Title	Description
URL	`/app/view/login`
Description	This web resource corresponds to a view. The user fills the login form and, if success, receives a response and then redirects to the user profile. If an error occurs the user receives an error message and continues on the login page.
Method	POST
Request Body	+username: String
	?password: String
Response Body	JSON101
Permissions	OWN

R02: LOGOUT ACTION

Title	Description
URL	`/app/action/logout`
Description	This web resource corresponds to an action. The user clicks on the logout button and, if success, redirects to the main page. If an error occurs the user is redirected to the tickets page with an error message.
Method	POST
Request Body	+logout: String
Redirects	… (success)
	… (error)
Permissions	OWN

R03: RECOVER CREDENTIALS ACTION

Title	Description
URL	`/app/action/recover_credentials`
Description	This web resource corresponds to an action, i.e perform changes to the information system and in the end there is a redirection to another web resource. In this case, this web feature sends an email with a link to the specified user with the procedures to recover its authentication credentials. After the email is sent, the user is redirected to the reset password page.
Method	POST
Request Body	+email: String
Redirects	… (success)
	… (error)
Permissions	OWN

R04: RESET PASSWORD

Title	Description
URL	`/app/view/reset_password`
Description	This web resource corresponds to a view. On this view is displayed a form where the user can type his new password (after clicking on the sent link by email).
Method	GET

Request Body	+token: String
Permissions	OWN

R05: RESET PASSWORD ACTION

Title	**Description**
URL	`/app/view/reset_password`
Description	This web resource corresponds to an action, i.e perform changes to the information system and in the end there is a redirection to another web resource. In this case, this web feature resets the password of a certain user.
Method	POST
Request Body	+token: String
	?password: String
Redirects	... (success)
	R04 (error)
Permissions	OWN

R06: EDIT PROFILE ACTION

Title	**Description**
URL	`/app/action/edit_profile`
Description	This web resource corresponds to an action, i.e perform changes to the information system and in the end there is a redirection to another web resource. In this case, this web feature changes the information about an user and, if successful, redirects the user to this own profile or, in case of failure, to the user edition form.
Method	POST
Request Body	+userID: Integer
	?name: String
	?email: String
Redirects	... (success)
	... (error)
Permissions	OWN

M02: Registration

R07: SIGN UP FORM

Title	Description
URL	`/app/view/sign_up`
Description	This web resource corresponds to a view. On this view is displayed a form where the user can register in the system.
Method	GET
Permissions	OWN

R08: SIGN UP ACTION

Title	Description
URL	`/app/action/edit_profile`
Description	This web resource corresponds to an action, i.e perform changes to the information system and in the end there is a redirection to another web resource. In this case, this web feature register a user in the system and, if successful, redirects the user to the Sign In page or, in case of failure, to the Sign Up form.
Method	POST
Request Body	+username: String
	?password: String
	?name: String
	?email: String
Redirects	R01 (success)
	R08 (error)
Permissions	OWN

R09: SIGN UP WITH FACEBOOK ACTION

Title	Description
URL	`/app/action/sign_up_fb`
Description	This web resource corresponds to an action, i.e perform changes to the information system and in the end there is a redirection to another web resource. In this case, this web feature enables the user to sign up using a Facebook account.
Method	POST
Request Body	...: ...

Redirects	… (success)
	… (error)
Permissions	OWN

M03: Ticketing Management

R10: LIST TICKET(S)

Title	**Description**
URL	`/app/view/tickets`
Description	This web resource corresponds to a view. On this view is displayed a list of tickets and when a user clicks in one of them, it pops up a window with the corresponding ticket information. If the request is specified a **ticketID** then that ticket information will appear. Otherwise it will be a list of tickets.
Method	GET
Parameters	+ticketID: Integer (optional)
AJAX Calls	…
UI	…
Permissions	PUB (limited info) or OWN (detailed info)

R11: ADD TICKET

Title	**Description**
URL	`/app/view/add_ticket`
Description	This web resource corresponds to a view. On this view is displayed a form where the user can add a ticket to the system.
Method	GET
Permissions	OWN

R12: ADD TICKET ACTION

Title	**Description**
URL	`/app/action/add_ticket`
Description	This web resource corresponds to an action, i.e perform changes to the information system and in the end there is a redirection to another web resource. In this case, this web feature registers a new ticket in the system.
Method	POST

Request Body	title: String
	categoryID: Integer
	description: Text
	entitiesIDs: Array
	statusID: Integer
	location: String
	images: Array
Redirects	… (success)
	… (error)
Permissions	OWN

R13: EDIT TICKET

Title	**Description**
URL	`/app/view/edit_ticket`
Description	This web resource corresponds to a view. On this view is displayed a form where the user can edit a ticket.
Method	GET
Parameters	+ticketID: Integer (optional)
Permissions	OWN

R14: EDIT TICKET ACTION

Title	**Description**
URL	`/app/action/edit_ticket`
Description	This web resource corresponds to an action, i.e perform changes to the information system and in the end there is a redirection to another web resource. In this case, this web feature updates a ticket.
Method	POST
Request Body	title: String
	categoryID: Integer
	description: Text
	entitiesIDs: Array
	statusID: Integer
	location: String

	images: Array
Redirects	… (success)
	… (error)
Permissions	OWN

R15: REMOVE TICKET ACTION

Title	**Description**
URL	`/app/action/remove_ticket`
Description	This web resource corresponds to an action, i.e perform changes to the information system and in the end there is a redirection to another web resource. In this case it removes a ticket from the system and the response is represented using HTTP codes.
Method	POST
Request Body	ticketID: Integer
Returns	200 OK
	404 Not Found
Permissions	OWN

R16: MY TICKETS

Title	**Description**
URL	`/app/view/my_tickets`
Description	This web resource corresponds to a view. On this view is displayed a list of tickets which belong to the logged user.
Method	GET
Permissions	OWN

M04: Ticketing Voting and Validation

R17: LIST TICKETS (MODERATOR VIEW)

Title	**Description**
URL	`/app/view/tickets_mod`
Description	This web resource corresponds to a view. On this view is displayed a list of tickets for the moderator to validate (if necessary) and increase or decrease users reputation.
Method	GET

Permissions	MOD

R18: VOTE ON TICKET ACTION

Title	Description
URL	`/app/action/vote_ticket`
Description	This web resource corresponds to an action, i.e perform changes to the information system. In this case, this web feature enables the user to vote on a ticket.
Method	POST
Request Body	ticketID: Integer
	userID: Integer
	option: Integer
Permissions	OWN

R19: VALIDATE TICKET ACTION

Title	Description
URL	`/app/action/validate_ticket`
Description	This web resource corresponds to an action, i.e perform changes to the information system. In this case, this web feature enables the moderator to validate a ticket (when some doubts arise).
Method	POST
Request Body	ticketID: Integer
Permissions	MOD

R20: INCREASE / DECREASE REPUTATION ACTION

Title	Description
URL	`/app/action/reputation`
Description	This web resource corresponds to an action, i.e perform changes to the information system. In this case, this web feature enables the moderator increase or decrease an user's reputation based on his published tickets (appropriate or inappropriate).
Method	POST
Request Body	userID: Integer
	points: Integer

Permissions	MOD

M05: Users Administration

R21: LIST USERS

Title	**Description**
URL	`/app/view/users`
Description	This web resource corresponds to a view. On this view is displayed the list of users and their relevant information. This view may be different depending on the user privileges.
Method	GET
Permissions	ADM, OWN, PUB

R22: VIEW USER

Title	**Description**
URL	`/app/view/user`
Description	This web resource corresponds to a view. On this view is displayed the information concerning a certain user.
Method	GET
Parameters	+userID: Integer
Permissions	ADM, OWN, PUB

R23: REMOVE USER

Title	**Description**
URL	`/app/action/remove_user`
Description	This web resource corresponds to an action, i.e perform changes to the information system and in the end there is a redirection to another web resource. In this case it removes an user from the system and the response is represented using HTTP codes.
Method	POST
Request Body	userID: Integer
Returns	200 OK
	404 Not Found
Permissions	ADM

M06: Entities

R24: LIST ENTITIES

Title	Description
URL	`/app/view/entities`
Description	This web resource corresponds to a view. On this view is displayed the list of entities and their relevant information.
Method	GET
Permissions	OWN, PUB

R25: EDIT ENTITY FORM

Title	Description
URL	`/app/view/entities`
Description	This web resource corresponds to a view. On this view is displayed the edition form for a certain entity and its corresponding information.
Method	GET
Parameters	entityID: Integer
Permissions	ENT

R26: EDIT ENTITY ACTION

Title	Description
URL	`/app/action/edit_entity`
Description	This web resource corresponds to an action, i.e perform changes to the information system and in the end there is a redirection to another web resource. In this case, this web feature changes the information about an entity and, if successful, redirects to the entity profile or, in case of failure, to the edit entity form.
Method	POST
Request Body	+entityID: Integer
	?name: String
	?categoryID: Integer
	?photo: File
Redirects	… (success)
	… (error)
Permissions	ENT

R27: REMOVE ENTITY ACTION

Title	Description
URL	`/app/action/remove_entity`
Description	This web resource corresponds to an action, i.e perform changes to the information system and in the end there is a redirection to another web resource. In this case, this web feature removes an entity from the system.
Method	POST
Request Body	+entityID: Integer
Redirects	… (success)
	… (error)
Permissions	ADM

M07: Google Maps API Integration

R28: GOOGLE MAPS LOCATION ACTION

Title	Description
URL	`/app/action/gmaps`
Description	This web resource corresponds to an action. This allows the system to get information about a certain location using Google Maps API.
Method	GET
Parameters	+location: String
Permissions	OWN

M08: Facebook API Integration

R29: FACEBOOK GET USER DATA ACTION

Title	Description
URL	`/app/api/fb_register`
Description	This web resource corresponds to an action, i.e perform changes to the information system and in the end there is a redirection to another web resource. In this case, this web feature gets relevant data about an user using his Facebook account.
Method	POST
Request Body	+fbSessionID: Integer
Redirects	… (success)

	... (error)
Permissions	OWN

M09: Search

R30: SEARCH TICKETS

Title	**Description**
URL	`/app/api/search_tickets`
Description	This web resource allows the user to search for tickets.
Method	GET
Parameters	+ticket_title: String
Response Body	JSON901
Permissions	PUB, OWN

R31: SEARCH USERS

Title	**Description**
URL	`/app/api/search_users`
Description	This web resource allows an user to search for other users.
Method	GET
Parameters	+username: String
Response Body	JSON902
Permissions	PUB, OWN

R32: SEARCH ENTITIES

Title	**Description**
URL	`/app/api/search_entities`
Description	This web resource allows an user to search for entities.
Method	GET
Parameters	+entity_name: String
Response Body	JSON903
Permissions	PUB, OWN

8.4 JSON/XML Types

JSON101: Login: {result}[]

```
{
   "result": [
      {
         "value": "true",
         "url": "app/view/profile",
      }
   ]
}
```

JSON901: Search Tickets: {tickets}[]

```
{
   "tickets": [
      {
         "ticketID": 1,
         "title": "There's a hole in the way",
         "category": "Road Problems",
         "description": "Please cover this hole.",
         "status": "Active",
         "location": "Disaster Street, 666",
         "user": "Peter Parker"
      }
   ]
}
```

JSON902: Search Users: {users}[]

```
{
   "users": [
      {
         "userID": 1,
         "username": "holebreaker",
         "name": "Peter Parker",
         "email": "peterparker@gmail.com",
         "User Type": "Moderator"
      }
   ]
}
```

JSON903: Search Entities: {entities}[]

```
{
   "entities": [
      {
         "entityID": 1,
         "name": "Hole Covers, Lda.",
         "category": "Road Problems",
         "email": "info@holecovers.com"
      }
   ]
}
```

9. PRODUCT PROMOTION AND DEMONSTRATION

This section includes the promotion, a presentation and demonstration of the developed product. For this project we made a flyer, a promo video and a live presentation for the Grand Finale.

9.1 Flyer

The flyer for CityFix is available on the following link.

9.2 Promo

The promo video is available on the following link.

9.3 Presentation

The live presentation is available on the following link.

A. USE CASES

Visitor

Identifier	Name	Priority	Description	Cost

US001	Login	High	As a Visitor I want to login into the system so that I can have access to restricted information	3
US002	Register	High	As a Visitor I want to register myself so that I can upload new tickets	5
US003	Password Recovery	Medium	As a Visitor I want to recover my authentication credentials so that I can access the platform if I forget my password or username	3
US004	List Tickets	Medium	As a Visitor, I want to see a list with a simple view of recent tickets so that I know the most recent activity	2
US005	Facebook Register	Low	As a Visitor, I want to register myself with my Facebook account so that my basic information is already filled automatically	8

Authenticated

Identifier	Name	Priority	Description	Cost
US101	Search	Medium	As an User I want to search all public information (entity profiles, etc.) so that I can know who I should contact if there is a problem near my location	5

US102	View Ticket	High	As an User I want to view a ticket's content so that I can know if there is a problem near my location and what actually happened	3
US103	Logout	High	As an Authenticated I want to be able to logout from the system so that I can terminate my session correctly	1
US104	Profile	Medium	As an Authenticated I want to see and edit my profile so that I can change my basic information and authentication credentials	3

User

Identifier	Name	Priority	Description	Cost
US201	Add Ticket	High	As an User I want to submit a ticket to the system so that I can show that there's a problem near my location	3
US202	Edit Ticket	High	As an User I want to edit a ticket i own in the system so that I can update its information	5
US203	Remove Ticket	High	As an User I want to remove a ticket I own so that I can hide its content from the public	2
US204	Associate Category	High	As an User I want to associate a ticket	1

			to a category so that it helps to solve a problem	
US205	Manual Location	Medium	As an User I want to manually add a location so that I can add a ticket after a situation when I didn't have internet connection	8
US206	Automatic Location	High	As an User I want the system to automatically detect my location so that I don't have to add it manually	8
US207	Vote	High	As an User I want to vote on a ticket so that I can contribute to the truthfulness of that ticket and its respective user	3
US208	List My Tickets	Medium	As an User I want to have access to a list of my submitted tickets so that I can keep track of their status	3
US209	Associate Image to Ticket	High	As an User I want to associate an image to a ticket so that an user or entity can easily detect what's happening on that location	5
US210	Ticket Suggestion	Medium	As an User I want to suggest changes to a ticket so that whenever I think an update must be, it shall be considered	5
US211	Ticket Search	High	As an User I want to search all public tickets so that I can see information on tickets others	3

			submitted	

Moderator

Identifier	Name	Priority	Description	Cost
US301	Validate Tickets	High	As a Moderator I want to validate tickets so that an entity can solve the problem	1
US102	View Ticket	Medium	As a Moderator I want to change information of a ticket so that I can update the current status, etc	5

Entity

Identifier	Name	Priority	Description	Cost
US401	Entity tickets	High	As an Entity I want to access the tickets associated with me so that I can take action on solving those problems	3
US402	Change Ticket State	High	As an Entity I want to change a ticket's state so that an active user may know the current state of that specific problem	1
US403	Filter Tickets	Low	As an Entity i want to filter tickets so that I can optimize my schedule	8

Administrator

Identifier	Name	Priority	Description	Cost
US501	Manage Users	High	As an Administrator I want to manage users so that I can delete or mute an user if necessary	3
US402	Manage Content	High	As an Administrator I want to manage the system's content so that whenever a change is necessary, there is permission to do so	5
US403	Statistics	Low	As an Administrator I want to get statistics so that I can know information about my system	8

B. SCRUM TEAM AND PO COMMUNICATION

Sidenote

Since our Product Owner belongs to the Scrum Team the meetings were far easier, however, far more demanding and in some cases quite confusing, i.e, we've debated too much about features and how to implement them. This created far more difficulties than to have a Product Owner who tells us what he really wants, which (perhaps) would facilitate the process instead of rambling on the idea for quite some time.

C01

Subjects

- Discuss the project current status;
- Discuss the current User Stories;
- Discuss the current software architecture;

Changes

Nothing to report.

Work Aspects

- Most of User Stories approved;
- System architecture disapproved;

Meeting Conclusions

- Some changes must be done to the User Stories;
- Still a lot to do connecting the different technologies;

C02

Subject(s)

- Discuss current User Stories;
- Discuss the system architecture;
- Discuss other specifications;

Changes

- User Stories changed since the last meeting;
- System architecture fully done (according to last meeting);

Work Aspects

- User Stories fully approved;
- System architecture fully approved;

Meeting Conclusions

- Features like Login and Logout are high priority features.
- Ruby and Rails integration must be done;
- Start testing technologies examples;
- Describe the possible modules and privileges based on the Actors;

C03

Subject(s)

- Discuss the current project status;
- Check if the Ruby and Rails integration was done;

Changes

- Team must start first with Ruby and Rails integration;

- After configuring Ruby and Rails, start implementing Login and Logout features;

Work Aspects
- Ruby and Rails integration disapproved;
- Login and Logout features disapproved;

Meeting Conclusions
- Ruby and Rails has a high priority status;
- Login and Logout features must be done after configuring Ruby and Rails correctly;
- Other specifications are on the right track;

C. SPRINT REVIEWS AND RETROSPECTIVES

Sidenote

Since our Product Owner is not actually a person outside the Scrum Team, we faced some difficulties during this iteration. One of them was to *figure out what we really wanted to do* - which took us a lot of time, hindering the prototype implementation phase. We focused so much on trying to understand how the system would work that we ended up on just documenting (like the Web Resources Specificiation on this wiki, using a more waterfall process), rather than implementing and obtaining results (iterative process). In other words, we were trying to give a big step, rather than giving *small steps*. Beyond that, we focused more on the technology, rather than the consumer and this led to delays on the implementation of the prototype. However, we strongly believe that the time we apparently "lost" during this iteration was a *big step* to work better as a team and deliver a better product on the next iteration.

SR01

Subject(s)
- Discuss aspects related with the system features;
- Discuss aspects related with the state of the art;
- Discuss aspects related with differ from other applications;
- Discuss other implementation aspects with the Product Owner;

What has been done

- User Stories and architecture;

What has been left to do
- State of the art;
- Differentiation from other apps;

What has been decided
- Search for other applications;
- Figure out a way to differ from other apps;

SR02

Subject(s)
- Discuss aspects related with the database;
- Discuss aspects related with Ruby on Rails configuration;
- Discuss aspects related with the documentation;

What has been done
- Simple database relations in Enterprise Architect;
- Layout design;

What has been left to do
- Ruby on Rails configuration;
- Layout design implementation details;

What has been decided
- Configure Ruby on Rails;
- Implement Login and Logout features after configurating Rails;

SR03

Subject(s)
- Discuss aspects related with Ruby on Rails configuration;
- User Stories implementation and distribution of work;
- Discuss aspects related with the documentation;
- Discuss technologies like AngularJS;

What has been done
- Ruby on Rails configuration;

- Ruby on Rails testing examples;

What has been left to do

- Ruby on Rails configuration;
- Login and Logout features due to some problems configurating Rails;

What has been decided

- Design the necessary database relations to ensure the intended US are implemented;
- Design the necessary layout;
- Implement Login and Logout features and, if possible, other important User Stories to the Product Owner;
- Document the necessary web resources;

SR04

Subject(s)

- Discuss the current work situation;
- User Stories implementation and distribution of work;
- Discuss technologies like AngularJS;

What has been done

- Description of some modules and privileges;
- Ruby on Rails configuration;
- Ruby on Rails testing examples;
- Ticket features implemented;
- User features implemented;
- Final product features validated by product owner;

What has been left to do

- Roles restrictions;
- Google maps integration;
- Administrative features implementation;

What has been decided

- Layout improvment for product presentation;
- Implementation of most important administrative feature;
- Implementation of indispensable basic features;

ROADMAP

On this section it's presented the Roadmap for this project. The following diagram summarizes the project's state as well as future development.

Roadmap Diagram

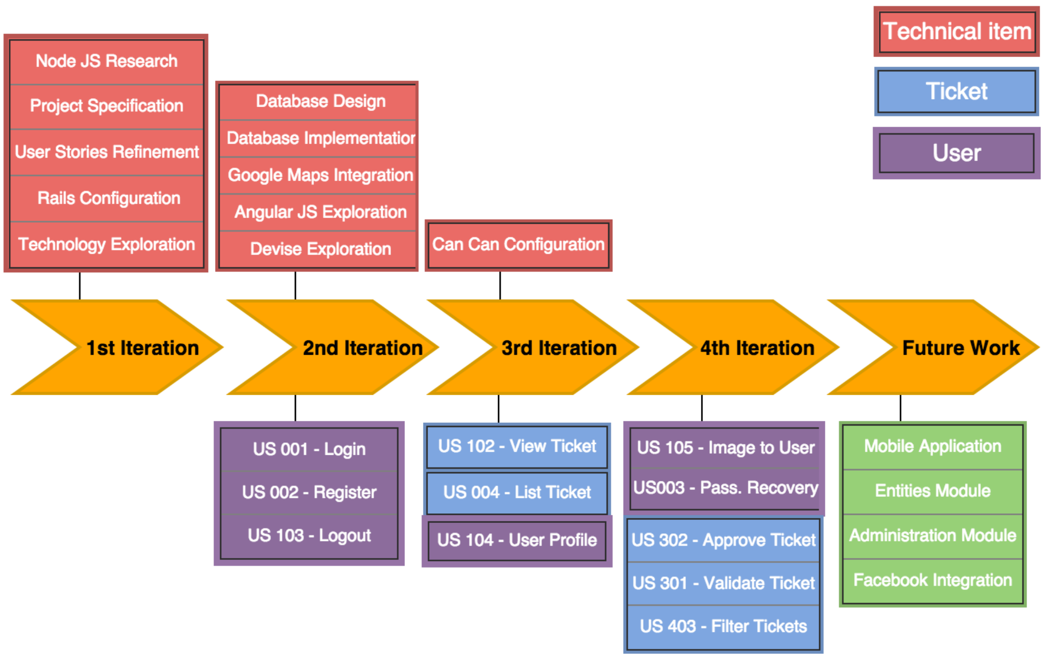


Figure 1: Roadmap Diagram

There are three types of relevant information:

- **Technical Items** - configuration, research, integration of some technologies, etc. but not specific features;
- **User** - features related with the User module;
- **Ticket** - features related with the Ticket module;

BIBLIOGRAPHY

Key	Reference
AgileAdobe12	Agile @ Adobe (Jun. 2012), *Does every item in the product backlog require a User Story?*. Available

	at http://blogs.adobe.com/agile/2012/06/20/does-every-item-in-the-product-backlog-require-a-user-story/ [Accessed Oct. 2015].
Ambler04	Scott Ambler, The Object Primer, Cambridge University Press, 3rd Edition, 2004, ISBN: 978–0–521–54018–6
Brown10	Dan Brown, Communicating Design, Peachpit Press, 2nd Edition, 2010, ISBN: 978–0–321–71246–2
Cohn04	Mike Cohn, User Stories Applied: For Agile Software Development, Addison-Wesley Professional, 3rd Edition, 2004, ISBN: 978–0–321–20568–1
Raymond04	Raymond, E. S., & Landley, R. W., The Art of Unix Usability, Pearson Education, Inc., 2004
Robinson08	Robinson, S. (2008), Conceptual Modelling for Simulation Part I: Definition and Requirements. Journal of the Operational Research Society, 59 (3): 278–290.
Shasha92	Shasha, D. (1992), Database Tuning - A Principled Approach, Prentice-Hall, 978–0132052467
ScrumInst15	International Scrum Institute, *The Scrum Product Log*. Available at: http://www.scrum-institute.org/The_Scrum_Product_Backlog.php [Accessed Oct. 2015]