

# **HOW TO DEVELOP A WEB-APPLICATION FOR TRAVELING SOLUTIONS WITH THE MOST INNOVATIVE TECHNOLOGIES**

**Author: Héctor Yone Lacort Collado**

**Director: David de Andrés Martínez**

**Co-director: Juan Carlos Ruiz García**

# Resumen

Se presentará el marco donde Waymate nació como idea conceptual, una aplicación que ofrece soluciones para viajar de una ciudad a otra mediante diferentes medios de transporte de una forma fácil e inteligente. Esta plataforma soporta aplicaciones cliente para web y dispositivos móviles. Sin embargo, este documento describe el procedimiento para crear la parte del cliente para la aplicación web, además de la parte del servidor, la cual es empleada para los diferentes clientes.

Las diferentes técnicas del Ciclo de Vida de Desarrollo de Software serán descritas y la selección de estos diferentes métodos para éste propósito que concierne al desarrollo de la aplicación justificados.

Existen diversas plataformas y tecnologías para implementar una aplicación de tal complejidad para la parte del cliente y del servidor, pero ¿cuáles de ellas son las adecuadas e innovadoras?

Este documento abordará la descripción de diferentes alternativas para la implementación de diferentes características y funcionalidades requeridas para este tipo de aplicaciones, especificando las ventajas y desventajas que presentan, las razones por las que han sido elegidas y los beneficios que aportan a la funcionalidad de la aplicación. Todas estas características descritas en los siguientes puntos, están consideradas a la vanguardia por diferentes comunidades de desarrolladores:

- Framework para el desarrollo de la parte del servidor. Ruby on Rails proporciona diversas características que permiten la implementación de una completa aplicación, en cuanto a cliente y servidor, ya sean incluidas en el mismo framework o en librerías externas. Entre estas funcionalidades, cabría destacar el almacenamiento de datos, servicios RESTful, internacionalización, configuración de activos, despliegue de nuevas versiones, testeo y el patrón de diseño y estructura de código MVC.
- Tecnologías de la parte del cliente. Librería que incluyen patrones innovadores para organizar el código y los archivos, como también una interfaz de usuario altamente interactiva.
- Despliegue de versiones. Exponiendo las últimas estrategias y procedimientos para llevar a cabo este proceso de una forma inteligente y fiable. Cómo testear antes de desplegar una nueva versión, porque usar *multi-staging*, monitorizar pruebas una vez la nueva versión es accesible a través de una URL a todos los usuarios y como configurar estos frameworks.
- Mejora del rendimiento de la aplicación en la parte del cliente gracias a diferentes tecnologías empleadas.
- Diferentes procedimientos para testear todo el código posible, siguiendo metodologías de Desarrollo Guiado por Pruebas o por Comportamiento, para el código del cliente y servidor, además del flujo de datos.

Con respecto a todas estas tecnologías, se proveerá de una descripción de consejos para su integración e implementación, de forma que un desarrollador de software pueda llevarlo a la práctica.

Finalmente el flujo de datos y las diferentes etapas a lo largo de la aplicación serán descritos. Para las diferentes etapas ha sido necesario incluir servicios externos para facilitar la experiencia del usuario, tales como APIs proporcionadas por Google. Sus ventajas e inconvenientes también serán contrastados.

# Abstract

It will be presented the frame where Waymate was born as a conceptual idea, an application which offers solutions to travel from a city to another by different means of transportation in an easy and smart way. This platform supports web and mobile client applications. However, this document describes the procedure for building up the client-side for the web app and the server-side, which is employed for all the different clients

The different techniques of Software Development Life Cycles will be described and the selection of the different methods for the purpose which concerns the application development justified.

There exist several platforms and technologies for building up such a complex application for the client and server-side, but which of them are the most appropriate and innovative?

This document will tackle the description of different alternatives for implementing different features and functionalities required for this kind of applications, specifying the upsides and downsides which present, the reasons because they've been selected and the benefits which bring to the application functionality. All these features described in the following points are considered in the vanguard from several developer communities:

- Server-side framework. Ruby on Rails provides several features which allow implementing a complete server and client-side, either included in the framework or as external libraries. Among these features, it should be mentioned data-storage, RESTful services, internationalization, assets configuration, deployment, testing and the Model/View/Controller pattern.
- Client-side technologies. Libraries which include innovative patterns to organize the code and the files within the project, as well as a highly interactive user interface.
- Deployment. Exposing the latest strategies and procedures to carry out this process in a smart and reliable way. How to test before deploying, why using multi-staging, test-monitoring once the new version is on the live system after being deployed and how to configure all these frameworks.
- Performance improvements of the application on the client-side by means of different technologies employed.
- Different procedures to test all the testable code, following the Test-Driven and Behavior-Driven Development methodologies, for the client and server-side code, as well as the workflow.

Regarding these technologies, technical insights and advices for their integration and implementation will be described in a way that could help to a developer who wants to use them.

Finally the workflow and the different stages along the application will be described. For the different stages it's been necessary to include external services to facilitate the user experience along the process, such as Google APIs. The pros and cons of all these features will be contrasted.

# Index

Resumen.....	1
Abstract .....	3
Table of images .....	6
Table of code snippets .....	7
1. INTRODUCTION .....	8
Waymate as a conceptual idea .....	9
Document structure .....	10
2. SOFTWARE DEVELOPMENT LIFE CYCLE.....	11
3. SERVER-SIDE .....	14
Alternatives for developing a web application .....	14
Ruby on Rails .....	16
3.1. Rails components .....	16
3.2. Rails environments.....	18
3.3. MVC in Rails.....	19
3.4. Rails internationalization.....	32
3.5. Testing .....	34
3.6. Asset Pipeline .....	39
4. DEPLOYMENT PROCESS.....	42
4.1 Web-based hosting service .....	42
4.2. Git. Distributed revision control.....	43
4.2.1. Git's data structures.....	44
4.2.2. Git from the console.....	45
4.2.3. Conflicts.....	48
4.3. Capistrano deployment.....	49
4.4. Multistage .....	52
4.5. Testing before deploying.....	53
4.5. Website monitoring system .....	54
TinyMon .....	54
5. APIs data abstraction into a common object .....	56
5.1. Deutsche Bahn API .....	56

5.2. Ypsilon API .....	57
6. CLIENT-SIDE .....	59
6.1 HTML .....	59
6.2. HTML appearance elements .....	60
6.2.1. CSS .....	60
6.1.2. SASS .....	61
6.3. Dynamic web-site .....	64
6.3.1. JavaScript.....	64
6.3.2. AJAX.....	64
6.3.3. jQuery .....	65
6.3.4. Backbone .....	66
6.3.5. Testing the client-side .....	76
7. Waymate. How it works .....	79
7.1. Object models on the server-side .....	79
7.2 Workflow along the entire process.....	83
7.2.1. Landing page .....	84
7.2.2. Search page .....	86
7.2.3. Results page .....	92
7.2.4. Details page.....	96
7.2.5. Booking form .....	99
7.2.6. Tickets .....	104
7.2.7. Hotels booking .....	105
Conclusion .....	106
References.....	108

## Table of images

Image 1: New feature development cycle .....	13
Image 2: Bug-fixing cycle.....	13
Image 3: Request forgery attack .....	29
Image 4: Git states, workflow and areas.....	44
Image 5: Master and develop branches.....	46
Image 6: Hierarchy of object models which compose an Order.....	80
Image 7: Hierarchy of object models which compose a Tour .....	81
Image 8: Landing page .....	84
Image 9: Email subscription .....	84
Image 10: Log in window in landing page.....	85
Image 11: Search page. Google Places auto-completion.....	86
Image 12: Search page. Calendar.....	89
Image 13: Search page. Travelers. ....	90
Image 14: Results page. Map and Navigation tabs .....	92
Image 15: Results page. Traveler connection preferences .....	93
Image 16: Mean of transportation filters.....	93
Image 17: Duration slider filter .....	93
Image 18: Price range slider filter .....	93
Image 19: Time slider filter .....	94
Image 20: Sorting functions .....	94
Image 21: Travelers profile customization.....	94
Image 22: Special Price checkbox .....	95
Image 23: Results page. Connections frame.....	95
Image 24: Details page.....	96
Image 25: Details page. General representation of a connection .....	97
Image 26: Details page. Connection details expanded .....	97
Image 27: Details page. Price switching.....	97
Image 28: Details page. Conditions for a type of price .....	97
Image 29: Details page. Travelers profile set.....	98
Image 30: Booking form. Traveler profile set and details view for selected connections .....	99
Image 31: Booking form. Seat reservation conditions.....	100
Image 32: Booking form. Seat selection.....	100
Image 33: Booking form. Customer personal data and address.....	101
Image 34: Booking form. Credit card information .....	101
Image 35: Booking form. Online identification .....	101
Image 36: Booking form. Travelers' personal information .....	102
Image 37: Booking form. Terms and conditions .....	102
Image 38: Pop-up errors when validation fails .....	103
Image 39: Tickets. Travelers preferences set and details view for purchase connections .....	104
Image 40: Tickets. Tickets purchase links and information .....	104
Image 41: Hotels booking.....	105

## Table of code snippets

Code snippet 1: Database configuration for development environment.....	18
Code snippet 2: Migration file template.....	22
Code snippet 3: Layout file skeleton.....	24
Code snippet 4: Book object bound to a form.....	26
Code snippet 5: Resources defined under the 'admin' scope.....	30
Code snippet 6: Route names specified explicitly for 'login' and 'logout'.....	30
Code snippet 7: Chunk from the English translation file.....	32
Code snippet 8: Implementation to set the locale from the browser header.....	32
Code snippet 9: Locale scope definition for the languages supported.....	33
Code snippet 10: Unit test with two assertions for a CustomerTest model.....	35
Code snippet 11: Required lines for BDD libraries inclusion in the BDD testing environment...	37
Code snippet 12: Scenario example.....	38
Code snippet 13: Step definitions for a scenario.....	38
Code snippet 14: JavaScript files inclusion in the project from the application.js file.....	40
Code snippet 15: JavaScript files organization.....	40
Code snippet 16: Compression libraries inclusion in the production environment.....	41
Code snippet 17: Repository creation in git.....	45
Code snippet 18: Template to proxy the Mongrel with a website.....	51
Code snippet 19: Multistaging configuration for two different stages.....	52
Code snippet 20: Integrate TyniMon in the production stage.....	55
Code snippet 21: ExceptionNotifier configuration.....	55
Code snippet 22: SASS. Variable definition example.....	62
Code snippet 23: SASS. Nesting classes example.....	62
Code snippet 24: SASS. Mixin function example.....	63
Code snippet 25: JavaScript library inclusion from a HTML template.....	65
Code snippet 26: Backbone. JavaScript files organization.....	66
Code snippet 27: Backbone. Model definition.....	69
Code snippet 28: Backbone. Router definition example.....	74
Code snippet 29: Backbone. View definition template.....	75
Code snippet 30: Test suite definition with 4 Specs.....	77
Code snippet 31: Migration file example which introduces or removes an airline.....	80

# 1. INTRODUCTION

Nowadays, people are used to travel quite often given the facilities that different means of transportation offer. Only inside the German boundaries, along one year, 10 billion of travels are carried out. However, there is no a complete support from the existing technologies along the journeys in the market.

When travelers land in an unknown city, they could feel lost and ask to some people how to get to their destination. They won't probably find the fast and cheap way to travel within the city from one point to another. How this can still happen in such a computerized world?

Currently, with the fast evolution of technologies, smartphones are becoming more powerful as days go by. These mobile phones provide much more than just making calls or sending messages. It can be used for communication and computing purposes, because they run a complete operating system, can connect to internet and navigate, have GPS, and more other features. They could be considered as small computers given the capabilities which present and possibilities which offer.

The number of users who own a smartphone is growing continuously day after day. People goes everywhere with a small computer in their pocket. Due its power, it's employ for making life easier having a large amount of application which helps daily to millions of users.

Nevertheless, travelers don't have yet an application available in the market which provides a complete support when traveling for long and short distance, all in one.

In the market, there exist applications which offer travel solutions for long distance from web and mobile apps in a smart way, such as *"Hipmunk"*.

For short distance, using public transportation, there exist several applications, but usually only support a single city. Google is in progress of developing *"Google Transit"* offering all the connections for determined cities all over the world, but there are a lot of cities missing in Germany, and not all the public means of transportation are included.

It also exist web applications which include travel solutions for short and long distance, but the way they present the information is not so clear, such as *"Rome2rio"*.

Other applications offer information about different kinds of places nearby such as *"Google+ Local"*.

Then imagine an application available for web and mobile devices which includes all these features in one, offering a clear and fancy interface as well as a simple interaction. Here is where Waymate as a conceptual idea begins.



## Waymate as a conceptual idea

The main idea is to build up an application which presents door-to-door travel solutions, including short and long distance, in a clear and smart way where the user could perceive advantages for its preferred conditions.

Besides, the application will present different solutions for going from an origin to a destination point. All the existing means of transportation which help to cover the journey will be included, such as trains, flights, public transport, car-sharing, car-renting, and so on. Within this set of solutions users will be able to set their preferences. Hence, the solutions which fit better accordingly to the user preferences will be presented.

Tickets purchase will be bundled in one single booking process, instead of buying each ticket separately, making the process easier and faster.

A web and mobile client application will support short and long-distance. However, the main functionality which users will employ in the web-application will be to plan in advance long-distance journeys. On the other hand, mobile application will be meant to provide support during the journey, such as finding places nearby, representation of the current stage along the journey such as remaining time and current location, as well as offer instantaneous solutions for short distance.

It will be possible to plan the trip regarding the connections the traveler will take before starting it. This means when the user arrives to a city where there is no internet access through his smartphone, the path over the map with all the connections and information will be saved

This will be the developing process for the web-application, becoming more complete after each step:

- 1) The first version will cover completely long distance within Europe by flights, but outside the German boundaries, train connections will be offered only for important cities.
- 2) The second version will cover long distance by flights all over the world.
- 3) The third will include solutions for short distance inside Germany.
- 4) In the following versions, once Germany is fully covered, the application will expand the boundaries to the adjacent countries for train and public transport.

In parallel will take place the development of the application for mobile devices. This development will follow the same steps, than the web app and as the hard work for getting all that functionality is implemented on the server side. Then, it will be time for developing the features mentioned previously for giving support to travelers along the journey as everything is advancing in the web app.

The business model is to get a percentage of the train and flight tickets bought through the platform, as well as hotels booking. Looking forward, probably this platform would offer discounts for going to a restaurant, for shopping in a determined shop or doing whatever activity in the destination city.

The model to follow will be to concentrate in Germany and then gradually expand to the surrounding countries, with the ambitious objective of providing complete public transport information and train connections for the entire European Union.

## Document structure

The frame where this document is been developed is in a start-up company, called Door2Door, located in Berlin during an Erasmus Internship.

The first section tackles the Software Development Life Cycle in the context of this application, describing the existing models and explaining which of them, in combination or alone, fit better for different development processes.

Afterwards, technologies in the vanguard for developing a complete and complex web-application from the server to the client-side are described. Regarding these technologies:

- Will be explained one by one, describing their components and features which will help to a developer to have better understanding.
- Will be compared to similar ones. Upsides and downsides of their usage will be accounted for as well, presenting the reason which promoted the developers team to select the chosen one among all the variety.
- Describing good practices, advices, useful tips and commands. All together will help a developer to follow the proper way of doing things when programming.
- Explanations about how-to configure a complete architecture for the server-side.
- Patterns to follow for some of these technologies.
- Usage to have a clean and organized code, making easier the maintenance of application.
- Different procedures for testing as much code as possible, as well as the workflow.

Finally, once all these technologies are described, the whole process along Waymate, the application built up, will be described, such as:

- Workflow.
- Models on the client and server-side.
- Views and events.
- Other features included.

## 2. SOFTWARE DEVELOPMENT LIFE CYCLE

Software development life cycle is a multistep process approach applied to the development of information system solutions.

The main function of a lifecycle model is to establish the order in which a project specifies, designs, implements, tests, and performs. It establishes the criteria for determining whether to proceed from one task to the next.

All the activities involved are highly related and interdependent. Therefore, in actual practice, several developmental activities can occur at the same time. So, different parts of a development project can be at different stages of the development cycle.

Waymate's team employs **Jira** as a proprietary issue tracking product for bug tracking, issue tracking and project management.

These are the main methodologies for fulfilling software development<sup>1</sup>:

- **Waterfall** model is a sequential development approach, in which development is seen as flowing steadily downwards through the phases of requirements analysis, design, implementation, testing, integration and maintenance.

Its basic principles are:

- Project is divided into sequential phases with some overlap between phases.
- Emphasis on planning, time schedules and implementation of an entire system at one time.
- Tight control is maintained over the life of the project.

- **Software prototyping** is the development approach of activities during software development, the creation of incomplete versions of the software program being developed.

The basic principles of this model are:

- It's an approach to handle select parts extracted from a larger one.
- The project is broken into smaller segments in order to provide an easy way of changing parts during the development.
- The user is involved throughout the development process.
- A basic understanding of the fundamental business problem is necessary to avoid solving the wrong problem.

- **Spiral** model combines elements of design and prototyping-in-stages. It's actually a meta-model which can be used by other models.

The basic principles which this model presents are:

- Break the project into smaller segments for more easy-of-change during the development process.
- Each cycle involves a progression through the same sequence of steps.
- Each trip around the spiral traverses:
  1. Analysis.
  2. Evaluation.
  3. Planning.
  4. Development.

---

<sup>1</sup>Software development life cycle: [http://en.wikipedia.org/wiki/Software\\_development\\_methodology](http://en.wikipedia.org/wiki/Software_development_methodology)

- **Rapid application development (RAD)** involves iterative development and prototypes' construction.

RAD basic principles:

- Fast development and delivery.
- Break the project into smaller segments for more easy-of-change during the development process.
- Aims to produce high quality systems quickly via Prototyping, active user involvement and computerized development tools.

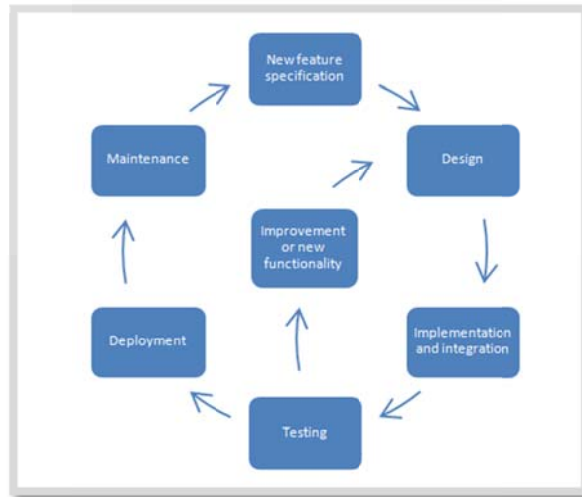
In this application it can be distinguished different kinds of development cycles for different purposes.

### 1. **New feature development cycle.**

It follows a mixture between the waterfall and the spiral model. Once the first version of the feature is implemented some aspects are improved or new features added. Thus, until the first version is implemented, the waterfall model drives the development process and then the spiral methodology is followed till the final version is achieved.

Using this procedure, when there are tough deadlines, first versions of new features can be implemented, being improved eventually.

- 1) New feature proposal by a single or several team members.
- 2) In the weekly product review, this new feature is discussed among all the team. If it's considered convenient to implement, afterwards is discussed the approximate moment to implement the feature.
- 3) The Product Manager creates the issue in JIRA and assigns it to the Chief Technology Officer (CTO).
- 4) The CTO assigns the issue to:
  - a. The designer if the feature involves some design issues. Once the design has been provided, the designer re-assigns the issue to the CTO and the CTO follows the step 4) b.
  - b. The developer who considers more appropriate, specifying the exact moment to implement and further details.
- 5) When it's time to implement it, the developer invests some time researching if necessary. Otherwise, he starts developing and creates the testing functions if necessary. If the new feature includes sub-features, as soon as they are already implemented, the host-based repository is updated and each update is documented in JIRA and in the commit messages.
- 6) The developer implements the corresponding tests.
- 7) If there are some improvements or functionalities to add, they are specified in JIRA and afterwards implemented when the CTO determines.
- 8) If the developer considers that the issue is finished, talks to the CTO or assigns him the issue through JIRA.  
If everything works as expected the issue is closed in JIRA, otherwise, the developer should go back the 5) step.
- 9) When the complete new feature is already pushed in the host-based repository, the new version is deployed on the live system, making accessible to the users.

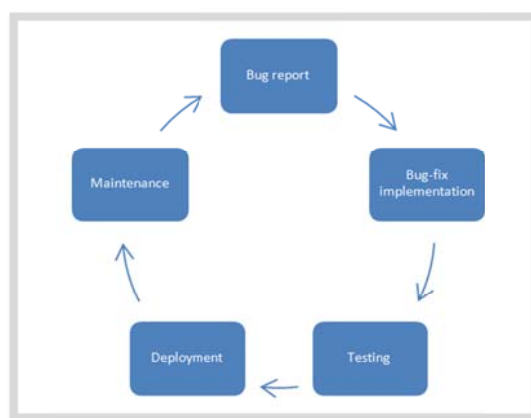


**Image 1: New feature development cycle**

## 2. Bug-fixing cycle:

An adapted waterfall model is followed to fix bugs. It's not completely equal because there is no theoretical analysis. The waterfall model fits perfectly here because it's a simple procedure, where the bug is fixed and nothing else is done from the development point of view.

- 1) Bug identified by a team component or a testing user.
- 2) Bug reported to the Product Manager.
- 3) Product manager creates a new issue in Jira and assigns it to the CTO.
- 4) CTO assigns the corresponding bug issue to the developer which considers will identify the problem sooner.
- 5) The developer fixes the bug and updates the corresponding tests if necessary.
- 6) Finally, the developer documents the bug in the commit message and in the corresponding issue in JIRA, in order that all the team members, not just developers, could know what caused the problem. Then, the issue can be closed in JIRA and consider the bug as fixed.
- 7) When the bug-fix is pushed and available from the host-based repository, the new version is deployed on the live system.



**Image 2: Bug-fixing cycle**

## 3. SERVER-SIDE

The server-side refers to a software program, in this case a web server, which runs on a remote server. It's needed in order to access information or functionality which is not available on the client-side. Furthermore, some operations must be executed from the server instead of from the client-side given its unreliable behavior.

It includes operations such as processing and storage from a client to a server. It also works providing functionality to a big amount of clients concurrently.

In Waymate client and server interaction is reduced to a minimum possible, only done when it's needed. The server tries to perform as many operations as it can, because Waymate will have web-application, iOS and Android clients eventually. If operations are not executed on the server will imply to have a triplication of these operations in the three different clients.

In this section, an overview will be taken about Ruby on Rails, framework which is becoming popular for developing the server side, due its power and facilities which provides to include some functionalities and features to a web application.

### Alternatives for developing a web application

**Ruby on Rails (RoR)** cannot be directly compared with other programming languages for web development such as PHP, Java, C++ or ASP.NET.

Ruby itself is a programming language object oriented which combines syntax inspired by Perl and Smalltalk. It supports multiple programming paradigms, such as functional, object oriented, imperative and reflective. It includes a dynamic type system and automatic memory management.

On the other hand, Rails is a complete framework which uses Ruby as programming language.

Hence, RoR could be compared against a programming language with a complete framework<sup>2</sup> with similar features:

- CAKEPHP, is a complete framework for programming in PHP.
- Apache OFBiz and Play, for Java web application developers.
- ASP.NET MVC, for ASP.NET web programmers.

All these frameworks support nearly the same features, following different procedures, plugins or mechanisms. These features are:

- Ajax
- Model/View/Controller pattern

---

<sup>2</sup> Comparison of web application frameworks:  
[http://en.wikipedia.org/wiki/Comparison\\_of\\_web\\_application\\_frameworks](http://en.wikipedia.org/wiki/Comparison_of_web_application_frameworks)

- i18n or L10n for supporting different languages

Besides, they include frameworks or have available plugins for specific functionalities:

- Testing
- DB migrations
- Security
- Template
- Caching
- Form validation, on the client or server-side.

However, even including all these features, it doesn't mean that are included in an intelligent way, completed or making things easier as it was expected.

The strongest competitor against RoR is CAKEPHP from the developers' point of view, given that both own nearly the same functionalities, completed and somehow making things easier. Actually, CAKEPHP has copied a lot of functionalities which were introduced originally in RoR. Obviously comparing each other, upsides and downsides can be found for RoR and CAKEPHP. Thus, in the end, what does make the difference?

- Ruby on Rails is becoming quite popular in the latest years.
- In cities such as Berlin, where the company is placed, where technology is quite developed and important, web applications developed in this framework have good reputation.
- Furthermore, Rails community is very active and is increasing exponentially as time goes on, organizing several events and giving good support for those developers who have some difficulties implementing some functionality or setting up a plugins (a few years ago this community wasn't as populated and helpful as is right now).
- Rails makes things easier. In the following Ruby on Rails introduction, more aspects will be explained, reflecting this fact and giving sense to the decision of choosing such framework.

# Ruby on Rails

Ruby on Rails is an open source full-stack web application development framework written in the Ruby language. It gives to the Web developer the full ability to gather information from the web server, talking or querying the database and template rendering out of the box.

In Rails prevails Convention over Configuration, which means making assumptions about what and how to do it, rather than requiring specifying every little thing through endless configuration files.

Rails includes a package manager called **RubyGems** for the Ruby programming language which provides a standard format for distributing Ruby programs and libraries, in a self-contained format called a **gem**. This tool was designed to easily manage the installation of gems, and a server for distributing them. For installing and uninstalling a *gem*, in the console it's typed:

```
gem install <gem_name> or gem uninstall <gem_name>
```

## 3.1. Rails components<sup>3</sup>

### Action Pack

Action Pack is a single gem that contains Action Controller, Action View and Action Dispatch, which is the "VC" part of "MVC".

1. **Action Controller** is the component that manages the controllers in a Rails application. The Action Controller framework processes incoming requests to a Rails application, extracts parameters, and dispatches them to the intended action. Services provided by it include session management, template rendering, and redirect management.
2. **Action View** manages the views of the Rails application. It can create both HTML and XML output by default. Action View manages rendering templates, including nested and partial templates, and includes built-in AJAX support.
3. **Action Dispatch** handles routing of web requests and dispatches them as you want, either to your application or any other Rack application.

### Action Mailer

Action Mailer is a framework for building e-mail service layers. These layers are used to consolidate code for sending out forgotten passwords, welcomes for signing up, inboxes for billing or any other notification. It provides a way to make emails using templates. It can also be used to receive and process incoming email.

---

<sup>3</sup> Ruby on Rails components: [http://guides.rubyonrails.org/getting\\_started.html](http://guides.rubyonrails.org/getting_started.html)



## **Active Model**

It provides a defined interface between the Action Pack gem services and Object Relationship Mapping (ORM) gems such as Active Record. Active Model allows Rails to utilize other ORM frameworks in place of Active Record if the application needs it.

## **Active Record**

It is the base for the models in a Rails application. It provides database independence, the four basic functions of persistence storage, *create, read, update and delete (CRUD)*, advanced finding capabilities, and the ability to relate models to one another, among other services.

## **Active Resource**

It provides a framework for managing the connection between business objects and RESTful (Representational state transfer) web services, an approach for getting information content from the server-side. It implements a way to map web-based resources to local objects with CRUD semantics.

## **Active Support**

An extensive collection of utility classes and standard Ruby library extensions that are used in Rails by the core code and applications.

## **Railties**

Railties is the core Rails code that builds new Rails applications and glues the various frameworks and plugins together in any Rails application.

## 3.2. Rails environments

Rails has the concept of environments to represent the stages of an application's lifecycle. It includes as default these three environments:

- **Test**, used for running Rails built in test suite.
- **Development**, is the default environment, optimized for running locally. Full error messages are displayed, caching is disabled, and classes are automatically reloaded, meaning that it's not necessary start and stop running the local server for updating the changes.
- **Production**, configured for speed and security. Caching is enabled, classes not reloaded, and error messages only have simple error codes, instead of full back traces and session information.

Each environment has a corresponding environment file *config/environments/environment\_name.rb* with its particular configuration code. Every environment has its own database as well, defined in *config/database.yml*. For development, the section for the production database is configured as follows:

```
development:
  adapter: mysql2
  database: db_development
  username: root
  password:
  host: 127.0.0.1
  encoding: utf8
```

**Code snippet 1: Database configuration for development environment**

In the console, this command needs to be introduced for starting the server in the other different environment than development:

```
rails server --environment test or rails server -environment production
```

## 3.3. MVC in Rails<sup>4</sup>

Rails uses the **Model/View/Controller (MVC)** architecture pattern to organize application programming. The benefits which MVC includes are:

- Isolation of business logic from the user interface. It keeps separated the representation of information, logic and user interaction.
- Easy to keep the DRY (Don't Repeat Yourself) philosophy. The idea behind is to reuse the different modules in several places, avoiding code repetition.
- Making it clear, where code is well structured and organized properly in their corresponding modules.

### 3.3.1 MVC components

1. **Models** represent the information or data of the application and the rules to manipulate that data. In Rails they are used for managing the rules of interaction with a corresponding database table. In most cases, each table will correspond to one model in the application.
2. **Views** represent the user interface of the application. In Rails, views are often HTML files with embedded Ruby code that perform tasks related solely to the presentation of the data. They provide data to be the display to the web browser.
3. **Controllers** are the link between models and views. In Rails, controllers are responsible for processing the incoming requests from the web browser, asking to models for data and passing that data on to the views for representation.

---

<sup>4</sup> MVC in Rails: [http://guides.rubyonrails.org/getting\\_started.html#the-mvc-architecture](http://guides.rubyonrails.org/getting_started.html#the-mvc-architecture)

## 3.3.2 Models

A model is an Active Record object which includes hooks into this object life cycle to the database persistency, controlling the application and its data.

Creating and saving a new record, Rails will send automatically an SQL INSERT operation to the database. Updating an existing record will send an SQL UPDATE. If any validations fail, the object will be marked as invalid and it won't perform any operation.

### 3.3.2.1 Validations

Validations<sup>5</sup> are preconditions defined in the class model, which are associated to attributes and should be accomplished by the attribute value before being stored in the database. These validations are defined in the file which contains the class and in the same line than the corresponding attribute which will be associated. Active Record uses them to ensure that only valid objects are saved into the database.

Active Record offers pre-defined validation helpers that can be used directly in the class definitions. These are most common helpers:

1. `format` => attribute's values should match to a given regular expression.
2. `length` => the value should have the same length than the one specified.
3. `numericality` => should only contain numeric values.
4. `presence` => the attribute is not empty.
5. `uniqueness` => the attribute's value must be unique.

### 3.3.2.2 Database

Rails includes as default Relational **Database Management System (RDBMS)** SQLite3.

This application is meant to have many clients accessing to a common database, writing and reading concurrently over a network. Hence, it will be needed a RDBMS which will allow a parallel access to different users.

SQLite is an embedded relational database into which runs in-process with the application and reads and writes a single database file. SQLite will work over a network filesystem, but because of the latency associated with most network filesystems, performance will not be great. Furthermore, SQLite shouldn't be used when accessing the database simultaneously from different computers over a network filesystem, due to when users write, lock on the entire file.

---

<sup>5</sup> Model validations: [http://guides.rubyonrails.org/active\\_record\\_validations\\_callbacks.html](http://guides.rubyonrails.org/active_record_validations_callbacks.html)

On the other hand, MySQL is a relational database server that is accessed via a network connection. It runs a server providing multi-user access, given that web servers can handle multiple requests concurrently, and each of these could be connected to the database.

Summarizing, SQLite is the proper RDBMS to work with standalone applications. For client-server applications and more complex deploys, it should be used MySQL.

### 3.3.2.3 Database migrations

Database migrations<sup>6</sup> are a convenient way to alter the database in a structured and organized way. Active Record tracks which migrations have already been run.

For running migrations already coded but not executed, it's just needed to type `rake db:migrate` and Rails will do the rest, working out migrations which still needs to be run, updating the database and the file `db/schema.rb`, which contains the structure of the database application.

When there is more than one migration to be executed, they will run sequentially as they were implemented in time. If something fails in this process, migrations completely executed will change the database, but not those implemented after the one which provoke the error, including itself.

Active Record's functionality abstract the syntax in a way that the structure defined and commands *CRUD* are independent of the *RDBMS* employed. It can be used SQLite for development environment and MySQL for production.

Migrations are subclasses of Active Record. Each of them implements the method `up`, to perform the pertinent transformations and `down`, to revert the changes. These are the functions to modify the database which Active Record provides:

*add\_column, add\_index, change\_column, change\_table, create\_table, drop\_table, remove\_column, remove\_index, rename\_column and rename\_table.*

It is possible to define migrations writing the SQL code without the help of the functions mentioned previously. Therefore, Rails keeps providing flexibility when defining a new migration.

Whenever a migration is run, can be undone by typing `rake db:rollback`. For those cases where a developer uses the methods which Active Record supply, Rails has introduced the *change* method which knows how to revert the changes automatically, without the need of writing the up and down methods. If other methods are required it is necessary to program the proper down function in order to get the previous state in the database in case those changes haven't been executed as it was expected.

---

<sup>6</sup> Database migrations: <http://guides.rubyonrails.org/migrations.html>

Rails includes a migration generator. It must be executed in the console the following to create a new one: `rails generate migration <name_migration>`

A file which would like this

```
class <name_migration> < ActiveRecord::Migration
  def change
    end
end
```

**Code snippet 2: Migration file template**

will be stored in the `db/migrate` directory in the form `YYYYMMDDHHMMSS_<migration_name>.rb`, including the timestamp and migration name. They are ordered in this directory as they occurred in time, so a developer has a better understanding about how the database has been modified regarding the time line.

Specific individual or sets of migrations can be run or reverted. Specifying a target version, Active Record will run the required migrations (up, down or change) until it has reached the stipulated.

Given all this set of functionalities, Rails allows retrieving the desired previous state and keeping track of every change which modified the database.

It is highly recommended just creating the structure of the database and not inserting data from migrations. If new developers join the team, or the system needs to be set it up again for some reason, executing the command `rake db:setup` should execute all the migrations sequentially until get the current state. However, when introducing data by migrations, this is not possible always, given that at some point, a single migration could interrupt the process if it cannot be executed. In this case, the database can be set properly with a dump of the current version from the production environment or from other programmer who owns a database with the current state.

### 3.3.2.4 Associations

In Rails an association<sup>7</sup> is a connection between two Active Record models. Associations are used in order to add features to the models declaratively. These are supported types:

1. `belongs_to` => one-to-one connection where each instance of the declaring model “belongs to” one instance of the other model
2. `has_one` => one-to-one connection where each instance of a model contains or possesses one instance of another model.

An instance can also be matched by proceeding though a third model by `has_many :through`.

---

<sup>7</sup> Associations: [http://guides.rubyonrails.org/association\\_basics.html](http://guides.rubyonrails.org/association_basics.html)

3. *has\_many* => Creates a one-to-many connection and each instance of the model has zero or more instances of another model.

Intances can also be matched by proceding though a third model by *has\_many :through*.

4. *has\_and\_belongs\_to\_many* => Creates a direct many-to-many connection with another model, with no intervening model.

### 3.3.3 Views

Views in Rails are HTML code accompanied of Embedded Ruby code, which allows to access information which is located on the server-side from the templates. Rails can make templates more dynamic due Ruby code could be used within them. All the views features in Rails are described in the sub-sections below.

#### 3.3.3.1 Layouts and Rendering in Rails

The code which renders the views is placed in the corresponding controller. By default, controllers automatically render the layout file, named as `<controller_name>.html.erb`, which is located under the directory `app/views/layouts` and if it doesn't exist a layout with a name which is corresponded to that controller will render the `application.html.erb` layout file. This happens where there is no call to the function `render` specifying explicitly what it's wanted to display. Nevertheless, it could be defined rendered a different layout file, by `render :layout => "<layout_name>"`, which has to be under the directory specified previously, or render a template which corresponds to the name of the controller, without rendering any layout with `render :layout => false`

Within the context of a layout, `yield` identifies a section where the content from the view should be inserted. The simplest way to proceed is:

```
<html>
  <head>
  </head>
  <body>
    <%= yield %>
  </body>
</html>
```

**Code snippet 3: Layout file skeleton**

Files with `html.erb` extension, HTML Embedded Ruby, are HTML files which can contain Ruby code and its helpers for HTML elements. These files are evaluated on the server side, so variables defined in the controller itself or in a different file, can be used within these files. To trigger Ruby code in this templates must be embedded inside the tags `<% ... %>`, for not displaying any content, being useful for operating with variables, creating loops, if-else conditions, and so on. For visualizing output inside this tag `<%= ... %>`

#### 3.3.3.2 Render method

The `render` method<sup>8</sup> does a heavy lifting of rendering the application's content for being used by a browser. It tells Rails which view or asset to use when constructing the response. The most common ways to customize this function:

- To render the view that corresponds to a different action within the same template:  
`render :edit` OR `render :new` OR `render :edit`, etc.

<sup>8</sup> Render method: [http://guides.rubyonrails.org/layouts\\_and\\_rendering.html#using-render](http://guides.rubyonrails.org/layouts_and_rendering.html#using-render)



- To render a template from a different controller:  
`render '<controller_name>/<action_view>'`
- For rendering a response when using a RESTful service, in JSON or XML format:

```
render :json => @class_to_be_rendered or render :xml => @ class_to_be_rendered
```

- For rendering a determine template or layout, which is not an action's view and it's located under the directory `app/views/<controller_name>` or `app/views/layout`:

```
render :layout => '<template_name>'
```

If it would be in a different file

```
render :layout => '<controller_name>/<template_name>'
```

Another way to handle returning responses to an HTTP request is with `redirect_to`. This method tells the browser to send a new request for a different URL. Comparing the method `render` with `redirect_to`, the difference is basically that the code stops running and waits for a new request for the browser.

### 3.3.3.3 Partials

Partial templates are another feature for breaking the rendering process into more manageable chunks. Furthermore, it makes easier the reusability of snippets which are repeated in different files. For instance, for content that is shared among all pages in the application, such as a footer, partials can be used directly from layouts.

Partials can even receive as arguments local variables, making them more powerful and flexible.

```
<%= render :partial => "book", :locals => {author: @book.author, title: @book.title} %>
```

Other important feature which partials include is to render collections. When a collection is passed, the partial will be inserted once for each member within the collection.

```
<%= render :partial => "book", :collection => "library" %>
```

### 3.3.3.4 Action View Helpers<sup>9</sup>

Rails provides a bunch of helpers for generating form elements such as checkboxes, text fields, radio buttons and drop down lists. Their names end with `_tag`, such as `text_field_tag`, which generates a single `<input>` element. Rails uses conventions to make possible to submit parameters such as arrays or hashes that will be accessible in `params`.

---

<sup>9</sup> Action view helpers: [http://guides.rubyonrails.org/form\\_helpers.html](http://guides.rubyonrails.org/form_helpers.html)

A common task for a form is to create a model object. These `_tag` helpers can be used for this task, but the developer should ensure that the correct parameter name is used and set default value of the input properly. There are three ways to bind them to an object:

1. Individually and explicitly. Using the first argument as an instance variable name for an object and the second the name of an attribute.

```
<%= text_field(:person, :name) %>
```

2. Binding a form to an object. This functionally implements the parsing function in the server side, assigning the parameters values to the corresponding attributes of an object model by itself.

```
<%= form_for @book, :url => { :action => "create" }, do |f| %>
  <%= f.text_field :title %>
  <%= f.submit "Create" %>
<% end %>
```

**Code snippet 4: Book object bound to a form**

If in the controller is created this book as `@book = Book.new`, then the title attribute will be bind and the value will be assigned when the form is sent.

3. Relying on Record Identification. If a RESTful resource is declared in `config/routes.rb` it's quite easy to attain this functionality. Rails figure out model name and the rest:

- Create a new book:

```
form_for(@article, :url => articles_path)
```

- Edit an existing one:

```
form_for(@article, :url => article_path(@article), :html => { :method => "put" })
```

Rails provides more helpers with simpler and more complex functionalities. However, it was considered to mention only these above, because are the more relevant, interesting and differentiating regarding other similar technologies.

### 3.3.4 Controllers

A controller is thought as the glue between models and views. It makes the model available to the view in order to display some specific data to the user, and it saves or updates data from the user's interaction with the view to the model.

After the application receives a request, the routing will determine which controller and action to run, creating an instance of that controller and producing the appropriate response and output. Rails Action Controller makes most of the groundwork using conventions to make it as easy as possible.

For a conventional RESTful application, the controller by itself will receive a request, will fetch the data from a model and use a view to create HTML. If something different needs to be done, additional code will be introduced in the corresponding function.

It inherits from ApplicationController which at the same time inherits from ActionController::Base, providing a number of helpful methods.

To render data in views or send parameters to the client-side, the format should be specified. For RESTful services, in order to send an object model to the client-side in JSON format it would be necessary to include this line, i.e. for a search object:

```
render :json => @search
```

#### 3.3.4.1 Parameters<sup>10</sup>

To retrieve the data sent from the client-side it will be necessary to access to the parameters. Controllers could receive parameters in two different ways:

- As a part of the URL, called string query string parameters. The query string is everything after "?" in the URL.
- The so-called POST, where the information comes from an HTML form filled in by the user. It can only be sent as part of an HTTP POST request.

The good thing is that Rails doesn't make any distinction and both are available in the `params` hash in the controller.

Parameters are not limited to one-dimensional keys and values, they could contain nested hashes or arrays. In this application JSON format is used due to it's easy to retrieve and manipulate attributes. If parameters are sent in JSON, like in this example,

```
{ "book" : { "title" : "value1" , "author" : "value2" }
```

To retrieve the value title, it would be necessary to code `params[:book][:title]`

---

<sup>10</sup> Parameters: [http://guides.rubyonrails.org/action\\_controller\\_overview.html#parameters](http://guides.rubyonrails.org/action_controller_overview.html#parameters)

### 3.3.4.2 Session<sup>11</sup>

The application has a session which stores small amount of data that will be persisted between requests. Each session use a cookie to store a unique ID, because it's more secure than pass the session ID by URL. Session is only available in the controller and the view and can use different storage mechanisms:

- `CookieStore`, everything on the client.
- `SessionStore`, in the database.
- `CacheStore`, in Rails cache.
- `MemCacheStore`, in a memory cached cluster.

The option used in Waymate is to store the session in the cache which itself is stored in the memory as default size 32 Mb, being enough, given that it's not required to store a big amount of data. In case that the size is exceeded in Rails, a cleanup occurs removing the least used entries.

The attributes stored in the session are in JSON format and can be accessed in the same way than `params`. For retrieving a search model stored in the session

```
session[:search]
```

### 3.3.4.3 Filters<sup>12</sup>

Filters are methods which run before, after or around a controller action. If filters are set on  `ApplicationController`, they will be run on every controller, avoiding the need to repeat the corresponding line in each controller.

Waymate uses filters to restrict the access to some URLs of the application, such as resources stored in the database as train stations, airports, and so on, to users with the admin role. This resources will only accessed by developers, due common users won't need to know about them. To access some specific sections in the application, such as user information, login will be required. This is achieved by introducing this line before defining the functions in a controller file:

```
before_filter :login_required
```

---

<sup>11</sup> Session: [http://guides.rubyonrails.org/action\\_controller\\_overview.html#session](http://guides.rubyonrails.org/action_controller_overview.html#session)

<sup>12</sup> Filters: [http://guides.rubyonrails.org/action\\_controller\\_overview.html#filters](http://guides.rubyonrails.org/action_controller_overview.html#filters)

### 3.3.4.4 Request forgery protection<sup>13</sup>

Cross-site forgery is a type of attack in which a site tricks a user into making requests on another site, possibly adding, modifying or deleting data on the site without the user's knowledge or permission.

The first step to avoid this is to make sure all destructive actions (create, update and destroy) can only be accessed with non-GET requests. Following RESTful conventions this is achieved.

A malicious site can still send a non-GET request to the site. The second step, which Rails introduces, is to add a non-guessable token which is only known in the server to each request. This token is added to every form using the form helpers, so the developer doesn't have to worry about.

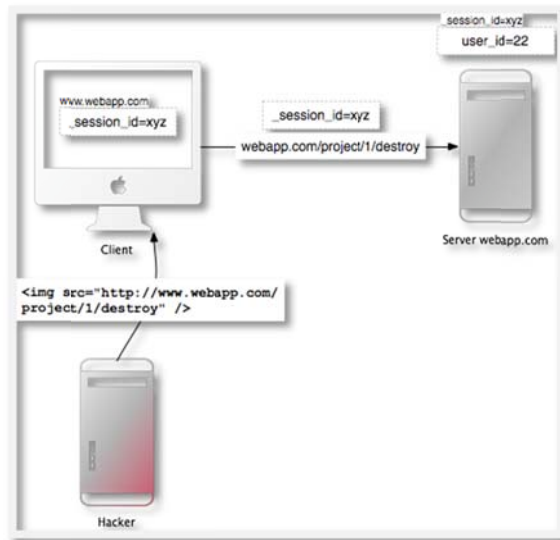


Image 3: Request forgery attack

### 3.3.4.5 Rails Routing<sup>14</sup>

The Rails router recognizes URLs and dispatches them to a controller's action. By default routing allows to declare common routes for a given resourceful controller. Instead of declaring separate routes for the *CRUD* actions, a resourceful route declares them in a single line of code. For the object model "searches", with this line all this functionality is obtained:

```
resource :searches
```

<sup>13</sup> Request forgery protection: [http://guides.rubyonrails.org/action\\_controller\\_overview.html#request-forgery-protection](http://guides.rubyonrails.org/action_controller_overview.html#request-forgery-protection)

<sup>14</sup> Rails routing: <http://guides.rubyonrails.org/routing.html>

This line adds seven different routes mapping to the controller functions.

HTTP Verb	Path	action
GET	/searches	index
GET	/searches/new	new
POST	/searches	create
GET	/searches/:id	show
GET	/searches/:id/edit	edit
PUT	/searches/:id	update
DELETE	/searches/:id	destroy

**Tabla 1: Routes mapping example**

Rails allows to have resources and determine URL under a determine scope. In Waymate the scope is used for defining the routes of the resources only available for users with the *admin* role.

```
scope '/admin' do
  resources :airport_classifications
  resources :airports
  resources :dbstations
  resources :countries
  resources :airlines
end
```

**Code snippet 5: Resources defined under the 'admin' scope**

The scope is used for the `:locale`, in order to match determine language for content translation, but this will be explained deeply in its corresponding section.

In addition to resource routing, Rails supports arbitrary URLs actions. Each of these URLs has to be set up explicitly in the *routes.rb* file. These simple routes makes very easy to map legacy URLs to new Rails actions.

In the application, for login and logout routes, the names are specified explicitly to make it more intuitive as 'login' and 'logout':

```
match 'login', :to => 'user_sessions#new', :via => :get
match 'login', :to => 'user_sessions#create', :via => :post
match 'logout', :to => 'user_sessions#destroy', :via => :delete
```

**Code snippet 6: Route names specified explicitly for 'login' and 'logout'**

The library *Authlogic* provides a complete management for user's authentication (log in and out functionality). It's added to the project by `gem install authlogic`. In addition, this library includes safe password storage, using the *Salt cryptography method*.

Furthermore, the root route must be specified explicitly:

```
root :to => "prelaunch#index"
```

In the end, to see all the entire list of available routes that router will recognize, it should be run this command `rake routes`. Each of these routes will be printed out in the console with:

- The route name
- The HTTP verb used.
- The URL pattern to match.
- The routing parameters of the route.

## 3.4. Rails internationalization

In order to provide a multi-language support for the application, Rails includes a Ruby gem, so-called I18n<sup>15</sup>, which provides an easy-to-use and extensible framework for translating the application.

For each language supported, a file with the format YAML and extension.yml must be included under the directory *config/locales*. These files are named with the `:locale` of the language that will support and contain the translation strings. The file which contains the English translations would look like this:

```
en:
  login:
    username: "Username"
    pass: "Password"
```

Code snippet 7: Chunk from the English translation file

Indentation is quite important, because it allows nesting translation, implementing a better structure to the file. But if there is some line with wrong indentation, the application complains triggering an exception.

The way it works is abstracting all string and other locale specific features, such as date or currency formats, in the application. Rails adds the files mentioned above, to the translations load path (`I18n.load_path`) automatically and makes them available in the application. Then translations are looked up by keys, getting the proper values and instantiating the variables defined in the view template.

To `<%= t 'login.pass' %>`, if the locale is set to `:en`, it will print Password.

If the default locale is not specified explicitly, it will set as `:en` (English). However, Waymate has the default as `:de` (German). To set this value, this line is added in `application.rb`:  
`config.i18n.default_locale = :de`

It's was considered convenient to get the locale from the client supplied information. This information may come either from the IP or the browser. To get the locale from the user's preferred language in the browser is a good solution and simpler, because otherwise the IP have to be looked up in a database to identify from which country comes the request to the server. A trivial implementation is provided in this snippet:

```
def set_locale
  logger.debug "** Accept-Language: #{request.env['HTTP_ACCEPT_LANGUAGE']}"
  I18n.locale = extract_locale_from_accept_language_header
  logger.debug "** Locale set to '#{I18n.locale}'"
end
private
def extract_locale_from_accept_language_header
  request.env['HTTP_ACCEPT_LANGUAGE'].scan(/^[a-z]{2}/).first
end
```

Code snippet 8: Implementation to set the locale from the browser header

<sup>15</sup> Rails internationalization with I18n: <http://guides.rubyonrails.org/i18n.html>



To add the locale to the routes the scope locale must wrap all the content defined within it.

```
scope('/:locale', :locale => Regexp.new(I18n.available_locales.join('|'))) do
  #declared routes
end
```

**Code snippet 9: Locale scope definition for the languages supported**

This line avoids the matching with locales not defined or supported, setting as available in the application en and de.

## 3.5. Testing

Running tests it ensures that the code adheres to the desired functionality and makes easier to detect bugs. Rails provides a skeleton test code in the background when models and controllers are created.

### 3.5.1 Test-Driven Development

On the server-side, Waymate developers follow the **Test-Driven Development (TDD)**<sup>16</sup> as software process. In this process, first the developer writes an (initially failing) automated test case that defines a desired improvement or new function, then produces the minimum amount of code to pass that test and finally refactors the new code to acceptable standards.

Though, sometimes developers follow the logic the other way around, implementing functionality and then creating tests. In the end, the main idea is not adding new functionality without having been tested.

To make this process possible, Rails provides the following components:

#### 3.5.1.1 Fixtures

Fixtures<sup>17</sup> allow populating the testing database with predefined data before the tests run and are database independent. Fixture are YAML-formatted, being human-friendly for describer data.

By default, Rails load all fixtures from the test/fixtures for unit and functional tests. The three steps which involve this process are:

- Remove any existing data from the table corresponding to the fixture.
- Load the fixture into the table.
- Dump the fixture data into a variable in case of accessing it directly.

Their format is as a Hash object. Nevertheless fixtures can also transform themselves into the form of the original class, having access to the methods for that class too.

#### 3.5.1.2 Unit testing

Unit tests<sup>18</sup> are useful for testing small chunks of code. A developer must program testable code, because otherwise these kind of test will be difficult to write or not possible at all.

In Rails unit testing's common use is for individual functions, where ideally each test case is independent from the other. As a good practice, at it should be created at least one test for validations and for every method within the model

---

<sup>16</sup> Test-Driven Development: [http://en.wikipedia.org/wiki/Test-driven\\_development](http://en.wikipedia.org/wiki/Test-driven_development)

<sup>17</sup> Fixtures: <http://guides.rubyonrails.org/testing.html#the-low-down-on-fixtures>

<sup>18</sup> Unit testing: <http://guides.rubyonrails.org/testing.html#unit-testing-your-models>

Unit testing files are used for **testing models**. It must be included in the first line `require 'test_helper'`, given the fact that the file `test_helper.rb`, specifies the default configuration to run the tests. A unit test example:

```
class CustomerTest < ActiveSupport::TestCase
  test "should initialize attributes" do
    customer = Customer.new(:first_name => 'John', :last_name => 'Doe')
    assert_equal 'John', customer.first_name
    assert_equal 'Doe', customer.last_name
  end
end
```

**Code snippet 10: Unit test with two assertions for a CustomerTest model**

How to execute tests:

- To run **unit tests** from the console:  
`rake test:units`
- For an **individual unit test file** :  
`rake test TEST=path_to_test_file`
- To call **an specific test method**:  
`ruby -I"lib:test" path_to_test_file -n name_of_the_method"`

### 3.5.1.3 Functional tests

Functional tests<sup>19</sup> are meant for testing various actions of a single controller. Some features which should be tested:

- Successful request. There are five request types supported for testing (get, post, put, head and delete).
- Correct redirection.
- Successful authentication.
- Correct object stored in the response template. After a request has been made and process, there exist 4 Hash objects for their use:
  - `assigns`. Objects stored as instance variables in actions for using views.
  - `cookies`.
  - `flash`. Objects located in the flash.
  - `session`. Objects living in session variables
- Proper message displayed to the user in the view, by asserting the presence of key HTML elements and their content.

---

<sup>19</sup> Functional tests: <http://guides.rubyonrails.org/testing.html#functional-tests-for-your-controllers>

For functional tests are also available three instance variables, quite self-explainable:

- `@controller`, for the controller processing request.
- `@request`, the request itself.
- `@response`, the response.

This task will be executed for running functional tests: `rake test:functional`

### 3.5.2 Behavior-Driven Development

**Behavior-Driven Development (BDD)**<sup>20</sup> is a specialized version of **TDD** which focuses on behavioral specification of software units and includes some features from **Domain-Driven Design (DDD)** as well.

**DDD** focuses on the strengths of object-oriented software development techniques to simulate the core of the problem. Thus, its core builds a problem simulation domain, minimizing technical influences in order to make more understandable between users, analysts and developers. This aim is achieved using a common vocabulary, called **Ubiquitous Language**.

Hence, **BDD** is a combination of two powerful approaches for software development integrated into one.

Rails doesn't include a framework to support BDD. However, there exist a few libraries which make it possible. There exist more than a single alternative, but the set of libraries chosen for BDD in Waymate are:

- **Cucumber**<sup>21</sup>. It lets to describe how software should behave in plain text. The text is written in a business-readable domain-specific language and serves as documentation, automated tests and development-aid, all rolled into one format.  
`gem install cucumber`
- **Webmock**. It's a library for stubbing and setting expectations on HTTP requests in Ruby.  
`gem install webmock`
- **Capbara**<sup>22</sup>. It helps to test Rails applications by simulating how a real user would interact with the application. It is agnostic about the driver running tests. WebKit is supported through Poltergeist.  
`gem install capybara`

---

<sup>20</sup> Behavior-Driven Development: [http://en.wikipedia.org/wiki/Behavior-driven\\_development](http://en.wikipedia.org/wiki/Behavior-driven_development)

<sup>21</sup> Cucumber: <http://cukes.info/>

<sup>22</sup> Capybara: <https://github.com/jnicklas/capybara>

- **Poltergeist**<sup>23</sup>. It's a driver for Capybara. It allows to run Capybara tests on a headless WebKit browser, provided by PhantomJS.  
gem install poltergeist
- **PhantomJS**<sup>24</sup>. It has several features, but the one involved in this case is *Site Scraping*. The meaning of this feature is to access and manipulate web-pages with the standard DOM API, or with usual libraries like jQuery.

All the files concerning BDD will be under the directory *features*. The file to configure the environment where the BDD will execute is named *env.rb* and located under *features/support*. The relevant lines to include the libraries explained above are:

```
require 'cucumber/rails'
require 'webmock/cucumber'
Capybara.default_selector = :css
require 'capybara/poltergeist'
Capybara.javascript_driver = :poltergeist
```

**Code snippet 11: Required lines for BDD libraries inclusion in the BDD testing environment**

Once these lines are included on the top of the configuration file, depending what is wanted, this environment could be configured in several ways. It's considered as not necessary to specify all the details regarding this configuration.

BDD testing involves an imitation of server controllers in Rails and client pages. There exist three kind of files:

- **Fixtures**, with the aim of simulating real data, such as responses, object models and so on. They can be found under */features/fixtures*.
- **Step definitions** are an analogous to a *method definition / function definition* in any kind of OO/procedural programming language. They can take 0 or more arguments are identified by Regexp matching. Located in the */features/step\_definitions* folder.
- **Features** describe the steps that should happen before the scenario's steps start to execute. These steps will match with a regular expression defined in the step definition in order to execute the corresponding lines.  
**Scenarios** are composed by steps as well. They are meant to imitate user interaction over dynamic pages containing JavaScript code. Features and scenarios are defined in the same file. Features are only defined if some preconditions are needed before executing the scenario's steps. These files are located under */features* directory.

<sup>23</sup> Poltergeist: <https://github.com/ionleighton/poltergeist>

<sup>24</sup> Phantom installation guide: <http://phantomjs.org/download.html>

An example of a Scenario with its corresponding Step definitions below:

```
@javascript
Scenario: Log in
  Given I am a user
  And I am on the login page
  When I log in
  Then I should be on the search page
```

**Code snippet 12: Scenario example**

```
Given /^I am a user$/ do
  name = "user#{rand(Time.now.to_i)}"
  @user = User.create!(:login => name, :password => 'password', :password_confirmation =>
'password', :email => "#{name}@email.com")
end

Given /^I am on the login page$/ do
  visit login_path(:locale => 'en')
end

Given /^I am a logged in user$/ do
  step "I am a user"
  step "I am on the login page"
  step "I log in"
end

When /^I log in$/ do
  fill_in 'user_session_login', :with => @user.login
  fill_in 'user_session_password', :with => @user.password
  click_on 'ANMELDEN'
end
```

**Code snippet 13: Step definitions for a scenario**

## 3.6. Asset Pipeline

The asset pipeline<sup>25</sup> provides a framework to concatenate and compress JavaScript and CSS assets. In production is essential to include such framework, due it reduces the number of requests that can be made in parallel, increasing the speed loading the application.

Rails is integrated with a library called Sprockets which provides this functionality. By default is enabled, but if it's needed to be disabled, in *config/application.rb*:

```
config.assets.enabled = false
```

### 3.6.1 Fingerprinting

In the production environment, Rails inserts an MD5 fingerprint, into each filename and the file is cached in the browser. Fingerprinting is a technique which makes the name of a file dependent on the contents of the file, hence if the content changes the fingerprint will change as well. Besides, HTTP headers can be set to encourage caches to keep their own copy of the content, reducing the requests.

### 3.6.2 Location

Assets in development should be placed under *app/assets* folder. In production, Rails precompiles these files and places them under *public/assets* directory, where the copies with the fingerprint included are then served as static assets by the web server.

### 3.6.3 Assets inclusion for their usage

Assets are linked from the HTML layout or template files. Within the HTML tag `<head>` `</head>` must be added the following lines, specifying the name of the file without its extension:

- Css files: `<%= stylesheet_link_tag "stylesheet1", "stylesheet2" %>`
- JavaScript files: `<%= javascript_include_tag "javascript_file1", "javascript_file2" %>`

Sprockets uses manifest files to determine which assets to include and serve over the entire application. Manifest files contain directives which tell the files to require in order to build a CSS or JavaScript file. It is very common to have a unique file under the directory *app/assets/javascripts* named as *application.js* which contains all the necessary directives to precompile the assets. In the application this file looks like this:

---

<sup>25</sup> Asset pipeline: [http://guides.rubyonrails.org/asset\\_pipeline.html](http://guides.rubyonrails.org/asset_pipeline.html)

```
//Libraries required
//= require jquery
//= require jquery_ujs
//= require jquery-ui

//= require underscore
//= require backbone
//= require mustache

//Files required
//= require app
//= require app_router
//= require_tree ./lib
//= require_tree ./models
//= require_tree ./views
//= require_tree ./collections
```

**Code snippet 14: JavaScript files inclusion in the project from the application.js file**

As JavaScript files contains jQuery, Backbone, Mustache and Underscore methods must be placed before requiring the JavaScript files which make use of them. The six formers `//=require` include the libraries (which will be explained in the client technologies section).

The other lines specify the JavaScript files which use the mentioned libraries. These set of `//=require` lines explain actually how the folders and files are organized within `app/assets/javascripts` directory.

```
app/
  assets/
    images/
    javascripts/
      collections/
      lib/
      models/
      views/
      app.js
      app_router.js
      application.js
    stylesheets/
```

**Code snippet 15: JavaScript files organization**

Thereby, before serving the assets, these libraries must be required, otherwise, it won't identify their methods and will produce several errors.



### 3.6.4 Compression libraries

For CSS files there exists a gem called **YUI** which provides *minification*, removing all unnecessary characters from the source code without changing the functionality.

For JavaScript, there are several libraries, but the one which obtains better performance is **Uglifier** and it's already included in the default Gemfile. It provides *minification* and other optimizations.

### 3.6.5 Environment configuration

It's crucial to have configured the assets properly for the **production** environment, given the enhanced performance that can provide to the application.

In *config/environments/production*, these lines will be necessary to **compress** the CSS and JavaScript libraries commented previously:

```
config.assets.compress = true
config.assets.css_compressor = :yui
config.assets.js_compressor = :uglifyer
```

**Code snippet 16: Compression libraries inclusion in the production environment**

To **specify the assets to be precompiled**, the name of the files, accompanied with the extension, must be included within the brackets of the following command:

```
config.assets.precompile += %w( <assets inclusion> )
```

If there is missing a single file in this section, the application, in the production environment, will fail when is needed.

For getting **fingerprinting enabled**:

```
config.assets.digest = true
```

As **live compilation is not needed** in this application, which would decrease substantially the performance, so:

```
config.assets.compile = false
```

## 4. DEPLOYMENT PROCESS

Specifically in Waymate, deployment is a methodical procedure for updating those repositories which are accessible from a web-browser. Each deployment could be considered as a release version which includes new features or fixes bugs.

The entire process from the starting point, the technologies employed with tips and insights for carrying the deployment out will be described in the sections below.

### 4.1 Web-based hosting service

Reasons for using code hosting services for the private code repository:

- Offsite backup. If something goes wrong in the system maintained by a small team, everything that developers have done so far could be lost. This means it is guaranteed that the code is safe.
- Having a dedicated administrator system would cost maintenance, resources and time. To maintain this system is even more difficult when developers have no experience regarding this issue.
- Team control access to the organization's code.

There exist several platforms which allow having a common repository where developers can deploy their changes and get the updates from others.

**GitHub** is a web-based hosting service for software development which uses the **Git** revision control system. This system provides management of changes to the files which composes the entire project.

Attached with each revision deployed, it could be found information such as the timestamp or the developer who did that deployment. Revisions can be compared, restored and merged. Such features provides to the developers team a powerful tool to work simultaneously on updates, tracking and providing code over changes to source code.

Furthermore, for locating and fixing bugs it's crucial to have the ability to retrieve and run different versions of the software to determine the one that produces it. If developers are geographically dispersed, this technology is quite helpful in the end, due to other team members could identify errors introduced by others comparing different versions of a code snippet.

It is also integrated a wiki powered by Gollum. This wiki simple, clean and open sourced, which can be either imported or exported given it's a repository by itself.

It provides different GUI clients for different Operating Systems which make this process easier.

Reasonably priced compared to other offerings. Each private repository cost s reasonable amount of money. As the company only owns a single one it is worth.

## 4.2. Git. Distributed revision control

**Git** was developed by Linus Torvalds in 2005 for the Linux kernel. It is a distributed version control system which provides creation of a history for a collection of files. It also allows going back in time and reverting the files to an older state.

Distributed version allows everyone to have a local copy, containing the complete source code and history.

Every commit is a complete snapshot of the whole tree and unchanged files are not saved unnecessarily again.

Concerning the integrity, everything is check-summed (SHA). It is impossible to change a file without git knowing it and there isn't file corruption.

Git provides the complete **repository revision history** with all the commit messages and their changes regarding the previous versions since it was created is provided.

Nothing is lost unless git is being forced really, really hard, given that almost everything is undoable. Obviously uncommitted changes can be lost, so the best strategy is to push changes to a different host, making backing-up generally not necessary.

Files can have three **states**:

1. **Committed**: data is safely stored.
2. **Modified**: changed, but not committed.
3. **Staged**: marked a modified file to be committed next.

The **basic git workflow** involves three **actions**:

1. **Modify files** in the working directory.
2. **Stage files**.
3. **Commit**.

Actions occur in these different **areas**:

1. **Working directory**. Single checkout of one version of the project. Files are pulled out of the compressed database for use and modification.
2. **Staging area**. A place to put the changes before they are committed. Changes are added to the staging area. It is still possible to add only parts of a changed file. Everything in the staging area is committed.
3. **Repository**. In the `.git` directory is where git stores metadata and object database. When a repository is cloned this is what is copied.

In the [Image 4](#): Git states, workflow and areas represents these three states and the actions along the workflow for the different areas.

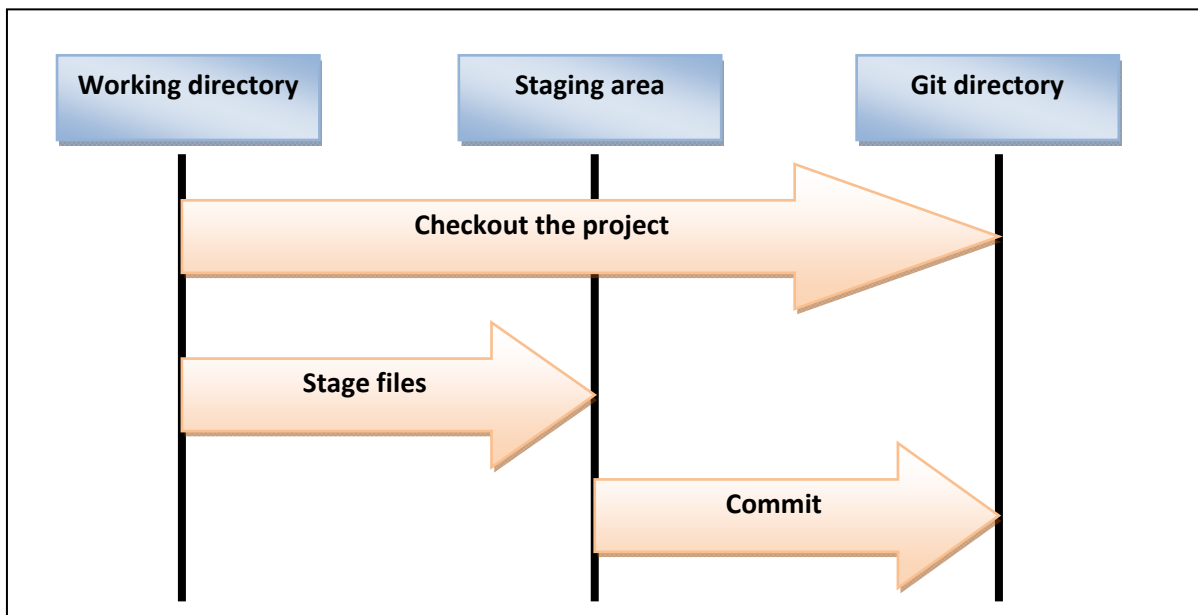


Image 4: Git states, workflow and areas

### 4.2.1. Git's data structures

Git stores just the contents of the file for tracking history, and not just the differences between individual files for each change. The contents are then referenced by a 40 character SHA1 hash of the contents, which means it's pretty much guaranteed to be unique. They are easily referenced by the first 7 characters, which are usually enough to identify the whole string. Hence, every object is referenced with SHA.

**Objects** are anything that it is worked in git such as commit, a tag and so on, which are stored in the database:

- **Blobs** are snapshots version of a source code file.
- A **tree** is a list of object references (*SHAs*) which represents a directory in a commit.
- **Commits** are snapshots of all files in the repository, containing the author, timestamp, commit message, reference to the tree and reference to the parent - commit.
- **Branches** separate code line with own history and are created from a commit. It's a moving named pointer to a commit. It's easy to switch between and merge them.
- **Tags** could be defined as named commits.
- **Refs** are references to commits and are stored in `.git/refs`.

## 4.2.2. Git from the console

Git provides a large bundle of **commands** to work with. In this section, the most important commands and their functionality will be explained briefly.

- **Create a repository:**

```
mkdir <name>
cd <name>
git init
```

Code snippet 17: Repository creation in git

- **Add new or changed files to staging** before committing them:  
`git add <filename>`
- **Remove file from staging:**  
`git reset HEAD <filename>`
- **Remove file from repository:**  
`git rm <filename>`
- To only **delete the file from the repository and not from disk:**  
`git rm -cached <filename>`
- **Moving a file** is automatically staged, but needs to be committed  
`git mv <from> <to>`
- **Creating commits**  
`git commit -m "Message"`
- Change or **amend commit** before being pushed.  
`git commit --amend`
- **Check which files differs**, comparing the local version with the version hosted in the repository.  
`git status`
- Compare the **differences between a changed file** in the local with the one located in the repository.  
`git diff <filename>`
- **Move changes away**  
`git stash`

### 4.2.2.1 Branching

Branching is a methodology to develop new features, considering the current version, separately from other new features being developed at the same time and which have no relation between each other. It's a good practice to create a new branch for a new feature recognizable independent from the current new functionalities which are in developing process.

Git makes really simple merging and branching, actions considered one of the core parts of the daily workflow. Comparing with other technologies as CVS or Subversion, where has always been considered a bit scary and complicated, git makes this process quite easy. This is one of the reasons that Git is being used in this project.

- **Creating a branch**

```
git branch <branch_name>
git checkout -b <branch_name>
```

- **Switching between branches**

```
git checkout <branch_name>
```

- **Deleting branches**

```
git branch -d <branch_name>
```

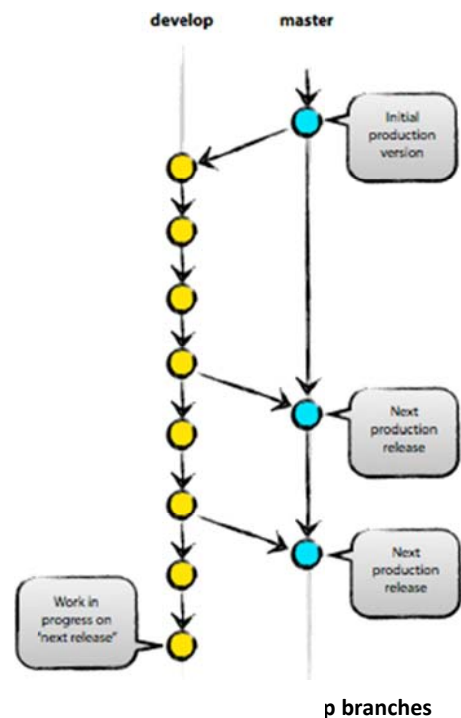
If the branch is not merged in the current fails.

For deleting it anyway

```
git branch -D <branch_name>
```

- **Merging a branch.** The destination branch must be checked out and the source merged.

```
git checkout master
git merge <branch_name>
```



### 4.2.2.3 Remote repositories

- **Cloning from remote.**

```
git clone git@github.com:whatever
```

- **Add/delete a repository**

```
git remote add <name> <remote_url> and git remote rm <name>
```

Deleting a remote also deletes all references from the remote, like branches and tags.

- **Show remote repositories**

```
git remote and git remote -v
```

- **Update all remote branches**

```
git fetch
```

- **Pushing/Pulling a remote branch**

`git push origin <branch_name>` and `git pull origin <branch_name>`

### 4.2.2.3 Tips for pushing

- A push fails if there is not fast-forward. It is solved by pull. For a linear history, **rebasing when pulling.**

`git pull --rebase origin master`

- **Checking out a remote branch.** It is necessary to make sure that the remote branch is known.

`git fetch`

`git checkout origin/<branch_name>`

- **Tracking a remote branch.** To work in a branch which is not in the local.

`git fetch`

`git checkout <branch_name>`

- **Deleting a remote branch.**

`git push origin:<branch>`

`git branch -d origin/<branch>`

And for deleting the local as well: `git branch -d <branch_name>`

### 4.2.2.4 Reverting changes

- **Discard untracked files and directories.**

`git clean -f <dir>` and `git clean -fd<dir>`

- **Undo a single commit.** It not necessary to be the last one

`git revert <commit>`

- **Undo a delete or change.**

`git checkout <filename>`

It also works for whole directory trees.

- **Delete a commit.**

`git reset --soft HEAD^`

The commit is gone, but all changes from the commit are in the staging area. If the **changes are meant to be gone** as well:

`git reset --hard HEAD^`

They are not referenced actually. To **get the changes back** again `git reflog`

### 4.2.2.5 Rebasing

- **Combine** a determine number of **commits into one**.  
`git rebase -i HEAD~<number>`
- **Rebasing branches**. To rewrite history, so that it looks like a branch was originated from another commit.  
`git checkout <branch_name>`  
`git rebase master`
- If there's a conflict, rebase stops. Conflicts must be solved, changes committed and the run.  
`git rebase --continue`
- To **cancel the rebase** if a conflict occurs.  
`git rebase --abort`

There actually exist more advanced topics and commands which Git supports. Nevertheless, with the ones previously commented, a developer's team have an advance control and good management over the repository where they develop simultaneously.

### 4.2.3. Conflicts

Several developers working concurrently in the same project require coordination in the deployment process. Sometimes it is necessary that developers program different functionalities in the same files simultaneously. This fact could cause conflicts merging these files in the local versions of the application which have each developer. Fortunately, GitHub helps to solve this problem. It contains an algorithm which tries to merge automatically those files that have been modified for different developers.

This algorithm doesn't work always, due to it only works when different modules or functions for a specific file have been modified. This case happens when some specific lines of the code have been modified by two or more developers and the changes haven't been pulled before starting to modify that snippet of the code. Hence, when a new version from a working copy is committed and pulled afterwards, in order to get the updated version, the local version of the application gets a conflict which has to be fixed manually. Once the conflicts are fixed by the developer, the changes could be committed. Then another pull is required to merge the files which contained conflicts previously and thereupon, these commits will be pushed, updating to a new version the hosted repository.



## 4.3. Capistrano deployment

Capistrano<sup>26</sup> is an open source tool for running scripts on multiple servers such as the case which concern us, to deploy web applications. It automates the process of making a new version of an application available on a web server.

It is written in Ruby and distributed using the *RubyGems*. To get it installed as rails gem, it's just needed to type `gem install Capistrano`. To configure it's not straightforward.<sup>ss</sup>

The use implementation is by Bash command line. This framework includes the execution of commands in parallel on multiple remote machines via SSH. It uses a simple domain-specific language (DSL), programming language dedicated to a particular solution technique.

It simplifies and automates the deployment to distributed environments and come bundled with a set of tasks designed for deploying Rails applications.

It also includes supporting tasks such as:

- **Databases migrations.** If a new release has some pending migrations, they can be run on the server.  
`cap deploy:migrate` or `cap deploy:migrations`  
Migrations could be configured to be executed after the code is updated on the server automatically by adding this line the file `deploy.rb`:  
after `'deploy:update_code'`, `'deploy:migrate'`
- **Start and stop the application servers.**  
`cap deploy:start`  
`cap deploy:stop`
- **Asset pipeline precompilation.** In `./Capfile` should be this line included  
`load 'deploy/assets'`
- **Cleaning-up old releases.**  
`cap deploy:cleanup`
- **Rollback deployment** of the previous version.  
`cap deploy:rollback`
- **Debugging deployments** which failed  
`cap deploy -d`
- **Test deployment dependencies**, which checks directory permissions, necessary utilities and reports things that appear to be incorrect or missing.  
`cap deploy:check`

---

<sup>26</sup> Capistrano guide: <https://gist.github.com/2161449>

- **Copy the project and update the symlink** to the most recently deployed version. The symbolic link is a special file which contains a reference to the directory in the form of an absolute or relative path.

```
cap deploy:update
```

To update the symlink explicitly, which will be rarely executed.

```
cap deploy:symlink
```

To configure this tool is not quite straightforward. Once Capistrano is already installed in the local machine, these are the steps to configure it properly:

- 1) **Set up SSH access.** To allow Capistrano to log into the OCS server and run the proper commands, the easiest way is by generating an SSH key.

- 2) **Server-side preparations:**

- a. **Request a port** for Mongrel to listen on. Mongrel is a library and web server written in Ruby to run web applications and presents a standard HTTP interface.
- b. **Cache the SVN credentials.** Every time a new version is deployed with this framework, it checks out a new version from the Subversion repository. To make this process work, it has to be checked it out manually on the server. Hence, the credentials used are cached and it won't cause any problem afterwards. To checkout manually the following steps have to be fulfilled:
  - i. Log into the remote server via SSH.
  - ii. Checkout a copy of the trunk.
 

```
svn co -username USER http://svn.domain.com/path/to/repo/trunk
```
  - iii. Let it run till the entire thing is checked out and then delete the folder
 

```
rm -rf trunk/
```

- 3) **Capify the application**

To add Capistrano support to the application the next steps should be followed after typing in the terminal:

```
capify.
```

- a. **Customize the deployment recipe.** The file *config/deploy.rb* which Capistrano has added to the app has to be customized to get it configured for the application. In this file has to be specified things such as:
  - i. Repository link.
  - ii. Port number.
  - iii. Website monitoring system callback.
  - iv. Different stages.

And other different features to make it work properly. Here the developer has kind of freedom because he could configure it differently.

- b. **Setup the remote directory structure.** This action will automatically log into the OCS server and create a new folder with name specified under `set:deploy_to`. In this folder must be created the folders "releases" and

“shared”. Furthermore, the file *database.yml* should be created in the server side, which is the database used by the application hosted.

- 4) **Deploy the website.** For deploying the first time it’s necessary to deploy a ‘cold’ application. It will deploy the code, run pending migrations and it will invoke `deploy:start` to fire up application servers.

```
cap deploy: cold
```

- 5) **Test the deployed website.** In order to make sure that Mongrel is serving up the website correctly, in a browser could be opened with this url <http://website.com:60040>. If the Rails application appears, everything went right.

- 6) **Setting up a proxy for the website.** For not referencing the site by port number for anything other than test it is necessary to proxy the Mongrel to the website. To have the website go to the Rails app without the port, it needs to be created the *.htaccess* file in the *public\_html* folder and place this code chunk inside:

```
RewriteEngine on
RewriteCond %{HTTP_HOST} ^www.domain.com$ [OR]
RewriteCond %{HTTP_HOST} ^www.domain.com$
RewriteRule ^(.*)$ http://127.0.0.1:port_number%{REQUEST_URI} [P,QSA,L]
```

**Code snippet 18: Template to proxy the Mongrel with a website**

- 7) **Starting the application on Reboot.** Once the Mongrel instance is running, it recommended making sure that gets started again if the server has to reboot.

## 4.4. Multistage

A stage is a version of the web-hosted application repository which can be accessed from the browser. It is recommended to have at least two different stages, one for the live system, so-called **production** stage, that users will access and the other one for testing, which is named **staging**.

Developers always have access to the latest version of the repository in their local repository. From time to time, the newest version developed so far it's deployed on staging. As other team members, who are not developers, want to have access and see how the development process is going in the half way of getting a new feature which is already not deployed in the live system, they could access to the staging server, with a relative new version and evaluate the progress to take determine decisions in their work. Hence, the staging server it is mainly used for letting other team members test the current state of the development process towards the version that will be live eventually with the new required functionalities.

Once the new features developed for the web application have been tested by the entire team, including not developers as well, and no bugs are detected, the new version is deployed on the live system and accessible to the users.

Capistrano provides support to multistage with a separate extension. It is needed to install an additional RubyGem which is integrated in the system by typing `gem install capistrano-ext`

To configure this multistage feature it would be necessary to add in "*config/deploy.rb*" these command lines for having the staging and production different stages:

```
set :stages, %w(production staging)
set :default_stage, "staging"
require 'capistrano/ext/multistage'
```

### Code snippet 19: Multistaging configuration for two different stages

Afterwards, under the folder *config/deploy* will be allocated the files *production.rb* and *staging.rb* where the custom stage-specific code will be placed.

Finally, for deploying in the production stage it's needed to type in the console `cap production deploy` and to follow suit for the staging `cap staging deploy` Or `cap deploy`, as staging is the default stage, specified in the line two of the snippet above.

## 4.5. Testing before deploying

It isn't possible to guarantee a completely version deployment which will work without triggering an exception. However, it is possible to enhance the reliability of the process by testing the application before the deployment from the local to the hosted repository and from the hosted repository to the live version which the users access from their browsers.

Once a developer decides to deploy the version which is in the host-based repository, executes `cap production deploy`. Nevertheless, this new version is not deployed instantly, due to it should be ensured to pass tests before having the fresh version on the live system. To achieve this purpose, Jenkins triggers all the existing tests.

- Unit and Functional tests for the server-side.
- Unit tests for the client-side.
- Behavior-Driven tests for both, client and server-side.

In order to deploy the latest version, all the tests should pass. In case of a single test fails, the corresponding code should be fixed, changes pushed to the host-based repository and then deploy the new version again.

**Jenkins**<sup>27</sup> is an open-source continuous integration services tool for software development. It is a server-based system running in a servlet, Java-based server-side web technology. It supports revision control (SCM) such as Git.

---

<sup>27</sup> Jenkins guide: <https://wiki.jenkins-ci.org/display/JENKINS/Use+Jenkins>

## 4.5. Website monitoring system

How a developer could know that the application which runs on the live system doesn't trigger any exception?

Even after all the testing before getting the new version running on live could be something which still could fail. In order to keep track of the exceptions produced in the website application, the system must be monitored.

### TinyMon

TinyMon<sup>28</sup> is a full-stack website monitoring system. While monitoring, it simulates a web browser and performs complex configurable steps.

It doesn't monitor the individual components that build up the website. Instead, it takes the user perspective and tests whether all components are working together to generate the expected browsing experience. It isn't a replacement of software test, but an important addition.

Features included in this framework:

- Fill in login forms and log in using a test account.
- Complete transaction for online shopping.
- Monitor the signup process.
- Monitor external web sites or services the system depends on.

To set this website monitoring tool up for tracking the application these are the steps that should be followed:

- 1) Create a new account.** A new account represents a cluster of websites belonging to the same client. It holds the access rights to sites and Health Checks within sites. A user that is given access to an account can always see all sites and checks within this account.
- 2) Create a new site.** It provides a base URL, so a sub domain would be a new Site unless it's provided a full path in visit steps for Health Checks for a Sites sub domain. The clean way is to treat sub domains like a new Site.
- 3) Create a new Health Check.** It can be chosen either from a few predefined templates or create a first step from the scratch. In the template must be provided a path for the visit step and a string for the check-content step.
- 4) Test drive for the Health Check.** It's just needed to click "Information" and then "Run". After a while the page should be refreshed or click "Check runs" to see how the tests worked out. If tests passed, the check can be enabled.

---

<sup>28</sup> TinyMon: <http://www.tinymon.org/pages/faq>

5) **Integrate it in the application code.** This snippet must be added in the file *config/deploy.rb*:

```
1 task :mark do
2   if stage == 'production'
3     callback_url =
4     'http://mon.tinymon.org/deployments/510478e487b921349e0599bfb64fef45'
5     system "curl -F deployment[revision]=#{current_revision} -F
6     deployment[schedule_checks_in]=3 #{callback_url}"
7   end
8 end
```

**Code snippet 20: Integrate TyniMon in the production stage**

The second line specifies that the callback it is only executed for the production stage, which is the one used for the live system. The fourth line sets the deployment revision for which the callback should be executed.

For sending emails to all team of developers, in *config/environment/production.rb*, it's been included this chunk of code:

```
config.middleware.use ExceptionNotifier,
  :email_prefix => "[Exception] ",
  :sender_address => %{"Exception Notifier" <exception@waymate.de>},
  :exception_recipients => %w{developers@waymate.de}
```

**Code snippet 21: ExceptionNotifier configuration**

These lines specify the subject, the sender and the address to be sent when TinyMon detects an exception.

To set this useful and important tool up, it is quite straightforward and programming skills are not needed at all. Once this framework is completely set up, if an error occurs on the live system, every developer will receive an email with an error description about what it failed.

This functionality is integrated by **Exception Notifier Plugin** for Rails, which provides a mailer object and a default set of templates for sending email notifications when errors occur. By default, the notification email includes the request, session, environment and backtrace. Nevertheless, each of those sections can be customized rendering a partial with the name of the part that wants to be modified (e.g., *\_session.html.erb*) under the folder *app/views/exception\_notifier*. The information available from these partials is the following:

- @controller: the controller that caused the error.
- @request: the current request object.
- @exception: the exception that was raised.
- @backtrace: a sanitized version of the exception's backtrace.
- @data: a hash of optional data values that were passed to the notifier.
- @sections: the array of sections to include in the email.

## 5. APIs data abstraction into a common object

Data that comes from different APIs to represent connections for different means of transportation or companies is provided in different formats, structures and attributes. This is a problem, because having different object models for each API makes the back-end even more difficult to maintain and test. Maintenance gets more complicated due to different object models have to be treated with different logic. Considering that in the future more APIs will be included, this fact implies additional logic for the functionality achieved so far, specifically for the new object model.

Therefore, having a single object model which encompasses into one the data that comes from different APIs presents the following advantages:

- Saves code duplication, due the developer only has to make the new object model fit in the global.
- Functions are more testable. Having different objects the code includes a lot of comparison if-else, which are not so easy to test.
- The project remains better structured.
- The code is cleaner and hence, more readable.

In the old approach trains and flights were different object models, but it's been managed to converge these two different objects into a flexible and consistent single object model. Probably it seems to be easy, because a lot of attributes are shared such as origin, destination, departure, time and so on. The main difficulty was found at the time to create a sub-object for tariffs and prices.

For each API where the data is retrieved, an independent file is needed to parse the response and transform it into a common object model. Regarding the data structure to store all different possibilities and the global connection object model is confidential, so no more further details will be explained.

Waymate offers travel solutions, connecting cities all over Europe and some cities outside the boundaries from the old continent. These connections are covered by train and flights. But, from which APIs does this data come from?

### 5.1. Deutsche Bahn API

**Deutsche Bahn AG (DB)** is the biggest German national railway company. They offer connections by train giving full coverage over Germany and some important cities within Europe. However, destination or origin must be a German city, implying that there is no train connection which links two cities as origin and destination outer from Germany.

DB Bahn has opened its API to Waymate, providing all the details information for different connections to travel in a determine date, from one place to another. To get access to this API Waymate and DB formalized a contract, providing in the first instance access to a testing server. After being developing with the testing server, DB test the application and if



there is no bugs or an incorrect way in the process, such as how to show the conditions, the real API is opened.

For each request is needed to include a user, password and key (provided by DB after signing a contract) in the body of each request and besides two XML files templates filled with the proper attributes to compose the request. To guarantee the encryption of confidential information, such as credit card numbers, the protocol HTTPS is used for sending this information. This protocol adds the security capabilities of SSL/TSL to the standard. The response of this API for a determine connection includes:

- **Train schedules.** In order to connect two different cities, there is not always a direct train, hence, some connections contain more than one train schedule, making one or more stopovers in other cities. Each schedule contains:
  - Arrival and departure time and date.
  - Origin and destination train stations name and EVA (unique train station identifier).
  - Train category and number.
- **A set of schedules** (or a single one if there is a direct train), compose a **connection**. A connection includes the element offer with:
  - Total price.
  - Offer code and its description.
  - Class (1<sup>st</sup> or 2<sup>nd</sup>).
  - Codes for long-distance trains.
  - Types of possible payments.

## 5.2. Ypsilon API

**Ypsilon** is a company which provides access to its Internet Booking Engine. Its API provides flight connections coverage between cities all over Europe, including several airlines. It also can provide some connections from Europe to other continent or the other way around, as well as multi-currency to handle the booking process for these flights.

All the process regarding the contract and starting getting data from a testing server follows the same logic than with DB, so is not necessary to repeat it again.

For sending requests it's needed to include an authorization code in the body, with the name of the company followed by a code provided by Ypsilon, specify a port, a secure port, a host, a secure host and a client certification. It's sent by HTTPS as well, but the encryption it's stronger given the client certification usage.

The response of this API is quite different from the Deutsche Bahn. The response for a connection includes the following attributes:

- **Legs.** As in the case of train connections, it's not possible to offer always a direct flight from a city to another or connections with a stopover could be cheaper, involving taking more than one flight from origin to destination. In a flight connections legs are the different flights that a traveler have to take for the entire journey. The relevant attributes which each leg contains are:
  - Departure and arrival time.
  - Origin and destination ICAO codes.
  - Flight number.
  - A Boolean which specifies if it's an e-ticket.
  - Airline IATA code.
  - Travel class.

**Tariffs**, which contains all the information related with the payment and prices, with a lot of attributes which in the end compose:

- Currency.
- Price.
- Possible ways of payment.

These APIs are pretty much complex than it's explained here, but summarizing, this is the data needed to display the results in the first instance, showing a list of different connection results to the user in a smart way.

## 6. CLIENT-SIDE

A client is an application which runs locally, in a web application case in a web browser, and connects to a server when is necessary for processing and getting data. Nevertheless, the client-side is able to perform operations when data is available.

When operations are executed by the client, without sending data to the server, they should take less time due less bandwidth is employed. Besides, when data is not sent it incurs lesser security risk, however this is not possible always. In Waymate, data is sent from the client to the server-side when is required to request some information to external APIs, because from the client is not possible.

In the following sub-sections the technologies employed for developing the client-side will be described.

### 6.1 HTML

**HyperText Markup Language**<sup>29</sup>, the so-called **HTML**, is the main markup language for displaying web pages in the web browser.

It is written in the form of HTML elements, consisting of tags enclosed in angle brackets such us <div>, <html> and so on. These tags should come commonly in pairs like <span></span>. However, some tags, known as empty elements, are unpaired, as <img>. The browser doesn't display tags, instead they are used to interpret the content of the page.

Hence, these elements form the building blocks of all websites. HTML documents allow images and objects to be embedded. These documents are read and compose into visible web pages by web browsers.

To define the appearance and layout for text and other features, web browser can also refer to Cascading Style Sheets (CSS).

To create an interactive web site, language scripts such as JavaScript can be embedded, which affect the behavior of the HTML web pages.

#### Markup

HTML markup consists of several key components, including elements (and their attributes), character-based data types, character references and entity references.

#### Elements

**HTML elements** compose HTML documents. In their most general form have three components:

- A pair of **tags**, *start tag* and *end tag*, such as <div></div>. However, some tags, known as empty elements, are unpaired, like <img>.

---

<sup>29</sup> HTML: <http://es.wikipedia.org/wiki/HTML>

- **Attributes** are name-value pairs, separated by "=", written within the start tag after the element's name and wrapped by double quotes.
  - The **id** attribute provides a document-wide unique identifier for an element. It is mainly used to identify the element to alter its visual properties by stylesheets or animate, modify or delete its content by means of scripts.
  - The **class** attribute provides a way of classifying similar elements.
  - The **style** is used to assign visual properties to a particular element. It is a better practice to use an element *id* or *class* attributes to select and style from the stylesheets.
  - The **onclick** attribute is used for invoking a function from a script file when element is clicked.
- **Textual and graphical content** between the start tag and end tag, which in the case of the so-called empty elements, are inside the start tag.

## Delivery

The World Wide Web is composed primarily of HTML documents which are delivered from web servers to web browsers using the Hypertext Transfer Protocol (HTTP). For allowing the web browser to know how to handle each document received, additional information called meta data is sent, including the MIME type and the character encoding.

## 6.2. HTML appearance elements

### 6.2.1. CSS

**Cascading Style Sheets (CSS)**<sup>30</sup> is a language used for describing the presentation semantics, the look and formatting, of a document written in a markup language. It is designed to enable separation of document content, written in HTML, from document presentation, including elements such as the layout, colors and fonts.

Via this separation the content accessibility is improved, providing more flexibility and control in the specification of presentation characteristics. It also enables multiple pages to share formatting, reducing complexity and code repetition in the structural content.

CSS specifies a priority scheme to determine which style rules apply if more than one rule matches against a particular element. In this so-called *cascade*, priorities or *weights* are calculated and assigned to rules, so that the results are predictable.

The CSS information could be provided from 3 different sources:

- **Inline styles.** Inside the HTML document, which is not a recommended use.

---

<sup>30</sup> CSS: [http://en.wikipedia.org/wiki/Cascading\\_Style\\_Sheets](http://en.wikipedia.org/wiki/Cascading_Style_Sheets)

- **Embedded style.** Blocks of CSS information inside the HTML itself.
- **External stylesheets.** The most common and recommended practice, given it separates properly the CSS code from the HTML in different and referenced files from the document.

## Syntax

A style sheet consists of a list of *rules*. Each rule or rule-set consists of one or more *selectors*, and a *declaration block*. A declaration-block consists of a list of *declarations* in braces. Each declaration itself consists of a *property*, a colon (:), and a *value*. If there are multiple declarations in a block, a semi-colon (;) must be inserted to separate each declaration.

In CSS, *selectors* are used to declare which part of the markup a style applies to, a kind of match expression. Selectors may apply to all elements of a specific type, to elements specified by attribute, or to elements depending on how they are placed relative to, or nested within, others in the document tree.

*Pseudo-classes* are used in CSS selectors to permit formatting based on information that is outside the document tree. An often-used example of a pseudo-class is *:hover*, which identifies content only when the user 'points to' the visible element, usually by holding the mouse cursor over it.

```
#<id_name> .<class_name>:hover{...}
```

A selector example which selects then all the children elements which contain a determine class under the element with a determine id when the user points over this element with cursor.

## 6.1.2. SASS

**Syntactically Awesome Stylesheets (SASS)**<sup>31</sup> is a scripting language interpreted into CSS, open-source and coded in Ruby. It's included from the server-side as *RubyGem* by `gem install sass`.

SASS extends CSS3 by providing several mechanisms available in more object-oriented languages, such as nested rules, variables, mixins, selector inheritance and some more features, which are not in CSS3. It can monitor the `.sass` or `.scss` file and translate it to an output `.css` file.

Consists of two syntaxes:

1. The *original syntax*, which uses indentation to separate code blocks and newline characters to separate rules.
2. The *newer syntax*, "SCSS" uses block formatting like that of CSS. It uses braces to denote code blocks and semicolons to separate lines within a block.

---

<sup>31</sup> SASS: <http://sass-lang.com/>

The indented syntax and SCSS files are traditionally given the extensions .sass and .scss respectively.

### 6.1.2.1 Advantages of using SASS

Given that CSS presents a large list of limitations, due to it's a styling language, it doesn't allow to the developer to use typical practices used in programming languages, which would save lines of code and will give a better structure to the stylesheets.

New features which SASS introduces to CSS3:

1) **Variable definition** support to reuse colors, sizes and other values without repeating them.

- In addition variables can be modified and combined using the standard **math operations** (+, -, \*, /, %) and useful **predefined functions**.
- **Interpolation**. Variables can be used to be inserted into property names or selectors by using #{}.

```
$common-width: 30px;
$relation: 2;
$position-element: bottom;
#navigation{
  width: $common-width;
  height: $common-width*$relation + 10px;
  border-#{ $position-element}: 1px solid;
}
```

Code snippet 22: SASS. Variable definition example

2) **Nesting** the child selectors within the parent selector. For pseudoclasses, like :hover, the special character & is used as parent selector.

```
#navigation{
  background-color: red;
  div{
    float:left;
    &:hover{
      background-color: blue;
    }
  }
}
```

Code snippet 23: SASS. Nesting classes example

3) **Mixins** allow re-using styles, properties or even selectors, without having to copy and paste them. Mixins are defined using the “@mixin” directive, which takes a block of styles that can be included in another selector using the “@include” directive.

Mixins allows passing arguments as well as function. They can be passed as a variable or normal variable declaration. For passing more than one argument, these should be separated by coma.

```
@mixin rounded($side, $radius: 10px) {  
  border-#{ $side }-radius: $radius;  
  -moz-border-radius-#{ $side }: $radius;  
  -webkit-border-#{ $side }-radius: $radius;  
}
```

**Code snippet 24: SASS. Mixin function example**

- 4) File importing.** The “*@import*” directive allows breaking the styles up into multiple stylesheets. It avoids code duplication when using common variables or mixin classes in different stylesheets. Files that are meant to be imported have to begin with underscore by convention and it won’t necessary to specify the extension when it’s imported.

To illustrate how it works, lets say that the previous chunk of code is within a file named “*\_rounded.scss*”. Thus, in the file where it’s going to be used it would be added this line: `@import “rounded”;`

## 6.3. Dynamic web-site

With HTML and CSS static web-sites can be built up, however there is no logic included and limited interaction. In order to include data processing with its corresponding logic and user interaction, the following technologies are included in the project.

### 6.3.1. JavaScript

**JavaScript (JS)**<sup>32</sup> is a **prototype-based scripting** and **multi-paradigm** language, supporting object-oriented, imperative and functional programming styles. It's **dynamic, weakly typed** (supporting implicit typing conversion and providing a lot of flexibility when operating with different types) and has **first-class functions**. Its syntax is influenced by C and Java, but both are actually unrelated and have very different semantics.

JavaScript doesn't have classes, instead uses prototypes, which can simulate many class-based features.

It's mainly used for **programming client-sides** for web pages and applications and can respond to user action quickly, making an application more responsive.

JavaScript is implemented as part of a Web browser, which works as a run-time environment itself, providing objects and methods by which scripts can interact with the "outside world" in order to give enhanced user interfaces and dynamic websites. The functions are embedded in HTML pages and interact with the Document Object Model (DOM), cross-platform and language-independent convention for representing and interacting with objects in HTML, XML and XHTML documents. In addition, is able to detect user action which HTML cannot do by itself, such as individual keystrokes.

### 6.3.2. AJAX

**Asynchronous JavaScript** and XML, commonly known as **AJAX**<sup>33</sup>, is a group of interrelated web development techniques used on the client-side to create asynchronous web applications. Therefore, web applications can send data to, and retrieve data from, a server asynchronously (in the background), without interfering with display and behavior of the existing page. Data can be retrieved using the XMLHttpRequest object, however, despite the name, the format often used is JSON.

This technology enhances the user interaction along the application, avoiding unnecessary full pages reloads, due to XMLHttpRequest object provide a method for exchanging data asynchronously between browser and server.

---

<sup>32</sup> JavaScript: <http://en.wikipedia.org/wiki/JavaScript>

<sup>33</sup> AJAX: [http://en.wikipedia.org/wiki/Ajax\\_\(programming\)](http://en.wikipedia.org/wiki/Ajax_(programming))



### 6.3.3. jQuery

**jQuery**<sup>34</sup> is a multi-browser JavaScript library designed with the intention to simplify the client-side scripting. It's the most popular and used nowadays, due it allows the creation of powerful dynamic web pages and web applications.

Its syntax makes easy to navigate a document, select DOM elements, create animations, handle elements and develop Ajax applications. It also provides capabilities to extend its functionality creating plug-ins on top of the JavaScript library, so developers can create abstractions for low-lever interaction, advanced effects and animations.

To **include** this JavaScript library, or another one, into the project it's necessary to include this line between the head tags.

```
<head>
  <script type="text/javascript" src="jquery.js"></script>
</head>
```

Code snippet 25: JavaScript library inclusion from a HTML template

The way to introduce a jQuery function is to use the `.ready()` function. Then, when the page is loaded, the code within this section will be executed. Functions will be defined out of it.

```
$(document).ready(function() {
  // code goes here
});
```

To access and manipulate DOM nodes jQuery provides the `$` function. It is called with a CSS selector string for the class or id of the DOM element, resulting an object reference after the matching.

```
$('#navigation .element').addClass('selected');
```

This line add the class "selected" to all children DOM elements under the one with id "navigation" which contain the class "element".

This library provides a large set of methods for:

- **Selecting DOM elements** and manipulate them.
- **Getting and setting attributes for input elements** such as checkboxes, input fields, radio buttons and drop down lists.
- Getting and setting the **CSS styling for HTML elements**.
- **Events methods** which register behaviors to take effect when the user interacts with the elements and manipulate those registered events.

---

<sup>34</sup> jQuery: <http://en.wikipedia.org/wiki/Jquery>

- **Visual effects** for adding animation to a web page, from simple standard animations to sophisticated custom effects.
- **AJAX** capabilities, to allow loading data from/to the server without browser page refresh.

There exists a standard bundle of jQuery library to interact with UI elements. In this application, the **Draggable** plugin, which makes selected elements draggable by mouse is included.

Regarding widget plugins, the ones included are:

- **Datepicker**, a configurable and customizable calendar to select specific dates or range dates.
- **Slider**, which makes selected elements into slider and provides various options such as multiple handles and ranges.

### 6.3.4. Backbone

**Backbone**<sup>35</sup> is a JavaScript framework based on the **model-view-controller (MVC)** application design paradigm. It is designed for developing **single-page web applications (SPA)** and for keeping various parts of web applications synchronized. It gives structure to web applications by providing model with key-value binding and custom events, collection with a complete and useful API of functions, views with declarative event handling, and connects it all to the server API through a **RESTful JSON** interface.

With Backbone, JavaScript files must follow a specific organization, dividing placing them under models, collections, helper libraries and views folders. Having a good files organization will help to find files faster and maintain them.

```

app/
  assets/
    images/
    javascripts/
      collections/
      lib/
      models/
      views/
      app.js
      app_router.js
      application.js

```

Code snippet 26: Backbone. JavaScript files organization

#### 6.3.4.1. Importance of using a MVC framework in the front-end

The entire application involves a lot of JavaScript code and this generally implies to end up with tangled piles of jQuery selectors and callbacks, all trying to keep data in synchronization between the HTML UI, JavaScript logic and the server database. Nevertheless

<sup>35</sup> Backbone: <http://backbonejs.org/>

it is not recommended to tie the data to DOM elements and for rich client-side applications, a more structured approach is helpful.

Backbone represents the data as Models, which can be created, validated destroyed and saved to the server. It also provides Views that displays the model's state and can be notified of the change of the model in order to be able to respond and re-render the view for the new model data. Hence, in the end it won't be necessary to write the code that looks into the DOM to find an element with an id and update the HTML manually, given that when model changes, the views updates by themselves.

The advantage of having a single-page web application is to provide a more fluid user experience, due to changes are performed loading new code on demand from the web server, driven by user actions. Continual page redraws disrupt the user experience due to the network latencies cannot be hidden. The page doesn't automatically reload during user interaction with the application, therefore no unnecessary data re-transmission it is performance. The technique which allows this continuous and fluid user experience is **AJAX**, allowing asynchronous communication between client and server.

#### 6.3.4.2 Comparison with similar frameworks

A lot of MVC frameworks for JavaScript have come up lately. The most famous frameworks, in addition to Backbone, are Spine, AngularJS and Ember.

Actually there is no best choice among these 4 alternatives, due they offer the same interesting features for such kind of framework, but in with different syntax.

So, how to select the best choice?

Waymate developers were asking through the Rails community in order to find a unanimous answer. However, it didn't happen, different developers were promoting different frameworks.

The second idea was to ask in a Ruby conference, where a lot of prestigious Ruby on Rails programmers use to attend. It happened the same as in the Rails community.

So, finally, Waymate developers decided to choose Backbone for three reasons:

- It's well documented.
- The client-side of prestigious projects have been built up using this framework.
- It's the most popular among all the alternatives.

### 6.4.3.3. Backbone components

#### 6.4.3.3.1 Backbone.sync

`Backbone.sync` is the function that Backbone calls every time it attempts to read or save a model to the server. By default, it uses `(jQuery/Zepto).ajax` to make a RESTful JSON request and returns a jQuery XMLHttpRequest object by `$.ajax (jqXHR)`.

```
sync(method, model, [options])
```

- The **method** could be either *create, read, update or delete*.
- The **model** to be saved or collection to be read.
- The **options** are success, error callbacks and other jQuery request options

This method could be overridden globally as `Backbone.sync`, or at a finer-grained level, by adding a `sync` function to a Backbone collection or to an individual model for using a different persistency strategy such as WebSockets, XML transport or Local Storage.

The default implementation sends up a request to save a model, its attributes will be passed, serialized as JSON and sent in the HTTP body with content-type *application/json*. When a JSON response is returned, send down the attributes of the model that have been changed by the server, and need to be updated on the client. Further on, it will be explain how could be fulfilled by means of a `Backbone.Model`.

The default **sync** handler maps CRUD to REST like so:

- **create** -> **POST** /collection
- **read** -> **GET** /collection[/id]
- **update** -> **PUT** /collection/id
- **delete** -> **DELETE** /collection/id

#### 6.4.3.3.2 Backbone.Events

**Backbone.Events** is a module that can be associated to any object, giving the ability to bind and trigger custom named events. They don't have necessarily to be declared before they are bound and can take passed arguments.

- A **callback function could be bound to an object**, so it will be invoked whenever the corresponding event is fired.

```
object.on(event, callback, [context])
```

To supply a context value for this when the callback is invoked, in the third argument `[context]` should be passed this.

Callbacks bound to the special all events will be triggered when any event occurs and are passed name of the event as the first argument.

- **Remove a previously-bound callback function** from an object:

```
object.off([event], [callback], [context])
```

If no context is specified, all of the versions of the callback with different contexts will be removed. If no callback is specified, all callbacks for the event will be removed and obviously if no event is specified, all event callbacks on the object will be removed.

- **Triggering callbacks** for the given event, or space-delimited list of events. Furthermore, subsequent arguments to trigger will be passed along the event callbacks separated with comas.

```
object.trigger(event, [*args])
```

### 6.4.3.3 Backbone.Model

Models contain the interactive data as well as a large part of the logic surrounding it such as conversions, validations, computed properties and access control. A Backbone.Model provides a basic set of functionality for managing changes. Nevertheless, to create a new one should be extended with their own specific methods.

To create a customized Model class, Backbone.Model should be extended and instance **properties** provided. Hence, the definition of a new model with its own functions should look something like this:

```
var name_model1 = Backbone.Model.extend({
  initialize: function(){...},
  name_function1 : function(){...},
  name_function2: function(param1,param2,..){...}
})
```

**Code snippet 27: Backbone. Model definition**

Accordingly, extend correctly sets up the prototype chain, so subclasses created with extend can be further extended as far as a developer wants.

```
var name_model2 = name_model1.extend({...})
```

When a model instance is created, optional **classProperties** can be attached directly to the constructor function.

```
new Model([attributes])
```

These attributes could be a simple attribute, an array or even another model, which will be set on the model in which have been passed. If an initialize function has been defined, it will be invoked when the model is created

```
var instance_model = new name_model2({attribute1: 'value1', attribute2: 'value2',...})
```

## Model functions interface

- **Getting the current value of an attribute** from the model.  
`model.get(attribute)`
- **Setting a hash of attributes** (one or many) on the model.  
`model.set(attributes, [options])`
- If an **attribute is set to a non-null or non-undefined value**, this function returns true.  
`model.has(attribute)`
- **Remove an attribute** by deleting it from the internal attributes hash. Fires a *change* event as well.  
`model.unset(attribute, [options])`
- **Remove all attributes from the model.**
- The defaults function (or hash) can be used to **specify the default attributes** for the model when creating an instance of the model.  
`model.defaults` or `model.defaults()`
- For **validating a model** this method should be overridden. This method will be called always before setting or saving a model. If it fails it returns an error, otherwise anything.  
`model.validate(attributes)`
- **Check if the model is valid.**  
`model.isValid()`
- **Get the relative URL** where the model would be located on the server.  
`model.url()`
- **Specify a urlRoot** if the model it's placed outside of a collection.  
`model.urlRoot` or `model.urlRoot()`
- **Parse the model data** when the model is fetched or saved. From the server side in Rails, it exists the implementation `to_json` for including model's attributes under a namespace.  
`model.parse(response)`
- **Return a new instance** of the model with the **same attributes**.  
`model.clone()`
- **Check if the model** has already been **saved on the server**. It returns false in that case, otherwise true.  
`model.isNew()`

- **Manually trigger the *change* event.**  
`model.change()`
- **Check whether the model has changed since the last *change* event.**  
`model.hasChanged([attribute])`
- **Retrieve a hash of the model's attributes that have changed.**  
`model.changedAttributes([attributes])`
- **Get the previous value of a changed attribute.**  
`model.previous(attribute)`
- **Get a copy of the model's previous attributes.**  
`model.previousAttributes()`
- **Return a copy of the model's attributes for JSON stringification.**  
`model.toJSON()`
- **Reset the model's state from the server** by delegating to `Backbone.sync`. It is useful either if the model never has been populated with data or for ensuring that it has the latest server state.  
`model.fetch(model, response)`
- **Save a model to the database server** (*persistence layer*), by delegating to `Backbone.sync`. The hash attributes should contain those desired to change. If the model has a `validate` method, and validation fails, the model won't be saved. If the model is `isNew`, the save will be a *create* (HTTP POST) and if it already existed will be an *update* (HTTP PUT).  
`model.save([attributes], [options])`
- **Destroys the model on the server** by delegating an HTTP DELETE request to `Backbone.sync`.  
`model.destroy([options])`

If any of the attributes change the models state, a *change* event will be triggered, unless `{silent: true}` is passed as an option.

#### 6.4.3.3.4 Backbone.Collection

Collections are ordered sets of models. It could be bound *change* events when any model in the collection has been modified, listen for *add* and *remove* events, *fetch* the collection from the server and use a full an Underscore.js method interface.

Any event triggered on a model within a collection will also be triggered on the collection directly, allowing listening for changes to specific attributes in any model which the collection contains.

- To **create a Collection**, Backbone.Collection should be extended, providing instance properties, as well as optional classProperties, as in the Backbone.Model creation, to be attached directly to the collection's constructor function.  
`Backbone.Collection.extend(properties, [classProperties])`
- The **model class** which the collection will contain should be specified.  
`collection.model`
- When **creating a Collection**, the array of model could be passed as an argument. If the **initialize** function is defined, it will be invoked as well.  
`new Collection([models], [options])`
- **Add a model or an array of models** to the collection. Models will be inserted regarding the definition of the comparator function.  
`collection.add(models, [options])`
- **Add a model or an array of models at the end** of the collection or **at the beginning** of the collection respectively.  
`collection.push(model, [options])` or `collection.unshift(model, [options])`  
These three latter methods fire the event *add*.
- By default there is no **comparator function** on a collection. If this function is defined within the collection, it will be used to maintain the collection in sorted order.  
`collection.comparator`
- Force a **collection to re-sort** itself when it is invoked.  
`collection.sort([options])`
- **Remove a model or an array of models** from the collection.  
`collection.remove(models, [options])`
- **Remove and return the last** or the **first model** of a collection respectively.  
`collection.pop([options])` or `collection.shift([options])`  
The latter three methods fire a *remove* event.



- Return an array of all **models which match the passed attributes**.  
`collection.where(attributes)`
- **Set the URL** to reference its location on the server.  
`collection.url` or `collection.url()`
- **Get a model from the collection**, specified either by **id**, **index** or **client id**. The latter is useful when the model hasn't been saved on the server side.  
`collection.get(id)`, `collection.at(index)` and `collection.getByCid(cid)`
- **Raw access** to the JavaScript **array of models** inside the collection.  
`collection.models`
- Return an **array containing the hash attributes of each model** in the collection.  
`collection.toJSON()`
- **Fetch the default set of models** for this collection from the server resetting the collection. It delegates to Backbone.sync under the covers for custom persistence strategies. It also calls `parse` automatically to parse, which returns the array of model attributes to be added to the collection.  
`collection.fetch([options])`
- **Create a new instance of a model** within a collection.  
`collection.create(attributes, [options])`
- Replace a collection with a **new list of models**.  
`collection.reset(models, [options])`
- 28 iteration functions provided by Underscore.js such as *forEach(each)*, *isEmpty*, etc.

#### 6.4.3.3.5 Backbone.Router

Backbone.Router provides methods for **routing client single-pages and connects them to actions and events**. For browsers which don't support yet History API, the Router handles graceful fallback and transparent translation to the fragment version of the URL. The History API provides permalinks to use standard URLs page.

During page load, after the application has already finished creating all its routers, should be called `Backbone.history.start()`, OR `Backbone.history.start({pushState: true})` to route the initial URL.

- Create a custom router class. Actions which triggers when certain URL fragments are matched must be defined, as well as a routes hash that pairs routes. Here is an example:

```

var Workspace = Backbone.Router.extend({
  Routes: {
    "help": "help", // #help
    "search/:query": "search", // #search/whatever
    "search/:query/p:page": "search" // #search/whatever/p5
  }
})

```

**Code snippet 28: Backbone. Router definition example**

- The routes hash **maps URL with parameters to functions** on the router. Routes can contain parameters parts, `:param`, which match a single URL component between slashes and splat parts `* splat`, which can match any number of URL components. The previous example shows how routes match dynamically with the defined within the Routes section.
- To **save a point** through the entire application **as URL** when is reached, **navigate** should be called in order to update the URL. If the route function is wanted to be called, `{trigger:true}` must be passed as an option. For not creating an entry in the browser's history, `{replace:true}` will be passed. An example which shows how to do invoke navigate:  
`App.navigate("route/sub_route", {trigger:true, replace:true})`

### 6.4.3.3.6 Backbone.History

History serves as a global router to handle *hashchange* events or *pushState*, match the appropriate route, and trigger callbacks. If Routers are used with routes, an instance of Backbone.history will be created, therefore it's not needed to create it explicitly.

- To **start the history**, `Backbone.history.start()` must be called after all Routers have been created. Then, it will start to monitoring *haschange* events and dispatching routes.

If the application is not being served from the root url `/`, the root should be specified when the history is started.

```
Backbone.history.start({pushState:true, root:"/example/search"})
```

### 6.4.3.3.7 Backbone.View

**Views are meant to be associated to a model.** The main idea to update automatically the view when the model changes. This could be done by binding to the view's render function the model's *change* event. A page could contain different views, so it just will be necessary to re-render the one that the model associated to has changed, instead the whole page.

- As all Backbone objects, to **create a custom** one the **view class**, `Backbone.View.extend(properties, [classProperties])`, have to be overwritten as the example follows:

```
Var ExampleView = Backbone.View.extend({
  className: "",
  id: "",
  tagName: "",
  events: {
    "click .item" : "add_item"
  }
  render: function(){...},
  add_item:function(){...}
})
```

Code snippet 29: Backbone. View definition template

- All views have a DOM element at all times whether they've already inserted into the page or not, which is named as `el` property.

`view.el`

- **\$el** is a cached jQuery object for the view's element.

`view.$el`

- **Attributes** are the set of HTML DOM element attributes on the view's `el`. If they are not defined `el` will be an empty div.

- `tagName` assigns the HTML element type which will contain all the content returned by this view.
- `className` gives the name of the class to this HTML element and `id` the id.

- **Render** will draw all the content from model the view template, updating `this.el` element with the new HTML.

`view.render()`

There exist different libraries for **displaying templates**. The one employed in this project is **Mustache.js**. An instance of how invoke this function:

```
Mustache.render(this.template(), { this.model.toJSON()})
```

By convention, a function such as `this.template()` will be used and it will be defined within the view class something like this:

```
template: function(){
  return $('#desired_id').html();
}
```

The second parameter that receives is the model in JSON format, to access the variables from the HTML template. For accessing variables, the name of the variable must be surrounded with curly brackets. If this model is passed: `{name : "John"}`, in the template could be access doing the following:

```
<div id="author_{{name}}" class="author">{{name}}</div>
```

Hence, in both instance which `{{name}}` appears, will be substituted by "John".

- **Remove the view** from the DOM.

```
view.remove() or $(view.el).remove()
```

- **Delegating events.** To provide declarative callbacks for DOM events within a view. If events hash is not passed directly, it uses `this.events` as the source. The callback may be the name of a method of the view or a direct function body. This function is called by default in the View's constructor binding the events defined in the corresponding section.

```
delegateEvents({"event selector" : "callback"})
```

- **Undelegating events.** It removes all of the view's delegated events. It could be used to disable a view from the DOM temporarily.

```
undelegateEvents()
```

### 6.3.5. Testing the client-side

For testing the client-side, for each testable function and regular expressions has been created its corresponding set of unit tests. Behavior-Driven Development testing which involved client and server side together it's already described in the server-side testing section. However, BDD where the only the client code is involved, it's been implemented independently.

A single framework is employed for unit testing and BDD exclusively the client-side code:

**Jasmine**<sup>36</sup> is a BDD framework for testing JavaScript code. It doesn't depend on any other JavaScript framework and it doesn't require a DOM. It includes the following features:

- **Suites for describing the tests.** A test suite begins with a call to the function `describe`, which takes a string as a title for a spec suite and the function that implements the suite. Within this functions could be defined more than one Spec.

```
describe("A suite", function() {  
  ...  
});
```

- **Specs** are defined by calling the function `it`, taking a string as title and a function. Inside this function context could be specified one or more expectations.

---

<sup>36</sup> Jasmine: <http://pivotal.github.com/jasmine/>

```
it("and has a positive case ", function() {
    expect(true).toBe(true);
});
```

- **Expectations** are built with the function `expect`. They are assertions which can be true or false. If an expectation is evaluated to false, the spec will fail.  
`expect(a).not.toBe(null);`
- **Matchers** implement boolean comparison between the actual value and the expected value. Jasmine has a rich set of matchers included such as, `toEqual()`, `toMatch()`, `toBeTruthy()`, `toBe()`, etc. Furthermore, any matcher can evaluate to a negative assertion by adding `not` between the expectation and the matcher. There is also the ability of writing custom matchers for specific assertions not included.

A suite case containing all the significant test cases, specified in the 4 Specs defined:

```
describe("is invalid month", function() {
    it("zero is not a valid month", function(){
        expect(methods.IsInvalidMonth(0)).toBeTruthy();
    });
    it("one is a valid month", function() {
        expect(methods.IsInvalidMonth(1)).not.toBeTruthy();
    });
    it("twelve is a valid month", function() {
        expect(methods.IsInvalidMonth(12)).not.toBeTruthy();
    });
    it("thirteen is not a valid month", function() {
        expect(methods.IsInvalidMonth(13)).toBeTruthy();
    });
});
```

**Code snippet 30: Test suite definition with 4 Specs**

For avoiding any duplicated setup and teardown code, Jasmine provides:

- `beforeEach` function, called once before each Spec. It's used for defining the initialization code.

```
beforeEach(function() {
    $('body').append('<div id="app"></div>');
});
```

- `afterEach` function, invoked after each Spec. It's meant to reset variables before executing the next Spec.

```
afterEach(function() {
    $('#app').remove();
});
```

## Backbone testing

For Backbone Views and Event testing **mock objects** support is needed. A mock object is a simulated object which imitates the behavior of a real object in a controlled way. Mocks are usually created for testing the behavior of some other object, so it wouldn't be necessary to employ the complete functionality of the object which will be emulated.

Jasmine itself doesn't include these mock objects, thus another library inclusion is needed. For this purpose, a gem named Rspec which supports objects mocking is included in the Rails framework.

**Rspec** contains its own mocking framework and its **domain-specific language (DSL)**, or Ubiquitous Language in this context, resembling a natural language specification. To get this gem embedded it would be necessary to execute `gem install rspec`. It will install rspec and its runtime dependency gems, `rspec-core`, `rspec-expectations` and `rspec-mocks`.

## Testing before pushing

Before pushing a new version to the hosted repository, a developer should run explicitly from the console a Jasmine server `rake jstest` and open in the browser this url `localhost:8888`. This will execute a set of unit tests written by the developer for testing the JavaScript code which is executed on the client side. If tests don't fail, it's the moment to push the changes. It's not an automated process, hence a developer must remember to follow it.

## 7. Waymate. How it works

Once all the technologies which build up this application it's time to explain the process and present some features included on the web client-side.

It's been considered necessary to explain the model located on the server-side to have a better understanding about the data which is processed in both sides, as well as the way of obtaining this data.

The entire work-flow is fully covered in this section for the client and server-side. All the functionalities and the purpose for each page along the process and its sections, accompanied by the corresponding pictures, are described.

### 7.1. Object models on the server-side

Waymate includes a large amount of Ruby models to make the application works. Models can be separated in 4 different classes:

**1) Imported databases.** When the data is retrieved from the APIs, instead of providing all the detailed information, the value returned is a unique code. In the back-end, this code is looked up in the corresponding object table and the row which matches is returned as an object model. To get the desired object for the following list of models, the unique code is:

- **Airline**: the IATA code.
- **Airport**: airports use IATA code as well as a unique identifier.
- **Country**: depending on the API where the data comes from, it will be used either IATA or ISO3166.
- **Dbstation**: EVA code.

In Rails, by using model helpers, such as `<Model>.find_by`, is pretty easy to retrieve the proper row in the database. For example, to retrieve a determine airline given its IATA code, and whose attribute name in the database is `IATA_code`:

```
Airline.find_by_IATA_code('AB')
```

These 4 databases were provided in CSV format, where each row, or instance, is defined in a line with the attribute values coma separated and in plain text. To insert each of these CSV as a database table, the following steps were executed:

- a) Create a migration creating the table with all the attributes or columns which will be filled.
- b) A Ruby script file extracted the relevant value attributes and created the lines to insert all the rows into the database, as well as its deletion and printed out into different files.

- c) Once, all the insertion lines were printed in a file were copied inside the empty self.up function and the same for the deletion lines within the self.down function.

```
class InsertAirlines < ActiveRecord::Migration
  def self.up
    Airline.create(airline_id: 214, name: "Air Berlin", alias:"",
iata_code:"AB", icao_code:"BER", callsign:"AIR BERLIN", country:"Germany")
  end
  def self.down
    Airline.delete_all(:airline_id => 214)
  end
end
```

Code snippet 31: Migration file example which introduces or removes an airline

In this snippet is shown how to insert and delete a single airline by a migration.

- d) Finally the migration were executed by `rake db:migrate`

2) **Request models.** These kinds of models are actually created and populated retrieving the information that the user has introduced on the client-side, as Backbone models. These models contain the following information:

- Search: contains the attributes for making a request to the APIs in order to get a list of connections from one city to another for a specific traveling day, and in case of a return trip is selected, for this day as well. It also contains traveler models. It's stored for creating user searches statistics.
- Traveler: has information for a single traveler, such as age and train card options. More than one traveler can be associated to a search
- Customer: contains the personal information when a user fills in the form for booking tickets, such as name, address, payment method, etc. This model is stored when the user is logged in and fills the booking form. So, next time, if the user is logged in, the form will be populated automatically.
- CreditCard: credit card information. Obviously is not stored, hence as soon as the request for buying tickets is sent, this model is destroyed.
- Order: contains a Customer, a CreditCard and one or more Traveler and Leg models. Customer, Travelers and Legs are stored in the database composing an Order model. Obviously, CreditCard information with all the confidential information is not stored.

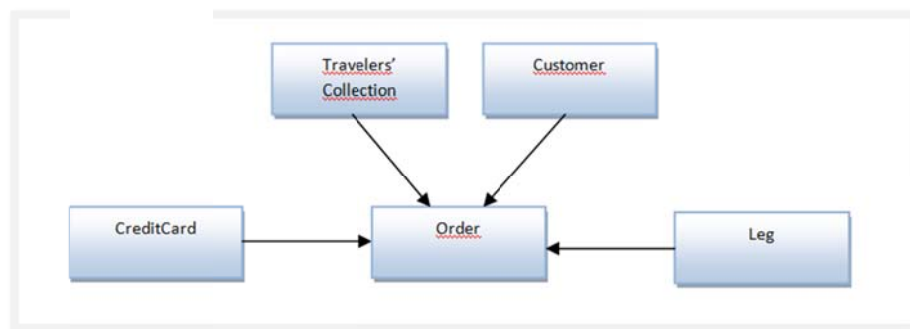


Image 6: Hierarchy of object models which compose an Order



### 3) Result connections

- **PriceContainer**: contains all the possible **Prices** for a single **Leg**, due sometimes are special prices included. **Price** model contains the prices with its currency and type.
- **SchedulePoint**: containing its name, type of station or airport, coordinates and date and time when the user should reach this point along the journey.
- **Schedule**: this model is composed by two **SchedulePoint** objects and the duration for traveling from one to another.
- **ModeOfTransport**: inside this model can be found the mean of transportation, the carrier code and flight number (for flights), the sub-mode of transportation and number (for trains and public transport supported).
- **SubLeg**: for representing the information to travel from one point to another, containing a **Schedule**, a **ModeOfTransport** and a **PriceContainer** models.
- **Leg**: represents the information to travel from a point to another with the same mean of transport and can contain more than one single **SubLeg**. The meaning of containing more than one **SubLeg** is because there exists at least a stopover in this travel.
- **Tour**: composed by one or more **Leg** objects, representing the whole journey for the origin and destination specified by the user.

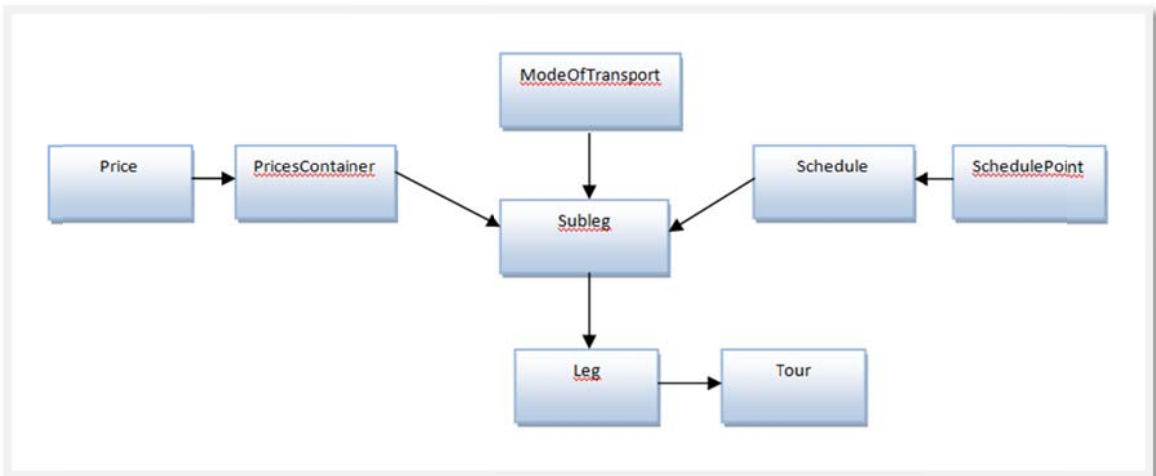


Image 7: Hierarchy of object models which compose a Tour

#### 4) User models.

- User, which contains the username, all the necessary attributes to encrypt the password, the role (normal or admin user).

It also includes references to a Customer model, containing all the information for filling automatically the booking form, without the CreditCard information and for each time he has purchased a travel in Waymate, an Order model.

- UserSession, model which corresponds to access restriction functionality.

These two models are created for storing safely the confidential information for user identification, as the user password.

## 7.2 Workflow along the entire process

The workflow defines:

- The data flow between client-server and server-API.
- Communication between different views in the same or different page via event triggering.
- Page navigation instead of paging reloads due to AJAX usage by Backbone.

It was considered appropriate to include the following information under this section as well:

- Client-side Backbone Model objects and Views.
- Google technologies employed, giving the details, upsides and downsides of using them.
- How validation works in the booking form

## 7.2.1. Landing page

Before launching Waymate as a public application, a private version, which is restricted to some users in order to get some feedback and improve some aspects, will be launched. This is a temporary page to inform users about what Waymate will present as travel solution platform.

This will be the landing page when users access to [www.waymate.de](http://www.waymate.de).

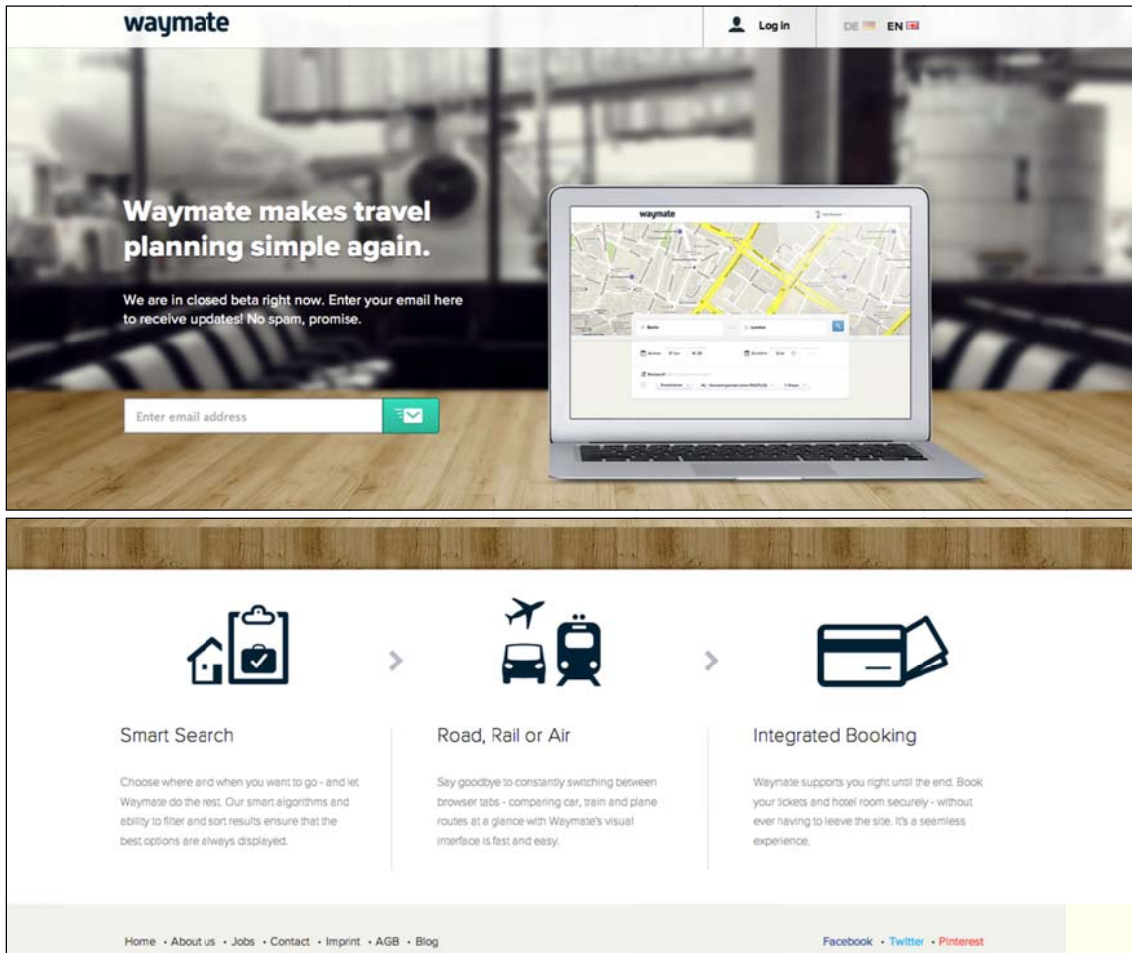


Image 8: Landing page

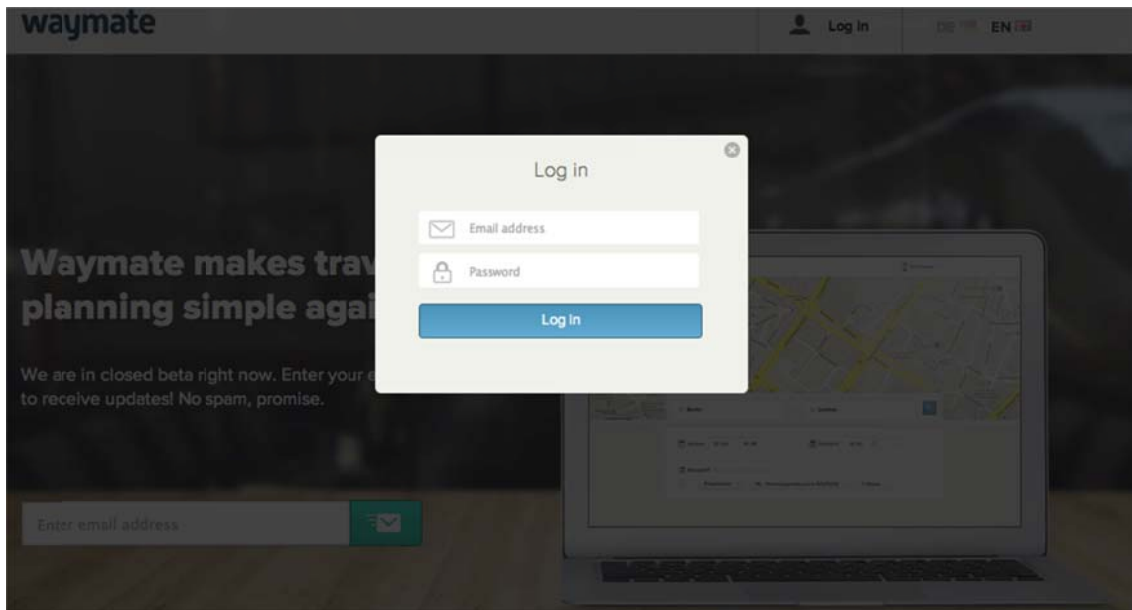
This page includes two functionalities:

- 1) Users who want to be informed about the state and updates about Waymate can subscribe to receive emails with this information.



Image 9: Email subscription

- 2) Testing users and team components can log in for accessing the application. A user which doesn't obtain a username and password provided by the company won't be able to go further than the landing page, because there is no possibility yet to sign up.



**Image 10: Log in window in landing page.**

Once the team has got feedback from the testing users, the improvements considered as critical or important will be implemented by developers. Nevertheless, the entire team is already aware that design issues, interface interaction and additional functionality features are still missing or need to be improved.

Hence, the following pages, which belong to the application and will be explained one by one, are not accessible yet to the public.

## 7.2.2. Search page

On the search page is where the traveler will start planning his journey. This page is composed by two different Backbone views which update each other via event triggering.

- Map view, where the origin and destination markers are displayed. This view will be preserved along the entire process undestroyed. This is possible due there is no page reload along the application process. The map will show more accurate information for connections in advanced stages.
  - Search view, where the user set all the attributes required for searching connections from a city to another for specific days and a determined number of travelers.
- There are 3 differentiated steps within the Search view:

### 1) Search for origin and destination cities or address.

In order to facilitate this process, when the user types something within the corresponding input fields, a list of suggestions shows up, preventing the user to type the complete name he is searching.

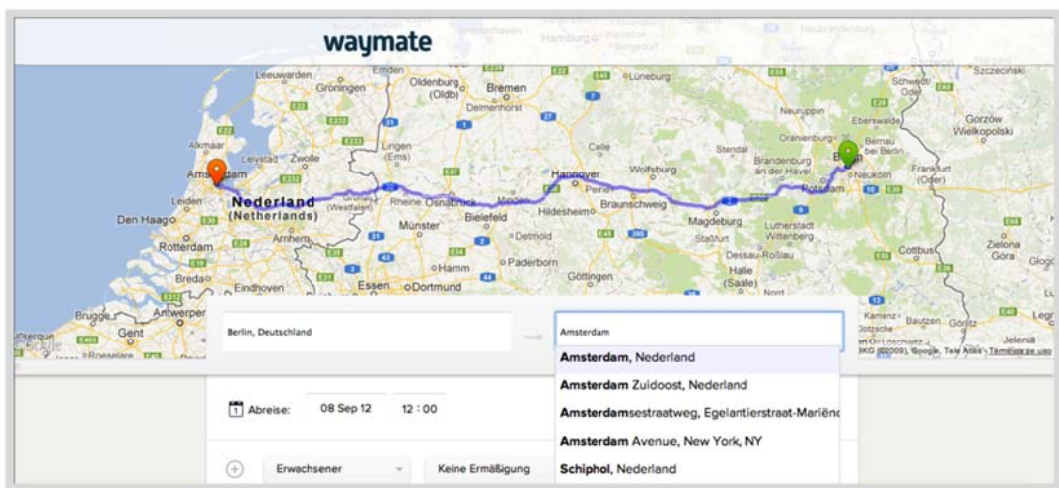


Image 11: Search page. Google Places auto-completion.

## Google Places auto-complete<sup>37</sup>

It can be said that is the best technology found so far for this purpose, given its fast response and accuracy displaying suggestions for cities and address. It's actually the auto-completion power in Google maps website embedded in this application.

It is a library which handles asynchronous requests to Google Places API and displays a suggestions list, which changes each time the input field content changes, either typing or deleting characters. Each time the suggestions list changes, due the input field has changed, means that an asynchronous request has been sent, received, parsed and displayed.

<sup>37</sup> Google Places auto-complete: <https://developers.google.com/places/documentation/autocomplete>

Once the user selects an option in the suggestions' list, another request is sent to Google Places API. This time the response is an object containing all the details about the place selected in JSON format, such as coordinates, name, place's type, etc.

In the end, the most important attributes are the coordinates, which are used to get the closest airports and trains stations which will be made the request in order to get connections.

## Geocoder<sup>38</sup>

When the user presses the search button and hasn't selected an option in the list, but has introduced the name of a city, either for origin or destination, what does it happen?

A request to the **Geocoder** service is done in order to obtain the coordinates and more details associated to that place. This service is also supported by Google and it also provides the **Geocoder reverse** functionality, which passing the coordinates in the request, it returns in the response the place details, such as name, address, etc.

This service guarantees always to obtain the coordinates from the name of the place introduced in the input field.

## Comparing auto-completion alternatives

The first technology employed for auto-completion was **Geocoder auto-complete**, which is not the same service than **Geocoder**. These technologies are powered by Google, nevertheless, the latter presented some important lacks which are solved using Google Places auto-complete:

- The priority of showing address suggestions prevails over cities. Travelers will search mainly for cities instead of address, therefore this was a problem for Waymate.
- When a few characters were introduced, the suggestion list hardly displays a few results and without any accuracy. As less characters the user has to introduce, the best experience will enjoy, due to the fast and easy process.
- When a city is searched uses to appear in the suggestions' list almost after typing the entire name.

The only advantage which Geocoder auto-complete presented was its better performance and accuracy, after having introduced more than 5 characters, for displaying address in the suggestions' list.

---

<sup>38</sup> Geocoder: <https://developers.google.com/maps/documentation/javascript/geocoding>

## Google maps<sup>39</sup>

The map plays an important role along the process in the application. The reason is because when traveling there exist two main dimensions, space and time.

Including the map, and showing the origin and destination after being selected, gives to the user a visual feedback, representing the space dimension.

At this stage, the user can play around with the map, selecting his origin or destination by clicking over the map or modifying it by dragging the markers after they have shown up. For both actions, the marker obtains the coordinates where the marker is placed in the map. Hence, having the coordinates, a request to reverse Geocoder is done, getting the details of the place and updating its corresponding input field, either for origin or destination, with the name or address of the place.

### How to use Google services

For using the services explained previously, it will be required to add between the `<head></head>` html tags the following line:

```
<script type="text/javascript"
src="//maps.google.com/maps/api/js?libraries=places,geometry&rankby=prominence"></script>
```

The source without any parameters gives access to Geocoder and Google maps services. More services are available, but not used in this application.

The parameter `libraries` with the value `places` includes Google Places auto-complete and Google Places services. In addition, with the value `geometry` adds a library to calculate the distance between two points, having into account that the Earth is a sphere.

The parameter `rankby=prominence` means to get suggestions by importance when using Google Places auto-complete.

### Google services downsides

Waymate functionality depends on Google services. Functionalities which are crucial along the process in the application, as getting coordinates, rely on these services. If one day, Google services are not available, the application won't work.

Additionally, for business applications it's required to get a business account. Before getting this account, the price of each request and the services which will be accessible must be negotiated and specified in a contract.

There two important downsides, but in the market, services as powerful as the ones which Google provides don't exist. An exception could be done regarding maps services, mentioning alternatives such as Nokia maps, OpenStreetMaps, etc. But Google provides additional features to the maps which eventually could be interesting to add.

---

<sup>39</sup> Google Maps API: <https://developers.google.com/maps/documentation/javascript/reference>



## 2) Select outbound and inbound date.

The second step, after having been selected origin and destination, is to select the outbound date, and if the traveler wants to come back from the same destination and origin, he'll have to select the inbound date. In this case the travel will have a return trip as well. If inbound date is not selected it will be a one way trip.

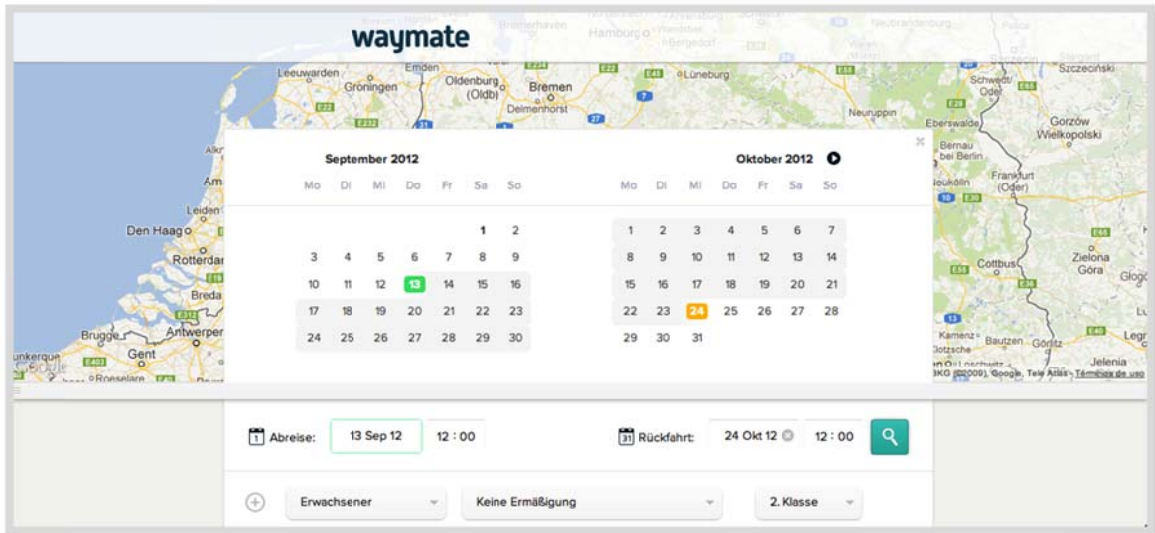


Image 12: Search page. Calendar.

Instead of having two separated calendars for outbound and inbound, Waymate presents a calendar with a customized functionality which makes easy and fast the way to pick specify date/s. The calendar's functionality is as follows:

- It's not allowed to pick a day in the past.
- Inbound date selection is optional.
- Once the outbound is selected, with the next click the inbound date will be selected, unless the day selected is before the current outbound chosen. In this case, the outbound will be reselected and the inbound will have to be selected in the next click if desired.
- When the box containing the date formatted below the calendar will indicate if outbound or inbound will be chosen after the next click over some day in the calendar.
- As inbound date is optional, could be removed by clicking the close icon within its box below the calendar.

The calendar component is actually a famous jQuery plugin, named **DatePicker**<sup>40</sup>, which offers a lot of options to be customized. To integrate it in the application this line should be added within <head></head> in the template where it's used:

```
<%= javascript_include_tag 'jquery.tools.min.js' %>
```

<sup>40</sup> Datepicker API: <http://jqueryui.com/demos/datepicker/>

### 3) Travelers options

Waymate admit until five travelers per each journey, because of API restrictions. There are a few attributes which must be set before doing a search request in order to calculate the prices for tickets accordingly to these values. The attributes that must be set are:

- The **age of every traveler** must be set due the ticket's prices are calculated depending on the age.
- If travelers have planned to travel by train and have a **train card**, each traveler must choose the one which owns. Train cards include discounts, so it's important to get cheaper prices.
- Only for traveling by train, the class has to be chosen.

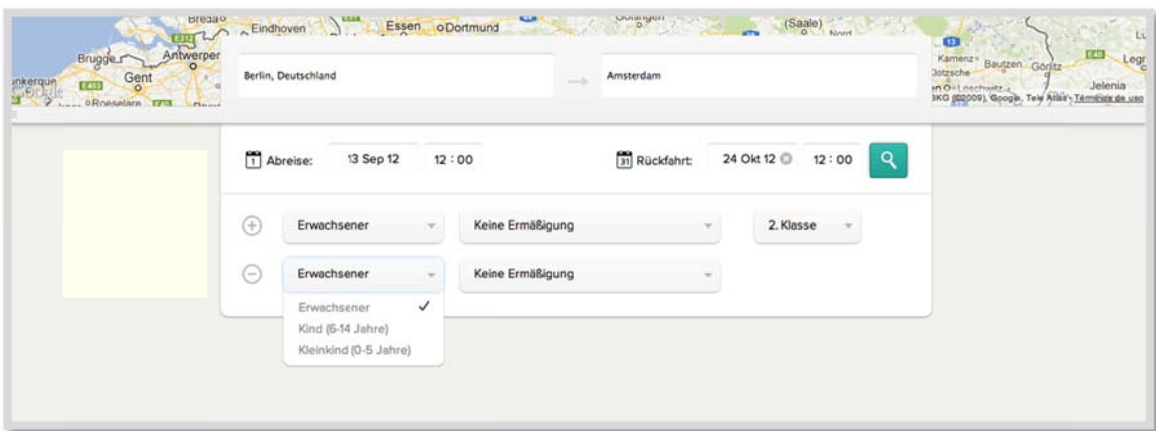


Image 13: Search page. Travelers.

### 4) Send a search request

This is the last step the user does, before getting a list of different connections for traveling from origin to destination to a specific date, or dates if it's a return trip. Once all the attributes mentioned in the previous step have been set and the user presses the search button will happen the following:

In the client-side as Backbone models are defined Search and Traveler which correspond to the object models defined on the server-side with the same name.

1. A Search and a Traveler collection models are populated from the client-side and sent to the server-side and stored in the database with the purpose of creating statistics and knowing the user preferences.
2. The closest airports are obtained for origin and destination, in the table Airports, if the distance is big enough. The same for the closest train stations, but looking up in the Dbstations table.
3. The requests to get flight and train connections, from Ypsilon and Deutsche Bahn respectively, are built up and sent in parallel.

4. Once the response is received, either from Ypsilon or Deutsche Bahn, is parsed and for each connections gotten a Tour object is created and populated on the server-side.  
The response from Ypsilon use to take longer. Therefore, even if Deutsche Bahn response is already parsed, as results for flights and trains will be displayed all together, the results are shown when both responses are parsed. Ypsilon response use to take 5 seconds approximately. On the other hand, Deutsche Bahn response takes around 1 second. Sometimes, a request for one of these API is not sent for different reasons:
  - For flights, if the distance is too short.
  - For trains, if origin and destination are not covered by Deutsche Bahn's influence areas.
5. After building up all the Tour objects on the server-side are sent to the client-side. On the client-side these Tour objects correspond to a Connection object, presenting an API to get all the attributes needed. On the client-side all the connections are ordered before being presented depending on the price and duration.
6. Present all the results in a way the user perceives them all together with the following features easily:
  - Duration of the journey.
  - Price.
  - Arrival and departure time.

### 7.2.3. Results page

In this page all connections available to travel from origin and destination will be displayed. The user can filter connections out following different criteria. Furthermore, the user can see the details for a determined connection.

Results page could be divided in 4 different sections:

#### 1) Map.

At this stage, the map functionality is only to show what was displayed in the search page after selecting origin and destination. The map will acquire more importance in the next step of the process.

#### 2) Navigation tabs.

It allows the user to navigate along different steps in the process. It corresponds to the NavigationTab view. From the left to the right side, it includes navigation to:

- a) **Search page**, to make a new search.
- b) **Outbound results**.
- c) **Inbound results** if the return trip exists. Otherwise a tab with the text “Select inbound trip” in German will show up.
- d) **Hotels**, accessing to an embedded iframe for booking a hotel in Booking.com.

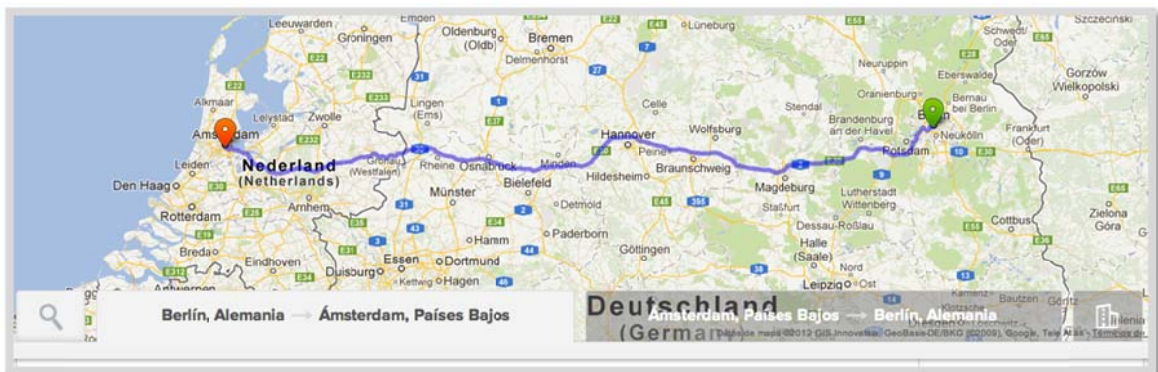


Image 14: Results page. Map and Navigation tabs

### 3) Traveler connections preferences.

This section allows to the user to display in the results frame the connections that better adjust to his preferences. It corresponds in addition with the results section to the Connections Backbone view.

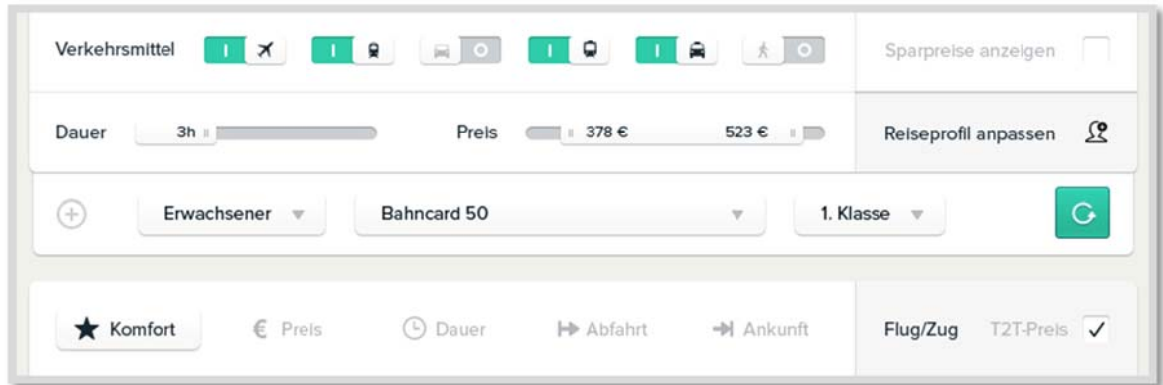


Image 15: Results page. Traveler connection preferences

**3.1. Filters** remove from the results frame those connections which don't fit within the options the user has set. There are two kinds of filters:

**3.1.1. Verkehrsmittel** (mean of transport), for filtering out those connections where a determine mean of transportation is included. If train checkbox slider is disabled, no train connections will appear in the results frame.



Image 16: Mean of transportation filters.

**3.1.2. Dauer** (Duration) hides connections whose duration is greater than the one specified.



Image 17: Duration slider filter

**3.1.3. Preis** (Price) filters out the connections whose price is out of the range defined.



Image 18: Price range slider filter

**3.1.4. Time filter** is directly connected to the results frame. It gives a visual approach to the user when filtering out the connections out the range of the time along the day specified.



Image 19: Time slider filter

**3.2. Sorting functions.** Depending which sorting function is selected, all the results will appear as follows:

- **Komfort** (Comfort) is a combination of price and duration.
- **Preis** (Price), the cheapest connections will appear on the top and the expensive ones on the bottom.
- **Dauer** (Duration), results will appear in an ascending order from the top as shorter and on the bottom the longer connections.
- **Abfahrt** (Departure), ascending ordered, being on the top the connections which will depart before and on the bottom later.
- **Ankunft** (Arrival), connections which will reach before the destination on the top and on the bottom the ones which will arrive later.

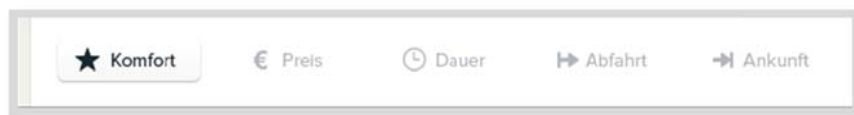


Image 20: Sorting functions

**3.3. Reisepprofil anpassen**(travelers profile customization). In case the user forgot to set a value properly or forgot to include another person, travelers can be modified from the results page. The problem is that as prices will be calculated again, another request, including its waiting time is needed.

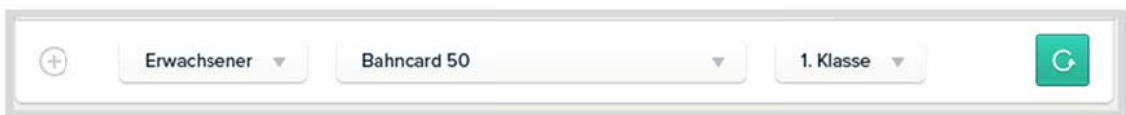


Image 21: Travelers profile customization

**3.4. Sparpreise anzeigen** (Special price). It's a checkbox for activate or deactivate special prices for connections in case they exist. Special prices should be bound to some restrictions, but are cheaper.



Image 22: Special Price checkbox

#### 4) Results.

In the results frame are all the connections retrieved after receiving the responses from the API. It can be seen that each line, with all the bars, prices, departure and arrival times (only displayed when the mouse is over the bars), corresponds to a connection. A connection is represented by a Tour object on the server-side and by a Connection object on the client-side.

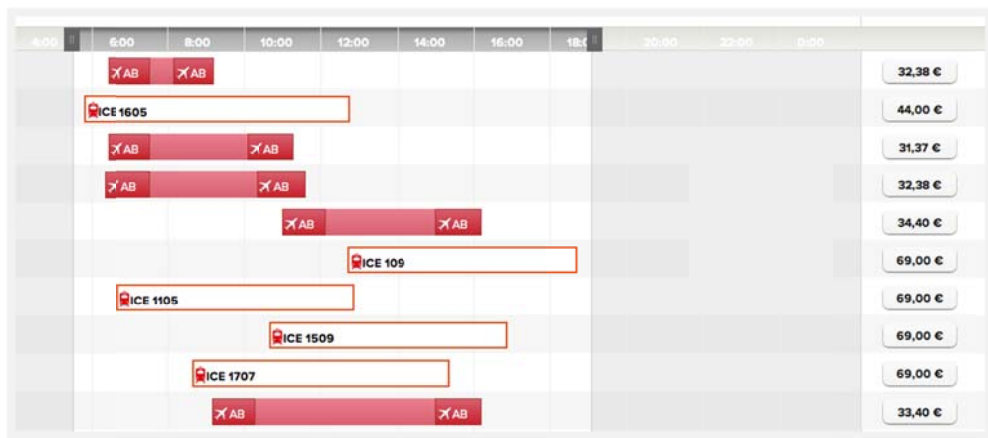


Image 23: Results page. Connections frame.

The idea behind of displaying connections as bars is to give a quick overview of the duration within the time frame. The price associated to each connection is displayed on the right side of the line. This approach allows identifying easily those connections which are faster and cheaper, selecting the corresponding sorting functions.

In order to differentiate easily between train and flights, bars which represent their connections are displayed in different ways:

- Train connections are displayed with a white background and red. Within these bars, if there is space enough, the train icon, the kind of train and its number will be displayed.
- For flights bars backgrounds are filled with the corresponding airline color. If inside the bar there is enough space, a plane icon and the IATA code of the airline is displayed.

## 7.2.4. Details page

After the user clicks over a connection, the next step in the process will be to display all the details respecting the connections selected. In the details there are a few sections which can be differentiated:

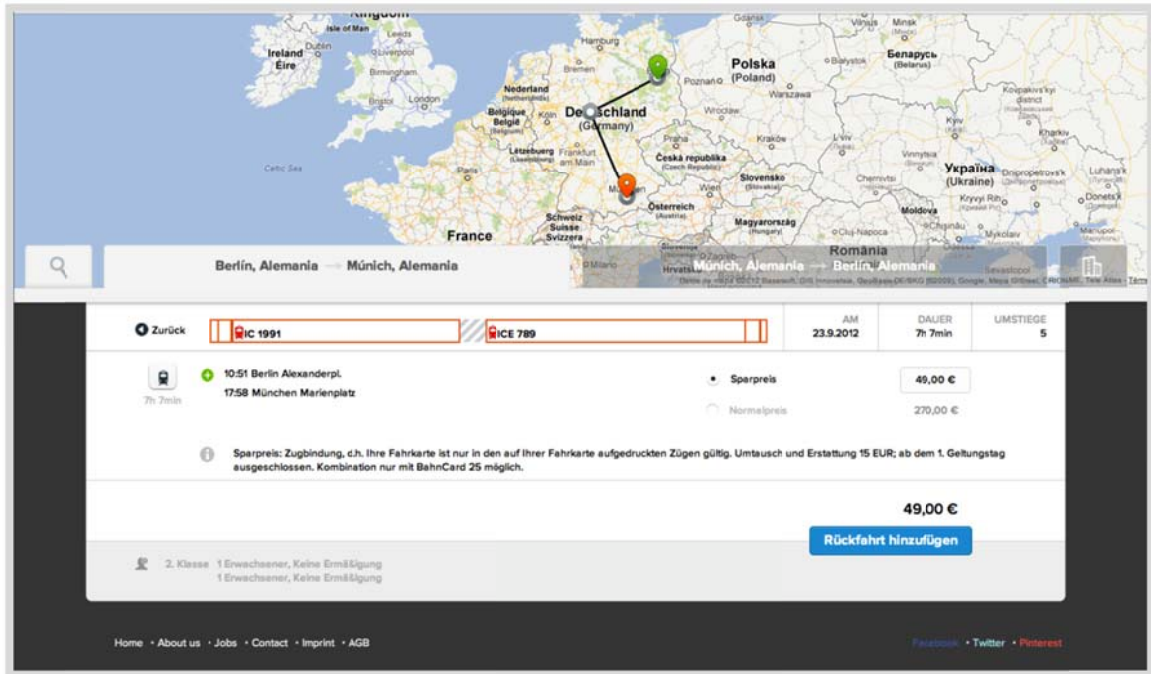


Image 24: Details page.

### 1) Map.

Here's the point along the process where the map acquires more importance. In this stage, map solves time and space, all in one.

The Connections view triggers an event with the connection selected information which is listened by the Map view, where the details are displayed. Origin, destination and all the stopovers associated to the selected connection will be displayed over the map, solving the space representation.

The time representation is solved displaying an info-window when the user clicks over a marker, displaying:

- 1) Name of the train station or airport.
- 2) Arrival time and date.
- 3) Departure time and date.
- 4) Stopover number.
- 5) Carrier information, airline and flight number for flights and type of train and number for trains.

All these information will be attached to each marker which represents a stopover. For origin marker's info-window will be displayed 1), 3) and 5).

For the one which corresponds to the destination only 1), 2) and 4).

### 2) Navigation bar which is already explained.



### 3) General representation of a connection.


Bar displayed as in the results frame, but expanded, date, duration and stopovers.



Image 25: Details page. General representation of a connection

### 4) Details.

It's made-up by Details view and a Connection model. It includes the following sub-views:

- **Connection details expanded.** When the  button is clicked, all the details regarding time, carrier and stations or airports are displayed giving another overview of the entire connection with its sub-connections.

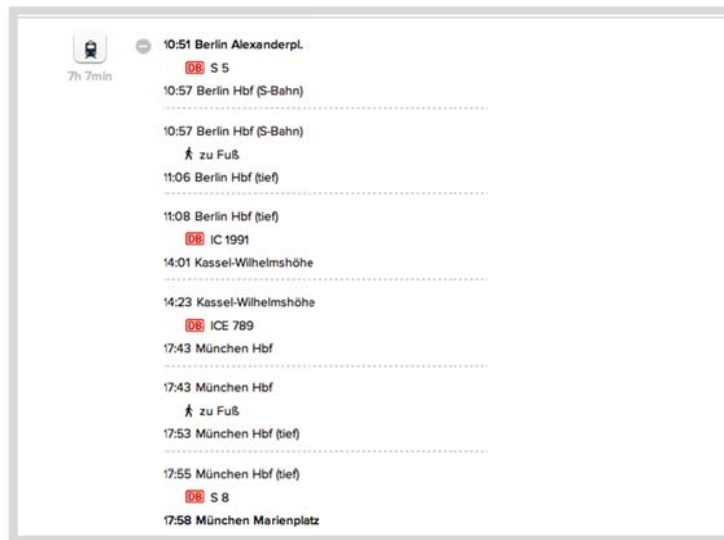


Image 26: Details page. Connection details expanded

- **Price switching.** If there exists more than one kind of price, the user can select the one which prefers.



Image 27: Details page. Price switching

The relevant information for the price is explained in this section:



Image 28: Details page. Conditions for a type of price

When this page is loaded a **price validation** is triggered for the price preselected and the same the first time that the type of price is switched. It's performed in case the price changes since the user has entered the first time to the results page.

It involves these steps:

- 1) A request from the client to the server-side with the relevant information of the connection and the price to validate.
- 2) Once this information is on the client-side, it's sent to Ypsilon or Deutsche Bahn API.
- 3) The corresponding API sends back to the server-side the validation with the new price in case it has changed.
- 4) From the server-side this information is sent to the client-side and displayed.

The connection details expansion and price switching composes ConnectionInfo view, which will be re-used in advanced steps.

#### 4) Travelers' information.

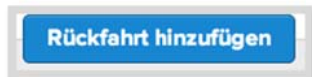
The user should be aware of travelers setting at each stage in case they're wrong. It is composed by the Travelers' collection with a specific template for the details page.



Image 29: Details page. Travelers profile set.

#### 5) Go to next step. There are two options:

- **Select inbound connection** if return trip exists and the connection selected was for the outbound travel by pressing this button.



In this case, the user will be a results page for the inbound trip with compatible connections regarding the one chosen for outbound. Once the user selects one will be redirected to the details page. If he is interested in the combination, he will press the button for going to the booking form.

- **Go to the booking form** for buying the tickets by clicking this other button.



## 7.2.5. Booking form

At this stage the user will fill the form with its personal data for associating him the tickets bought and besides the payment method. Different sections could be distinguished.

- 1) **Map.**
- 2) **Navigation tab.**
- 3) **Details for connections selected.**

The details for each connection corresponds to the sub-view employed in the Details, named ConnectionInfo view.

It's important to keep informed the traveler about his choices along the whole process. Reached this point, the traveler probably would like to check again the details of the connections chosen or even change the type of price, if it exists, he is going to pay for.

Travelers setting are displayed as well, but there is no choice to reset them unless a new search is done or they are modified from the results page.

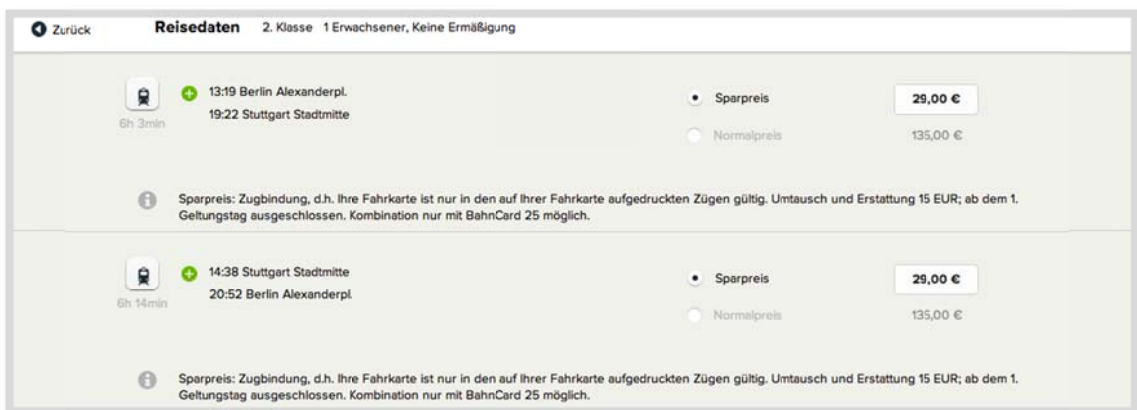

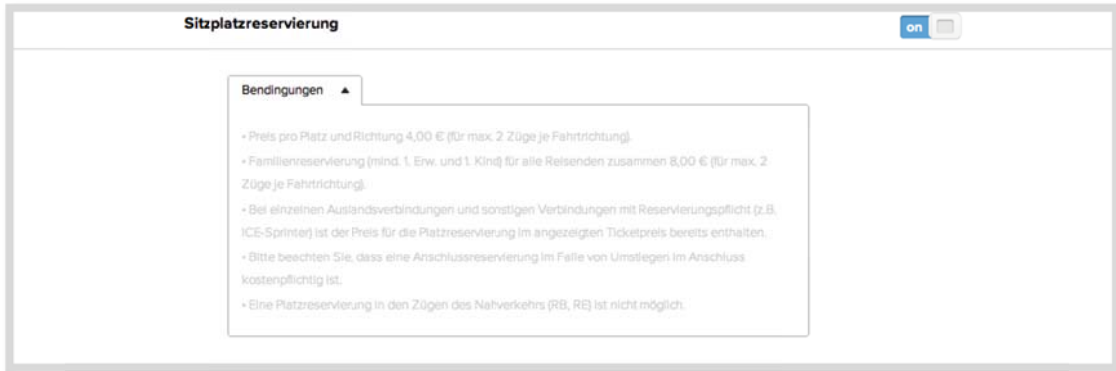


Image 30: Booking form. Traveler profile set and details view for selected connections

#### 4) Seat reservation.

It composed by BookingReservation view and its model Reservation associated to the view.

This section will only appear when train is included in the connections selected. The traveler will have to activate reservation by enabling the button  and the select his seat. All the terms and conditions regarding seat reservation are included in this area, as well as the additional price which will be charged.



**Sitzplatzreservierung** on

**Bedingungen**

- Preis pro Platz und Richtung 4,00 € (für max. 2 Züge je Fahrtrichtung).
- Familienreservierung (mind. 1. Erw. und 1. Kind) für alle Reisenden zusammen 8,00 € (für max. 2 Züge je Fahrtrichtung).
- Bei einzelnen Auslandsverbindungen und sonstigen Verbindungen mit Reservierungspflicht (z.B. ICE-Sprinter) ist der Preis für die Platzreservierung im angezeigten Ticketpreis bereits enthalten.
- Bitte beachten Sie, dass eine Anschlussreservierung im Falle von Umstiegen im Anschluss kostenpflichtig ist.
- Eine Platzreservierung in den Zügen des Nahverkehrs (RB, RE) ist nicht möglich.

Image 31: Booking form. Seat reservation conditions



**Abteillart** **Platzlage** **Bereich**

beliebig  beliebig  beliebig

Abteillwagen  Fenster  Ruhebereich

Großraumwagen  Gang  Handbereich

Großraumwagen mit Tisch

Eine Sitzplatzreservierung ist nicht immer möglich. Mit dieser Option stimmst Du zu dass Du das Ticket erwirbst, auch wenn kein Sitzplatz reserviert werden kann.

Tickets auch buchen, falls die Sitzplatz-Reservierung nicht durchgeführt werden kann. i

8,00 €

Total 66,00 €

Image 32: Booking form. Seat selection

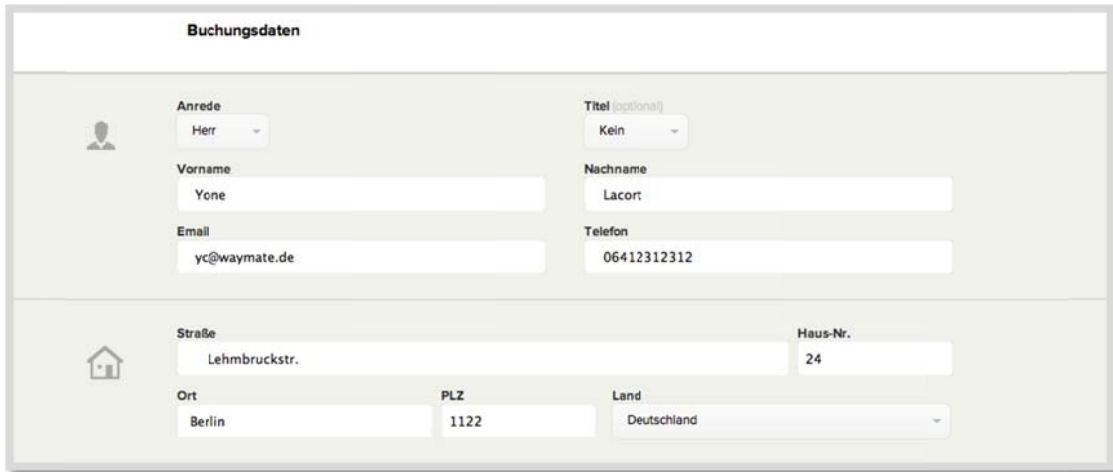
#### 5) Customer personal data and address.

In this section begins the booking form where the user has to introduce his personal data.

It's composed by BookingCustomer view and the model Customer to represent the data associated.

Customer personal data will identify the user who will purchase the tickets. Tickets will be associated to the person whose personal information belongs. In case that some error occurs when the payment is performed, Waymate can obtain his personal data, contact him and solve the problem.

This information will remain in the input fields next time the user enters the booking form to buy another ticket. Obviously, the first and the following times, the user must be logged in if he wants to store/keep this information and not fill this section again.



**Buchungsdaten**

Anrede: Herr | Titel (optional): Kein

Vorname: Yone | Nachname: Lacort

Email: yc@waymate.de | Telefon: 06412312312

Straße: Lehmbruckstr. | Haus-Nr.: 24

Ort: Berlin | PLZ: 1122 | Land: Deutschland

Image 33: Booking form. Customer personal data and address

## 6) Credit card information and online identification.

This section is integrated by BookingCreditCard view and CreditCard model. The purchase will be fulfilled by credit card payment. The types supported are:

- Visa
- MasterCard
- American Express
- Diners Club
- JCB



Kreditkartentyp: VISA Card | Kreditkartennummer: | Gültig bis: MM | Sec-Code: ||||

Image 34: Booking form. Credit card information

Online identification section will show up only when train tickets are involved, because Deutsche Bahn requires it. This identification must be with data of the user who will purchase the tickets. Identification can be fulfilled by:

- Credit card.
- Deutsche Bahn card.
- Id number.
- Maestro card.



Identifizierung für Online-Ticket: Kreditkarte | Kreditkartennummer: | Gültig bis: MM | ||||

Image 35: Booking form. Online identification

## 7) Travelers' personal information.

This information will only be required always for flights and for trains when the travel is from Germany to another country.

Each traveler is a BookingTraveler view and its corresponding Traveler model.

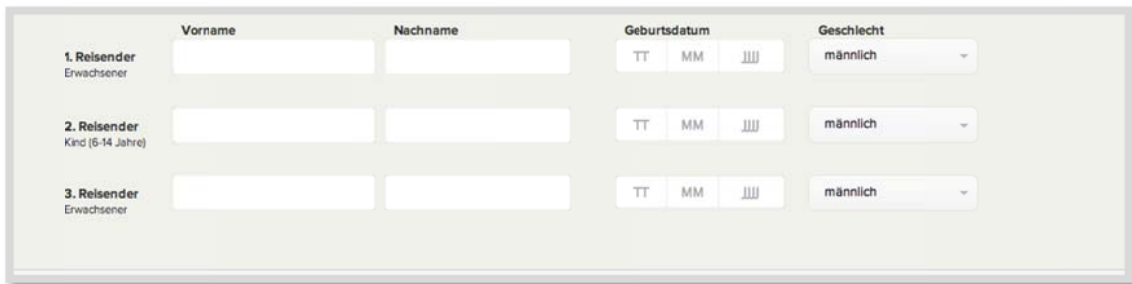


Image 36: Booking form. Travelers' personal information

## 8) Terms and conditions.

The lines specified for this section will vary depending on which API, Ypsilon or Deutsche Bahn, has provided the connection selected.





Image 37: Booking form. Terms and conditions

## Validation

After filling all the input fields and select the corresponding options in the drop down lists, the user will press this button  to purchase the tickets.

There is still an important process to explain, but not visible to the user. It's not possible to guarantee that the user has filled all the input fields properly. However, it's possible to make sure that all the input fields are filled in the proper format.

Therefore, when the user clicks the button for buying tickets, all the input fields must pass the validation, as well as checkboxes which must be checked before the form is sent. If a single input field doesn't pass the validation, the form won't be sent. For each input field which not passes the validation, a pop up window will appear above them informing about the error. Validation is done inline as well, which means when the user leaves an input field, validation it's done for that single input field in that moment, providing to the user instantaneous feedback.



**Image 38: Pop-up errors when validation fails**

Validation types:

- Input fields within customer data-address section are validated by matching their value with a predefined regular expression. Thus, can be said that are validated statically. The same validation is done for travelers' name and surname.
- Credit card number input field implies more complexity. Each type of credit card has its own regular expression to match and furthermore, it must pass the Luhn algorithm, which is a checksum formula to validate identification numbers.
- Individual input fields which compose a date are validated with a regular expression. If the date includes a day input field, the day must be valid depending on the month introduced. i.e. 30/02/2013 wouldn't be valid.
- Expiration date input fields are validated dynamically considering the current date.
- Birthday dates for a determine age are also validated dynamically. Hence, the age introduced must be in the range of specified age to pass the validation.

## Tickets purchase request

Once validation success for all the input fields the form can be sent. Reservation, Customer, CreditCard, collection of Travelers and Connections object models will compose a containing object named as Order.

Furthermore, when the Order object is completed with the information that comes from the booking form, is sent to the server-side and:

- Stored in the database in order to keep track about all the purchases that the users have done. When the ticket is bought, if the user was logged in, this Order object will be associated and accessible to the user. Otherwise, it will be store without any reference to the user.
- Afterwards request in XML format is built to send it either to the Ypsilon, Deutsche Bahn or both APIs. The response follows the same flow as always, received on the server-side, parsed and sent to the client-side.

If an error occurs, the payment will be cancelled and the user will be notified about what happened. Otherwise, if the booking and payment success, the application will redirect to the tickets page.

## 7.2.6. Tickets

If the user has reached this point along the application, it means that all the tickets for the connections selected have been bought. No matter how many were or if there were trains and flight tickets involved. All tickets are bought in a single process. Map, NavigationTab and Details views will remain on the top, as in the booking form. The only distinction will be that the user won't be able to switch prices because he has already bought the tickets.

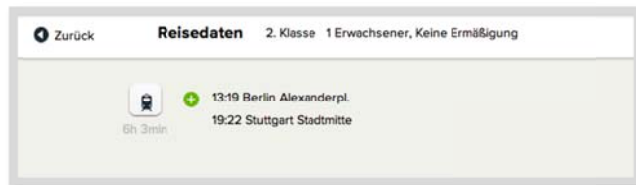



Image 39: Tickets. Travelers preferences set and details view for purchase connections

All tickets will be sent to the email address specified in the booking form without exception. Deutsche Bahn tickets will be available as well from this page by clicking in this icon  and being redirected to an URL where a single ticket is in PDF format.

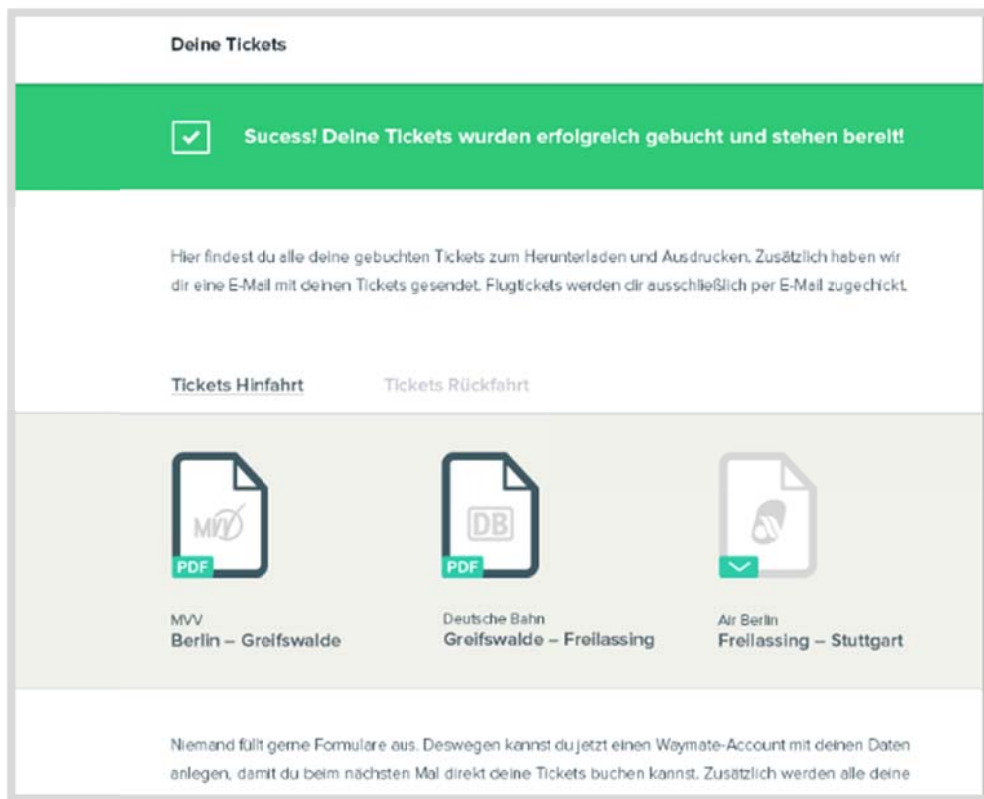


Image 40: Tickets. Tickets purchase links and information



## 7.2.7. Hotels booking

The user will be able to book a hotel during his stay in the city which specifies in **Booking.com**. He will have the freedom to book it in the moment which he decides along the process. If the user doesn't select the corresponding tab for hotels, this page will never show up.

In Waymate, the hotels page it's an embedded iframe defined in the Booking view and provided by Booking.com. It has an id associated to Waymate, so if a user will book a hotel, Booking.com will know that it's been done through this application. All the functionality for carrying out a hotel booking is included in the iframe.

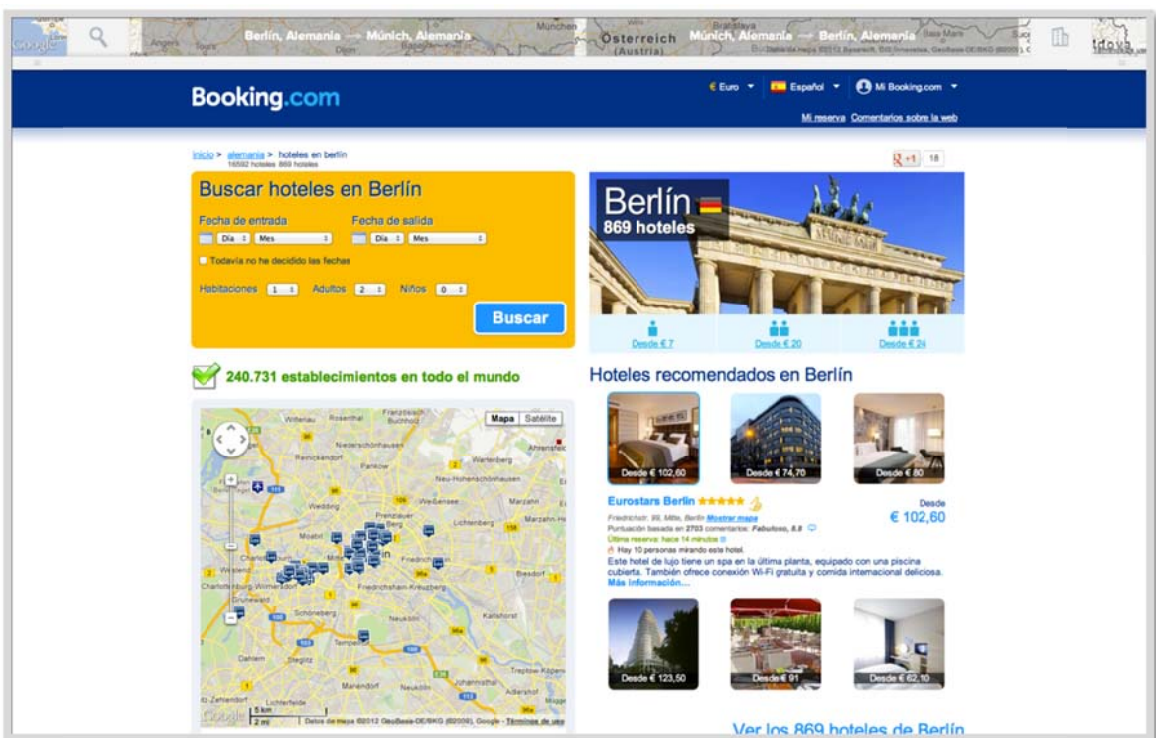


Image 41: Hotels booking.

## Conclusion

In the end, employing all these technologies described in conjunction will help to develop a complete web-application with all the needs covered.

The Model-View-Controller design pattern will provide a structure and organized code on the client and server-side. Moreover, the code will be more reusable and maintainable, being a safe of code lines and time when developing.

Using Ruby on Rails will guarantee to have a complete framework with a big amount of libraries (*RubyGems*) which will provide support for all features the needed for such complex application. Summarizing, Rails components and included *RubyGems* make easy important features such as:

- Database storage. Object models have the ability to be persistent simply without the need of using SQL commands.
- RESTful services management.
- RubyGems/libraries integration.
- Object model associations.
- Database migrations.
- Easy and flexible routes specification.
- Internationalization to support different languages.
- Assets management in production environment to optimize the performance.
- Testing. Rails include support for testing the server-side code. Furthermore, including the libraries exposed will provide testing frameworks for the client-side per separate and both together. In the end, all the code testable is tested properly in several ways.
- Deployment. Capistrano automates this process in an easy way, adding a lot of useful features.
- Rails provides implicitly solutions for security issues.

Having JavaScript in addition with jQuery and Backbone libraries will provide a highly interactive single-paged web application.

Employing a web-based hosting service such as Git provides a complete revision control system to provide management of changes along the application development.

The Software Development Life Cycle model chosen, allows to implement new features and improve them gradually, having a working first version available to all the team. This method lets different team components to see and test it, in order to decide if still needs to be improved soon or eventually.

Finally, the outcome is an application in the vanguard with several frameworks embedded in Rails, providing support to all the required functionalities and features.

The application is still in development phase and thus not all the features thought for the final product are implemented. In Waymate, the state achieved so far only includes solutions for long distance.

Waymate has already access to public transport information to connect two different addresses inside whatever city in Germany. As other milestones were considered more important by Waymate's team the solution for short distance, firstly inside Germany, is still work in progress due to the data obtained from the API is not 100% reliable. Hence, it requires more time and resources investment as expected.

Once all the functionality is covered inside Germany, the idea is to sign contracts with other train companies and hence have access to APIs which provide train connections for other countries, as well as public transportation. In some countries, this information is not available yet, or only for the important cities.

The development of the mobile app for iOS is carrying out in parallel, but there is still a lot of work to do before having the first version in the market. The short distance acquires more importance here, thus this is the main reason because other milestones were considered more important. In the future the Android app will be implemented as well, because otherwise a large market is lost.

Waymate has been invited to a European Transport Ministers' Conference in Cyprus, where the platform was exposed. The idea behind was trying to convince the ministers from all the countries inside the European Union to create these APIs and open them in a control way. All these authorities seemed to be very pleased and willing to speed up this process given the high benefits that will provide to travelers and their own country.

The business model is to earn a percentage of each ticket sold through the platform. As Waymate is brand new, it's not so easy to be known in such huge market. Anyway, step by step Waymate is acquiring acknowledgement in the European market:

- It has been awarded as 1<sup>st</sup> Smart Mobility Challenge in 2012 by the European Commission of Transportation.
- Nominee as European Startup of the Year by ICT Spring Awards 2012.
- Nominee as Best Travel Technology Solution by Travel Industry Club in 2012.

# References

## **2. Software Development Life Cycle**

Software development life cycle: [http://en.wikipedia.org/wiki/Software\\_development\\_methodology](http://en.wikipedia.org/wiki/Software_development_methodology)

Comparison of web application frameworks:

[http://en.wikipedia.org/wiki/Comparison\\_of\\_web\\_application\\_frameworks](http://en.wikipedia.org/wiki/Comparison_of_web_application_frameworks)

## **3. Server-side. Ruby on Rails**

MVC in Rails: [http://guides.rubyonrails.org/getting\\_started.html#the-mvc-architecture](http://guides.rubyonrails.org/getting_started.html#the-mvc-architecture)

Ruby on Rails components: [http://guides.rubyonrails.org/getting\\_started.html](http://guides.rubyonrails.org/getting_started.html)

Model validations: [http://guides.rubyonrails.org/active\\_record\\_validations\\_callbacks.html](http://guides.rubyonrails.org/active_record_validations_callbacks.html)

Database migrations: <http://guides.rubyonrails.org/migrations.html>

Associations: [http://guides.rubyonrails.org/association\\_basics.html](http://guides.rubyonrails.org/association_basics.html)

Render method: [http://guides.rubyonrails.org/layouts\\_and\\_rendering.html#using-render](http://guides.rubyonrails.org/layouts_and_rendering.html#using-render)

Action view helpers: [http://guides.rubyonrails.org/form\\_helpers.html](http://guides.rubyonrails.org/form_helpers.html)

Parameters: [http://guides.rubyonrails.org/action\\_controller\\_overview.html#parameters](http://guides.rubyonrails.org/action_controller_overview.html#parameters)

Session: [http://guides.rubyonrails.org/action\\_controller\\_overview.html#session](http://guides.rubyonrails.org/action_controller_overview.html#session)

Filters: [http://guides.rubyonrails.org/action\\_controller\\_overview.html#filters](http://guides.rubyonrails.org/action_controller_overview.html#filters)

Request forgery protection: [http://guides.rubyonrails.org/action\\_controller\\_overview.html#request-forgery-protection](http://guides.rubyonrails.org/action_controller_overview.html#request-forgery-protection)

Rails routing: <http://guides.rubyonrails.org/routing.html>

Rails internationalization with I18n: <http://guides.rubyonrails.org/i18n.html>

Test-Driven Development: [http://en.wikipedia.org/wiki/Test-driven\\_development](http://en.wikipedia.org/wiki/Test-driven_development)

Test-Driven Development: [http://en.wikipedia.org/wiki/Test-driven\\_development](http://en.wikipedia.org/wiki/Test-driven_development)

Fixtures: <http://guides.rubyonrails.org/testing.html#the-low-down-on-fixtures>

Unit testing: <http://guides.rubyonrails.org/testing.html#unit-testing-your-models>

Functional tests: <http://guides.rubyonrails.org/testing.html#functional-tests-for-your-controllers>

Behavior-Driven Development: [http://en.wikipedia.org/wiki/Behavior-driven\\_development](http://en.wikipedia.org/wiki/Behavior-driven_development)

Cucumber: <http://cukes.info/>

Capbara: <https://github.com/jnicklas/capybara>

Poltergeist: <https://github.com/jonleighton/poltergeist>

Phantom installation guide: <http://phantomjs.org/download.html>

Asset pipeline: [http://guides.rubyonrails.org/asset\\_pipeline.html](http://guides.rubyonrails.org/asset_pipeline.html)

#### **4. Deployment process**

Capistrano guide: <https://gist.github.com/2161449>

Jenkins guide: <https://wiki.jenkins-ci.org/display/JENKINS/Use+Jenkins>

TinyMon: <http://www.tinymon.org/pages/faq>

#### **6. Client side**

HTML: <http://en.wikipedia.org/wiki/HTML>

CSS: [http://en.wikipedia.org/wiki/Cascading\\_Style\\_Sheets](http://en.wikipedia.org/wiki/Cascading_Style_Sheets)

SASS: <http://sass-lang.com/>

JavaScript: <http://en.wikipedia.org/wiki/JavaScript>

AJAX: [http://en.wikipedia.org/wiki/Ajax\\_\(programming\)](http://en.wikipedia.org/wiki/Ajax_(programming))

jQuery: <http://en.wikipedia.org/wiki/Jquery>

Backbone: <http://backbonejs.org/>

Jasmine: <http://pivotal.github.com/jasmine/>

#### **Google APIs:**

Google Places auto-complete: <https://developers.google.com/places/documentation/autocomplete>

Geocoder: <https://developers.google.com/maps/documentation/javascript/geocoding>

Google Maps API: <https://developers.google.com/maps/documentation/javascript/reference>