UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

Design and implementation of a web platform for running events

Dissertation presented by
Michaël Heraly

for obtaining the Master's degree in
Computer Science
Option(s): Software Engineering

Supervisor(s)
Charles PECHEUR

Reader(s)
Kim MENS, Sascha VAN CAUWELAERT

Academic year 2015-2016

Abstract

Currently, the organizers of running events have to handle the promotion and the registrations by themselves. Some organizers have a website to promote their event, but they often rely on external services for either registrations, timing results, or both. Furthermore, there is not a single place where runners can get information and register to runs.

The goal of this work is to design and implement a web platform for helping the organizers to present and manage the registrations for their events. The platform allows runners to register to running events. The system is based on the client-server architecture for building a single-page application.

The platform consists of a web application developed with the Ruby on Rails and Ember frameworks. The communication between both parts of the application is performed with JSON documents. The system is usable in different languages and on multiple devices (desktop, tablet and mobile phone). Some external services are used, in particular for allowing the organizers to draw the route of their runs on a map. We tried to build an application that is modular and maintainable. Tests and validation are conducted to verify the system behaviour.

Acknowledgements

We would like to thank all the people who played a role and helped us through our studies at the Université Catholique de Louvain.

We are grateful to those who helped and supported us to achieve this project.

In particular, we would like to thank our promoter M. Charles Pecheur for his wise advices during the whole project. We would like to thank our readers M. Kim Mens and M. Sascha Van Cauwelaert.

We would like to thank our friends and family for supporting us throughout our formation.

Contents

Chapter 1

Introduction

1.1 Context

Running is an activity that has been gaining popularity recently. More and more people are participating to running events which bring together hundreds of runners. The goal of this work was to design and build a platform that allows organizers to manage their events, with a focus on running events.

1.2 Problem

Organizers of running events have to take care of a lot of things:

- Presentation of their event
- Manage the different distances proposed for an event
- Manage the pre-registrations (before the event)
- Manage the registrations (on the day of the event)
- Manage the payments
- Manage and present the results of the event

During the whole process of managing an event, an information system would be helpful.

1.3 Existing solutions

Some solutions exist to help the organizers through this process:

Chronorace [1] service supporting the pre-registrations and the timing results for different sports (athletics, jogging and cycling). It is specialized in the electronic timing of sport events.

njuko [2] service supporting the registrations to events. The results can be shown on the website of the organizer. It is a web application focused on the registration part of an event.

Bigger events (e.g. "20km de Bruxelles") have their own website, and manage the different steps of the process on their own. But most organizers cannot afford having their website.

[1] `http://chronorace.be`
[2] `http://njuko.com/`

Another issue is the lack of concern for runners. Apart from Chronorace which allows runners to create accounts, the participants have to fill their personal information each time they register to an event.

From a runner's perspective, accessing the agenda of the running events is not really practical. Some websites (e.g. GoRunning[3] and L'Avenir[4]) present an agenda and a basic presentation of the running events (mostly text data), but they do not provide additional features such as registrations and results. Table 1.1 shows a comparison of the features offered by the different services.

	Chronorace	njuko	GoRunning	L'Avenir
Agenda	partial	✘	✔	✔
Presentation	✘	✘	basic	basic
Registrations	✔	✔	✘	✘
Results	✔	external	✘	✘

Table 1.1: Features comparison table

1.4 Objectives

The goal of this work is to build a web platform that allows organizers to easily manage their events. It should also help the runners to see the available runs, and facilitate their participation to running events. The platform should also be usable on the different device formats (phones, tablets and desktops), and in different languages (english and french).

1.5 Approach

The platform will be developed as a single-page application [1]. A single-page application is a web application or a web site that fits on a single web page with the goal of providing a more fluid user experience similar to desktop applications. This solution enforces the architecture to be composed of two communicating parts: the frontend and the backend. [2] In that case, the web browser does not have to load a HTML file for each page of the application. The web browser loads a HTML file the first time the application is accessed. Then, only the data needs to be transfered between the frontend and the backend. It allows to provide a fluid experience, especially for users accessing the application with a mobile device (phone or tablet).

1.6 Roadmap

The remainder of this document is structured as follows: the next chapter provides an insight of the technology choices. Then, a chapter is dedicated to an overview of the principles of Ruby on Rails and Ember. After, the design of the proposed solution is presented. The software architecture of the implementation is explained in relation with the different features. Tests and validation are then conducted to assess the solution. Finally, a conclusion reports the main findings of the project and we present some ideas for future work.

[3] `http://gorunning.be/calendrier_fr.php`
[4] `http://www.lavenir.net/extra/running/agenda`

Chapter 2

Technology choices

2.1 Software architecture

Building a web application requires to choose a software architecture. Historically, web applications (and web sites) performed server-rendering to serve the HTML pages. [3] For this application, we opted for a client-server architecture. Client-server architecture allows multiple frontends (e.g. web application, iOS app, Android app) to communicate with a single REST[1] API.

REST is the software architectural style of the World Wide Web. It consists of using the HTTP verbs (GET, POST, PUT, DELETE, etc.) to communicate data with a remote server. In general, the data are presented to external systems as web resources identified by Uniform Resource Identifiers (URIs) such as `/runs/1` (which corresponds to the run whose id is 1). External systems can use HTTP verbs to perform standard operations, such as `GET /runs/1` to retrieve the data about the run whose id is 1. This architecture is adapted to build single-page applications.

Single-page applications provide certain benefits:

Fluid user-experience The application feels more like a native application than a web page. This can improve the user-experience, which is an important aspect for user applications. [4–6]

Less server load The server is only responsible for the data, and does not have to compute the HTML layout of the pages. The page rendering is performed by the client. The server can generate JSON faster than a complete HTML template [7], thus responding faster to client requests.

Less bandwidth The ressources necessary for the client-application (HTML, CSS and JavaScript) are loaded once. Subsequent pages only require data from the server.

But developing a single-page application is also more complex: it is composed of (at least) a frontend (client-side) and a backend (server-side). The communication between the two parts must also be handled. This increases the system complexity and the amount of potential causes of failure compared to a monolithic application, where the server manages every step from data access to the user interface.

Considering this, we decided to build a single-page application based on the client-server architecture. In order to do this, we chose the frameworks that will support the development of our web application.

[1] `https://en.wikipedia.org/wiki/Representational_state_transfer`

2.2 Backend framework

For the backend development, three frameworks were considered: Rails [8], Django [9] and Express.js [10]. These are all based on the Model-View-Controller architectural pattern.

Rails is developed in the Ruby language. It provides a lot of facilities for building web applications. It is built on the principle of "Convention over configuration" [11]. It means that sensible defaults are provided. Rails allow developers to only specify unconventional aspects of the application. As an example, the database table of the `Run` model is called `runs` by default.

Django is based on Python. Among the list, it is the framework with the fewest packages. A lot of those packages are not compatible with Python 3.

Express.js is developed in JavaScript and is based on Node.js. It is a minimalist framework which depends a lot on plugins for more features. Being based on JavaScript, it relies heavily on callbacks which make the code harder to read and maintain.

	Rails	**Django**	**Express.js**
Language	Ruby	Python	JavaScript
Latest version	11/03/2016 (4.2.6)	02/05/2016 (1.9.6)	22/01/2016 (4.13.4)
Community	Very active	Active	Active
Extensions	118.000 gems	3150 packages	275.000 modules
ORM	✔	✔	via plugin
Migration	✔	via package	via plugin
REST	✔	✔	✔

Table 2.1: Backend frameworks comparison

Table 2.1 shows a comparison of the frameworks. In addition to the comparison points shown in the table (ORM, migration, REST), the important criteria for the choice of the framework were the conciseness of the code, the speed of prototyping and stability of the framework. In relation to those criteria, Rails seems to do a better job than the other frameworks. The Ruby language offers code expressiveness and thanks to the "Convention over configuration" approach, the code can stay small while offering a lot of features. A lot of gems are also available and the community is supporting them actively.

2.3 Frontend framework

In respect of the framework that would run on the client-side, 2 frameworks were considered: Angular.js [12] and Ember.js [13].

Angular.js uses plain JavaScript object for the models. This might seem like some sort of freedom, but it also means that the framework does not provide additional features to handle model objects. Angular directives are extending the HTML tags with their specific attributes. This idea was not really pleasant as the directives are cluttering up the HTML.

Ember.js is based on the same principle as Rails: "Convention over configuration". It makes assumptions about the application structure and the backend API. One illustration of this principle is that Ember is able to find the different modules used in the application based on their name. Only a few explicit links must be declared (e.g. for including mixins). Ember.js offers a clear application structure among the different files and directories. Ember-Data [14]

is a library for managing model data and the JSON communication with a backend API. The Ember environment also offers a lot of useful development tools. `ember-cli` [15] is a command-line utility that performs live-reloading of the code changes, checks the code-style using JSHint, and manages the assets pipeline (JavaScript, CSS and extensions). The Ember-Inspector [16] is a web browser addon which helps during the development of an Ember application. It allows to see the routes available in the application, the different views/components that are displayed, the data stored on the client-side (and their state), plus the Promises [17] that were created in the application.

The choice went for Ember.js as it shares the same principle as Ruby on Rails, respects a clear application structure, provides useful features and offers tools helping the application development.

2.4 Responsive framework

To build a application that works on the different device formats (phones, tablets, desktops, etc.), a responsive framework might be useful.

The SASS language[2] is a meta-language over CSS. It provides useful features: nesting syntax, mixins, rules inheritance, operators, and more. A preprocessor is necessary to compile the SASS rules to CSS (web browsers only support CSS for styling purposes); this can be handled by the `ember-cli-sass`[3] Ember addon.

Knowing this, the following responsive frameworks were considered: Bootstrap, Foundation and Semantic UI.

Bootstrap [18] is a popular responsive framework. It supports the SASS language and provides a lot of HTML and CSS components. It uses mainly pixel units to define its components. Customization is not easy as components define a lot of CSS rules. These rules must be overriden in order to customize the component.

Foundation [19] is a mobile-first responsive framework. It proposes a solid responsive grid and common components (using CSS and JavaScript). These are designed using few styling rules, so the visual design can be modified easily. Its objective is to provide enough styles to prototype quickly, while allowing an easy customization.

Semantic UI [20] is a framework that uses semantic approach to build the user interface. The concepts proposed are interesting, but it is still a young framework, and it turned out after some experimentation that some features (e.g. navigation menu and reveal component) are not really performant on mobile phones. At the time of writing, there is no support for the SASS language. This makes using the framework a bit unpractical to use as the predefined classes must be added directly in the `class` attribute of an HTML element (rather than using a mixin in the application-defined component).

Table 2.2 reports the differences between the frameworks. Apart from Semantic UI, Bootstrap and Foundation provide SASS support. Semantic UI relies on a larger grid (16 columns) than Bootstrap and Foundation (12 columns). A larger grid size can be interesting as it gives more flexibility for the design. Bootstrap and Foundation allows to customize the grid size through configuration. The main differences are the size unit and the components.

When designing a web application, different size units are available to define the size of elements on a web page: pixels, ems and rems. Pixels are an absolute unit of measurement. It means that it represents the same size regardless of the device screen. Ems are a relative

[2] `http://sass-lang.com/`

[3] `https://github.com/aexmachina/ember-cli-sass`

	Bootstrap	**Foundation**	**Semantic UI**
Latest version	24/11/2015 (3.3.6)	08/04/2016 (6.2.1)	22/01/2016 (4.13.4)
Community	Very active	Very active	Active
SASS support	✔	✔	✘
Grid size	12 columns	12 columns	16 columns
Size unit used	pixels	rems	ems
Components	✔	✔	✔

Table 2.2: Responsive frameworks comparison

measure of length. It is relative to the font-size of the parent element. Ems might cause problems: when a parent element is added, the size of the nested elements change as ems depend on the direct parent element. Rems (root ems) units solve that problem as the unit is always relative to the font-size of the `<html>` element. It does not depend on its parent elements. The three frameworks do not prevent other units from being used, but it is easier to use the same unit as the framework components. For these reasons, the preference is to use rems unit.

The three frameworks provide predefined components: menus, dropdowns, buttons, etc. These are useful and help to prototype a design. Bootstrap provides a lot of components, but these are defined with a lot of CSS rules using pixel units. This makes it difficult to customize the components as part of those rules must be overridden. Semantic UI offers interesting components. But the experimentation pointed out that some components (e.g. navigation menu, reveal menu) were not fluid on mobile devices. Foundation provides enough components to build a prototype, with few CSS rules. This helps customizing the components to create a unique design.

Considering the different criteria, the framework chosen was Foundation. It is a responsive framework with a powerful grid. Furthermore, it provides useful components whose default styles can be easily redefined.

Chapter 3

Background material

In this chapter, we explain the main principles upon which Ruby on Rails and Ember rely. This might be necessary to understand the Implementation chapter.

3.1 Ruby on Rails

This section presents the main concepts of Ruby on Rails. More complete information can be found in the Ruby on Rails guides [21].

Ruby on Rails is based on the Model-View-Controller architectural pattern. It follows the principle of "Convention over configuration". Only unconventional aspects of the application need to be specified.

Ruby on Rails applications are composed of different elements:

Model Models are representing the persistent application data. Ruby on Rails uses ActiveRecord as ORM (Object-relational mapping) for the communication with the database. It manages the access to the database and makes the database objects available to the Rails application. Models contain the business logic. The relations (e.g. `belongs_to` and `has_many`) are defined in the models. This allows the ORM to retrieve relation-objects by performing queries to the database. Models also contain the validation rules for the different attributes.

View The View provides the content of the response to requests. In the case of our application, the views render JSON documents instead of HTML pages.

Controller The Controller is responsible for handling a request. It gets the models from the database and renders the View. Depending on the action, it can also save or update data.

Routes The Rails router is responsible for the mapping between the URLs (and their correspondig HTTP verb) and the Controller's action. The source file `router.rb` contains the declaration of all application routes.

Gems Gems are Ruby libraries that can be installed and used within a Ruby project. A lot of gems are available and they can provide extra functionalities. Rails is distributed as a gem.

When a route (represented by a URL) is accessed, the Rails router dispatches the request to the corresponding Controller's action (represented by a method). The Controller fetches or saves data from a Model and uses the View to produce the output. By default, Rails makes the link between the elements of a resource: the `Run` model is accessed by the `RunsController`. As an example, when a run is accessed, the method `show` of the `RunsController` is called. The controller fetches the corresponding model from the database, and Rails automatically uses the appropriate view to produce the output (i.e. `RunSerializer` in our application).

3.2 Communication between Ruby on Rails and Ember

Usually, Rails applications are developed as monolithic applications. They manage every step from querying the database for getting models data to rendering HTML views. In our application, the View will serve JSON documents for communicating data rather than rendering HTML pages. This allows the client (Ember application) to get the data and manage the view (user interface). Section 5.2 provides more information about the communication between both applications.

3.3 Ember.js

This section presents the main concepts of Ember. More information about the concepts can be found in the Ember Guides [22].

The Ember architecture is based on six main concepts: Model, Route, Controller, Template, Component and Service. The relations between these concepts is depicted on Figure 3.1.

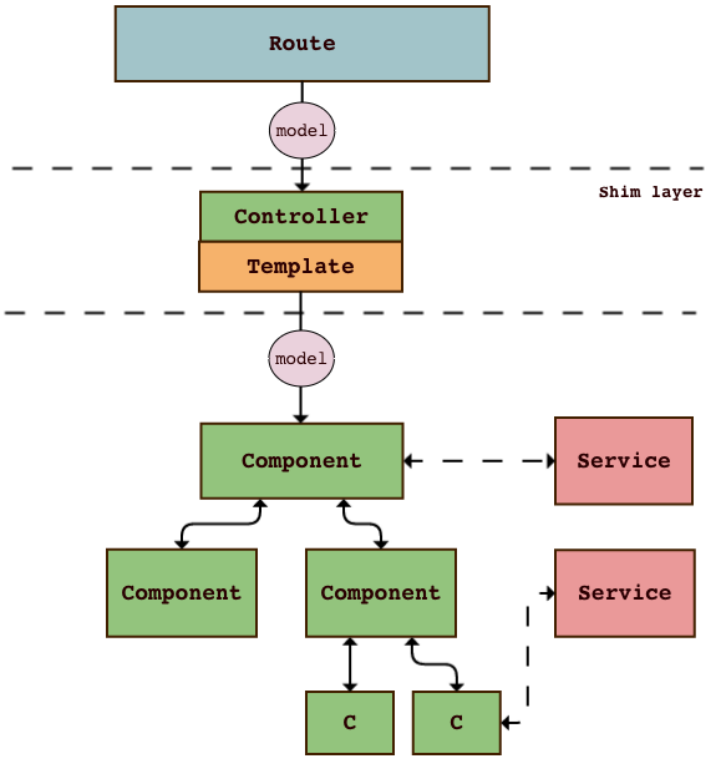


Figure 3.1: Overview of the Ember architecture [23]

When a route (corresponding to a URL) is accessed, the Route object is responsible for retrieving the data needed from the backend API. The Template (layout of the user interface) is displayed using the data. The Template can use multiple Components to organize the layout of the HTML. Each Component (e.g. race selector) manages a part of the user interface. It allows to divide the complexity of the view through multiple independent components. Services provide functionalities that can be used application-wide (e.g. authentication, geolocation service).

Before diving into the main concepts of Ember, it might be interesting to explain some features offered by `Ember.Object` as all the following concepts inherit from it.

3.3.1 Ember.Object

`Ember.Object` is the main base class for all Ember objects. It provides some features that are not present in JavaScript classes (e.g. possibility to define computed properties). Any class can inherit from `Ember.Object` (or any of its subclasses), and mixins can also be applied. Classes can define properties and methods. There exists two kinds of properties:

Property corresponds to a value.

Computed property is a property whose value depends on other properties. They are defined as functions, and the dependent keys must be declared. Ember updates automatically the value of the computed property when the property is requested (lazy evaluation). Computed properties can be chained and depend on other computed properties.

Ember also provides the possibility to use Observers. An Observer is defined as a function. It contains behaviour that reacts to changes in another property (it can be a base-property or a computed property).

3.3.2 Model

Model objects correspond to the underlying data of the application. Ember Data is a library that integrates with Ember to manage models and the communication with the server. Models define *attributes* and relationships. Relationships can be declared as synchronous (nested in the model) or asynchronous (an additional request might be needed to retrieve it).

The Ember Store keeps the *records* used in the application. A *record* is an instance of a model that contains data loaded from a server. Records can also be created by the application, and be saved to the server.

3.3.3 Route

The Ember Router maps the URLs to one (or more, in case of sub-routes) Route handler(s).

The Route can perform the following operations:

- Get the model from the backend API. Then, the `model` property is set on the Controller.
- Render a template.
- Handle some route-actions (e.g. `didTransition`, `willTransition`) to perform additional work (e.g. cancelling changes made to a model object).
- Transition to another route.

3.3.4 Controller

The Controller receives the `model` from the route. It is responsible for setting the properties used in the route-template.

The Controller concept is being phased out in Ember. It will be replaced by routable components[1], which would allow to render directly a Component on a Route. But routable components are not yet implemented in the latest stable version of Ember (2.5.0). For this reason, Controllers are still necessary in some cases.

3.3.5 Template

A Template defines the HTML layout. It can correspond to a Route or a Component. Ember templates can display properties using the Handlebars[2] syntax.

[1] `https://github.com/ef4/rfcs/blob/routeable-components/active/0000-routeable-components.md`
[2] `http://handlebarsjs.com/`

3.3.6 Component

A Component represents and controls an element of the interface. It is composed of a Template and a JavaScript file that defines its behaviour. Components are particularly useful as they define the behaviour of an element that can be used in different routes of the application.

3.3.7 Service

Services are long-lived objects that can be used in different parts of the application. Services are useful for features that require shared state or persistent connections.

3.3.8 Ember Addons

Ember Addons are libraries that can be used in an Ember application. Addons can provide useful functionalities, such as authentication, common components, or integration with JavaScript libraries.

Chapter 4

Design

4.1 Introduction

The goal of the design part was to analyze and describe the user requirements. It is useful to choose the features that should be prioritized and to define an architecture for the application.

There are two kinds of stakeholders in the system: organizers and runners. The focus was put on the features that are important to those users.

We wanted to build a system that helps the organizers through the process of managing their running event. First, the management of the event participations is a must-have. Second, we wanted to provide a way for organizers to present their running event (using pictures and videos) as this functionality is not offered by the existing solutions. Third, we also wanted to allow the organizers to show the route of their event on a map. Moreover, we were interested in integrating online payments and timing results management but due to time constraints, these features were not implemented.

For the runners, the goal was to allow them to be informed about the available running events. They could see the presentation of the event and register easily.

4.2 Use case diagram

The use case diagram (figure 4.1) provides an overview of the relations between the actors of the system. It shows the main functionalities of the system. It does not include all user interactions with the system. Section 4.3 presents all functionalities available to the different users.

The system interacts with 3 actors: Organizer, Runner and Guest.

A Guest is a user who is not yet registered or logged into the system. He can perform general actions: accessing the list of the runs (running events), filtering the runs, and consulting a run page. He can also change the language of the application, but he is not able to register to a race (a particular distance of a running event). A Guest can register and log in as a Runner or an Organizer.

An Organizer is a user who creates runs (and races) for Runners to participate. To promote his run, he is able to add media (pictures and videos). An Organizer must create at least one race for his event (but he can create multiple races for one run). He must also define a registration fee. An Organizer can also create the routes of his races on a map. Furthermore, an Organizer can perform the same actions as a guest (not shown on the diagram): accessing the list of the runs, filtering the runs, and consulting a run page.

A Runner is a user who participates to races. He can register to races created by Organizers. A Runner can register individually or by group, if the group registration has been allowed by the organizer. The same way as other users, he can perform general actions: accessing the list of the runs, filtering the runs, and consulting a run page.

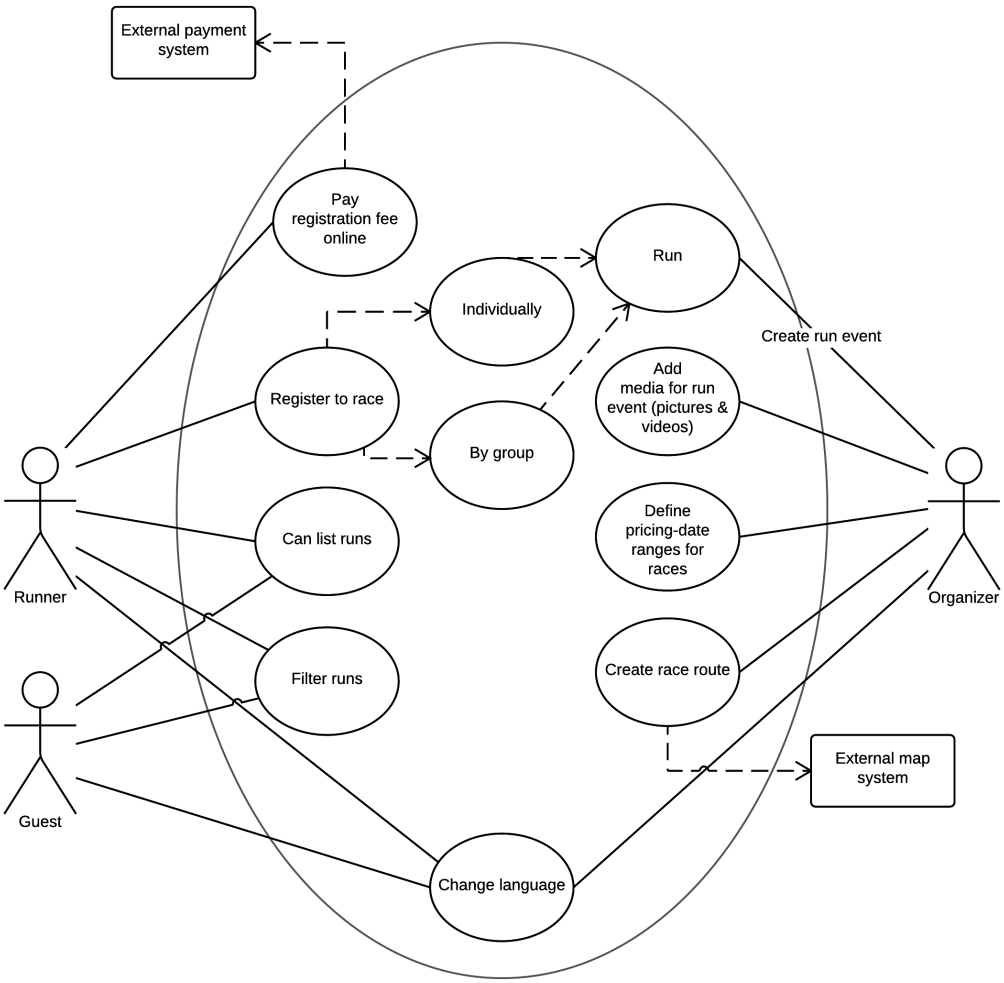


Figure 4.1: Use case diagram

4.3 Context diagrams

The context diagrams provide a more detailed overview of the users contexts, and which features are available to them.

Guest context A guest is a user who is not yet registered or logged into the system. A guest can register or log in as a runner or an organizer (figure 4.2). The other actions are available to all users: consult the list of runs (cf. section 4.4.1), filter runs (section 4.4.2), consult run page (section 4.4.3).

Runner context When a user is logged as a runner, he can perform certain actions (figure 4.3). The available features are described in section 4.4.

Organizer context The organizer is the user who has the most actions available (figure 4.4). The features available to organizers are described in section 4.4.

4.4 Features

Features from sections 4.4.1 to 4.4.3 are available to all users. Sections 4.4.4 to 4.4.9 are specific to organizers. Section 4.4.10 is available to organizers and runners. Section 4.4.11 is dedicated to runners.

4.4.1 Upcoming events

Organizers and runners can see the list of the upcoming running events, as shown in figure 4.8. They have instant access to the main data of an event (name, date, city, distances, price and number of participants). When the user clicks on a row, the page of the event is shown.

4.4.2 Filtering events

On the page of the upcoming events, a click on the Search button make the search fields appear. As shown in figure 4.9, the list of the events can be filtered by filling the search fields.

4.4.3 Run page

A run page allows an organizer to present his running event, manage the races and the registrations. The first two "tabs" (Presentation and Information) are available to all users.

The Presentation tab (figure 4.10) shows the pictures and videos related to the event. At the bottom stands a textual description of the event.

The Information tab (figure 4.11) displays the practical information a runner might need. At the left of the screen, there are the date, address, distance, price, and participants information. The runner can choose the distance he prefers. Depending on the distance, the price and the route (shown on the map) update accordingly. At the bottom of the page, some additional information might give more precise insight about the course of the event (e.g. What is the schedule like? Are there parkings?).

When a user is on a run page, multiple actions are available depending on the type of user.

An organizer visiting one page of his own events can edit (and delete) it. Concerning the event pages of other organizers, he is only allowed to visit them.

A runner visiting a run page can register to one of the races proposed.

4.4.4 Run creation and edition

When an organizer creates a running event, he must fill in the fields, as shown on figure 4.12. Upon the creation of an event, the Registrations tab should not be present.

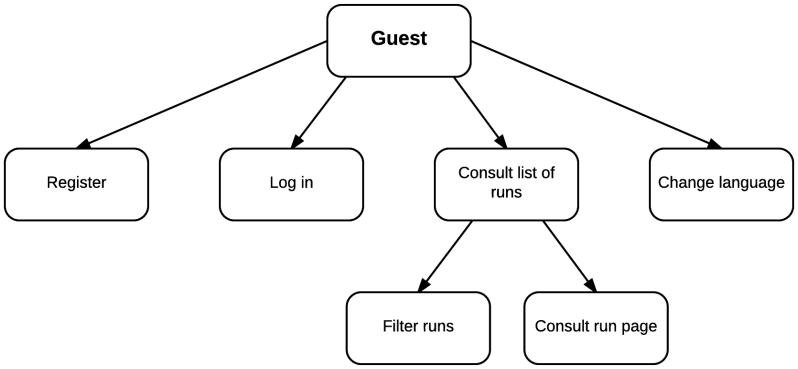


Figure 4.2: Context diagram – Guest

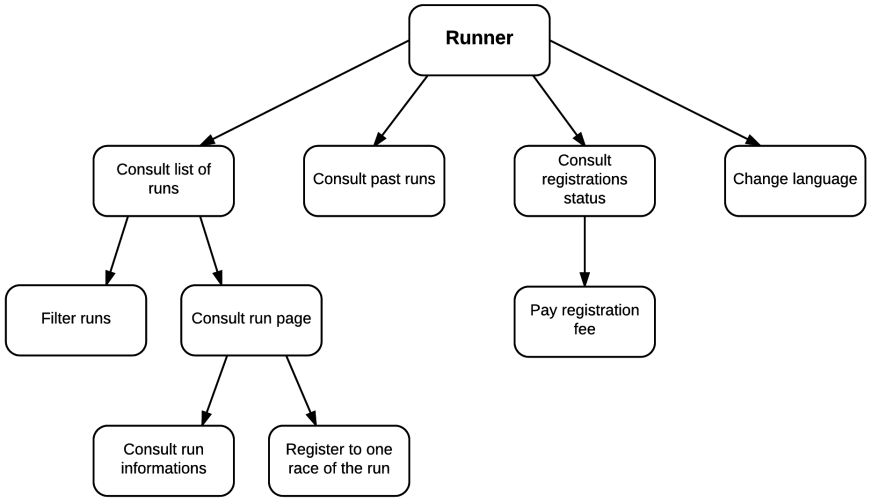


Figure 4.3: Context diagram – Runner

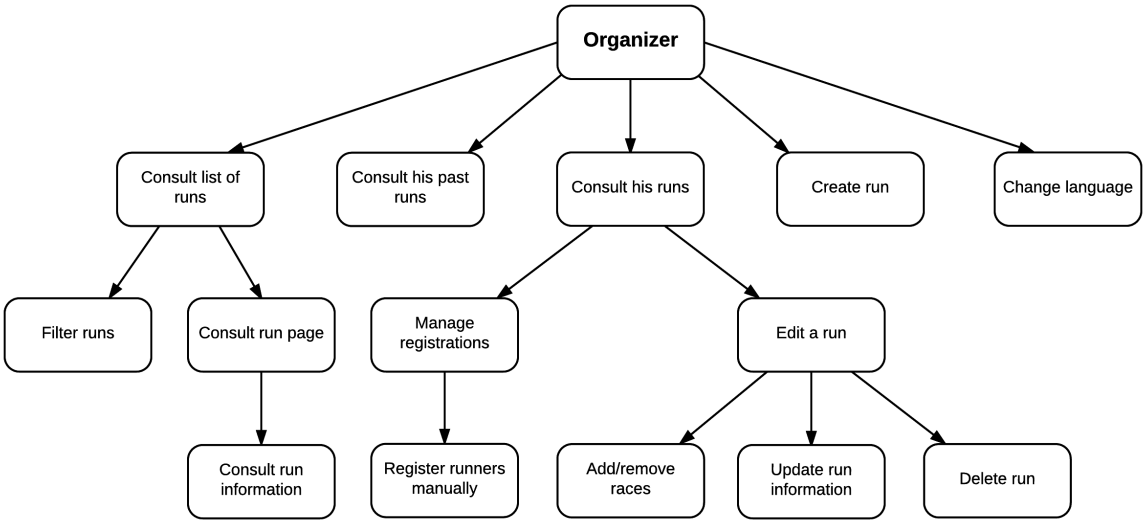


Figure 4.4: Context diagram – Organizer

Use hierarchy diagram

The use hierarchy diagram (figure 4.5) shows the actions that must be fulfilled in order to create an event. The actions highlighted in green are optional.

Here is a description of the different information items of an event :

Name of the run Identifies the running event.

Media The organizer can add pictures and videos to promote his run.

Description General presentation of the run.

Date & hour Date at which the run happens.

Location The organizer must define the address (street and city) of the running event.

Define races A run must be offering at least one race, and might propose multiple races. A race contains a distance, a price and an optional route.

Distances of the races A run might provide multiple races of different distances (e.g. 5km, 10km and 15km).

Define prices Price corresponding to a race.

Race route Route of the selected race displayed on a map. The operation is explained at section 4.4.6.

Maximum number of participants Limit of participants (common to all races of the run event).

Additional information Practical information about the races.

Event state diagram

A run can go through different states:

Created The organizer created the running event, but the registrations are not opened yet.

Registrations open When the first pricing-date passed, the registrations become opened.

Registrations closed When the last pricing-date passed, the registrations become closed.

Happening At that state, the organizer might be able to register runners manually. It can be useful to register new participants on the day of the event.

Terminated The running event is over, and no more changes should happen.

4.4.5 Add media on an event page

The videos should be stored on YouTube (or another video service), and the organizer should enter the link of the video. The pictures should be available through cloud service, and the link should be provided.

4.4.6 Draw the route of a run on a map

During the creation/edition of a run, the organizer should be able to define the route of a race on a map. He could add markers on the map, and the route would be drawn between the markers. The markers should be ordered. He could also remove and drag the markers to a new position.

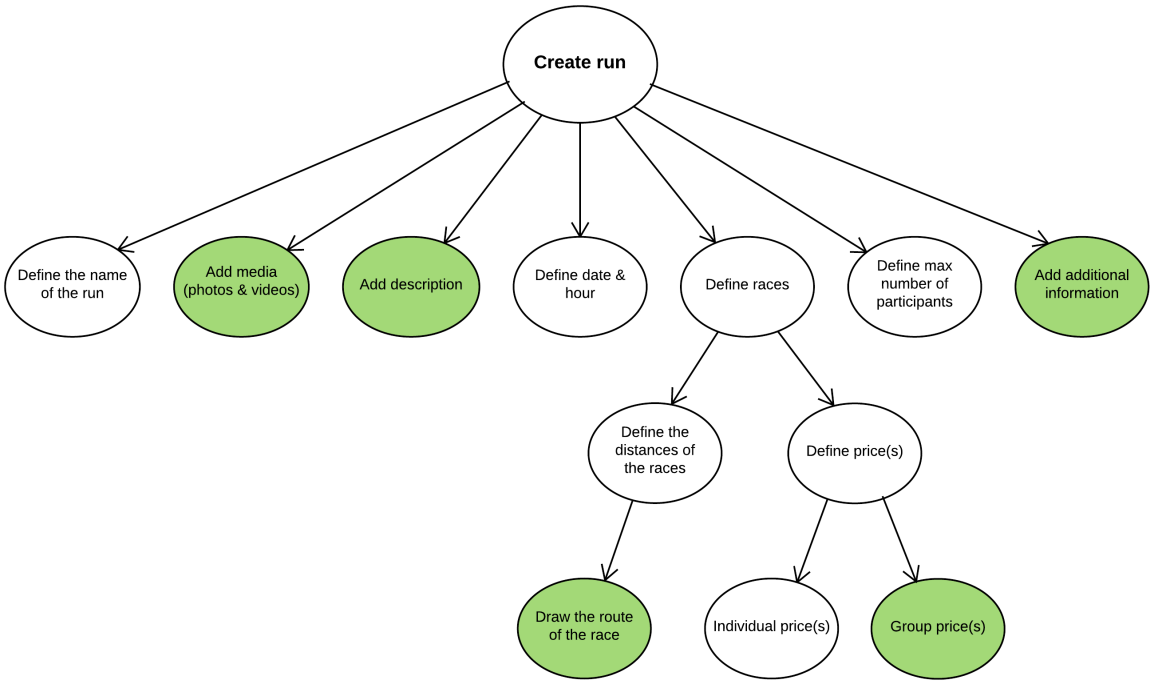


Figure 4.5: Use hierarchy – Create run

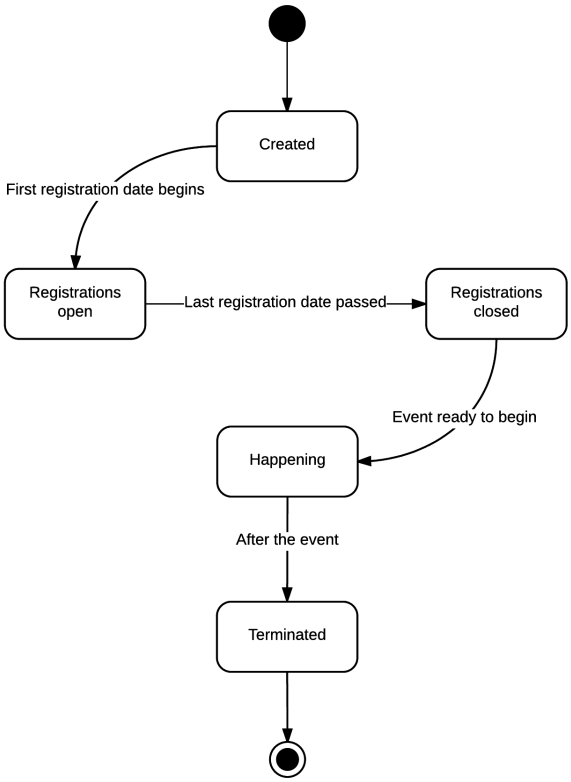


Figure 4.6: State diagram – Running event

4.4.7 Manage registrations

An organizer should be able to consult the runners participating to his races. He should be able to filter the runners by race, and find a runner using a search field. An organizer should also be able to register runners manually, even if they are not yet registered onto the platform.

4.4.8 Run edition

The organizer of the run must be able to perform the following actions:

- Update the name of the run
- Update the description
- Update the date
- Update the address (street and city)
- Update the races. The organizer must be able to change the distances, the prices and the route.

4.4.9 Run deletion

To delete one of his events, an organizer must first go in editing mode. Then he has to click on the "Delete run" button, and a confirmation dialog appears.

4.4.10 Personal list of runs

The users of the application should be able to access a personal list of the running events, as shown on figure 4.14. For organizers, it corresponds to the runs that they created. For the runners, it should include the runs corresponding to one of the races they participate to. Runners are also able to consult their registration status to the different races.

4.4.11 Register to a run

A runner can register to one race of a run. As shown on figure 4.7, the registration of a runner can have 3 different states :

Unregistered At first, the runner is considered as Unregistered to a run. When he registers to a run (using the "Participate" button on a run's page), he becomes Pre-registered.

Pre-registered A Pre-registered runner means that he is taken into account in the number of participants to the event. But he is not yet *officially* registered to the event. At this point, he can still cancel his registration. When the runner fulfills the payment, he becomes Registered.

Registered The runner is *officially* registered to (one race of) the run.

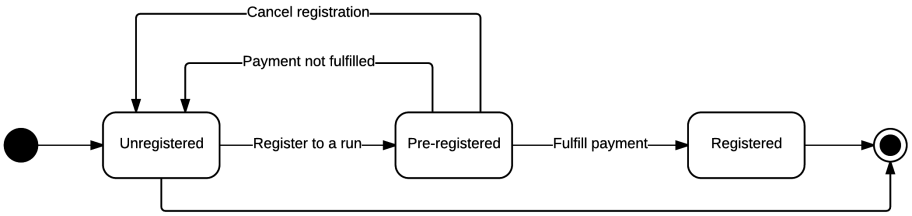


Figure 4.7: Registration – State diagram

4.4.12 Wireframes

The following wireframes have been created to illustrate the different functionalities that should be offered by the system.

On figure 4.8, the user sees the list of the upcoming runs. Clicking on a row shows the corresponding run page.

Figure 4.9 shows how users could perform a search to filter the runs.

Figure 4.10 represents the run page as seen by the organizer. It shows the presentation, where videos and pictures can be used to present the running event with a description at the bottom.

The practical information of the event and the different races are shown on figure 4.11. The date, address and number of participants are available and common to all races. When the distance (race) is selected, the price and the route on the map are updated accordingly. Runners can participate to the race selected by clicking on the "Participate" button.

Figure 4.12 shows the creation of a running event by the organizer. He can define the different information related to the run and races.

Figure 4.13 allows the organizer to see the runners information and manage their participation to the races.

Finally, figure 4.14 shows the personal list of runs for an organizer. He can see the available runs on the top of the screen. He has still access to the past runs he created.

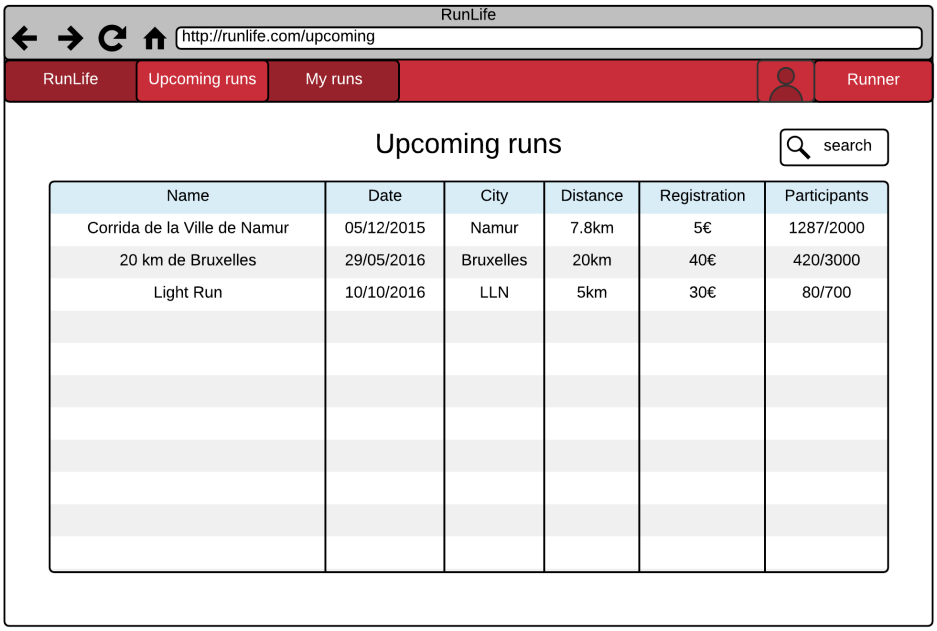


Figure 4.8: Wireframe – Upcoming runs

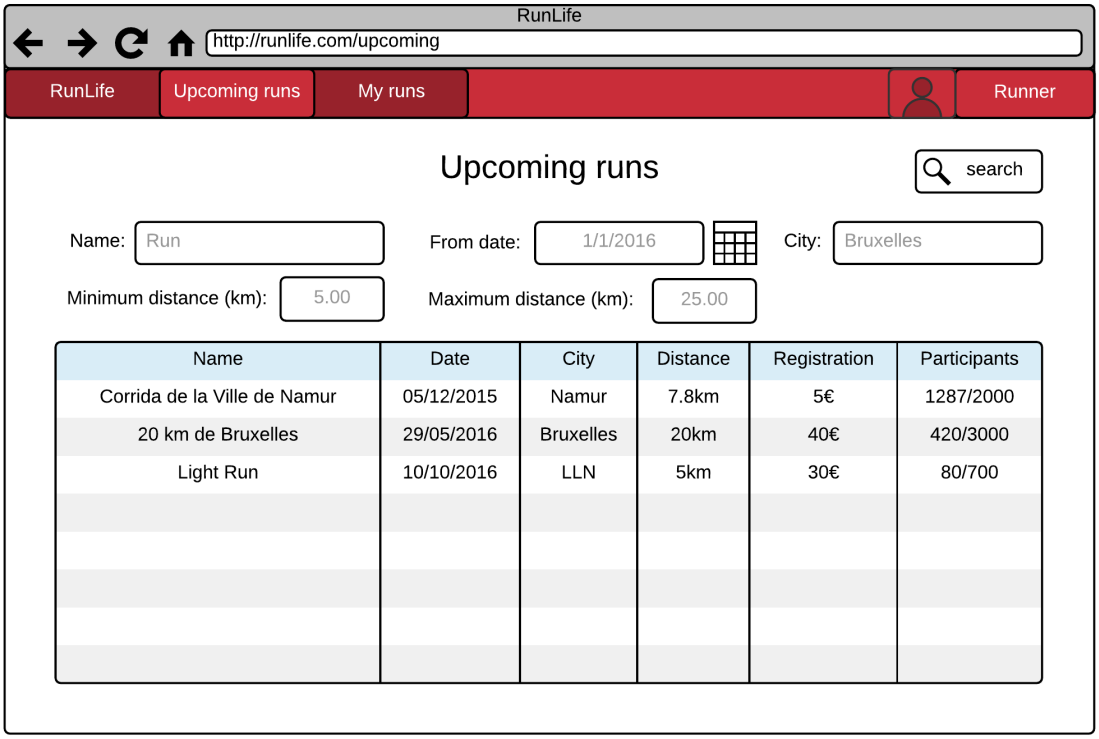


Figure 4.9: Upcoming runs – Filter

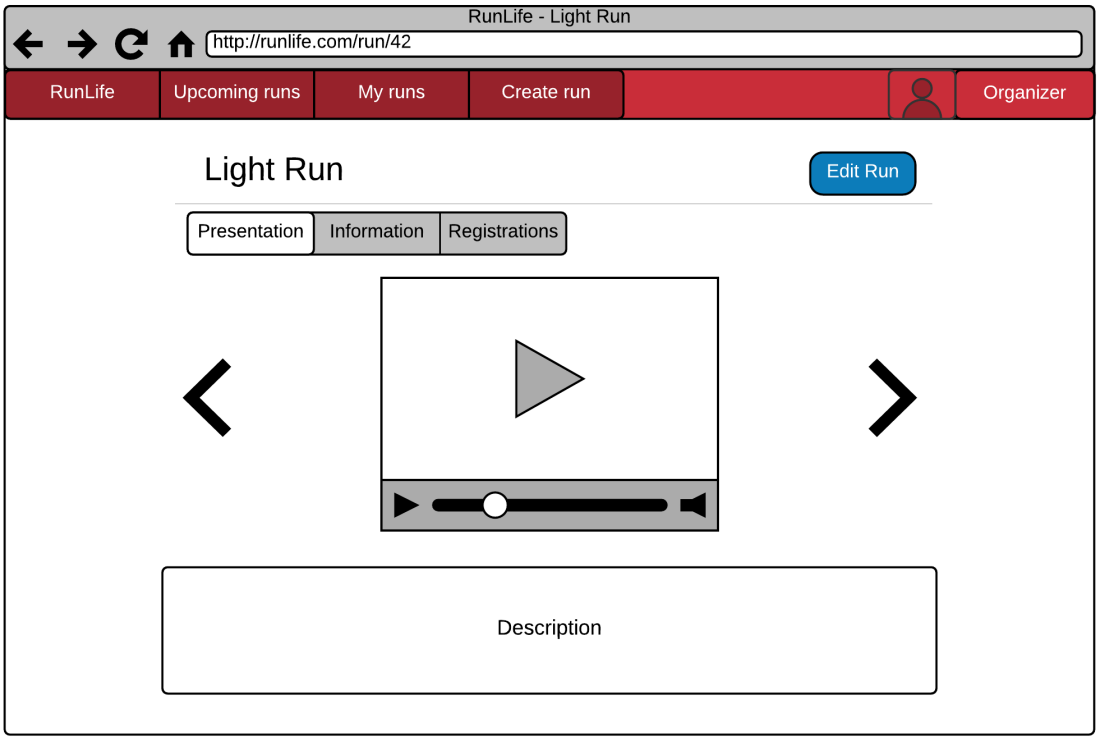


Figure 4.10: Wireframe – Run page (organizer)

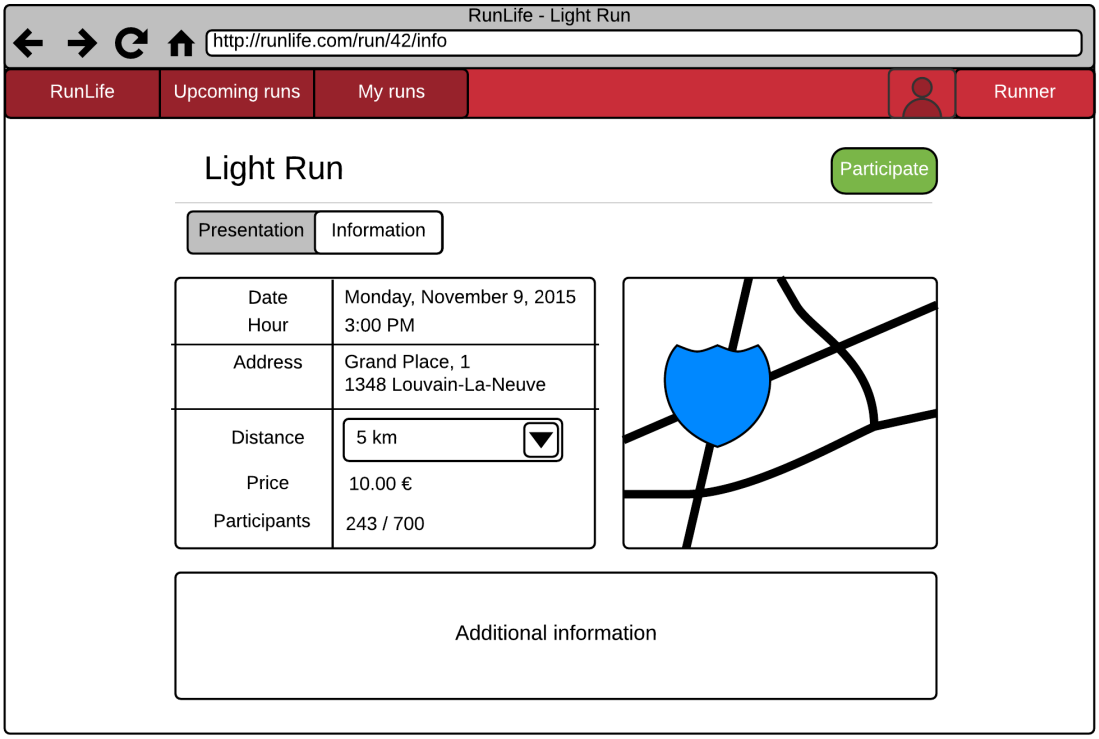


Figure 4.11: Wireframe – Run information (runner)

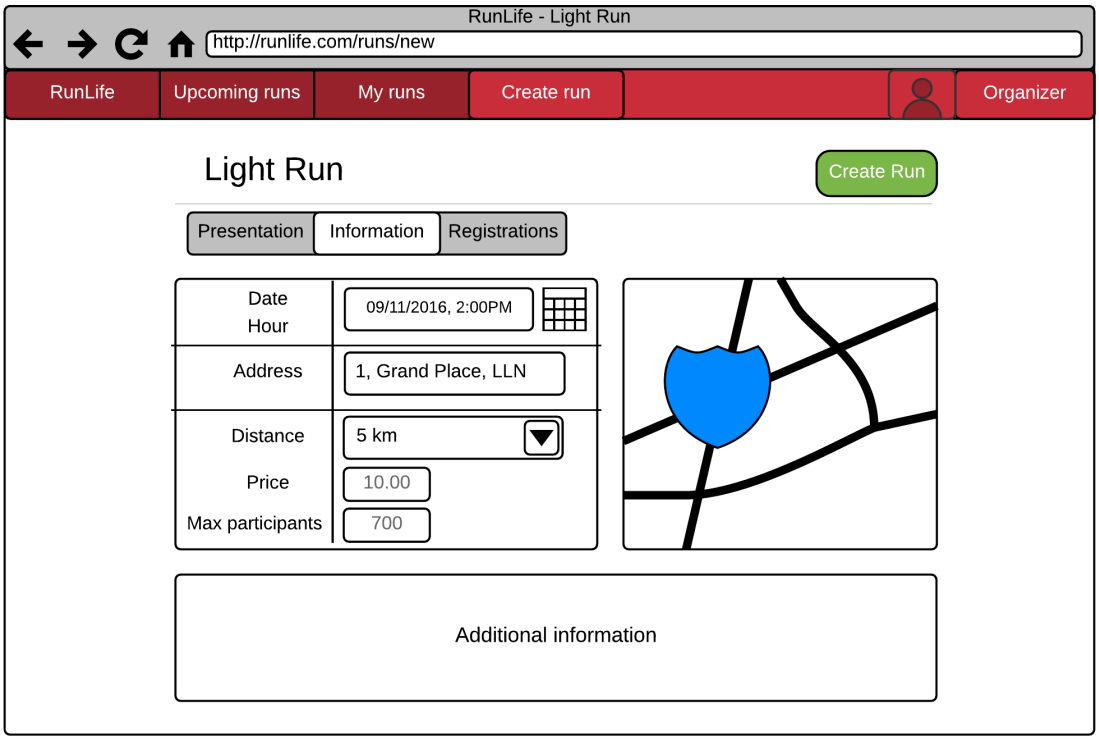


Figure 4.12: Wireframe – Run creation (organizer)

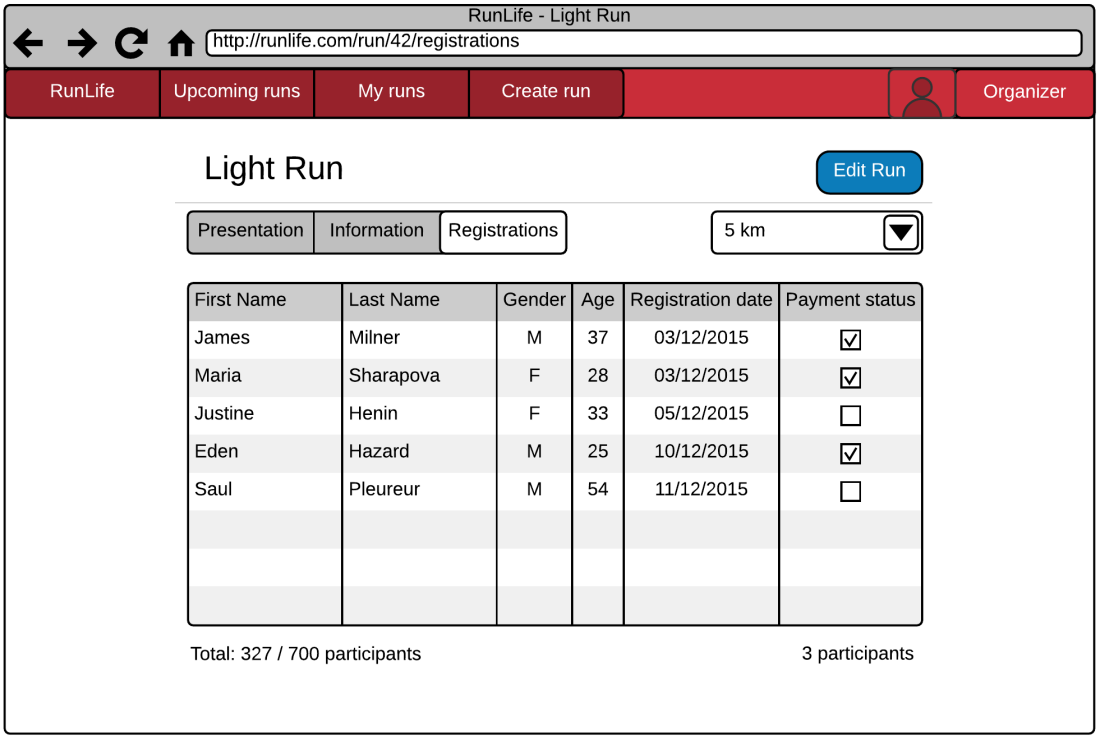


Figure 4.13: Wireframe – Run registrations (organizer)

Figure 4.14: Wireframe – My runs (organizer)

4.5 ERD diagram

Figure 4.15 depicts the ERD diagram of the database of the application.

User An abstract entity representing a user. It regroups the common attributes of Organizer and Runner. We created the Organizer and Runner to allow the validation rules to be defined separately for each type of user. Certain attributes (e.g. phone number) can be optional for runners and mandatory for the organizers.

Runner User who can register to Races. A Runner might register individually or by group, depending on the availability of the corresponding Races.

Organizer User who can create Runs (and the underlying/corresponding Races).

Media Pictures and videos associated with a Run. It corresponds to an URL as these media will be stored on external services. The `index` attribute is used to order the media of a Run.

Run Entity corresponding to a running event. A Run regroups the common information about the different Races. A Run can have many corresponding Races. A Run is happening at an address (composed by `street` and `city`) and at a specified `date`. It has also a maximum amount of participants allowed to register to the event (`max_participants`). This constraint holds among the different Races of the Run. Runners cannot register to Runs.

Race Sub-entity of an Run. A Race corresponds to a particular `distance` of a Run. Runners can only register to one Race of a particular Run.

PricingDateRange Regroups the pricing-date ranges of a Race. The registration fee of a Race might change depending on the date of the registration. This entity specifies the registration `price` between the dates `from` and `to`. It also defines whether it corresponds to a group price or not with the attribute `is_group_price`. *This entity is not included in the final implementation.*

RoutePoint Point of a Race route. A Race route is composed by many RoutePoints. The `index` indicates the position of the point in the Race route.

RaceParticipation Registration to a Race. It might be a GroupRegistration or an individual RunnerRegistration. The attribute `participation_type` is used to know the type of the participation (group or individual). It contains a `created_at` (which is automatically handled by Rails), a `payment_date` and the `total_payment` amount. The latter attribute is used to determine if the runner paid the total amount of the registration fee (as he could have made a mistake and paid a different amount).

RunnerParticipation Individual participation of a Runner to a Race.

GroupParticipation Participation of a group of Runners. It corresponds to a set of RunnerParticipations. *This entity is not included in the final implementation.*

RunnersGroup Association between GroupParticipation and RunnerParticipation.

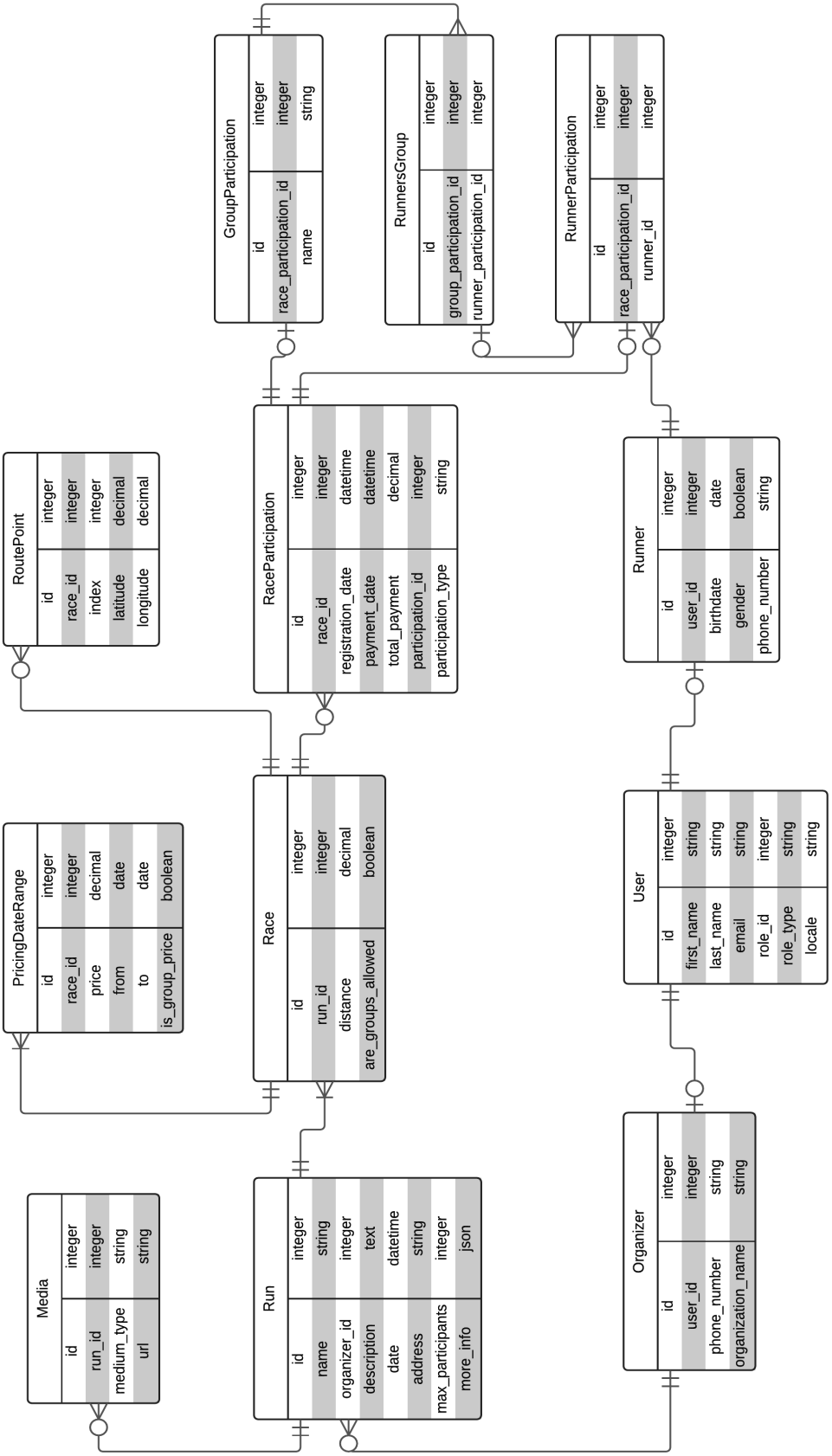


Figure 4.15: ERD diagram

Chapter 5

Implementation

This chapter will begin by presenting the general architecture of the application. Then, the structure of the Ember application and the Rails backend API will be explained. The communication between both parts will be presented. Finally, the implementation of the different features is showcased.

5.1 Application architecture

5.1.1 High-level overview

For this project, the Rails and Ember applications have been developed as two separate applications. The server runs the Rails application and the database. The Ember application runs in the browser of the device (phone, tablet, desktop) accessing the application. It manages the user interface and sends requests to the Rails application.

Figure 5.1 provides a high-level overview of the application architecture. The Rails backend gets the models from the database and serves JSON documents through its API. The Ember frontend retrieves the data from the backend API and displays the interface to the end user. The Ember application sends requests to the backend for retrieveing, creating, updating and deleting resources. The backend receives the requests, controls them and effectively modifies the database.

Each application has a different role. The backend (Rails) manages the persistent state (database) of the application. It provides an API to the frontend (Ember). The API is composed of different URLs for each resource (runs, races, participations, etc.). The frontend sends requests to the backend API to get or modify the persistent data. For example, the frontend can retrieve the data about a race by sending the HTTP request `GET /races/9`. A run can be created with the request `POST /runs`. The communication between both applications is performed by using JSON documents (see section 5.2).

5.1.2 Ember application

The Ember framework provides a general architecture for the applications. In Ember, the URL plays a central role: it corresponds to a certain state of the application.

We chose to define a Route for every URL of the application. For example, the run page (`/runs/9`) contains the following subroutes corresponding to each tab:

Presentation Presentation of the run. This is the default route when accessing a run page (`/runs/9`).

Information General information about a run. It allows to select a race and consult the route of the race. It corresponds to the `/runs/9/info` URL path.

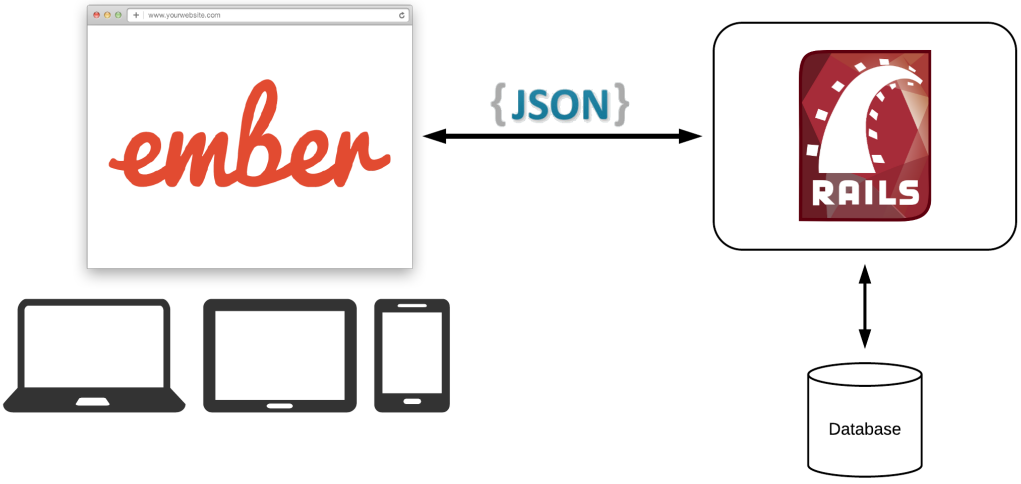


Figure 5.1: Overview of the application architecture

Races It allows the organizer to manage the races (with their corresponding price and route) of his run. This route is only accessible to the organizer of the event. Its URL path is `/runs/9/races`.

Registrations The organizer can see and manage the registrations to the different races of his running event. He has access to runners information and he can update their registration status. The URL path for this route is `/runs/9/registrations`.

The same functionality could have been performed without defining subroutes, by updating the content of the view. But we chose to define these subroutes for allowing the users to see the same page when they share URLs.

Each route has a corresponding template defining the layout of the user interface. These templates use components to create the final view.

We chose to create many components for the application. These helps to divide the complexity of the user interface by controlling a part of the view. Components can be used in multiple places of the application. Using components also improves the modularity of the architecture as components can be easily moved from one place to another, making it easier to modify the user interface. Here is a description of some components we defined for the application:

run tabs We defined one component for each tab of a run page: `run-presentation`, `run-infos`, `run-races`, `run-registrations`. Each of these components uses other components to build the view. For example, the `media-carousel` component is used in `run-presentation`. It displays the videos and pictures of a run in the slider. The `route-map-editor` component displays the route of the race on a map. It is used in `run-infos` and `run-races`. As these components are independent (they do not depend on the context where they are used), they can be used in any place of the application. We did this because we wanted to provide the same user interface for creating and editing a run. In both routes (for creating a run and during run edition), the same components cited above are used.

run-infos Each attribute shown in the Information tab is managed by a component (e.g. `run-infos/run-datehour` for displaying the date and hour). Each component displays its data and manages the layout in edition mode. Figure 5.2 shows the structure of the view using the Ember-Inspector [16]. We created such components to facilitate the

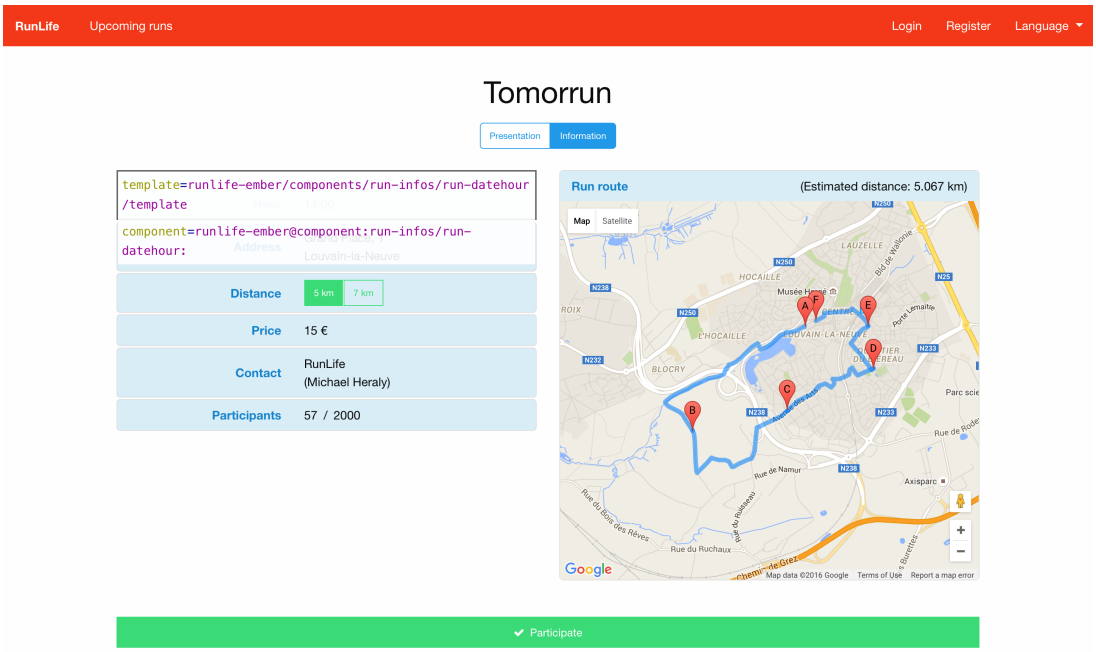


Figure 5.2: Ember Inspector – Run information components

modification of the user interface and allowing to move any information to another view of the application.

race-selection This component displays the available races and allows the user to select a race. It is used in the `run-infos`, `run-races` and `run-registrations` components.

run-page This component manages the different states of a run page: show or edition mode. It is responsible for handling the "cancel" and "save" actions performed by an organizer after editing his running event. This component has been defined to share this behaviour between the routes for creating a run and editing a run.

route-map-editor This component displays the route on the map and manages the different actions an organizer might perform. It creates and updates the RoutePoints when the organizer adds, removes or drag a marker on the map.

Some features require a shared state among the application and do not necessarily need a user interface. These include the authentication and the geolocation API. We created the following services:

session This service manages the authentication. It maintains the user session and the locale of the application.

Google-Maps This service is used to communicate with the Google Maps JavaScript API [24]. It acts as a wrapper for functionalities offered by the API. It contains functions to geocode an address, compute a route for multiple points and calculate the distance of a route.

The `session` service is used in multiple parts of the application for the components to adapt their view depending on the user session. The `Google-Maps` service is used in the `route-map-editor` component. The `route-map-editor` component uses the Google Maps API to display the map and managing the markers. We could call the Google Maps API directly in the component to compute the route (or calculating the distance), but we prefered to create a

service to separate the concerns between the display and the communication with an third-party API.

The Ember application is structured with the pod structure. The pod structure uses one directory by route. The directory structure is shown in figure 5.3. In each route directory, we can find 3 files: `route.js`, `controller.js` and `template.js`. The components are stored in `app/components`. Every component has its own directory, regrouping two files: `component.js` and `template.js`.

Other directories contain files serving other purposes:

application This directory contains 4 files:

- **adapter.js** declares the adapter used by the application (i.e. DS.JSONAPIAdapter).
- **serializer.js** declares the serializer used by the application (i.e. DS.JSONAPISerializer).
- **route.js** defines the route that gets fired the first time (of the session) when the user access the application. In particular, it defines the user locale.
- **template.js** defines the template for the whole application. The routes defined in `app/router.js` will insert their template within the application template.

authenticators contains the authenticators supported by the application. More explanations are provided in the section 5.3.1.

authorizers contains the authorizers supported by the application.

helpers contains files that provide more functionalities to the templates. We did not add custom helpers for the application as we did not need more.

mixins contains mixins that are used in other parts of the application.

models contains one file for each model of the application (Run, Race, User, etc.).

services contains services that are used throughout the application:

- **ajax.js** defines a custom Ajax service. It is used when a custom request must be performed to the backend API. It automatically includes the authorization header.
- **google-maps.js** is a wrapper for calls to the Google Maps JavaScript API [24].
- **session.js** contains logic related to the user session. It keeps a reference to the current user and the session locale.

styles contains the stylesheets of the application (SASS files).

templates contains files describing HTML content (using Handlebars expressions).

The `app` directory also contains some general purpose files:

app.js configuration of the Ember application.

formats.js contains formats generated by the `ember-intl` addon [25].

index.html file regrouping the assets used by the Ember application (JavaScript and CSS files). This file is served to start the application.

resolver.js file generated by Ember to resolve the files of the application.

router.js contains the declaration of the application routes.

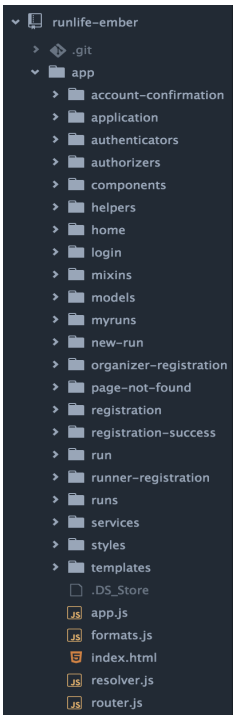


Figure 5.3: Ember application – Project structure

5.1.3 Rails server API

For the backend API, we chose to define a flat-structure for the routes. It means that the resources can be accessed (and modified) independently. The routes defined by the backend API are shown on figure 5.4. Each resource (run, race, runner participation, etc.) has routes for interacting with the persistent data:

GET /resource Retrieve all the resource models. This route might receive query parameters (e.g. `GET /runs?user=1`) for filtering the resources to be included in the response.

POST /resource Create a new resource model. The client request must provide JSON parameters to create the resource.

GET /resource/:id Retrieve the resource whose id is `:id`. It allows to retrieve one resource model at a time.

PUT /resource/:id Update the resource whose id is `:id` with the new attributes provided by the client request. PUT and PATCH requests are handled in a similar way. The difference between PUT and PATCH is that PUT replaces the entire resource where PATCH only applies the requested modifications to the resource.

DELETE /resource/:id Delete the resource whose id is `:id`. This action is permanent and removes the resource from the database.

We think it is an interesting architecture as each resource can be accessed independently from the others by the client. An alternative solution would be to define nested routes, but in that case the API might have to change radically if the relations between the models change. For example, the route API for creating a new RoutePoint would be `POST /runs/:id/races/:race_id/route_points`. This might cause problems if any of the Run or Race resource changes its structure. If a change would imply to remove the Race model as runs could only contain one distance by event. In that case, the backend API for creating a new RoutePoint would need to change radically and become

`POST /runs/:id/route_points`. With the flat-structure API, this is not a problem as the API stays the same (i.e. `POST /route_points`). Only the JSON parameters sent by the client needs to be adapted.

We use the same flat-structure for the other elements of the Rails application. For example, the Run resource defines three elements: Run model, RunsController and RunSerializer. Each application resource (run, race, runner participation) defines the following elements:

Model Manage the interaction with the database. It defines relations with other models and validation rules.

Controller Handle the request for the resource. It defines one method for each possible type of request (index for GET, create for POST, show for GET :id, update for PUT/PATCH and destroy for DELETE). It retrieves the models from the database and updates the models depending on the client request.

Serializer Define the attributes to be included in the JSON responses.

The remaining of this section presents the elements of the Rails framework that are used in the application.

Database

The Rails server uses a PostgreSQL [26] database. The database queries are managed by the ActiveRecord interface, the ORM (Object-relational mapping) provided by Rails.

Models

The models are located in `app/models`. Each model is an `ActiveRecord`. It declares the relationships with the other models, and defines the validation rules. The application contains the following models: User, Organizer, Runner, Medium, Run, Race, RoutePoint and RunnerParticipation. To facilitate the communication with the frontend, as the frontend is only aware of the User model (and its type: organizer or runner). The Organizer and Runner models is an implementation choice of the backend to manage the different User models. Therefore, the RunnerParticipation model has a direct relationship with Race and User.

Serializers

Each model of the application has a corresponding serializer. The serializers are located in `app/serializers`. Their role is to define which attributes and relationships should be serialized when responding to client requests. Each resource has its own serializer: User, Medium, Run, Race, RoutePoint and RunnerParticipation.

Controllers

The controllers are defined in `app/controllers`. They are responsible for handling the requests and producing the appropriate output. The controllers might receive parameters via the URL (e.g. `/api/runs/:id`) or through request parameters (in JSON format). Most resource controllers define five methods: `index`, `show`, `create`, `update` and `destroy`. These methods are called by the Rails router when it receives the appropriate request. The mapping between the routes and the controller methods are defined in `config/routes.rb`.

```
                Prefix Verb   URI Pattern
organizer_registration POST   /api/user/register_organizer(.:format)
   runner_registration POST   /api/user/register_runner(.:format)
     user_confirmation GET    /api/user/confirm/:confirmation_token(.:for
            user_login POST   /api/user/login(.:format)
                 users GET    /api/users(.:format)
                  user GET    /api/users/:id(.:format)
                       PATCH  /api/users/:id(.:format)
                       PUT    /api/users/:id(.:format)
                 media GET    /api/media(.:format)
                       POST   /api/media(.:format)
                medium GET    /api/media/:id(.:format)
                       PATCH  /api/media/:id(.:format)
                       PUT    /api/media/:id(.:format)
                       DELETE /api/media/:id(.:format)
                  runs GET    /api/runs(.:format)
                       POST   /api/runs(.:format)
                   run GET    /api/runs/:id(.:format)
                       PATCH  /api/runs/:id(.:format)
                       PUT    /api/runs/:id(.:format)
                       DELETE /api/runs/:id(.:format)
                races GET    /api/races(.:format)
                       POST   /api/races(.:format)
                 race GET    /api/races/:id(.:format)
                       PATCH  /api/races/:id(.:format)
                       PUT    /api/races/:id(.:format)
                       DELETE /api/races/:id(.:format)
 runner_participations GET    /api/runner_participations(.:format)
                       POST   /api/runner_participations(.:format)
  runner_participation GET    /api/runner_participations/:id(.:format)
                       PATCH  /api/runner_participations/:id(.:format)
                       PUT    /api/runner_participations/:id(.:format)
                       DELETE /api/runner_participations/:id(.:format)
          route_points GET    /api/route_points(.:format)
                       POST   /api/route_points(.:format)
           route_point GET    /api/route_points/:id(.:format)
                       PATCH  /api/route_points/:id(.:format)
                       PUT    /api/route_points/:id(.:format)
                       DELETE /api/route_points/:id(.:format)
 frontend_confirmation GET    /user/confirm/:confirmation_token(.:format)
```

Figure 5.4: Backend API routes

Routes

The backend provides routes API to interact with the data. Figure 5.4 shows the different routes available to client applications. These are defined within the `/api/` namespace. The system provides a RESTful [27] API. It means that clients can communicate with the server using HTTP requests with the different HTTP verbs (GET, POST, PATCH, PUT and DELETE). Each resource (run, race, media, etc.) has its own routes with the possibility to retrieve, create, update and delete the corresponding resource.

5.2 Frontend – backend communication

Although the client and server have been developed as two separate applications, they have to agree on a common communication format. We chose to use JSON data to this end.

JSON documents can be structured in many different ways. Fortunately, a JSON API [28] exists and defines shared conventions. It defines how a resource should be formatted, how the

relationships should be communicated, how include nested relationships, etc.

Figure 5.5 shows an example of JSON response sent by the backend API. This response is sent when the client requests run information (i.e. `GET /api/runs/1`). In this example, we can observe that the run attributes are included as key-value fields. The relationships are declared, and only the id (and the type) of the corresponding resource is communicated. A Run has relationships with an organizer, some media, and multiple races. In the example, the media are included (nested) in the response but these were omitted in the figure due to space limitation. The frontend must perform additional requests to retrieve the organizer and races information.

Implementing the JSON API by hand would not be practical. Any modification concerning the attributes and relationships would require non-trivial modifications and be error-prone. To help us implementing the JSON API in the application, we used some libraries.

5.2.1 JSON API – Backend implementation

For the backend, we used the ActiveModel::Serializers gem. [29] It provides certain facilities to serialize Rails models into JSON respecting the JSON API format.

When a Controller is responding to a request, it first retrieves the models from the database. It may perform additional work (e.g. filtering the models). When the Controller is ready to respond to the request, it calls the method `render json: run` (`run` being the model to serialize). The corresponding serializer is automatically called and the JSON is sent back to the client.

In the Rails backend API, all models have their corresponding serializer. Figure 5.6 shows the `RunSerializer` implementation in the Rails application. The relationships are declared using the `belongs_to` and `has_many` methods. The attributes are declared using the `attribute` directive. In this example, the `belongs_to :organizer` directive is overridden to resolve the User object (as the frontend is only aware of the User model, and its type). The `number_of_participants` is a method of the Run model and it is defined as an attribute to the clients. In particular, this attribute is useful in the "Upcoming runs" feature, for avoiding sending all races and all participations for the client to compute that value.

5.2.2 JSON API – Frontend implementation

For the frontend, we used the DS.JSONAPIAdapter and DS.JSONAPISerializer provided by the Ember-Data library.

Figure 5.7 depicts the architecture used by Ember to retrieve a model from a server. When a model is requested (within a route or a controller), the application will query the Store for the record. The Store immediately returns a Promise [17]. A Promise allows to perform asynchronous computations. It represents a proxy for a value not necessarily known when the promise is created. The first time, the Store will not have the record requested, so it will ask the Adapter to perform an asynchronous request to the server. When the server responds, the Adapter receives the JSON payload. The JSON payload is normalized by the DS.JSONAPISerializer. Then, the promise returned to the Store is resolved with the JSON data. Finally, the Store resolves the promise with the corresponding record to the Ember application.

5.3 Features implementation

5.3.1 Authentication

We used two libraries to implement the authentication. On the backend side, the Devise [31] gem provides an authentication solution for Rails. On the frontend side, the Ember-Simple-Auth [32] addon provides authentication for Ember applications and supports multiple authentication mechanisms, including the interoperability with Devise.

```
{
  "data": {
    "id": "1",
    "type": "runs",
    "attributes": {
      "name": "Tomorrun",
      "description": "Enjoy the run!",
      "date": "2016-11-09T13:00:00.000Z",
      "street": "Grand Place, 1",
      "city": "Louvain-la-Neuve",
      "max_participants": 2000,
      "more_info": [
        {
          "key": "Bonus d'inscription",
          "value": "Un t-shirt Tomorrun!"
        },
        {
          "key": "Programme",
          "value": "Depart des 7km a 14h. Depart des 5km a 14h30."
        }
      ],
      "number_of_participants": 57
    },
    "relationships": {
      "organizer": {
        "data": {
          "id": "1",
          "type": "users"
        }
      },
      "media": {
        "data": [
          {
            "id": "5",
            "type": "media"
          },
          {
            "id": "6",
            "type": "media"
          },
          {
            "id": "7",
            "type": "media"
          },
          {
            "id": "8",
            "type": "media"
          },
          {
            "id": "9",
            "type": "media"
          },
          {
            "id": "10",
            "type": "media"
          }
        ]
      },
      "races": {
        "data": [
          {
            "id": "1",
            "type": "races"
          },
          {
            "id": "2",
            "type": "races"
          }
        ]
      }
    }
  }
}
```


Figure 5.5: Example of backend response (JSON API format)

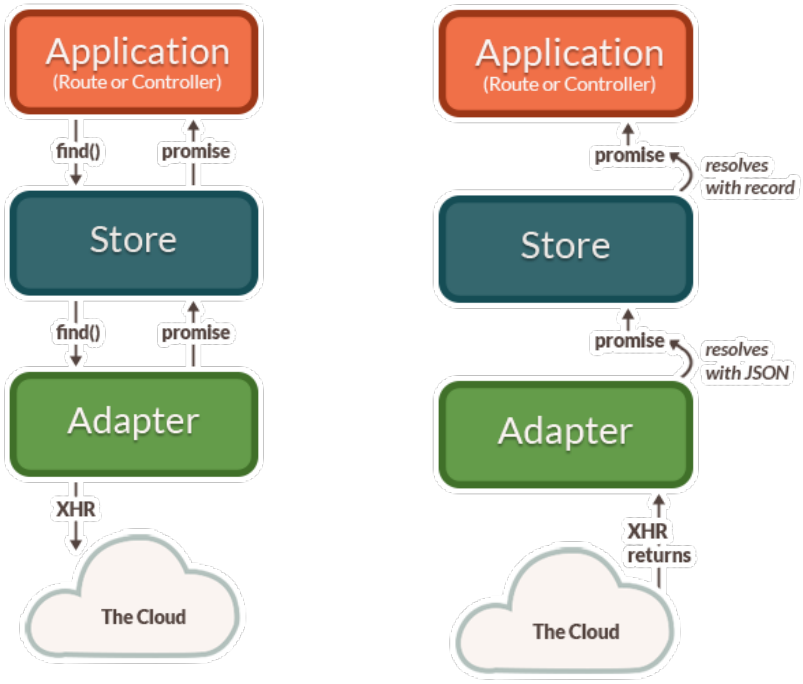
```ruby
class RunSerializer < ActiveModel::Serializer
  belongs_to :organizer do
    object.organizer.user
  end

  has_many :media
  has_many :races

  attribute :id
  attribute :name
  attribute :description
  attribute :date
  attribute :street
  attribute :city
  attribute :max_participants
  attribute :more_info

  attribute :number_of_participants
end
```

Figure 5.6: Run Serializer



(a) Get model – request (b) Get model – response

Figure 5.7: Ember models architecture [30]

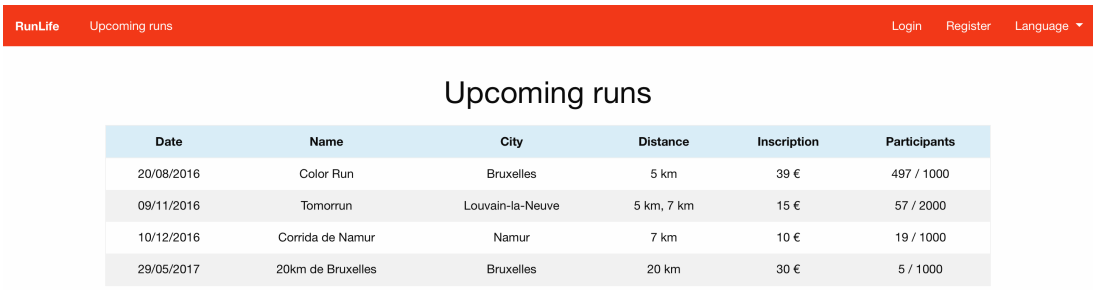


Figure 5.8: Upcoming runs

When a user logs in, he must enter his email and password. The Ember application sends the credentials to the Rails backend. The backend authenticates the user, and sends back an authentication token. The client keeps the session information (authentication token) in the HTML5 Local Storage[1] provided by the browser. This token and the user email will be included in the Authorization header by the Ember application for the subsequent requests. It allows the backend to check the client authenticity without requiring the user to enter his credentials for each request.

5.3.2 Upcoming runs

Figure 5.8 shows the "Upcoming runs" feature. Every user (guest, runner and organizer) can consult the list of the upcoming runs.

Concerning the implementation, the Ember application performs a request in order to retrieve the runs. For the backend to only return the runs happening in the future, the client includes the `from` query-parameter[2]. To allow the backend to cache the response, the `from` parameter corresponds to the date of the current day at midnight. This date must also be communicated using the UTC timezone, otherwise different users might perform different requests on the same day. Thus, the frontend requests the backend through the `GET /api/runs?from=2016-06-01T00%3A00%3A00%2B00%3A00` request.

The backend receives the request, and the RunsController is responsible for providing an appropriate response. The `index` method is called for this purpose. The controller filters the runs whose date is later than the `from` query-parameter. Then, the resulting runs are serialized (using the RunSerializer for each run), and the RunsController responds with the JSON data.

The frontend receives the list of the runs. It sorts them by date, and displays the template (`app/runs/template.js`).

5.3.3 My runs

Figures 5.9 and 5.10 show the "My runs" feature, respectively from the organizer and the runner point of view. The organizer sees the list of the runs he created, with the general data for each run. The runner sees the list of the runs he participates to. The distance highlighted in green corresponds to the race that the runner is participating to. Each run of the list also shows the registration date and the registration status of the runner.

[1] `http://www.w3schools.com/HTML/html5_webstorage.asp`
[2] `https://guides.emberjs.com/v2.5.0/routing/query-params/`

Figure 5.9: My runs (organizer)

Figure 5.10: My runs (runner)

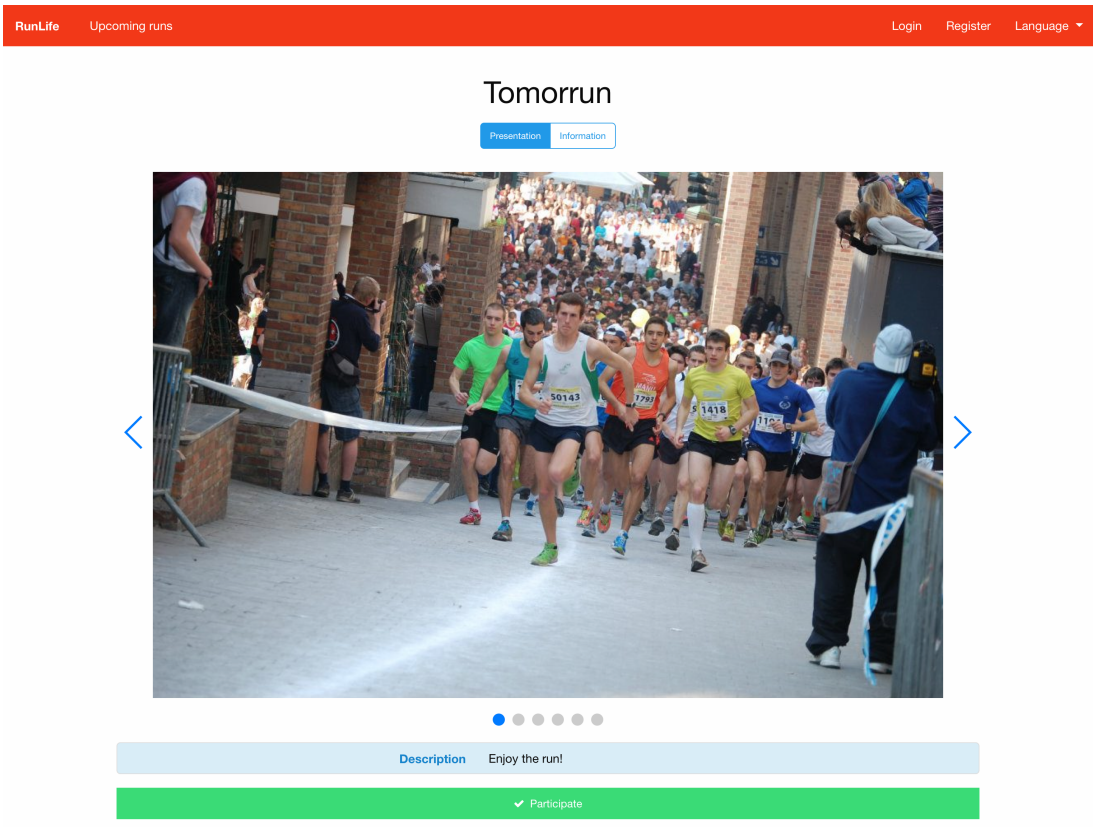


Figure 5.11: Run presentation

To achieve this, the Ember route makes a request to the backend. The request contains the `user` query-parameter (`GET /api/runs?user=:id`). In the backend, the `index` method of RunsController filters the runs linked to the current user. For the organizer, the operation is rather straighforward, as the filtering can be performed on the `organizer` attribute of the runs. For the runners, the filtering is a bit more complex. First, the RunnerParticipations must be retrieved. Then, a mapping is performed between the RunnerParticipations and the Races. Finally, a mapping is performed from the Races to the Runs. Only at that point, the RunsController can return the resulting runs.

The frontend receives the set of runs linked to the current user. The `myruns` route chooses the component to display depending on the type of the current user. For the organizers, the `{{my-runs}}` component manages how the runs should be displayed (located in `app/components/my-runs`). For the runners, the `{{my-participations}}` component displays a list of the participations with the run information. These two components define two computed properties: `futureRuns` and `pastRuns` (`sortedFutureParticipations` and `sortedPastParticipations`, respectively). Each property sorts the runs (and participations) by date.

5.3.4 Run page

Each run has a page to present its different information. Figure 5.11 shows an example of a run presentation. The presentation tab showcases the run thanks to media of different types (pictures and videos), with a general description of the event.

Information

The information tab presents other information: date, address, distance, price, organizer and number of participants. Each information is displayed by a different component. Figure 5.2

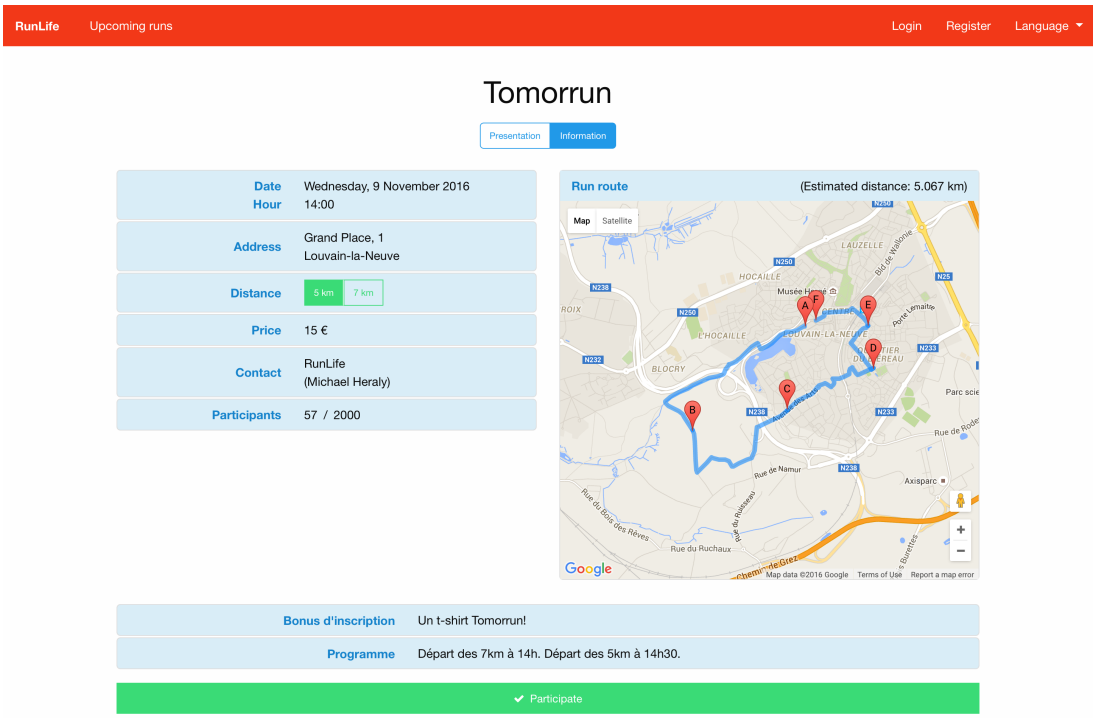


Figure 5.12: Run information

shows that the `date-hour` component is used to display the data.

An organizer can add more information about a run in a flexible way. It allows him to display information as key-value attributes. Figure 5.12 showcase how this feature can be used (e.g. “Bonus d’inscription” and “Programme”).

This functionality is provided by the `more_info` attribute of the Run model. At first, we planned to implement this as a dictionary (JavaScript object). But this was not possible as the keys are intended to be modified. This also causes problems when these information are updated and one key-value field is removed, because if the key has been modified before the changes are saved, there is no way to know the previous value of the key.

To solve this, we implemented this as an array of dictionaries. The format of the array is `[{key: "...", value: "..."}]`. This has a few advantages:

- the key-value information are ordered by default.
- the key can be modified as it corresponds to a text (and not the name of a key).

The `more_info` attribute is stored as a JSON object in the database.

Media management

The media of a run can be managed with the gallery, as shown on figure 5.13. The media can be reordered using the arrows, and a medium can be removed using the cross on the top-right of a thumbnail.

Finding a slider library that works in all cases was pretty challenging. Three different sliders were tried: Foundation-Orbit [33], Slick [34] and Swiper [35]. But every library had some flaw: unpractical to add slides dynamically, unpractical to display a custom slide (that contains more advanced layout than just a picture or a video), bad handling of touch events or bad display on mobile devices. In the end, the Swiper library has been chosen for the slider as it provides interesting features and works well on mobile devices. The integration for Ember applications is done by the Ember-Cli-Swiper addon [36]. It did not provide some functionalities like the

arrows to navigate through the slides. Fortunately, the addon is open-source and we made a pull request[3] to complete the implementation and add the features needed.

Concerning the implementation of the slider in the application, the `{{media-carousel}}` component is responsible for displaying the slider and the different slides. It manages the slider and its different states (showing or edition mode). When the `{{media-carousel}}` is in edition mode, two more slides are available: one slide to add a video or a picture, and a slide to manage the different media. The gallery is displayed by the `{{media-manager}}` component. It manages the actions performed by the user (reordering the media or removing a medium), and updates the medium records accordingly.

When the organizer wants to add a medium for presenting the run, he can access the penultimate slide. This slide is managed by the `{{add-media}}` component (figure 5.19 shows how the slide is displayed). A menu proposes to add a picture or a video. Then, the user can enter the URL of the medium.

In the case of a YouTube video, a regex is applied to the URL to get the identifier of the video (e.g. the identifier of `https://youtu.be/LZxtfY1NKBo` is `LZxtfY1NKBo`). This is done for two reasons: infering the embed URL to use in the `<iframe>` and being able to display the thumbnail of the video in the `{{media-manager}}`. Three computed properties have been defined in the Medium model for this purpose:

youtubeVideoId performs the regex to extract the YouTube identifier.

youtubeEmbedURL corresponds to the URL of the embedded video.

thumbnailURL corresponds to the thumbnail for the video.
(e.g. `http://img.youtube.com/vi/LZxtfY1NKBo/mqdefault.jpg`)

Races management

The organizer can define multiple races for a single run. Figure 5.14 shows the Distances tab where the organizer can manage the races. He is able to add races, remove them, define their distance and price. He can also define the route of the race on map (see Route editor).

The `{{races-manager}}` component is responsible for any change performed to the races (adding/removing a race) and keeping the consistency between the selected race and its information (distance, price and route).

Route editor

The organizer is able to define the route of his races through the Route editor. It is available on the Information and Distances tabs of a run. An example of a route can be seen in figure 5.14.

When the run is being edited, the organizer must first enter the city where the run will happen. Then, the map is shown and he can draw the route of the race(s) on the map. Here are the actions that can be performed on the map:

Add a point by clicking on the map.

Remove a point by double-clicking on a marker.

Move a point by dragging it to the new location.

[3] `https://github.com/Suven/ember-cli-swiper/pull/11`

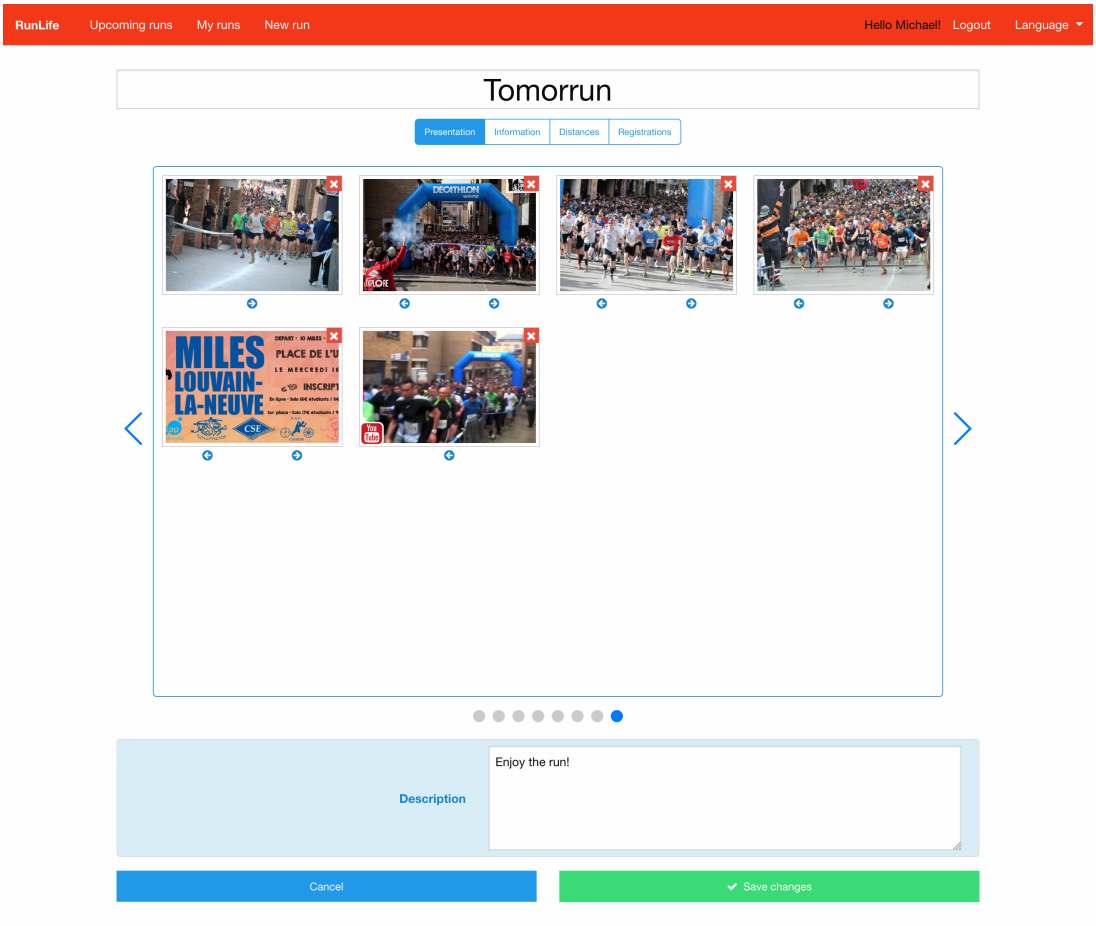


Figure 5.13: Media management

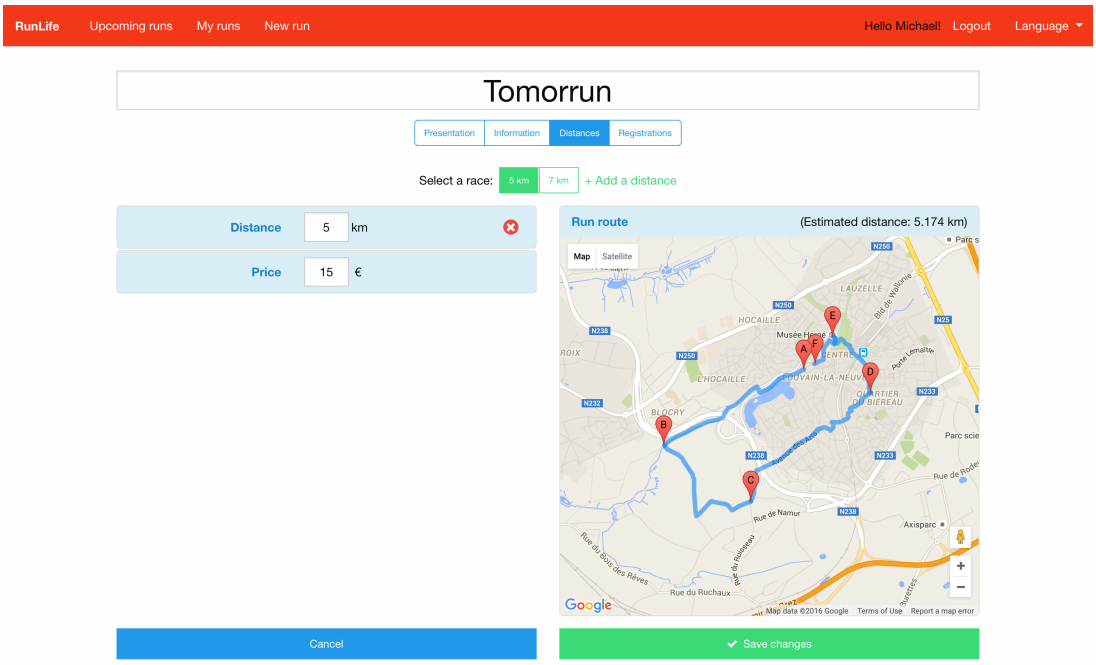


Figure 5.14: Races management

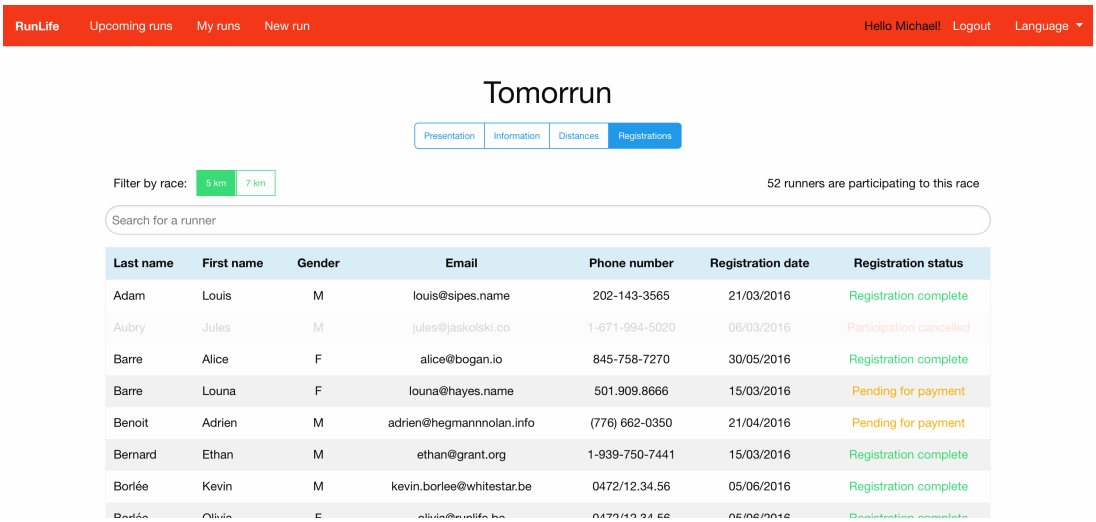


Figure 5.15: Races registrations

Whenever the organizer achieves one of these actions, the route is automatically updated on the map. Each point corresponds to a marker, and they are labelled to indicate their order. The estimated distance of the route is also computed and shown above the map.

The map has been integrated in the application by using the Google Maps JavaScript API [24]. The implementation is composed of two main modules:

route-map-editor component managing the display of the map and its different states: centering the map on the city, adding/removing/moving points, showing/edition mode, update of race selected (must show the corresponding route).

Google-Maps service wrapping requests to the Google Maps API. It is responsible for loading the library and defines methods that facilitate the communication with the Google Maps API. This service is implemented in the `app/services/google-maps.js` file.

Participations management

On the Registrations tab, the organizer is able to see and manage the participations to the different races. He has access to runners information, the date they registered to the race and the registration status.

The organizer can filter the registrations of the different races by clicking on one of the races. He can also filter the participants by name with the search field. The Ember application is managing the filtering so the computation is performed on the client.

When the organizer clicks on a row, a dialog opens and the organizer can update the registration status of the runner. Figure 5.16 shows the options available in the dialog. When a participation has been cancelled, the dialog displays who cancelled it (it can be the organizer or the runner itself) and when it was cancelled. The organizer can add a note for each participation to keep more information if needed.

Run edition

Runs can be modified by their organizer. When an organizer is on the page of one of his running event, he can edit it by clicking on the “Edit run” button at the bottom of the page. The edition allows to change any data about a run (figure 5.17).

The `{{#run-page}}` component displays the run page, manages the show/edit mode, and performs the work needed by “Cancel” and “Save changes” actions. Ember offers useful features

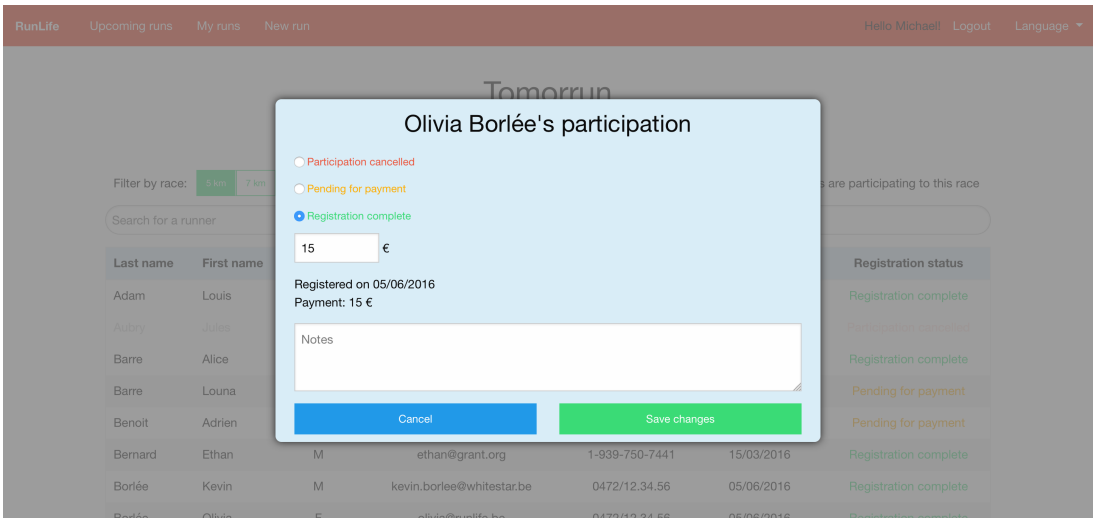


Figure 5.16: Participations management

to manage the cancel and save actions for a model, but the relationships must be handled explicitly. In the application, when the organizer wants to save its changes, the run must be saved but also its relationships (media, races), plus the second-level relationships (i.e. routePoints for the races).

To manage these actions (cancel and save), the `{{#run-page}}` component creates a backup of the run relationships (media, races and routePoints of races). Figure 5.18 illustrates the different cases to handle when the user cancels or save the changes. To simplify the illustration, we depicted a relationship as an array of records. The orange group corresponds to the state of a relationship before editing (it might be media, races or routePoints). The blue group corresponds to the state of a relationship after editing. The "Task" column summarizes the work to perform depending on the user action (cancel or save).

In the case 1), `D` is added to the group. If the changes are cancelled, the group should be set to its previous state `[A B C]`. For the save event, `D` should be saved to the backend. In the case 2), `C` is removed from the group. The cancel task is the same as for a). For the save event, `C` should be removed from the group (and be persisted to the backend). In the last case 3), `A` and `B` have been modified, but no record has been added or removed. If the edition is cancelled, the changes made to `A` and `B` should be reverted (Ember-Data provides a handy method for this[4]). If the changes are saved, the new states of `A` and `B` should be persisted to the backend. Note that any combination of these three cases might happen during the edition.

5.3.5 New run

Organizers can create new runs. The link in the top bar provides an access to the "New run" page (figure 5.19).

The `newRun` route is responsible for creating a run record and setting the controller. As the "New run" page and the run page are very similar, the `RunControllerMixin` has been created to share the common behaviour between the controllers of both routes. The difference between the two routes is that a run page (i.e. `/runs/:id`) provides sub-routes (`/info`, `/races` and `/registrations`) while the "New run" page does offer the tab-functionality, but within a single URL (`/runs/new`). The `{{#run-page}}` component is responsible for displaying a run page (and managing its different states). Depending on the value of the `includeTabLinks` option, the

[4] `http://emberjs.com/api/data/classes/DS.Model.html#method_rollbackAttributes`

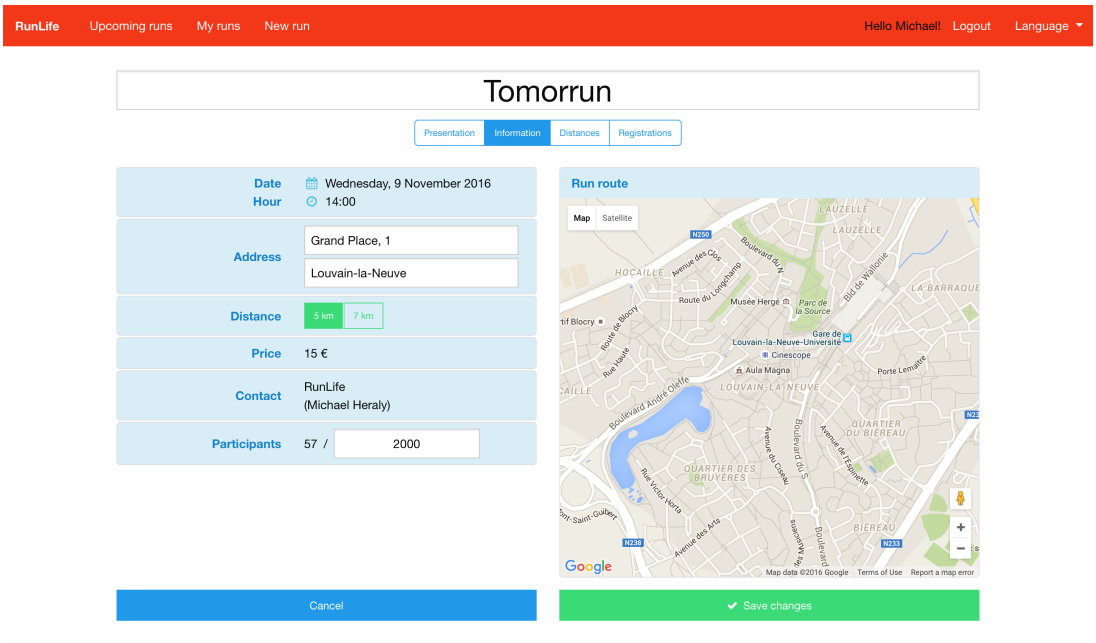


Figure 5.17: Run edition

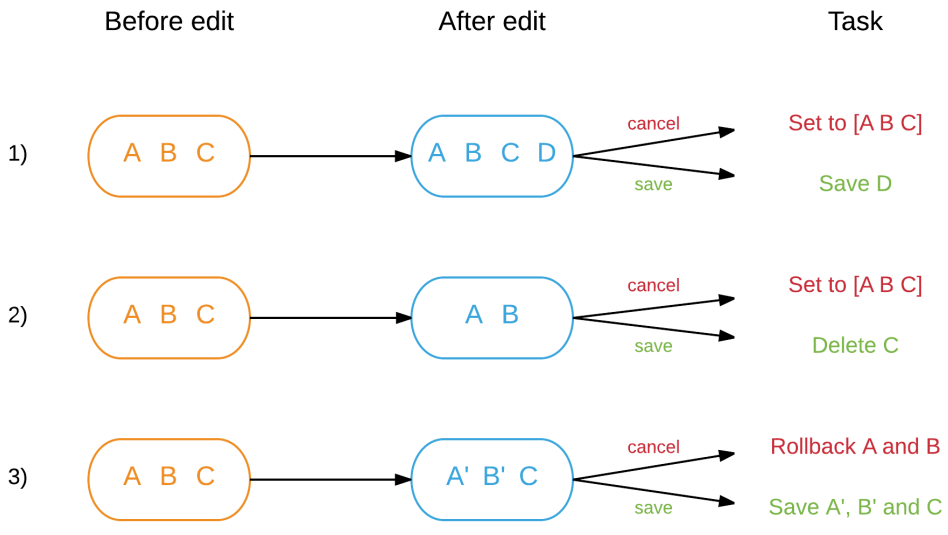


Figure 5.18: Cancel and save diagram

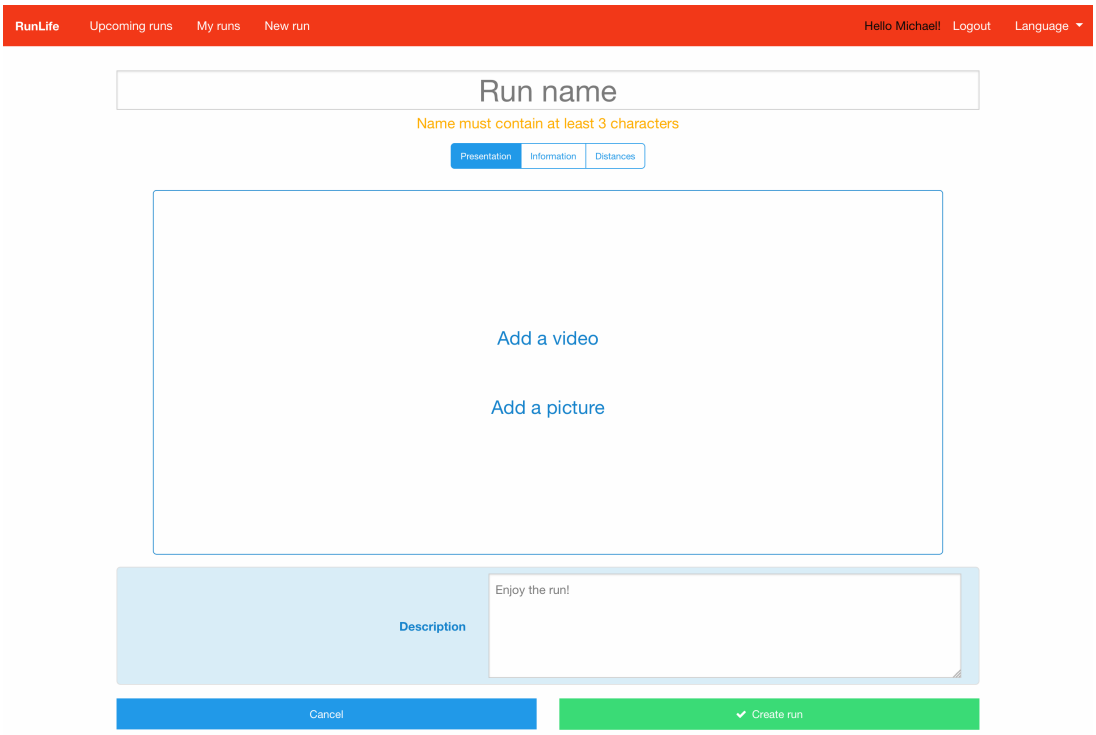


Figure 5.19: New run

`{{#run-page}}` component displays real links to sub-routes or emulates the same behaviour by updating the content.

5.3.6 Change language

Any user of the platform can change the language of the application at any time. The language can be changed through the dropdown-menu in the top bar.

The Ember-Intl [25] addon provides a localization service to support multiple languages in Ember applications. It comes with helpers to format the numbers (currencies), date, time, and messages.

The translation messages stand in the `translations` directory at the root of the Ember project structure. It consists of YAML files containing the messages formatted using the ICU message syntax[5].

Concerning the implementation in itself, if the user is connected (i.e. he is a runner or an organizer), the user locale is saved (to the backend) as soon as the language is updated through the dropdown menu. And when a user logs in, the session locale is updated to the user locale.

[5] `http://userguide.icu-project.org/formatparse/messages`

Chapter 6

Tests and validation

6.1 Tests

Two main types of tests have been performed for the application:

Unit tests These tests consists of checking an individual unit of software (object, function, etc.).

Integration tests These tests are interesting to check the behaviour of the software components.

6.1.1 Unit tests

Unit tests have been written, both for the frontend and the backend. We used unit tests to check model properties and methods. In particular, unit tests were useful to assert the behaviour of the method (in the Medium model) performing a regex on the YouTube video URLs. It allowed to check that the method worked for different links and that the properties depending on the regex (i.e. for the thumbnail URL) were also correct. Unit tests also helped for checking that the values of computed properties (defined on Run and Race models) were correct. For example, a Race property is used to know whether the current user is participating to the race. We wrote tests for asserting that the value of that property was appropriate in the different cases and that it was correctly updated.

Frontend

The frontend contains unit tests in the `tests/unit` directory. They test the computed properties and the validation rules defined on the models.

Backend

The following gems were used to manage the tests on the server:

RSpec testing framework for Ruby applications. It provides tests facilities with a behaviour-driven syntax. The RSpec-Rails gem [37] was used to test the Rails server.

Factory-Girl library simplifying the model creation during the tests. It allows to define model factories. The Factory-Girl-Rails gem [38] provided an integration with Rails.

In particular, unit tests were written for checking the validation rules on the models.

6.1.2 Integration tests

The following Ember addons were used to manage the tests:

QUnit default test framework supported by Ember. The Ember-QUnit [39] addon provides wrappers to facilitate testing of Ember applications.

Ember Data Factory-Guy [40] library that allows to simulate model data during the tests. It allows to define model factories and simplifies the creation of models within the tests.

The model factories are defined in the `tests/factories` directory.

Integration tests have been written for the components of the Ember application. These are located in the `tests/integration/components` directory. These tests consists of rendering the component and performing some actions. Then, the effects of these actions are asserted by checking the DOM[1].

We wrote many integration tests to check the components in the different scenarios. It was especially interesting for asserting that the actions during run edition (i.e. cancel and save) were working as intended.

Testing some features (e.g. sorting/filtering the races registrations and the media management) was particularly useful. It helped to implement the features and perform refactoring.

6.2 Validation

To ensure that the application works as expected, a validation has been performed by simulating different scenarios. As two different types of users are using the application, these scenarios are split in two groups: organizer scenarios and runner scenarios.

6.2.1 Organizer

Creating a run

Once the organizer has set up his account, he is able to create runs. He can create a run through the following steps:

- Access the "New Run" page with the link in the top bar.
- Fill in the run information (name, description, address and maximum participants).
- (Optional) Add media for presenting the run.
- Add one or more races on the Distances tab.
- (Optional) Draw the route of the race(s).
- Create the run by clicking on the "Create run" button at the bottom of the page.

For this scenario, we acted as an organizer and created runs manually. We entered information (media, distances) based on a real runs (e.g. 5 & 10 miles de Louvain-la-Neuve).

After the successful creation of a run, we saw that the run was added to the list of runs. We also refreshed the page in the web browser to make sure that the run information could be retrieved from the backend server and to check that the runs were not only created locally (within the Ember application) but also persisted on the server. The run information were correctly retrieved and the run page was displayed as expected during the creation of the run.

[1] `https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model`

Drawing the route of a race on the map

We wanted to verify that the organizer could define the route of a race on the map.

For this scenario, we edited a run where no route was already defined. First, we changed the city of the run to see if the map would be updated and center onto the new location. The map was successfully updated.

Then, we added markers onto the map. The route was automatically drawn and the route distance was shown. Then, we dragged and removed some markers. The route and the estimated distance were correctly updated. We saved the route, and we could see that when the map is not in edition mode, the markers cannot be modified.

We continued the validation by editing and updating the route we defined previously. We performed some changes and cancelled the edition. The route was back to its state before the edition. Then we made the changes and saved them, the route was correctly updated with the new points composing the route.

Managing the race registrations

For this scenario, we wrote a task that generates participants data to simulate the registration to the races. We used the Faker gem [41] which allows to generate fake data.

To test this scenario, we generated 100 runners and registered them to a single race. We acted as the organizer of the run and we watched the list of the participations. Then, we cancelled some participations. We checked that the number of participants to the race was correctly updated. We also modified the registration status of other runners. The number of participations completed is shown at the bottom of the table and was correctly updated. We can also assert that the notes associated to a participation are effectively persisted.

6.2.2 Runner

Consulting run informations on a mobile phone

The application must be usable on the different devices (mobile phones, tablets, desktops). For this scenario, we chose to focus on mobile phones because it is the smallest screen format and the hardest case to adapt the user interface.

Runners should be able to access run information and to perform the same actions as on desktop. When a user clicks on a running event in the “Upcoming runs” list, he is led to the run page.

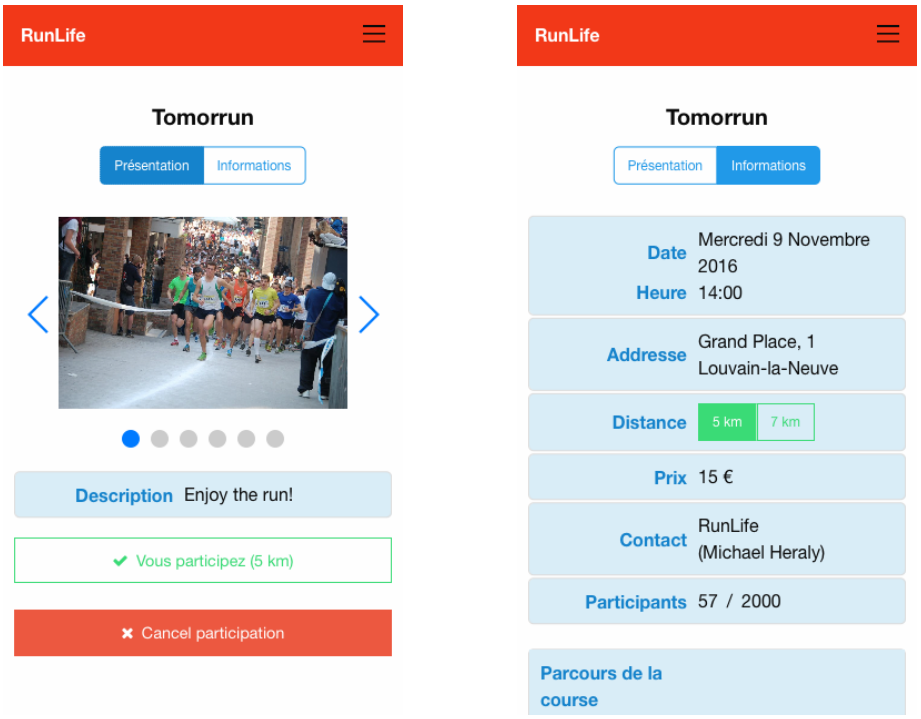
For this scenario, we used an iPhone and accessed the application via WiFi. The different pages (login, list of runs) were functional and accessible with the menu on the top of the screen. As we can see on figure 6.1, the different views of a run page are adapting to the screen format, and the actions (participation state, cancelling participation) are available to users accessing the application with mobile devices. The route of the selected race is also visible on the map at the bottom of the Information tab.

Consulting the registration status

In this scenario, we wanted to verify that the runner could have access to his registration status for the races.

As a runner, we began by participating to a race. The participation was shown on the run page as well as on the “My runs” page. Then, we cancelled the participation and the registration status was correctly updated.

We registered again to the same race. The system creates a new participation with the “Pending for payment” status. After, we logged in as the organizer and modified the status to



(a) Run presentation (b) Run information

Figure 6.1: Run page on mobile phone

"Registration complete". We logged back in as the runner and could see that there were the two participations: the first one that was cancelled, and the new one that is complete.

6.2.3 Results

Performing different scenarios was interesting to check that the application works as expected. It allowed us to see and use the application from the user perspective as an organizer and a runner. It could be interesting to push the validation further to detect edge cases that should be handled appropriately. The application behaviour was checked through different scenarios, and we can assert that the application is usable and functional on the different devices (desktop, tablet and mobile phone).

Chapter 7

Conclusion

The goal of the project was to build an application for helping organizers to manage their running events, from the presentation to the management of the registrations. We also wanted to put an emphasis to the features offered to runners, as the existing solutions did not put a real focus on that part.

The application allows the organizers to present their events on their own, without needing a website. They can showcase the different aspects of their running events, plus the route on the map. Organizers can also manage the registrations within the application, and notes allow them to store additional information in a flexible way. Runners have a quick access to the runs. They can register to a race, and stay informed of their registration status to the different races. The application is usable in different languages and on multiple devices (desktop, tablet and mobile phone).

To achieve this, we began by analyzing the domain and designed an architecture for the application. We developed a single-page application by relying on the client-server architecture. Implementing such a system was interesting, it allowed us to learn and use modern technologies. It was a motivating challenge and it made us appreciate the complexities implied by the solution.

Chapter 8

Future work

Some features were not implemented due to the lack of time. Here are some ideas of future features:

Group registration The organizers could allow runners to register in groups to a race. Groups could benefit from special prices.

Filter runs Users of the platform would be able to filter the running events by different criteria (run type, date, min/max distances).

Custom prices Organizers would be able to define different prices. Each price would be applicable during a date range. This would allow the organizer to provide incentives for runners to register early.

Manual registration The organizer could be able to register a participant manually. This could be useful on the day of the running event.

Online payments This feature would allow runners to pay the registration fee online, through an external payment service.

Glossary

Convention over configuration software design paradigm that aims to decrease the number of decisions that a developer must make without necessarily losing flexibility.

Model-View-Controller software architectural pattern for implementing user interfaces. It defines three interconnected part which communicate together. The Model manages the data and the rules of the application. The View is responsible for displaying the information through the user interface. The Controller manages the connection between the Model and the View.

Monolithic application software application where the user interface and the data access code are combined into a single program.

ORM programming technique for converting data from a database to a type in an object-oriented programming language.

Promise JavaScript object used for asynchronous computations. It represents an operation that is not completed yet but is expected in the future.

Single-page application web application or web site that fits on a single web page with the goal of providing a more fluid user experience similar to desktop applications. The page does not reload at any point in the process, and the resources are dynamically loaded from the server when necessary.

Bibliography

[1] Wikipedia. Single-page application. https://en.wikipedia.org/wiki/Single-page_application, 2016. [Online; accessed 3-June-2016].

[2] Wikipedia. Front and back ends. https://en.wikipedia.org/wiki/Front_and_back_ends, 2016. [Online; accessed 3-June-2016].

[3] Smashing Magazine. Server-Side Rendering With React, Node And Express. https://www.smashingmagazine.com/2016/03/server-side-rendering-react-node-express/, 2016. [Online; accessed 3-June-2016].

[4] Goran Čandrlić (GlobalDots). How website speed affects conversion rates. http://www.globaldots.com/how-website-speed-affects-conversion-rates/. [Online; accessed 3-June-2016].

[5] Fast Company. How one second could cost Amazon $1.6 billion in sales. http://www.fastcompany.com/1825005/how-one-second-could-cost-amazon-16-billion-sales, 2012. [Online; accessed 3-June-2016].

[6] Jake Brutlag (Google). Speed matters. http://googleresearch.blogspot.be/2009/06/speed-matters.html, 2009. [Online; accessed 3-June-2016].

[7] Alexander Zarges (m-way solutions). Client vs serverside rendering – the big battle? http://blog.mwaysolutions.com/2013/11/08/client-vs-serverside-rendering-the-big-battle-2/, 2013. [Online; accessed 3-June-2016].

[8] Ruby On Rails. http://rubyonrails.org/, 2016. [Online; accessed 3-June-2016].

[9] Django. https://www.djangoproject.com/, 2016. [Online; accessed 3-June-2016].

[10] Express. http://expressjs.com/, 2016. [Online; accessed 3-June-2016].

[11] Convention over configuration. https://en.wikipedia.org/wiki/Convention_over_configuration, 2016. [Online; accessed 3-June-2016].

[12] Angular.js framework. http://angularjs.org/, 2016. [Online; accessed 3-June-2016].

[13] Ember.js framework. http://emberjs.com, 2016. [Online; accessed 3-June-2016].

[14] Ember-Data. https://github.com/emberjs/data, 2016. [Online; accessed 3-June-2016].

[15] Ember-cli. http://ember-cli.com/, 2016. [Online; accessed 3-June-2016].

[16] Ember Inspector. https://github.com/emberjs/ember-inspector, 2016. [Online; accessed 3-June-2016].

[17] Mozilla Developer Network. Promise (JavaScript). https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise, 2016. [Online; accessed 3-June-2016].

[18] Bootstrap. http://getbootstrap.com/, 2016. [Online; accessed 3-June-2016].

[19] Foundation for Sites. http://foundation.zurb.com/sites.html, 2016. [Online; accessed 3-June-2016].

[20] Semantic UI. http://semantic-ui.com/, 2016. [Online; accessed 3-June-2016].

[21] Ruby on Rails Guides. http://guides.rubyonrails.org/. [Online; accessed 3-June-2016].

[22] Ember guides. https://guides.emberjs.com/, 2016. [Online; accessed 3-June-2016].

[23] Frank Treacy. Should we use controllers in Ember 2.0? http://emberigniter.com/should-we-use-controllers-ember-2.0/, 2016. [Online; accessed 3-June-2016].

[24] Google. Google Maps JavaScript API. https://developers.google.com/maps/documentation/javascript/, 2016. [Online; accessed 3-June-2016].

[25] Jason Mitchell. Ember-Intl addon. https://github.com/jasonmit/ember-intl, 2016. [Online; accessed 3-June-2016].

[26] The PostgreSQL Global Development Group. PostgreSQL. https://www.postgresql.org/, 2016. [Online; accessed 3-June-2016].

[27] Wikipedia. Representational state transfer (REST). https://en.wikipedia.org/wiki/Representational_state_transfer, 2016. [Online; accessed 3-June-2016].

[28] JSON API. http://jsonapi.org/, 2016. [Online; accessed 3-June-2016].

[29] Rails-API. ActiveModel::Serializer. https://github.com/rails-api/active_model_serializers, 2016. [Online; accessed 3-June-2016].

[30] Ember Models architecture. https://guides.emberjs.com/v2.5.0/models/, 2016. [Online; accessed 3-June-2016].

[31] Plataformatec. Devise. https://github.com/plataformatec/devise, 2016. [Online; accessed 3-June-2016].

[32] Simplabs. Ember Simple Auth. https://github.com/simplabs/ember-simple-auth, 2016. [Online; accessed 3-June-2016].

[33] Zurb. Foundation Orbit. http://foundation.zurb.com/sites/docs/orbit.html, 2016. [Online; accessed 3-June-2016].

[34] Ken Wheeler. Slick. http://kenwheeler.github.io/slick/, 2016. [Online; accessed 3-June-2016].

[35] iDangero.us. Swiper. http://idangero.us/swiper/, 2016. [Online; accessed 3-June-2016].

[36] Ember-Cli-Swiper. https://github.com/Suven/ember-cli-swiper, 2016. [Online; accessed 3-June-2016].

[37] RSpec-Rails gem. https://github.com/rspec/rspec-rails, 2016. [Online; accessed 3-June-2016].

[38] Factory-Girl-Rails gem. https://github.com/thoughtbot/factory_girl_rails, 2016. [Online; accessed 3-June-2016].

[39] Ember-QUnit. https://github.com/rwjblue/ember-qunit, 2016. [Online; accessed 3-June-2016].

[40] Ember-Factory-Guy. https://github.com/danielspaniel/ember-data-factory-guy, 2016. [Online; accessed 3-June-2016].

[41] Faker gem. https://github.com/stympy/faker, 2016. [Online; accessed 3-June-2016].

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve **www.uclouvain.be/epl**