HAAGA-HELIA
University of Applied Sciences

Implementation of the new Swiss-Hema website using headless CMS and React.js

Mathias Tanner

Bachelor's Thesis
Degree Programme in Business Information Technology
2020

Haaga-Helia

Abstract

Author
Mathias Tanner

Degree program
Business Information Technology Bachelor

Thesis title	**Number of report pages and appendix pages**
Implementation of the new Swiss-Hema website using headless CMS and React.js	51+0

The goal of this thesis is to develop a new website for the swiss-hema federation. The old site is outdated and is lacking key functionalities. The final product must meet the requirements of the client and, to be considered functional, have CMS capabilities, be multilingual, and have a registration process with a payment system. Are not included in the scope of this thesis: the content of the website. The text will be copy-pasted from the old site, or, if they are not available, be composed of generic placeholder texts.
The product uses a CMS called Strapi as the backend; the frontend is a React application. The communication between the two is done using GraphQL
The website is fully functional and working. All essential features are implemented. The feedback from the client has been scarce but positive.

Keywords
React.js, Strapi, GraphQL, Javascript, Development

Table of contents

1 Introduction ... 1
1.1 HEMA ... 1
1.2 The Swiss-Hema Federation ... 1
1.3 My Involvement in the Federation ... 1
1.4 Project Oriented Thesis ... 2
1.5 The need for the project ... 2
1.6 My Qualifications ... 2
1.7 Scope of the project ... 3
1.8 Workflow ... 3
1.9 Requirements ... 5
1.9.1 CMS capabilities without IT Knowledge requirement ... 5
1.9.2 Multilingual ... 5
1.9.3 Event registration with payment system ... 5
1.9.4 Event calendar ... 6
1.9.5 Newsletter ... 6
1.9.6 Document and Media Library ... 6
1.9.7 Events Archives ... 6
1.10 Tools used ... 6
1.10.1 Infomaniak ... 7
1.10.2 Google cloud ... 7
1.10.3 GitHub ... 7
1.10.4 React.js ... 8
1.10.5 Redux ... 8
1.10.6 Material UI ... 8
1.10.7 GraphQL ... 9
1.10.8 Headless CMS ... 9
2 Implementation ... 11
2.1 Architecture ... 11
2.2 Header ... 14
2.3 Footer ... 16
2.4 Website Pages ... 18
2.4.1 About us page ... 18
2.4.2 Member page ... 20
2.4.3 International page ... 23
2.4.4 Commission page ... 25

2.4.5 Event page 26
2.5 Event Calendar 31
2.6 Registration system 32
2.6.1 Form Builder 33
2.6.2 Registration System 36
2.6.3 Payment system 37
2.7 Media Sharing space 39
2.8 Event Archives 41
2.9 Publications List 42
3 Conclusion 46
4 References 48

1 Introduction

In the first chapter of this thesis, I'll be going over the general concept of the project. Where it comes from, why it was needed, my involvement with the client, and my qualifications. This part's purpose is to give the reader an idea of what the project is about and why it was required.
I will then describe how I envisioned the workflow with a client that's in another country, the client's requirements in detail, and define their priorities for the final project. This part will describe the objectives of the project and give a measuring tool for the project's success in the conclusion.
In the last part of the introduction, I will describe the tools I used during the project. This way, the reader will have an understanding of the technical terms used in the rest of the project.

1.1 HEMA

Historical European Martial Art (HEMA), is a sport that studies ancient fighting techniques that were used in Europe. It is practiced by analyzing historical treaties and books that were written by the masters of their time. It is similar to Kendo in Japan, with two key differences. In Japan, the tradition and knowledge perdured until modern days, whereas it was lost in Europe until the 1990s. Furthermore, HEMA does not limit itself to sword fighting or one period of time. People study any weapon and any period, providing there are written sources. Since the 1990s, the research has evolved into a sport with competitions and federations.

1.2 The Swiss-Hema Federation

The Swiss Hema Federations represents 21 HEMA clubs that promote the sport nationwide in Switzerland. It was founded in 2012 by ten clubs and has since then grown steadily to serve around 400 practitioners. The Swiss Hema also organizes multiple events, including tournaments, workshops, and formation seminars. The Swiss Hema is also part of the IFHEMA (International Federation for Historical European Martial Arts), an international organization that regroups various national federations.

1.3 My Involvement in the Federation

I have been practicing HEMA since 2007 in Militia Genavae, a club in Geneva, that is a founding member of the federation. I started to get involved at a national level in October 2017 as a translator from German to French. In October 2018, I got elected on the committee of the Federation as Secretary. As of Octobers 2019, I am also head of the media commission in the federation.

1.4 Project Oriented Thesis

A product-oriented Thesis consists of two parts: the first one is a project that is developed by the student, and the second one a thesis report containing the relevant background information as well as the backstory and the theoretical parts of the project. In this project, the report will cover the development process, explication on design and architecture, as well as document the difficulties, roadblocks, and solutions encountered during the development.

1.5 The need for the project

The website used by the federation was made at the beginning of the organization with few resources. Over the years, it has accumulated significant technical debt. It is a WordPress website privately hosted, running on an outdated version of PHP and no SSL certificate. Both these points have become significant issues over the years, creating vulnerabilities and hindering the communication of the federation, due to the website's appearance as "suspicious," on some navigators.
Due to the growth of the federation over the years, there has been an increase in both menus and pages on the current website. Furthermore, some elements repeat themselves or are difficult to access.
The website also lacks some key functionalities that are required, such as an events calendar, and a registration system and payment system for events. Up until now, everything has been managed by time-consuming manual control and excel sheets leading to human error.
The federation also wishes to use this opportunity to add new functionalities, not present on the old version of the website, such as a Newsletter, and a sharing space for document and media.

1.6 My Qualifications

I am currently finishing a bachelor's in business information systems, which has taught me to analyze business processes, identify their requirement, and design a suitable IT solution according to their needs. From my work inside the federation, I have a good overview of their processes and their needs. Therefore I am a suitable candidate to design a solution that will satisfy the federation, taking into account the time frame and the costs involved.
I have also completed several courses and projects with the technologies used within this thesis, noticeably React.js and Material-UI, with which I am both proficient. These elements will reduce the time usually necessary to learn the technologies, allowing the focus on development.

1.7 Scope of the project

My work in this project is to produce a website. Therefore I will focus on the technical aspect and usability. The text and images used are placeholders, and most of them are taken directly from the old website.
The client will need to go over them again once the project is over to correct and improve them. In the scope of this thesis, I work as a developer and not as a writer nor as a translator.

1.8 Workflow

For the project, the federation will act as the client. As they are located in Switzerland, this requires special considerations to maintain implication in the project despite the distance. The idea was not to present them with a final product that might not correspond to their expectations. The solution was to use part of the SCRUM working method. SCRUM puts the client at the center of the development process and search to reduce the time for a product to reach the market through iterative processes(Maria, Rodrigues et al. Oct 25, 2015). In the SCRUM method, developers work by "sprints," an event that is whose duration is predefined, and that will be done consecutively.
In figure 1, there is a schematic of the SCRUM method workflow. It starts with a product backlog, which consists of a list of prioritized to-do tasks that aim at creating and maintaining a product. The person responsible for managing this product is called the Product Owner and is accountable for raise the value of the product by incrementally managing expectations development teams(Scrum Glossary). I kept part of these concepts in my workflow. At the start of the project, I defined with the client a "product backlog," which is a list of features required within the final product according to the clients' priorities. After that, I select one feature and develop it in a "Sprint." I use quotation marks because, unlike the complete method, my "Sprints" are not of fixed length since they involved features of varying size and complexity.

Scrum Flow (Sutherland, Schwaber and Beedle)

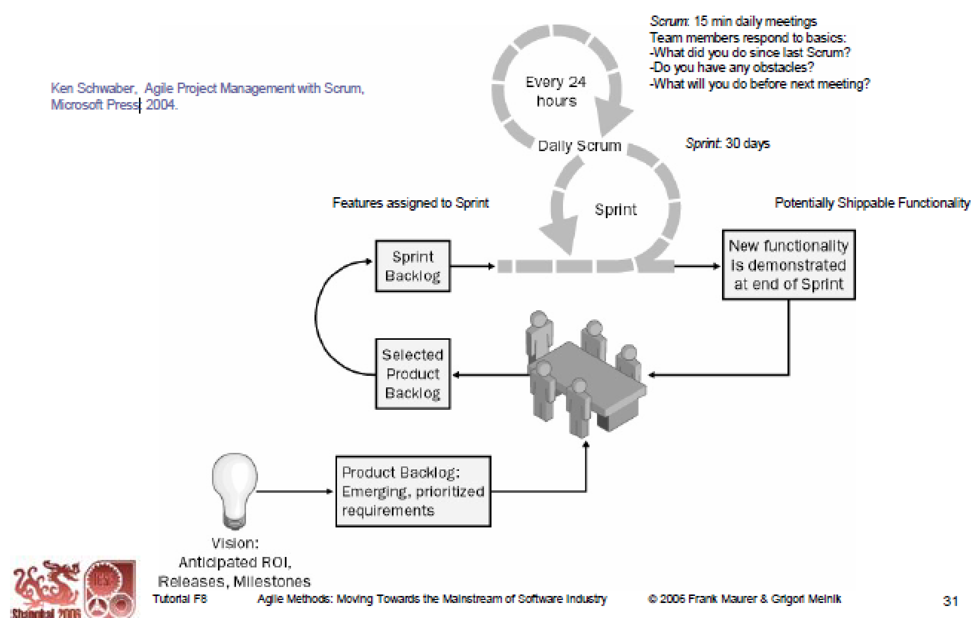

Figure 1: Schematic of the SCRUM workflow(Maurer, Melnik May 28, 2006)

Since I am working alone, I skipped the "daily scrum" meeting, which is a "meeting done daily by the developer team to define the work plans for the day(Scrum Glossary). Once the feature is done, I publish it, and it can be reviewed by the client, which gives me feedback that I can use to create new product backlog items for improvement or design change. The SCRUM method also has the concept of Definition of Done, which consists of the expectations that need to be met for the current work to be fit for production (Scrum Glossary). I have a similar concept; to be able to publish a feature it needs to:

- Fulfill the client's minimal requirements
- Work on any screen size
- Have no errors in the console.

I decided to go with the minimal requirement for the definition of done because it is better to deliver a finished but improvable product in time than a perfect product too late. I keep track of the improvements I can make on the product and store them as a "second priority" product backlog. If I have time to spare after having completed the main features, I'll develop these improvements.
To be able to implement this workflow, the project is deployed in an unfinished state under a non-final domain name, namely "tannerdev.tech." The goal is to have the product available for the client at all times so he can give feedback and information for the rest of the development. The idea is to deploy functionalities as soon as they're done for the client to review. Some may be faster, and some may take longer, it all depends on the difficulty and unforeseen roadblocks encountered

during development. Still, while I'm developing the next one, the client can give me feedback on what is already available, which will ensure the project will not deviate from the client's idea.

1.9 Requirements

The client has defined various elements that need to be met in the new website. These requirements represent the core development steps of the project. After discussion with the federation's committee, I was able to prioritize these features to help define was is a necessity and was is a luxury.

1.9.1 CMS capabilities without IT Knowledge requirement

The website needs to be updatable and modifiable without the need for technical knowledge. The personal of the federation is bound to change and evolve over the year. There is no guarantee there will always be an informatician or somebody that knows code available.
The structure of the website can be a bit more rigid, but the content must be updatable easily.

1.9.2 Multilingual

Switzerland has four national languages: German, French, Italian, and Romansh. Therefore the website needs to be available in multiple languages. There won't be any Italian or Romansch version since they represent respectively 10% and 0.85% of the Swiss population, and we don't have any regular translator available in the federation. Therefore the website must be available in French, German, and English. The client has mentioned that on the home page, the descriptive of the federation should be available in Romansh and Italian with a message explaining why the rest of the website is not available in their language.

1.9.3 Event registration with payment system

The federation organizes multiple events over a year. Those events require registration from the members, and, sometimes, a registration fee. Up to now, the registration used a form that would send an email to the federation containing the information about the registered person. Then this info would be manually transferred on an excel sheet. If the registration required a fee, the member was put on a list until the federation received his payment by bank transfer. He would then be transferred on another list of confirmed registration. All this was done manually.
For the new website, the idea is to have a Form builder on the CMS side of the website, which will allow creating a personalized registration form for each event. On top of that, have a payment system, that provides credit cards to automate this part of the website, and finally, have a registration list for the event to have an easier to manage registration list.

1.9.4 Event calendar

For now, the website only has a list of events and their relative date. The point is to make this list dynamic, sortable, and contain more information on the events. In a second priority, have an event that's exportable to calendar apps.
For the events organized by the federation, have a contact form where they can email us their event information so that we can add it to the list. Since this part is not authenticated, the client doesn't wish to have to possibility to add the event programmatically to the calendar but wants to pass through the federation's email.

1.9.5 Newsletter

For it's day to day communication with its members, the federation requires a Newsletter tool with subscription, un-subscription functionalities, and mailing list. The feature is a high priority; newsletters have been a lacking functionality for quite some time now.

1.9.6 Document and Media Library

On a lesser priority, the federation would like to have a sharing space for documents and media, where members can send their training practice, fighting videos, and so on. This library should be sortable and filtrable.

1.9.7 Events Archives

The client required archives for past events, with the list of participants, winners (in cases of tournaments), pictures, and videos. Just as the document library, this feature is a second priority as not necessary for the functioning of the website.

1.10 Tools used

For the development of this project, I won't do everything from scratch. Just like every current development project, I will be relying on tools and functionalities already available. To define what I will be using or not, I had to select a few criteria. First, the cost, for a tool to be used, it has to be either free or at a minimal cost. The federation, as a sport's association, does not have a steady income, and most of its funds are used to organize events. The availability of documentation related to the tool was also a determining factor. If I encounter a roadblock, I need to be able to find help and solutions online to avoid losing to much time and ensure I don't sacrifice quality in workarounds.

1.10.1 Infomaniak

It was important for the federation to keep the data inside Switzerland. For that reason, I decided to go to Infomaniak, which is Switzerland's leader in terms of IaaS and PaaS. Informanik's standard web and mail hosting offer relies on PHP servers. I, however, was orienting myself in the direction of a Node.js backend. So considering the technologies I chose to use, I had to improvise a bit. As a new product, Infomaniak proposes Jelastic Cloud, an IaaS where unlike the web plus mail hosting platform, you don't pay per month but follow the "pay as you go" principle. The more resources your application needs, the more you pay. Considering that the traffic on the website isn't regular, and spikes at specific times of the year, this solution could be advantageous financially in the long run. The account works with a system of credits that the user can charge via bank transfer. This functionality is practical for a federation that doesn't own a credit card. The downside of this solution is that I need to use multiple services at once. The Jelastic Cloud, the mail hosting, and the domain management are all separate products, instead of having everything grouped in a single package. Also, considering this is a new platform, the documentation available is relatively scarce compared to more well-established services. The server interface is also less user-friendly for a nontechnical user than the all in one offer.

1.10.2 Google cloud

To reduce the cost to a minimum, I decided to split the back and frontend hosting. The frontend is just the appearance of the application, with no personal data stored on it; it also did not need to be inside Switzerland.
The federation already had a Google Account, and therefore there was no new subscription needed to be made, just a service to activate.
For these reasons, I decided to use the Firebase hosting service offered by the Google Cloud services for my frontend React app. The service provides SSD backed hosting across the globe and integrates free SSL certificates for the domain(Fast and secure web hosting). Using Firebase also allowed me to implement a trigger in Google Cloud linked to the GitHub repository. The trigger is activated every time a merge happens on the master branch and then deploys it on Firebase. So, every time I merge on the master branch, the deployed app gets updated.

1.10.3 GitHub

GitHub is a version control hosting company that uses Git. It allows hosting of source code in multiple versions, having a backup, and enables iterative progress(GitHub 2020).
The React app is stored in a GitHub repository to ensure backup, efficient development processes, and version control. The repository has two branches: develop and master. The develop branch is

used as a backup, and the master branch serves as the deployment branch to send a new version of the app for review.

1.10.4 React.js

The frontend app is based on the React library developed by Facebook. This library is one of the most popular JavaScript libraries available on the market. It was designed to help create a user interface in a much more user-friendly way than going through the Document Object Model in JavaScript, which is known to be hard to grasp. It will take any data, no matter the structure(Hunt, O'Shannessy et al. 2016), which is perfect for the headless CMS approach I decided to use. I have previous experience with the framework due to courses and projects, which gives me a good knowledge of the technology. The popularity of the tool is also an advantage since there is plenty of resources online to help and speed up the development process.

1.10.5 Redux

Redux is an open-source state manager for applications; it ensures predictability, allows centralization of the application state, and is easily debuggable(Abramov Dan 2015). In a standard React app, components communicated with each other through props. A parent component will give a prop component, and when the child modifies its state, it'll also change the prop and, therefore, the state in the parent component. The deeper your hierarchy goes, the more complex your communication between components becomes. Redux creates a central store, to which components are subscribed. If a modification happens in that datastore, the new state will be dispatched to all subscribed components. Redux allows for horizontal communication instead of the traditional vertical communication.
To ensure data transmission from component to component inside the react app, I decided to use a central redux store to which all relevant components are connected; this allows for example to do a fast and efficient language change on the entirety of the app. The alternative would be, for example, to have the language state of the app stored at the highest level component and transmit to all its children as props, who would then all transmit it to their children, etc.

1.10.6 Material UI

I have no experience or background in design or user interface. Therefore, to ensure quality and design efficiency thorough the app, I decided to use the Material-UI library for React based on Material Design, which is a design language developed by Google in 2014(Material Design 2020). By using components produced by a corporation as big as Google, I ensure that a lot of UI research

has been conducted on the different elements and that the quality is constant. And by sticking to one library only, I avoid contrast in design between multiple components.

1.10.7 GraphQL

To ensure data communication between my frontend app and my backend app, I decided to rely on GraphQL. It is a query language developed by Facebook and the biggest concurrent to the traditional REST API. I decided to use GraphQL because it allows me to control better the data that's transferred than the REST model (Basics Tutorial - Introduction), and is a fast-growing service, with a big community. Studies also show that GraphQL is more performant than REST in most cases, at least for workload under 3000 requests(Seabra, Nazário et al. Sep 23, 2019). Considering that even during user spike periods, the website project should not encounter that kind of traffic, GraphQL seemed like the better option overall.

1.10.8 Headless CMS

One of the requirements for the project was for the content to be updatable and modifiable without any technical knowledge. The initial approach is to use a Content Manager System akin to what was used in the previous website. The problem with that approach is that the website is limited to what the CMS is thought to do. For a CMS like WordPress, that would be mostly displayed articles and post in a blog approach. For every specific functionality, you need to rely on plugins that are not sure to be compatible with each other and could be dropped by their publisher without any further updates.
Furthermore, those solutions date from a period when the web was mostly consulted by computer screens, before the rise of tablets and smartphones, which makes them suboptimal for responsive web design (Heslop 2018). For those reasons, I decided to orient myself in the direction of a headless CMS, which means an API oriented content manager. The downside is that I must program the entirety of the frontend, but as a positive, I can design it as I want. The client losses a bit in term of flexibility, since he won't be able to customize the structure of the website, only the content (a new page would need to be coded on the front end app before being available). Still, he gains a product that is more specifically tailored to his needs, more stable than an assembly of multiple independent functions.
This choice was made evident since the structure of the website has not drastically changed in the past years. The pages have been mostly the same. The few that were added could, after analysis and discussion with the client, be integrated into pre-existing pages. They were added due to inconsistent and inadequate design, rather than because of necessity. On the other hand, the content of the pages has changed quite a bit over the year, but it'll still be as, if not more, flexible with the headless CSM approach than it was before.

For CMS, I chose to use Strapi.io, a French CMS that is relatively new on the market. The main reason for my choice was that Strapi allows designing content-type (a.k.a tables in the databases) from a user-friendly interface. This feature will enable me to create components that are specifically tailored to the needs of the federation. Those elements will also be easily updatable by the website's admin without any technical knowledge. I also wanted to avoid having to rely on multiple plugins coming from different sources and not made to be compatible with each other. The Strapi team develop its plugins depending on the need of the community; this way, we ensure that they will be up to date as long as Strapi is maintained. The drawback is that there isn't as much choice as it is with WordPress, but this is compensated by Strapi's core structure, which limits the need for plugins in the first place.

If the need arises for a structural change or the addition of a page, the job of the future webmaster will also be more comfortable. It'll be quite intuitive for him to create a new content-type, and he'll just need to code the corresponding element in the React app.

The entire process has been thought for being a compromise between an easier to manage and maintain structure with customs needs and freedom of implementation outside the frame of a closed complete CMS. This way, for day to day operations, the federation can do without any informatician, which is suitable for an organism that has a high executive member turnover. For significant updates or changes, the organization can focus on finding somebody needing technical knowledge of the frontend part of the website only.

2 Implementation

This chapter of the thesis will describe my thought process and how I implemented every element into the website. I will start by describing the architecture, and the permanents element displayed no matter where on the website (footer and header). Those sections will also cover some aspects that span the entirety of the website, like language change.
Then I will describe the different pages used on the website. In some cases, the page acts mostly act as a container for specific elements, and in others, they have their complexity. For this reason, the size of each subchapter might vary quite a bit.
In the last sections, I will discuss functionalities that are not specific to one page or are too complex to explain just in a page section. Those will involve some of the core functionalities like the registrations system.

2.1 Architecture

The website is composed of two different applications that are separately hosted on two subdomains. First, as backend, we have a Strapi app using an SQLite database hosted on the subdomain "admin.tannerdev.tech." Secondly, as frontend, we have a React.js app hosted on the main "tannerdev.tech" domain. For its internal state, the frontend app relies on a Redux store (see section 1.9.5). For the communication between the two apps, I chose to use the GraphQL standard developed by Facebook. By doing this, I had to wrap my React app with Appollo GraphQL client, a GraphQl library for React.js. Added to all this, I use, for the newsletter functionality, Infomaniak's native system, with an embedded form. I decided to use a datastore to avoid constant back and forth between the two apps. Ideally, when a page is loaded, it gets all the required data in one transfer, and then works internally with the redux store to render what's necessary for the component. For example, a page receives the data from all available languages in one batch from the backend. Redux is then used to store the user's language choice, and when the language changes, the app changes the pre-loaded corresponding data. Without Redux, each time a user would change the language, the app would have to re-query the relevant data from the backend, which would not be as fast or as efficient as the actual method. Figure 2 represents a schema of the architecture of the app.

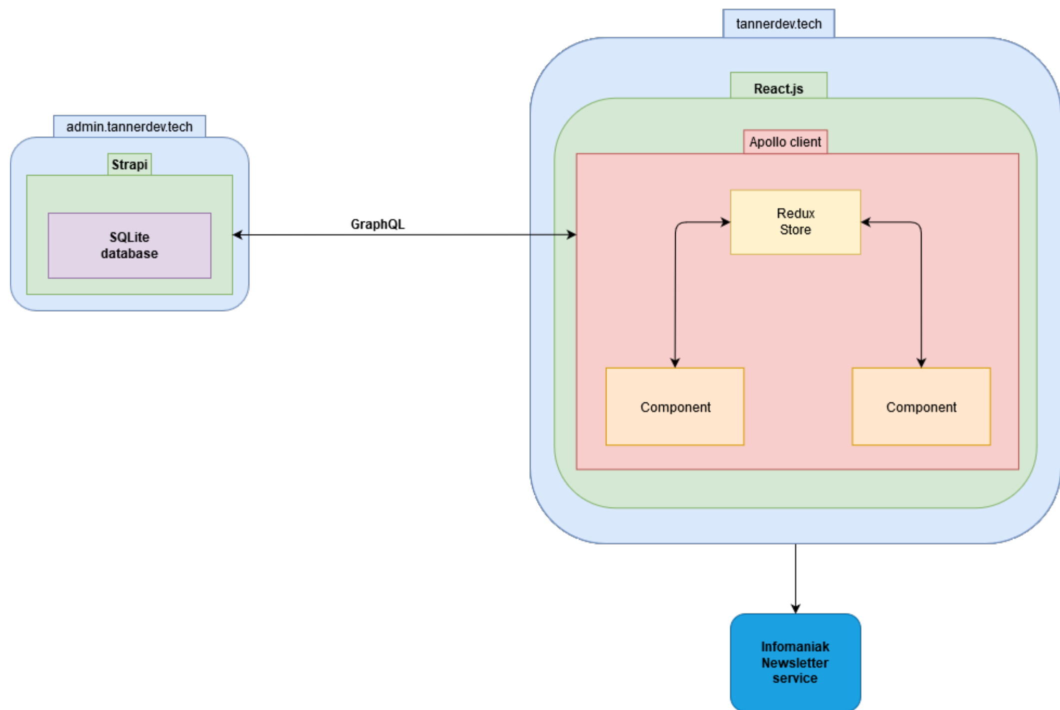

Figure 2: Schema of the project's architecture

I Decided to use GraphQl to communicate between the apps because it allows for high control over the data I receive. With the rest API and embedded objects, you encounter the risk of having memory leaks (an object querying and array of objects, with each element of the collection, also querying an array of objects, etc.).
GraphQL allows you to define precisely what you fetch in your query and cuts that problem altogether. Some elements are useful at a database level but not at a frontend level, such as unique identifiers, for example. By not systematically querying those, I slightly optimize the performance of the app.
As we can see in figures 3, 4, and 5, by using GraphQL, we were able to cut a significant amount of data that may be relevant for the database, but not for the frontend app. This reduction may not bring a considerable difference in performance. Still, if we take a page that has multiple publications, for example, this might lead to less loading time and an overall more enjoyable experience for the end-user.

```
Result of Query: https://admin.tannerdev.tech/publications/1

{
   "id":1,
   "published_at":"2020-03-24T14:00:00.000Z",
   "created_at":"2020-03-24T14:59:14.741Z",
   "updated_at":"2020-03-30T18:56:01.742Z",
   "title":{
      "id":2,
      "FR":"my title en français",
      "DE":"my title in Deutsch",
      "EN":"my title in english"
   },
   "content":{
      "id":3,
      "FR":"contenu en FR",
      "DE":"Inhalt in DE",
      "EN":"Content in EN"
   },
   "image":{
      "id":1,
      "name":"IMG_20191231_114541.jpg",
      "hash":"5466d2a10fe24ca592d18a39974b980a",
      "sha256":"Vg8cxY_l8soSKIMHbS1JkY7z6zGNoXnYwGH44qTGqz4",
      "ext":".jpg",
      "mime":"image/jpeg",
      "size":3844.57,
      "url":"/uploads/5466d2a10fe24ca592d18a39974b980a.jpg",
      "provider":"local",
      "provider_metadata":null,
      "created_at":"2020-03-24T14:59:14.868Z",
      "updated_at":"2020-03-24T14:59:14.868Z"
   }
}
```

Figure 3: Result of API query for a publication using REST and containing all the objects' attribute present in the database

```
1  query Publication{
2    publication(id: 1){
3      title{
4        FR
5        DE
6        EN
7      }
8      content{
9        FR
10       DE
11       EN
12     }
13     image{
14       url
15     }
16   }
17 }
```

Figure 4: GraphQl Query for the same publication as figure 3, but with only the frontend relevant data defined

```
{
  "data": {
    "publication": {
      "title": {
        "FR": "my title en français",
        "DE": "my title in Deutsch",
        "EN": "my title in english"
      },
      "content": {
        "FR": "contenu en FR",
        "DE": "Inhalt in DE",
        "EN": "Content in EN"
      },
      "image": {
        "url": "/uploads/5466d2a10fe24ca592d18a39974b980a.jpg"
      }
    }
  }
}
```

Figure 5: Result of the GraphQL Query. The content has been reduced compared to the result in figure 3

The newsletter element is entirely separate since it's a service that Infomaniak offers freely to its customers. It comes with mailing list management, customizable newsletter builder, and subscribe/unsubscribe functionalities. Therefore, I saw no need to code this on my end, since it was already available and filled all the requirements.

2.2 Header

The header in an app is an element that will be visible on all pages. It usually contains the company's logo, the menu if there is not a specific bar or table dedicated to it and elements that need to be accessible on each page, such as a language selector or search bar.
The client does not require a search bar but needs its website in multiple languages. I also decided to include the menu to avoid crowding the screen with permanent elements.
The logo and the name don't change, no matter what the language is. The subtitle, on the other hand, will vary depending on the language. For this reason, I needed a subcomponent with all the possibilities. The language element will define which languages are available on the website. I need to have every bit of text available in all the possible languages. A Menu can have multiples submenus, and therefore I had to put a recursive link on the object, which means that a menuItem can also have menuItems. One will be defined as the uppermenu, and the other one as the submenu, to be able to establish a clear hierarchy as is seen in figure 6's diagram

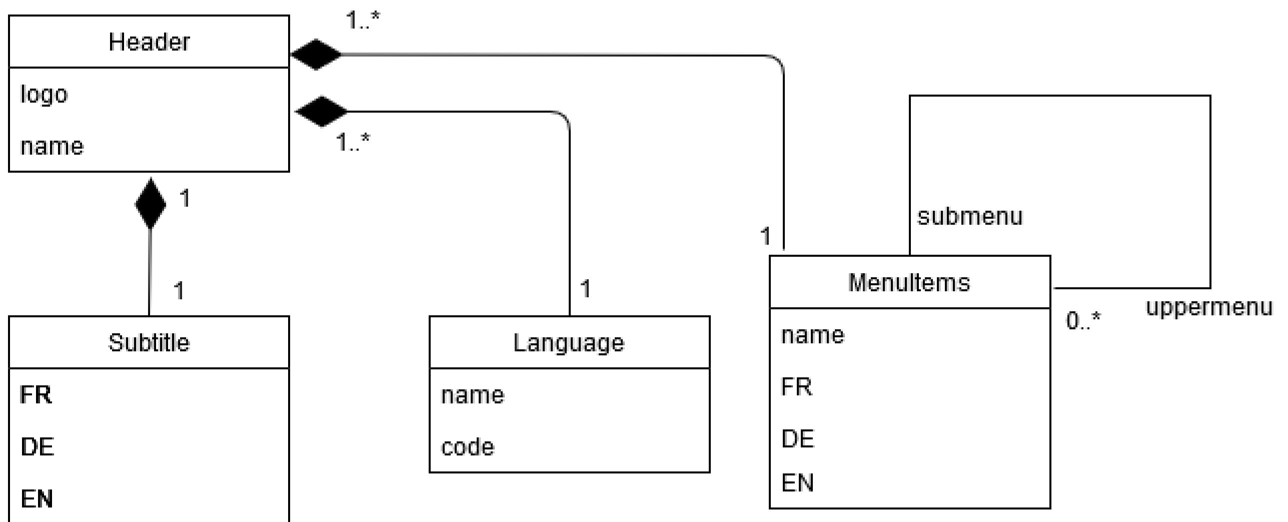

Figure 6: UML diagram of the Header component

The most significant difficulty I encountered was finding an efficient way to adapt the visual elements to the size of the screen. A menu that looks good on a computer could be challenging to use on mobile, and vice versa. For this reason, I decided to have three "versions" of the header. They are not entirely different headers; just some elements are changed and adapted depending on the screen size of the device used by the end-user. For large screens, the menu is simply shown by buttons; if a Menu has a submenu, when you click on it, it'll open it and show the available options.

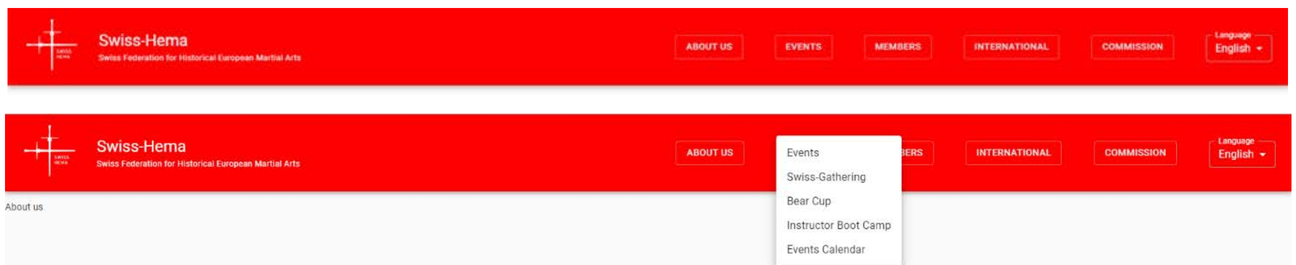

Figure 7: Large screen menu with submenu open

For the smaller screen, I simply used a "Menu" button and just the icon for tiny screens.

Figure 8: small and extra small screen menus

By clicking on the button, you open the menu, which has been changed to a side screen. If there are submenus available, you can unfold them by clicking on the element.
This usage is mainly thought for tablet and mobile users. If I kept the menu as it is in the computer version, it would take all the screen and make the app completely unusable. By using an element that can be hidden, I ensure that the content will still be easy to use on any device.

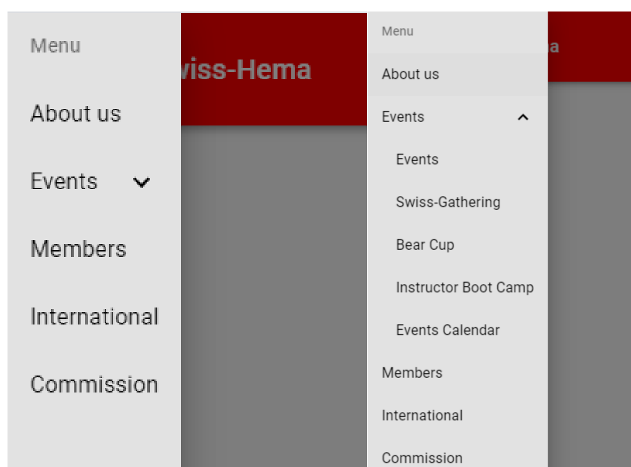

Figure 9: small screen menu with submenus open

Lastly, the language selector at the right end of the menu allows the user always to have it at hand, no matter where on the website he is. By using the Redux store, a soon as the user changes the selected languages in the header, the information is dispatched to all rendered components. When they receive the modified data, it triggers them into re-rendering, thus making the language change instantaneous. The non-rendered components, when they are loaded, fetch the user's language info directly from the Redux Store. Texts in all languages are loaded from the API during the initial render, but only the corresponding are displayed. For international purposes, the client chose to put English as the default language when the website is loaded for the first time

2.3 Footer

The footer is another element of the app. Just like the Header, it is a permanent element displayed no matter what, at the end of the page. In this project, it contains the contact form, the newsletter subscription tool, and the link to social media.
The footer's structure is much simpler than the header since it doesn't require as much complex sub-elements. The newsletter tool is outsourced to Infomaniak, and I just needed to translate and store all the required labels and messages, which was done in two text elements. The same goes for the contact form's label. The content of the contact form is stored in a separate database table, which is queried when a different element is loaded. Since one of the requirements of the client was a form builder, I'll discuss its architecture more in detail in section 2.6.1 below.

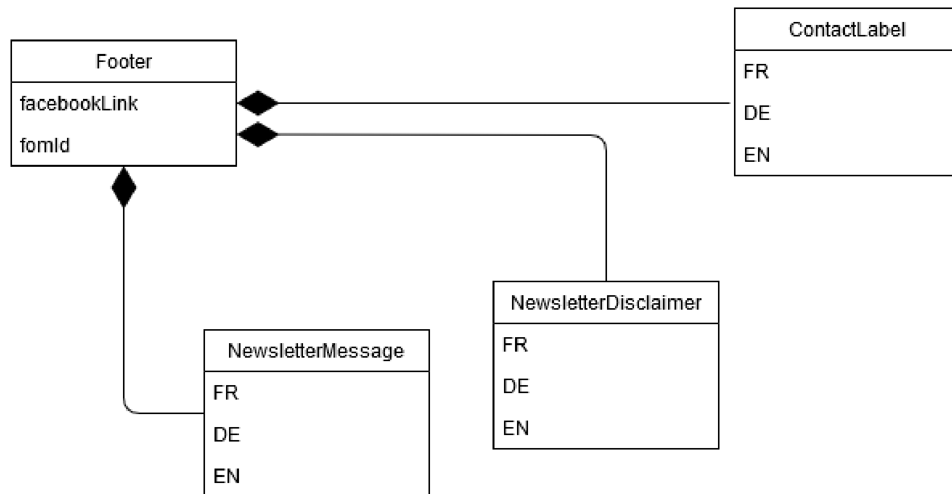

Figure 10: UML diagram of the Footer component

The footer has few elements, so it was not necessary to redesign it for smaller screens. The "Contact us" button opens a modal pop-up containing the form. To avoid having a small unreadable form on mobile and tablets, it grows to take the entire space available on smaller screens. The same principle has been applied to the newsletter form.

Figure 11: Footer element

The contact form just sends an email to a predefined mail address of the federation from the end-users personal email, his subject as mail subject, and his message, which makes it easier to answer requests and questions for the administrators. In the final version, a dedicated email address linked to the definitive domain name will serve as the primary recipient for theses messages.

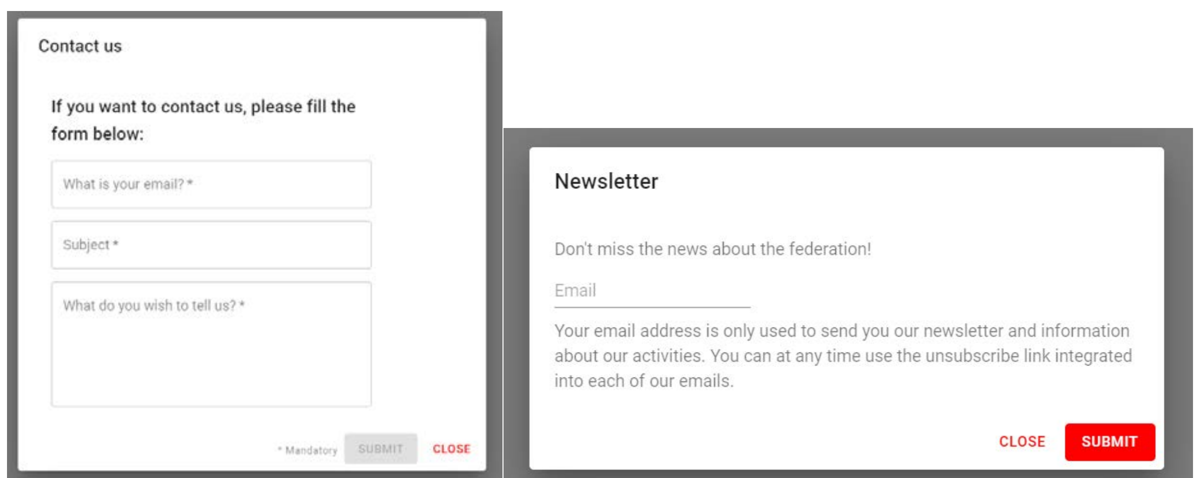

Figure 12: Contact and newsletter forms

The newsletter part of the footer is outsourced entirely on Infomaniak. They manage subscriptions, mailing lists, etc. It will, therefore, not be described in this thesis. The only element I modified is the appearance of the subscription formulary to match the general theme of the website.
The Facebook icon is a plain button redirecting to the Facebook page of the federation.

2.4 Website Pages

This section will cover the different pages of the website. They are more or less the menu items displayed in the header. Each one of them is a single content in Strapi, which means it's a unique object on the CMS.
The size of the respective sections depends on the complexity of the page. Some require a bit of logic, while others are just plain containers.

2.4.1 About us page

The "About Us" component is the landing page of the website. The first element a visitor will see. Therefore, it needs to give an excellent first impression to the user. In terms of the content, it displays who the federation is and what the sport is. It also shows the actual members of the committee, the news, and finally, the partners of the federation. One of the requirements of the client was for this page to be the only page available in all national languages of Switzerland. Therefore, all the labels, titles, and content have five language attributes. The website is technically already completely available in those five languages. The limitation on what languages you can choose to use comes from the language object on Strapi. To add a new language, you'd just have two steps to complete: create a new language object in the CMS, and file the IT and RO elements of all the content and label components (title, content, etc.) It would automatically be translated everywhere on the frontend app.

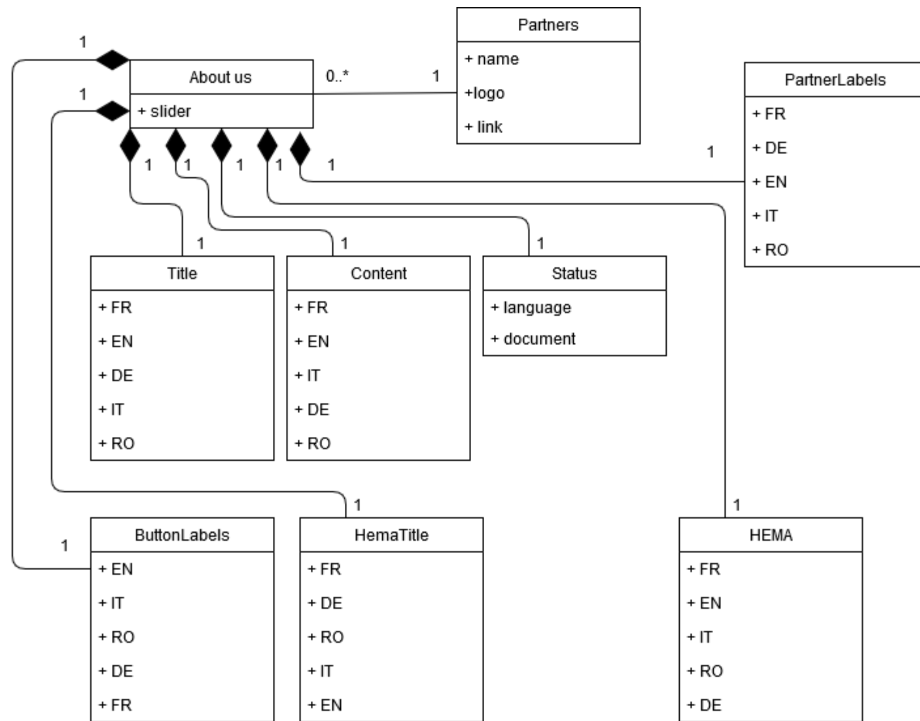

Figure 13: UML diagram of the "About us" component

The about us page is quite simple. First, we have a carousel component used to display pictures related to the sport. The images in use, for now, are just placeholders until I receive some that are more recent and of better quality. Then the five language buttons are displayed, followed by the title of the federation.

Figure 14: Carousel, language buttons and title of the "About Us" page

I then displayed an explicative text about the sport in an expansion panel. The text is quite long and is not destined to the main public of the website, which are people who already practice HEMA and therefore know about it. Consequently, I chose to use an expansion panel to avoid crowding the screen with information that is irrelevant to the most significant target group.

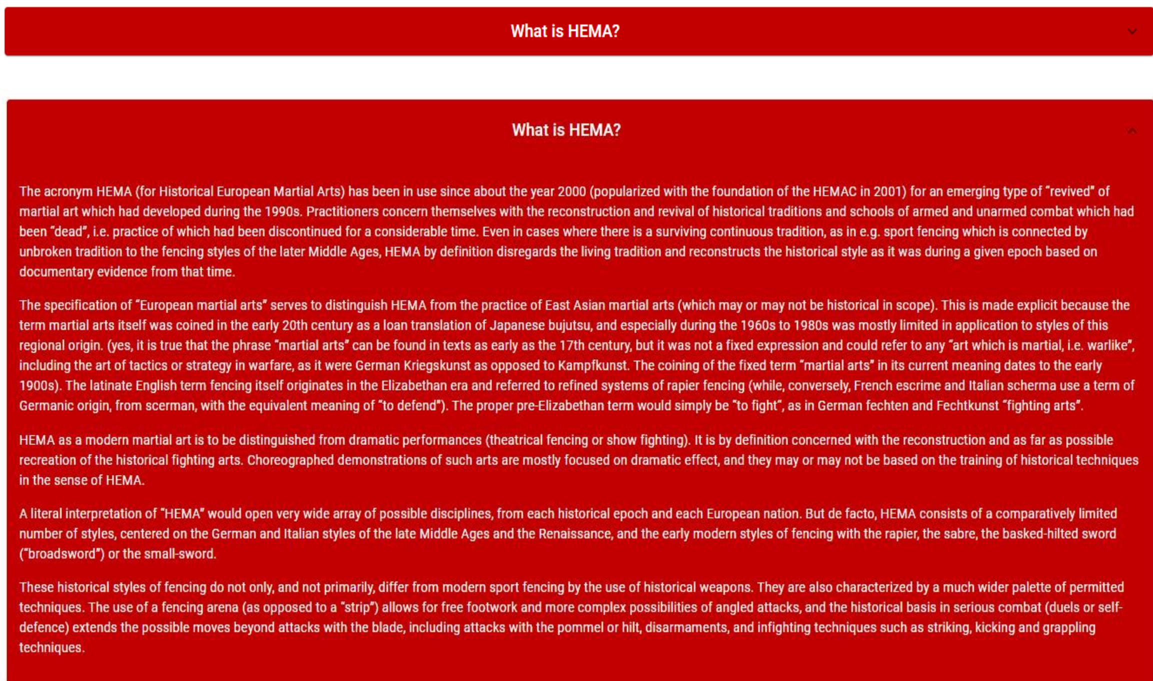

Figure 15: Expansion panel with explanation text

The presentation text and two download buttons for the statutes of the federation are displayed after this element. The statutes only exist in French and German. If they get translated in Italian, the admin can add them to the CMS, and a new button would be added dynamically.

The next element is a publication list. In figure 13, there is no link to any publication object. The reason is that all publications are displayed on this page. There is no need to filter with a category. Also, by displaying all publications by default, we avoid a publication getting forgotten due to human error (an admin creating a publication and forgetting to link it to the page, for example). More information on the publications is given in section 2.9 below.

The last element of the about us page is a grid containing the partners. Each cell includes the logo, the name, and a link to the partner's website.

If the number of partners increases significantly, a carousel element can replace the grid used for now, but that is an unlikely scenario for the foreseeable future, and therefore not a priority.

2.4.2 Member page

The member page contains the list of the member of the federation, with their essential pieces of information. As seen in figure 16, the page is just a container with labels for the other element to be displayed. The federation has two kinds of members: passive and active. To be an active member, a club has to train a certain amount of times during a year., which is not the case for a passive

member. Then there are other groups, which also practice HEMA but are not affiliated to the federation. The federation just displays their name and the link to their website.

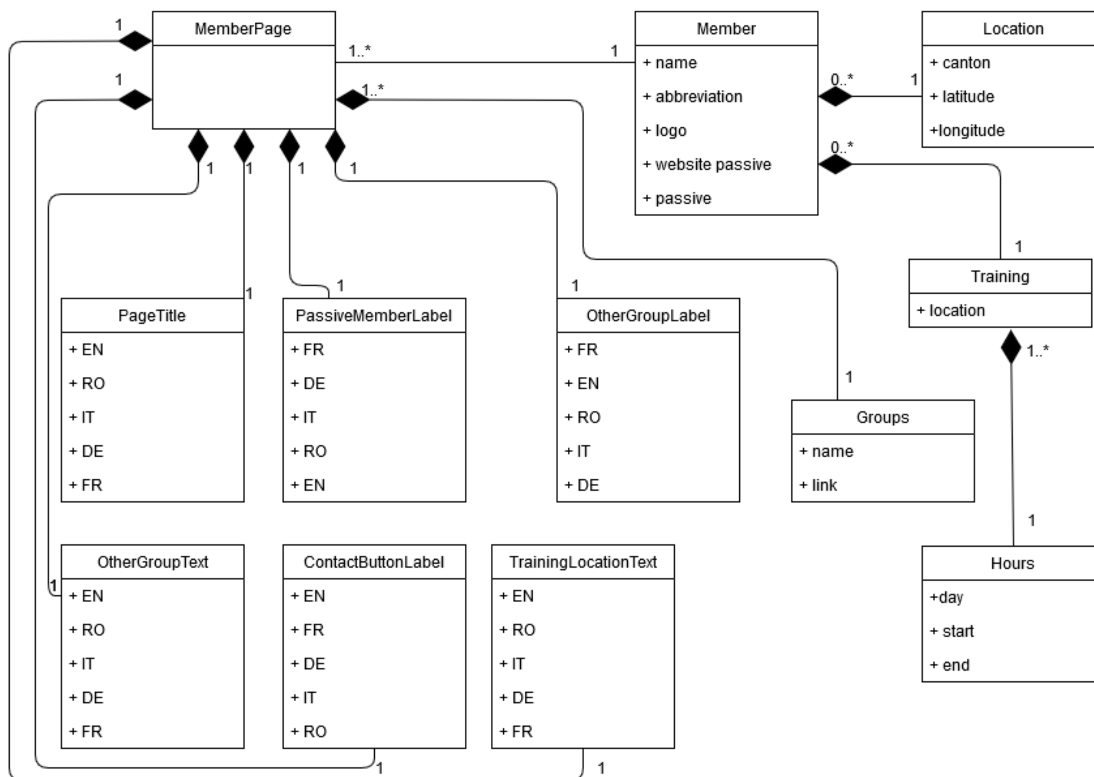

Figure 16: UML diagram of the Members page

The first element to be displayed is a map containing all the training locations for the members of the federation. It is a simple google API call that shows a map with custom markers. On figure 16, the members have locations attributes that contain the canton, which is the administrative region of the club, and the latitude and longitude of their training locations. These pieces of information are used to display a marker on the map. The canton is just there to have the element more readable for the administrator and is not yet used on the frontend side of the app. It can be used in a later update to filter the markers by language speaking area or by canton. It wasn't a priority for the client, though, so hasn't been implemented yet.

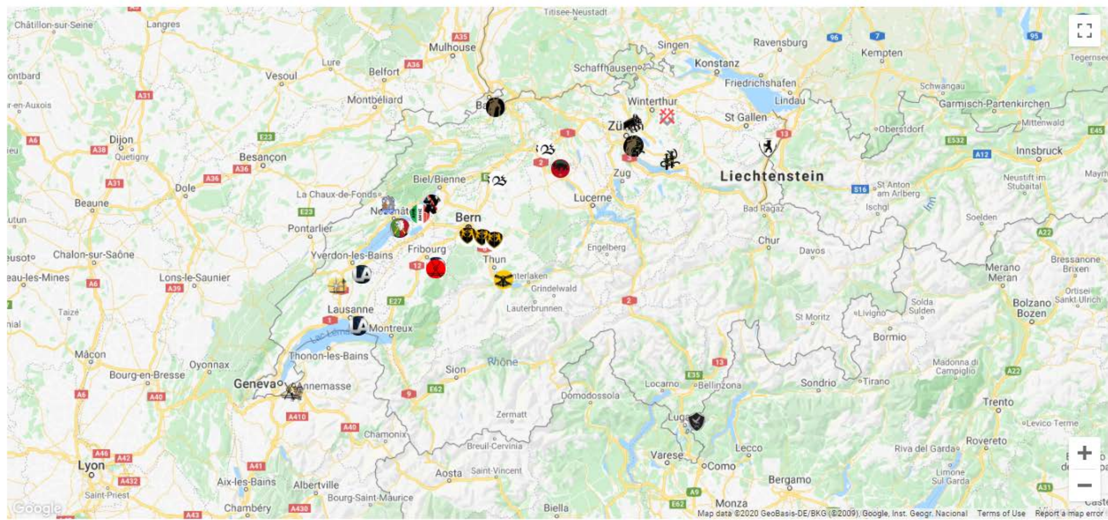

Figure 17: Map with custom markers

Beneath the map, I displayed the member list. The component is a table with the logo, the name (plus an abbreviation if the club has one), and the website of the member. The list is sortable by name, and there is a search function implemented with it.

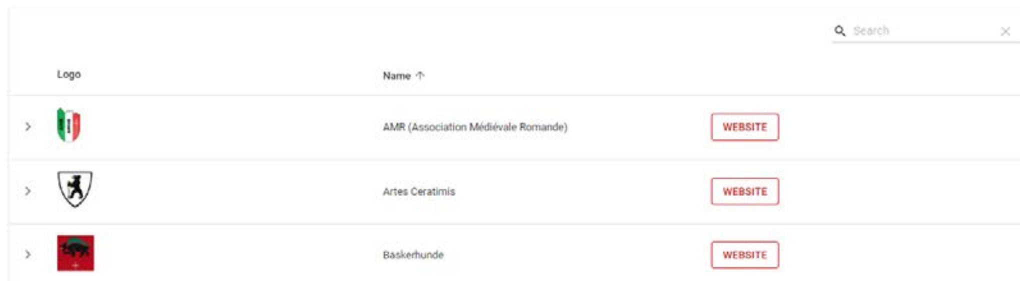

Figure 18: Member list with the search bar

The trainings of the club are displayed in an expansion panel that is displayed when you click on the club's row.

Figure 19: Member row expanded with training information

The training display element is a simple grid containing the location, the day, and start and end hours of the training. On the mobile version, the list works the same. The only difference is that I removed the button about the website of the club. Instead, you can access the club's page by clicking on its logo.

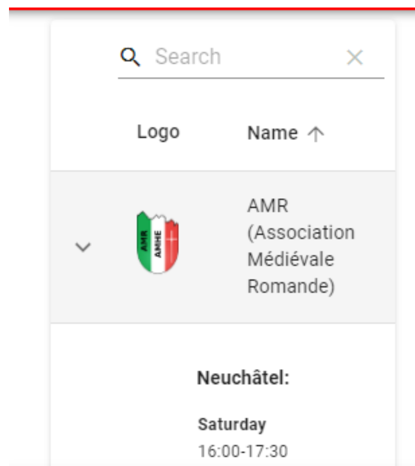

Figure 20: Mobile version of the Member list

After this element, two more lists are displayed, the passive members and the other groups.

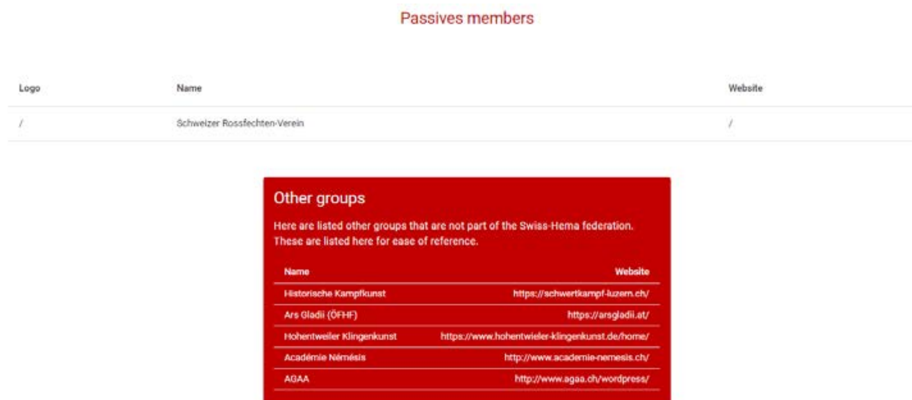

Figure 21: Passive members and other groups lists

Those are quite simple components that display the passive members and the information about the other groups not affiliated with the federation. The first component works just like the member list, without the training expansion panel and the sorting and searching functions. There are not enough passive members and won't be in the foreseeable future to justify those functions.
The other groups' element is just a simple card displaying the name and website of these groups.

2.4.3 International page

The international page displays our ties to other national federations and supranational entities. The page is just showing a succession of lists. Just like the member page, the international page is quite repetitive, even more so since all the lists are based on the same template. The list displays the name, the country, and a button to the website of the organization.

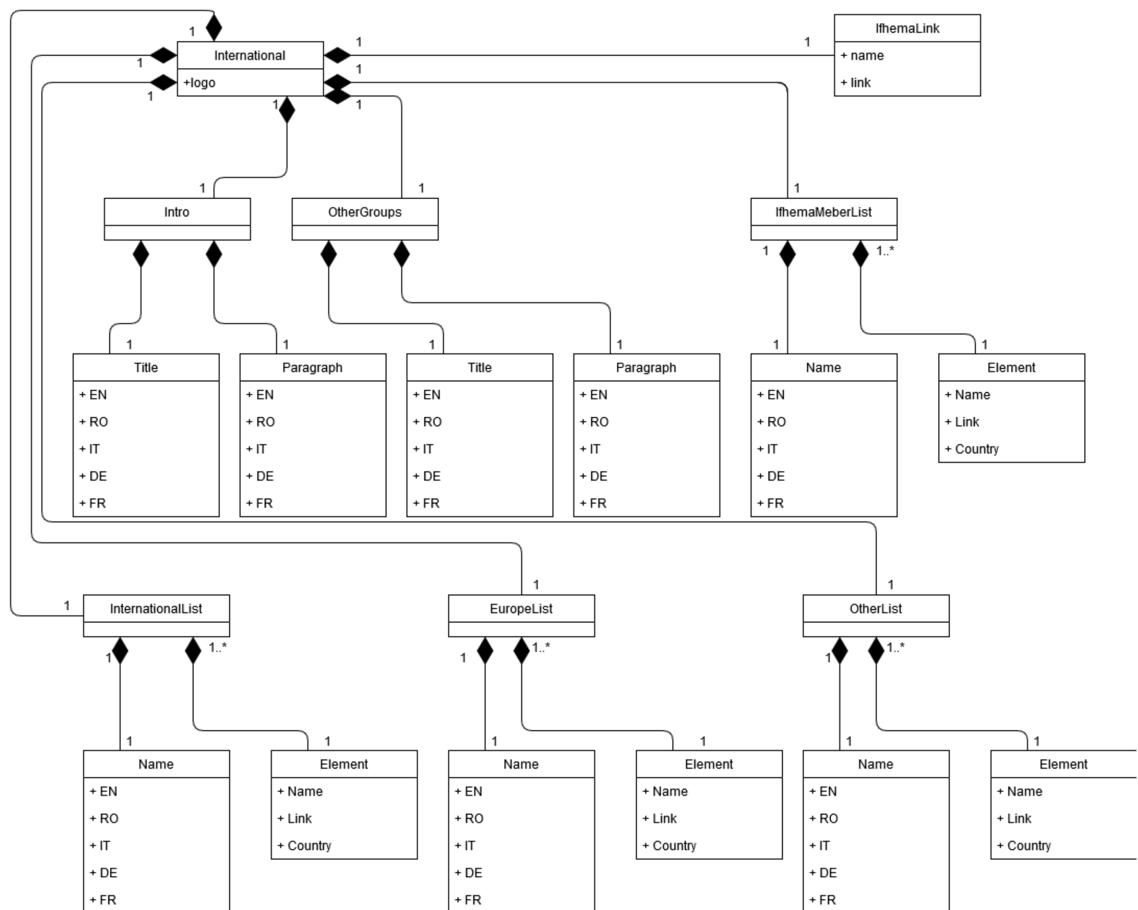

Figure 22: UML diagram for the International page

For the country to be able to be displayed in all languages, I had to create a separate country object in Strapi: It contains the name in all languages and a code. Each list is composed of elements that store the name, the link, and the code of the country. To display each list, I created a FederationList component in React that takes as props one of the lists, iterates over the object, and queries the countries from the backend using the country's code that is stored in the object.

ÖFHF
Austria
WEBSITE

SBSN
Belgium
WEBSITE

Figure 23: Example of list elements

The international list doesn't have any country, so if the result of the backend query is null, simply nothing is displayed. I couldn't link the country directly to the element because of the type of content I used for the element. It is not a table, just a content component. Therefore only its structure

is saved in the database, not it's content; it can not have any relation to another database object. For this reason, I had to make the second query to get the country's information in all languages.

2.4.4 Commission page

The committee of the federation has implemented multiple commissions that help it in the management of the organization. Those commissions create articles or organize events depending on the circumstances.
The commissions' page contains an intro describing what a commission is. The commission object contains all the relevant data for each one of them. Some of the commission publish articles (safety and equipment recommendation, for example). Therefore, there is a repeatable articles component that will be displayed after the commission's intro. The category element is used to display publications specific for this commission—more information on that on the publication list section 2.9 below.

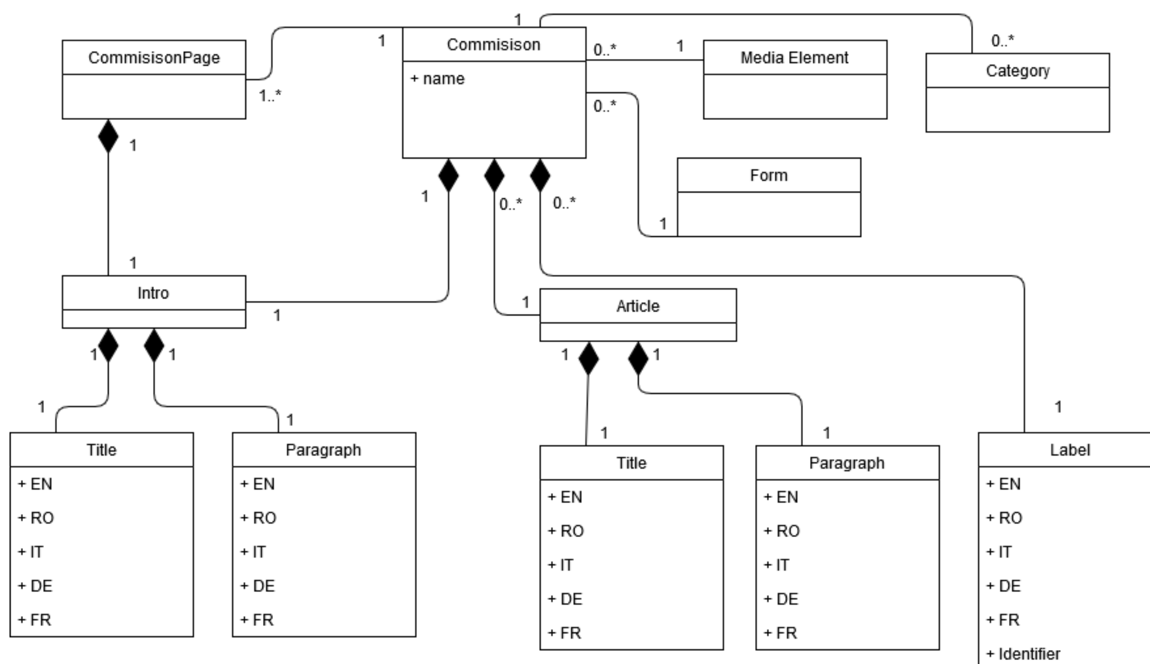

Figure 24: UML diagram of the Commissions page

The commissions are displayed using a tab selector that will query the correct commission depending on the user's choice. On mobile, I switched it to plain buttons, to be able to display them in a column and save space.

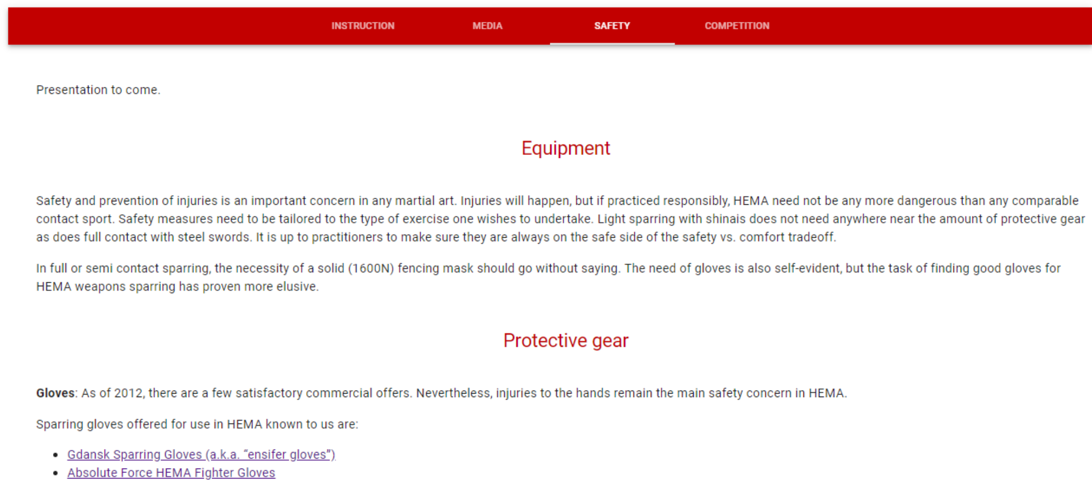

Figure 25: Commission Page with tab selector

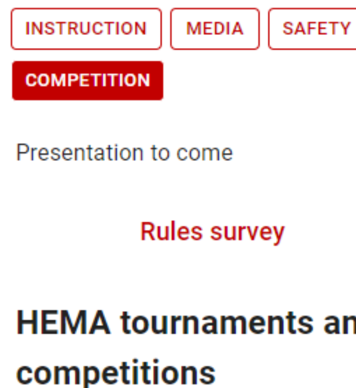

Figure 26: Commission Page with button selectors

The media element and the form linked to it in figure 24 are here for the media sharing zone. The form is to (in the future) implement a contact form instead of the email used in the current version. The media elements are simply the media object displayed in the media sharing zone. More information on this component in section 2.7.

There is also a repeatable label component whose purpose is to display the different labels (contact form, media zone) in all available languages.

2.4.5 Event page

The event page is technically two components in one. We have the page called events, which is just a container to the other events and the calendar, and then there's the component for the specific events information.

First is the "Event" page. The purpose of this page is to serve as a link for the specific events with dedicated pages and the calendar. It's also used to display the publications that are about one or another event. As seen in the figure, the component is straightforward, an introduction element is

there to explain the purpose of the page to the user, and then labels are the element used on the buttons to reach the specific required pages.

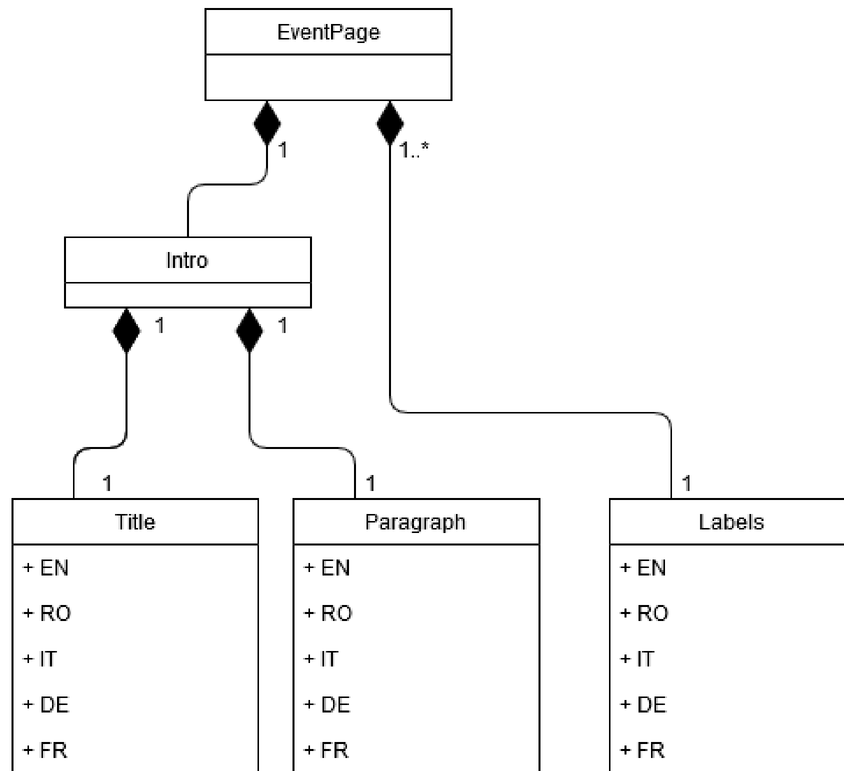

Figure 27: Event Page UML diagram

Visually the page is also elementary. It's just a title, a description, and two sections for the different page buttons. The first is for the calendar page and the second for the specific event pages.

Events

The Federation and it's member organize Events on a regular basis. You can check the events calendar to see what's coming next. The events organized by the federation over the course of a year are:

CALENDAR

Events of the Federation

SWISS GATHERING   BEAR CUP   INSTRUCTOR BOOT CAMP

News

INSTRUCTOR BOOT CAMP

Figure 28: Event Page

The second element of this component is the specific page displaying an evenment's information. First, there are all the essential pieces of information: The name, starting date, ending date, registration start and registration end, the location address, longitude, latitude, presentation image, and a pagename. On the screen, all these information are displayed one after the other only if their value isn't null. An event could be created, but the registration date not yet defined, for example. In this case, no registration date section would appear on the screen. The same is applied to the schedule and the elements of the presentation. The latter is used for when workshops will be held during the event to present the workshop's content and the author. There will be no such section for tournament events, and therefore nothing will be displayed.

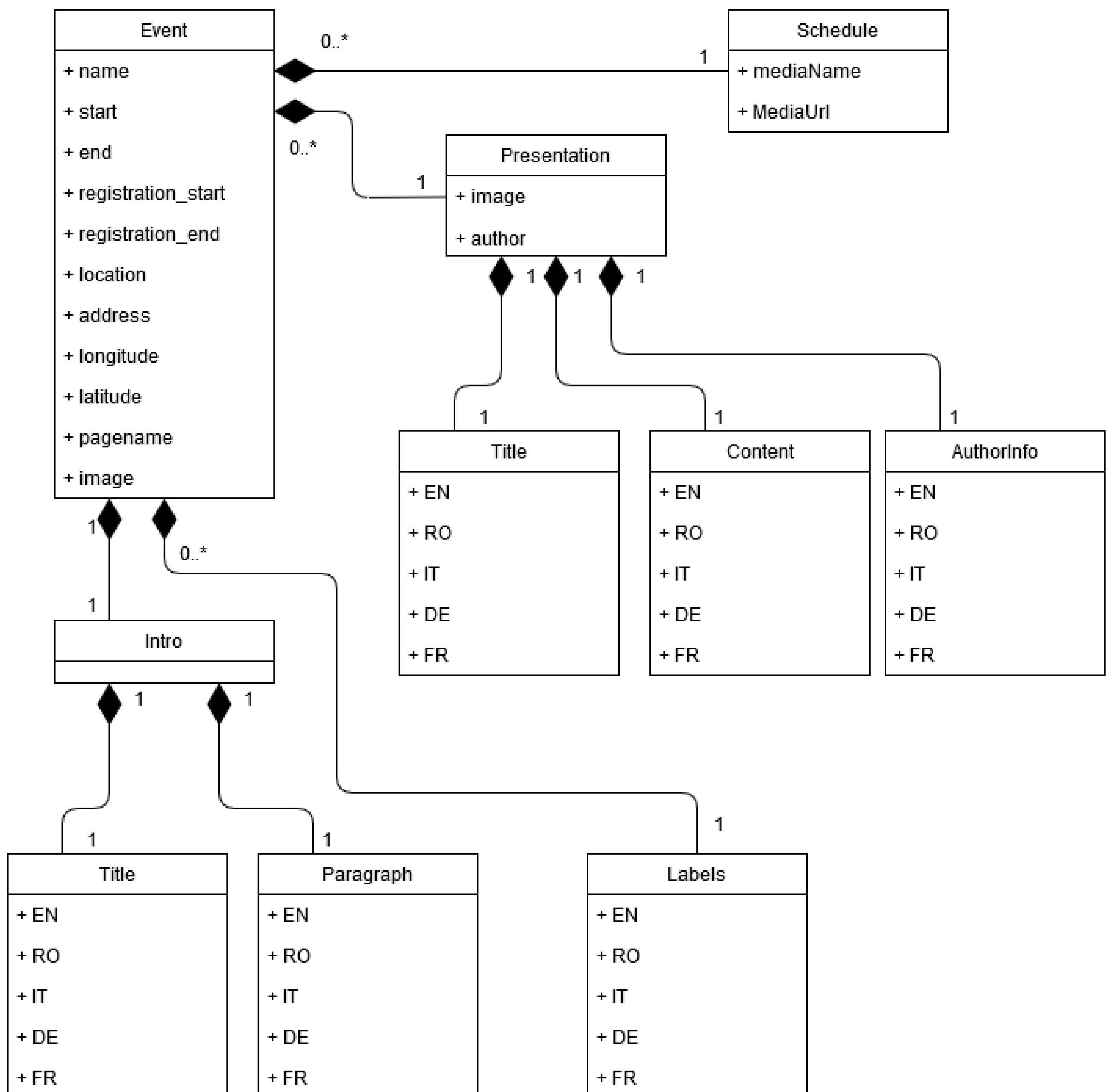

Figure 29: Event UML diagram

Visually if we take an event that has all the possible elements, it starts with the presentation image, the title, and the introduction text.

The Swiss Gathering is the annual gathering of the Swiss HEMA Federation, first held in 2011, hosted by Leo et Ursus in Schönbühl (BE). It is the annual event for historical fencers organised by the SFHEMA. Open to everybody, it offers workshops a tournament and some conferences. It also hosts the SFHEMA general assembly. This event has for aim the meeting of all Swiss associations which are members of the FSAMHE, so that they can discuss about HEMA and share their various approaches and practices.

Figure 30: Event introduction

It is then followed by the dates section on the pages. If the event has a location, the "Add to my calendar" buttons are displayed, as it is a required element for calendar objects: Underneath is the registration segment. If the registration dates are defined, they are posted. If the consultation date is between the start and the end of the registration dates, the fee and the registration button are displayed. If not, nothing appears.

Event Dates

23/10/2020 21:00 - 25/10/2020 17:30

Registrations

16/05/2020 - 01/09/2020

REGISTRATION FEE: 120 CHF

REGISTER

Figure 31: Event dates sections

Beneath that is the location section. This section requires a location, latitude, and longitude to be displayed. The element shows a map with a marker of the event's location and the precise address.

Location

SPORTHALLE MZA, HOLZGASSE, 3322 URTENEN-SCHÖNBÜHL (GPS: WSG84: 47.0236,7.4992; CH1903: 604620 208060)

GYM: HOLZGASSE DORMS: HOLZGASSE PIZZERIA (SATURDAY EVENING): ZENTRUMSPLATZ 8

Figure 32: Event location section

Then there is the event's schedule. An event can have multiple schedules (one per day, for example). The easiest way to display it was to use images. The admin can just create an excel table with all the required data and legends and then screenshot it. Since all events are different, have a different number of columns, etc., it was the easiest way to display a flexible schedule adapted to all situations.

Schedule

Saturday				
Time	Gym 1	Gym 2	Gym 3	Gym 4
08:30	Gym open			
09:30	Opening / warmup			
10:00-12:00	Dussack	Wresteling	Sparring	Bartitsu
12:00-13:00	Lunch			
13:15 - 15:15	Knife	Sabre	Sparring	Dagger
15:30 -17:30	Sabre	Montante	Judges	Dagger
18:00	Gym closes			

Figure 33: Event schedule section

The last section of a page is the presentation of the workshops. First, there is a picture and a small introduction paragraph of the person giving it. Then there is the title and the paragraph describing the content. An event can have no presentations, and then the section is simply not displayed.

Presentations

GAFSCHOLA

I am currently a PhD student at the University of Bern and my research subjects are the military organization of Swiss towns in the Middle Ages and 19th century military fencing in Switzerland. I practice at GAFSchola, an association founded in 2014 in Fribourg and which explores fencing from the Middle Ages to the 19th century. For several years GAFSchola has been exploring Swiss sources in the 19th century in particular.

Kickstarting Swiss Military Sabre Fencing – Preparation for the Assaut by GAFSchola

In 19th century Switzerland, sabre fencing was not only taught to the Swiss officers, but was also practiced as a sport. If military regulations and fencing manuals were used to train young people to defend themselves on the battlefield and physically train their body, the pinnacle of the teaching was the assaut, where students would show their worth as fencers. Like in actual fencing, the assaut was part of the sportive routine, were fencers fought for pleasure and determine, in large fencing contests, who was the best. However, finding rules related to the practice of the assaut in 19th century fencing clubs and events is difficult and military manuals usually only outline some aspects of this practice; solely one set of rules can be found in a sabre fencing manual, Emil Probstt's Instruction sur l'escrime au sabre (Anleitung zum Säbelfechten) from 1887, which describes the concours d'escrime, linked to the federal military celebrations (fêtes fédérales) and established to judge of the young officers' martial capacity. This workshop will thus focus on this rule-set and on other traces in military manuals to explore the practice of the assaut in Swiss military fencing; the attendants will first train basic movements to understand the core of the system, then will experiment one or several rule-sets. The workshop's final purpose is to offer some fundamental tools for practicing Swiss military sabre fencing as a sport.

Figure 34: Event presentation section

2.5 Event Calendar

On top of the events organized by the federation, it's members also host their fair share of activities. To be able to display all the general information about them, the website needed a calendar component.

The calendar page is composed of an introduction and some labels; it is mostly a container for the other elements. The page is in relation to multiple calendar events. They contain all the data related to one event. The selection for those attribute was defined by what was required to create a calendar event item for most of the calendar application providers (Google, Ical, Yahoo calendar, etc.)

On top of this, there is a link to the event's organizer's website. The paragraph element shown on the calendar event in figure 32 is just there to improve readability on the backend of the project. The attribute is not used on the frontend. The calendar page is also linked to a form that is used by the members to send new events to the federation. The client did not want the members to be able to put an event by themselves on the calendar list. Since there is no authentication, they wish to validate each entry. For these reasons, a form is a contact form that sends an email to the federation.

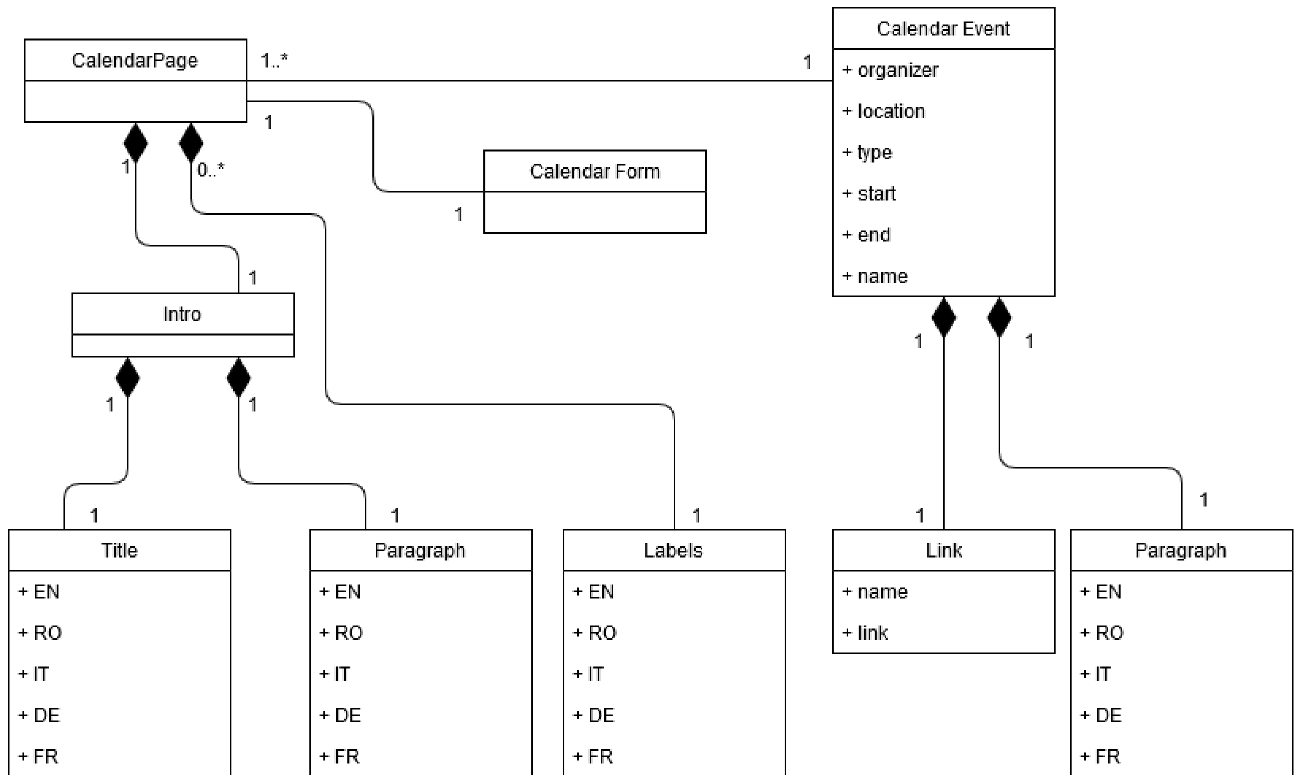

Figure 35: Calendar UML diagram

Visually the calendar is a list of events sorted by date. The columns are the date, the name, which also acts as a link to the organizer's website, the general location, the type of event, and a button to add it to any calendar app.

Event calendar

List of events ogranized by the federation or it's members during 2020

Members: share us your event by using the form below

Search

Date	Name	Location	Type	
06.06.2020-07.06.2020	Swiss Hema Challenge	Lausanne	Tournament	Add to My Calendar
03.07.2020	test workshop	Bern	Workshop	Add to My Calendar
04.07.2020	test sparring	Thun	Sparring	Add to My Calendar

Figure 36: Calendar page

The form is displayed in a standard expansion panel. Forms will be discussed more in detail in section 2.6.1, and I will therefore not explain it further here.

2.6 Registration system

The federation organizes multiple events over a year. Each of them requires a different registration form, and if they create new events in the future, they need to be able to create a new form without

going through somebody with technical knowledge. For this reason, the first requirement for the registration system was a form generator. The second requirement was a payment system since most of the federation's events require a registration fee.
Separately those requirements are not very difficult to implement, but altogether it proved to be a challenge. The following sections will describe the different components of the registration system and their interaction with each other.

2.6.1 Form Builder

The client requested to be able to create custom forms to be linked to the registration system. There is two main challenge to this request. First, being able to dynamically render a complex questionary without knowing in advance what kind of questions are going to be received by the frontend part of the form. Secondly, being able to send those questions back to a registration object that will not know in advance which items will be used for that specific registration.
The first challenge was the easier one to tackle. The way I solved it was to create a form object which will be linked to multiple questions. We can then link that form to their respective event. The administrator can then create new items for the form, add and remove them at leisure.
Figure 37 shows that the form object itself is quite simple. There is a name, a type, and a description. The type is used to separate the contact forms, which will just send an email from the registration forms that will create a registration object in the database.

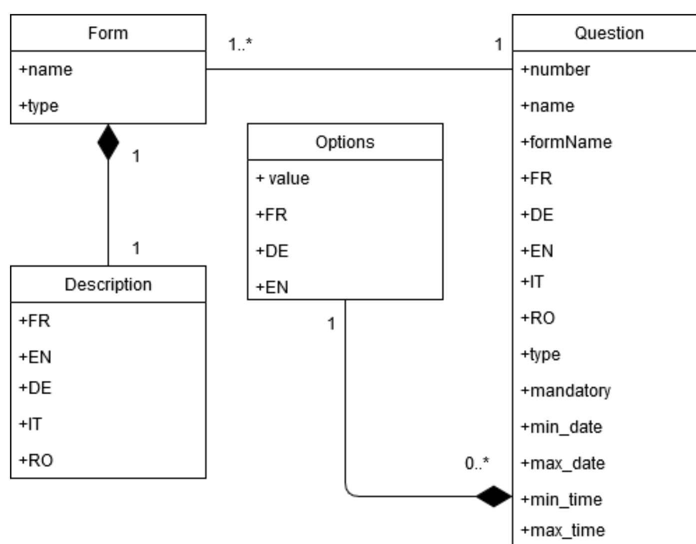

Figure 37: UML diagram for the Form and Question objects

For the contact forms, the main difficulty I encountered was to send the email containing all the information. I must use a system that keeps the markdown of the message. Strapi's default mail plugin does not provide this service, it just sends the string object, and cors policies block any message containing HTML.

I set up the Nodemailer provider for the plugin and used it in conjunction with the Infomaniak mail service. I lost quite some time because I did my first tests using the Gmail account of the federation in the configuration, and the sender (which was supposed to be the end user's email) was always replaced by the Gmail address. For a contact form, it was quite a big problem since the federation would not receive the mailing information from the person that sought to contact them. I thought that It was a configuration error with the plugin, but after some research discovered that Gmail replaces the sender by the account address by default to avoid identity theft. That's why I had to switch to the Infomaniak service which then worked perfectly
The question object is a bit more complicated and contains a larger quantity of attributes.
The number is used to define the order of the questions. Since the questionnaire is dynamically stored, we can not determine in which order they'll appear in the query result. The number is, therefore, a sorting attribute.

```
questions(sort:"number"){
  name
  number
```

Figure 38: GraphQL query with the number as sorting attribute

The name and the formName are just user-friendly elements to make the questions objects more readable on the CMS part of the app. Without the first, the only identifier available without opening the detail of the question would be the id, which is not useful for an admin. Each question is linked to one form, but without the formName element, you'd have to open the detail of the question to see which is displayed in the question list on the cms. The attribute makes the information available at first glance. It has a drawback, though, since it introduces the possibility of human error. The admin could enter the wrong name. The attribute is not used programmatically, though, and therefore this error would not render any bug on the website.
The two most important parts of the questions are the type and the "mandatory" element.
The first is a selector with different options that the administrator can choose from which will define what kind of question it is (text, long_text, boolean, etc.)
The type attribute of the question is essential for the frontend part of the app. In the form component, there is a switch that will loop on it, and render a different input with different validation process depending on different types. For example, the type "email" will render a small text input linked to a regular expression that ensures it similar to: [something]@[something].[something]. The type "select" will generate a select element with a series of options to choose from.
The mandatory is also a critical part of the question. It defines if the question must be answered or not. If not, you can submit the form without it. When a form is generated, the first thing that the component does is to add an answer element to each question.

The second thing that is done is to generate a validator that checks if the input is correct and to look if the question has to be completed or not for the form to be valid. The state of the component ties the type and mandatory attributes together as the form cannot be submitted if, for each question, the "complete" attribute of the state is "true," and the "wrongInput" attribute is "false."

```
useEffect(() => {
  if (!props.form.questions[0].hasOwnProperty("answer")) {
    props.form.questions.map((q) => (q["answer"] = ""));
  }
  if (dynamicValidator === null) {
    let generated = {};
    for (const q of props.form.questions) {
      switch (q.type) {
        case "email":
          generated = {
            ...generated,
            [q.name]: { complete: !q.mandatory, wrongInput: true },
          };
          break;

        default:
          generated = {
            ...generated,
            [q.name]: { complete: !q.mandatory, wrongInput: false },
          };
          break;
      }
    }
    setDynamicValidator(generated);
  }
}, [props.form.questions, dynamicValidator]);
```

Figure 39: React Hook that adds the answer attribute to the question and generates the validator for the form

The last elements of the question table are just options attributes that are not mandatory.
If the question type is either date, time, or dateTime, they will serve to define the boundaries with a minimum and a maximum option. There is a risk that the administrator will enter a date question and forget to enter the limits. Sadly, Strapi does not allow conditionally mandatory attributes. They are either required or not. To avoid having errors, on the frontend part, I'll just set the limitation as non-existent if the date/time options are null.
In terms of ergonomic, there are some flaws in my design that I could not workaround.
For example, a question cannot be reused over multiple forms. An email question will have to be recreated for each form, even if the text of the question is the same. The reason for that is mainly sorting and readability. I must have an element on which to order the questions by form to avoid having illogical results. This flaw is shadowed by the fact that there will not be an excessive number of questionnaires so that this redundancy will not have a significant impact on memory consumption.

2.6.2 Registration System

The registration system works closely with the form builder. The challenge was to create a registration where the data comes from a dynamically generated form. In short, the registration object does not know what it will receive as data. Since Strapi restricts the creation of content-type in production mode, I couldn't just recommend the client to create a new kind of registration object each time he creates a new form. The solution was to have some fixed elements common to all registrations. The name, the first name, the club of origin, the email, and the name of the event to which the registration is linked are used in all registrations. For the rest of the form's questions, I have a repeatable component that stores the question's name and the user's answer.

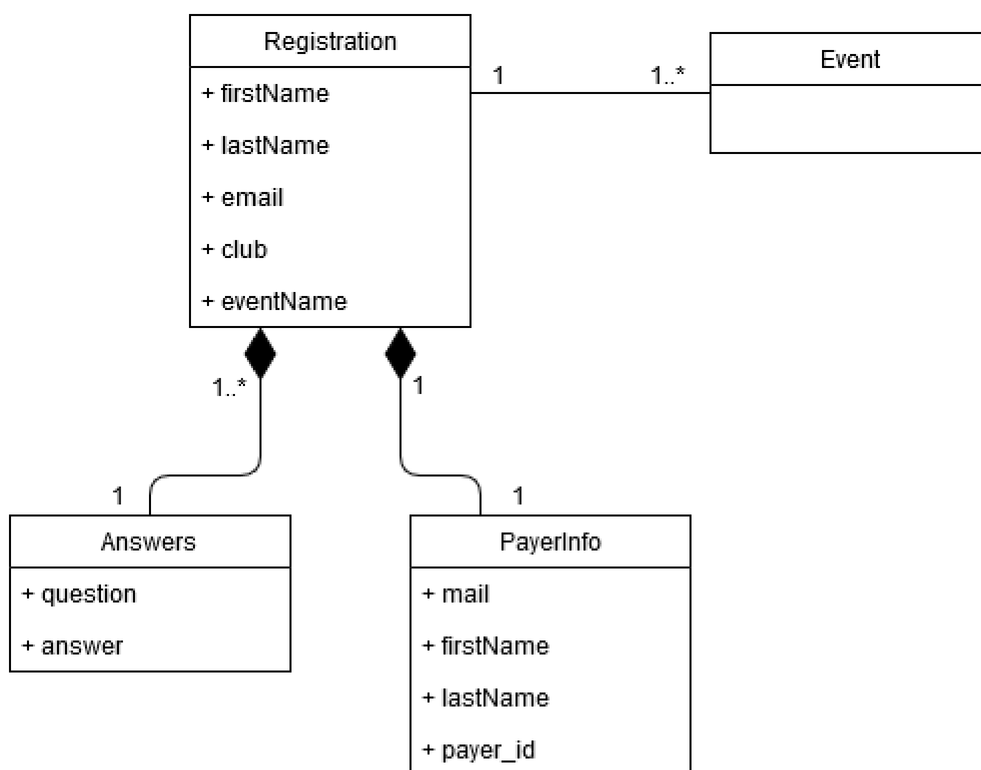

Figure 40: UML diagram of the registration object

The payer info element is linked to the payment system and will, therefore, be discussed in section 2.6.3

This configuration has a few drawbacks. First, all forms need to have one and only one question of type "email." This question is used as a point of contact with the registered person. If the form requires additional email questions, they will have to be sent via a text type question, which will not go through the mail validator. Secondly, as you can see in figure 41, every registration form needs to have a question named "First Name," "Last Name," and "Club" since those identifiers will be used in the frontend two separate them from the rest of the answers.

```
for (const q of props.form.questions) {
  if (q.type === "email") {
    setEmail(q.answer);
  } else if (q.name.toUpperCase() === "FIRST NAME") {
    setFirstname(q.answer);
  } else if (q.name.toUpperCase() === "LAST NAME") {
    setLastname(q.answer);
  } else if (q.name.toUpperCase() === "CLUB") {
    setClub(q.answer);
  } else {
    setAnswers((prevState) => {
      return [...prevState, { question: q.name, answer: q.answer }];
    });
  }
}
```

Figure 41: Loop that separates the personal data from the rest of the answers

This constraint leaves room for human error, as registration will not work unless the question has the correct name. The only thing I was able to do to make it less sensitive is to remove case sensitiveness by using the toUpperCase() function in the test. Nonetheless, the client will need to be briefed and warned about this specificity.

2.6.3 Payment system

Most of the events will require a fee to be paid upon registration. I had to find a way to incorporate this payment method in the registration process. This aspect was a sensitive topic, as it involves money. The system needs to work correctly, both for the sake of the federation's finance and it's reputation.

My initial choice was to outsource that part to Stripe. That way, I'd have integrated a stripe payment button in the form submission process, which would have sent all the payment data to the server. The payment info would then be rechecked there and sent to Stripe using the registration object's controller. This way, I'd have a double-check, both at the frontend and the backend, as seen in figure 42.

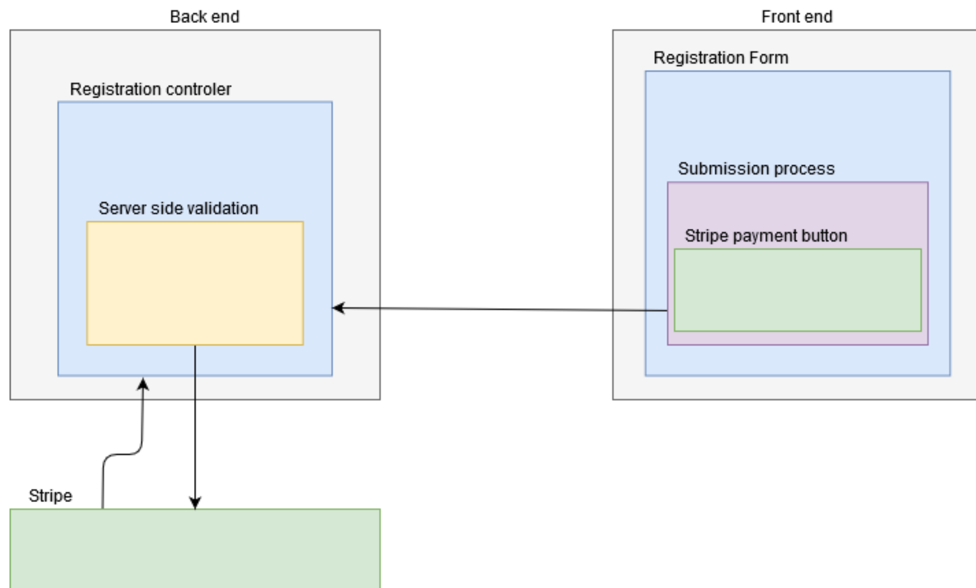

Figure 42: Stripe flow schema

Sadly the federation is an association, and for that type of entity, Stripe requires a VAT-number to activate the account. The federation being a nonprofit entity, it does not have a VAT-number, and could, therefore, only use Stripe in testing mode a.k.a without real money.
I had to switch for another provider and decided to rely on PayPal. PayPal does not offer any React support by itself, only plain javascript and HTML buttons. I had to rely on a third-party React button. This system works more straightforwardly than Stripe, as it doesn't go through the backend. When the customer uses the PayPal buttons, all the process is outsourced to PayPal, which handles the entire transaction. The app just receives a validation token if the operation is a success.

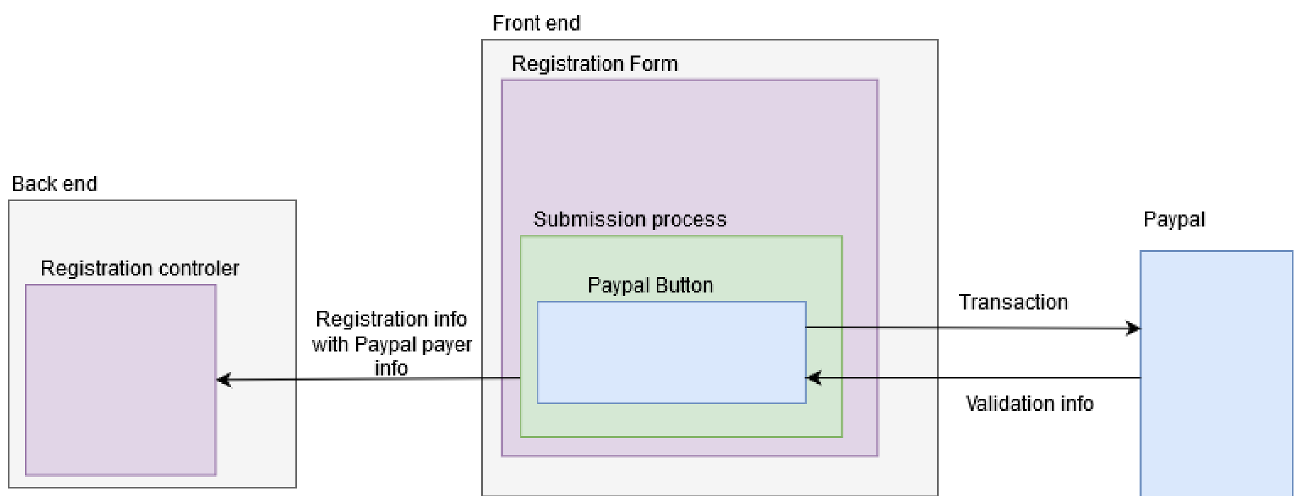

Figure 43: Paypal workflow schema

This method raises a few concerns—first, the reliance on a third-party component. Secondly, the server-side check added a layer of security. There should not be any problem as PayPal is a well-

known provider that has been on the market for years, but still, if the client decides not to rely on this system, PayPal offers an invoice sending service. If the client chooses to rely on this instead, I can just add a Boolean element on the registration element to see if the person registering has paid. Even though this method would not be as automated as the one with the third party button, and what the client initially envisioned during the initial discussion, it'll still be a substantial step-up compared to the one actually in place. In the end, by using Paypal, I gave the client two possible processes, and he'll have to choose which one he wants to use in the final version.

2.7 Media Sharing space

The client also required a space where the visitor of the website could share and view documents or videos related to the sport. I decided that the best place to put this element would be the Media commission space since it would be this organism that would manage and update it regularly. This sharing space didn't include pictures or images since it would be more training video and source documentation. Therefore, I decided to orient myself on a list presentation that would display media-elements.

The element itself is quite simple; the identifier is just there for readability from the user on the Strapi CMS. The type element is a selector with three options: Youtube, Document, Video. The first will simply open a new tab with the youtube link provided. The second will play the video in a new tab, and the third will either open the document in a new tab if it's a pdf or downloads it if it's something else.

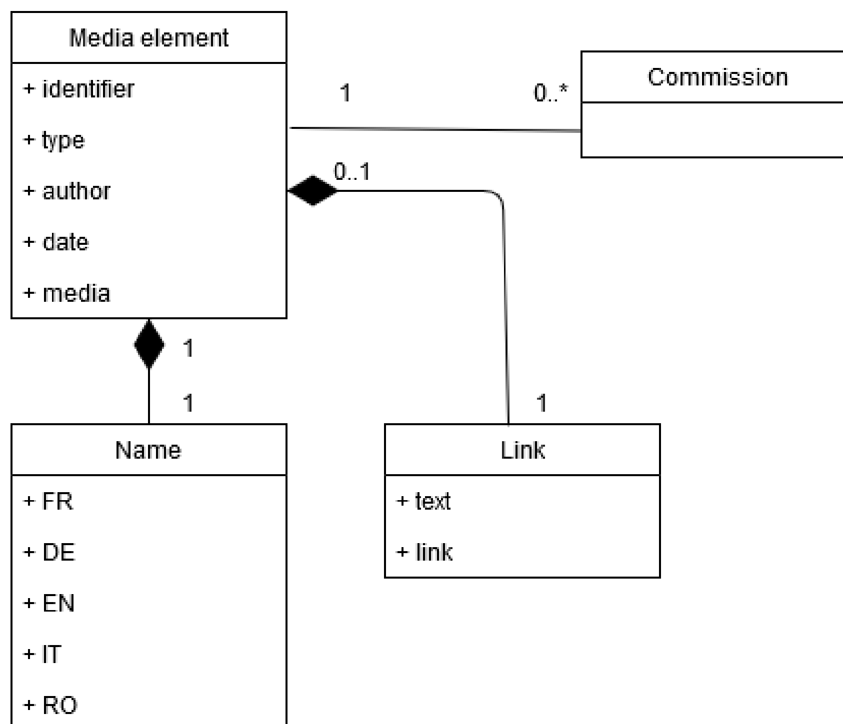

Figure 44: UML diagram of the Media Sharing component

Visually the component looks like a table with its attributes act as columns. The button is rendered depending on the type to be more explicit for the user.

Media Zone

Intro text for the Media Zone

Name	Author	Date	Type	Link
Trailer Swiss Hema Challenge 2015	Gladius et Codex	24/10/2015	Video	SHOW
Swiss Hema Challenge 2015 - Final Fight	Gladius et Codex	24/10/2015	Youtube	YOUTUBE
Statutes	Committee	06/05/2020	Document	DOWNLOAD
test	Me	07/05/2020	Document	DOWNLOAD

SEND US THE DOCUMENT YOU WISH TO SHARE

Figure 45: Media Sharing space on larger screens

On the mobile version, the date and author and type are not displayed to fit the table, even on the smaller screens.

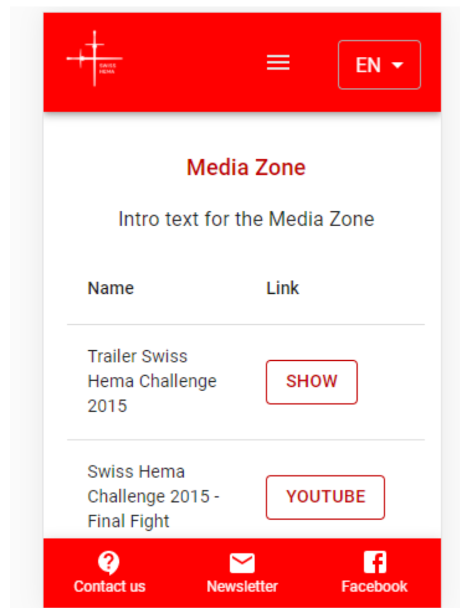

Figure 46: Media sharing space on smaller screens

If a user wishes to share a file with the community, I just displayed an email address for him to send the document to.

There are multiple reasons for this choice.

First, this element was a second priority from the client, and therefore I tried to keep the complexity at the minimum.
Secondly, the clients wished to have control over what is displayed, so if the user were to upload a new media element via form, I'd have to send an email to contact form the federation, for him to validate the document and display it on the list.
I can't just send an email using a contact form because of the attachment required. The attachment doesn't exist on the server, so I'd have to upload it, fetch it with the path on the server, send it back to the server as an attachment for the email, and finally remove it to avoid duplicates. This back and forth is entirely suboptimal. The simple creation of a media element is a bit problematic to do from a user form. I can't create a media element without the media existing to the media library of the CMS. So, I must upload the media to the library, fetch it again with the newly created ID and path, and use them to connect the media to the media element. While possible, if you add the fact that someone must validate said media element before publishing it, the complexity is rising exponentially for a module that was a second priority in the first place.

2.8 Event Archives

The event archives were a second priority element that the client wished for. Considering the time frame, I did not have the time to implement this element in the scope of this thesis. It was not an essential element for the function of the website, so it'll not significantly affect the final product. Even though I did not have the time to implement it, I did think about it and designed the first draft. I'll present this draft in this section instead.
The idea of the component was to save pictures, documents, and results for the past iterations of an event. The element itself would then be quite simple: The name of the event it's related to, the year the archive refers to, a description and two media galleries, one for the result (in case of a tournament), and one for the pictures.

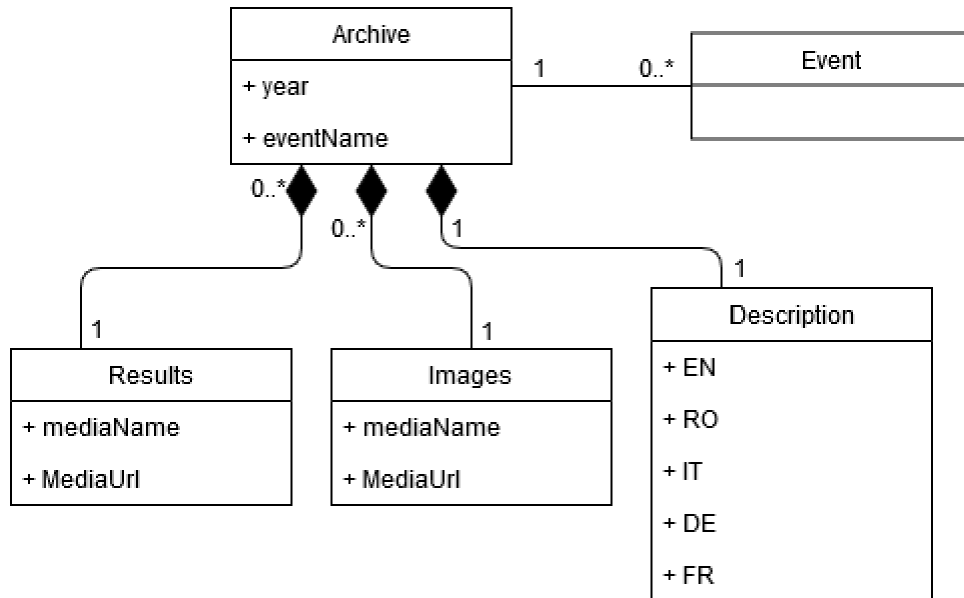

Figure 47: Archive element UML diagram

On the front end site, I just see this as an expansion panel containing a tab selector displaying the different years. Maybe in a mobile version, I will replace the tab selector by buttons for the same reason I replaced it on the commission page (see section 2.4.4). The element will be quite easy to implement and should not take much time to create after the end of this thesis

2.9 Publications List

An essential element of any information-oriented website is the ability to publish new information related to the association's activity regularly. To implement this element in the project, I developed the publication functionality, which is the most blog-like feature of the website.

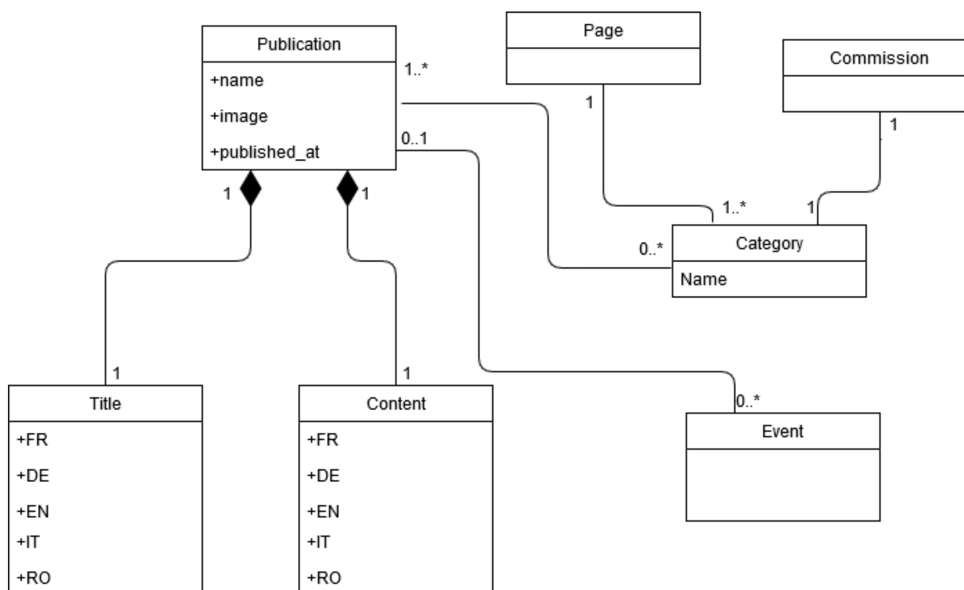

Figure 48: UML diagram of the publication list

To avoid having to copy-paste the publications at multiple places to have them displayed there, I linked that with a system of category. This element is straightforward and consists of only a name. It serves as an association class with other items, such as pages. This way, the app can dynamically decide where each publication will be displayed. The only exception to that being the About Us page because all publications are showcased there.
As you can see in the query in figure 49, a publication can have multiple categories, which is possible since the commissions of the federation can also organize events. In this case, the publication would be displayed in both the event and the respective commission page.
For de design, I needed to have a list that will display multiple elements dynamically without knowing how many from the get-go. I decided to use expansion panels, to avoid taking to much space on the screen and being able to display the image linked to the publication if it is expanded.

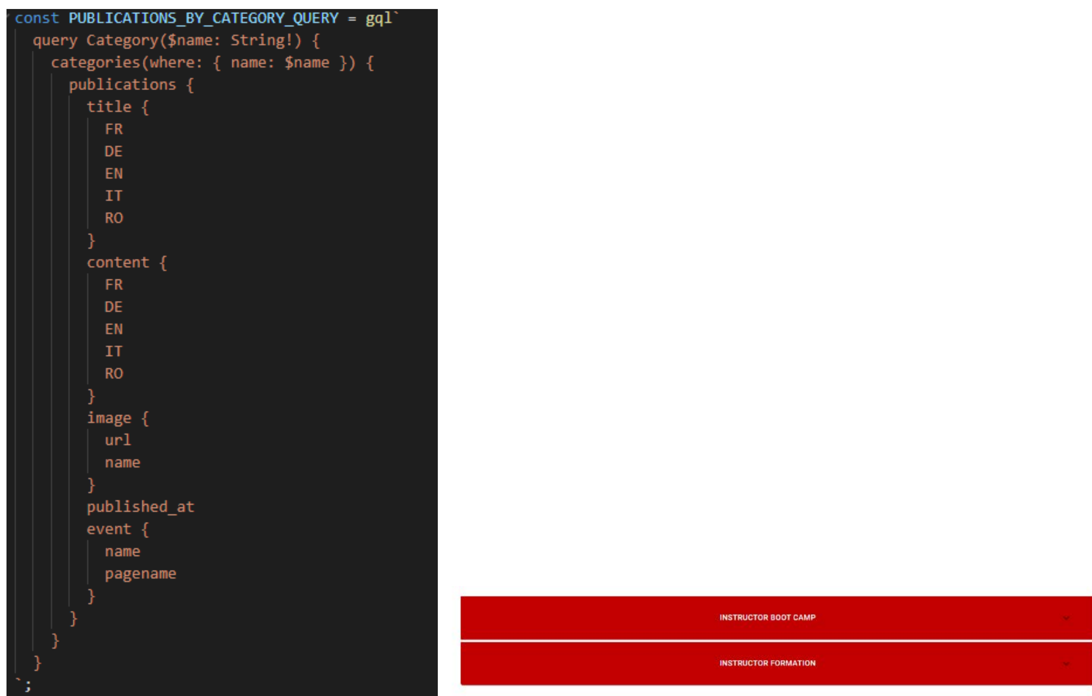

Figure 49: GraphQL publications query by category and result in the React App.

In figure 50, we see the publication list and, in figure 51, one of the publications in it's expanded form.

Figure 50: Publication list component

Figure 52: Publication expanded

On the diagram in figure 48, we also can see that the publication can be linked to an event. This specificity allows linking the page of said event to the publication via a button. Since not all publications are related to events, it is not an obligation.

Figure 49: Publication with link to an event

The main challenge for this part was to make sure the expansion panel would still be readable on mobile. I had to make sure there would be a different set of margins and padding to the element. The image was also a challenge for smaller screens; on the first version, it became small, keeping

the same padding as it's parent container. In the end, I decided to make the image full width on smaller screens to be sure everything was kept readable and visible.

3 Conclusion

After two and a half months of development, I concluded this thesis. If we look at the requirements as described in section 1.9, the client asked for the following features: a website with CMS capabilities that doesn't require technical knowledge to operate, that is multilingual, has an event registration system with payment capabilities, an event calendar, a newsletter, a document media library, and an event archive. If we compare these requirements to the final product as it is described in this thesis, all conditions but the archive component were met. The latter being a second priority for the client, the project can still be called a success even more so when you consider that this component is not too complicated to create. Furthermore, the design has already been described in this paper.

During the development process, the world encountered a global pandemic that disturbed the lives of everybody. On top of this, one of the people that acted as my client became a father. While this is excellent news, it reduced the time he had available for the project drastically. The other person acting as my client is currently finishing his master's thesis and, therefore, doesn't have that much free time available either. All those factors combined made the workflow I had planned at the start of the project fail. The initial idea was to have the clients continuously check the website as soon as features were out in a fashion that would be similar to agile development. In practice, most of my contacts with the client were about critical choices that needed to be made. For the most part, I was on my own. Now, in the end, they seem to be able to free themselves more and are giving more feedback. But due to the timeframe of this thesis, their input will be taken into account outside of the academic part of the project.

It was also the first project that I had to deploy on a server, and I lost some time at the start because it was something I had never done before and had to learn from the start. Luckily, my previous experiences with React allowed me to catch up on some of that time during the development process. Once the basic setup in places, I was able to deploy new functionalities at an ever-increasing pace.

The thesis is done, but the project itself can still be improved. The first step would be to implement the archive component. I did not have the time to develop during the academic timeframe. The second step will be to start testing everything with users. They may still be some bugs I overlooked or missed during the development process. The third step would be to check all the texts, labels, and pictures used to ensure everything is correctly written, translated, and at a high quality. The fourth step would be to produce a small tutorial for Strapi to make sure the other members of the federation's committee understand how the CMS works. Finally, the last step would be to refactor a part of the code. My thought process has evolved and changed during the development phase as I became more used to the tools I was using. Therefore, I would not develop some components I coded at the start of the project the same way now that I reached the end. With the knowledge I

have now, it is guaranteed that I can optimize parts of the website without changing the core of its functionalities.

In conclusion, this project, with its difficulties and obstacles, has allowed me to develop my skills and reach a better understanding of the technologies I was using. With this new knowledge, I can improve the project and develop myself further.

4 References

Wikipedia 2020. GitHub. URL: https://en.wikipedia.org/wiki/GitHub#GitHub. Accessed 18.04.2020.

Wikipedia Fev 09, 2020. Material Design. URL: https://en.wikipedia.org/wiki/Material_Design. Accessed 18.04.2020.

Howtographql 2020. Introduction. URL: https://www.howtographql.com/basics/0-introduction/. Accessed 18.04.2020.

Firebase 2020. Fast and secure web hosting. URL: https://firebase.google.com/products/hosting. Accessed 18.04.2020.

Scrum 2020. Scrum Glossary. URL: https://www.scrum.org/resources/scrum-glossary. Accessed 10.05.2020.

ABRAMOV D. 2015. Redux A Predictable State Container for JS Apps. URL: https://redux.js.org/. Accessed 19.04.2020.

HESLOP, B. 18 Dec, 2018, History of Content Management Systems and Rise of Headless CMS. URL: https://www.contentstack.com/blog/all-about-headless/content-management-systems-history-and-headless-cms. Accessed 18.04.2020.

HUNT, P. O'SHANNESSY, P. SMITH, D. and COATTA, T. 2016. React: Facebook's Functional Turn on Writing JavaScript. *Queue,* **14**(4), pp. 96-112.

MARIA, R. RODRIGUES, J. Luiz and PINTO, N. 25 Oct 2015. ScrumS, Oct 25, 2015, ACM, pp. 43-47.

MAURER, F. and MELNIK, G. 28 May 2006. Agile methods, May 28, 2006, ACM, pp. 1057-1058.

SEABRA, M. NAZÁRIO, M. and PINTO, G. 13 Sep 2019. REST or GraphQL? Sep 23, 2019, ACM, pp. 123-132.