Politecnico di Torino

Master Degree Thesis

Study and development of a mobile-oriented application for the efficient management of a radiation test

Master degree course in Computer Engineering

Thesis Advisors:
Prof. Luca Sterpone

Candidate:
Antonio Giordano
matricola *182698*

July, 2020

Abstract

Nowadays, electronic devices are the major part of many applications starting from automotive market to military equipment, from personal computers and cellphones to satellites and spacecrafts. Each application has its own requirements and specification to be fulfilled by the designers. Some of these applications known as mission critical application are involved in a huge amount of money and resources that could be lost if something goes wrong. Therefore, electronic devices used in the mission critical application require special attention in terms of dependability to guarantee that they can tolerate faults induced by environment that could lead to the failure of the entire system. One of the main environmental aspects that can reduce the reliability of electronic devices used in mission critical application is radiation. In space, several kinds of radiation can interact with the electronic devices and create faults and damages. Strategies and techniques should be applied to the used electronic devices to tolerate faults and errors. However, the applied techniques and methods should be analyzed and tested to confirm their efficiency. Performing radiation test in the radiation facilities in one of the most efficient method to emulate radiation effect on the electronic devices since it provides the most realistic scenario with respect to the space environment. However, due to the high cost of radiation test, typically, engineers and designers can afford few hours of radiation chamber to perform the test on their developed design and collect the huge amount of results of their test. Given the high number of these necessary tests, and the conditions in which these tests take place, i.e. the limited time available, as well as the limited if not absent internet connection, lead to the development of more efficient flow management tools. The goal of my thesis was precisely the creation of a tool that supports the execution of a radiation test providing a comprehensive and reliable archive for radiation test projects by:

1. Managing all static material, from design sheets to datasheets
2. Associating the corresponding facility where it would have taken place
3. Preparing in advance all the tests to be carried out, with data relating to ions, number of bits, and being able to catalog each execution with a system of tags
4. Seeing the results in real time without having to rely to custom excel sheets, prone to accidental errors.

The developed application was therefore tested on the results of a real case of radiation test performed at CERN facility.

Contents

CHAPTER 1

Introduction

The development of a web application involves at least two parts, a server and a client, or backend and frontend.

In the standard server / client model, the backend is that IT system that manages protected resources, static files, access to the DBMS, user authentication, and all the application business logic that for architectural, security, or of performance, must run server side.
The client, also called frontend, is the set of static resources and public code accessible to the end user, also including the UI part and all the business logic for the communication with the server and for populating the graphical interface.
In the development of this tool, it was necessary to deepen the HTTP communication protocol at the base of the world wide web and the internet, in order to have the theoretical knowledge preparatory to the development of a web application.
So a paradigm of data exchange between client and server was chosen, in this case the REST pattern.
Javascript was chosen as the programming language, for its peculiarity of being able to be implemented both on the server side, through the NodeJS runtime processor, and on the client side, where it has been natively used for years for the scripting of web pages, with the aim to make dynamic and changeable the HTML language which is a purely static language using XML.
In particular, HapiJS was chosen as the framework for the development of the backend APIs, in order to have access to all the work done by the community in terms of open source software ready for use and tested and optimized over the years.
MongoDB was chosen as DBMS, for its excellent integration with Javascript and its non-relational logics which came in handy during development.
On the frontend side, ReactJS was chosen, the reference library in the Web environment on a par with Angular and Vue.
For the management of the business logic, Redux was chosen instead.
We will see later how and why of choosing these solutions.

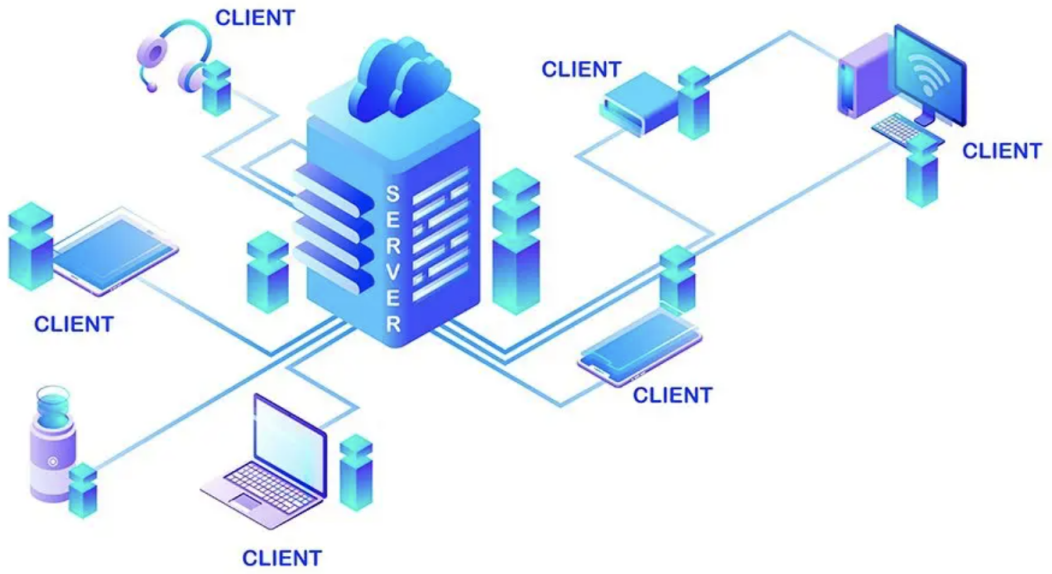

Figure 1.1: Server / client architecture [2]

1.1 HTTP

For the development of the tool, a classic server / client system based on the HTTP protocol was chosen.
The HTTP (Hypertext Transfer Protocol) is the de facto standard of the World Wide Web since 1990 [1].
Its use has in fact allowed the creation of documents containing hyperlinks for quick navigation from one web page to another.
The protocol allows it to fall within the classic request / response paradigm of server / client systems, and basically provides for: 1) a URL (Uniform Resource Locator), or a unique address that identifies a resource within the network 2) a method, or verb, which describes the operation to be performed on the resource.

1.2 HTTP request

Each HTTP request contains several pieces of data, each of which has a specific purpose.
The first section is the header.
The header of an HTTP request contains a set of "configuration" information or specifications about the request itself. The format of an HTTP header is that of the key / value pairs.
There are very common and standard keys, such as: Authorization, used to pass authentication data, Content-Length, to specify the total length of the request packet, Content-Type, to specify the type of content sent in the body of the request, User-Agent, which identifies the type of client that is making the request or the Cookie, very common as a term also in daily use, as it contains sensitive data such as authentication data, usage statistics and user

preferences.
We have seen how every resource on the web is identified by a URL, which is a unique address.
The same URL also carries other standard information used by server-side logical business.
The first of this information are the so-called path parameters, and they are specific portions within the URL that are parsed by the server and used for example as filters on the DBMS.
For example, the URLs:

```
http://localhost/users/1
```

and

```
http://localhost/users/2
```

requested for reading, are managed by the server in a completely similar way, but the last parameter (1 in the first case, and 2 in the second) is usually interpreted as the ID of the users object to be read.
The other information passed through URL is the so-called query string.
This string is another set of key / value pairs, "attached" to the end of the URL after the special character "?".
Each pair is separated from the other by the special "&" character.
The key and value are separated by the special character "=".
For example, the URL:

```
http:/localhost/users?orderBy=name&limit=10
```

in read request, could tell the server to read the first 10 users sorted by name.
Finally, an HTTP request that goes to create or modify a resource on the DBMS always provides for a body, also called a payload.
This content, at a protocol level, is basically binary data, but it can represent the content of a static file, during an image upload for example, or more often the content of a resource in a serialized way, for example through a JSON object.
In order for the server to know the type of body format passed, it is important to specify the body's content type, using the appropriate "Content-Type" key of the HTTP header.

1.2.1 HTTP methods

The main methods are GET, PUT, PATCH, DELETE and POST.
GET is the method used to read a resource, whether it is an HTML page, a static file such as a stylesheet or a Javascript script, or even the description of a resource as a user, or a post, in serialized format, typically XML or JSON.
The POST method is the method for creating a new resource within the computer system controlled by the server.
Usually, along with the request, a payload is passed, with the content of the resource to be created.
The PUT method is used to modify an already existing resource.
Usually the URL is aimed at the target resource, and the payload specifies the new content (in full) of the resource itself, excluding the ID or primary key that identifies it.

The PATCH method makes a partial modification of the resource.
In this way it is not necessary to indicate the new content in full if, for example, only one property of the resource is modified.
The DELETE method is ultimately used to delete a resource from the DBMS.
It does not include a payload, but only the unique URL that points to the resource to be deleted.

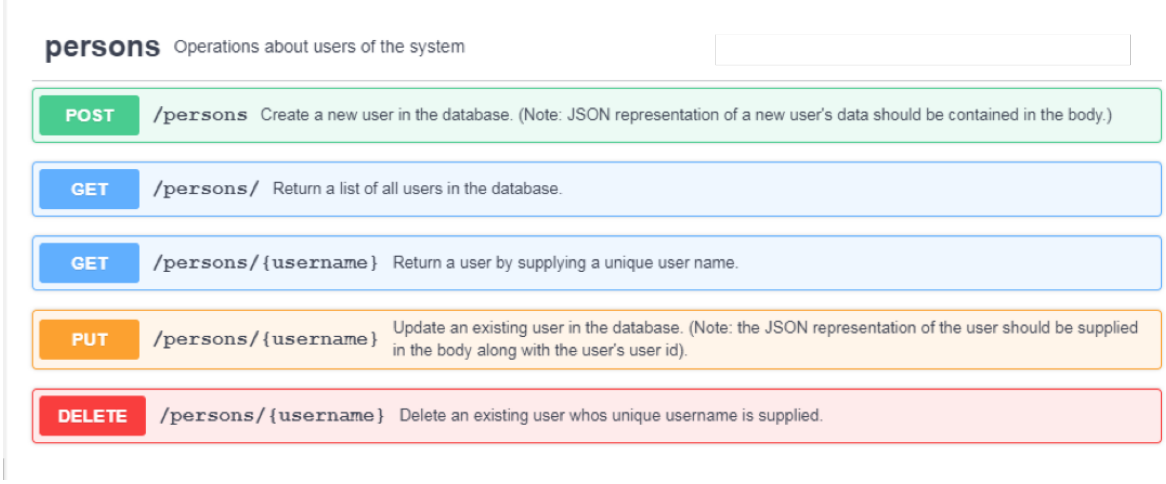

Figure 1.2: HTTP methods example [3]

1.3 HTTP response

At each request the server should provide a response.
The response on the server side also contains information inside a header, for example for saving a new cookie, or for specifying the type of content returned and its length in bytes.
The response also contains a payload, which can represent an HTML page or other static files or serialized data in XML and JSON.
In addition, each response contains a status code, which quickly and accurately identifies the outcome of the request.

1.3.1 HTTP status codes

The status codes are basically divided into groups of hundreds, the 1**, 2**, 3**, 4** and 5**.
For example, status codes 2 ** indicate a successful status. Code 200 is the correct request code par excellence, especially when reading.
The 201 instead is the code of successful creation of a resource.
The 1** codes are called information, the 3** redirection codes.
The 4** codes instead indicate client-side errors, for example the famous 404 error indicates the request for a resource, or URL, not found.
The error 400 indicates a bad request, the 401 an unauthorized request (for example, the request to read a private resource before logging in).
Finally, all 5** errors indicate an internal error on the server side.

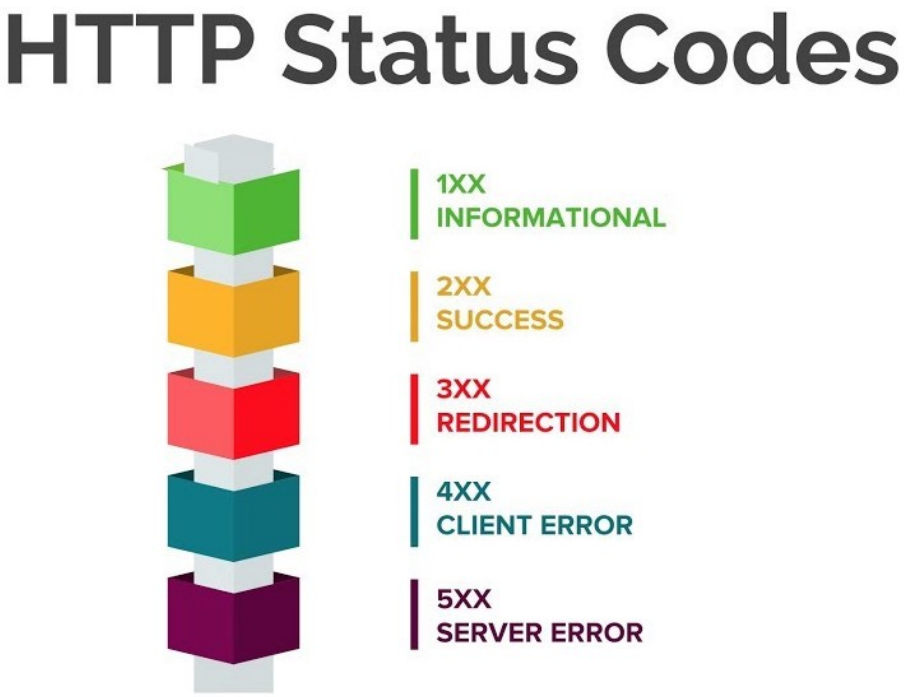

Figure 1.3: HTTP status codes [4]

1.4 The REST pattern

The Representational state transfer (REST) pattern is a software architecture for creating web services [5].
A web service can be simplified like an HTTP call.
A web service compliant with the REST pattern, called RESTFul, is based on the concept that a URL identifies a specific resource.
Each HTTP method corresponds to a specific operation on that resource.
The payload, of a textual type, usually XML or JSON, specifies the content that that resource must have, following the creation or modification calls (total or partial) corresponding to the POST, PUT and PATCH methods.
Furthermore, to be RESTFul, communication must be stateless.
Stateless means that all the information necessary for the correct execution of the call to the web service is contained within the HTTP package, that is, that there is no information previously held by the server necessary for execution.
An example of a non-stateless connection is for example the FTP protocol.
In this type of connection, the server constantly maintains information on the connected client such as authentication data, the current working directory, the current transfer method, etc.
Therefore, two identical FTP calls, but performed at different times and contexts, giving two different results, are not considered stateless.

The HTTP protocol is stateless in nature, since each packet contains all the information necessary for the execution of the request on the server side.
However, server-side implementations at the logical level cannot be taken for granted that they are stateless in the same way, which is why the REST paradigm became a standard over the years.

1.5 Javascript

The language adopted for the development of the web application, both backend and frontend, is Javascript.
Javascript was born as a client-side scripting language.
With the birth of the WWW, the web pages written in HTML complied with the aim of creating a static navigation based on hyperlinks, but the experience within the single web page was that of consulting text and images, and little else.
Over the years, it has increasingly been necessary to give dynamism to web pages, allowing them to be modified and populated even in partial parts in realtime.
Javascript was born exactly with this goal.
The birth of client-side libraries such as jQuery have dramatically increased its popularity and use, until the Google V8 engine, at the base of the compilation of Javascript within Chrome, was included in an application software that could run directly within the operating system, creating NodeJS.
Over the years, the language itself has evolved ever more, to meet the most diverse needs on both the client and server sides, where NodeJS has become increasingly popular and used by the community.
On the client side, at the same time, the evolution has continued, especially thanks to two large digital realities such as Facebook and Google who have developed React and Angular respectively in open source model.
These two frameworks have created a new client-side development model, almost completely replacing the previous one, based on HTML + CSS + JS, with a new model based entirely on JS.

In particular, for React, the new templating language called JSX has been adopted, which is a mix between XML and Javascript.
Furthermore, new libraries and frameworks in JS have created new possibilities also in terms of style, allowing the developer to create style sheets or inline styles directly in Javascript, using the styeld components or CSS-in-JSS technologies.
In fact, very often you don't even need to write a line of CSS code, but you can directly access ready-to-use component libraries, such as Material-UI.

1.6 NPM

The popularity of NodeJS derives also and above all from the great work carried out by the community, which over the years has developed countless libraries, frameworks, or simple utilities, some of which are now indispensable for development with download peaks of several million.

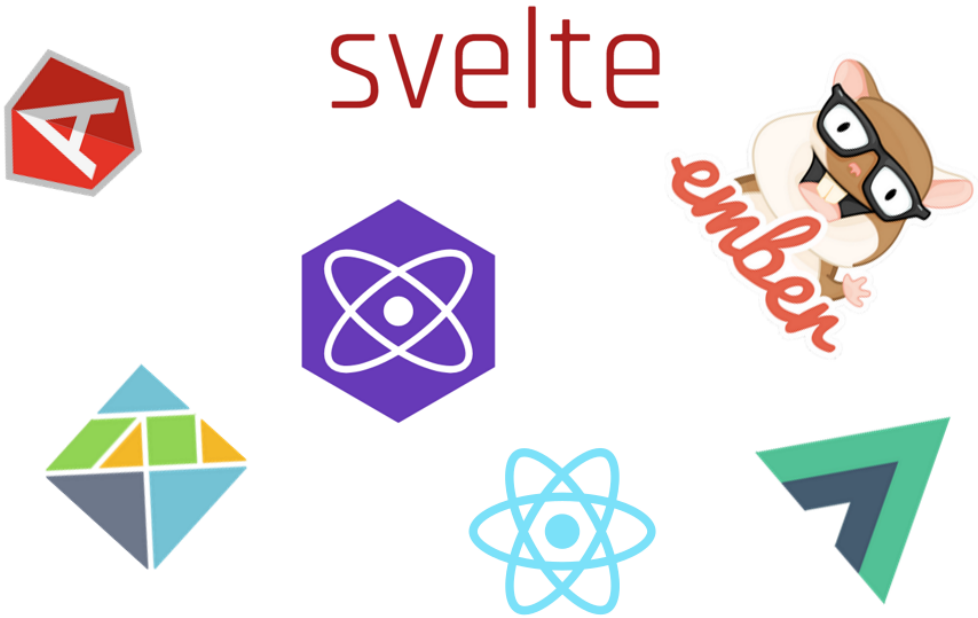

Figure 1.4: React and Angular are just the most two popular frameworks among the others

The largest and most popular NodeJS package repository is NPM.
NPM contains millions of packages, some of which are downloaded only a few times, others used daily globally countless times. The same React and Angular for example are available as NPM packages.
Installing a package is trivial, just run the "npm install -package-" command from the terminal inside the project folder.
The npm command is available immediately after the installation of NodeJS, in fact both executables are distributed concurrently, facilitating and speeding up accessibility and use even more.

Figure 1.5: Npm logo

CHAPTER 2

UX Analysis

The development of an application, whether desktop, mobile or web, should always start from a preliminary study on the objective to be achieved.
First of all, it is essential to define the user of the application, his habits, what he needs during use, what he wants to achieve, and how to get it in the shortest time and number of operations.
In our case, the typical user of the platform is an engineer or scientist who works on a daily basis with IT tools, which therefore has a medium-high digital literacy level.
Once the target is found, it is essential to speak with her to collect all the information necessary for the construction of an effective user experience.
Having gathered all the information, in terms of technical specifications, we move on to the creation of a prototype.
The prototype can be static or semi-dynamic.
A prototype of the latter type offers not only the possibility of analyzing the correctness screen by screen, but also of being able to navigate, according to "fake" or in any case pre-established paths and interactions, the so-called user journeys, or those typical navigation maps of the user while using the platform.
Once the prototype has been defined, tested, and approved, we move on to the design phase, in which the screens come to life in graphic form, choosing a color palette, fonts suitable for the case, and perhaps a library of components so as not to start from scratch.

2.1 Interview

The first phase of UX analysis concerns the interview with the stake holders.
During this phase, the state of the art was first evaluated, such as the tools currently used by the engineers during radiation tests to refer to datasheets, papers, various notes, details on the structure, etc.
From the first interview, a platform with user authentication was required.
In fact, it would be unthinkable to create a single repository of files, data, calculations, etc., for all users indiscriminately.
By being able to authenticate, you can guarantee the security, confidentiality, and maintainability

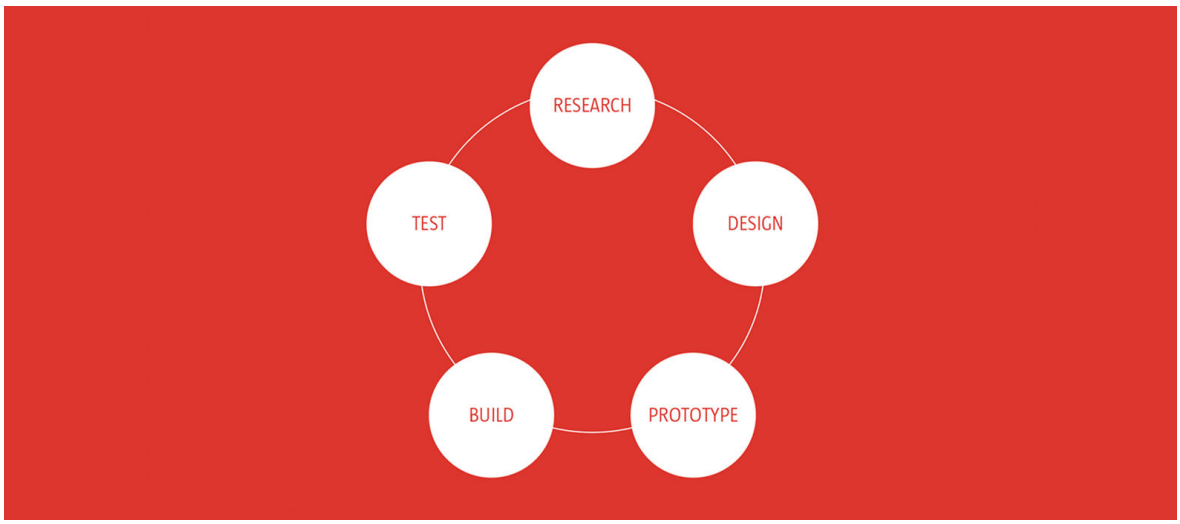

Figure 2.1: UX research flow [6]

of the resources and all the data loaded.
After logging in, a first subdivision of the working environment by projects was envisaged. Each project can be considered as a "mission", that is, a visit to the infrastructure.
Each project in turn is a set of 3 specific phases:

1. the first is that of the collection of technical material, useful during the experiment
2. the second is that of carrying out the tests, where the various tests carried out and populated are inserted with the data
3. the third and final section is that of the results, where the algorithms are applied to the entered data, and graphs are generated

A further need of the engineer is to insert, read, modify and eliminate facilities.
The facilities are the places where the tests are carried out, and each one has more or less constant peculiarities over time that it would not insert every time a project is planned, but should always be available as an editable catalog and from which to select quickly from time to time. in turn the facility for that particular project.
Once all the general information has been collected, we proceed with the creation of the so-called user journeys, i.e. those maps of user interaction with the platform to achieve the set objective.

2.2 User Journey

A user journey (or experience journey) tells the interaction between a user and a service, through a summary representation of all phases of the experience and the description of all actions (or activities) step by step.

The user journey is represented by tracing the timeline along a horizontal axis: all phases of the experience are listed along the timeline following a logical sequence of interaction between user and service.

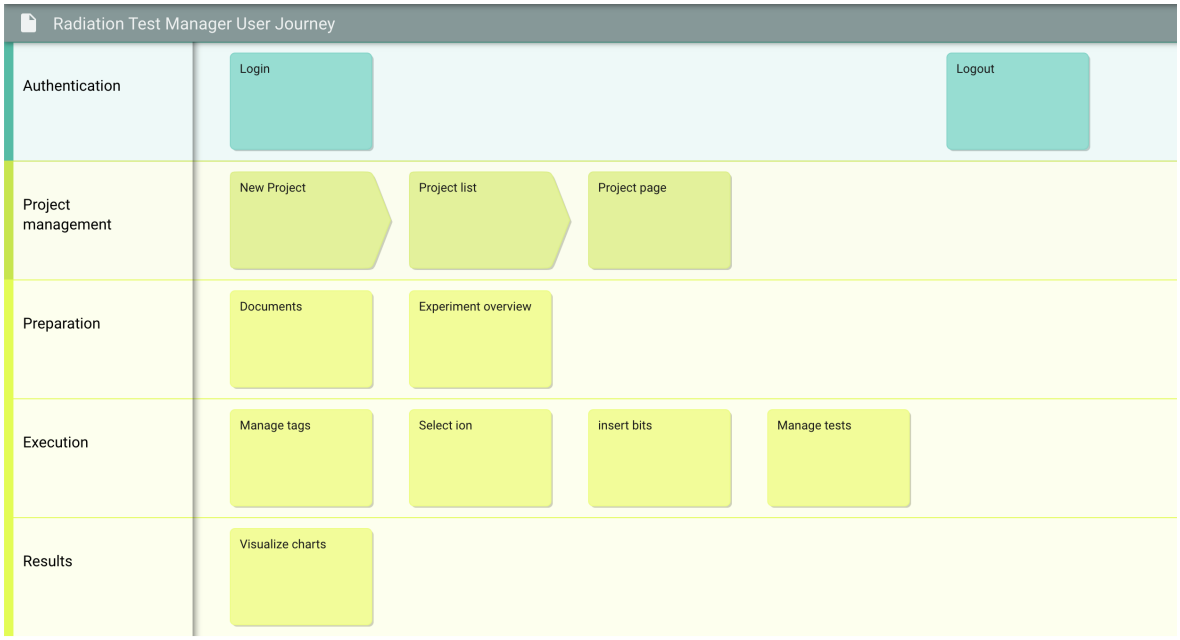

Figure 2.2: The Radiation Test Manager user journey

For each phase, the activities that the user performs, the critical issues that hinder the path and the consequent level of satisfaction or frustration in the experience are then specified.
The user journey allows at first to generate ideas starting from existing critical issues, and at a later time to tell and validate the new elaborated model, illustrating how the user experience is taking place now and how it could be carried out in the future.
In the analysis of the UX relating to the application, a simple user journey was produced which included all the main phases of the activity resulting from the first interview.
In Fig. 2.2 it is possible to view the scheme obtained, divided into phases.

A first line represents the authentication phase, trivially login and logout.
Already from this simple line it can be deduced that currently the web application does not provide the classic features in the platforms that we use every day as a user registration flow or a password recovery flow.
The choice was made because these operations were considered off topic with respect to the technical development of the platform.
Operations of this kind in fact provide for example the use of email sending services to confirm ownership of an account that has been ignored on purpose for this first phase.

Once logged in, the next need that came out was that of a CRUD service (create, read, update, delete) related to the projects.
In fact, at each access, the first mandatory operation is to create or select an existing project that acts as a work base.
Once the project has been selected, it is possible to navigate between 3 macro sections, preparation, execution and results.

Within each of these sections the needs and current problems have been identified, and starting from them a flow has been designed that takes into account everything.
In particular, in the preparation phase, there are two basic activities, organizing a file

repository that is easily accessible and categorizable exactly like a file system, and having access to the creation, modification and selection of a list of facilities.

During the execution phase, on the other hand, it is necessary to quickly create and modify tags to be assigned to the various tests, select from among all those that the facility makes available the desired ion, and insert or import the experiments carried out with the relative results.
Finally, the results section is a simple static display page without interaction.

2.3 Prototype / Wireframe

The prototype represents the creation of what will likely be the final product. It is an evolution of the wireframe concept, which represents the front-end skeleton of the web application, where the correct position of the main content blocks is ensured, a correct structure of information and the main user-interface interactions are described.
The extra prototype is interactive but in general the same rules related to the creation of wireframes as flexibility (ease in making changes) and low fidelity to the final product apply.

Producing a prototype costs much more time than a wireframe, but it represents the most powerful means in the testing and documentation phase as this type of documentation represents a real prototype of the application (desktop, mobile or web that it is) and allows the user to test the browsing experience.

The prototype also serves to validate specific technical feasibility.

The pages of a prototype can be graphed in a very similar way to the final site, but they will be composed of few indispensable contents, just those necessary to fully expose the project.

Prototyping is a process that also serves to validate specific technical feasibility thus avoiding returning to consider any variations that may lead to dead ends.

For the prototyping of this web application, Adobe Experience Design (XD) software has been considered.
One of the most important features of Adobe XD is that related to the test phase that allows you to see how the project looks in desktop and mobile version, thus testing the various features and correcting where necessary. While in Design mode the graphics are created and structured, the Prototype mode allows you to connect the navigation between multiple graphic tables, set the Home and prepare everything for the final Preview.
Once the connections between the screens have been created, you can start with the actual test, giving life to the actions and functions designed in the first phase of the study work of the user journey.

In order not to start completely from scratch, a preset of ready-to-use components, imported from the official Material Design library, was also used.
Material Design is a graphic concept widespread in recent years, thanks also and above all to the push of Google that adopts it in practically all its Web products as well as on Android devices.
Thanks to its diffusion and practicality, multi-level tools have been developed to make the most of it.

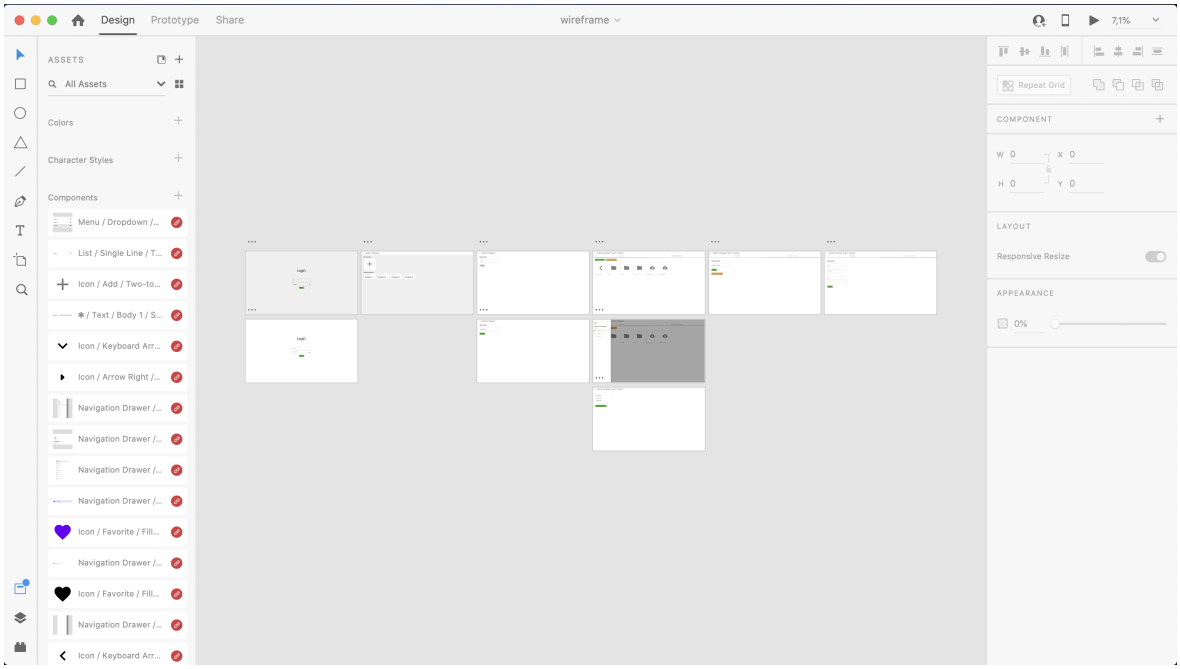

Figure 2.3: The Radiation Test Manager screens wireframe

As for example the first mentioned component library for Adobe XD but also as we will see later a library of web components in React ready for use.

Taking advantage of an already established graphic model, it was possible to equip the application with a functioning graphic design without necessarily having to design and develop everything from scratch.

Once the basic library was chosen, it was possible to build the individual tables on XD 2.3.
Each table corresponds to a screen. Following the flow of the user journey, it was possible to populate the individual screens with the appropriate components.
Once the screens were populated, the connections were made from one to the other, to test the correctness and practicality of the whole 2.4.

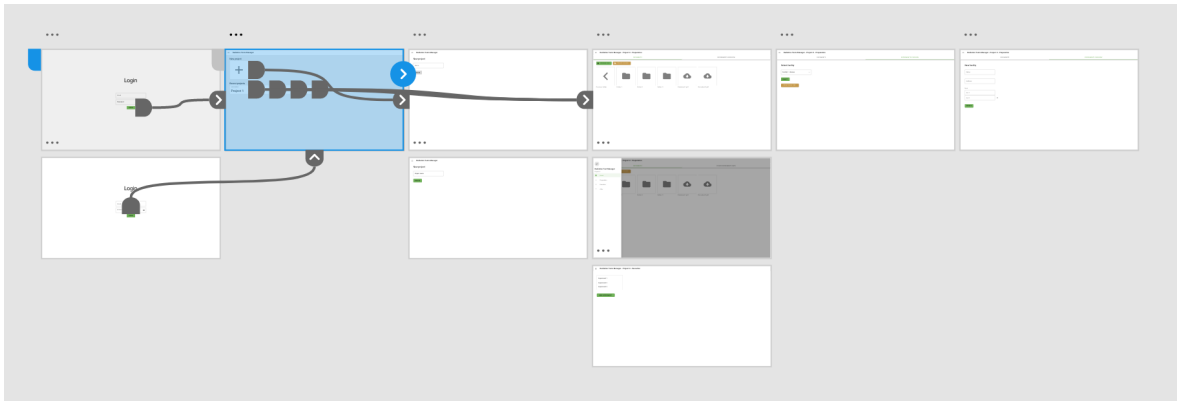

Figure 2.4: Example of interactions connections inside a screen

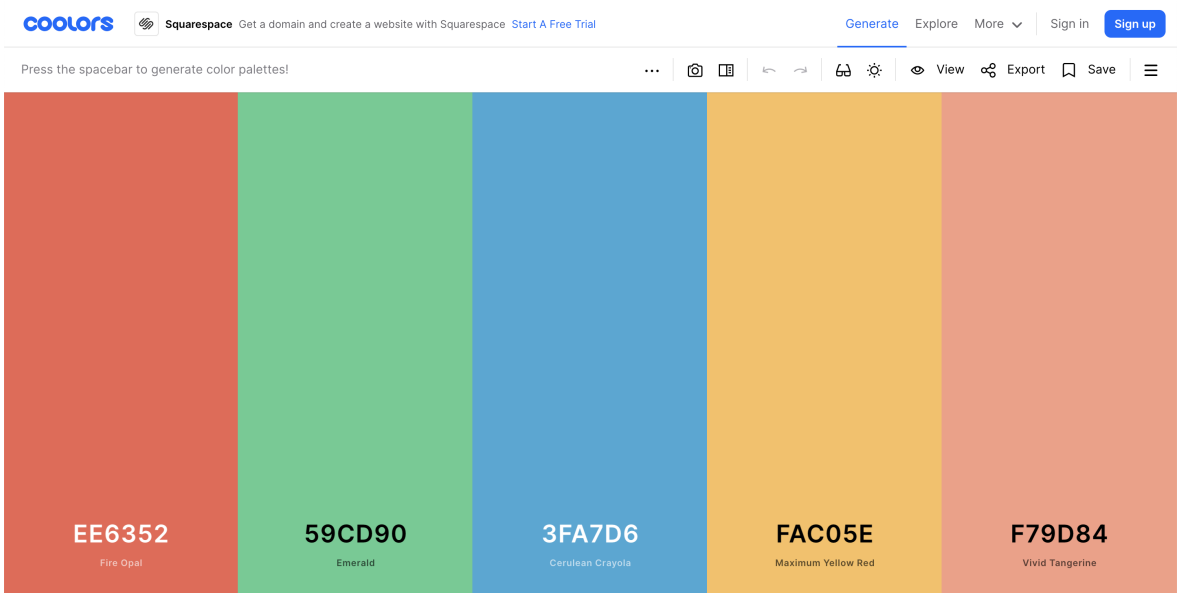

Figure 2.5: Example of palette generation with a free web service [7]

Thanks to the construction of the prototype, every single problem was dealt with on time, and at the end of the process it was possible to start with the development, limiting as much as possible the risk of a possible refactoring during the race.

2.4 Graphic Design

At this point, it would have been necessary to develop the final graphics starting from the prototype. Since the creation of a design was off topic, a graphic palette was simply generated with two colors 2.5, a primary and a secondary one, and was directly adopted on the web components which for the initial choice mirrored those of the prototype, both starting from the same graphic concept of Material Design.

CHAPTER 3

Backend development

In the client / server architecture, the server takes care of receiving client-side requests, processing them, usually accessing a first-level storage for authentication, and a second-level one such as one or more DBMS for access to data. It usually also distributes static files to the client, such as HTML, CSS, Javascript scripts, but also images and other multimedia content.

The software that runs on the server side is called the backend. The language chosen for the development of the backend is Javascript, implemented through NodeJS.

NodsJS can run locally, both on unix machines and on Windows, as well as on any type of dedicated and virtual machines. All cloud computing services support NodeJS.
AWS for instance allows the execution of NodeJS (as well as obviously on dedicated Linux machines) on various proprietary services.

Figure 3.1: AWS (Amazon Web Services) logo

Among these there is Elastic Beanstalk (EBS) 3.2, which is a service that takes as input a zip file with the source files of a project in NodeJS inside, and with each deployment of a new version it takes care of downloading its dependencies from the repository, and to launch the application.
In addition to this, it has many other features, including retrying the startup of the application a certain number of times in case of error, or the possibility of configuring a load balancer (for example NGINX) that allows you to set auto-scaling policies automatic.
In this way, as traffic increases, for example, when the single machine cannot handle all the requests, the system automatically starts a new machine, copying the zip file, reinstalling all the dependencies, and therefore running the application on it.

At this point, the load balancer automatically takes care of redirecting the requests to the most "unloaded" machine. When the use of the machines falls below a certain threshold, the second machine is "unmounted" and the service costs are amortized.

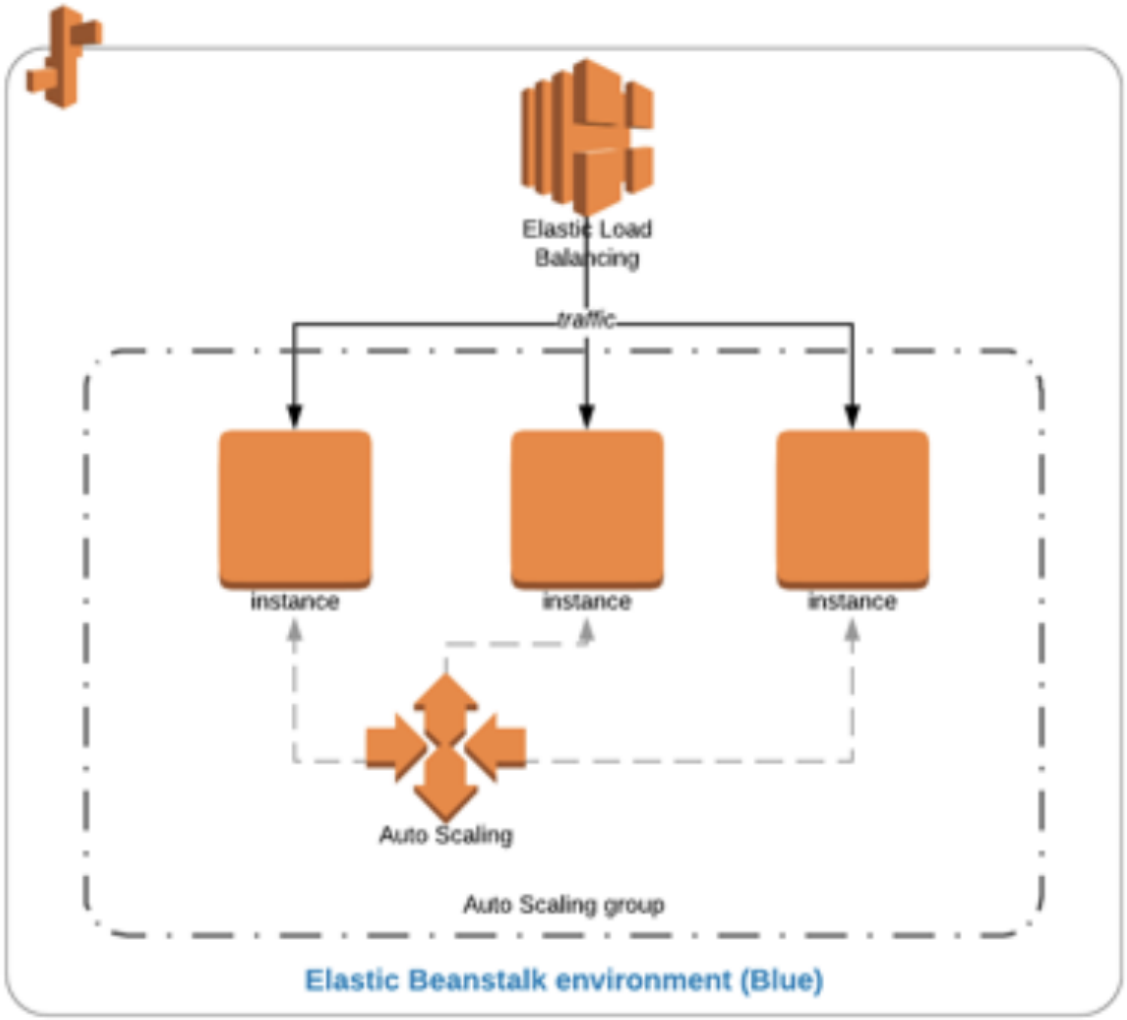

Figure 3.2: AWS Elastic Beanstalk architecture

AWS also supports Docker and Container based technology. Again, you can create a NodeJS application, install it on a ready-to-use Docker image, and upload it to AWS through the ECR service. Another service, called ECS, instead deals with running Docker images on physical servers with the same auto-scaling policies seen above.
An advantage of using Docker is that of greater flexibility in the use of different versions of NodeJS, since EBS for example supports a limited number of them.
In addition, sometimes errors occur during the installation of dependencies on EBS that do not occur with Docker-based technology, because the Docker image contains everything needed to run, without having to perform any preliminary actions.

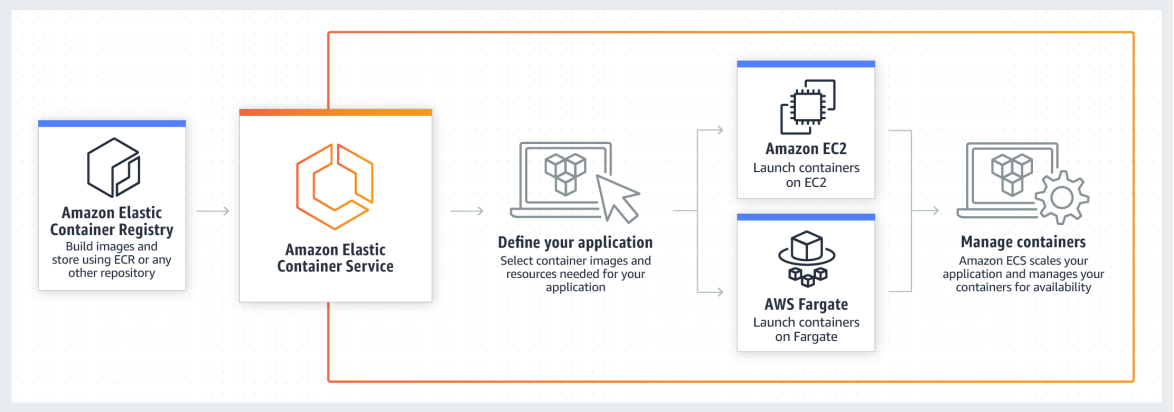

Figure 3.3: AWS ECS architecture [8]

Another service present on AWS is that of the lambda functions. A lambda function is a single code script (Javascript is supported, as well as Python and other languages) that runs on demand.
It does not require any type of configuration, it is pure and simple code.
The use of lambda functions has allowed the development of a revolutionary new backend architecture called "serverless" (3.4). In this architecture, the developer no longer has the duty to build his server, even only on a logical level through code, but he only cares about the business logic of each single endpoint, or route, or pair of URL / HTTP method.
On AWS all this is possible thanks to a service called API gateway, which takes care of input a list of endpoints or routes, and related associated lambda functions.
It will be the same AWS to take care of creating a virtual server, to which a single domain is associated, and to connect to each individual client side request the respective execution of the lambda function, providing to pass all the parameters (body, query string, etc.).

In order for these services to be usable, it is important to develop the server following the RESTFul pattern, that is according to stateless specifications, since at any time, both on EBS, on Docker technology, and on lambda functions, the code could always run in different environments , and any local dependencies, be it the file system, or top-level memory, would compromise the functionality of the system.

3.1 NodeJS

Node.js is an event-oriented cross-platform Open Source JavaScript runtime for executing JavaScript code, built on Google Chrome's V8 engine (3.5). Many of its basic modules are written in JavaScript, and developers can write new modules in JavaScript.

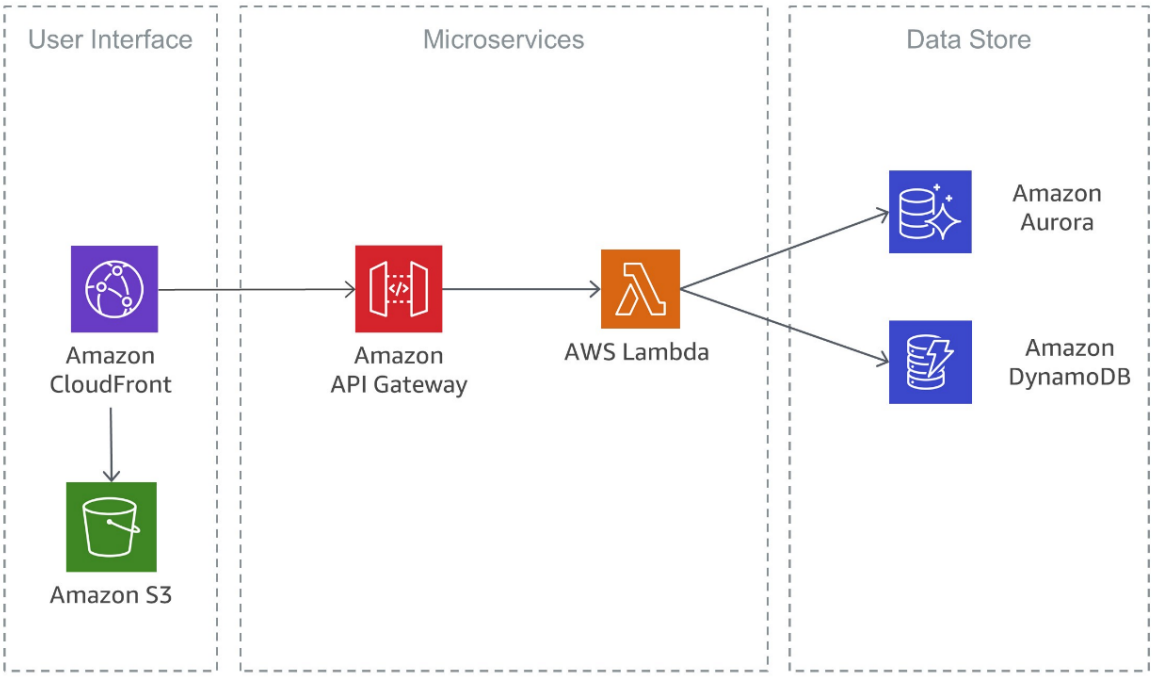

Figure 3.4: AWS serverless microservices architecture [9]

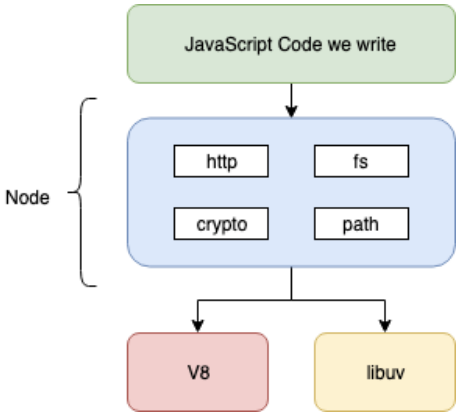

Figure 3.5: NodeJS and V8 architecture [10]

Node.js allows developers to use JavaScript for writing code executed on the server side, for example for the production of the content of dynamic web pages before the page is sent to the user's browser. In this way, Node.js allows to implement the so-called "JavaScript everywhere" paradigm, by unifying the development of Web applications around a single programming language (JavaScript).

NodeJS was chosen as the programming language for this web application due to its peculiarity of using Javascript both on the backend and frontend side. Furthermore, given the great success it has had from the developer community, it is widely supported at multiple levels of server architectures.

In this way, even if for the purpose of this project the server can simply run inside a PC, the

code at the base can be deployed on more scalable infrastructures without having to rewrite almost nothing new at the business logic level.

3.1.1 http module

NodeJS exposes out-of-the-box a solution for creating web servers through the http module. As can be deduced from the name, the module deals with the management of the http protocol.
In particular, to create a web server, a createListener function is exposed which takes as input a handler function and a port number to be used as a listening port within the local network. The handler function acts as a callback for each http call made to the server.
This function receives two parameters, the first one is a request object, and the second one a response object. The first one contains all the information deriving from the incoming http package, such as the URL, the method, the payload, the header, etc. The second one allows you to set everything necessary to answer the http call, that is the status code, the payload and the response header.

NodeJS also provides a second module, called url, for the management and parsing of the URL of the http request. This module allows you to quickly access the domain, the relative path, the query string, etc.

3.1.2 Frameworks

However building a RESTFul server from these simple basic modules would be too slow and tedious, which is why several frameworks have been developed over the years that use the basic http and url modules to provide an organized development environment and efficient.

Most of these frameworks also provide a series of plugins and features out-of-the-box to quickly manage validation, access to the DB, organization of individual endpoints, etc.

In addition, the most popular frameworks can count on a community of thousands of developers who continuously make corrections and improvements, as well as developing a series of plugins and integrations from third-party libraries from scratch.

3.2 HapiJS

HapiJS is a web framework based on NodeJS. It was chosen for its excellent performance in terms of security, speed in handling HTTP requests, its integrated authentication and authorization system, and for its rich ecosystem of open source libraries and plugins created by the NodeJS community. At the time of choosing this framework, the latest stable version was 17.8.1.

The latest version was also chosen for its support to the asynchronous management system of Javascript based on async / await.

3.2.1 HapiJS plugins

Thanks to the use of a framework such as HapiJS, it was possible to include some libraries that automatically deal with the management of some features necessary for the operation

Figure 3.6: HapiJS logo

of the application.

These plugins are: hapi-mongodb, to connect to the MongoDB DBMS, joi, to validate the incoming data for each HTTP request, boom, to manage HTTP error responses, intert, to manage the static files generated by the frontend project and distributed from the backend, catbox-memory , for managing top-level cache memory.

There are also other third-party libraries used by the project, which have not been developed specifically for HapiJS, but which being part of the NodeJS ecosystem are perfectly compatible with HapiJS itself.
These plugins (which we will see later in detail later) are redis, for the connection with the caching service of the same name, bcrypt, for the encryption of access passwords based on the Blowfish algorithm and moment for advanced time and dates management.

3.3 Server.js

The entry point of the HapiJS-based web application is the server.js file.
Within this file the server object is created by the hapi.server (options) function.
The options passed to the server determine the execution environment of the server. In order to have flexibility and scalability, an environment-based configuration system was therefore created.
For each desired environment it is possible to create a different configuration file. The configurations of the environments are specified inside the config folder. The config folder contains a main index.js file, and then as many files as there are different environments.
For local development, for example, a local.js file was created, containing a simple object exported to the outside.
Using environment variable, it is possible to specify to the server.js file which environment "to load".

Once loaded, the configuration file is passed both to the creation function of the server itself, as well as to all the plugins associated with it.
Immediately after creating the server, in fact, it is necessary to perform at least three fundamental operations on it:

1. register the HapiJS plugins
2. define an authentication strategy
3. define routes

3.3.1 Custom plugins

As we have already seen, some plugins are directly available from the community and easily installable and configurable.
This is the case, for example, of the MongoDB plugin.

This plugin provides for its registration a configuration object with host data and authentication to the DB server.
Since this data may vary depending on the server's execution environment (locally during development, or on a virtual machine during the internship, or on a cloud computing platform in production, etc.), this data is part of of the configuration file we mentioned earlier.
The object used locally to connect to MongoDB is the following:

```
mongo: {
    url: 'mongodb://localhost:27017/rtm',
    settings: {
      native_parser: false,
      useNewUrlParser: true
    }
  }
```

Localhost because the DB resides on the same execution machine as the local server, 27017 is the default port used by MongoDB, and rtm is the name chosen for the database of this web application, which stands for Radiation Test Manager.

Other third-party libraries, on the other hand, require an ad hoc "wrapper" to be easily used within the life cycle of an HTTP request based on HapiJS.

Not having found any suitable plugins for the case, a "plugins" folder was created within the backend project, inside which a subfolder was created for each plugin needed.
The plugins created are the following: redis-client, response-handler and user-auth.

To be compliant with HapiJS, a Javascript object registered as a plugin must have a "plugin" key whose value must be an object with an asynchronous register function, a string name necessary to call the plugin from other plugins or within the routes, and an optional string version.
The register function includes all the business logic for initializing and using the plugn. For example, within the redis-client plugin, the register function takes care of establishing a connection to the redis server and exposing the necessary functions directly connected to redis itself.

Next we will see how to access these features within the handlers of each endpoint or route.

3.3.2 Authentication plugins

There are special plugins that expose useful features to the authentication process.
This is the case with the user-auth plugin. Within this plugin it is possible to add authentication schemes, in this way:

```
server.auth.scheme('user', implementation);
```

Where "user" is the name of the schema, and implementation is a function that takes the server object as input, and returns an object that must contain at least the authenticate key inside it, whose value is an asynchronous function that is called during life cycle of an HTTP request that needs that specific authentication. This function, when invoked by HapiJS for those routes that require it, takes the object of the request (indicated with req) and the handler object of the response (called h), and according to the internal business logic, it can throw an exception or accept the request as authenticated.

3.3.3 Routes

After configuring the various plugins, the HapiJS server object must be populated with routes.
Each route corresponds to a unique URL / method pair.
A route object in its simplest form looks like this:

```
{
    method: 'GET',
    path: '/ projects',
    config: {
      handler: handlerCallback
    }
  }
```

Where method is the HTTP method of the request that must be handled, path is all that appears in the URL after the domain, for example locally the URL

```
http://localhost:3000/projects
```

is divided into:

1. protocol: HTTP
2. domain: localhost
3. port: 3000
4. path: /projects

The object config can contain several configuration keys, the main one being the handler which accepts an asynchronous function.

This asynchronous function is the one that takes care of managing the request, taking the request and response objects at the input. To add a route object or an array of them to the server, simply invoke the function:

```
server.route (route);
```

Where route is a route object in the form just described, or an array of them.

3.4 MongoDB

The DBMS chosen as the database for the web application is MongoDB. The main characteristic of MongoDB is that it is a non-relational DBMS.

In contrast to SQL-based languages, such as MySQL or MSSQL, the tables of a database in MongoDB, called collections, do not have a fixed schema, but are simple JSON objects that can vary totally from a tuple (called a document) to the other and it is possible to change the data structure over time.
Due to this peculiarity, the use of MongoDB is indicated above all for applications in which there are few or no relationships between the various collections, therefore where it is not necessary to join them.
Furthermore, it turns out to be much faster in writing than in reading.

The main feature of MongoDB, however, which has determined its choice for this project, is that it accepts Javascript objects as queries.
In contrast to SQL-based languages, in which the queries are declarative and human-readable strings, the queries on MongoDB are simple functions that accept Javascript objects as inputs, whose keys and values determine the operations to be performed, the filters to be applied. , etc.

Furthermore MongoDB is originally conceived as a distributed database; high availability, horizontal scalability and geographic distribution are therefore native and easy to use.
Thanks to these features 3.7 (in addition to being free), therefore, it lends itself optimally both from a code development point of view and from a point of view of future scalability.

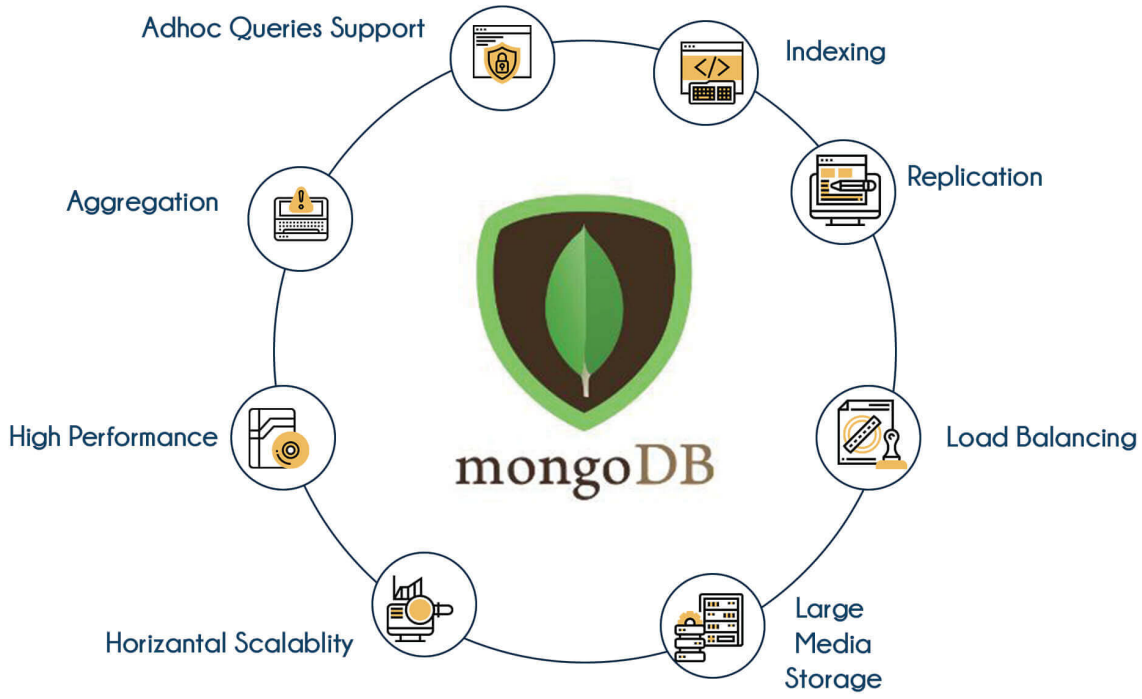

Figure 3.7: MongoDB features [11]

3.4.1 Data structure

The DB data structure has been divided into three collections, users, projects and facilities. The users collection contains users' access data.
Each document of a user has 3 keys which correspond to 3 values, the unique ID within the collection, called `_id`, an email and a login password.

The facilities collection instead collects all the facilities entered in the system by users. Each facility has a unique `_id`, a userId field with the user `_id` corresponding to who created the facility, a name, an address, and an ion array.
In fact, since a document on MongoDB is a JSON-like object, it is possible to insert another object or an array as a key value, and to create substructures within the same document. The ions array is therefore a set of ion objects, each of which has a name and the reference technical values, i.e. mq, dut, range and letEnergy.

The third collection is that of the projects. A project document is the object that takes up the most space on the DB and is also the most complex, in fact it has the following keys: a unique `_id`, the userId of the user who created it, a name, the facilityId of the reference facility among those created by the user, an array of strings called tags, a copy of the reference ion, a numeric value of bits, and a run array. Each run in particular is made up of a tag string, and two numbers, errors and particles.
Furthermore, the ion and facility selected for the project are copied entirely by cloning the original data from the facilities collection, this is because in order to make the most of

the MongoDB features, the classic rules on data normalization are no longer available, and sometimes it is better to make copies of objects and values, rather than referencing them through `_id`.
This is primarily due to the fact that not being performing on joins, having foreign keys on which to apply the joins does not make much sense.

Furthermore, even by modifying the initial facility, for example by removing an ion that is no longer supported after months or years of operation, there is no risk of eliminating the reference on an old project, which being able to have its local copy in its document, is protected from these updates.

3.5 REST APIs

HapiJS routes are used to implement the RESTFul web server. The 6 main resources created are:

1. authentications
2. facilities
3. file-systems
4. projects
5. storage
6. users

Each of these resources implements one more HTTP methods, for each of which corresponds one of the 4 CRUD operations (Create, Read, Update, Delete).

All REST routes are inside the controllers folder, which contains a subfolder for each main resource.
Each macro-folder-resource in turn contains a subfolder for each implemented method, or an additional subfolder for a new "piece" of path.
Each folder exports the array of the routes of its competence upwards, up to the controllers folder, which exports the merge of all the micro-arrays of each resource into a single array.
In this way, only this single array must be imported from the server.js file and all routes will be automatically uploaded to the HapiJS web server.

Before going into the details of each individual resource, together with its routes and methods, let's see in detail how the subject of the request (req) is populated and how the response object (h) is managed, which represent the parameters received by the handler functions of each route object.

3.5.1 Request & response objects (req h)

As we have seen, each object in a route takes an asynchronous function as the parameter of the config.handler key.
This asynchronous function is expressed like this:

```
async (req, h) => {};
```

Before using the async / await pattern, HapiJS relied on using the response object (h, previously called res) to end the execution of an HTTP call, using the re response object as a callback.
From the moment the async / await pattern was used, the standard is to return the return of the handler function as the body of the response.

The use of the h object is therefore limited to specific cases in which, for example, it is necessary to set variables in the header, or to return a file instead of a simple JSON object.
The request (req) object instead remained almost the same between the various versions, and does not differ much even from the native version exposed by the http module of NodeJS.
The main keys used are in fact: req.payload, containing the body of the request, converted into a Javascript object, req.query, which contains the query parameters passed via URL, and req.params, which contains the parameters of a parametric route.

To better explain this concept it is necessary to give an example. Take for example the HapiJS route defined as:

```
{
    method: 'PUT',
    path: '/projects/{_ id}/fileSystem',
    config: {
      handler: update
    }
  };
```

The {_id} parameter is a special syntax which is useful for HapiJS to parse that piece of path as a parameter.
Any call that respects that pattern, with the PUT method, will therefore be managed by that route.

For example `/projects/1/fileSystem`, `/projects/test/fileSystem` are two valid paths that will be managed by the same parametric route defined above.
The string inserted between the curly brackets of the route path definition specifies the key of the req.params object.
The value actually called by the client, in place of that string, will represent the corresponding value. In the first case, the req.params object will contain {_id: 1}, in the second {_id: 'test'}.

3.5.2 Request validation

One of the most important concepts in the development of a web server, especially when it comes to REST APIs, is the validation of the input data. The vast majority of online hacker attacks exploit bad or non-existent server-side validation rules.

One of the most famous and classic attacks is for example SQL injection (3.8).
This type of attack exploits those web servers based on SQL technologies, such as MySQL, which use client-side forms to filter data and show them for example in an HTML table.

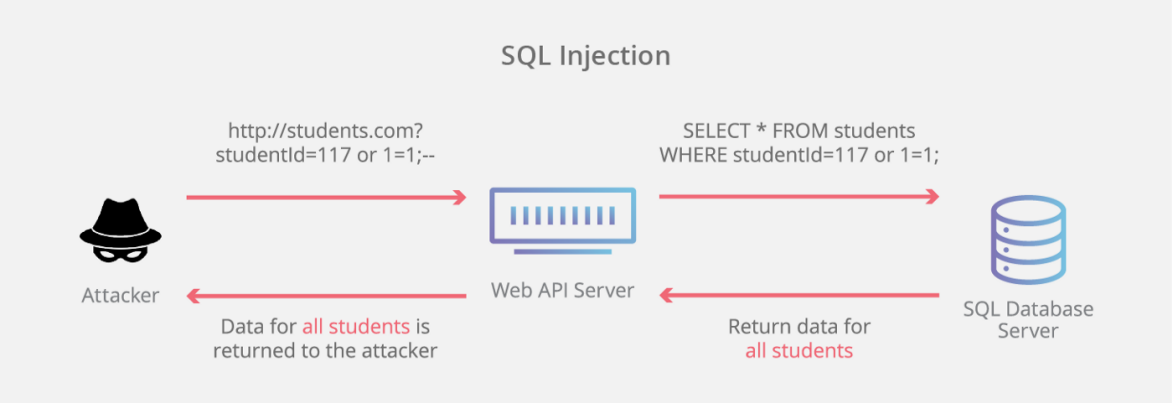

Figure 3.8: SQL injection example [12]

A very common problem especially in the 90s and 2000s, was to accept string filters directly on the client side and pass them to MySQL without any validation.
A malicious client, instead of sending a simple string that acts as a filter, for example in a WHERE clause, could send an adequately formatted string to perform malicious actions, such as for example deleting data from the DB, or even in the worst cases being returned whole tables with sensitive data as the return value of that query.

To avoid this like other similar attacks, it is essential to start from the concept that the client is not only the frontend part that we prepare for our backend, but it could be any tool, a bot as well as a hacker equipped with a command line or more advanced tools.
The validation of the input data is therefore a very important factor for the success of an efficient and secure backend.

HapiJS natively provides a library and a standard system for data validation.
All this is done at the route object level, specifying a validate object inside the config object (which as we have seen previously also contained the asynchronous function of the request handler).
This validate object can be populated with the payload, params and query keys to validate respectively the request body, the parameters in the URL path and the query string object always derived from the URL.
Each of these keys accepts a Joi object as a value.

Joi is the native library that is part of the HapiJS ecosystem for data validation.

Figure 3.9: Joi logo

Thanks to joi it is possible to describe validation rules through a clear and easily human-readable syntax. Starting from the base Joi object exported by the require module ('joi'), it is possible to access properties that allow you to quickly define data type and restrictions. For example, let's say we have a path /users with POST method for creating a new user. If we wanted to limit and validate the body sent by a client, we could build our route in this way:

```
{
  method: 'POST',
  path: '/ users',
  config: {
    handler: createUser,
    validate: {
      payload: joi.object ({
        name: joi.string().min(3).max(20),
        email: joi.string().email(),
        password: joi.string().min(8).max(100),
        age: joi.number().integer().min(18),
      })
    },
  }
}
```

In this way we are telling HapiJS to accept a Javascript object, containing a name, an email, an age and a password, as a payload.
n addition, we specify both the type and some limits for each variable. For example the name can be from 3 to 20 characters long, the age must be at least 18 and must be an integer, and the email must be a valid email.

Thanks to Joi, therefore, we don't have to worry about filling our code with if conditions on each input variable, but we can easily build a validation object that takes into account all the specifications.

When a request arrives at the web server, after applying the appropriate routing rules, to decide which route / method pair should manage it, HapiJS parses all parameters (payload, params and queries) and validates them thanks to Joi and validation objects specified.
If even one of these objects does not pass the validation rules, the request is not passed to the handler function, but a Bad Request 400 status code is returned directly from HapiJS automatically.

In some cases, however, it is also necessary to have control over these cases of failure, for example to manage the log of failed requests for validation rules.
To achieve this, HapiJS also offers another parameter within the validate object of the route, which is the so-called failAction.

For example, for the development of this application, an ad hoc failAction has been implemented for log reasons on the use of the platform.
The failAction function receives the request and response objects req and h as parameters, in

addition it also receives an additional error object containing details on the validation error triggered by HapiJS or Joi specifically.

Furthermore, through the binding function of Javascript, an additional api parameter was passed which specifies in addition to these request parameters, also in which API the error occurred.
This ad hoc api parameter is a constant defined in the project useful at the log level to uniquely identify a route on the DB that has generated a response or an error.

The failAction function therefore receives all the information on the HTTP request received from the web server, including the error, and calls an ad hoc function created to manage all the backend responses.

3.5.3 Response handler

This ad hoc function is actually exported by a custom plugin also called within each single handler of each route, called the response-handler.

This custom plugin mainly deals with two things, the first is to log all requests made on the platform, in order to create statistics, useful information for debugging in case of errors, big data, etc.
The second is to take care of generating an adequate response to the client, based on its request.
In most cases this means simply sending a response JSON generated by the handler function, but in other cases, such as during execution of the failAction, it can mean generating an ad hoc JSON containing the error code.

Two types of error code have been generated on the backend side, an internal one, for log and statistics purposes, and a client one, to give a precise indication to the client (intended both as software for executing HTTP calls, both as user and uses it) what to show as an error message, or how to behave automatically when, for example, a 401 error of non-authorization is received, in the case of authentication expired, for example.
Moreover, having implemented the backend not only as a RESTful server, but also as a repository of static files for the scientist's documentation part, the handler function also manages requests that try to access these files, returning in these cases not a JSON file, but the static resource read by the file system. We will see in detail how this happens in the chapter dedicated to storage and the inert plugin of HapiJS.

3.6 Authentication

Authentication was managed through a custom authentication plugin, created within the project.
The plugin was then called in the server.js file in this way:

```
await server.register ({
  plugin: hapiUserAuth,
  options: {
    host: opts.redisUrl [0] .host,
    port: opts.redisUrl [0] .port,
```

```
      param: 'rtm-auth-key',
      decoratorName: 'user',
      cookie: 'rtm-auth-token',
      secure: opts.env! == 'LOCAL'
    }
  })
  server.auth.strategy ('session', 'user')
```

In particular, server.register is the function of Hapi to register a new plugin on the server. It takes as an parameter an object with plugin values, which is the module of the plugin usually imported from an ad hoc file containing the code of the plugin itself, and an options object that varies from plugin to plugin, and which in practice is validated through a schema Joi inside the plugin itself to make the developer pass the correct parameters to the plugin, in the correct format.
In this case, our authentication plugin takes the host and port of a redis instance as input, which acts as a first level cache with session data, a param string used as the name of the db inside redis itself, a decoratorName string that indicates with what name to decorate the request with the user object that contains the information of the logged in user, the name of the cookie to be set on the client side containing the session key, and a secure flag used to understand if the connection is under SSL or not.
Once you have instanced an authentication plugin, you need to associate it with a strategy. In fact, the same plugin with the same system could be used several times with different strategies.
For example, let's say we have two different types of users, normal users and admin, we could use the same cookie-based authentication, and session on redis, but with two different strategies and application uses, one for the routes to be protected at the user level, and one for routes accessible only by administrators.
In our case having only one type of user, we define a single strategy called "session", which will use the authentication plugin called "user".
This "user" string appears as an export inside the plugin exactly at this point:

```
server.auth.scheme ('user', implementation)
```

Where "user" is precisely the string with which we will call the plugin to define a strategy, and implementation is the authentication implementation function defined above in the same file.

From a routes point of view, authentication was managed through two endpoints on the authentications virtual resource.
The resource is called virtual because it does not correspond to a model, or a collection, on the DBMS, but is a useful abstraction for falling within the parameters of the REST paradigm.

The first method is POST, and the second DELETE. The first corresponds to the login phase, the second the logout phase.

3.6.1 Login

The login, which is the POST method on the route / authentications, receives a credentials object as a payload.
This object in turn contains an email and a password, both validated through Joi as strings of maximum 50 characters and required.
The email in particular must necessarily be a valid email, and therefore also has that validation parameter.

In our Hapi route object, inside the config object, which as we have seen contains both the handler function and the validation object, we add a third auth object.

```
auth: {
  strategies: ['session'],
  mode: 'try'
}
```

This object tells Hapi that the route is protected by authentication.
In particular, an array of strategies is specified to be considered, in our case the only strategy called "session" in the server.js file.
In addition, the authentication mode is also specified, in this special case "try".
This mode indicates to HapiJS, as well as to the underlying authentication plugin, that that route may not yet be protected by authentication, but that it may try to authenticate itself at that moment.
It is clear that this mode is the one to be used during login.

When the login route handler is invoked, it receives the req request object as a parameter. This contains the payload, already validated by HapiJS using the corresponding Joi object declared in the route.
The payload in turn contains credentials with email and password.
So the email is used as a key to search for the corresponding user on Mongo.

If the user is not found, a WRONG_CREDENTIALS error is triggered for the client, an internal error EMAIL_NOT_FOUND on the server side.
If the user is found, the password is compared with the one sent by the client, using the bcrypt library algorithm.
If the password does not match, a WRONG_CREDENTIALS error is generated for the client, and WRONG_PASSWORD on the server side.
We do not give indications to the client if it is the email or the password that is wrong, because this could cause a security problem, such as allowing malicious users to check whether a certain email is present on the system or not.

If email and password match, the login function is invoked on the req.user object, passing the identification and unique id of the user on Mongo.
The req.user object is the result of decorating the authentication plugin on the request object. The literal is really "user" because this is the string that we passed as an option to the plugin to the "decoratorName" key. This is the code that takes care of the decoration inside the implementation function.

```
const decoration = req => new internals.CookieAuth (req, settings)
```

```
  const isDecorated = server.decorations.request.indexOf (settings.decoratorName) >= 0
  if (! isDecorated) {
    server.decorate ('request', settings.decoratorName, decoration, {
      apply: true
    })
 }
```

In this case, settings corresponds to the options object passed to the authentication plugin in the server.js file.
The server.decorate function is responsible for decorating the req object with the user object. The latter is an instance of the class internal to the CookieAuth plugin.
It is the CookieAuth class that implements the login function called in the route handler, and its goal is to generate a new session on Redis with the information of the connected user, receive a unique session key and set it as a cookie on the client via the h.state function on the response object h.

```
h.state (settings.cookie, key, {
  isSecure: settings.secure,
  TTL: 30 * 24 * 60 * 60 * 1000,
  isSameSite: false
})
```

Where settings.cookie is the name of the cookie we passed as a parameter of the authentication plugin options in the server.js file, and key is the unique key on Redis that will be used to authenticate the client to subsequent requests.

Finally, the response is populated with the user object read from the DB and returned to the client.
Through the cookie set via h.state, the client (in this case the browser) will save the cookie in its session and send it to subsequent requests, managing to be identified.

3.6.2 Authentication check

Routes that provide authentication will take the following form:

```
{
  method: 'GET',
  path: '/users/self',
  config: {
    handler: get,
    tags: ['api'],
    auth: {
      strategies: ['session'],
      mode: 'required'
    }
  }
}
```

The auth field indicates to Hapi that the route must be validated before it can execute the respective handler and generate a response.

At each request, the authenticate function implemented in the authentication plugin is therefore invoked.
This function is as follows:

```
const authenticate = async (req, h) => {
  const key = req.state[settings.cookie]
  let session, err
  try {
    if (!key) {
      throw new Error('No session found')
    }
    try {
      session = await getUserAuth(key)
    } catch (err) {
      if (err.code === 'WRONGTYPE') {
        await deleteUserAuth(key)
      } else {
        throw new ApplicationError(err, internalErrorCodes.REDIS)
      }
    }
    if (!session) {
      throw new Error('Session expired')
    }

    const { validationError } = joi.validate(session, internals.sessionSchema)
    if (validationError) {
      await deleteUserAuth(key)
      throw validationError
    }
  } catch (_err) {
    err = _err
  } finally {
    if (err) {
      throw boom.unauthorized(err)
    }
    return h.authenticated({ credentials: session, artifacts: session })
  }
}
```

The function takes as input the same variables req and h that arrive at the handler function.
The authentication cookie is extracted from the request, in the key variable.
If the cookie is not set, a first type of error is generated for no session found.
If the cookie is set, look for the key on Redis, through the getUserAuth function.

If the session is null, it means that the session key is invalid or expired, therefore a second type of error is generated.
Finally, it is checked whether the session on Redis is in the correct format or not, or if it has been corrupted incorrectly or accidentally, and if a third type of error is triggered.
In each of these cases, an unauthorized error is triggered via Boom and Hapi automatically returns a 401 error to the client, without going through the endpoint management handler.
Otherwise, the authenticated function is called on object h, passing the session object containing the ID of the logged-in user read by Redis.
This function takes care of decorating the request with the data of the logged in user, so that they are available in the endpoint handler.

3.6.3 Logout

The logout route works like this, first the auth object inside config of the route object has this form:

```
auth: {
  strategies: ['session'],
  mode: 'required'
},
```

which indicates that in this case the session must already be active, that is required.
In fact, a user not logged in does not have permission to log out.
Inside the handler we use the ID of the connected user, readable by req.auth.credentials.userId decorated by the authentication plugin, to search for the user on Mongo.

If the user is valid, the logout function is therefore called from the req.user object, which as we have seen previously is an instance of the CookieAuth class defined in the authentication plugin.
In the logout function, the key is read from the client's cookies:

```
const key = req.state [settings.cookie]
```

and is used to delete the session from Redis:

```
await deleteUserAuth (key)
```

Furthermore, through the Hapi h.unstate function, the client is suggested to delete the cookie from their browser session:

```
h.unstate (settings.cookie)
```

This eliminates all session information and the user no longer has access to the private APIs.

3.7 Storage

When a project is created, a fileSystem object is generated within the corresponding document on MongoDB.
This object contains two arrays, folders and files. Folders is an array of objects containing

the keys name, folders and files, where name is the name of the folder, and folders and files are two new arrays identical to the previous ones.
With this structure, therefore, a virtual file system has been created within MongoDB, since each folder can recursively have a subset of other folders.
The arrays of type files instead contain arrays of objects having the keys name and field, where name is the name of the file, and field is a unique id on the ObjectID format of MongoDB which corresponds to the name of the physical file on the hard disk.

In this way the virtual and physical file systems have separate structures and are not bound to each other, but are linked by the simple unique ID.

3.7.1 Uploading new files

To manage the virtual file system, 3 routes were mainly created. The first route is

`/projects/_id/fileSystem`

with POST method. This route is used to create a new file on the virtual file system. The input data are the `project_id`, at the parameter level in the path, the content of the file, contained in the body of the request, and the path inside the virtual file system inside which the file must be inserted.
This is an extract of the route in question:

```
{
  method: 'POST',
  path: '/ projects/{_id}/fileSystem',
  config: {
    auth: {
      strategies: ['session'],
      mode: 'required'
    },
    payload: {
      maxBytes: 100000000,
      output: 'stream',
      parse: true
    },
    validate: {
      query: FileSystemValidators.uploadFile,
      failAction: failAction.bind (this, apis.FILE_SYSTEMS.CREATE)
    },
    handler: uploadFile
  }
}
```

We note first of all how it is necessary to be logged in and authenticated in the system, specifying the session strategy as required.
The payload parameter instead indicates to Hapi that the body of the request must be

interpreted as binary content, that is a file, and must be sent to the handler via a stream, which is a special javascript object to which you can "connect" to read data that arrive precisely in a stream and are not immediately available.

We also give as a configuration parameter a maximum number of Bytes that the route can receive, before triggering a MAX_LENGTH error to the client. Finally, the query string is validated through a Joi object contained in the FileSystemValidators model

```
The object in question has this form:
static get uploadFile() {
  return joi.object().keys ({
    path: joi
      .string()
      .required()
  })
}
```

that indicates to Hapi that the query object must contain a string called path.

Within the handler function, we first start saving the incoming byte stream, inside a file inside the storage folder on our hard disk.

```
try {
  fileId = new ObjectID()
  await new Promise((resolve, reject) => {
    const wstream = fs.createWriteStream (
      `${__ dirname}/../../../storage/${fileId}`
    )
    req.payload.file.on ('data', wstream.write.bind (wstream))
    req.payload.file.on('end', wstream.end.bind (wstream))
    req.payload.file.on ('error', reject)
    wstream.on ('finish', resolve)
  })
} catch (err) {
  throw new ApplicationError(err, internalErrorCodes.FILE_UPLOAD)
}
```

Once a new unique ObjectID has been created and assigned to the fileId variable, the read stream file available on req.payload.file is read through event callbacks to which it is possible to subscribe.

Since writing files via the createWriteStream function also takes place via streams, the process is straightforward and easy to read.
The writing of the stream further is managed through Promise, linking the error and finish events to the respective error callback reject and successful callback resolve.
At this point the controls on the connected user and on the projectId indicated as path parameter inside the url follow.
If the data are correct, the project file system is read by MongoDB and updated with the insertion of the new file.

```
const fileSystem = new FileSystem(project.fileSystem)
try {
  fileSystem.addFile(req.payload.file.hapi.filename, fileId, req.query.path)
  await fileSystem.save(projectId, db)
} catch (err) {
  throw new ApplicationError(err, internalErrorCodes.MONGO)
}
```

FileSystem is the model / class created for the management of the fileSystem object within the project documents of the projects collection on Mongodb.

```
The addFile function in particular performs these operations:
addFile (name, fileId, path) {
  const folders = path.split ('/')
  let folder = this
  for (let f in folders) {
    if (folders [f]! == '') {
      folder = folder.folders.find(f1 => f1.name === folders [f])
    }
  }
  folder.files.push({
    name,
    fileId
  })
}
```

First, the virtual path is split into the path folders.
Through this array of folder names, the folder tree of the virtual file system is searched, looking for the next folder at each step by its name.
Once the destination folder is reached, the new file is pushed into the files of that folder specifying the virtual name and its corresponding ID in physical memory.

3.7.2 Modifying the virtual file system

The second API for managing the virtual file system is on the route:

```
/projects/_{id}/filesystem
```

using the HTTP method PUT.
This route receives as parameters the unique ID of the project through path parameters, and the new format of the file system through payload.

```
{
  method: 'PUT',
  path: '/projects/{_ id}/fileSystem',
  config: {
    handler: update,
```

```
    auth: {
      strategies: ['session'],
      mode: 'required'
    },
    validate: {
      payload: FileSystemValidators.update,
      failAction: failAction.bind (this, apis.FILE_SYSTEMS.CREATE)
    }
  }
}
```

This route also requires user authentication, but this time it is the payload that is validated through this Joi object:

```
static get update() {
  return joi.object().keys({
    fileSystem: joi
      .Object()
      .keys ({
        folders: joi
          .array()
          .required()
        files: joi
          .array()
          .required()
      })
      .required()
  })
}
```

In this case we tell Hapi that our payload is an object containing two arrays, folders and files. The handler function of this endpoint therefore verifies the validity of the connected user, checks that the project indicated by the ID specified in the path is owned by that user, and only in this case modifies the virtual file system object with the version client side receipt.

3.7.3 Accessing the storage

Access to the files is made through another endpoint which acts as an intermediary to verify the actual ownership of the file by the user requesting it. This route has the following path

```
/Storage/{ProjectID}/{filePath}/{fileName}
```

accessible via GET method. The route has this form:

```
{
  method: 'GET',
  path: '/storage/{projectId}/{filePath}/{fileName}',
```

```
  config: {
    handler: get,
    auth: {
      strategies: ['session'],
      mode: 'required'
    },
    files: {
      relativeTo: path.join(__ dirname, '../', '../', 'storage')
    }
  }
}
```

The peculiarity of this route lies in the file parameter that we have not yet encountered. This object takes a relativeTo string, which tells Hapi the root to consider when returning a file to the client.

In this case, a storage folder has been created on the project root, in order to contain all the files uploaded by users, in the ways seen in the previous chapters.
To be totally stateless, the application should be based on a file management system not on the local file system, but for example on an AWS S3 type external service.
But for the purpose of this project, this solution was considered a good trade off.

This file parameter therefore replaces the one defined in the server.js file, through which we had indicated to Hapi to use the client folder for the management of static files.
In fact, we have two different file sources, on the one hand we want our client (intended as a browser) to receive static files to run the web application, on the other we want our user to receive his personal files when requested, that's why we use this approach.

The handler function that manages the return of the static file checks the validity of the logged in user, then checks that the project indicated by the ID projectId passed through path parameters is owned by the user himself.

At this point the parameters filePath and fileName are passed to the response handler, the plugin that we have already seen in the previous chapters which has the task of dealing with the response to the client.
Also in this case it is the plugin to take care of returning a static file, contrary to what was previously seen in which a simple JSON object had returned.
This is done through the response handler h, which deals with managing responses that are not simple JSON objects.
In fact, the following control takes place within the custom response-handler plugin:

```
} else if (filePath) {
  return h.file (filePath, {
    fileName
  })
} else {
```

that is, in case the filePath variable is present, and this happens only for the above GET route, the response must be managed in this specific way, that is through h.file.

This function works thanks to the presence of the inert plugin, downloaded via NPM, which allows us to return a static file simply by passing its path.

The fileName variable also indicates to Hapi and the inert plugin the name of the file to be used, in fact on the hard disk the file is still named by the unique ID of MongoDB, but we want the user to receive it with the name that he himself specified at creation time.

CHAPTER 4

Frontend development

By the term frontend we mean the set of static resources, including scripting code, which is used, rendered, and executed on the client side.
The frontend can therefore be represented by a desktop application installed locally, which communicates with a central server, such as Spotify, or by any web and mobile application.

The development on the frontend side therefore regards various aspects, from the realization of the graphic interface, to the communication with the server, to the management of the internal state and the cache, up to the business logic performed on the client side.

The use of Javascript as a programming code both on the backend and frontend side, offered the possibility to share pieces of code between client and server, and to decide to move part of the processing from the server to the client, to exploit the computational power of the user, and not of the service provider, or from the client to the server, to protect and obscure the algorithm.
The development of the frontend performed for this project was based solely on the web side, and the React framework was used.

Before the advent of libraries and frameworks such as React and Angular, web-side development was based on the development of HTML code, combined with one or more CSS style sheets, combined with one more Javascript script, sometimes inserted directly into HTML, to give dynamism to the web page.

Today, almost all web applications are based on Javascript libraries that offer a complete development environment, integrating the layout declaration, the business logic, and the design efficiently and out-of-the-box.

4.1 React

The React or ReactJS library was chosen for the development of the frontend.
React is an open source project born from Facebook, which has had many follow-ups from the frontend developer community.
Unlike its direct competitors, which are considered complete frameworks, React has the sole purpose of creating user interfaces.
To be honest, over the years, some tools have played a decisive role for their high effectiveness,

Figure 4.1: React logo

determining a standard in the development and deployment of the applications developed with React.
Among the many, for example, Redux has established itself as the main state manager, or webpack as the main bundler of static resources in 2 or 3 total files, as we will see later.
React is based on the component philosophy.
A component is an atomic portion of UI, which contains the declarative part of the graphic itself, the business logic, and the internal state.
Each component receives as input some parameters, called props, and returns a piece of UI as output.
The declaration of the graphic interface is made directly in Javascript, or better in JSX, which we will see later in detail.
The programmer therefore no longer has the task of writing HTML code, and interacting with the DOM, but uses "virtual" React components that reside in memory, in the so-called "virtual DOM", and React autonomously and efficiently performs the DOM update only when necessary, usually following an internal or external change of the component status.

4.1.1 JSX

JSX is an extension of the Javascript language to facilitate the declaration of React components. It is not a template language, but uses markup code.

Being an extension of Javascript, the JSX code must first be interpreted by a program such as Babel, which converts the markup code into primitive React declarations.

4.2 Development toolchain

For the creation of the React project, a starting boilerplate was chosen, which was then configured and customized later.

The boilerplate offers two initial scripts, in addition to a series of packages installed via npm.
The two scripts are start and build, the first is used to launch the development environment locally, the second to produce a release build that can then be deployed online.
The first operation that the start script does is to verify that the DEFAULT port on which it will be possible to connect locally via a browser to view the result is available.
This is done via the "detect-port" package.

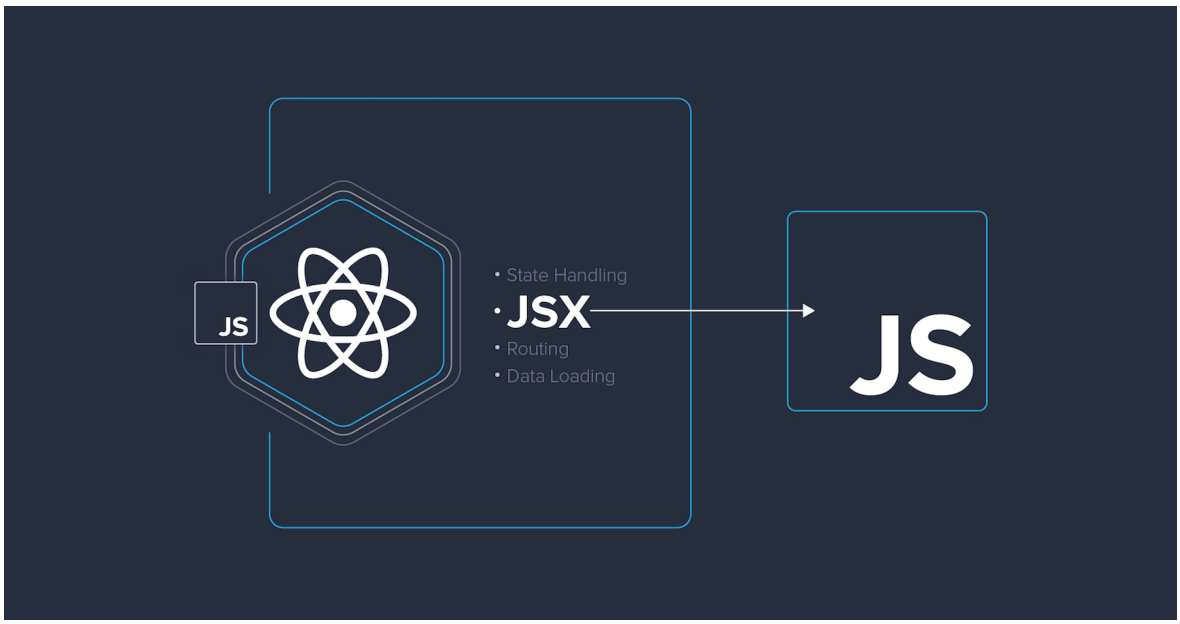

Figure 4.2: JSX role inside React [13]

```
detect(DEFAULT_PORT).then(port => {
  if (port === DEFAULT_PORT) {
    process.env.PORT = DEFAULT_PORT
    run(port)
    return
  }

  console.log(
    chalk.red('Something is already running on port ' + DEFAULT_PORT + '.')
  )
})
```

If the port is already busy, an error message is displayed, formatted in red by another npm package, "chalk".
Otherwise, the run function is called, defined within the start script itself.
The latter, is responsible for launching webpack, in charge of compiling all the sources in a single bundle, and the webpack-dev-server, in charge of creating a development server that exposes an HTML index page that incorporates all the bundles generated by webpack.

4.2.1 Webpack

Webpack is a static module bundler for Javascript applications through the use of the latest generation platform and framework, such as React, Angular, Vue, etc.

Thanks to webpack, it is possible to obtain from a set of Javascript files, both local, ie the sources, and those imported from npm, a single file called bundle, which includes all the

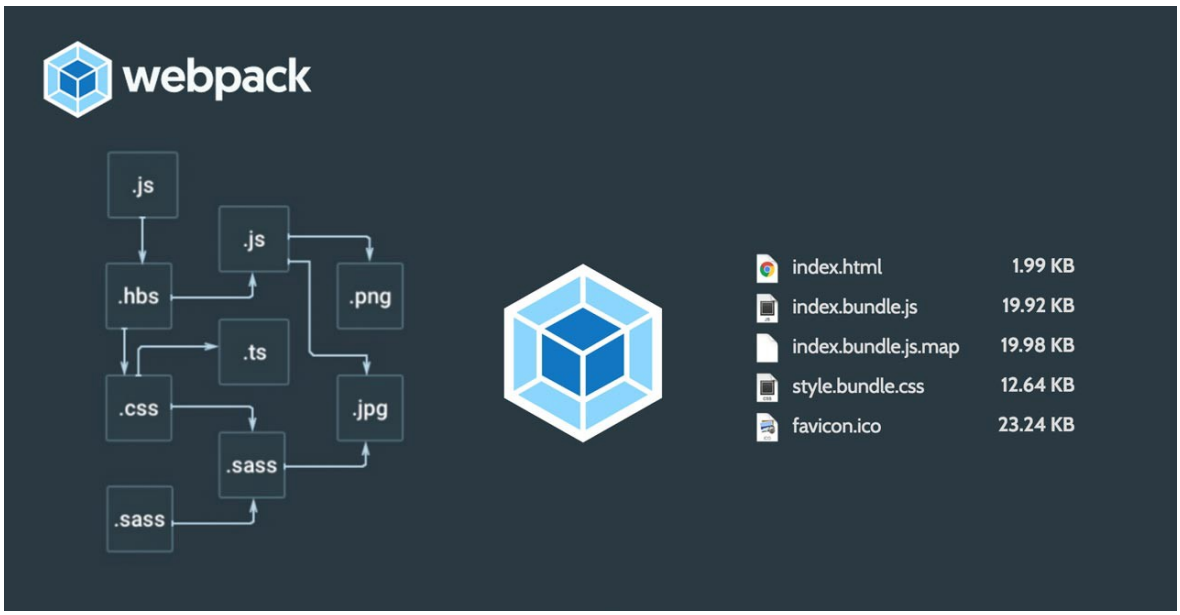

Figure 4.3: Webpack flowchart [14]

frontend business logic.

This is an amazing service, because it simplifies a lot of steps. First, it is much easier to copy and manage a single file instead of multiple files with different paths.
Furthermore, the generated file is versioned, therefore it is optimized for the browser and invalidate the cache in case of an update of the same bundle.

But not only, Webpack includes the possibility of including a series of third-party plugins in its buildchain, such as uglyfier and minifier, which respectively take care of making the JS code incomprehensible to reading, through obfuscation methods (which is the maximum which can be obtained given that the client-side code is visible by nature, and Javascript in a particular way since it is not compiled but interpreted), and to make it as small as possible, in terms of size in bytes, in order to improve latency while loading inside the browser.
And again, webpack allows you to add and use custom loaders.

A loader takes care of managing the use of other static resources, and where possible bundling them in a single file.
For example, the css-loader takes care of creating a minified and optimized bundle for all css files to avoid collisions on class naming.
Other loaders exploit the fact that some resources, such as images or videos, can be loaded via a base64 formatted string, in this way the multimedia resource is converted into a string and inserted directly into the bundle.
Only when the resource is too large or unmanaged, the file-loader takes care of dynamic loading via direct URL.
It is evident how a tool like webpack is fundamental for applications like React that aim to fully describe the part of UI and business logic using only Javascript, in which not only the code is divided into dozens if not hundreds of different files, but a lot it is often imported by node_modules.

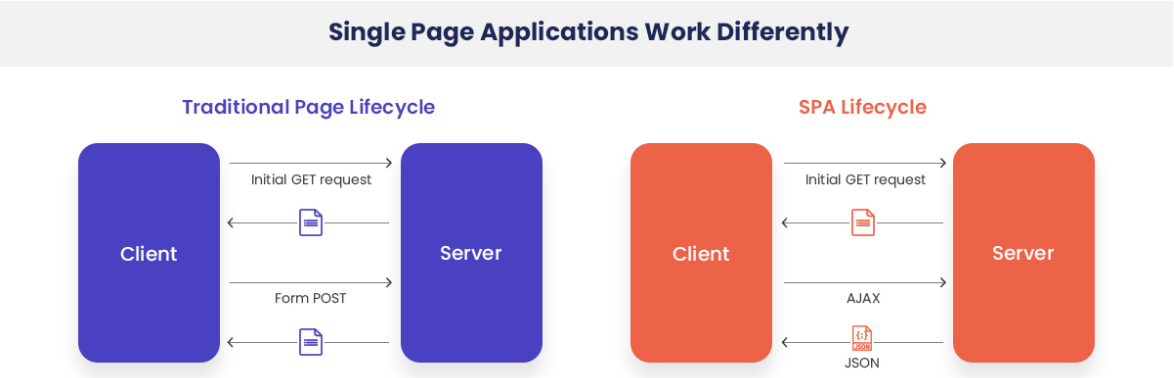

Figure 4.4: SPA and traditional page lifecycles comparison [15]

Webpack is installed via npm. It takes as input a configuration file where all possible parameters, plugins, loaders, and details on the output are specified.
For example, webpack includes a plugin that integrates the Babel preprocessor, which takes care of converting modern versions of Javascript, or JSX files, into classic Javascript that webpack and other eventual loaders can interpret without errors.

Usually two configuration files are created, one for development, and one for deployment. In fact, depending on whether it is being developed or whether the code should be available for download, there are plugins and modes that can completely differ.
For example, in development, we want the code not to be minified, indeed we want to include source mapping to find any errors and bugs. In contrast, in production we want the code to be uglyfied and minified.
Furthermore, in development we do not want the code to be written on the file system, but we want it to be available in the top-level memory, and to use features such as the hot module reloading, which allows you to see the changes we make live in the browser in the code.
In production, we want to have static files to distribute or copy on a server.
A similar discussion can be made on how and where static resources such as CSS, images, videos, fonts, etc. are to be managed.

4.2.2 Webpack Development Server

If webpack takes care of creating a bundle of resources and source code, webpack-dev-server takes care of serving this content in a local test environment, fundamental for development.

4.2.3 App.js entry point

The react project is defined as a SPA 4.4. SPA stands for Single Page Application. In fact, the whole application is included in an index.html file returned to the base path "/" by the server.
The index.html file contains some static resources, but all of the logic and CSS style are loaded dynamically from the bundle files. The project is divided into scenes and components. A scene recursively encompasses other scenes or components. A component is a reusable UI element usually not connected to the global state of the application, while a scene is an

element used once only within the application usually connected to the global state.
The entry point of the application is the app.js file, which aims to instantiate the state manager redux, and to initialize the router.

4.3 Hash Routing

Since the application is of the SPA type, a hash router was used to switch between the different screens (login, project list, etc.). This type of router works through the hash inside the url.
This parameter at the origins of HTML was used to quickly navigate the paragraphs of HTML pages.
In fact, by assigning an "id" tag such as "exampleID" to an element of the page, and navigating within the same page but with "exampleID" at the end, the browser automatically scrolls up to that element.
Since the birth of the SPA, since the hash change does not trigger a page refresh, the hash has been used to navigate the same web page between different screens, exploiting the potential of modern Javascript frameworks.
To achieve this behavior, an ad hoc React component was built with an eventListener on the "hashchange" DOM event which is triggered at each hash change.
A change of hash is triggered by clicking on an anchor link. This component receives one or more routes as children, encapsulated in a component of type Route. This component receives a hash route as a prop, which is read and interpreted by the hash router.
The latter, based on the current hash, and the changes of links on the page, decides which route to display on the screen.
This is for example the main router inside app.js:

```
<HashRouter
  routerStatePath='routing.mainRouter'
  loaderComponent={<div className={cn(sa.DF, sa.W100, sa.H100VH, sa.JCC, sa.AIC)}>
    <CircularProgress />
  </div>}
>
  <Route hash='#/sign-in' initAccess={false} routeKey='signIn'>
    <SignIn />
  </Route>
  <Route hash='#' initAccess={false} routeKey='dashboard'>
    <Dashboard />
  </Route>
</HashRouter>
```

The HashRouter component has the task of deciding on the basis of the current hash of the page in the browser which of its child components best "fits". In the example above, two sub-screens are stated. The dashboard and the login page.
The HashRouter component also receives as a prop a variable that indicates where the state information of the router itself is saved within the state manager (redux, as we will see later).

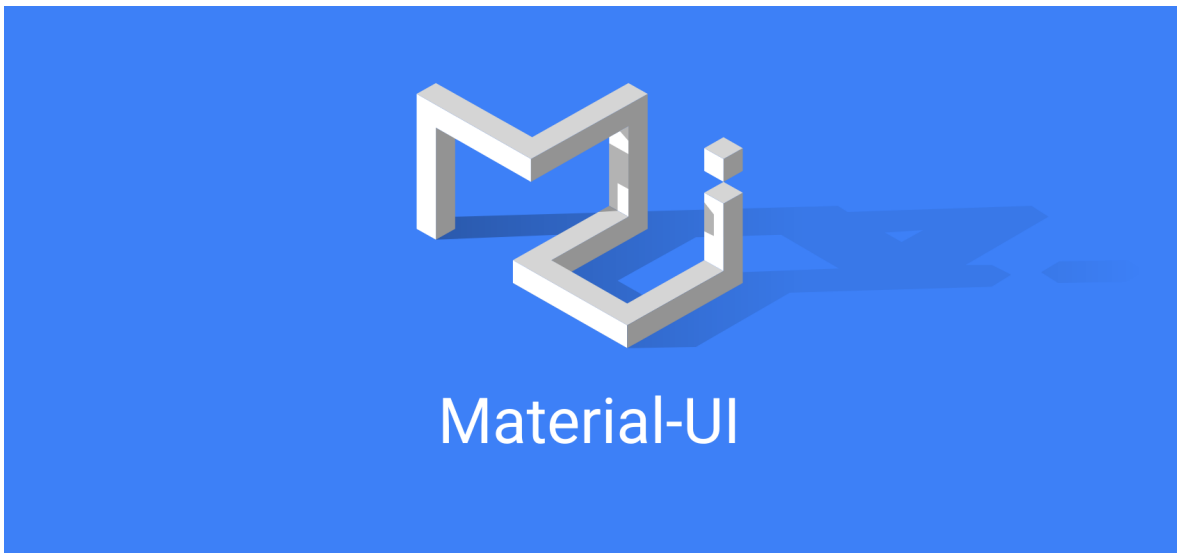

Figure 4.5: Material UI logo

And it also receives a loaderComponent, which is the component used in case none of the declared routes are available at the time.
In fact, each route has an initial access state, configured through the prop initAccess, and a dynamic access state set within the state manager, which communicates to its parent HashRouter at any time, that that screen is viewable or not, even if the hash of the page fits. This has been implemented because the user can easily modify the hash of the page by hand to reach screens that logically should not be reached. For example, if the user is not logged in, the dashboard screen is locked a priori, therefore even manually setting the hash to "" the user will still see the login page.
The Route components have in addition to the already seen prop initAccess, and the prop hash indicating their hash route, the prop routeKey which is used as a key to and from the state manager to read or write the accessibility to that route.
This component also receives a child element with the UI element to be displayed if that route is accessible and is the one chosen based on the current hash.

4.4 Material UI

Material UI was chosen as the library of basic graphic components. This library for React incorporates all the best practices in terms of performance and user experience, perfectly aligning with the guidelines of the material design project.
Instead of starting from the basic HTML elements, and investing time to create a semblance that at the same time optimizes performance, usability, stylistic pleasantness, in fact, it is much more convenient especially in projects like this where a graphic customization is not a priority, rely on the work community and "take advantage" of the work done over the years by frontend developers with great skills and experience.
The library provides a set of basic components, such as the paper, which is a mere container,

the button, which can function both as a link and as a button itself, the text input, etc.
But it also contains more complex graphic components but very common at the same time, such as the chip, which is a graphic element used to list tags or properties of an element, customizable with a small button for editing or deleting, or the tab bars, ie elements made up of several tabs (similar to buttons) that allow you to navigate multiple views on the same page.
For the management of the texts there is also the Typography component that contains all the typology of texts, from the header texts (h1, h2, h3, etc.) up to the caption label, or for the normal text called body.
From a purely user experience point of view, many if not all of these elements, especially the interactive ones, contain out-of-the-box basic animations to click or focus-in events, which make interaction with the platform.
From a graphic point of view, however, Material UI offers a very powerful system based on themes.
In this system, it is possible to specify some high level variables, such as a primary and a secondary color, and all the components used within that realm of that specific theme, will adapt their graphics following those two basic colors.
All this is favored also thanks to the use of props on the components themselves. For example, by switching to the prop color of a button the text "primary", that button will be colored with the primary color, and the animations on the click will have relevant shades.
The creation of the Javascript object for the theme, since it is not wanted to use the default one, is the only graphic choice that has been made.
In particular, for the choice of the two main colors, primary and secondary, the online platform coolors.co was used, which randomly and automatically generates color palettes optimized from a UX and UI point of view.
The theme object of Material UI actually has many other properties that allow you to adjust the graphics and size of many elements, from interactive to textual ones.
Each theme also has a palette.type property that can be light or dark, which communicates to the library and all its components whether the main theme of the UI is dark, therefore with a dark background or not.
This is useful to Material UI to automatically set the constrastText, that is the color of the texts in contrast to the background.

4.5 Redux - the state manager

Redux is a JavaScript library for state management.
Since it is not tied to any other technology other than the JavaScript language, it is considered agnostic.
Thanks to this peculiarity it can be used in any context, backend, frontend and mobile, and with any library and framework.
However, its main use is on the frontend side, and is practically a de facto standard for React applications with a certain degree of complexity.
The redux feature killer is the need for a centralized state and source of truth.
If the props allow us to pass properties, therefore information, between a parent and a child

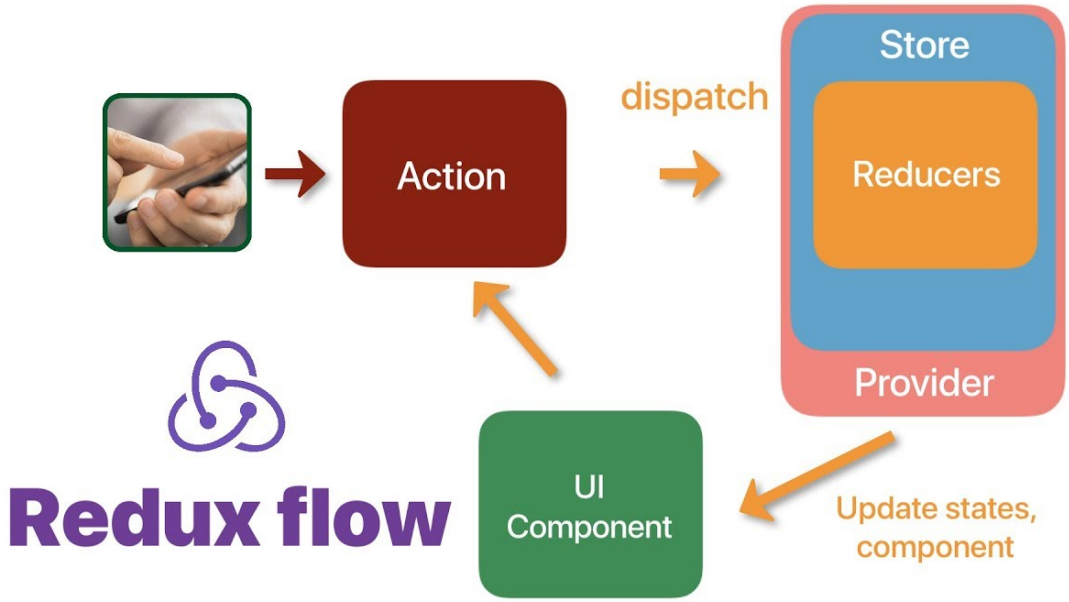

Figure 4.6: Redux flow

component, and the internal state of a generic component allows that same component to have a local and persistent data source, we need a global state of our application, which allows you to share information not only between directly connected components, but also far away in the hierarchy of the application and its tree structure.
In redux, this global data source is called the store.
The store is an immutable javascript object that contains the state of an application in a serialized form.
To change the state of the store, an action is dispatched (4.6).
An action is a Javascript object, containing a unique ID and parameters.
Action objects are intercepted by redux, and the store's status is updated through the reducers.
A reducer is a pure and synchronous function which, given the same input data, always generates the same output.
The inputs of the reducer are the state of the current store, and the action object.
Based on these two objects, the reducer returns a new store state, which is updated accordingly.
The reducer should not mutate the previous state, but generate a new one, exploiting the JavaScript cloning features or other immutable libraries.
This is really important to exploit all the other redux features, like the time-traveling, that allows developers to jump from one state of the app to an other, and check step by step how the application changes and how this reflects on the UI.

4.5.1 React-redux

We said redux create a global source of truth object, the store, but how this is connected and communicate with React components?React provides an ad hoc tool for this type of need, the Context.
Through the context, the components can perform pub / sub operations to a data source.
Redux, or rather the plugin that connects the redux agnostic library to the react world, react-redux, uses this technology to share with all the components, its internal store data.

React-redux uses a wrapper component, called Provider, which receives a store as a prop, created through redux's createStore function.

The Provider takes care of populating the context with the state of the store, and making it available to all children components.

The React-redux connect function takes care of mapping the state of the store to props, as well as binding the callback props to the dispatch function of redux.
This connect function is a High Order Component (HOC), and acts as a wrapper around the component.

4.5.2 Redux store

The redux store can be created by combining multiple slices.
Since the store is a JavaScript object, a slice represents a sub-object, and is identified by a key.

Four different slices were created within the project, each with an isolated scope. These slices are: global, routing, ui and project.

Each slice is initialized with an initial state. The slice global contains useful information within the whole application, that is the feedback object, which contains the information to be shown to the user in toast format, and the restUri string which represents the base URL of the Ajax calls to the backend API.

The slice routing contains information useful for the internal routing of the Single Page Application, and indicates to the various scenes which routes are active, the route currently visible, if there is a loading status, etc.

The UI slice instead contains generic information about the UI, useful for example to show and hide dialogs in specific cases. Finally, the slice project is that relating to the management of the project model, which is the basis of the logic of the application, and contains the data on the projects downloaded from the server and ready for viewing on the user side.

4.5.3 Redux actions

Each redux action is composed of a type, which is a unique ID, and parameters.

An action-types folder was created within the project, with sub-folders, each of which logically linked to a slice of the store, plus some that are not part of a particular slice but belong to a different scope, such as for example the auth folder, which contains the actions for authentication, which does not have a specific slice within the store. This is an example of the global action type file:

```
export const APP_STARTUP = 'APP_STARTUP'
```

```
export const CLOSE_FEEDBACK = 'CLOSE_FEEDBACK'
export const SHOW_FEEDBACK = 'SHOW_FEEDBACK'
export const UPDATE_AJAX_LOADING = 'UPDATE_AJAX_LOADING'
export const UPDATE_AJAX_UPLOAD_PROGRESS = 'UPDATE_AJAX_UPLOAD_PROGRESS'
```

An actions folder, on the same level as the action types, contains action creators. This folder has the same sub-folders as the previous one, and each action type corresponds to an action creator, with some exceptions.

The simplest actions receive parameters or not, and return an object with the specific type, such as:

```
export const showFeedback = feedback => ({
  type: SHOW_FEEDBACK,
  feedback
})
```

This function takes as input a feedback object, and returns a redux object with the type field populated with the corresponding action type, plus the same parameter that will be intercepted by the reducer.

In addition to these more trivial functions, there are actions that trigger a connection with the server.

These actions have some specific parameters, and their action types always end in _AJAX_REQUEST. Thanks to this standard, it is possible to build a unique Ajax call handler, which manages all the side effects.
This is possible thanks to the technology introduced by redux-saga, as we will see later.

4.5.4 Redux reducers

As we have seen previously, the reducer aims to change the state following the firing of an action.

Since the application store consists of multiple slices, a reducer must be specified for each slice. In this way it is also easier to manage the entire state of the application, since the reducer will receive only the object corresponding to that slice, and not the entire store.

It will be up to redux to merge each slice into the output of each reducer in a single global state by updating the store.

This is for example the slice global reducer code:

```
import {
  SHOW_FEEDBACK,
  CLOSE_FEEDBACK,
  UPDATE_AJAX_LOADING
} from '../../action-types/global'

export default (state = initialState, action) => {
  let api, isLoading, uid, feedback
```

```
  switch (action.type) {
    case UPDATE_AJAX_LOADING:
      ;({ api, isLoading, uid } = action)
      state = state.setIn(['ajaxLoaders', `${api}_${uid}`], isLoading)
      return state
    case SHOW_FEEDBACK:
      feedback = get(action, 'feedback', null)
      if (feedback) {
        state = state.setIn(['feedback', 'isOpen'], true)
        state = state.setIn(['feedback', 'message'], feedback.message)
        state = state.setIn(['feedback', 'type'], feedback.type)
      }
      return state
    case CLOSE_FEEDBACK:
      state = state.setIn(['feedback', 'isOpen'], false)
      return state
    default:
      return state
  }
}
```

Each reducer is a pure function that takes the current state (of that single slice) and the input action.

Through a switch case it is possible to identify the current action, and each case corresponds to the creation of a new slice.

The immutability of the state, property to be guaranteed by working with redux, is ensured by the use of the immutable library, which allows you to generate a new object with each state change.

Each slice is in fact an immutable object, and changes are made through the setIn function which clones the entire object, applies the changes, and returns it.

4.5.5 Redux getters

The last folder in the redux folder is the getters folder. This folder also has as many subfolders as there are slices.
For each slice, functions called getters, or selectors, are exported.

A getter receives the global state of redux as an input parameter, and returns a part, or a single variable. The division into slices is therefore to keep the code ordered, since there is no specific constraint on the single slice.

This is the file containing the getters for the global slice:

```
export const getAjaxLoading = (state, type, uid = 1) =>
  type &&
  !!state.getIn([
    'global',
```

```
    'ajaxLoaders',
    `${type.replace('_AJAX_REQUEST', '')}_${uid}`
  ])
export const getFeedback = state => state.getIn(['global', 'feedback'])
export const getRestUri = state => state.getIn(['global', 'restUri'])
```

Since the immutable library was used to represent the state on redux, the getIn function is used to access the individual properties.
If this library had not been used, for example to access the feedback object of the slice global, the getter would have been:

```
export const getFeedback = state => state.global.feedback
```

4.5.6 React-redux connect

We have previously seen Redux-react's wrapper connect function. Now that we also have an idea of how the folders are organized, and the various functions involved, we can connect everything with an example:

```
const mapStateToProps = state => ({
  projects: getProjectsList(state),
  restUri: getRestUri(state)
})

const mapDispatchToProps = dispatch =>
  bindActionCreators(
    {
      getProjectsListAjax,
      openProject
    },
    dispatch
  )

export default connect(mapStateToProps, mapDispatchToProps)(Header)
```

Header is the component that renders the information at the top of the page. This component is connected to redux via the connect function, which accepts two input functions.
These functions are called mapStateToProps and mapDispatchToProps.
The first receives the status of the redux store at the input, and returns an object with the props that will be passed to the component. The second function takes the redux dispatch function as input, and returns another set of props-callbacks that the component can call to invoke actions on the store itself.

For the first function, the getters functions are used to access the state and populate the two props projects and restUri. Instead, the bindActionCreators function used in the second function takes care of binding action creators, with the dispatch function of redux.

The simple functions / actions seen previously are thus injected as props to the component, but after being binded with the dispatch function, so that when invoked, the resulting object is intercepted by redux. The following is the code of the Header component.

The four props, two from the mapStateToProps, and the two from the mapDispatchToProps, are injected into the component as if they had passed through the father, which is precisely the purpose of the HOC connect.

```
class Header extends PureComponent {
  static get propTypes() {
    return {
      getProjectsListAjax: func,
      openProject: func,
      projects: object,
      restUri: string
    }
  }

  componentDidMount () {
    this.props.getProjectsListAjax(this.props.restUri)
  }

  render () {
    const {
      openProject,
      projects
    } = this.props

    return (
      <div className={cn(sa.W100, sa.H100, sa.P20)}>
        <Typography>New project</Typography>
        <Button
          className={s.AddButton}
          onClick={() => window.location.href = '/#/new-project'}
          variant='outlined'
        >
          <Add />
        </Button>
        <div className={cn(sa.MT20)}>
          <Typography>All projects</Typography>
          {
            projects.map(project => (
              <Button
                className={sa.MR20}
                key={project.get('_id')}
```

```
                onClick={openProject.bind(this, project.get('_id'))}
                variant='outlined'
              >
                {project.get('name')}
              </Button>
            ))
          }
        </div>
      </div>
    )
  }
}
```

4.5.7 Redux sagas

If redux allows you to have control of the global state of the application through actions and reducers, which are pure and synchronous functions, you need a tool that takes into account asynchronous operations. In the frontend area, the most classic of asynchronous operations is the Ajax call to the server.

Ajax stands for Asynchronous JavaScript And XML, and it is now standard to use this browser-side library for loading dynamic data that populates and enriches the static HTML content downloaded at the initial request of the page 4.7.

Ajax technology is therefore essential for a SPA, since the only synchronous request made to the server is the initial one, and all subsequent contents are dynamically downloaded.

Redux saga allows us to manage asynchronous call flows, such as an Ajax call, through the use of generating functions. a generating function is a function that can interrupt its flow of executions at any given moment, and resume from the same point. Unlike a classic function, which generates a single result, or return, a generating function can generate none, one, many, or infinite results, before ending its execution. Redux saga takes advantage of this technology by defining side effects, that is, functions that communicate directly with the redux store, for example by waiting for a certain action to be sold dispatched or not. Redux saga manages to interact with the redux store, acting as a middleware. A middleware is a software component which can be connected to redux, and which has access to the dispatch, to the state, and is also subscribed to every dispatched action. Middleware is initialized in this way, and passed to redux.

```
...
import createSagaMiddleware from 'redux-saga'
...
const sagaMiddleware = createSagaMiddleware()

const store = createStore(
  combineReducers(reducers),
  initialState,
```

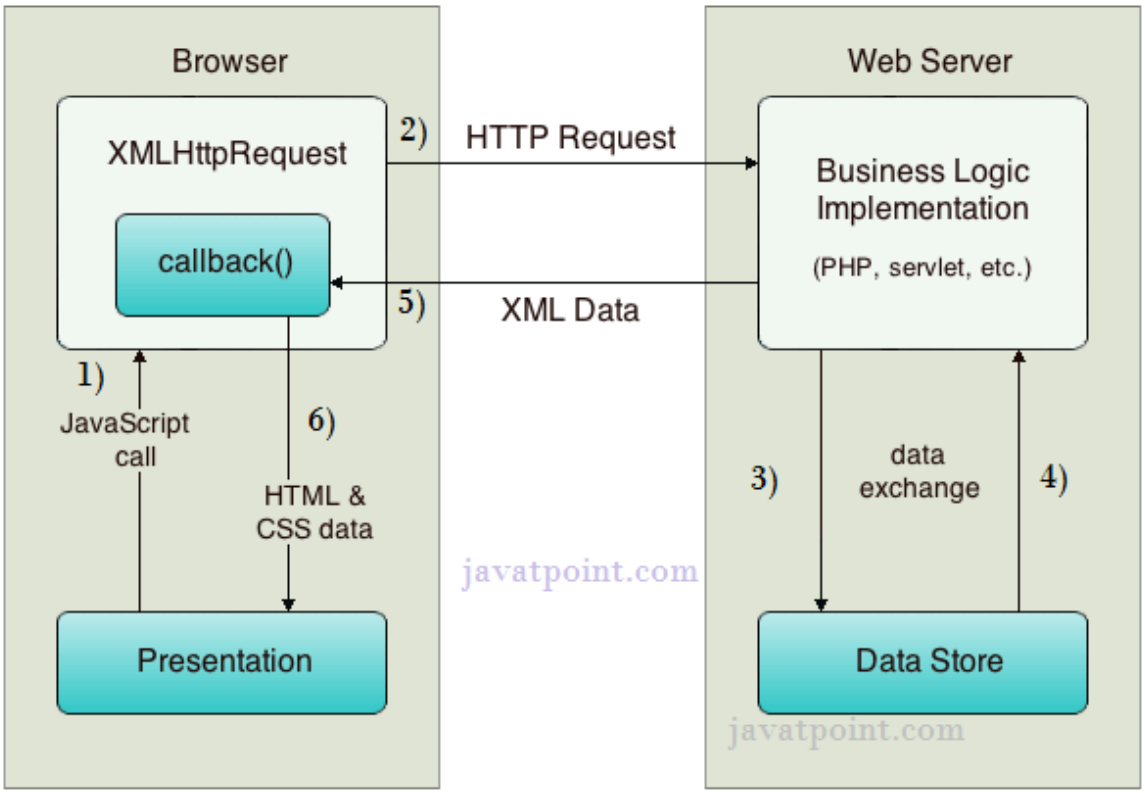

Figure 4.7: Ajax lifecycle [16]

```
  composeEnhancers(applyMiddleware(sagaMiddleware))
)
sagaMiddleware.run(rootSaga)
```

The rootSaga is a generating function, which in turn can define multiple generators with several side effects.

```
export default function * rootSaga () {
  yield all([
    ajaxRequestSaga(),
    ajaxUploadSaga(),
    routingsSaga(),
    startupSaga(),
    showFeedbackError(),
    newFolderSaga(),
    createProjectSaga(),
    updateFacilitySaga()
  ])
}
```

This is an example of a generator function used to intercept all actions that correspond to an Ajax call.

```
import { fork, take, takeEvery, put } from 'redux-saga/effects'

function * requestSaga () {
  yield takeEvery(action => /^(.*?)_AJAX_REQUEST/.test(action.type), request)
}
```

The takeEvery effect generates the execution of a new function passed as a second parameter, upon the occurrence of an action (or a series of actions, via a Regular Expression) dispatched on the store. This is an example of an action invoked for user login. The action is a simple object with parameters, it is redux saga which takes care of calling the side effect that generates the Ajax call, and of managing the result.

```
export const loginAjax = (restUri, credentials) => ({
  type: LOGIN_AJAX_REQUEST,
  url: `${restUri}${routes[apis.AUTHENTICATION.CREATE]}`,
  method: 'POST',
  data: {
    credentials
  }
})
```

When the action is intercepted by redux saga, the following generating function is invoked, with the parameters of the action itself. Within this generating function, the axios library is

used for the ajax call. At the end of the Ajax call, the result is checked, and based on the return values, a success or failing action is generated, which in turn will be intercepted by the reducer (s), which will update the global status, and therefore the user interface .

```
function * ajaxTask (action, cancelToken) {
  const {
    type,
    url,
    method,
    data,
    query,
    uid = 1,
    options = {
      showFeedbackOnError: true
    }
  } = action
  let api = type.replace('_AJAX_REQUEST', '')

  try {
    yield put(updateAjaxLoading(api, true, uid))
    let response = yield axios({
      url,
      method,
      data,
      params: query,
      withCredentials: true,
      cancelToken: cancelToken.token
    })
    let resultCode = _.get(response, 'data.resultCode', '1').toString()
    let responseData = _.get(response, 'data.data', null)
    if (resultCode === '0') {
      yield put({
        type: `${api}_AJAX_SUCCESS`,
        data: responseData,
        requestData: data,
        requestQueries: query,
        uid,
        options: action.options
      })
    } else {
      yield put({
        type: `${api}_AJAX_FAILED`,
        errorCode: resultCode,
        data: responseData,
```

```
        requestData: data,
        uid,
        status: _.get(response, 'status', 200),
        showFeedbackOnError: options.showFeedbackOnError
      })
    }
  } catch (err) {
    if (!axios.isCancel(err)) {
      if (_.get(err, 'response.status', 200) === 401) {
        yield put(authenticationExpired())
      }
      yield put({
        type: `${api}_AJAX_FAILED`,
        errorCode: codes.INTERNAL,
        uid,
        status: _.get(err, 'response.status', 200),
        showFeedbackOnError: options.showFeedbackOnError
      })
    }
  } finally {
    yield put(updateAjaxLoading(api, false, uid))
  }
}
```

4.6 Project flow

Once logged in, the project management flow follows three phases, preparation, execution and results. It is possible to access each of the three sections at any time, even if from an operational point of view, we start from the preparation, which involves loading the documents useful during the test, and from the definition of the facility that will host the test; therefore the test is carried out with the live loading of the results obtained from the test itself; at the end it is possible to consult the results obtained (on site in real time or after days in your laboratory).

4.6.1 Preparation

The preparation phase is divided into two sub-phases, the loading of documents, and the experiment overview. Uploading documents is organized as a virtual file system, with the ability to create folders, subfolders, and to upload and delete files in each of them.
For each folder on the root, and for each file, a button is created.
The folder button changes the current path displayed, updating the list.
The file button instead is a link to the static resource at the backend, which is downloaded after the click.
A special button inside each subfolder with the back icon takes you back to the virtual path

tree.

To upload the file, another button has been created, connected to an HTML input, which triggers the upload call to the backend at the event onchange through an ad hoc redux action.

The other subsection is the experiment overview. In this screen it is possible to manage the facilities. Each facility is identified by a name and an address, in order to be uniquely identified.

Furthermore, for each facility it is possible to preload all the ions available to the structure, in order to select it quickly during the test. Each ion is identified by the name and the 4 characteristic energy values of that facility.

By clicking on the facility, it is possible to set it as the active one for the current project. In this way, during the next phase of execution, it will be possible to quickly access the predefined ions and select them for the experiment in progress.

4.6.2 Execution

In the second phase of execution, the researcher has the possibility to quickly populate all the data necessary for the calculation of the results. The first input field is a textbox that adds a tag when the enter key is pressed. The tag (or chip) is a label that identifies the family to which a single insertion belongs. This list of labels is saved on redux, and it is obviously possible to delete added tags that are no longer needed. Another input receives the number of bits which at each run can or may not generate an error (the so-called bitflip). Subsequently a table contains the results of each run. Each row of the table has a command to quickly modify and delete a run, and at the bottom of the table an empty row allows the insertion of a new run. The edit button in particular converts individual cells from output to input, so you can quickly re-input the wrong value and save the row. Furthermore, for each run it is possible to specify a tag, from the list created before. This tag will be essential for viewing the results. To optimize and further speed up this work, especially in cases where the data is already present, and a test has already been simulated, a "read from clipboard" button has been created. This command opens a dialog, in which a list of rows and columns copied, for example, from excel is read from the clipboard, and it is possible to "map" these columns in fields of the run table. This way you are not tied to a particular model, but dynamically, differently formatted excel files can be loaded into the platform. Before reading from the clipboard, it is important to check that the browser has the permission to do that:

```
const result = await navigator.permissions.query({ name: 'clipboard-read' })
```

If the permission is granted, it is possible to read it with this command:

```
let data = await navigator.clipboard.readText()
```

4.6.3 Results

The final stage is that of results. All runs are first grouped by tag. For each run, the error rate is therefore calculated as the ratio between the number of errors detected, on the number of total bits inserted upstream of the previous table.

```
  for (let run of runs) {
      const index = charts.findIndex(c => c.tag === run.tag)
      if (index !== -1) {
        const errors = parseFloat(run.errors) + (charts[index].data.length === 0 ? 0 : chart
        const errRate = errors / parseFloat(bits)
        charts[index].run += 1
        charts[index].data.push({
          errors,
          errRate,
          run: charts[index].run
        })
      }
    }
```

For each tag, and for each chart data, a LineChart is then printed:

```
<LineChart
  width={chartWidth}
  height={300}
  data={chart.data}
>
  <Line type="monotone" dataKey="errRate" stroke="#8884d8" />
  <CartesianGrid stroke="#ccc" />
  <XAxis dataKey="run" />
  <YAxis />
</LineChart>
```

The npm package used to create the charts with React is "rechart".

```
import { LineChart, Line, CartesianGrid, XAxis, YAxis } from 'recharts'
```

CHAPTER 5

Real case scenario

With technology advancements in recent years, Filed Programmable Gate Array (FPGA) devices are gaining more and more attention due to increasing performance and high flexibility.

Even in safety and mission critical applications, FPGA presents a flexible solution to be considered to meet the ever-increasing computational demand of certain tasks such as image and signal processing ([17], [18], [19], [20], [21]).

However, the same as other electronic devices, to be deployed in safety and mission critical applications such as avionic and space missions, the reliability of FPGA device has to be evaluated against faults and errors induced by radiation effect.

Radiation test is one of the methods that are commonly used for such evaluation as it could emulate the space environment using accelerated particles to apply to the device under test.

Therefore, a radiation test has been performed on the Xilinx7 SRAM-based FPGA device using the first-ever available ultrahigh energy (UHE) HI beam, provided in CERN radiation facility targeting the sensitivity of the configuration memory of the device against radiation induced Single Event Upset (SEU).

SUE in a change of state caused by one single ionizing particle (ions, electrons, photons. . .) striking a sensitive node in a micro-electronic device. This change of state is caused by ionizing radiation strikes that discharge the charge in storage elements such as configuration memory cells, user memory, and registers.

During the radiation test, a Xilinx Kintex 7 FPGA KC705 Evaluation Kit equipped with a Kintex7 XC7K325T SRAM-based FPGA was used as design under the test. An ARM-based SoC was used as benchmark circuit which contains an ARM Cortex-M0 processor provided by ARM as flattened netlist through University Program. For the radiation test, two version of the ARM-based SoC have been prepared:

Plain Version: The original ARM-based SoC TMR version: The original version of ARM with the applied TMR. Triple Modular Redundancy (TMR) is one of the most popular fault tolerant techniques which triplicates the logic path and other sequential elements including Flip-Flops and block memories together with automatic voter insertion.

The Xe HI beam was used for the radiation test with energy level set to 40GeV/n and the effective LET is 3.7 MeV.cm2/mg obtained by FLUKA while considering the volume around

1m3.

The particles with such high energy level are capable to penetrate the device with package, and possible to generate single event multiple upsets in the configuration memory of the SRAM-based FPGA.

Since it was the first time UHE beam in CERN was open for third-party testing, they only had the chance for testing the DUT for one LET value. However, though the LET is quite constant due to high penetration, there is still high-LET fragment production up to 12 MeV.cm2/mg.

As a result of exposing the device to mentioned radiation particles, the error rate has been calculated.

The error rate is calculated as the probability of application generating an error at the output with respect to a certain number of SEU accumulated in the configuration memory. Figure 5.1 shows the Error rate for the original and mitigated version of the design under the test.

5.0.1 Preparation phase

First we create the project, giving an identifying and self-explanatory name (Fig. 5.2).

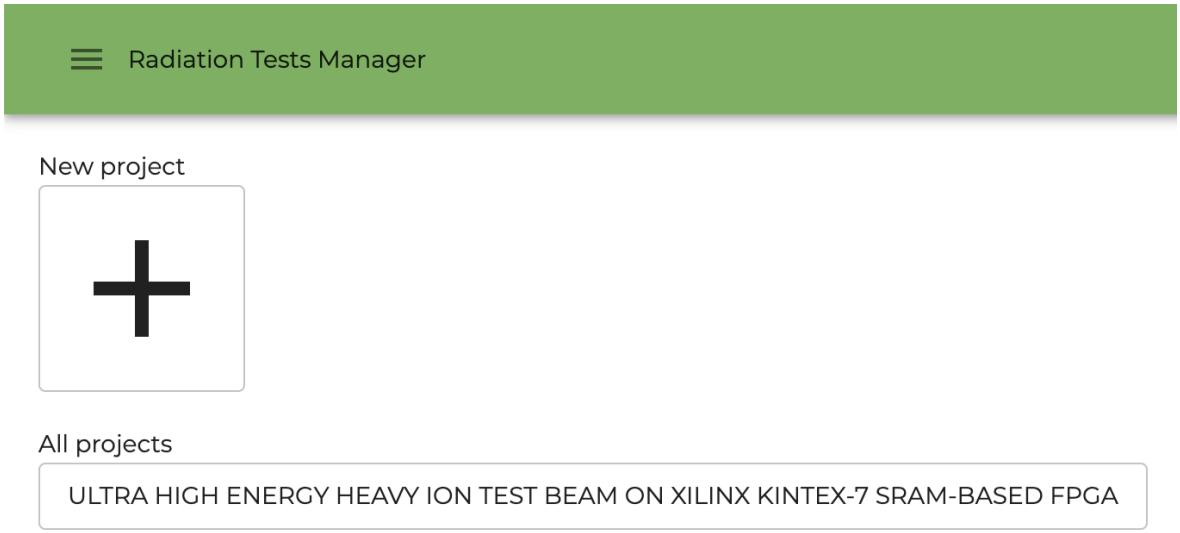

Figure 5.2: Projects list

The first phase is that of preparation. In this phase it is possible on the one hand to upload the necessary documents during the test, on the other select (if already present in the system) the facility where it will be possible to carry out the test. If the facility is not present, you can create it using the appropriate button.

We select the facility by clicking on it (Fig. 5.3), checking that the list of ions available is updated and complete. Otherwise it is possible to update the data and save (Fig. 5.4).

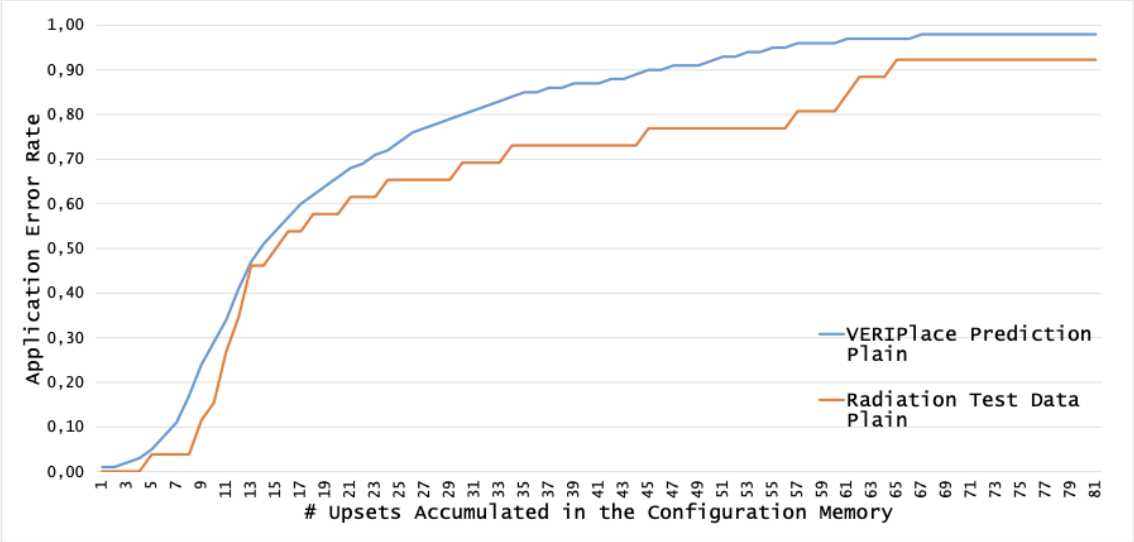

Fig. 7. VERI-Place error rate comparison with radiation test data for Plain version.

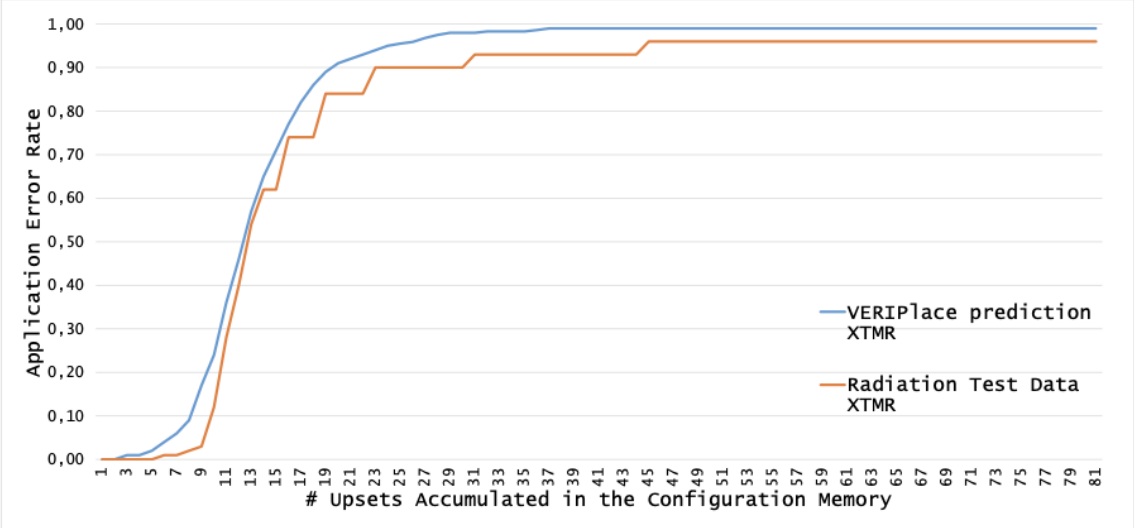

Fig. 8. VERI-Place error rate comparison with radiation test data for XTMR version.

Figure 5.1: Test error rates

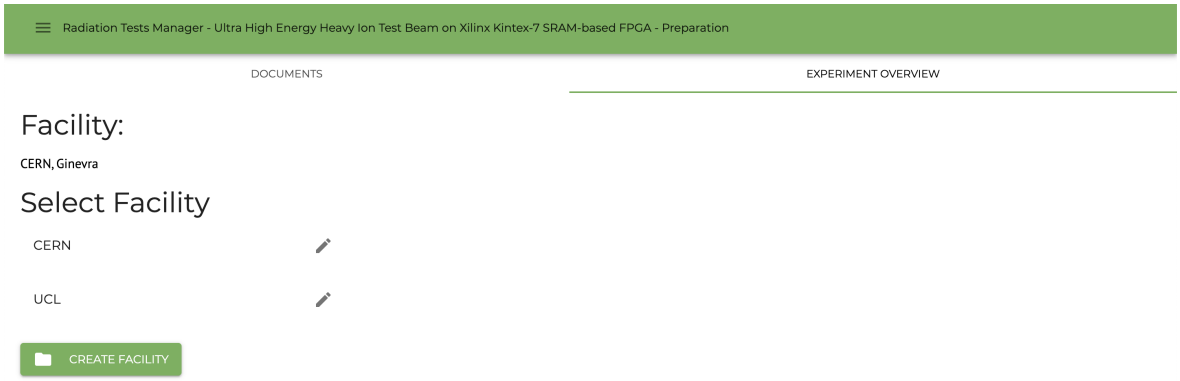

Figure 5.3: Facility selection

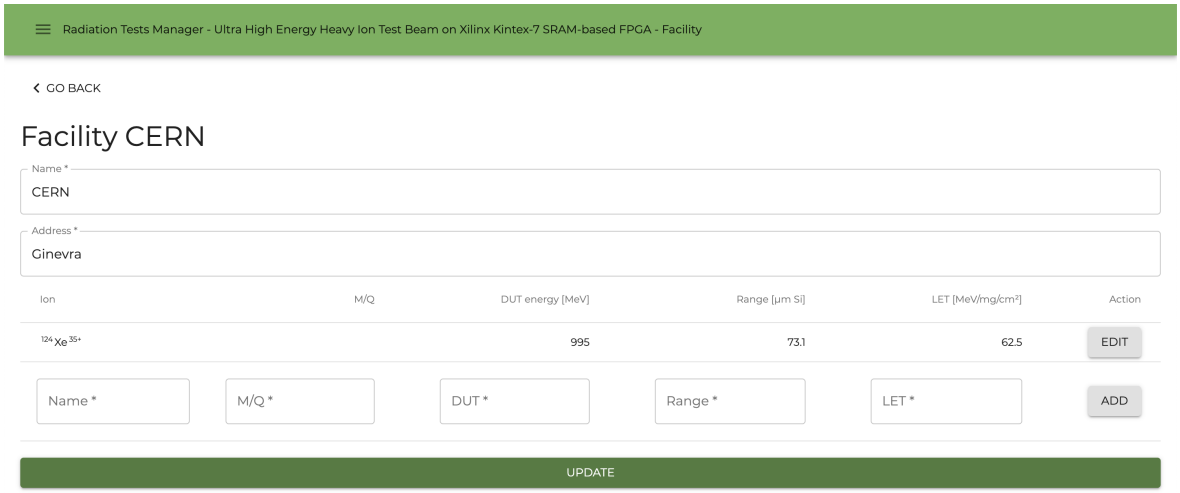

Figure 5.4: Update CERN ions

5.0.2 Execution phase

Once the material has been prepared and the facility set up, with the ions to be used and their data, it is possible to move on to the second execution phase.

The first step of this phase is the definition of one or more tags. Each tag represents a particular experiment. In this particular project, two experiments were considered, one called "plain", and one "XTMR", where XTMR stands for Xilinx Triple Modular Redundancy (Fig. 5.5). The first experiment does not involve any type of error mitigation, while the second does.

Since it is practically impossible for a research group to switch between multiple ion beams during the same project, the selection of the ion considered at the project level is unique, and takes place through a select box, populated with the list of all the preloaded ions in the facility.

During this project, the research team used Xenon as an ion, and therefore this was selected.

Another global project data is the number of bits tested at the output. The goal is in fact to minimize accidental errors, due precisely to radiation, therefore to calculate the percentage of errors, also called error rate, we need not only to know for each run how many bits have had a flip, but the total number Of these. The heart of the platform is the insertion of the runs. Each run corresponds to a tag, to indicate the type of experiment in progress, the number of errors detected after irradiation of the ions, and the number of particles used.

The insertion of the runs is done through direct input into the table, or through a special shortcut created specifically to load data from the clipboard. This function has been provided for those cases in which the experiment has been previously saved on an excel sheet, and you want to quickly map and load the data within the platform.

This tool focuses entirely on user experience and flexibility of use. By copying an excel table, with for example two columns, one with the numbers of errors, and the second with the number of particles, and clicking "Read from clipboard", the platform will automatically show a dialog with the total number of lines read, and for each column found a name will be associated ("Field 0", "Field 1", etc.). Each Field can therefore be associated with one of the three inputs, tags, errors and particles.

If a data is not present in the copied table, one can select or specify a default one. In this way it is possible to load quickly even old formatted data without a specific criterion. For this project, the research team had transferred all the results on an excel sheet, with which through repetitive operations, it had obtained the published results.

To test the platform, these data were copied from the excel, and saved in the DB thanks to the reading function from the clipboard.

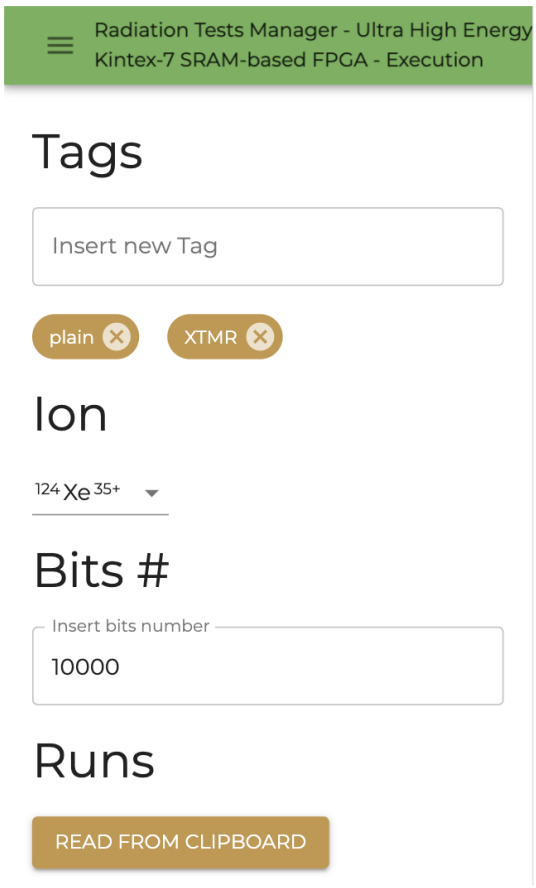

Figure 5.5: Project runs' tags

5.0.3 Results phase

Once all the data have been loaded, with a few simple clicks, it is therefore possible to access the results instantly and immediately, without having to resort to ad hoc excel sheets anymore. In the figure 5.6) it is possible to view the graphs resulting from data entry. A graph is generated for each tag, with the representation of the error rate.

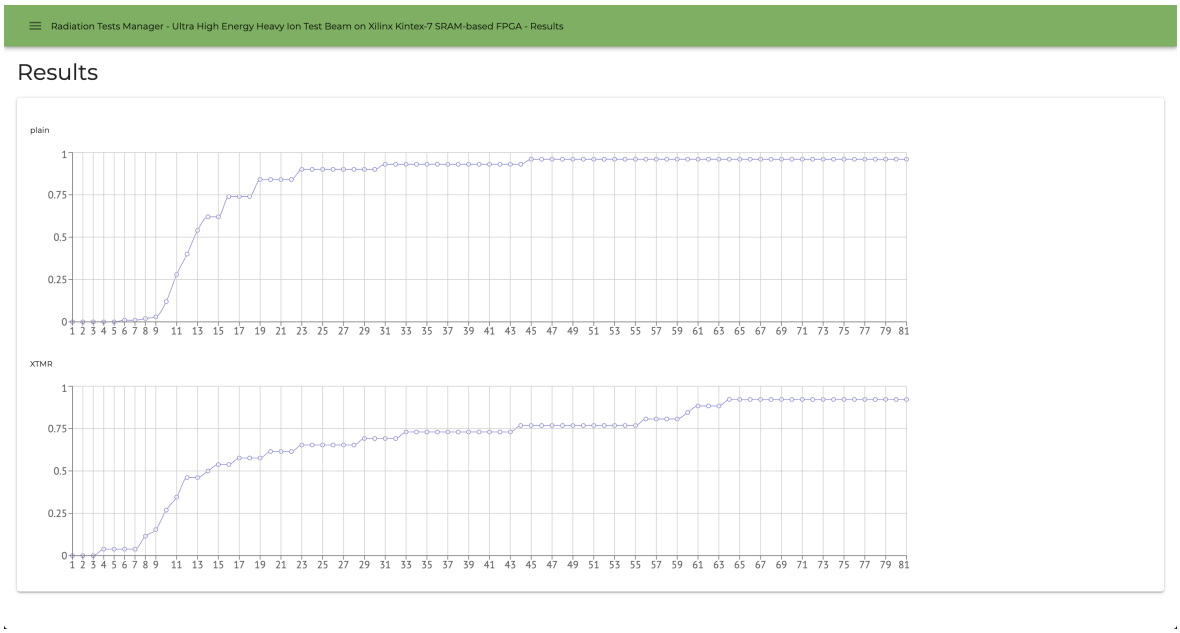

Figure 5.6: Project results

CHAPTER 6

Conclusions

Given the good results based on a past real case scenario, the next phase for the project will be the test in a real radiation test inside a facility.

Error rate is just one of the many output results that could be automatically generated based on the data entered.

As any software, this and many other features could be developed, to provide a better and complete experience during the tests, and to achieve this next real tests will be crucial to find weak and strong points of the tool, allowing future implementations to improve the overall quality and user experience.

Also, thanks to the adopted stateless architecture, after facility tests and improvements, the tool could be distributed as a Saas (Software as a Service) to other research groups, creating not only a place where the community of people working on the same field can share results and best practices, but also a revenue model.

References

[1] *Hypertext Transfer Protocol – HTTP/1.1 https://tools.ietf.org/html/rfc2616*

[2] *https://www.mailsenpai.com/cosa-e-un-server.*

[3] *https://www.dropsource.com/blog/rest-api-best-practices.*

[4] *https://itnext.io/api-calls-and-http-status-codes-e0240f78f585.*

[5] *Fielding, Roy Thomas (2000). "Chapter 5: Representational State Transfer (REST)" https://www.ics.uci.edu/ fielding/pubs/dissertation/rest_arch_style.htm*

[6] *https://parallexis.com/a-guide-to-user-experience-ux-research/*

[7] *https://blog.shikisoft.com/which_elastic_beanstalk_deployment_should_you_use/*

[8] *https://aws.amazon.com/it/ecs/*

[9] *https://docs.aws.amazon.com/whitepapers/latest/microservices-on-aws/serverless-microservices.html*

[10] *https://fabcodes.hashnode.dev/is-nodejs-really-single-threaded-here-is-what-i-think-ck1x8v2st00ap9ns1wg45u2ij*

[11] *https://www.traininginbangalore.com/mongodb-training-institute-in-bangalore/*

[12] *https://towardsdatascience.com/being-aware-of-malicious-data-corruption-as-a-data-scientist-sql-injection-attack-63f235fb2a97*

[13] *https://blog.usejournal.com/everything-react-all-about-jsx-4a5123ac8606*

[14] *https://www.ma-no.org/en/programming/what-is-webpack-and-how-does-it-work*

[15] *What Is A Single Page Application? Meaning, Pitfalls Benefits https://www.excellentwebworld.com/what-is-a-single-page-application/*

[16] *Building Ajax Request And Caching The Response With URL https://medium.com/@mithun.kadyada/building-ajax-request-and-caching-the-response-with-url-b85c45431acc*

[17] *L. Sterpone, F. Luoni, S. Azimi and B. Du, "A 3D Simulation-based Approach to Analyze Heavy Ions-induced SET on Digital Circuits," in IEEE Transactions on Nuclear Science, doi: 10.1109/TNS.2020.3006997.*

[18] *C. De Sio, S. Azimi, L. Sterpone and B. Du, "Analyzing Radiation-Induced Transient Errors on SRAM-Based FPGAs by Propagation of Broadening Effect," in IEEE Access, vol. 7, pp. 140182-140189, 2019, doi: 10.1109/ACCESS.2019.2915136.*

[19] *S. Azimi, B. Du, L. Sterpone, D. M. Codinachs and L. Cattaneo, "SETA: A CAD Tool for Single Event Transient Analysis and Mitigation on Flash-Based FPGAs," 2018 15th International Conference on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD), Prague, 2018, pp. 1-52, doi: 10.1109/SMACD.2018.8434897.*

[20] *C. De Sio, S. Azimi, L. Sterpone and B. Du, "Analyzing Radiation-Induced Transient Errors on SRAM-Based FPGAs by Propagation of Broadening Effect," in IEEE Access, vol. 7, pp. 140182-140189, 2019, doi: 10.1109/ACCESS.2019.2915136.*

[21] *S. Azimi, B. Du, L. Sterpone,"On the prediction of radiation-induced SETs in flash-based FPGAs," Microelectronics Reliability, Volume 64, 2016, Pages 230-234, ISSN 0026-2714,*