



ISIMA - Campus Universitaire des Cezeaux  
1, Rue de la Chebarde  
TSA 60125, CS 60026  
63178 Aubière CEDEX



CERN  
Route de Meyrin 385  
Meyrin, Switzerland

Engineer Report

3rd Year Internship

# LHCb Performance and Regression Development

Presented by: **Amine Ben hammou**

CERN Supervisor:  
**Benjamin Couturier**

Internship duration:  
**5 months**

ISIMA Supervisor:  
**MESNARD Emmanuel**

Date of presentation:  
**August 28th 2015**

# Acknowledgments

# Figures Table

Figure 1: Database Entity Relation Schema .....	5
Figure 2: The files structure of the frontend application .....	9
Figure 3: NodeJS and npm installation commands on Ubuntu .....	10
Figure 4: Git installation command on Ubuntu .....	11
Figure 5: Bower and Gulp installation commands .....	11
Figure 6: Dependencies installation commands .....	11
Figure 7: Gulp task declaration example .....	12
Figure 8: Starter example of using AngularJS .....	14
Figure 9: AngularJS module declaration example .....	15
Figure 10: AngularJS Controller example .....	15
Figure 11: Using Angular Controller in the view .....	16
Figure 12: Sample View .....	16
Figure 13: Data table sample directive template .....	17
Figure 14: Data table sample directive code .....	18
Figure 15: Data table sample directive result .....	19
Figure 16: Example of injecting \$http service .....	19
Figure 17: Routes configuration example .....	20
Figure 18: Example of requesting data from the REST API .....	22
Figure 19: Example of ngTable directive: the view code .....	23
Figure 20: Example of ngTable directive: the controller code .....	23
Figure 21: Example of Data Table .....	24
Figure 22: Examples of predefined chart directives .....	24
Figure 23: Example of using search-jobs directive: the view code .....	25
Figure 24: Example of using search-jobs directive: the controller code .....	25
Figure 25: Module files structure .....	26
Figure 26: Trends analysis module files structure .....	30

# Abstract

# Contents Table

<b>Acknowledgments</b>	<b>i</b>
<b>Figures Table</b>	<b>ii</b>
<b>Abstract</b>	<b>iii</b>
<b>Contents Table</b>	<b>iv</b>
<b>Abbreviations Table</b>	<b>vii</b>
<b>Introduction</b>	<b>1</b>
<b>1. Presentation of CERN and LHCb</b>	<b>2</b>
1.1. Presentation of CERN	2
1.2. Presentation of LHCb	2
<b>2. The LHCbPR project</b>	<b>3</b>
2.1. What is LHCbPR	3
2.2. Why LHCbPR	3
2.3. Users of LHCbPR	3
2.4. The old LHCbPR application	4
<b>3. New LHCbPR application</b>	<b>4</b>
3.1. Architecture	4
3.2. Data Storage Layer: Database	4
3.3. Application Logic Layer: REST API	6
3.4. Frontend	6
<b>4. Frontend Development</b>	<b>7</b>
4.1. Tools	7
4.1.1. AngularJS	7
4.1.2. jQuery	7
4.1.3. Git	7
4.1.4. npm	8
4.1.5. Bower	8

4.1.6. Gulp .....	8
4.1.7. jade .....	8
4.1.8. less .....	9
4.2. Files structure .....	9
4.3. Tools installation .....	10
4.4. Building and running the application .....	11
4.4.1. The gulpfile .....	12
4.4.2. Gulp tasks .....	12
4.5. Basics of AngularJS .....	13
4.5.1. Starting example .....	14
4.5.2. Angular Module .....	15
4.5.3. Controller and Scope .....	15
4.5.4. Directives .....	17
4.5.5. Services .....	19
4.5.6. Routing .....	19
4.6. Frontend core application features .....	21
4.6.1. States and Lazy loading .....	21
4.6.2. LHCbPR service .....	21
4.6.3. Directives .....	22
4.6.3.1. DataTables .....	22
4.6.3.2. Charts .....	24
4.6.3.3. Search Jobs .....	25
<b>5. Application Modules Development .....</b>	<b>26</b>
5.1. Files structure .....	26
5.2. Helper classes .....	27
5.2.1. Module class .....	27

5.2.2. Dependencies class .....	28
5.2.3. Colors class .....	29
5.3. Step by step example: the Trends analysis module .....	29
5.3.1. Use Cases .....	29
5.3.2. Files structure .....	30
5.3.3. Module declaration .....	30
5.3.4. Adding Search Form .....	31
5.3.5. Showing Attributes on a table .....	32
5.3.6. Filtering attributes by name .....	34
5.3.7. Showing the Chart .....	36
5.3.8. Binding selected values to the URL .....	38
<b>Conclusion .....</b>	<b>40</b>

# Abbreviations Table

<b>3TP</b>	Three Tiered Programming
<b>Ajax</b>	Asynchronous JavaScript and XML
<b>ALICE</b>	Compact Muon Solenoid
<b>CERN</b>	The European Organization for Nuclear Research
<b>CMS</b>	Compact Muon Solenoid
<b>CSS</b>	Cascading Style Sheets
<b>DOM</b>	Document Object Model
<b>HTML</b>	HyperText Markup Language
<b>ISIMA</b>	Institut Supérieur d'Informatique, de Modélisation et de leurs Applications
<b>JSON</b>	JavaScript Object Notation
<b>LHC</b>	Large Hadron Collider
<b>LHCb</b>	Large Hadron Collider beauty
<b>LHCb PR</b>	LHCb Performance and Regression
<b>npm</b>	Node Package Manager
<b>REST</b>	Representational State Transfer



# Introduction

As part of my third year at ISIMA, I did my final internship at CERN. More specifically, I worked on the LHCb experiment and participated on the development of LHCb Performance and Regression framework (LHCb PR).

The goal of the LHCb PR project is to provide developers with a profiling framework helping them to evaluate their recent changes by running analysis modules and comparing results. The results can be used to detect fails in functionalities or performance issues.

During my internship, I worked on the development of the new version of LHCbPR. My tasks were to upgrade the web application using AngularJs framework and the development of analysis modules for the new version. This version should provide the same functionalities as the old one and improve the user interface. The application should also give developers the ability to add new modules easily and without having to change the code of the core application.

This report presents the different task I have done during my five months internship and the results I obtained. After presenting CERN and LHCb experiment, I will explain the idea of the LCHbPR project and describe the old version of its web application. Then I will present the different parts of the new version. After this I will explain in details how the new frontend web application was made. Finally I will present the analysis modules made in the new version.

# 1. Presentation of CERN and LHCb

## 1.1. Presentation of CERN

CERN is the European Organization for Nuclear Research. Physicists and engineers are working in this organization to discover how the universe works by studying the basic constituents of matter - the fundamental particles. Particles are made to collide together using the largest machine in the world to see how they interact and try to figure out the fundamental laws of nature. It was founded in 1954, and it sits astride the Franco-Swiss border near Geneva. It was one of Europe's first joint ventures and now has 21 member states.

Before they collide, the speed of particles is increased to close the speed of light using a sequence of accelerators. The largest accelerator is called the Large Hadron Collider (LHC) which is a ring with a perimeter of 27km. Detectors are used in the locations of collisions to observe and record the results.

There are seven experiments running on the LHC. The biggest four are: ATLAS, CMS (Compact Muon Solenoid), ALICE (A Large Ion Collider Experiment) and LHCb (Large Hadron Collider beauty). ATLAS and CMS are general-purpose experiments investigating the largest range of physics possible. ALICE and LHCb have detectors specialized for focussing on specific phenomena.

## 1.2. Presentation of LHCb

The LHCb experiment is one of seven particle physics detector experiments collecting data at the LHC. This experiment is trying to answer the question: "Why our universe is composed almost entirely of matter, but no antimatter?".

Antimatter is made of anti-particles. An anti-particle of a particle  $p$  is a particle having exactly the opposite properties of  $p$ . When  $p$  and its anti-particle fusion, they cancel each other and produce pure energy. We think that just before the big bang, all the energy of the universe was compressed in one small point. So, at the moment of the big bang, this energy should have given the same quantity of matter and antimatter. The question is why do we observe only matter in our current universe?

When the LHCb experiment is running, the detector registers about 10 million collisions per second. It is impossible to record all of these collisions. So the solution was to use an electronic system that will select the best and the most interesting collisions to be recorded. The recorded events are stored in binary files into disks. Then Physics are using specific softwares to extract and study data on these files.

## 2. The LHCbPR project

### 2.1. What is LHCbPR

LHCb Performance and Regression (LHCbPR) is a service designed to record important measurements about results of integration and performance tests of the LHCb applications. These applications receive input in the form of configuration files and produce, as an output, various information. LHCbPR is not intended to actually run the jobs, but instead to manage and track the job results. The LHCbPR is a framework that allows LHCb software developer to push information for a run of their code(job characteristics, results, performance measures, files) to a central database. The main goal is to give developers the ability to perform analysis of the data across versions, running options and platforms.

### 2.2. Why LHCbPR

In the past, the users had to run test jobs manually using a configuration file. A job could be run for a specific application and results should be saved by the user. These results was shared between users using static documents such as HTML, CSV or e-mail.

LHCbPR was conceived as a tool to reliably organize the process of configuring and monitoring a test job execution. The framework, solves the problem of gathering the results. It provides a complete script that can produce the desired output, according to a configuration file. This script uses a list of handlers that collect the defined aspects of the job results and push them to the database. By collecting the results in such an organized manner, it provides the solution to the second problem, mentioned above; the running of the analysis.

Since, all the data are stored in the database, the framework provides an abstract and easy way to deploy algorithms and functions which perform analysis on the saved data.

Finally, LHCbPR handles the presentation of the collected data by providing a set of templates for the creation of web pages specific to each analysis and customized to the preferences of each user.

### 2.3. Users of LHCbPR

The LHCbPR users can be divided into three categories: Administrators, Application developers and End users.

The administrators group is responsible for the integrity of the collected data and the efficiency of the service. They maintain and support the system, making sure the application is functional and available to the end users.

Application developers are the users who actively design and develop modules for the framework, thus extending the functionality of the application.

Finally, the end users are the main body of the users of the service. They put the tools provided by the application developers to practical use and their job results are populating the database.

## 2.4. The old LHCbPR application

As my task is to upgrade the LHCbPR web application. I first started by reviewing the old version.

The LHCbPR follows the architectural model of three tiered programming. 3TP provides a way to theoretically categorize the components of an application, providing abstract modularity. In essence the 3TP model is a client-server architecture which is composed of three layers. Namely, the presentation tier, the functional process logic tier and the data storage tier. These tiers communicate with one another in a strictly defined way, which allows the developer to independently maintain each tier. In most practical applications of the model the presentation layer includes all forms of user interaction with the service. The process logic tier encompasses the whole of the application functionality, while the data storage tier handles the flow of the information from and to the data source (in most cases a database).

The Django framework was used to create the LHCbPR web application and the 3TP model was applied as follows:

**Presentation Layer:** A set of web pages using the jQuery library.

**Application Logic Layer:** A set of python modules made using the Django framework.

**Data Storage Layer:** An Oracle database storing all the applications and jobs informations and results.

## 3. New LHCbPR application

Instead of trying to improve the old version of the LHCbPR application, we started the new version from scratch to be able to use the new technologies of web development which was not used in the old version like Gulp task runner and the AngularJS framework. This framework enables us to build a rich client application using REST protocol for data access.

### 3.1. Architecture

The new version follows the 3TP model too, but in a more modern way. In particular, the presentation layer and the application logic layer are no longer in the same application. They are now two totally separated applications communicating using a REST API. ...

### 3.2. Data Storage Layer: Database

We kept the same database as the old version but made some little changes to it. The new database entity relation schema is presented on figure in the next page. The main entities are the following:

**Application:** represents a software used by LHCb to process or analyse data. it is described by a name and has many versions

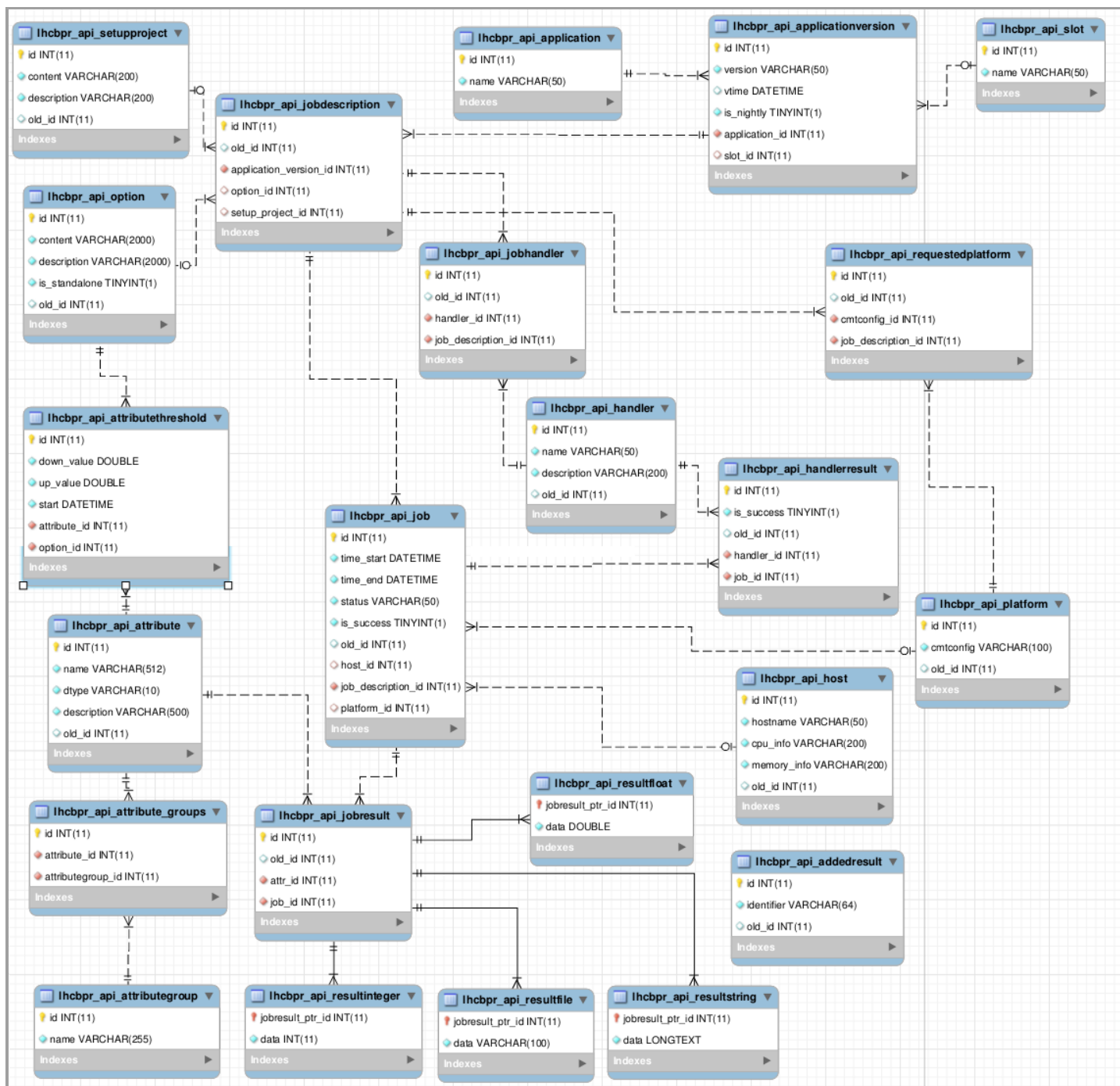


Figure 1: Database Entity Relation Schema

**Application Version:** a version belongs to an application. It can be a release or a nightly build (intermediate version compiled every night).

**Job Description:** This entity describes all that is needed to define a job. In particular, it specifies the application version, the project setup, the running option, the platforms in which jobs will be executed and the handlers that will collect results.

**Handler:** a handler has a name and a description. It gets data after the job has been run. It is linked to multiple job descriptions via the "job handler" pivot table. It is also linked to many jobs via the "handler result" pivot table.

**Option:** describes the command-line options to pass when running the jobs. It has a many to many relation with the attribute entity.

**Attribute:** an attribute can belong to multiple attribute groups. It also has a many to many relation with the job entity via the jobresult. Job results can have different types (integer, float, file, string).

**Job:** This is the main entity that describes a job that has been run. It has a start time, an end time, and a status. It belongs to a job description and is run in a specific platform and host. Their results can be collected by multiple handlers and it can have multiple results. Every job result concerns an attribute.

### 3.3. Application Logic Layer: REST API

This layer is made using the **Django REST framework** which is a powerful and flexible toolkit that makes it easy to build Web APIs using Python language. Making the application logic layer as an API is the best choice to keep layers totally independent. This way we can build multiple frontend applications ( one for the web, another for mobiles for example ) based on the same API.

REST stands for **Representational State Transfer**. (It is sometimes spelled "ReST") It relies on a stateless, client-server, cacheable communications protocol. In virtually all cases, the HTTP protocol is used. REST is an architecture style for designing networked applications.

In our case, the frontend application sends requests to the API and the API responds with the requested data in JSON (JavaScript Object Notation) format which is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript programming language which makes it easy to parse it, as our frontend is written using JavaScript.

### 3.4. Frontend

The frontend application is made using the AngularJS javascript framework which adds many features to HTML to make a new generation of web applications (single-page web applications). Instead of starting the web application from scratch, we have used a template called **Angular** which offers a collection of ready-to-use styles and tools based on AngularJS and many javascript libraries. A more detailed description of the Angular features can be found in the next section.

## 4. Frontend Development

### 4.1. Tools

We have used the Angle template as a starting point for the development of the frontend application. This template includes AngularJS, jQuery and many other libraries. It offers also some useful predefined Angular directives and services. In order to understand the structure of this template and be able to customize it, a good understanding of AngularJS architecture and terms is essential. The most important tools used to build the frontend application are the following:

#### 4.1.1. AngularJS



**AngularJS** is an open-source web application framework maintained by Google and by a community of developers and corporations to address many of the challenges encountered when developing single-page applications. It aims to simplify both the development and the testing of such applications by providing a framework for client-side web application which is able to add new reusable features to HTML.

Official Website: <https://angularjs.org>

#### 4.1.2. jQuery



The usage of AngularJS does not eliminate jQuery which is the most used javascript library in web applications today. It is designed to simplify document navigation, DOM (Document Object Model) elements selection, animations, events handling, and Ajax (Asynchronous JavaScript and XML) calls. jQuery also provides capabilities for developers to create plug-ins on top of the JavaScript library. This enables developers to create abstractions for low-level interaction and animation, advanced effects and high-level, theme-able widgets.

Official Website: <https://jquery.com>

#### 4.1.3. Git



Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency. It helps to keep track of every change made on the code and rollback when needed. It is also the best way to work on group on the same project.

Official Website: <https://git-scm.com>



#### 4.1.4. npm



npm is package manager for JavaScript, and is the default one for NodeJS packages. NodeJS gives the possibility to run the javascript on the server side making it possible to write javascript files and run them from the terminal.

After the release of NodeJS on 2009, developers started making reusable node modules and the **node package manager** (npm) was created on 2011 to make it easy to share and reuse node modules. But now it is used to manage all types of javascript modules and packages.

Official Website: <https://www.npmjs.com>

npm is used in our application to manage packages such as Bower and Gulp.

#### 4.1.5. Bower



Bower is a package manager for frontend components such as CSS and javascript libraries. It is installed using npm as it is a node module. Bower does not replace npm. npm manages server side packages will bower manages frontend packages.

Official Website: <https://www.bower.io>

#### 4.1.6. Gulp



Gulp is a javascript task runner or build tool. It can run several tasks automatically to improve the developer workflow. Tasks are written in javascript and can do various jobs from concatenating multiple files into one to compiling a Less source code (the less language is presented below) and compressing the resulting CSS code and storing it in a destination file.

Gulp uses Streams (the NodeJS objects representing a buffer of data) during the task operations. So there is no need to store intermediate results in temporary files. It also runs the tasks in parallel which is more efficient. Dependencies between tasks can be declared to force a task not to start before the end of an other one. Gulp can also watch files and run corresponding tasks when a file is changed.

Official Website: <https://www.gulpjs.com>

#### 4.1.7. jade



Jade is a templating language which produces HTML code from a less verbose syntax. The Jade syntax is more easy to write, read and maintain. More than that, the Jade language has many features which make the code dynamic and reusable.

Official Website: <https://www.jade-lang.com>



All the HTML files in our frontend application are written using Jade and compiled by Gulp.

### 4.1.8. less



Less is a CSS pre-processor, meaning that it extends the CSS language, adding features that allow variables, mixins, functions and many other techniques that allow the developer to make CSS that is more maintainable, themable and extendable.

There are also many 3rd party tools that help to compile less files and watch for changes. Gulp is one of these tools.

Official Website: <https://www.lesscss.org>

## 4.2. Files structure

After modifications made to the initial template, the files structure of the application is the following:

```
bash
app/           # the build folder
master/        # the sources folder
|-- bower_components/ # bower installed packages
|-- jade/         # the core application views
|-- js/           # the core application js files
|-- less/         # the core application styles
|-- modules/      # modules folder
|-- node_modules/ # NodeJs installed modules
|-- bower.json    # bower manifest file
|-- gulpfile.js   # Gulp tasks definition
|-- package.json  # npm manifest file
|-- vendor.base.json # packages to load with the application
|-- vendor.json   # packages that will be loaded when needed
vendor/         # the libraries folder
index.html
```

Figure 2: The files structure of the frontend application

All the application source code is inside the master directory. Under this directory we find:

**bower\_components:** When we install a package using Bower, it is stored into this directory. Then when Gulp is run, it takes all needed Javascript and CSS files from this directory, minify them (by removing all optional white spaces and renaming variables and functions with short names in order to reduce the file size) and store them into the libraries directory named **vendor**.

Gulp knows which files are needed from the `vendor.base.json` and `vendor.json`. `vendor.base.json` lists the packages to be loaded with the core application (like Angular and jQuery). `vendor.json` lists other needed packages, which will be loaded specifically depending on the visited module.

**jade:** The core view files are stored into this directory. Views are written using the Jade language and compiled into HTML files by Gulp. The compiled HTML files are stored into the `app/views` directory.

**js:** This directory contains all the core javascript files of the application. Since AngularJS is used to build this application; we find controllers, directives and services directories inside the `js` one following the Angular conventions. Additional javascript classes are stored inside the `js/classes` folder. One of this classes is the Module class which will be used to create modules for the application.

**less:** This directory holds all application style definitions. They are written using a CSS pre-processor called Less. Gulp will compile all this files into CSS files and store them in `app/css` directory.

**modules:** This directory contains source code of modules. the module structure is detailed below.

**node\_modules:** Nodejs modules installed using `npm install` will be stored into this directory.

**gulpfile.js:** This file contains the Gulp tasks definitions.

## 4.3. Tools installation

**1. Setup NodeJS and npm:** installation instructions for different systems are described in this link: <https://nodejs.org/download>

In my case, I am using Ubuntu 14.04 and the setup commands are:

```
bash
# adding the NodeSource PPA
curl --silent --location https://deb.nodesource.com/setup_0.12 | sudo bash -
# update packages list
sudo apt-get update
# installing NodeJS (npm is installed with it automatically)
sudo apt-get install --yes nodejs
```

Figure 3: NodeJS and npm installation commands on Ubuntu

**2. Setup Git:** Git can be installed on Ubuntu from the default package manager:

```
sudo apt-get install git
```

bash

Figure 4: Git installation command on Ubuntu

**3. Setup Bower and Gulp:** Bower and Gulp are installed using npm. We install them globally to be able to use their commands from any directory:

```
sudo npm install -g bower  
sudo npm install -g gulp
```

bash

Figure 5: Bower and Gulp installation commands

**4. Setup the application dependencies:** Inside the `master` directory, running the following commands will install all the needed dependencies for the application to work. The list of dependencies is read from the file `package.json` for npm and `bower.json` for Bower.

```
sudo npm install  
bower install
```

bash

Figure 6: Dependencies installation commands

## 4.4. Building and running the application

The application source code is built using Gulp to produce HTML, CSS and javascript files and store them into the `app` folder. Gulp also runs a local server into this folder so that the developer can see the resulting application in the browser. After the build process, Gulp watches all source files. When a source file is changed and saved, Gulp re-compiles this file and refreshes the application page on the browser. This way, the developer can change the source code and see the result immediately in the browser.

### 4.4.1. The gulpfile

The file `gulpfile.js` contains the declaration of Gulp tasks. Declaring a task in this file can be done as follows:

```
var gulp = require('gulp'); // requiring the gulp module
gulp.task('copy', function(){
  gulp.src('folder1/*')
    .pipe(gulp.dest('folder2'));
});
```

javascript

Figure 7: Gulp task declaration example

In the code above, we created a task called `copy` to copy all files under the `folder1` into `folder2`. To run this task from the command-line, we type:

```
gulp copy
```

bash

### 4.4.2. Gulp tasks

In our application, the main Gulp tasks are the following:

#### **scripts:vendor**

This task runs two sub tasks **scripts:vendor:base** and **scripts:vendor:app**. The first one minifies and concatenates all the base vendor files (read from `master/vendor.base.json`) into the file `app/js/base.js`. The second one reads the list of additional vendor files from `master/vendor.json` and copies them into the folder `vendor`.

#### **scripts:app**

This task minifies and concatenates all the core application javascript files into `app/js/app.js`.

#### **styles:app, styles:themes and bootstrap**

Those are stylings tasks. The **bootstrap** task compiles the Less source code of the bootstrap library. The two other tasks compile the core application Less files. The resulting CSS files are stored into `app/css`.

#### **templates:app, templates:views and templates:pages**

Those tasks compile the Jade source files of the index, views and pages respectively and store the resulting HTML files into `app`, `app/views` and `app/pages` respectively.

### **modules:scripts**

This task concatenates all the javascript files of each module and stores them under `app/modules/module_name/all.js`

### **modules:styles**

This task compiles the less file of each module and stores the result into `app/modules/module_name/style.css`

### **modules:views**

This task compiles all jade files under the views directory of each module and stores the resulting HTML files into `app/modules/module_name/views`

### **watch**

This task watches all source files for changes and runs the corresponding task when a file is modified.

### **connect**

This task runs a local webserver into the app directory to see the results in the browser.

### **start**

This task compiles the application without vendors ( by running the other tasks ) and starts the server by running the `connect` task.

### **default**

This task compiles the application and vendors and starts the server by running the `connect` task. This task is run automatically when we execute the command "gulp" in the master directory.

## 4.5. Basics of AngularJS

AngularJS is a very powerful framework for client-side applications. In this section, I will try to cover the minimum required basics in order to understand how the frontend application is built and be able to add new modules to it.

### 4.5.1. Starting example

A very simple example of using AngularJS could be the following:

```
markup
<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title> Angular Sample </title>
</head>
<body data-ng-app="demo">
  <p> Your name : <input type="text" ng-model="name"></p>
  <p> Hello {{ name }} </p>

  <script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js">
  </script>
  <script>
    var app = angular.module('demo', []);
  </script>
</body>
</html>
```

Figure 8: Starter example of using AngularJS

In the code above, we included the AngularJS javascript file from Google and we defined a new angular module called "demo". Please note that we put the scripts at the end of the body element to speed up the page loading. The attribute `data-ng-app` of the body element tells AngularJS which angular module to be applied to the page.

We defined a text input and set the attribute `ng-model` to "name". This will define a variable called "name" and bind it's value with the content of the input. So if we type some text in this input, the value of the variable name will be set to this text. Finally we show the value of the variable name using the syntax `{{name}}`. This example shows the data binding feature of AngularJS, so by changing the value of the input, the text in the next paragraph is changed too.

In the following, you find short definitions and examples of the most important AngularJS terms.

## 4.5.2. Angular Module

A module is a reusable collection of the application parts ( controllers, services, directives, ...). An application can use one or multiple modules. When declaring a module, we can specify the list of dependent module that this module will use. The parts of the related modules could be injected by AngularJS automatically and used.

### Example

```
// declaration of new angular module
var app = angular.module('myApp', ['module1', 'module2']);
```

javascript

Figure 9: AngularJS module declaration example

Here we declared a module called "myApp" which depends on "module1" and "module2".

## 4.5.3. Controller and Scope

A controller is a javascript function that handles a view or a part of the page. It can bind variables and events to this part of the page. The variables binded to a view are held into an object called the scope. This object can be injected automatically into the controller simply by adding `$scope` to the function arguments.

### Example

```
// using the app variable which is the module
app.controller('HomeController', function($scope){
  // use the scope to define binded variables
  // A name variable for example
  $scope.name = 'LHCbPR';
  // or an array
  $scope.tools = ['AngularJS', 'Bower', 'Gulp'];
});
```

javascript

Figure 10: AngularJS Controller example

The code above will not work after minification. A recommended way is to declare dependencies into a list of strings and add them to the function arguments in the same order like this:

```
app.controller('HomeController', ['$scope', function($scope) {  
    // ...  
}]);
```

To use the controller in the view, we first declare it using `ng-controller` then use the variables and methods of the scope inside this view like the following:

```
<div ng-controller="HomeController">  
  <p> The {{name}} web application is built using: </p>  
  <ul>  
    <li ng-repeat="element in tools"> {{ element }} </li>  
  </ul>  
</div>
```

Figure 11: Using Angular Controller in the view

The attribute `ng-repeat` we added to the `<li>` element is a predefined angular directive which loops over a array and repeat the element for each item of the array. The resulting view will be something like:

The LHCbPR web application is built using:

- AngularJS
- Bower
- Gulp

Figure 12: Sample View



## 4.5.4. Directives

A directive is a new element or behavior added to the HTML. `ng-controller` and `ng-repeat` are examples of directives which are used as attributes. When defining a directive we specify whether it will be used as an element (an HTML tag), an attribute or a class. This choice depends on the feature added by the directive:

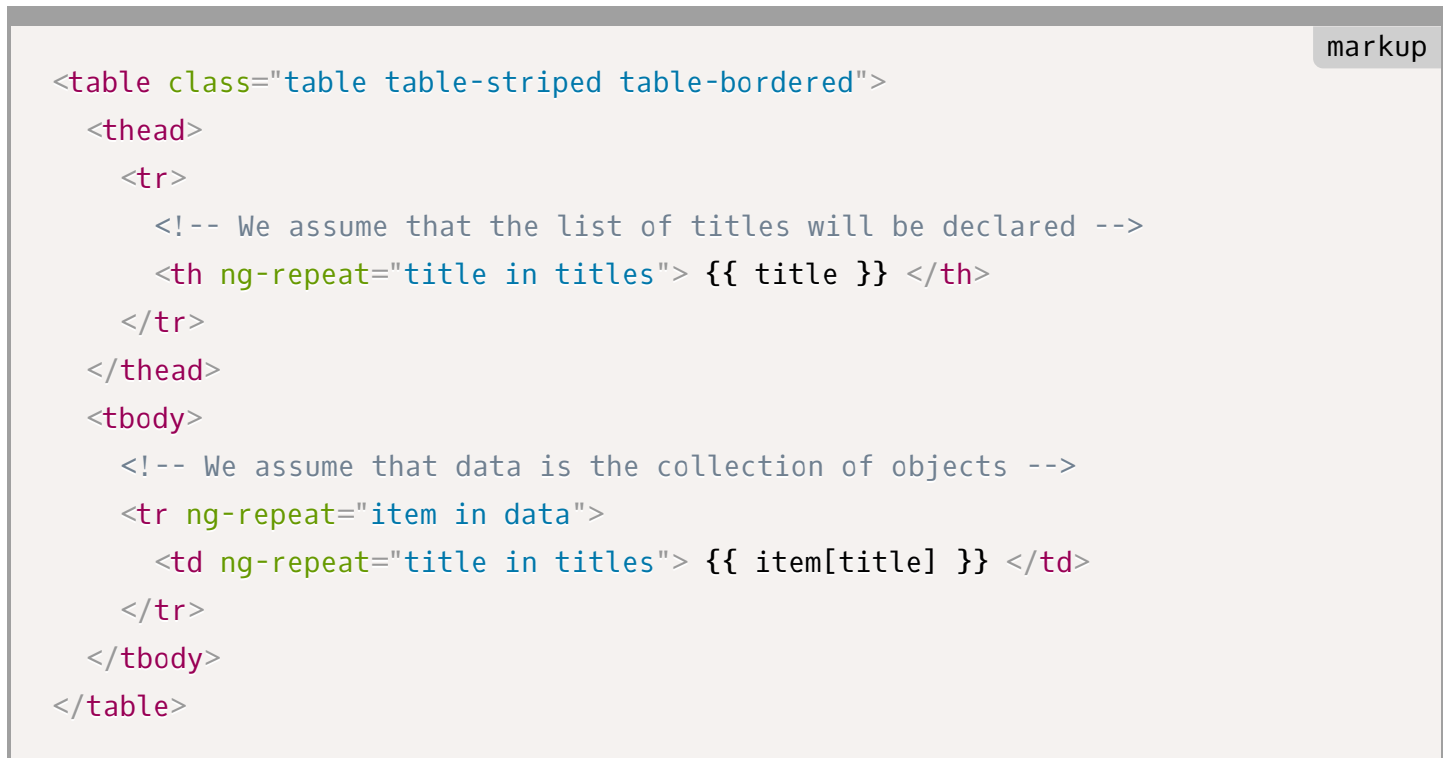
A directive which is replaced by a template or view is more likely to be used as an element.

A directive which adds a behavior to HTML elements based on an input is more likely to be used as an attribute.

A directive which adds a generic decoration or behavior to HTML elements is more likely to be used as a class.

These rules remain subjective as AngularJS enables the developer to give multiple uses to the same directive ( a directive that can be used as an element and as an attribute for example).

For example, let us define a directive that takes a collection of objects and shows them as an HTML table. First we will define the template as follows (note that we are using some predefined CSS classes):



```
<table class="table table-striped table-bordered">
  <thead>
    <tr>
      <!-- We assume that the list of titles will be declared -->
      <th ng-repeat="title in titles"> {{ title }} </th>
    </tr>
  </thead>
  <tbody>
    <!-- We assume that data is the collection of objects -->
    <tr ng-repeat="item in data">
      <td ng-repeat="title in titles"> {{ item[title] }} </td>
    </tr>
  </tbody>
</table>
```

Figure 13: Data table sample directive template

Then we define the directive as follows:

javascript

```
app.directive('myTable', function(){
  // The directive consists on an object that this function should return
  return {
    restrict: 'E', // define how the directive will be used
    // 'E' for element, 'A' for attribute and 'C' for class
    // 'EA' means element or attribute
    templateUrl: 'data-table.html', // the template file
    // if specified, the content of the HTML element to which the directive
    // is applied will be overitten by this template
    scope: { // contains the data passed to the directive as attributes
      data: '='
    },
    // This function is executed when the directive is rendered
    link: function(scope){
      // scope.data will refer the data passed to the directive
      // Defining titles of the table as the properties of the first
      // object in data, as we assume all objects have the same properties
      scope.titles = [];
      if(scope.data !== undefined && scope.data.length > 0){
        for(attr in scope.data[0]){
          scope.titles.push(attr);
        }
      }
    }
  };
});
```

Figure 14: Data table sample directive code

Now assuming that we have a collection of objects in the scope like this:

javascript

```
$scope.collection = [
  { Experiment: 'CMS', 'LHC experiment ?': 'Yes', Type: 'General' },
  { Experiment: 'ATLAS', 'LHC experiment ?': 'Yes', Type: 'General' },
  { Experiment: 'LHCb', 'LHC experiment ?': 'Yes', Type: 'Specific' },
  { Experiment: 'ALICE', 'LHC experiment ?': 'Yes', Type: 'Specific' }
];
```

We can use the directive in the view (the HTML code) as follows:

```
<my-table data="collection"></my-table>
```

markup

The result is a table of the given data:

Experiment	LHC experiment ?	Type
CMS	Yes	General
ATLAS	Yes	General
LHCb	Yes	Specific
ALICE	Yes	Specific

Figure 15: Data table sample directive result

### 4.5.5. Services

A service is an object which is instantiated once (Singleton) and injected into components using it (like controllers and directives). Service is a good way to share functionalities between different components of the application.

AngularJS provides several predefined services like `$http` which handles AJAX requests. To use this service inside a controller for example, all we have to do is to add it to the arguments:

```
app.controller('MyController', ['$scope', '$http', function($scope, $http){  
    // Now $http can be used here  
}]);
```

javascript

Figure 16: Example of injecting `$http` service

In our application, we are using a service called **Restangular**. It is built on top of `$http` and simplifies the communication with a REST API.

### 4.5.6. Routing

the routing functionality in AngularJS helps to create single-page web applications. The `$routeProvider` contained in the `ngRoute` module is used to configure routes of the application. A route definition consists on the definition of an URL pattern with the corresponding view and controller. Routes are configured using the `config()` method as follows:

```
// The module depends on the ngRoute module
// because we need to use $routeProvider
var app = angular.module('demo', ['ngRoute']);
// Routes configuration
app.config(['$routeProvider', function($routeProvider) {
  $routeProvider
    .when('/home', {
      templateUrl: 'home.html',
      controller: 'HomeController'
    })
    .when('/experiments/:name', {
      templateUrl: 'experiment.html',
      controller: 'ExperimentController'
    })
    .otherwise('/home');
}]);
```

Figure 17: Routes configuration example

Here we defined two routes. The first route for the home page. The second route for the experiment page. Note that the URL pattern of the second route contains a parameter called `:name` which will be matched as an alpha-numeric string. A parameter in the URL pattern always starts with ":" and its value can be retrieved using the `$routeParams` service. The default URL to apply when no pattern is matched is defined with the `otherwise()` method. In the previous code, if no pattern is matched, the user will be redirected to the home page.

In addition to `home.html` and `experiment.html` mentioned in the route definitions, we need to create `index.html` which is the single page to be loaded directly by the browser. Inside this page, we use the `ng-view` directive to insert the view corresponding to the URL pattern.

Assuming that the link of our application is "http://demo.com/", when this link is visited, the browser will load the `index.html` file and then load all javascript and CSS files referenced inside it. The browser will then run the javascript code. In particular, the routes will be defined and angular will match the URL and find the current route. The URL of the route is the part of link after the first "#" character. This character is used because the browser does not reload the page when the string after "#" is changed; which enables Angular to handle this part of the link and load views via AJAX.

When the link does not contain any "#", the route URL is supposed empty. In this case, Angular will redirect us to the home page by appending "#/home" to the link and loading `home.html` inside the `ng-view` directive. In the same way, visiting "http://demo.com/#/experiment/LHCb" will load the `experiment.html` and the value of the parameter name inside the `ExperimentController` will be "LHCb".

## 4.6. Frontend core application features

Many AngularJS modules are used in our frontend application to provide module developers with useful functionalities. The most important functionalities and their use case are presented below.

### 4.6.1. States and Lazy loading

Our application contains many analysis modules. To map URLs to modules we are using **states**. States are like routes with the ability to be nested. This way a module of the application is described by a state and can have sub states. The states feature is offered by **AngularUI Router** which is an AngularJS module providing `$state` service and `$stateProvider` to handle states.

When declaring a state, in addition to the view and the controller, a list of dependencies can be provided. These dependencies will be loaded with the state. A dependency is a name referencing a collection of javascript and CSS files. The file `vendor.json` contains the starting list of dependencies. The process of loading these files only when needed is called `Lazy loading` which makes the application faster. The application analysis modules are added automatically to the list of dependencies so that one module can depend on an other.

The code to define states will be presented in the next section.

### 4.6.2. LHCbPR service

All analysis modules need to communicate with the REST API to retrieve data from the database and present it. The frontend application contains the `lhcbprResources` service which extends the Restangular service and simplifies sending and requesting data from the REST webservice. Here is an example retrieving the list of active applications:

```
app.controller('TestController', [
  '$scope', 'lhcbrResources', function($scope, lhcbrResources)
  {
    // Get the list of active applications
    lhcbrResources.all('active/applications')
      .getList()
      .then(function(response){
        // Do what ever you want with the response
        // This will show the list on the console
        console.log(response);
      });

    // Some other code here
    // Please note that the AJAX calls are asynchronous
    // which means that the code here maybe executed while
    // waiting for the response
  }]);
```

Figure 18: Example of requesting data from the REST API

### 4.6.3. Directives

The core application contains a collection of predefined directives that can be used in the analysis modules development. The most important directives are presented in the following.

#### 4.6.3.1. DataTables

the `ngTable` directive can be used to insert a table of data with sorting, filtering and pagination features. For example, to insert a table of jobs, we add the following code in the view (note that this is Jade source code; all analysis modules views are written using Jade which is less verbose than HTML):

```

table.table.table-striped.table-bordered.table-hover.be-responsive(
  ng-table="tableParams")
thead: tr
  th ID
  th Name
tbody
  tr(ng-repeat="attr in $data")
    td {{attr.id}}
    td {{attr.name}}

```

Figure 19: Example of ngTable directive: the view code

Then we add the following code in the controller:

```

App.controller('TestController', [
  '$scope', 'ngTableParams', function($scope, ngTableParams)
{
  // We define some hard coded data
  $scope.attrs = [
    { id: 5, name: "EVENT_LOOP" },
    { id: 6, name: "EVENT_LOOP_count" },
    { id: 7, name: "EVENT_LOOP_rank" }
  ];

  // Table parameters
  $scope.tableParams = new ngTableParams(
    { page: 1, count: 10 },
    { total: 0, getData: function($defer, params) {
      // We just show the hard coded values
      $defer.resolve($scope.attrs);
    } });

  // Add this line to fix a bug in the ng-table directive
  $scope.tableParams.settings().$scope = $scope;
}]);

```

Figure 20: Example of ngTable directive: the controller code

The result is something like this:

ID	Name
5	EVENT_LOOP
6	EVENT_LOOP_count
7	EVENT_LOOP_rank

Figure 21: Example of Data Table

### 4.6.3.2. Charts

Many charts directives are included and can be used in the analysis modules. These chart directives are based on five javascript libraries. The library that was used in developing charts on the analysis modules is **ChartJS** which is flexible and provides many types of charts. Exemples of charts provided by this library are the following:



Figure 22: Examples of predefined chart directives



### 4.6.3.3. Search Jobs

This is the most used directive in analysis modules. It shows a form in which the user can filter jobs by application, options and versions. And notifies the controller each time the selection changes.

To insert the directive in the view, we use the Jade code below:

```
search-jobs(on-found="updateJobs(searchParams)")
```

jade

Figure 23: Example of using search-jobs directive: the view code

And then we define the callback function in the controller as follows:

```
$scope.updateJobs = function(params){  
    // You can use the selected parameters:  
    // params.apps[0]: the selected application id  
    // params.options: array of selected options ids  
    // params.versions: array of selected versions ids  
};
```

javascript

Figure 24: Example of using search-jobs directive: the controller code

This directive binds selected parameters to the URL on every change which means that visiting the generated URL will select these parameters automatically.

## 5. Application Modules Development

The frontend interface contains three main parts:

A header containing the logo and loading icon displayed when data is being loaded from the API.

A sidebar containing the list of analysis modules.

A content area showing the selected module.

### 5.1. Files structure

All modules are stored under the `master/modules` directory. Each module have the following files structure:

```
module_name/
|-- js/           # the javascript files
|   |-- controllers/ # controllers
|   |-- directives/ # custom directives
|   |-- init.js     # initialization file
|-- less/
|   |-- style.less  # custom styles
|-- views/         # views
```

Figure 25: Module files structure

`js/init.js` contains the definition of the module. Here we describe what are the states of module and what are links to add on the sidebar.

`js/controllers/` This directory contains the controllers of the module. Each controller is written on it's own javascript file.

`js/directives/` This directory contains additional directives defined for this analysis module. This directory is optional. Other directories like services or filters could be added to contain custom components used within the module.

`less/style.less` This file contains custom styles code for the module elements.

`views/` This directory contains the views of the module. Each view is defined by a Jade file.

## 5.2. Helper classes

In order to make the modules development process more simple. I added some helper classes to hide a part of the Angular complexity.

### 5.2.1. Module class

The module class simplifies the declaration of new modules. It offers the following methods:

#### **Module.create(name, title, position, settings)**

**name** The name of the module, it should be unique for each module.

**title** The title of the module menu in the sidebar.

**position** The position of the module menu in the sidebar. If two modules have the same position, there are shown one under the other.

**settings** An optional plain object specifying the folder name of the module.

```
Module.create('name', 'Title', 1, {  
  folder: 'my_folder_name'  
});
```

javascript

If no settings is provided; the folder name is assumed to be the same as the module name.

This method returns an instance of the **Module** class.

#### **addMenuItems(items)**

**items** Menu object or array of menu objects. A menu object has the following format:

```
{  
  text: "Title", // Text to show on the sidebar  
  sref: "app.state.name", // Target state name prefixed by 'app.'  
  icon: "icon-grid", // Icon of the menu item  
  alert: "new" // Added a budge to the menu item  
}
```

javascript

This method returns the same calling Module instance so that we can chain calls to other methods.

## addStates(states)

**states** State object or array of state objects. A state object has the following format:

```
javascript
{
  name: 'test.state_one', // Required
  url: '/my-url', // If not provided, it will be defined
  // based on the name ('/state-one' for this example)
  title: 'Title', // If not provided, it will be defined
  // based on the name ('State One' for this example)
  templateUrl: 'view.html', // name of the view file
  // with html extension instead of jade. If not provided
  // the view name will be 'state-one.html' in this case.
  controller: 'MyController', // name of the controller
  // handling the state. If not provided, it will be
  // 'TestStateOneController' for this case.
  resolve: ['test', 'chartjs'] // names of dependencies
  // on which this state depends. The name of its own
  // module should be part of the dependencies.
}
```

This method returns the same calling Module instance so that we can chain calls to other methods.

## start()

This method should be called once all the desired menu items and states were added to the module. It executes the corresponding code on the AngularJS instance and integrate the module with the rest of the application.

### 5.2.2. Dependencies class

This class is named **Deps** and it handles the list of dependencies which are loaded with modules on request. The Module class uses this one to know which files to load for each dependency name. The main methods of this class are:

**Deps.addLibraries(items), Desps.addAngularModules(items) and Deps.addModules(items)**

These methods are used to add dependencies. The **items** arguments in a dependency or a list of dependencies. A dependency has the following structure:

```
{
  name: "The dependency name",
  files: [
    "my-dependency.css",
    "my-dependency.js"
  ]
}
```

### **Deps.get(name)**

This method gets the list of files of a module based on the name. If no module with the given name is found, the method returns `null`.

### 5.2.3. Colors class

This class makes it simple to request colors and use them to draw charts without having to write the HEX or RGB code of colors. It also gives the possibility to make a color darker or lighter by a percentage. Its main methods are:

**Colors.add(name, hex):** add a color with the specific name and HEX code.

**Colors.get(name [, percentage]):** gets the hex code of the color. A percentage can optionally be applied on the color. If the percentage < 1 the color becomes darker; else it becomes lighter.

**Colors.lighten(hex [, percentage]):** make a color lighter based on a percentage. The percentage is 0.25 by default.

**Colors.darken(hex [, percentage]):** make a color darker based on a percentage. The percentage is 0.25 by default.

## 5.3. Step by step example: the Trends analysis module

In this part I will explain how the Trends analysis modules (one of the modules I created for the frontend application) was made step by step.

### 5.3.1. Use Cases

The first step is to define the features or use cases of our module. The use cases for the Trends module are the following:

User should select an application and possibly multiple options and versions.

On every change of the selection; We have to retrieve all the job results of jobs from the selected application, options and version. Then a table should show the list of attributes found on these results.

User should be able to filter attributes by name.

User should be able to show the Trends plot (showing the mean and deviation by version) of any attribute by clicking on the corresponding button.

If a plot is shown and the user copied the link. Visiting that link again should re-produce the plot with the same selected application, options and version for the same attribute.

### 5.3.2. Files structure

This module will have only one state, so one view and one controller; the files structure can be like this:

```
bash
master
|-- modules
|   |-- demo/
|       |-- js/
|           |-- controllers/
|               |-- demo.js
|           |-- init.js
|       |-- less/
|           |-- style.less
|       |-- views/
|           |-- demo.jade
```

Figure 26: Trends analysis module files structure

### 5.3.3. Module declaration

The module declaration is written on the file `js/init.js` :

```
javascript
Module.create('demo', 'Demo', 2)
  .addMenuItems({
    text: "Demo",
    sref: "app.demo",
    icon: "icon-grid"
  })
  .addState({
    name: 'demo',
    resolve: ['demo', 'chartjs', 'ngTable', 'ngDialog']
  })
  .start();
```

According to the previous code, we are creating a module named demo having one menu item pointing to the `app.demo` state. The only state of the module is named demo. Note that the menu item is pointing to `app.demo` while the state's name is just demo. That's fine because the Module class prefixes all state names with "app.". This module depends on:

**chartjs:** used to draw plots

**ngTable:** used to show the attributes table

**ngDialog:** used to create the popup showing the plot.

### 5.3.4. Adding Search Form

Now let's write this code on the view to show the search form:

```
jade
h3 Trends
  small.text-muted Attributes values by version
  .row.traditional(ng-class="cssspinner")
    .col-lg-4
      search-jobs(on-found="requestStatistics(searchParams)")
    .col-lg-8
      p The attributes table will be added here !
```

As you see, we have told the `search-jobs` directive to call the function `requestStatistics()` on every change; So we have to define this function in the controller. So for the moment the controller's code will be like this:

```
javascript
App.controller('DemoController', ['$scope', function($scope) {
  $scope.appId = undefined;
  $scope.options = undefined;
  $scope.versions = undefined;

  $scope.requestStatistics = function(params) {
    $scope.appId = params.apps[0];
    $scope.options = params.options;
    $scope.versions = params.versions;
  };
}]);
```

On every change, we are saving the selected parameters in the variables `$scope.appId`, `$scope.options` and `$scope.versions`.

### 5.3.5. Showing Attributes on a table

Let's add the attributes table to our view; the code will become like this:

```
h3 Trends
  small.text-muted Attributes values by version
  .row.traditional(ng-class="cssspinner")
  .col-lg-4
    search-jobs(ng-found="requestStatistics(searchParams)")
  .col-lg-8
    table.table.table-striped.table-bordered.table-hover.be-responsive(ng-table="attrTableParams")
      thead
        th ID
        th Name
        th
      tbody
        tr(ng-repeat="a in $data")
          td
          td
          td
        button.btn.btn-primary(ng-click="showChart(a)") Show
```

We applied the `ng-table` directive to our table using `attrTableParams` variable which will be defined in the controller. Each row of the table shows the id of the attribute, its name and a button to show its corresponding plot. Note that we pass the attribute to the function `showChart(a)` when the button is clicked using the `ng-click` directive.

Now let's add the variable `attrTableParams` and the function `showChart` to the controller. First of all, we need to inject the `ngTableParams` and `lhcbprResources` services into our controller to be able to use them; we do this by Adding them to controller parameters like this:

```
App.controller('DemoController', ['$scope', 'ngTableParams', 'lhcbprResources',
  function($scope, $tableParams, $api) {
```

Note that we add the `ngTableParams` and `lhcbprResources` to the array and to the function arguments. The order of arguments should be the same as in the array. So here the argument `$scope` is referencing the `$scope` service, `$tableParams` is referencing the `ngTableParams` and `$api` is referencing `lhcbprResources`.



Now we can use `$tableParams` to define `attrsTableParams` like this:

javascript

```
$scope.attrsTableParams = new $tableParams(  
  {  
    page: 1, // the page to show initially  
    count: 10 // number of rows on each page  
  },  
  {  
    total: 0, // total number of rows initially  
    // function that fetchs data to fill the table  
    getData: function($defer, params) {  
      // We check if an application and options were selected  
      if($scope.appId && $scope.options){  
        // We construct the data to send with the request to the API  
        var requestParams = {  
          app: $scope.appId,  
          options: $scope.options,  
          versions: $scope.versions,  
          page: params.page(),  
          page_size: params.count()  
        };  
        // TODO: The filter code will be added here  
        // We send the request to the '/trends' url of the API  
        $api.all('trends')  
          .getList(requestParams)  
          .then(function(trends){ // When we receive the response  
            // We set the total number of rows  
            if(trends._resultmeta){  
              params.total(trends._resultmeta.count);  
            }  
            // We fill the table with the response  
            $defer.resolve(trends);  
            // TODO: the code to show the plot based on the URL will be added  
          });  
      }  
    }  
  }  
);  
// We add this line of code to fix a bug in the ngTable service  
$scope.attrsTableParams.settings().$scope = $scope;
```

After this, we need a function which updates the table's content after every change of the selected application, options or versions.

```
javascript
$scope.update = function(){
    // Set the current page of the table to the first page
    $scope.attrsTableParams.page(1);
    // reloading data
    $scope.attrsTableParams.reload();
}
```

Now we go back to the function `requestStatistics` and call the function `update` after every change:

```
javascript
$scope.requestStatistics = function(params) {
    $scope.appId = params.apps[0];
    $scope.options = params.options;
    $scope.versions = params.versions;
    $scope.update();
};
```

Finally let's add the `showChart` function:

```
javascript
$scope.showChart = function(a){
    console.log('Showing chart of ' + a.name);
};
```

This function just prints the message "Showing chart of [attribute name]" on the console for the moment. We will add the Chart code below.

### 5.3.6. Filtering attributes by name

Now let's add a text input to filter attributes by name. We start by adding a form containing one input before the table on the view; the code of the view becomes as follows:

### Trends

```

    small.text-muted Attributes values by version
    .row.traditional(ng-class="cssspinner")
    .col-lg-4
        search-jobs(ng-found="requestStatistics(searchParams)")
    .col-lg-8
        form
            .form-group
                label Filter Attributes
                input.form-control(ng-model='attrFilter', ng-change='update()', type='text')
            table.table.table-striped.table-bordered.table-hover.be-responsive(ng-table="att
            thead
                th ID
                th Name
                th
            tbody
                tr(ng-repeat="a in $data")
                    td
                    td
                    td
                button.btn.btn-primary(ng-click="showChart(a)") Show

```

Please note that in this line:

```
input.form-control(ng-model='attrFilter', ng-change='update()', type='text')
```

We bind the value of the input to the scope variable named `attrFilter` and we call `update()` on every change.

Let's go to the controller and replace this comment on the `getData` function:

```
// TODO: The filter code will be added here
```

With this piece of code:

```

// remove additional spaces from the input value
$scope.attrFilter = $scope.attrFilter.trim();
// if the value is not empty; we add it to the request parameters
if($scope.attrFilter && $scope.attrFilter != '')
    requestParams.attr_filter = $scope.attrFilter;

```

### 5.3.7. Showing the Chart

Till now, when we click the "Show" button to show the chart; nothing happens. Let's fix this.

The plot will be shown into a ngDialog (which is a popup). So we have to define the content of this popup. One of the simple way to do it is to include it in the view as an inline template by adding this code at the end of the view code:

```
jade
script(type="text/ng-template", id="chartTemplate")
  .row
    .col-lg-10.col-lg-offset-1.col-md-10.col-md-offset-1.col-sx-10.col-sx-offset-1
      h2
    .chart
      canvas(linechart='', options='lineOptions', data='lineData', height='chartHe:

```

In this code, we defined an inline template with the name "chartTemplate". The plot will be shown into the canvas element because we applied the directive linechart to it. This directive needs two properties options and data which we filled with the scope variables `lineOptions` and `lineData` ; We will define these variables in the controller. We have also specified the width and height of the chart using the scope functions `chartWidth` and `chartHeight` that will be defined on the controller too.

Now we move to the controller and add the following code:

```
javascript
$scope.lineOptions = {
  animation: false,
  errorDir : "both",
  errorStrokeWidth : 3,
  datasetFill : false,
  scaleOverride : true,
  scaleSteps : 10,
  scaleStepWidth : 1,
  scaleStartValue : 0,
  tooltipTemplate: "<%if (label){%><%=label%>: <%}%><%= value %><%if (errorBar
  legendTemplate : '<% for (var i=0; i<%if(datasets[i].label){%><%=datasets[i].lab
};
$scope.chartHeight = function() {
  return $(window).height() - 140;
};
$scope.chartWidth = function() {
  return $(window).width() - 60;
};

```

Then we write the code of the showChart function:

javascript

```
$scope.showChart = function(a){
    console.log('Show chart of ' + a.name);
    // We store versions, averages and deviations in separated tables
    // with the same order and we compute the min and max values
    var minValue = a.values[0].average - a.values[0].deviation,
        maxValue = a.values[0].average + a.values[0].deviation,
        value = 0,
        versions = [],
        averages = [],
        deviations = [];
    a.values.forEach(function(v){
        v.average = parseInt(100 * v.average) / 100.0;
        v.deviation = parseInt(100 * v.deviation) / 100.0;
        averages.push(v.average);
        versions.push(v.version);
        deviations.push(v.deviation);
        value = v.average - v.deviation;
        if(value < minValue)
            minValue = value;
        value = v.average + v.deviation;
        if(value > maxValue)
            maxValue = value;
    });
    minValue = Math.floor(minValue);
    maxValue = Math.floor(maxValue) + 1;
    // We configure the chart to show only values between minValue and maxValue
    $scope.lineOptions.scaleStartValue = minValue;
    $scope.lineOptions.scaleSteps = maxValue - minValue;
    $scope.lineOptions.scaleStepWidth = 1;
    while ( $scope.lineOptions.scaleSteps > 25 ){
        $scope.lineOptions.scaleSteps /= 2;
        $scope.lineOptions.scaleStepWidth *= 2;
    }
}
```

```
// ...
// We set the data to show on the chart
$scope.lineData = {
  labels: versions,
  datasets: [{
    label: a.name,
    fillColor : "rgba(220,220,220,0.2)",
    strokeColor : "#2F49B1",
    pointColor : "#5E87D6",
    pointStrokeColor : "#fff",
    pointHighlightFill : "#fff",
    pointHighlightStroke : "#5E87D6",
    data: averages,
    error: deviations
  }]
};
// We set the name of the attribute
$scope.name = a.name;
// We show the popup
$dialog.open({
  template: 'chartTemplate',
  className: 'chart-dialog',
  scope: $scope
});
}
```

### 5.3.8. Binding selected values to the URL

When we show the chart, we need to add the id of the shown attribute to the URL, and when the popup is closed we just remove this information from the URL. To do this, we need the `$location` service that we will inject into our controller:

```
App.controller('DemoController', ['$scope', 'ngTableParams', 'lhcbrResources', '$location',
```

Then we change the last part of showChart function:

```

$scope.showChart = function(a){
    // ... code here remains the same
    // We show the popup
    $dialog.open({
        template: 'chartTemplate',
        className: 'chart-dialog',
        scope: $scope,
        preCloseCallback: function() { // When the popup is closed
            $scope.$apply(function(){
                // We remove the attr from the URL
                $location.search('attr', null);
            });
        }
    });
    // After showing the popup we set the attr on the URL
    $location.search('attr', a.id);
}

```

The last step is to check if the attr is set on the URL when the user first visits the page and if set show the chart directly. We will have to do this check after loading the data from the API. So we go back to the `getData` function of the `$scope.attrsTableParams` and we replace the comment:

```

// TODO: the code to show the plot based on the URL will be added here

```

With the following code:

```

// Read the attribute id from the URL
var paramsAttr = $location.search().attr;
// if found
if(paramsAttr !== undefined){
    // Search the corresponding attribute on the response
    var a = undefined;
    trends.forEach(function(t){
        if(t.id == paramsAttr) a = t;
    });
    // If the corresponding attribute found
    if(a !== undefined){ // Show the chart
        $scope.showChart(a);
    }
}

```

# Conclusion

My final internship at CERN is the best job experience I had till now. During this period, I discovered new cultures by meeting people from different countries, I worked within a group in a new flexible way and I learned new web development technologies.

This internship was part of my third year studies at ISIMA. I worked as a software engineer to develop and improve the LHCbPR (LCHb Performance and Regression) application. The applications used to analyse data collected by the detector on LHCb are tested using configurable test jobs. The main goal of LHCbPR is to provide physicists and developers with a framework in which they can easily do analysis of test jobs results. We started the development of the second version of this application from scratch and used flexible architecture and recent technologies. This version contains three layers: A database, a REST API and a frontend web application. My task was mainly to develop the frontend application and the analysis modules using AngularJS and new web development tools. But I did also some changes in the backend which was built using Django REST framework (written with Python programming language).

I started the application based on a template containing AngularJS and other libraries. After understanding the structure of the template I started customizing it to fit our needs. Modifications have been done on the files structure, the build system and the javascript code. The next step was to create some analysis modules and to add new helper classes which simplify this process. Finally I have written a development guide for users explaining how to add new analysis modules to the application.

The first version of the frontend application was done with three analysis modules. Other analysis modules should be added to have to same functionalities as the old version. One perspective was to simplify adding module for persons with no AngularJS knowledge by giving an easy to use group of functions hiding the complexity of this framework. This simplification layer could be made in a generic way giving the possibility to use it to build other modular web applications.