

**DAS** Departamento de Automação e Sistemas  
**CTC** Centro Tecnológico  
**UFSC** Universidade Federal de Santa Catarina

Platform and Methodology for  
Developing Modern Systems in  
Restricted Enterprise Environments,  
using Elixir/Erlang, Docker, CI/CD  
and Microservices

*Report submitted to the Universidade Federal de Santa Catarina  
as a requirement to approval on the subject:  
DAS 5511: Projeto de Fim de Curso*

*Rafael Jung*

*Berlin, June 2018*



Examiner Committee:

Ricardo Grützmacher  
Mentor at Rolls-Royce

Prof. Rômulo Silva de Oliveira  
Mentor at UFSC

Prof. Hector Bessa Silveira  
Responsible for the Course

Prof. Jomi Fred Hübner, Committee member

Luiz Alberto Serafim Guardini, Committee member

Roger Perin, Committee member



# Abstract

Due to the many threats of the digital age, the need for security is every day higher. The cost associated securing systems to protect strategic digital assets is very high. Then is in countless enterprises a very common pattern to have very strict IT and data security rules, thus causing loss of productivity especially in the engineering departments. The modern software development environment often requires fast changes, which incur in having to quickly change tools and deploy new versions.

To overcome this problem, a set of tools were used with the objective of creating a development environment completely outside of the enterprise systems. Within these tools are GitLab, with the integrated CI/CD pipelines; Docker, to "simulate" the destination server and compile the project dependency free and Elixir/Erlang to be the layer between data and views and serve as proxy and web server.

At the same time, a deployment process needed to be well defined and automated, in a way that would follow the best agile software deployment practices. This made sure all the out-of-network developed software would work just as specified in production, without the developers needing to worry with production-only issues.

At the end, this platform allowed to achieve very high development speeds, compatible with that of the best startup teams, by using latest technology. And at the same time, keeping compatibility with all the legacy tools in existence.

**Keywords:** Web-development, agile, git, docker, elixir, react, modern web, ci/cd, spa.



# List of Figures

Figure 1 – Rolls-Royce Dahlewitz - Main Entrance . . . . .	19
Figure 2 – Rolls-Royce Financial Highlights . . . . .	20
Figure 3 – Webpack . . . . .	30
Figure 4 – LAMP Architecture just as in this case . . . . .	34
Figure 5 – Engino Application . . . . .	35
Figure 6 – Scrum Methodology . . . . .	43
Figure 7 – GitLab Workflow . . . . .	44
Figure 8 – GitLab Issue Tracker . . . . .	45
Figure 9 – New Issue screen . . . . .	45
Figure 10 – GitLab Issue Board . . . . .	46
Figure 11 – Milestones . . . . .	46
Figure 12 – Sprint Planning . . . . .	47
Figure 13 – GitLab CI Pipeline . . . . .	48
Figure 14 – Pipelines with error are rejected . . . . .	48
Figure 15 – Complete CI/CD Pipeline . . . . .	49
Figure 16 – List of Milestones . . . . .	50
Figure 17 – Engino Admin . . . . .	60
Figure 18 – Engino Admin . . . . .	61
Figure 19 – GraphQL . . . . .	62
Figure 20 – Engino Deployer . . . . .	64
Figure 21 – Burndown Milestone 1 . . . . .	66
Figure 22 – Burndown Milestone 2 . . . . .	66
Figure 23 – Productivity Comparison . . . . .	68





# List of Tables



# List of abbreviations and acronyms

<i>API</i>	Application Programming Interface
<i>BEAM</i>	Bogdan/Björn's Erlang Abstract Machine
<i>CD</i>	Continuous Deployment
<i>CI</i>	Continuous Integration
<i>CRUD</i>	Create, Read, Update and Delete
<i>DAS</i>	Departamento de Automação e Sistemas
<i>DB</i>	Database
<i>DDD</i>	Domain-driven Design
<i>DOM</i>	Document Object Model
<i>ES</i>	ECMAScript
<i>IDE</i>	Integrated Development Environment
<i>IT</i>	Information Technology
<i>JSON</i>	JavaScript Object Notation
<i>LDAP</i>	Lightweight Directory Access Protocol
<i>MVVM</i>	Model-view-viewmodel
<i>NPM</i>	Node Package Manager
<i>NVM</i>	Node Version Manager
<i>OS</i>	Operating System
<i>R&amp;D</i>	Research and Development
<i>REST</i>	Representational State Transfer
<i>RHEL</i>	Red Hat Enterprise Linux
<i>SPA</i>	Single Page Application
<i>TDD</i>	Test Driven Development
<i>UFSC</i>	Universidade Federal de Santa Catarina

<i>UI</i>	User Interface
<i>UX</i>	User Experience
<i>VPS</i>	Virtual Private Server

# Contents

<b>1</b>	<b>INTRODUCTION</b>	<b>15</b>
<b>1.1</b>	<b>Reasoning</b>	<b>15</b>
<b>1.2</b>	<b>Objectives</b>	<b>16</b>
1.2.1	General objective	16
1.2.2	Specific objectives	16
<b>1.3</b>	<b>Document structure</b>	<b>17</b>
<b>2</b>	<b>ROLLS-ROYCE PLC</b>	<b>19</b>
<b>2.1</b>	<b>Rolls-Royce Deutschland</b>	<b>19</b>
<b>2.2</b>	<b>Engino</b>	<b>20</b>
<b>3</b>	<b>TECHNOLOGIES AND TECHNICAL BASIS</b>	<b>21</b>
<b>3.1</b>	<b>Technologies and methodologies</b>	<b>21</b>
3.1.1	Agile development	21
3.1.1.1	Continuous Integration	22
3.1.1.2	Continuous Delivery	22
3.1.1.3	Continuous Deployment	23
3.1.1.4	Test Driven Development - TDD	23
3.1.2	Domain-Driven Design - DDD	23
3.1.3	Microservices	24
3.1.4	Git	24
3.1.4.1	GitLab	24
<b>3.2</b>	<b>Application technologies</b>	<b>25</b>
3.2.1	JSON	25
3.2.2	SPA	25
3.2.2.1	Angular JS	26
3.2.2.2	React	26
3.2.3	Functional programming	27
3.2.4	Erlang	27
3.2.4.1	Elixir	28
3.2.4.2	Phoenix Framework	28
3.2.5	GraphQL	28
3.2.6	ECMAScript	29
3.2.6.1	Webpack	29
<b>3.3</b>	<b>Engino architecture</b>	<b>29</b>
3.3.1	Frontend	30

3.3.2	API . . . . .	30
3.3.3	Backend . . . . .	31
3.3.4	Server software . . . . .	31
<b>4</b>	<b>THE PROBLEM . . . . .</b>	<b>33</b>
<b>4.1</b>	<b>Access of information . . . . .</b>	<b>33</b>
<b>4.2</b>	<b>Current web development environment . . . . .</b>	<b>33</b>
<b>4.3</b>	<b>Engino . . . . .</b>	<b>34</b>
<b>5</b>	<b>PROJECT PLANNING . . . . .</b>	<b>37</b>
<b>5.1</b>	<b>Requirements and information compilation . . . . .</b>	<b>37</b>
5.1.1	Engino - soft requirements . . . . .	37
5.1.2	Server - hard requirements . . . . .	38
<b>5.2</b>	<b>Technology stack . . . . .</b>	<b>39</b>
5.2.1	Frontend . . . . .	39
5.2.1.1	AngularJS . . . . .	39
5.2.1.2	Vue.js . . . . .	40
5.2.1.3	Angular 4 . . . . .	40
5.2.1.4	React . . . . .	41
5.2.2	Backend . . . . .	41
<b>5.3</b>	<b>Methodology . . . . .</b>	<b>43</b>
5.3.1	GitLab development workflow . . . . .	43
5.3.1.1	GitLab Issue Tracker . . . . .	44
5.3.1.2	Milestones . . . . .	45
5.3.1.3	Sprint Planning . . . . .	46
5.3.1.4	CI/CD . . . . .	46
5.3.2	Development environment . . . . .	47
5.3.3	Issues and Milestones planning . . . . .	49
<b>6</b>	<b>IMPLEMENTATION . . . . .</b>	<b>51</b>
<b>6.1</b>	<b>Workstation . . . . .</b>	<b>51</b>
<b>6.2</b>	<b>Repository . . . . .</b>	<b>52</b>
6.2.1	Base project setup . . . . .	52
6.2.1.1	Phoenix application . . . . .	52
6.2.1.2	React Frontend (Webpack) . . . . .	54
6.2.2	CI/CD . . . . .	56
6.2.2.1	Testing pipeline . . . . .	56
6.2.2.2	Build pipeline . . . . .	57
<b>6.3</b>	<b>Application development . . . . .</b>	<b>58</b>
<b>6.4</b>	<b>Production server . . . . .</b>	<b>59</b>

6.4.1	Server setup . . . . .	60
6.4.1.1	PostgreSQL . . . . .	60
6.4.1.2	RSVG and ImageMagick . . . . .	61
6.4.1.3	Oracle Instant Client . . . . .	62
6.4.2	Deployment . . . . .	63
<b>7</b>	<b>RESULTS . . . . .</b>	<b>65</b>
<b>7.1</b>	<b>Development workflow . . . . .</b>	<b>65</b>
7.1.1	Milestones burndown charts . . . . .	65
7.1.2	Freedom for using tools . . . . .	65
7.1.3	Version Control . . . . .	67
<b>7.2</b>	<b>Productivity comparison . . . . .</b>	<b>68</b>
7.2.1	Case Study - new feature development speed . . . . .	68
7.2.2	Case Study - new developer in the team . . . . .	69
7.2.2.1	Old Engino . . . . .	69
<b>8</b>	<b>FUTURE DEVELOPMENTS . . . . .</b>	<b>71</b>
<b>8.1</b>	<b>Conclusion . . . . .</b>	<b>71</b>
<b>8.2</b>	<b>Further development . . . . .</b>	<b>71</b>
	<b>References . . . . .</b>	<b>73</b>
	<b>APPENDIX A – WEBPACK CONFIGURATION FILE . . . . .</b>	<b>75</b>
	<b>APPENDIX B – GITLAB CI CONFIGURATION . . . . .</b>	<b>79</b>
	<b>APPENDIX C – PRODUCTION DOCKERFILE . . . . .</b>	<b>81</b>





# 1 Introduction

Rapid advances in digital technology are redefining our world. The Digital transformation provides industry with unparalleled opportunities for value creation. The advantages of using automation in the industry has increased at a rate never to be imagined before. This includes being able to raise the worker's productivity, improve overall quality, reproducibility and much more.

Using software for automating workflows in companies is not something new. Companies need to be always innovating to withstand the competition. With the advent of the internet, personalized software of all kind has become much more accessible. For this reason the demand keeps growing every day.

In the last years, the web has been under very heavy development and changed a lot along the way. Until very recently, it only allowed for very simple interfaces with limited resources. But this changed completely with the advent of Single Page Applications (SPAs) with Javascript. As soon as browsers started getting more and more powerful, it allowed for applications to start being more and more complex on the client side. This created a booming increase in web application and related technologies, making them number one in open source development and in demand for jobs.

With facilities of web based application, many "traditional" desktop applications are migrating to the web or requiring web based features. Within the advantages are for example:

- No installation required;
- Device independent (client);
- Server based, meaning single storage base allowing for easy content sharing and data consistency;
- Similar look and feel as other web based applications, easing the learning curve;
- Fast applications, by combining well client side with server side.

## 1.1 Reasoning

All those advantages have been very well known by companies. Though, since web applications require a server, where critical data are stored, this can lead to serious security and data privacy issues. Therefore, it is much harder for big companies, such as Rolls-Royce, with more restrictive IT policies to tackle these problems quickly.

Another big issue is with the development of the softwares itself. The biggest reason why web development has been so upwards in the latest years is because of the open source movement. It has never been so easy for people to contribute to software. This lead to new technologies being constantly released, which allow the developers to focus on the features instead of language boilerplate. These normally allow for very productive software development and therefore lower costs in development, allowing for bigger projects. Although with the IT restrictions big companies impose, it is nearly impossible to be able to use these new technologies and libraries, thus lowering the productivity and effectiveness of the developers.

## 1.2 Objectives

### 1.2.1 General objective

The project described in this document aims solve the problems listed above:

- Difficulty to access to information;
- Slow and inefficient development environment;
- Current slow and unreliable softwares, with outdated interfaces;
- Expensive development and addition of features;
- Not being able to compete with the outside market in web technologies.

### 1.2.2 Specific objectives

These problems should be solved through integrating the best practices in agile web development, by:

- Using a Version Control System (VCS), allowing traceability of errors and easy rollback ability;
- Implementing an Agile Methodology, for faster development cycles without previously well defined specification and constraints;
- Implementing a Microservices based architecture, thus unleashing the best of each language/framework/libraries in a hybrid ecosystem;
- Using the latest web technologies, allowing better practices, better structured code and easier changes;

- Implementing Continuous Integration: where all pieces of change in the software are verified for backwards compatibility with automated testing;
- Implementing Continuous Deployment: every new feature generates a new version, corresponding to a package which can be easily deployed to the server;
- Implementing Deployment automation: the deployment to the server is automated via scripts for faster deploy times and to be less prone to human errors;
- And making sure the application runs in the internal production server.

## 1.3 Document structure

This document is divided in 8 chapters, following the rational order which was used for the development of the project. The chapters are:

- **Chapter 1:** Mentions briefly the problem to be solved with it's context;
- **Chapter 2:** Presents the company where the project was executed;
- **Chapter 3:** Contains all the theory behind this project with all relevant bibliographic references;
- **Chapter 4:** Expands on the problem this project aims to solve, showing all the difficulties and possible complications there might exist;
- **Chapter 5:** Planning which was made before the execution of the project;
- **Chapter 6:** Presents all the steps taken to implement the project as a whole. This chapter should also serve as documentation to what has been developed;
- **Chapter 7:** Presents the results obtained, measured by the productivity gains, the quality of the softwares generated and system metrics;
- **Chapter 8:** Discusses the final findings and future aspects, how the project could be further improved, which possibilities are now open, etc.



## 2 Rolls-Royce plc

Rolls-Royce [18] grew from the engineering business of Sir Frederick Henry Royce, first established in 1884. Charles Stewart Rolls established a separate business with Royce in 1904 because Royce had developed a range of cars which Rolls wanted to sell, which was incorporated in 1906 with the name Rolls-Royce Limited.

In 1971 the same company, Rolls-Royce Limited, entered voluntary liquidation because it was unable to meet its financial obligations though it remains in existence today. Its business and assets were bought by the government using a company created for the purpose named Rolls-Royce (1971) limited. This (1971) company remains in existence today and carries on Rolls-Royce business under the name Rolls-Royce plc, as a private company.

Rolls-Royce has established a leading position in the corporate and regional airline sector through the development of the Tay engine, the Allison acquisition and the consolidation of the BMW Rolls-Royce joint venture. In 1999, BMW Rolls-Royce was renamed Rolls-Royce Deutschland and became a 100% owned subsidiary of Rolls-Royce plc.

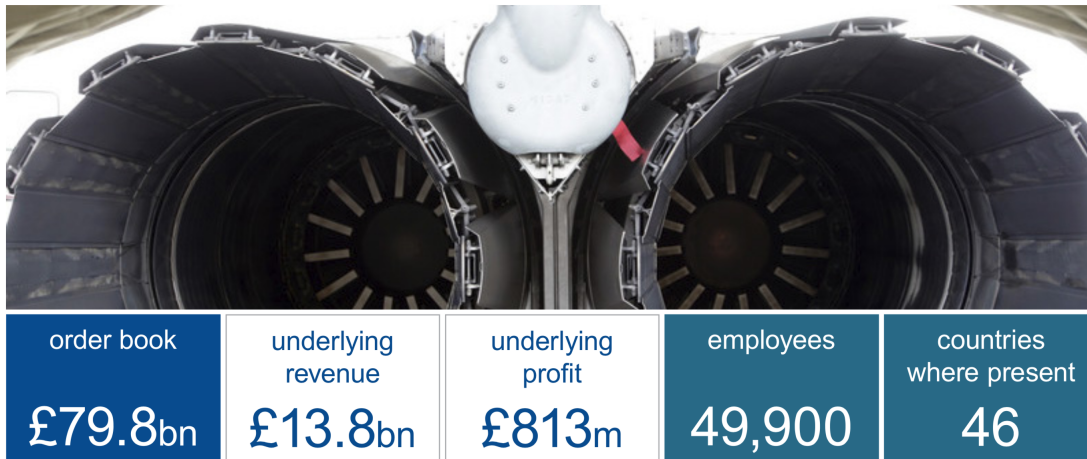
### 2.1 Rolls-Royce Deutschland



Figure 1 – Rolls-Royce Dahlewitz - Main Entrance

Rolls-Royce Deutschland employs around 3,600 people in Dahlewitz, near Berlin, and Oberursel, near Frankfurt am Main.

## Latest (2016) financial highlights



Rolls-Royce proprietary information



Figure 2 – Rolls-Royce Financial Highlights

Source: Rolls-Royce Corporate Presentation 2017

At the Dahlewitz location is the development and final assembly of all BR700 engines. The success story of the engines of this series is reflected in more than 22 million accumulated operating hours. As a competence center for twin-shaft engines, the Dahlewitz site is also responsible for the engine series Tay, Spey and Dart. Overall, Rolls-Royce Germany services around 9,000 engines in operation worldwide. [19]

Rolls-Royce took full control of the company in 2000, renaming it Rolls-Royce Deutschland.

## 2.2 Engine

Engino is an internal software used by all the Rolls-Royce Deutschland - Dahlewitz, which started to be developed in 2014. It aims to connect information scattered across many different sources in a visual and intuitive interface.

## 3 Technologies and technical basis

This chapter presents all technologies and paradigms mentioned throughout this document.

It is structured in 3 sections, where the first and the second list the technologies and the third explains how the engine application architecture was planned.

### 3.1 Technologies and methodologies

Here are introduced all the technologies and methodologies from the planning phase and/or that supported the development process as a whole.

#### 3.1.1 Agile development

The Agile Software Development Manifesto [1] was created to overcome the big overhead older methodologies caused on software development. The whole idea is to focus on the development of the software itself instead of too much in its methodologies and documentation.

The four most important principles are:

- **Individuals and Interactions over processes and tools:** Tools and processes are important, but it is more important to have competent people working together effectively;
- **Working Software over comprehensive documentation:** Good documentation is useful in helping people to understand how the software is built and how to use it, but the main point of development is to create software, not documentation;
- **Customer Collaboration over contract negotiation:** A contract is important but is no substitute for working closely with customers to discover what they need;
- **Responding to Change over following a plan:** A project plan is important, but it must not be too rigid to accommodate changes in technology or the environment, stakeholders' priorities, and people's understanding of the problem and its solution.

Since the Agile Manifesto, there has been multiple Agile Development methods, frameworks, etc. created and it has received the enormous support from the software development community.

Since it greatly reduces overhead and focuses on features, it works really well with web development, with fast paced development processes, where sometimes the requirements are constantly changing.

With the increase of usage in Agile methodologies and they becoming a standard in software teams, many tools and practices were created to help support the implementation and daily work.

Below are some examples of such tools and practices, as described from two main Git based software providers: Atlassian [3] and GitLab [12]

### 3.1.1.1 Continuous Integration

When working in teams, many times more than one person is editing the source code of the software at the same time. Continuous Integration is a software development practice in which you build and test software every time a developer commits and pushes code to the main repository. This normally happens several times a day, especially in bigger teams. This works really well with tools like Git and GitLab, which make the process of merging much simpler and intelligent.

Some of the best practices are listed below:

- Use a source control system: nowadays normally used together with Git;
- Make the build self-testing: the process that build the software should test before building if it works as expected;
- Keep the build fast: the build process should be completed in minutes, not hours;
- Every commit (to baseline) should be built: the test process should be executed for every commit to the source control system;
- Automate deployment: always deploy the merged software to a pre-production server, for final testing and eventually, bug-fixes (3.1.1.3).

Workflow Continuous Integration: TEST -> BUILD

### 3.1.1.2 Continuous Delivery

Similarly to Continuous Integration [3.1.1.1], Continuous Delivery is a software engineering approach in which continuous integration, automated testing, and automated deployment capabilities allow software to be developed and deployed rapidly, reliably and repeatedly with minimal human intervention. Still, the deployment to production is defined strategically and triggered manually.

Workflow Continuous Delivery: TEST -> BUILD -> DEPLOY  (manual)



### 3.1.1.3 Continuous Deployment

In Continuous Deployment, the code is put into production automatically, resulting in many production deployments every day. It does everything that Continuous Delivery [3.1.1.2] does, but the process is fully automated, there's no human intervention at all.

Workflow Continuous Deployment: TEST -> BUILD -> DEPLOY 🛠️(automatic)

### 3.1.1.4 Test Driven Development - TDD

Test Driven Development is a practice where the tests are created before the actual software. This practice works really well for very specific pieces of software, that must fulfil some very well defined requirements. For example import scripts.

The development workflow follows:

- Add a test and check if tests start to fail;
- Develop feature:
  - Write the new code;
  - Run all tests;
  - Repeat until tests pass.
- Repeat process until feature is completely implemented.

More information about Test Driven Development can be found in the url <http://agiledata.org/essays/tdd.html>.

## 3.1.2 Domain-Driven Design - DDD

Domain-Driven Design was initially introduced and made popular by Eric Evans in 2004 [5]. It is an approach for software development by decomposing complex software into understandable and manageable pieces. The modeling is done based on the reality of business as relevant to our use cases. DDD describes independent steps/areas of problems as bounded contexts, emphasizes a common language to talk about these problems.

The main concepts defined by DDD are:

- Context: The setting in which a word or statement appears that determines its meaning. Statements about a model can only be understood in a context;
- Model: A system of abstractions that describes selected aspects of a domain and can be used to solve problems related to that domain;

- Ubiquitous Language: A language structured around the domain model and used by all team members to connect all the activities of the team with the software;
- Bounded Context: A description of a boundary (typically a subsystem, or the work of a specific team) within which a particular model is defined and applicable.

Tough this pattern makes the software much less complex, by dividing monoliths <sup>1</sup> into many smaller pieces, it has the disadvantage of requiring a lot of time for the domains to be throughout understood and adding a lot of overhead because of its separation. It should only be used when the Domain is really well defined and very complex.

### 3.1.3 Microservices

Software projects can get very big with time. Thus, very difficult to maintain. The whole idea of Microservices comes all the way back from the unix architecture, where all softwares are small, have a well defined objective/context and can really easily exchange messages, through a well defined bridge.

Microservices are originally used in completely independent software, that connect to each other over the network. But its ideas of bounded context separation can be applied also inside one single software.

The ideas behind Microservices are many times shared with that of [Domain-Driven Design - DDD](#).

More information about Microservices can be found on the website <http://microservices.io/>.

### 3.1.4 Git

Git was created by Linus Torvalds in 2005 for development of the Linux kernel. It's popularized to a level, where it's considered the standard among software development.

As described in the Git website [10]:

“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.”

#### 3.1.4.1 GitLab

GitLab is a web-based Git-repository manager with wiki and issue-tracking features, using an open-source license, developed by Dmitriy Zaporozhets and Valery Sizov under GitLab Inc. As stated on the website [11]:

<sup>1</sup> A monolithic application describes a single-tiered software application in which the user interface and data access code are combined into a single program from a single platform

“GitLab is the first single application built from the ground up for all stages of the DevOps lifecycle for Product, Development, QA, Security, and Operations teams to work concurrently on the same project. GitLab enables teams to collaborate and work from a single conversation, instead of managing multiple threads across disparate tools. GitLab provides teams a single data store, one user interface, and one permission model across the DevOps lifecycle allowing teams to collaborate, significantly reducing cycle time and focus exclusively on building great software quickly.”

The code written in Ruby on Rails, with some parts later rewritten in Go. The core is completely open source, with the MIT License, enabling users to have an instance of the platform running on premises.

One of the biggest competitors from GitLab is GitHub <sup>2</sup>.

## 3.2 Application technologies

This subsection list all paradigms and technologies related strictly to the application development itself.

### 3.2.1 JSON

Javascript Object Notation (JSON) is a lightweight data-interchange format. The idea was to make it easy for humans to read and write and easy for machines to parse and generate at the same time as stated on the website [15]. JSON is a text format that is completely language independent, making it an ideal for data-interchange language.

Due to its simplicity and since it is so integrated with javascript, JSON has become the standard for web applications, specially when sending very small specific data.

JSON has less features than other formats like XML for example. But those feature are most of the time not needed and may only add overhead.

### 3.2.2 SPA

SPAs, short term for Single Page Applications are web applications or web sites that interacts with the user by dynamically changing the current page, via Javascript, rather than loading entire new pages from a server.

Basically, there is only one initial request to the webserver, which includes the whole layout and frontend features bundled. The subsequent requests are then only very

---

<sup>2</sup> GitHub is a similar platform, but is better known because it was the first of its kind in the market

fast and lightweight, with only exchange of specific information. These requests are then read by the Javascript and the necessary DOM<sup>3</sup> is altered. They are normally in the JSON format.

This approach makes web applications really fast and responsive, since the whole layout is managed only on the client side. As the requests are small, they are executed very fast.

These kind of applications started to get common with the release of Angular JS in 2010.

More information about Single Page Applications can be found on the website <http://singlepageappbook.com/goal.html>

### 3.2.2.1 Angular JS

AngularJS is an open-source front-end web application framework, under the MIT License, created by Google [2] to address many challenges developing SPAs [3.2.2].

The framework handles all of the DOM and AJAX glue code and puts it in a well-defined structure. This makes the framework categorized as opinionated.

Some main characteristics are worth listing:

- Opinionated: it comes with all libraries included, so less decisions needs to be done at the cost of flexibility;
- Two-way data binding: view updates automatically when data model changed. This can cause some undesirable side-effects;
- Performance: since all changes in the state are propagated directly to the DOM, AngularJS can be slow if too much data is updated.

### 3.2.2.2 React

React [17] is a view library (to build user interfaces) that facilitates the development of Single Page Applications [3.2.2]. It was created by Jordan Walke, a software engineer at Facebook and open sourced in 2013 and is today under the MIT License.

React became really popular because breaks through some of the features of Angular JS with much better approaches. For example using one-way data binding, instead of two-way<sup>4</sup>. There is also the virtual DOM, where React creates an in-memory data structure

---

<sup>3</sup> The Document Object Model is a cross-platform and language-independent application programming interface that treats an HTML, XHTML, or XML document as a tree structure wherein each node is an object representing a part of the document

<sup>4</sup> In Two-way data binding, view (UI part) updates automatically when data model changed. In one way data binding, this does not happen and we need to write custom code to make for updating the UI.

cache, computes the resulting differences, and then updates the browser's displayed DOM<sup>3</sup> efficiently.

### 3.2.3 Functional programming

Functional programming is a programming paradigm, that avoids state-changing and mutable data. These are also referred to as side effects. Eliminating side effects can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming.

Programming in a functional style can also be accomplished in languages that are not specifically designed for functional programming. In the snippet below is an example of Higher Order Functions written in Java 8 [9]. Since the language was not created with these concepts in mind, the syntax is really bad. But that's inherent from the Java programming language.

```
@FunctionalInterface
interface DogAge {
    Integer apply(Dog dog);
}

List<Integer> getAges(List<Dog> dogs, DogAge f) {
    List<Integer> ages = new ArrayList<>();

    for (Dog dog : dogs) {
        ages.add(f.apply(dog));
    }

    return ages;
}
```

### 3.2.4 Erlang

Erlang [7] is a general-purpose, concurrent, functional programming language, as with a garbage-collection runtime system. The Erlang Runtime System (OTP) consists of a number of ready-to-use components mainly written in Erlang, and a set of design principles for Erlang programs.

---

<sup>3</sup>Two-way might be simpler to use on the beginning, but much harder to debug applications and can account to many future problems as the applications grow

It was originally a proprietary language within Ericsson, developed by Joe Armstrong, Robert Virding and Mike Williams in 1986, but released as open source in 1998.

The main features of the languages are:

- Distributed;
- Fault-tolerant;
- Soft real-time;
- Highly available, non-stop applications;
- Hot swapping, where code can be changed without stopping a system.

The Erlang BEAM Virtual Machine executes bytecode which is converted to threaded code at load time. It also includes a native code compiler on most platforms, developed by the High Performance Erlang Project (HiPE) at Uppsala University. It is now fully integrated in Ericsson's Open Source Erlang/OTP system.

#### 3.2.4.1 Elixir

Elixir [6] is a functional, concurrent, general-purpose programming language that runs on the Erlang (BEAM) virtual machine. It can be used to create distributed, fault-tolerant, soft-real time, and permanently-running programs.

Elixir supports compile-time metaprogramming with macros and polymorphism via protocols, enabling the language's API to be easily extended.

#### 3.2.4.2 Phoenix Framework

Productive. Reliable. Fast. A productive web framework that does not compromise speed and maintainability. [16]

---

*Phoenix Framework*

Phoenix is a web development framework written in Elixir. It uses a server-side model-view-controller (MVC) pattern and is strongly based in the Ruby on Rails approach for building web application.

#### 3.2.5 GraphQL

GraphQL [13] is a data query language developed internally by Facebook in 2012 and open sourced in 2015, under the MIT License.

This query language was developed to solve some of the biggest problems from REST APIs<sup>5</sup> in bigger applications. The most common are:

- The need to do multiple round trips to fetch data required by a view;
- Clients dependency on servers;
- The bad front-end developer experience.

### 3.2.6 ECMAScript

ECMAScript is a trademarked scripting-language specification, created to standardize JavaScript, so as to foster multiple independent implementations. JavaScript has remained the best-known implementation of ECMAScript since the standard was first published in 1997 [14]. ECMAScript is commonly used for client-side scripting on the World Wide Web, and it is increasingly being used for writing server applications and services using Node.js<sup>6</sup>.

The sixth edition, known as ECMAScript 6 (ES6) or ECMAScript 2015 (ES2015) [8] adds significant new syntax for writing complex applications, including classes and modules. Other new features include iterators and for/of loops, Python-style generators and generator expressions, arrow functions, binary data, typed arrays, collections (maps, sets and weak maps), promises, number and math enhancements, reflection, and proxies (metaprogramming for virtual objects and wrappers). These new features revolutionized Javascript development, enabling the creation of much more complex applications.

#### 3.2.6.1 Webpack

Webpack [20] is an open-source JavaScript module bundler, under the MIT License. It takes modules with dependencies and generates static assets representing those modules. Since Webpack compiles all the code developed into a bundle, it is commonly used for converting new technologies to be supported in older browsers, as well as some many speed optimizations.

The Figure 3 gives an overview on the idea behind the creation of Webpack:

## 3.3 Engino architecture

Engino is the first application that will be developed using the platform described in this project. An overview about the project was mentioned in Chapter 2 [2.2].

<sup>5</sup> REST is an architectural style that defines a set of properties based on the HTTP protocol. It is used for data exchange between systems.

<sup>6</sup> Node.js is simply the Google V8 engine - a javascript interpreter used by Google Chrome® - bundled with some libraries to do I/O and networking

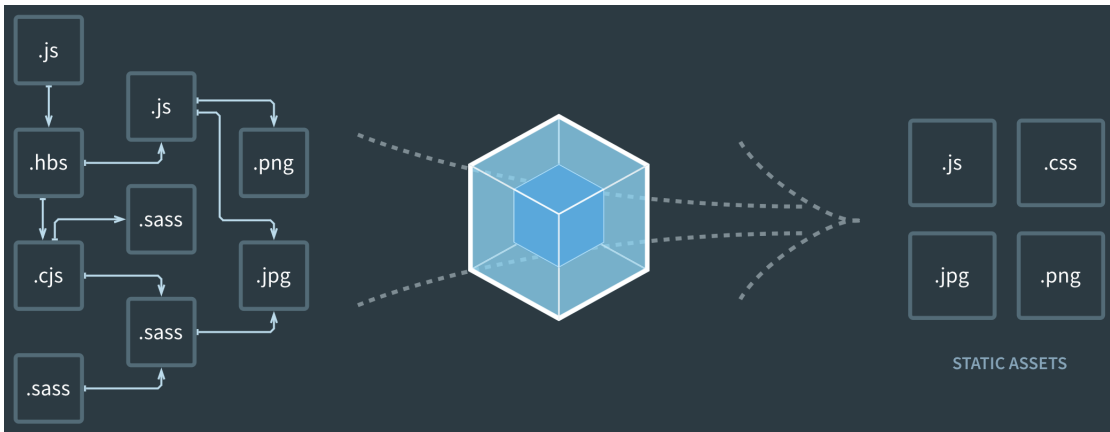


Figure 3 – Webpack

Source: Webpack Website [20]

Since the application was a rebuild from the old application and the requirements are constantly changing, the biggest requirement was really being as flexible and modular as possible.

An Agile approach was in this case fundamental. And by using the latest web technologies, it helped to have the application very modular and easy to be changed in the future.

### 3.3.1 Frontend

To make the software as flexible as possible, the interfaces should be built completely modular and as reusable as possible.

For this reason, the frontend stack was based around the React library (more information in 5.1.1), with the following libraries/technologies:

- **React**: Main view library;
- **Redux**: Handles state management;
- **React Router**: For generating and managing the routes;
- **Material UI**: React Components that Implement Google's Material Design.;
- **React Leaflet**: For implementing the Mapping functionality;
- **Superagent**: A REST Client for Javascript.

### 3.3.2 API

Since there should be multiple applications being glued together, the connection between backend and frontend should also be as flexible as possible. Knowing the problems



with REST APIs, the newer GraphQL query format interface was chosen to integrate the application with the server.

- The server uses the Elixir Library Absinthe, which makes a GraphQL API Endpoint;
- On the client side, the library used is Apollo, for connecting to the GraphQL API Endpoint;
- To connect the backend and frontend together, the communication will be done through websockets, for enabling real time updates on the applications.

### 3.3.3 Backend

The backend is composed basically by the platform Erlang/OTP, where code is written in Elixir and compiled into it.

Multiple libraries are used for the backend applications:

- **Elixir**: The language of the server applications;
- **Ecto**: The database wrapper library;
- **Phoenix**: The Elixir web application framework, to help developing the web facing applications;
- **Postgrex**: Elixir library to help connecting to the PostgreSQL database;
- **Cowboy**: The web server, that receives all the http requests;
- **Exldap**: Library to help making authentication through the internal LDAP servers (Microsoft Active Directory);
- **Tesla**: HTTP Client, to connect to external services;
- **Honeydew**: Library to manage worker pools and queues, for concurrent batch processing.

### 3.3.4 Server software

The server is defined by the internal IT from Rolls-Royce. The server available is a Red Hat Enterprise Linux, running in a virtual machine. Since there is no root access available, most software needed to be pre-compiled into binaries, that would run dependency-free on user space on the system. These software required by the Engino Backend application are:

- **PostgreSQL**: The internal database for the system;

- **ImageMagick**: Image processing library used;
- **Ghostscript**: Utility to read and process PDF files;
- **RSVG**: Utility to interpret SVG files;
- **Oracle Instant Client**: Proprietary tool to connect to the Oracle Databases.

## 4 The problem

This chapter explains the problematic of current web software development inside the company and the problems it aimed to solve.

### 4.1 Access of information

Rolls-Royce has a total of about 50000 employees worldwide, 3500 from them only in Dahlewitz. This impressive amount of people are all working on a very few number of projects, which sometimes take many years to complete. This leads to data about a project being scattered across many departments and even across different countries.

Another very common pattern seen in bigger companies is the widespread use of Excel. Many departments use it to store the masters of their data. The problem is with the nature of Excel itself, where data is not necessarily structured and it was not made to be distributed.

These conditions make it sometimes really hard to find consistent and dependable information. Therefore there are many different tools being developed by different teams with similar objectives: find and relate scattered information.

### 4.2 Current web development environment

The current development environment consists of a Windows Machine (required by Rolls-Royce IT) with a Notepad++ installed. The current production server is a shared RHEL<sup>1</sup> machine, with the common LAMP server<sup>2</sup>.

The production server is inside the Rolls-Royce network, allowing it to be accessible from any computer logged into the internal network. To access this server, for the purpose of changing the application, a Samba share<sup>3</sup> can be mounted and the code directly changed.

This method of development, although very fast to make changes in the server, does not have any of the advantages that using Git or any other VCS tool brings and presents multiple issues:

- There is no version control. New changes always overwrite the last version;

---

<sup>1</sup> RHEL: Red Hat Enterprise Linux

<sup>2</sup> LAMP = Linux + Apache + MySQL + PHP. This is a common web stack in older machines, since it simplifies server installation. It is mostly used for landing pages (websites)

<sup>3</sup> Samba is a free software that runs in Unix systems to provide the ability to mount Unix folders in Windows computers

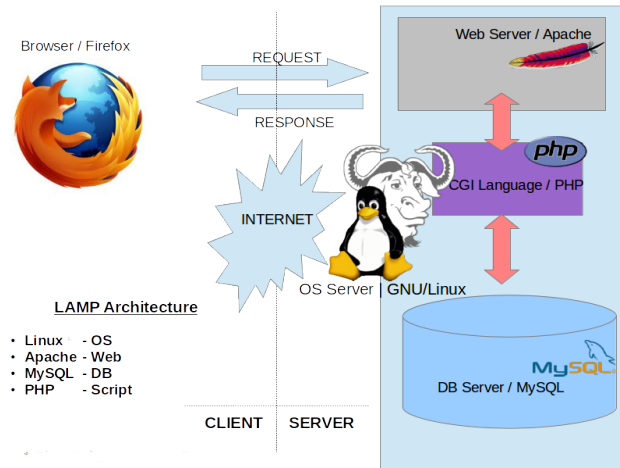


Figure 4 – LAMP Architecture just as in this case

Source: [https://en.wikipedia.org/wiki/LAMP\\_\(software\\_bundle\)](https://en.wikipedia.org/wiki/LAMP_(software_bundle))

- Secrets <sup>4</sup> are not secret. Anyone with access to the code is able to see the database connection strings for example;
- Testing is made in the same server and database as production is running. There is no automated testing to make sure the changes does not break other parts of the code;
- It is very difficult for 2 or more persons to develop at the same time;
- A programmer error can be directly propagated to the production system;
- It is very difficult to install/use new libraries or different settings;
- The development is only possible for someone with direct access to the shared folders and permissions are dependent on the server administrator;
- No authentication, meaning no way to recognize users and define different permissions, thus limiting the interaction possible and trustworthiness of data.

### 4.3 Engine

This tool, which was shortly mentioned in chapter 2.2, has been through constant changes, both in features and in conception. The development process was also very primitive, to stay within the constraints that Rolls-Royce's IT imposes. This led to a unstructured source code, which is very difficult to maintain and further develop.

<sup>4</sup> Secrets are credentials that are required for an application to run. Examples include database credentials, session keys, passwords, etc

The Engino project is the first software that will be developed under the platform and methodology described in this document. Since it ends up being more difficult to add features to the old version, it was intended that the Engino would be completely rewritten under the new environment. This will give us deep insight in how effective this platform will be and which changes are needed to enhance it.

Engino was developed using pure PHP and Javascript.

Below is a screenshot from the old engino application:

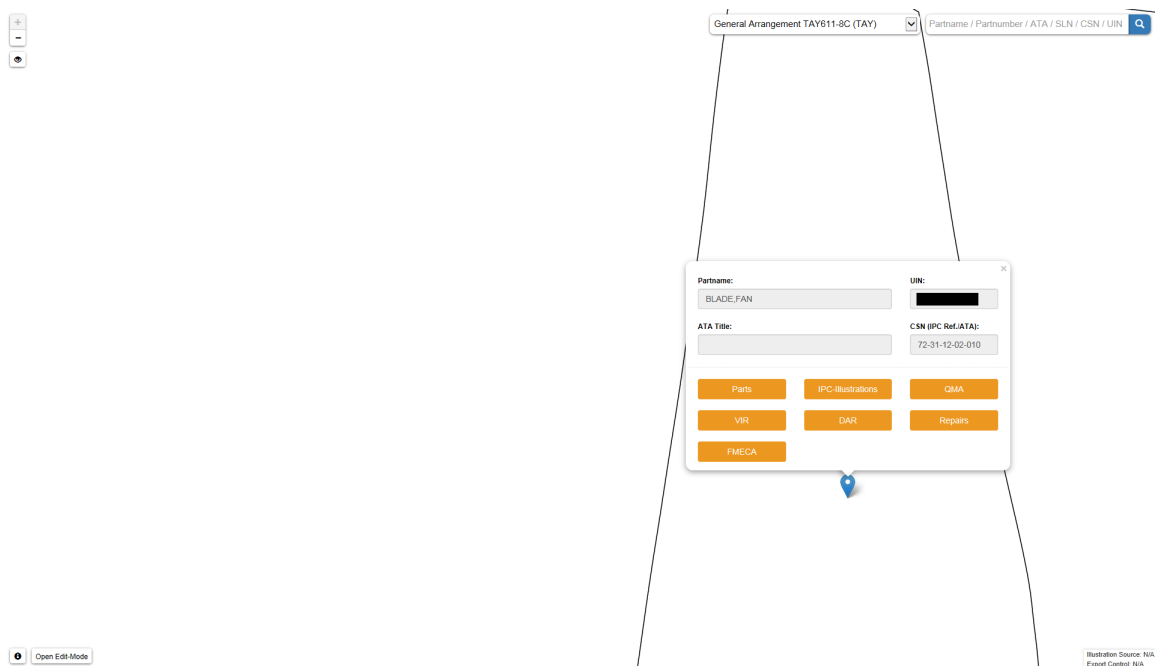


Figure 5 – Engino Application

The initial requirements that drove this project was to add new features to the existing system, such as new data sources, and to solve some of the problems with its interface. These problems included making it more visual to find data, slowness, missing information about the sources of the data and date of last update, have a better UI, a better UX and other similar features.

By the way the old application was developed, given it was only a proof-of-concept, a complete rewrite of the whole application into a new platform, with new languages and methodologies was seen as desirable. This was even more clear when the possibility was given to create new similar applications – within the same repository without needing to rewrite existing code.

The main problems that needed to be solved, from a non technical point-of-view, could be summarized with:

1. Fixing problems in the old application;

2. Adding new features;
3. Enabling code reuse with other projects;
4. Having a history of changes done to the project, with easy rollback;
5. Upgrading stability and security;
6. Enabling developers to have high productivity;
7. Having productivity working in teams;
8. Having a dependency-free server software for easy migration between servers;
9. Allowing project managers to have better insights on the status about the development of the software.

# 5 Project planning

Project planning defined here is related to how the “statement of work”<sup>1</sup> was created.

The planning was made mainly by the author, in conjunction with the project manager, to define the priorities and check the list of requirements.

This planning was done in three main steps:

1. First all information about the requirements and constraints were collected, as listed in section 4;
2. Then a deep throughout analysis was done, about the technology stack to be used and how the parts were to be connected to each other;
3. Finally, the implementation methodology was planned: Issues and Milestones were defined and the priorities and deadlines were set.

## 5.1 Requirements and information compilation

### 5.1.1 Engino - soft requirements

As seen in the last chapter 4.3, the current application is very difficult to maintain and was created with a very inflexible codebase. The main problems with the application are inherent to the current development environment and tools. Having a solid base to develop the new application is fundamental to overcome these problems. These problems 4.2 could solved by:

- Using a Version Control System;
- Have a well defined and automated deployment system (use CD - Continuous Deployment - principles), with a staging (testing) server;
- Store all secrets in environment variables, in a unique file in the server, with very restricted permissions;
- Develop an automated testing system (CI - Continuous Integration), for checking all the code for new errors, upon every new block of code developed (git commit);

---

<sup>1</sup> statement of work in this case refers basically to the list of tasks necessary to be executed, to have the problems described solved.

- Development inside a sandbox, with no access restrictions. This allows for much bigger productivity and code testing;
- Complete control on the permissions for allowing access to the code;
- Develop an authentication service;
- Modularize application, where everything should be a reusable component, both in Backend as in Frontend. This allows for much better code reuse.

Apart from these main problems in the application architecture and development environment, all current features and data source should be also available in the next version. This basically creates the requirements into which data sources should be imported and which/how much data should be displayed to the users.

While taking advantage of the flexibility on the new platform, some features are also wished to be available in the next version:

- Better layout and usability;
- A dedicated admin page, with full control to all data in the database;
- New data sources, or with different formats;
- Better and faster searching;
- Marker verification, allowing only predefined authenticated people to realize the verification;
- Direct connection to the main Oracle database.

### 5.1.2 Server - hard requirements

The server to which the whole application will be deployed is predefined from the IT department, with a bare metal Red Hat Enterprise Linux 7 (RHEL7) with only standard packages. Also, there will be no root access available and no internet access. All applications related to Engino should be in a single folder (sub-folders allowed) and be easily movable.

This restriction makes the deployment process much harder to control and automate. For this reason, some scripts are necessary to overcome this limitations:

- Pre-compilation of all libraries [3.3.4](#) in a container similar system (CentOS7);
- Package the compiled application into a single file;



- Deploy the single packaged file to an accessible location (not blocked by the Rolls-Royce firewall);
- The only machines in the network that have both access to the server and to the internet, are the common Windows® workstations. So a windows application is needed, to download the packaged application, copy to the server and run the necessary deployment scripts.

Details about the implementation of this deployment system will be given in the section [6.4](#)

## 5.2 Technology stack

This section explains which technologies are needed and on which criteria they were selected.

### 5.2.1 Frontend

As seen above [5.1.1](#), the platform should be set up with a flexible system, that allows the use of the newest technologies for Frontend development, but not require any specific. With this in mind, we are free to chose any library and/or framework for the Frontend (client) application, which could even be composed of multiple.

The analysis has taken into consideration the biggest four Frontend technologies to date:

- AngularJS;
- Angular 4;
- React;
- Vue.js.

Each of these four libraries are listed below, with the factors that were considered when connecting its features with the requirements from Rolls-Royce:

#### 5.2.1.1 AngularJS

AngularJS, or Angular 1 is the first version of the framework released. This framework has a very different approach than to the other technologies listed here, by supplying with much more features out of the box. Whereas this could be a good feature

in some cases, it really limits flexibility, by being opinionated. Also, this makes it much more difficult to learn and to start with.

Another undesirable features the framework are for example:

- Two-way data-binding: makes it really hard for debugging and managing state in bigger applications
- No virtual DOM: all changes are made directly into the HTML, making it slow when too many changes are happening

While this was the first option considered, since it is present in many other systems at Rolls-Royce, due to its lack of flexibility and problems above, it was also the first one to be discarded.

#### 5.2.1.2 Vue.js

Vue is also a newer view library, with principles very similar to React, but a little more opinionated, meaning it also packages some other libraries for state management, routing and others.

But since the library is not backed by any big company and has seen less activity as the other libraries, its use for such a project was considered too risky. For this reason, it was the second to have the use discarded.

#### 5.2.1.3 Angular 4

Angular 4 is actually the next version of Angular 2, which is a complete rewrite of AngularJS (Angular 1). For this reason, the analysis between Angular 1 and 4 is made here completely independent. Many of the problems existent in the first version of the library were solved, many others are still there:

- Compatibility with TypeScript 2.1 and 2.2, increased security in type casting, as well as increased speed of the ngc-Compiler. Strict typing, requires less attention on arguments and variables, leading to a decreased chance for errors caused by lack of attention;
- MVVM architecture. This template for building applications allows to associate elements of the View with the properties and events of the Model;
- Two-way data binding. This is rather controversial, since it speeds up development but at the same time, makes the debugging and production tests harder;
- Problematic to use with any languages other than TypeScript. Angular 4 coders can only use the tools inherent to the ecosystem of this framework;

- Difficult to master. As already noted above, regardless of the version, Angular is quite difficult. This is due to the need to use the relatively low-spread TypeScript language, as well as the purely theoretical knowledge of many beginning developers in the field of OOP practices. Because of this, mastering the advanced practices can be hard.

Even though this version has a much better approach than the Angular 1, it is a complete rewrite, so it is completely not backwards compatible. This means that its use will not have any advantage with the many projects written with Angular 1 already in development at Rolls-Royce.

#### 5.2.1.4 React

- Virtual DOM. The main advantage of React, in comparison with other similar tools, is virtual DOM - a lightweight copy of a complete DOM tree. This works really well when the application does not have too many states changing but can be slow in the contrary case<sup>2</sup>;
- Orientation to the creation of custom user interfaces. This library is claimed by its creators as being one of the most versatile from all. It also offers developers a lot of tools for UI element creation, such as Material-UI, React-Bootstrap, React Toolbox, React Native, etc;
- Lack of a unified approach. The big flexibility this library provides, comes at the cost of every project having its own stack of dependencies. This factor makes it harder to follow the best practices and may not play in favor of React developers in situations where the deadline is close, or where it is necessary to find the best ways to solve current tasks in the shortest possible time.

React was considered the most flexible and convenient from all of the libraries taken into consideration. Because of the need of customization required by the project and the flexibility provided by React, this was considered the best option for this project.

## 5.2.2 Backend

Since the client application (Frontend) was completely decoupled from the backend technologies (server), both sides of technologies were of independent choice. The backend technologies could even be heterogeneous, using a microservices approach, where each specific part of the application (context) would be developed using a specific language and framework/technologies.

---

<sup>2</sup> In AngularJS, every change is modified in the DOM, in React only the layout changes, not the whole state

Taking into consideration the nature of the application, with requirement of good modern web frameworks, for faster development speeds and greater capabilities, the following web languages and frameworks were considered:

- Python/Django;
- PHP with Laravel;
- Ruby on Rails;
- Elixir/Phoenix;
- Node.js with Express.

All of these frameworks are based on very similar paradigms and therefore present a similar programming interface. They are all based on MVC patterns, offer great community support (for libraries and problem solving) and would have all the necessary features.

The most crucial aspects were based on these key facts:

- Django is already in use in other projects inside Rolls-Royce. This could make setup easier and issues in production were already known (or should have been);
- PHP is already installed in many of the servers on the company and the language is being also used in many projects;
- Ruby on Rails was the most mature framework from all in the list, with best community support, best development patterns;
- Elixir projects can be compiled to a dependency-free in BEAM (Erlang Virtual Machine) executable. This leads to a very simple deployment procedure. Elixir, since it runs on the Erlang VM, also has its capabilities, like built-in support building distributed, fault-tolerant applications;
- Node.js projects work really well with Single Page Applications, like with React.

Elixir was considered the best choice for this project because of all the new concepts it has brought, which work really well with API only Backends and for flexible application development. The language's ecosystem allows for easy development of very flexible modules to be used in a micro-services based architecture, which was very desirable. Another great feature was the ability to easily compile the whole project into a single package that is able to run dependency-free in the server. This was a very desirable feature to have in the rather strict IT environment.

The other frameworks listed above were tough not discarded to be used alongside the main Elixir application for more specific parts of the project. The project was made

flexible to allow that. Along with Elixir, Ruby on Rails was planned to be used to make the connection to the Oracle database, since it had a very good library for it, which would make this much easier.

## 5.3 Methodology

Given that the project's scope was not well defined and was continuously changing, an Agile Methodology, such as Scrum, was very suitable.

In the Scrum Methodology, the project is cut in multiple pieces, which are developed together in a period of 2-4 weeks and is called a Sprint.

The image below illustrates this process.

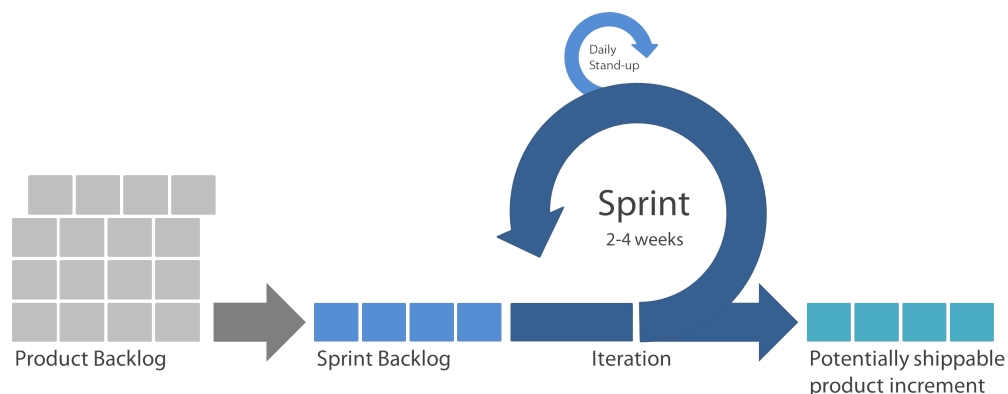


Figure 6 – Scrum Methodology

### 5.3.1 GitLab development workflow

GitLab is a Git-based repository manager and a powerful complete application for software development.

With an "user-and-newbie-friendly" interface, GitLab enables to bring all parts responsible for the project into the same platform. For example, project managers can monitor the development of the software through GitLab's interface in real time, testers can report bugs by creating an "Issue", maintainers can give code reviews and developers can easily see their tasks and submit the code.

The natural course of the software development process passes through 10 major steps:

1. **IDEA:** Every new proposal starts with an idea. This can be a completely new feature, a change in an existing feature or a bug fix;
2. **ISSUE:** The most effective way to discuss an idea is creating an issue for it. This issue is created inside the GitLab Issue Tracker;



Figure 7 – GitLab Workflow

3. **PLAN:** Once the discussion in the issue comes to an agreement, on how it should be done, the coding starts. For prioritizing issues, there is the Issue Board;
4. **CODE:** The actual issue is fixed, by writing code;
5. **COMMIT:** Once we're happy with our draft, we can commit our code to a feature-branch with version control;
6. **TEST:** With automated testing, the GitLab CI, run the testing scripts to check for errors introduced by the new code;
7. **REVIEW:** Once our script works and our tests and builds succeeds, we are ready to get our code reviewed and approved;
8. **STAGING:** The new feature is deployed to a testing server, with exactly the same features as the production server. There the software is checked if everything works as expected or if it still need adjustments. This is also a place to test for usability problems and issues that only happen in the production environment (because of type of real data, browser versions, etc);
9. **PRODUCTION:** When we have everything working as it should, the code is deployed to the live production environment;
10. **FEEDBACK:** Collect feedback about the deployed feature from real clients and see if it needs further improvement.

### 5.3.1.1 GitLab Issue Tracker

The Issue Tracker enables creation and management of all issues in the project. One issue can be a bug-fix, a new feature or a change in an existing feature. The issue tracker enables creation, management and reports about the project's development.

The Issue Board is a tool for planning and organizing the issues according to the project's workflow, setting priorities according to its context.

Issues will be used for planning each work package to be developed for the applications. For example, each importer of a new source of data is a different issue as well as each part of the frontend development.

The screenshot shows the GitLab Issue Tracker interface. At the top, there are tabs for 'Issues', 'Board', 'Labels', and 'Milestones'. Below the tabs, there are filters for 'Open' (216), 'Closed' (318), and 'All' (534). A search bar with 'Filter by name ...' and a 'New Issue' button are also present. The main content area displays a list of issues with columns for checkboxes, titles, descriptions, assignees, milestones, labels, weights, and update times. The issues listed are:

- Deploy our blog faster** (#438) - opened about a month ago by Marcia Ramos. Labels: Blog, Content Marketing, Design, Top Priority. Updated 27 days ago.
- EE Features - update docs** (#336) - opened 3 months ago by Marcia Ramos. Labels: GitLab EE, Tech Writing, Top Priority. 11 of 28 tasks completed. Updated about a month ago.
- Mexico Summit 2017 Shirt** (#535) - opened about 3 hours ago by Emily. Label: Design. 0 of 2 tasks completed. Updated about 3 hours ago.
- London Sticker** (#534) - opened about 3 hours ago by Emily. Labels: Design, Events. 0 of 1 task completed. Updated about 3 hours ago.
- Product Descriptions to Zuora and Salesforce** (#533) - opened about 3 hours ago by Francis Aquino. Updated 9 minutes ago.
- Photo Box Artwork** (#532) - opened about 19 hours ago by Emily. Labels: Design, Events. 0 of 1 task completed. Updated about 19 hours ago.
- Conversational Development Pillar** (#531) - opened about 20 hours ago by Erica. Labels: Content Marketing, Demand Generation. 0 of 3 tasks completed. Updated about 20 hours ago.

Figure 8 – GitLab Issue Tracker

The screenshot shows the 'New Issue' screen in GitLab. At the top, there is a checkbox for 'This issue is confidential and should only be visible to team members with at least Reporter access.' Below this, there are several form fields:

- Assignee:** Marcia Ramos (dropdown menu)
- Due date:** Select due date (calendar icon)
- Milestone:** 8.13 (dropdown menu)
- Labels:** feature proposal (input field with a close button)
- Weight:** 4 (dropdown menu)

At the bottom left, there is a 'Submit Issue' button and a note: 'Please review the [contribution guidelines](#) for this project.' At the bottom right, there is a 'Cancel' button.

Figure 9 – New Issue screen

To follow the best Agile [3.1.1] patterns, the issue creation process should also be dynamic. As soon as changes are seen to be necessary, an issue for this change should be created.

For the bigger scope, Milestones [5.3.1.2] should be used.

### 5.3.1.2 Milestones

Milestones are basically a tool to track the work based on a common target, in a specific date. An issue is normally part of a Milestone.

The goal can be different for each situation, but the panorama is the same: you have a collection of issues and merge requests being worked on to achieve that particular objective. This goal can be basically anything that groups the team work and effort to do something by a deadline.

Milestones should be used for each bigger feature to be produced, which is comprised

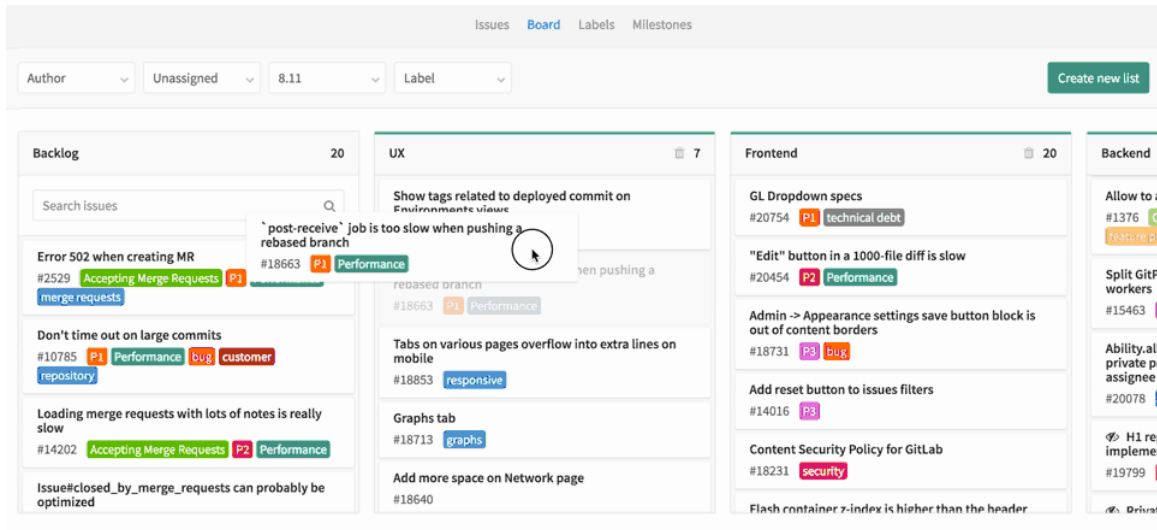


Figure 10 – GitLab Issue Board

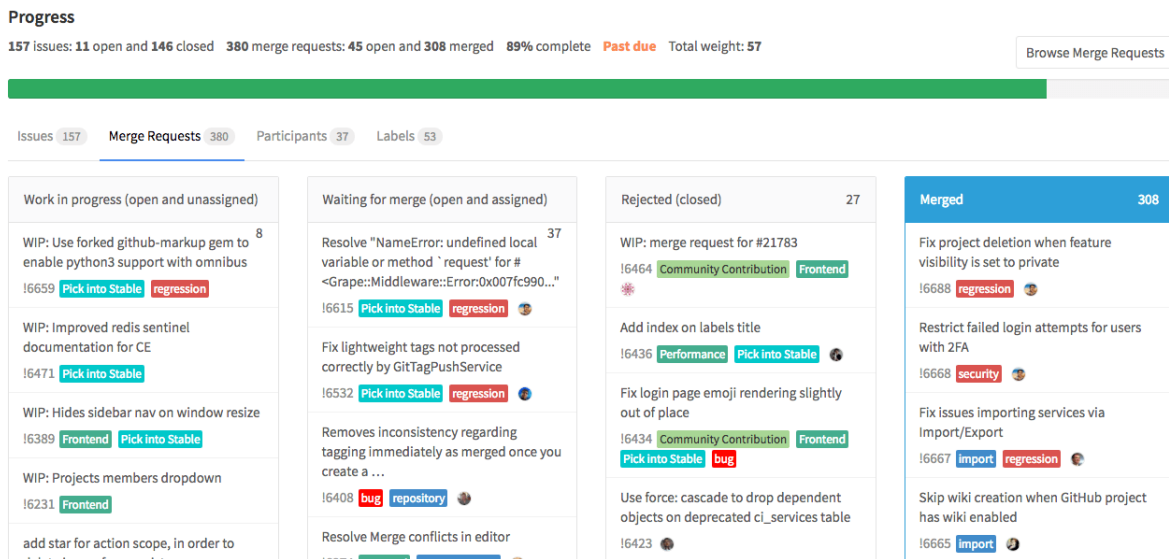


Figure 11 – Milestones

of multiple issues. They can be dependent of each other or not. An example of a milestone would be for example a “First version of the mapping frontend”

### 5.3.1.3 Sprint Planning

Using GitLab, each piece of software to be developed is called an Issue. To plan a Sprint, all related Issues were grouped together and developed in the same period. As illustrated in the image 12 GitLab provides a tool to make this process easier.

### 5.3.1.4 CI/CD

Throughout this project, when we talk about CD, we are actually talking about Continuous Delivery.



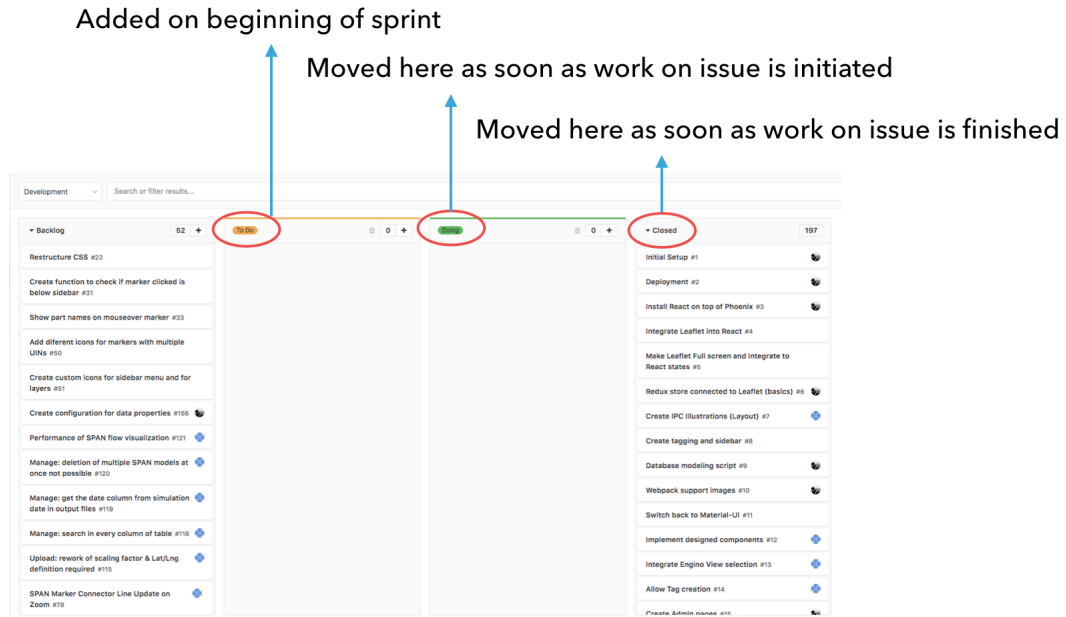



Figure 12 – Sprint Planning

Because of the the production servers do not have no access to the internet, the repository server (GitLab.com) does not have any interaction to the production server (internal at Rolls-Royce).

The approach used to deploy the application will be: TEST - BUILD - - DEPLOY, with the HAND icon symbolizing the manual deployment. This will be done through the help of a simple windows application, which has access to the compiled release packages and also access to the production server. Its job will be to basically copy this file to the server and run the necessary commands (move files, delete tmp directories, unpack the release, run migrations, etc), thus enabling a semi-automatic deployment.

The images 13 and 14 are examples of a CI/CD pipeline in GitLab, showing respectively a successful test and a commit with an error, detected by the automated testing. Branches with errors are rejected and are not able to be merged into the master branch, thus avoiding future problems in the code.

In image 15 the complete build process is shown, from testing to package building.

### 5.3.2 Development environment

To avoid all the restrictions imposed by the IT, because of security and data protection, the whole project structure is planned to run outside of the main network infrastructure.

The project repository will be hosted on the cloud, for ease of management and even allow remote teams and external companies to give consultancy for future further development. This gives a lot of flexibility, but also requires great care with the handling

### WIP: Importers bugfixes

Closes #228, #225

- Fix NPI uploader

Edited 5 days ago by Rafael Jung

Request to merge `importers-bugfixes` into `master`  
importers-bugfixes is 81 commits behind master

Open in Web IDE Check out branch

Pipeline #26414916 passed for 943b15a8 on importers-bugfixes  
Coverage 2.40%

No Approval required

Merge There are merge conflicts Resolve conflicts Merge locally

Closes #225 and #228  
Assign yourself to these issues

- backend\_elixir
- backend\_rails
- frontend

Figure 13 – GitLab CI Pipeline

Pipeline #26644211 failed for 25c5521a on fix-span-upload  
Coverage 2.30%

Merge The pipeline for this merge request failed. Please retry the job or push a new commit to fix the failure

Overall statistics

- Total: 1237 pipelines
- Successful: 1036 pipelines
- Failed: 169 pipelines
- Success ratio: 85%

Commit duration in minutes for last 30 commits

Pipelines charts

Pipelines for last week (23 Jul - 30 Jul)

Figure 14 – Pipelines with error are rejected

of data, meaning that no controlled data should ever be put into the repository (no real internal data, no server keys, etc). All the controlled files need to be avoided from git (by using the “.gitignore” file), should they be required in the development of the project.

For this to be possible, there will be extra workstations connected to an external WiFi network, that does not have access to the internal network and thus, no limitations. This allows for full productivity on the development of the project, without necessity to

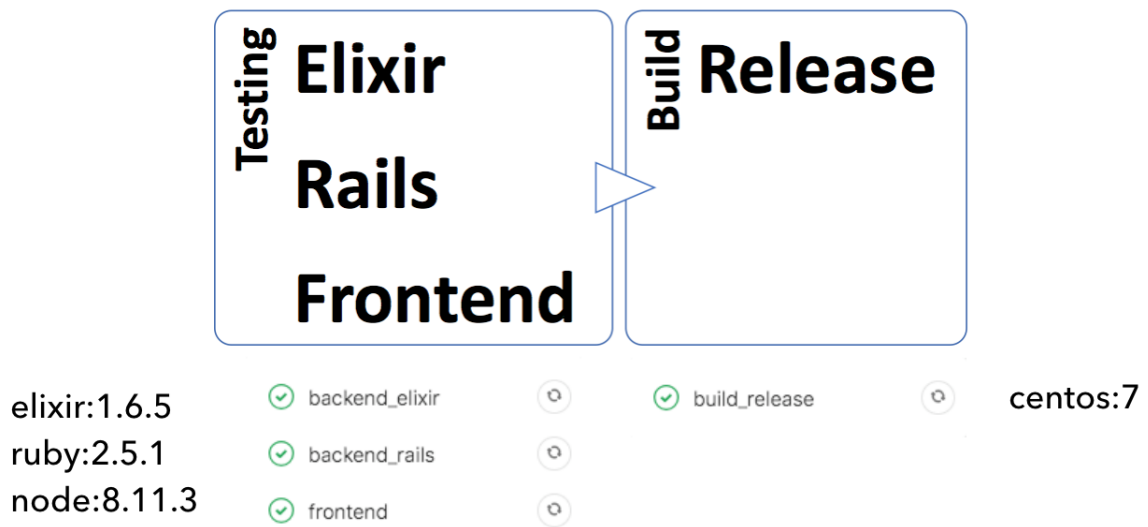


Figure 15 – Complete CI/CD Pipeline

worry with IT policies and being able to use the best practices and tools.

### 5.3.3 Issues and Milestones planning

The main project was basically divided into 2 Milestones: the setup and initial repository and the first release. There were created other milestones for other apps as well, but are not shown here.

The list of milestones can be seen in the image below:

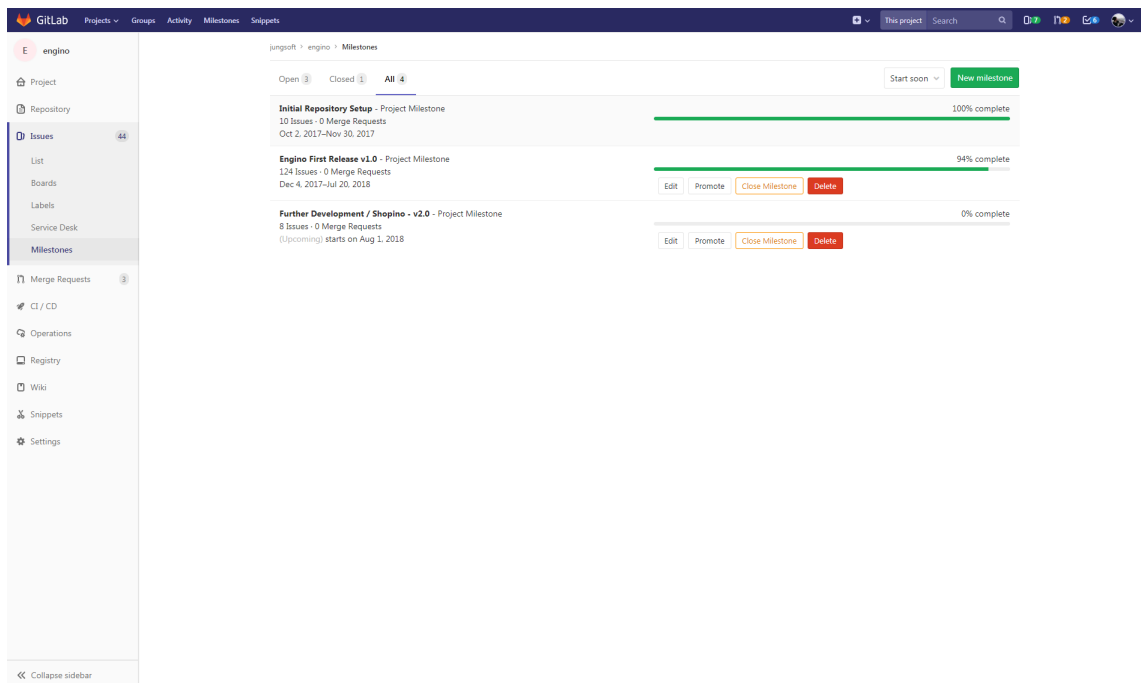


Figure 16 – List of Milestones

This image was taken on the second of July, therefore the project's status is from his date.

## 6 Implementation

This chapter discusses about how the planning made on [Project planning](#) was implemented and all the changes made during the process due to problems or simply changes in the requirements.

All the items commented in this chapter were completely done by the author, except when explicitly stated the opposite.

### 6.1 Workstation

To be able to start with the implementation of the project, the first necessity was obtaining and configuring the workstations.

These consisted of a clean computer, without any operating system installed. This allowed for a complete customization of the development environment tools, such as using an operational system without any restrictions and installing the best suited IDEs and other supporting software available.

There was one computer per developer available, which all the work was to be done with.

The workstations were configured with the following software:

- **Ubuntu Desktop** is the GNU/Linux distribution installed. The choice of distribution was made to be generic, with an OS that any new developer would either already know or be able to get used to very quickly;
- **Visual Studio Code** as the main editor, since it works very good with the languages that is was going to be used with, such as Javascript and Elixir. This IDE was chosen for its ability of customization and extensions. Any developer could be able to install a new IDE and use it, if better suited;
- **Node + NPM**: Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine, which is required for running Javascript developed code, such as the web-pack [3.2.6.1] dev server. NPM is used for easily adding packages on the Frontend application;
- **PostgreSQL** is the database used in production. For replicating the production environment, the same database was installed and used in the development machines;

- **Erlang + kiex**: The Erlang OTP and Elixir are required for compiling and running the server in development. Kiex is an Elixir version manager, for easily changing the Elixir version on the system;
- **Docker** is required for testing the CI pipeline and to simplify running an Oracle database server. It would ease the development of tools to integrate with Rolls-Royce's Oracle databases.

By using similar setups in all workstations, the risk of coming to problems in the development environment is lower, since the variability is reduced. Also, setting up a new software is easier, since the process would be the same for all workstations.

## 6.2 Repository

To start the well spoken application development, first the repository needs to be initialized. This consists of configuring all pieces of software to work together and setting up the automated applications deployment pipelines.

### 6.2.1 Base project setup

The base project is basically all boilerplate code necessary with setting up all the languages and build processes together. This defines the development workflow and the technology stack for the application.

With the complete technology stack already defined in [Technology stack](#), there were basically 3 repositories which needed to be setup and configured to work together:

- Phoenix Application: Backend API server and Main Application;
- Rails API: Connected to the main application using a GenServer;
- React Frontend: Connected using webpack dev server in development and independent from the main application in production, by producing the releases independently.

#### 6.2.1.1 Phoenix application

As described in [3.2.4.2](#), the Phoenix Framework is built using Elixir and its libraries. It is installed by running:

```
$ mix archive.install \
  https://github.com/phoenixframework/archives/raw/master/phx_new.ez
```

After having elixir and phoenix installed, the project can be created. Phoenix by default comes integrated with “brunch”, which is the frontend assets manager and bundler. Since the frontend application needs to be independent from Phoenix, there is no need to include brunch in the project. This backend project is then an API-only application. To create such a project, we can use the command line helpers from Phoenix and with the following command, create the project:

```
$ mix phx.new engino --no-brunch
```

This creates the following folder structure:

```
$ ls
README.md      config          lib             mix.lock       test
_build         deps           mix.exs        priv
```

After the creating the initial repository, the GraphQL server library needs to be installed. To install the Absinthe library (GraphQL server) from the elixir package manager “mix”, we need to add it to the package managements file, specifying the library and the version:

```
{:absinthe_plug, "~> 1.4.4"},
{:absinthe_ecto, "~> 0.1.3"},
```

Then with the following command, the packages are obtained:

```
$ mix deps.get
```

To finish installation, the Absinthe needs to be included in the Phoenix routing file, by adding this to the “router.exs”:

```
forward "/graphql", Absinthe.Plug, schema: Engino.Schema
forward "/graphiql", Absinthe.Plug.GraphiQL, schema: Engino.Schema
```

To finish the configuration, the database connections were set up. This is done by simply setting the variables in the config files in Phoenix. Due to its irrelevance, this configuration will not be shown here.

### 6.2.1.2 React Frontend (Webpack)

A big requirement is that the React applications would be completely independent on the Backend application. That being said, we should not rely on any backend tools to create the initial files. The handy *create-react-app* command line helper was used instead.

For frontend and backend separation, all frontend source files were put inside a folder called *frontend*. Other relevant files are packages list *package.json*, the webpack configuration file *webpack.config.js* and the folder node modules, where the installed frontend libraries reside.

To simplify the configuration, these are going to be moved to the root of the application.

With *react-create-app*, the main frontend application was generated, with some files moved around to integrate better with Phoenix and make development easier:

```
$ npx create-react-app engino
$ mv engino/ frontend/
$ mv frontend/src frontend/app
$ mv frontend/node_modules .
$ mv frontend/package.json .
$ mv frontend/yarn.lock .
$ cat frontend/.gitignore >> .gitignore
$ rm frontend/.gitignore
$ rm frontend/README.md
```

After configuring the initial application, *webpack* needs to be configured to compile the application both in production a single time and in development constantly after every change.

Webpack can be started as a development server, which supplies all the files to the frontend on-the-fly as they are available. It was also configured to enable React's Hot Module Reloading, which enables changes to be applied in the frontend without needing to reload the application, thus making development more productive.

The final *webpack* configuration file is available in the attachments [Webpack Configuration File](#).

For the frontend to be included in the web server, a simple javascript file is generated and the output is written to the phoenix's assets folder *priv/static*. Phoenix then serves this files normally as a web request.



To simplify the development, the phoenix application can be configured to automatically start the webpack development server as soon as the phoenix server is started, by the command *mix phx.server*. For this, the phoenix endpoint needs to be configured as follows:

```
config :engineo, EngineWeb.Endpoint,
  url: [host: System.get_env("HOST") || "localhost"],
  http: [port: 4000],
  debug_errors: true,
  code_reloader: true,
  check_origin: false,
  watchers: [
    node: [
      Path.expand("node_modules/webpack-dev-server/bin/
        webpack-dev-server.js"), "--no-inline", "--stdin", "--config",
      Path.expand("webpack.config.js"), "--host", "0.0.0.0"]
  ]
```

In the phoenix web server, the javascript is then read directly from the production generated file, if in production or from the webpack development server, if in development. This configuration is available in the frontend application's main HTML file (non relevant parts of the HTML were removed and others simplified):

```
<!DOCTYPE html>
<html lang="en">
  <body id="body">
    <div id="app-container">
      </div> <!-- /container -->

      <%= if Application.get_env(:engineo, :environment) == :dev do %>
        <script src="http://localhost:4001/js/app.js"></script>
      <% else %>
        <script src="<%= static_path(@conn, "/js/app.js") %>"></script>
      <% end %>
    </body>
  </html>
```

In the code, we can see the differentiation between the production and development environments. In development, the file is served through another server, which is constantly

being recompiled and updated. In production, the file is completely static.

## 6.2.2 CI/CD

Having the complete application setup, the next step is to create the Continuous Integration and Continuous Delivery pipelines.

This was developed using GitLab's integrated CI tool. It works by creating a file called *.gitlab-ci.yml* in the root of the application, declaring all the steps of the CI script.

The complete file is available in [GitLab CI Configuration](#).

Basically, the CI process is divided into multiple jobs and multiple pipelines. Pipelines run sequentially and can pass files between each other and each pipeline can run multiple jobs simultaneously. For simplicity, there were created two pipelines: one for testing purposes and the other for building the release, called respectively *test* and *build*.

### 6.2.2.1 Testing pipeline

The purpose for the Testing pipeline, as the name says, is to really test the application, by using automated testing. Each repository developed has its own testing framework. For this reason, it makes sense to divide the testing pipeline into three different jobs:

1. Elixir backend testing: Runs all the tests created in the Elixir backend application. For this testing, Elixir's own testing framework was used (ExUnit);
2. Rails API application testing: Runs tests on the rails application. For this, the library *RSpec* was installed, to be used to facilitate developing the automated tests;
3. Frontend applications testing: Run all the tests in all frontend applications. The library *jest* was installed and configured to be used on the frontend testing. This library was chosen because of its very good integration with React and its ecosystem.

By checking the configuration in [GitLab CI Configuration](#), we can see every job defined in the root of the file, where the commands each job runs inside a docker container based on the language of the test itself. For example, in Rails testing uses the Docker image *ruby:2.4*<sup>1</sup>, which already has everything installed and ready for testing the application. This simplifies maintenance and updating the base images.

As the desired for integration testing it to have it testes after each code commit to the repository, this pipeline is set to run every time there is a new commit. This makes sure that problems are discovered during the development of features instead of only in

---

<sup>1</sup> This Docker image is available publicly on Docker Hub - [https://hub.docker.com/r/\\_/ruby/](https://hub.docker.com/r/_/ruby/)

the end. It also allows for knowing in which commit a specific test stopped to work, thus helping with finding the problem.

### 6.2.2.2 Build pipeline

In the opposite of the test pipeline, this should be run only when the code developed for the issue is done. That means, when it is merged into the master branch. Also, it should only be triggered if the tests pass. This makes sure there won't ever be any release available for deployment to the server which breaks some feature, detected by the automated testing system.

The packing of the release is done by the help of the Elixir library Distillery. It basically compiles the whole Elixir code and packs it into a single file *engino.tar.gz*. To be able to have the React application and the Rails application bundled together into this file, we need to make sure they are in the correct folders and ready for packaging. For the rails application, this means installing all libraries (*gems*) into a specific folder and for the frontend, installing all packages and generating the output javascripts into the phoenix application.

Therefore, the build process follows the following order:

1. Install Elixir libraries, by using *mix deps.get*;
2. Install Frontend packages, by using *yarn install --prod*;
3. Install Ruby gems (libraries), by using *bundle install --gemfile=engino-rails-api/Gemfile --path=gems*;
4. Build Frontend applications into *priv/static* and process them on phoenix with *mix phx.digest*;
5. Create release file, with *mix release*;
6. Copy generated release to a path accessible by the deployment script. In this case, Amazon S3.

Since the Erlang release binaries needs to run under the same environment and it was generated, the environment where the package is generated though the scripts above needs to be the same as where the application will later run. This means, the release needs to be generated under the same environment as the production server, in this case a Red Hat Enterprise Linux 7 (RHEL7).

Since both RHEL7 and CentOS 7 share the same architecture, and since Cent OS 7 is more publicly available, the application release can be generated under this system.

This is very simple to achieve by using a public Docker image, already with CentOS 7 installed. Though for the generation of the release, there are many requirements, such as having Elixir, Node, Python, Ruby, Oracle Client, etc installed. This makes finding a base Docker image very hard and thus the best option is to create our own image.

For the generation of this image, basically all the required applications needed to be installed, on top of a Cent OS 7 machine. The requirements list follows:

- Oracle Client;
- Python + PIP;
- Node.js;
- Yarn;
- Erlang/OTP;
- Elixir;
- Ruby on Rails;
- Extra dependencies of the above applications, such as curl, gcc, openssl, zip, git, postgresql-devel, wget, etc.

Taking all these requirements into consideration, the Docker image was create on top of the official CentOS 7 image, the *centos:7*. The steps to create the image are inside the Dockerfile, which in this case basically consisted of installing all these requirements above. The complete file is in the appendix [Production Dockerfile](#)

After building this Dockerfile, the image can be pushed to GitLab's Registry <sup>2</sup> and used in the CI, by declaring the job's image with

*image: registry.gitlab.com/jungsoft/engino/build/production:centos7*

With this, all the commands declared in the job will be run inside the environment prepared in the Docker container.

## 6.3 Application development

Only at this point, the application could in fact start to be developed. Changes in the initial repository setup were foreseen, but should only be incremental in making the interfaces better, instead of completely changes how everything was set up.

---

<sup>2</sup> A Registry is where Docker images can be stored

Having the initial repository setup, the development of the application could be split into many different work fronts, which could be done simultaneously and interchangeably. The two main work fronts were the Backend and the App Frontend, which could be done completely simultaneously, due to the complete separation between them both.

The work fronts were defined as in the Milestones planning in [Issues and Milestones planning](#), which are:

- Frontend App;
- Frontend Admin;
- Backend Elixir API;
- Rails Oracle connector.

The work fronts were started as soon as the repository was initialized and done simultaneously, by different team members. Since they were all very independent, there were no issues synchronizing the work done.

All the work fronts cited above were developed by the author, except the Frontend App.

The workflow done was basically to synchronize the frontend development with its backend. So as soon as a new feature in the frontend was created, a query/mutation in the GraphQL server (*Absinthe*) was also created and added to the frontend.

The Admin area was created as an additional app afterwards. Other applications were also added as additional frontend apps afterwards with very little effort, proving the flexibility of the platform to add new projects. There were no changes needed in the server or in the deployment process. An example of app was a standalone gallery, built reutilizing the same components from the Frontend App, developed by the same team member.

An image of the with React developed admin area can be seen in the figure 17 or in the figure 18, in respectively white and dark themes.

On the Backend Elixir a tool called GraphQL was installed, together with the Absinthe library. This tool allows for executing queries in the backend, for testing purposes and is a great tool for debugging. The Figure 19 is a screenshot from its interface.

## 6.4 Production server

As already mentioned before, the production server is a Red Hat Enterprise Linux 7 (RHEL7). Also, there is no root access to the server available and no internet. This makes it much harder to install the application's requirements.

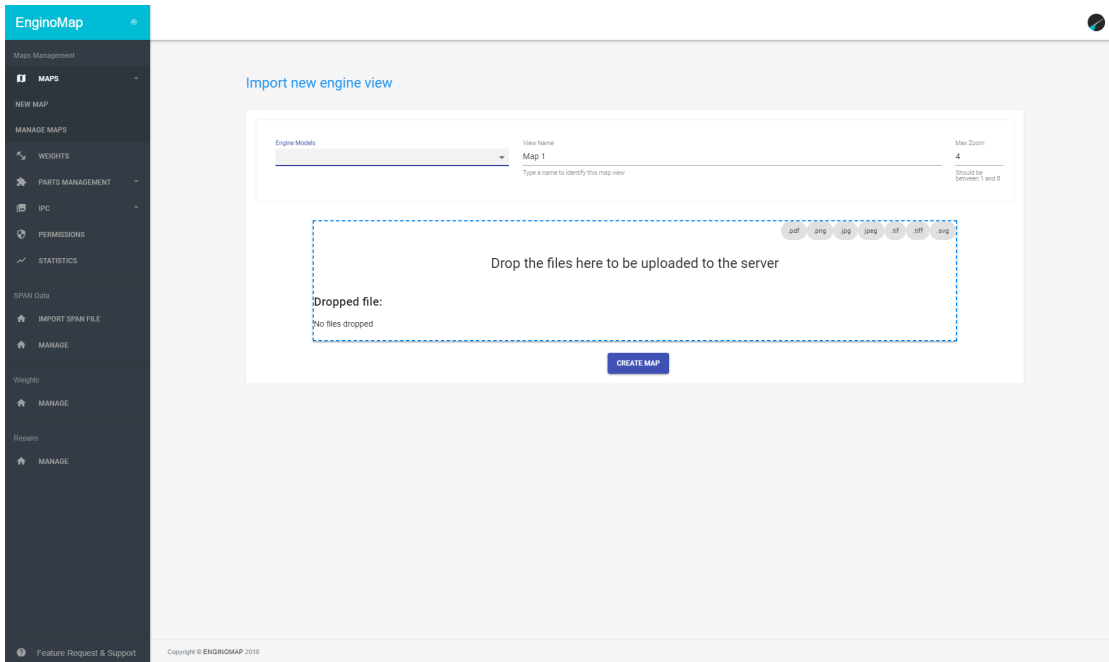


Figure 17 – Engino Admin

Admin area showing the engine view upload drag-and-drop. The new views in the software are added via this uploader.

### 6.4.1 Server setup

Installing the application’s dependencies into the server should follow the same principles as installing the application itself: we install it inside a similar environment, in an unrestricted Docker container and then copy the generated binary files to the server. This is simplified by the fact that these requirements will change very little with time, meaning a manual copy once should suffice.

Also, the list of requirements to run the application is different as the requirements to generate it. For example, *ImageMagick* is only required in runtime, whereas *Node* only during the build.

As listed in [Server software](#), these requirements need to be available in the server. The only software now already installed in the server is GhostScript, all the others need to be built inside a similar environment and copied to the server. Luckily, the exact same Docker image used in the build pipeline [6.2.2.2] can be used here as well.

The steps to generate the application binaries were taken from their respective websites and then simply run inside the Docker container.

#### 6.4.1.1 PostgreSQL

For building the PostgreSQL binaries, all the requirements were already met. Then simply by running the following script the binaries were generated:

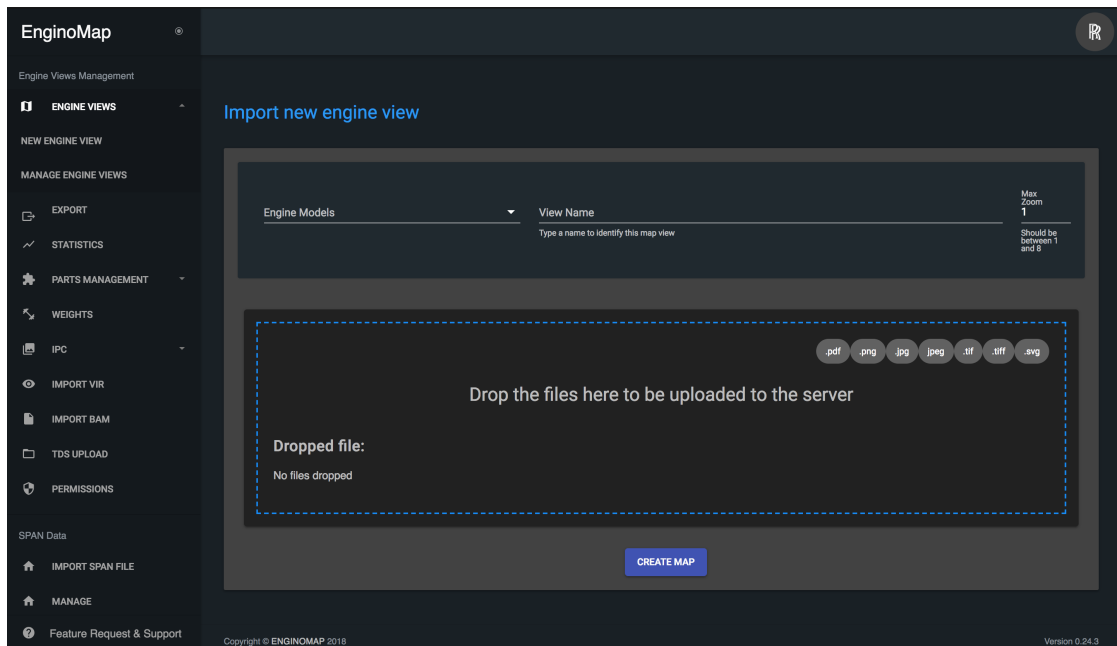


Figure 18 – Engino Admin

Admin Area in dark colors.

```
# Generate PostgreSQL binaries -> Only for Binaries Generation
RUN curl \
  https://ftp.postgresql.org/pub/source/v10.1/postgresql-10.1.tar.gz \
  | tar xz && cd postgresql-10.1 \
  && ./configure --with-openssl --prefix=/opt/postgresql \
  && make -j $NUM_CPU && make install
```

Then, the only needed step is to package these files into a tar, send it somehow inside the Rolls-Royce network (for example using Amazon S3) and unpack in the server.

#### 6.4.1.2 RSVG and ImageMagick

Very similarly to PostgreSQL, the RSVG and ImageMagick also need to be compiled and built. But in this case, some dependencies were missing, but could easily be installed in this unrestricted Docker container. The following script was able to generate the binaries:

```
# Generate binaries for librsvg2: dependency of ImageMagick
RUN yum-builddep -y librsvg2 \
  && wget https://download.gnome.org/sources/
      librsvg/2.40/librsvg-2.40.16.tar.xz \
  && tar xvf librsvg-2.40.16.tar.xz && cd librsvg-* \
```

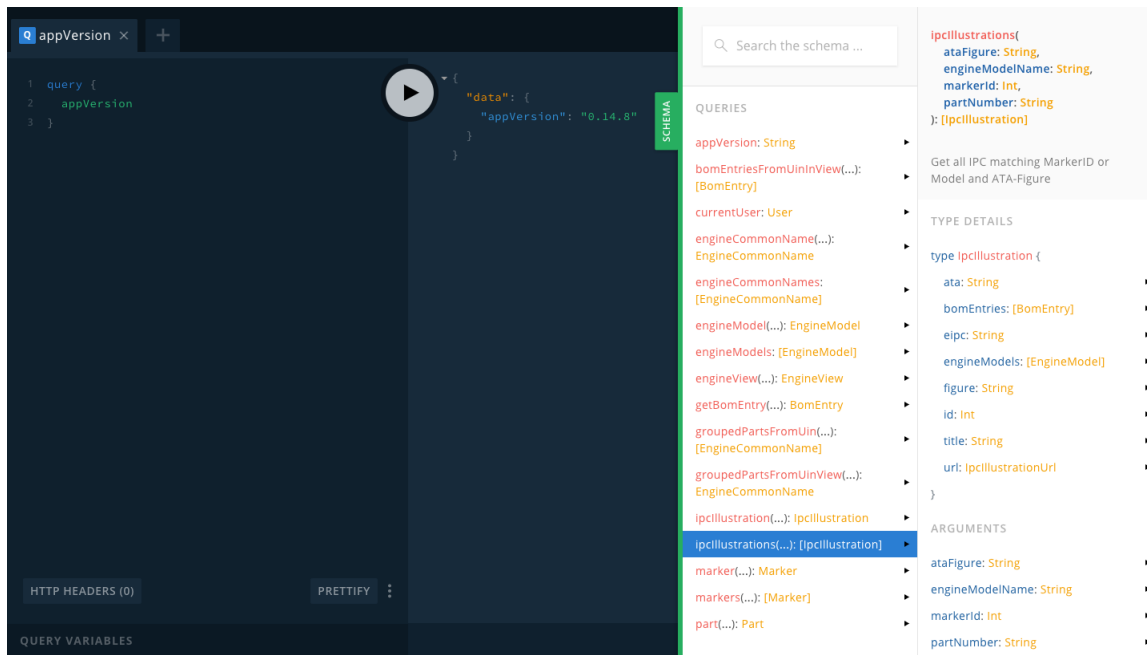


Figure 19 – GraphQL

GraphQL Tool. On the left side, is where the GraphQL query is written. On the middle, is the results and floating on the right side is the schema definition, where the API can quickly be checked by developers.

```

&& ./configure --prefix=/opt/librsvg \
&& make -j $NUM_CPU && make install

# Generate ImageMagick binaries
RUN yum install -y ImageMagick-devel libpng-devel librsvg2-devel \
ghostscript \
&& wget https://www.imagemagick.org/download/ImageMagick.tar.gz \
&& tar xvzf ImageMagick.tar.gz && cd ImageMagick-7* \
&& ./configure --prefix=/opt/imagemagick --with-png=yes \
--with-rsvg=yes --with-gslib=yes --disable-installed \
--disable-shared --enable-delegate-build \
&& make -j $NUM_CPU && make install

```

Exactly the same steps were necessary to copy the built binaries into the Rolls-Royce server.

### 6.4.1.3 Oracle Instant Client

As seen in the [Production Dockerfile](#), the Oracle Client binaries are already included. Then, we only need to package them and copy to the server, just as already done before.



## 6.4.2 Deployment

With the server completely setup, the deployment process can just assume that all requirements are already there.

To simplify it even further, a file called *.env* was created on the root of the application, with all the path to the software installed on [Server setup](#). This file is read during application boot and all its environment variables, such as server keys and paths, are automatically set. This way, the deployment can be done without the need to worry about the server configuration.

At this point, the deployment consists of performing the following steps:

1. Download release;
2. Copy release to server;
3. Stop running application;
4. Unpack release files into the corresponding folder;
5. Run database migrations<sup>3</sup>;
6. Start new application.

Since this application needs to run inside the internal Windows Active Directory network, the easiest way is to create a simple Console Application developed in *C#* that will perform exactly the steps above.

Since *Amazon S3*<sup>®</sup> is available inside the network, this seems like a very good location to store the releases. Using *C#* libraries for the connection to Amazon S3, the app is able to authenticate (via a *AWS S3 C#* library), find the latest release automatically and download it.

The connection to the server can be done via *SSH*. There is also another *C#* library called *Tamir.SharpSSH* that facilitates connecting and running commands via *SSH* on the server. The path inside the server, where the application needs to be deployed to is also predefined inside the application.

With a simple `$ tar -xvzf file.tar.gz -C production`, where *file.tar.gz* is the release package and *production* is where the application will be stored, the application is updated.

The BEAM binary release has the capability of running elixir commands via the command line and executing specific functions. We can then create a module<sup>4</sup> responsible

<sup>3</sup> A change in the database structure (change tables, columns, etc) is called a migration

<sup>4</sup> A module in elixir is similar to a class in object oriented languages

for Deployment tasks. In this example, this module was called ReleaseTasks. This module should include all release related tasks, for example, running migrations, or any other scripts that needs to run on every deployment. To use this, one simple command is needed:

```
$ ./bin/engino command Elixir.Engino.ReleaseTasks migrate
```

By running this command, we ensure the application has its database tables up-to-date with the code.

The BEAM binary also has a simple command to start the application as a daemon<sup>5</sup>, restart and stop. Then, simply by wrapping both commands above with a stop and then a start, we ensure the application is deployed without problems. We could also use the restart command, to have a 100% uptime deployment. But considering the 4 commands together take only about a second to be run, this one second downtime is not a problem. There should also be noted, that when starting the application, the PATH should be set

```
$ ./bin/engino stop  
$ PATH=$ENV_PATH ./bin/engino start
```

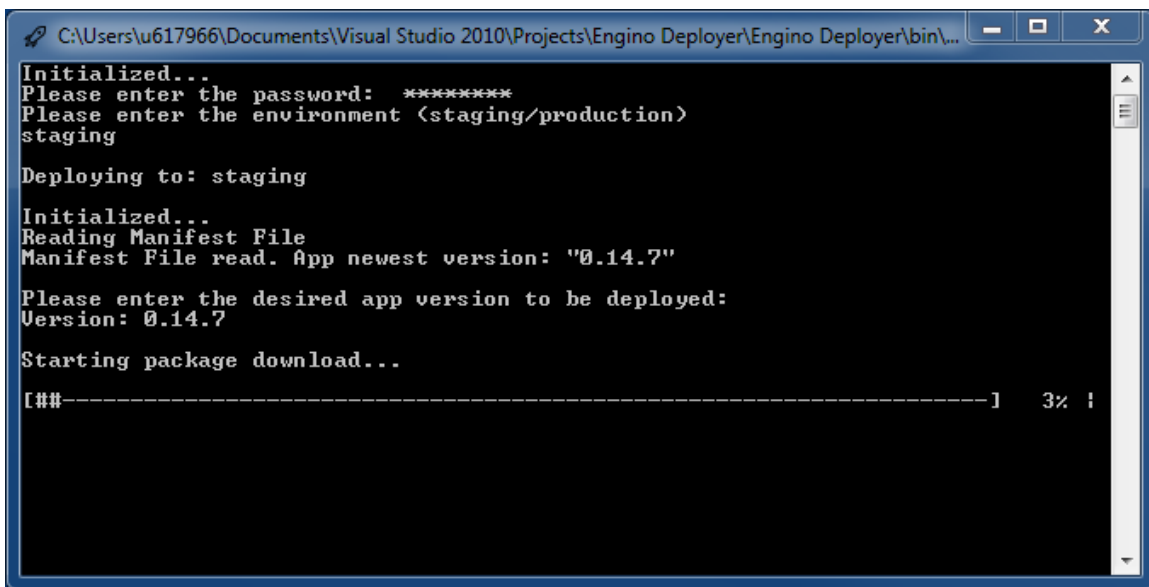


Figure 20 – Engino Deployer

<sup>5</sup> Daemon is running the application in the background

# 7 Results

This chapter discusses how well this project helped to solve the problems in the chapter 4. These problems were summarized by the items in 4.3. The solutions are described in the sections below:

- Development workflow and productivity comparison: describing how the development workflow has been improved, solving problems of developers not having high productivity, project managers to have better insights;
- Freedom for using tools: which describes the results of having this freedom, which solves most of the problems, related to stability, security, easier migrations between servers, and being able to use state-of-the-art libraries. The latter are the most impacting, since it, when well applied, solves the problems of code reuse, being able to quickly add new features, fix bugs, etc;
- Version Control: which describes how using version control helped the project to enable high productivity when working in teams and having a history of changes.

## 7.1 Development workflow

### 7.1.1 Milestones burndown charts

As seen in the [Issues and Milestones planning](#), we can plan milestones as being long term objectives and issues as short term.

By setting the dates in the milestones and associating issues with it, we are able to generate very insightful “burndown” charts. They are used to follow the development of the project and manage resources.

Since the issues in the project were create under the milestone, we can see the burndown charts for both the first and the second Milestones below (both are to the status of 2. ):

This give a very good insight in how the project planning is going, if it is delayed or on time and how many issues are still to be developed.

### 7.1.2 Freedom for using tools

It is well known that the more restrictions there are in a project, the less cost/benefit it will turn out to be.

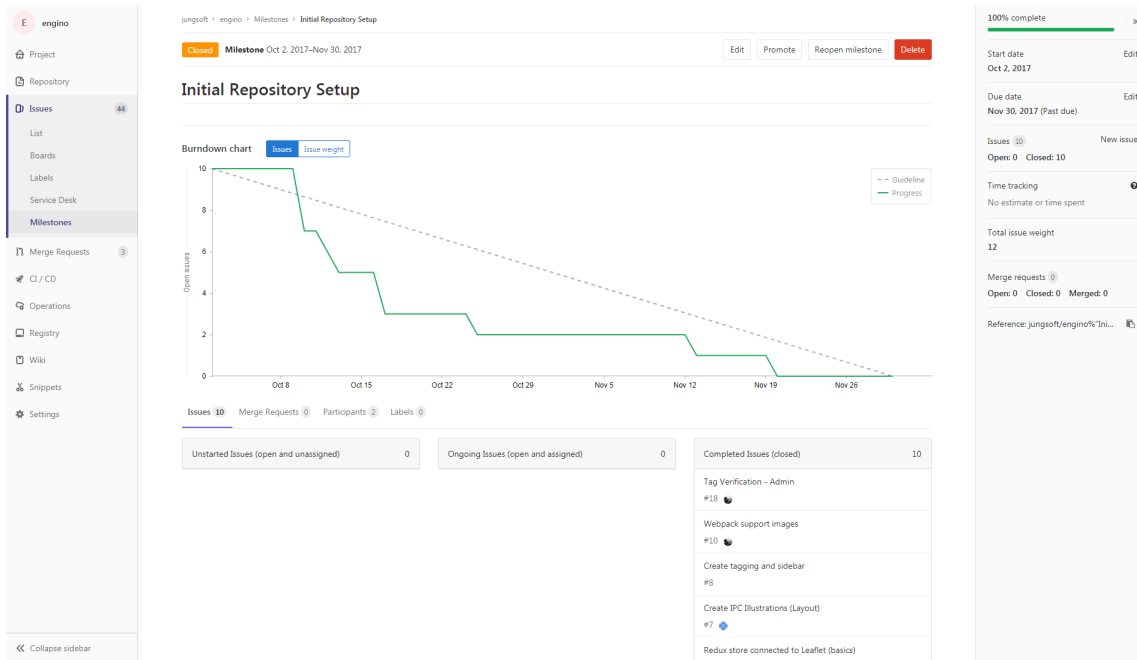


Figure 21 – Burndown Milestone 1

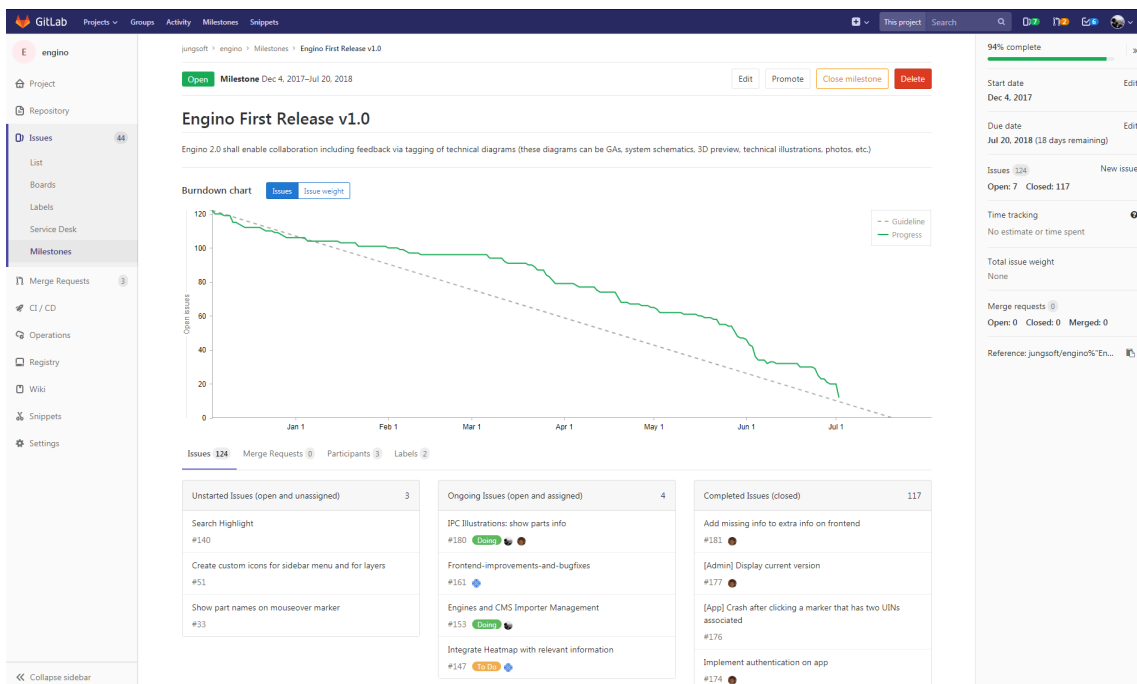


Figure 22 – Burndown Milestone 2

Through the freedom to use any libraries needed, many tasks were very fast to be done and changes in the source code, which was able to be developed as multiple modules could be done very quickly.

The freedom to use any tool can have a great impact on many areas of the development workflow and the final software. It is though to note, that this power should be used with responsibility.

This freedom is directly connected to the ability of developing high tech solutions, since there is no restrictions imposed in the software development.

This freedom extends though all parts of the application:

- Through the dependency-free releases, there is no restrictions on to which server to use and which server tools;
- Through an unrestricted development environment, there are no restrictions to which tools to be used on the workstations;
- Through using online tools, there are no restrictions to where the development team needs to physically be at.

This leads gives the possibility of using high end technologies in restricted environments, such as of big companies. Thus being able to keep up with the very competitive engineering market.

Two examples that were made possible only by this freedom of using the tools available is the authentication and the search. Because of the databases and libraries being used before were restricted and very old, thus with bad performance, the search was very slow and inefficient. The authentication method is made possible by connecting directly to the LDAP server, which was not possible with the tools restricted by the previous environment.

### 7.1.3 Version Control

The new development workflow not only enables using source control, but enforces it, since is made intrinsically together with it.

The advantages of using source control are very well known, but some features could be listed below:

- **Rollback:** When a non-working code is deployed to the server, by generating and keeping track of releases, a rollback to any previous version is made possible with only one command;
- **Bug tracking:** Through Git's commit history it is possible to track where a bug was added and to which feature this problem was associated;
- **Software development history:** We can easily track the software development flow. Adding this to an online project management tool, the project management is much simpler;

- **Collaboration:** Though using a Version Control System, collaborating to a common repository is made very straightforward. This would cause many problems without source control, such as loosing code in merges, etc.

## 7.2 Productivity comparison

The most important aspect in a software development team is productivity. This is exactly the reason why new technologies are created every day: to raise productivity.

A very good way to compare the productivity is to compare the development of 2 very similar features in the old and in the new development environment. Image 23 shows the productivity comparison between the old and the new system:

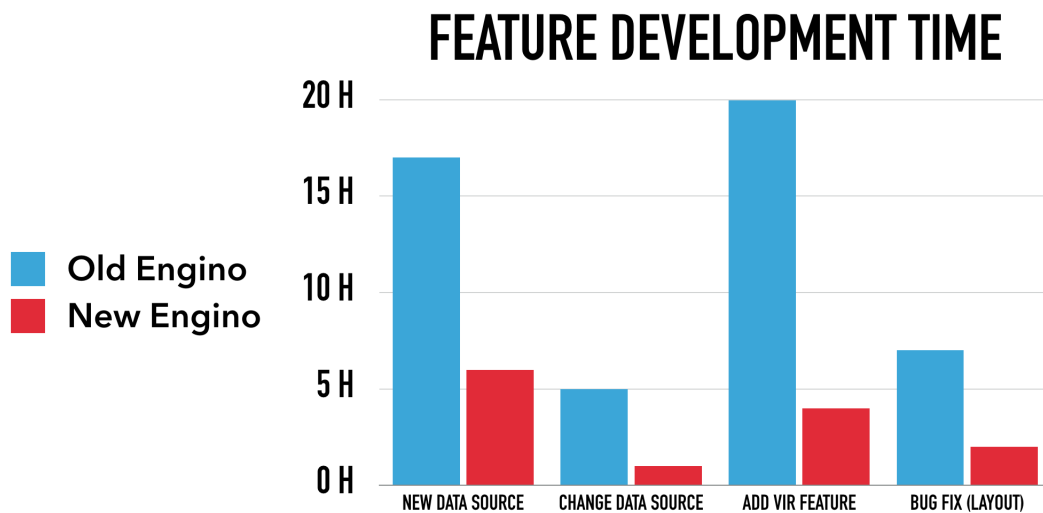


Figure 23 – Productivity Comparison

### 7.2.1 Case Study - new feature development speed

The feature was very simple: add a button in the frontend, load a table in the database and show a table when this button was clicked. This was the first feature developed still in the “Old Engine”.

Even though this sounds very simple, it took roughly a full week to be completely developed. Most of the problems were related to the development environment, such as the code being very difficult to read and many bugs were happening, which were very difficult to track.

This same feature was also added lately to the “New Engine” and even though it was much more complex, since it included an uploader interface to load the data, it took less than a full working day to be completely done.

## 7.2.2 Case Study - new developer in the team

In every company, the team changes constantly. For this reason, another very important aspect is how much time a new developer will take to understand the code and be able to start producing value.

For this case study, the comparison will be made between the first month with the “Old Engine” [4.3] and the first days of a new developer with the new Engine.

### 7.2.2.1 Old Engine

During the whole first couple of months many features were added, such as the VIR button above [7.2.1]. All of them were very simple and requires more or less the following steps:

- Create a new table in the database;
- Add a button in the frontend;
- Integrate this button to the database.

This was repeated about 5 times, for different data sources. All data was always loaded directly into the database, though an Excel file. Even though the features were very simple to be done, it took roughly one full month to make them.

A new developer entered the team a week ago. The first task was to create the authentication to the Rolls-Royce LDAP server.

Even though this task was much more complex than all 5 tasks from the old engine combined, it took only about 3 days to be completely done, since much of the code could be reused and it was very easy to get started with it.





# 8 Future developments

## 8.1 Conclusion

Since 2000, more than half of the Fortune 500 companies have either gone bankrupt, been acquired or gone out of business [4]. Digital disruption has a large share of responsibility in this. The technology/software market is changing constantly, with new products, technologies, libraries, languages, paradigms, etc. are being released every day. With this amount of change, some companies simply can't keep up.

The biggest reason companies have a hard time adapting to be competitive are not technical, but cultural. That means processes, paradigms, methodologies, etc. that change and companies do not know how to adapt, which are left to the old practices. This change in the culture, especially in big companies, can be really hard and very expensive.

The main objective of this project was to implement modern software development practices in a company that still have very old systems and paradigms, with regards to software. This enabling high productivity and the ability to adapt faster, thus becoming more competitive. Since this project is very general, as well as web development, it can be applied in many other projects, to use the same base and practices, with very little effort.

Since the biggest problems are cultural, such a big swift in the teams culture would be really difficult. Having a modular, general architecture, that could be applied, to some extend, to all software projects, could facilitate its implementation. This allows for gradually shifting the software development practices to a more modern era.

## 8.2 Further development

The complete software developed was made in a “plug-and-play” way, meaning many of its individual parts could be applied easily to other softwares and other teams. This includes database, the server software, backend technologies, frontend technologies, the CI system, and even simply the practices. Also, it is very easy to keep adding features and extending even more the software developed.

For example, some new features planned for Engino are:

- Add new data sources, which can connect information through relating to parts;
- Improvements in the image processing scripts;

- Add information about date and responsible person to every piece of data saved in the software;
- Heatmaps of costs, failure rates, number of quality issues, etc. for parts;
- Create walkthrough, for visually teaching users how the system works.

There are also multiple separated projects already planned, or being developed, that will use the same practices cited throughout this project and the server infrastructure. These software are all from different teams:

- **SAS**: Uses the base of Engino to show, on the same views, information about Gas Turbine Secondary Air System;
- **Shopino**: The same idea behind Engino, but adapted for the Shopfloor and with new features;
- **eBAM**: Management of tasks related to BAMs (report of anomalies during production), which is a Rolls-Royce process, to eliminate the necessity of using a paper based workflow.

# References

- [1] *Agile Manifesto*. English. URL: <http://agilemanifesto.org/> (cit. on p. 21).
- [2] *AngularJS*. English. URL: <https://angularjs.org/> (cit. on p. 26).
- [3] *Atlassian - CI vs CD vs CD*. English. URL: <https://www.atlassian.com/continuous-delivery/ci-vs-ci-vs-cd> (cit. on p. 22).
- [4] *Digital Transformation Is Racing Ahead and No Industry Is Immune*. English. URL: <https://hbr.org/sponsored/2017/07/digital-transformation-is-racing-ahead-and-no-industry-is-immune-2> (cit. on p. 71).
- [5] *Domain-Driven Design: Tackling Complexity in the Heart of Software*. 2014 (cit. on p. 23).
- [6] *Elixir*. English. URL: <https://elixir-lang.org/> (cit. on p. 28).
- [7] *Erlang*. English. URL: <https://www.erlang.org/> (cit. on p. 27).
- [8] *ES6 Specification*. English. URL: <http://www.ecma-international.org/ecma-262/6.0/index.html> (cit. on p. 29).
- [9] *Functional Programming In Java*. English. URL: <https://hackernoon.com/finally-functional-programming-in-java-ad4d388fb92e> (cit. on p. 27).
- [10] *Git SCM*. English. URL: <https://git-scm.com> (cit. on p. 24).
- [11] *GitLab*. English. URL: <https://about.gitlab.com/> (cit. on p. 24).
- [12] *GitLab - CI vs CD vs CD*. English. URL: <https://about.gitlab.com/2016/08/05/continuous-integration-delivery-and-deployment-with-gitlab/> (cit. on p. 22).
- [13] *GraphQL*. English. URL: <https://graphql.org/> (cit. on p. 28).
- [14] ECMA international. *ECMAScript: A general purpose, cross-platform programming language*. 1997. URL: <https://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,%201st%20edition,%20June%201997.pdf> (visited on ) (cit. on p. 29).
- [15] *JavaScript Object Notation*. English. URL: <https://www.json.org/json-en.html> (cit. on p. 25).
- [16] *Phoenix Framework*. English. URL: <http://phoenixframework.org/> (cit. on p. 28).
- [17] *React*. English. URL: <https://reactjs.org/> (cit. on p. 26).
- [18] *Rolls-Royce*. English. URL: <https://www.rolls-royce.com/> (cit. on p. 19).
- [19] *Rolls-Royce Corporate Presentation*. English. 2017 (cit. on p. 20).

- [20] *Webpack*. English. URL: <https://webpack.js.org/> (cit. on pp. 29, 30).

# APPENDIX A – Webpack Configuration File

```
1  const env = process.env.NODE_ENV || 'development'
2  const isProduction = (env === 'production')
3
4  const webpack = require('webpack');
5
6  const path = require('path');
7  const ExtractTextPlugin = require('extract-text-webpack-plugin');
8  const ResolveEntryModulesPlugin = require('resolve-entry-modules-webpack-plugin');
9
10 function join(dest) { return path.resolve(__dirname, dest); }
11
12 function web(dest) { return join('frontend/' + dest); }
13
14 const publicPath = `http://${process.env.HOST || 'localhost'}:4001/`;
15
16 const devServerClient = isProduction ? ['babel-polyfill'] : [
17   'babel-polyfill',
18
19   'react-hot-loader/patch',
20   // activate HMR for React
21
22   'webpack-dev-server/client?' + publicPath,
23   // bundle the client for webpack-dev-server
24   // and connect to the provided endpoint
25
26   'webpack/hot/only-dev-server',
27   // bundle the client for hot reloading
28   // only- means to only hot reload for successful updates
29 ];
30
31 module.exports = {
32   devtool: isProduction ? 'source-map' : 'eval',
33
34   devServer: {
35     headers: {
36       "Access-Control-Allow-Origin": "*",
37       "Access-Control-Allow-Methods":
38         "GET, POST, PUT, DELETE, PATCH, OPTIONS",
```

```
39     "Access-Control-Allow-Headers":
40         "X-Requested-With, content-type, Authorization"
41     },
42     contentBase: join('priv/static'),
43     port: 4001,
44     hot: true,
45     publicPath,
46     stats: {
47         colors: true,
48         version: false,
49         chunkModules: false
50     }
51 },
52
53 entry: {
54     app: devServerClient.concat([
55         web("app/css/app.scss"),
56         web('app/main.jsx')
57     ]),
58     admin: devServerClient.concat([
59         web('admin/main.jsx')
60     ]),
61 },
62
63 output: {
64     path: join('priv/static'),
65     publicPath: isProduction ? "/" : publicPath,
66     filename: 'js/[name].js',
67 },
68
69 resolve: {
70     extensions: ['.js', '.jsx', '.json'],
71     modules: ['node_modules', 'frontend/assets/images', web('')]
72 },
73
74 module: {
75     rules: [
76         {
77             test: /\.js$/,
78             exclude: /node_modules/,
79             loaders: [
80                 {
81                     loader: 'babel-loader',
82                 },
83                 { loader: 'haml-jsx-loader' }
```

```
84     ]
85   },
86   {
87     test: /^(?!cssmodule).*\. (sass|scss)$/,
88     loaders: [
89       { loader: 'style-loader' },
90       { loader: 'css-loader' },
91       { loader: 'sass-loader' }
92     ]
93   },
94   {
95     test: /\.css$/,
96     loaders: [
97       { loader: 'style-loader' },
98       { loader: 'css-loader' }
99     ]
100  },
101  {
102    test: /\.(gif|png|jpe?g|svg)$/i,
103    loaders: [
104      {
105        loader: 'file-loader',
106        options: {
107          name: 'images/[name]-[hash].[ext]'
108        }
109      }
110    ]
111  },
112  {
113    test: /\.(ttf|woff2?|eot)$/,
114    loader: "file-loader",
115    options: {
116      name: 'fonts/[hash].[ext]'
117    }
118  }
119 ],
120 },
121
122 plugins: [
123   new webpack.NamedModulesPlugin(),
124   new webpack.NoEmitOnErrorsPlugin(),
125   new webpack.ProvidePlugin({
126     $: "jquery",
127     jQuery: "jquery",
128     "window.jQuery": "jquery",
```

```
129     Popper: ['popper.js', 'default']
130   }),
131   new ResolveEntryModulesPlugin(web(''))
132 ].concat(isProduction ? [
133   // Production only Plugins
134   new webpack.DefinePlugin({
135     'process.env.NODE_ENV': JSON.stringify('production')
136   }),
137   new webpack.optimize.OccurrenceOrderPlugin(),
138   new webpack.optimize.UglifyJsPlugin({
139     sourceMap: true,
140     minimize: true,
141     compress: {
142       warnings: false,
143       screw_ie8: true,
144       conditionals: true,
145       unused: true,
146       comparisons: true,
147       sequences: true,
148       dead_code: true,
149       evaluate: true,
150       if_return: true,
151       join_vars: true
152     },
153     output: {
154       comments: false, // remove all comments
155     },
156   }),
157   new webpack.optimize.ModuleConcatenationPlugin(),
158 ] : [
159   // Development only plugins
160   new webpack.HotModuleReplacementPlugin()
161 ]),
162 }
```



# APPENDIX B – GitLab CI Configuration

```
1 variables:
2   MIX_ENV: "test"
3   POSTGRES_DB: engino_test
4   POSTGRES_USER: engino
5   POSTGRES_PASSWORD: Og78g807F8G078GZ078ZG787ZGV8Z9
6
7 stages:
8   - test
9   - staging
10  - production
11
12 backend_elixir:
13   tags:
14     - docker
15   image: elixir:1.6.5
16   services:
17     - postgres:10.1
18   stage: test
19   before_script:
20     - mix local.rebar --force
21     - mix local.hex --force
22     - mix deps.get
23   script:
24     - mix test
25     - mix coveralls
26   coverage: /\[TOTAL\]\s+(\d+\.\d+)/
27
28 backend_rails:
29   tags:
30     - docker
31   image: registry.gitlab.com/jungsoft/engino/build/test:rails
32   stage: test
33   variables:
34     RAILS_ENV: test
35   before_script:
36     - cd engino-rails-api
37     - bundle install
38   script:
39     - bundle exec rake test
40
41 frontend:
```

```
42 tags:
43   - docker
44 image: node:8.7.0
45 stage: test
46 before_script:
47   - yarn install
48 script:
49   - npm test #-- -u --coverage
50
51 build_release:
52   tags:
53     - docker
54 image: registry.gitlab.com/jungsoft/engino/build/production:centos7
55 stage: production
56 variables:
57   REPLACE_OS_VARS: "true"
58   MIX_ENV: "prod"
59   NODE_ENV: "production"
60 before_script:
61   # Install project dependencies
62   - mix deps.get
63   - yarn install --prod
64 script:
65   # Install Ruby libraries
66   - bundle install --gemfile=engino-rails-api/Gemfile --path=gems
67
68   # Build elixir release
69   - mkdir -p priv/static
70   - ./node_modules/webpack/bin/webpack.js
71   - mix phx.digest
72   - mix release
73   - VERSION=`mix release.version show`
74   - mv _build/prod/rel/engino/releases/$VERSION/engino.tar.gz engino.tar.gz
75
76   # Copy release to AWS
77   - aws s3 cp engino.tar.gz s3://$PATH
78 environment:
79   name: production
80 only:
81   - master
82 artifacts:
83   paths:
84     - engino.tar.gz
```

# APPENDIX C – Production Dockerfile

```

1 FROM centos:7
2
3 # Dockerfile author / maintainer
4 LABEL maintainer="Rafael Jung <jungrafael@jungsoft.com.br>"
5
6 ARG RUBY_INSTALLATION_PATH=/var2/data/engino/ruby
7
8 ENV HOME /root
9
10 # Prerequisites for `all`
11 ARG NUM_CPU=8
12 RUN yum -y update && yum -y upgrade
13 RUN yum install -y epel-release
14 RUN yum install -y curl gcc-c++ make m4 file pkgconfig perl expat-devel \
15     zlib-devel python-hashlib gettext ncurses-devel openssl-devel \
16     wget bzip2 bzip2-libs bzip2-devel zip unzip git which openssh-clients \
17     rsync readline-devel mesa-libGLU-devel readline-devel sqlite-devel \
18     tk-devel db4-devel gdbm-devel postgresql-devel libaio
19 ADD clean /root/
20 WORKDIR /tmp
21
22 RUN curl -L https://$GOON_PATH/goon_linux_amd64.tar.gz | tar xvz && mv goon /bin/
23
24 # Install Oracle Client
25 RUN mkdir -p /opt \
26     && mkdir /opt/oracle \
27     && cd /opt/oracle \
28     && wget https://$ENGINO_TOOLS_PATH/instantclient-basic-linux.x64-12.1.0.2.0 \
29     && unzip instantclient-basic-linux.x64-12.1.0.2.0 \
30     && rm instantclient-basic-linux.x64-12.1.0.2.0 \
31     && wget https://$ENGINO_TOOLS_PATH/instantclient-sdk-linux.x64-12.1.0.2.0 \
32     && unzip instantclient-sdk-linux.x64-12.1.0.2.0 \
33     && rm instantclient-sdk-linux.x64-12.1.0.2.0 \
34     && wget https://$ENGINO_TOOLS_PATH/instantclient-sqlplus-linux.x64-12.1.0.2.0 \
35     && unzip instantclient-sqlplus-linux.x64-12.1.0.2.0 \
36     && rm instantclient-sqlplus-linux.x64-12.1.0.2.0 \
37     && cd instantclient_12_1 \
38     && ln -s libclntsh.so.12.1 libclntsh.so
39
40 ENV LD_LIBRARY_PATH="/opt/oracle/instantclient_12_1"
41

```

```
42 # Configure default locale to UTF-8 -> Elixir
43 RUN localedef -c -f UTF-8 -i en_US en_US.UTF-8 \
44     && export LC_ALL=en_US.UTF-8
45
46 # Install PIP
47 RUN curl https://bootstrap.pypa.io/get-pip.py >> get-pip.py \
48     && python get-pip.py \
49     && pip install awscli \
50     && /root/clean
51
52 # Install `node` and `yarn`
53 RUN curl --silent --location https://rpm.nodesource.com/setup_8.x | bash - \
54     && yum install -y nodejs
55
56 # Install `yarn` - https://yarnpkg.com/lang/en/docs/install/#linux-tab
57 RUN curl --silent --location \
58     https://dl.yarnpkg.com/rpm/yarn.repo | tee /etc/yum.repos.d/yarn.repo \
59     && yum install -y yarn
60
61 # Install `erlang`
62 RUN wget https://packages.erlang-solutions.com/erlang-solutions-1.0-1.noarch.rpm \
63     && rpm -Uvh erlang-solutions-1.0-1.noarch.rpm \
64     && yum install -y esl-erlang
65
66 # Install `elixir`
67 ENV LC_ALL=en_US.UTF-8
68 RUN wget https://github.com/elixir-lang/elixir/archive/v1.6.5.zip \
69     && unzip v1.6.5 \
70     && mv elixir-1.6.5 /opt/elixir \
71     && cd /opt/elixir && make clean test \
72     && /root/clean
73 ENV PATH="/opt/elixir/bin:${PATH}"
74
75 RUN mix local.hex --force && mix local.rebar --force
76
77 # Install Ruby
78 RUN wget http://cache.ruby-lang.org/pub/ruby/2.5/ruby-2.5.1.tar.gz \
79     && tar -xvzf ruby-2.5.1.tar.gz && rm ruby-2.5.1.tar.gz && cd ruby-2.5.1 \
80     && ./configure --disable-shared --prefix=$RUBY_INSTALLATION_PATH \
81     && make -j $NUM_CPU && make install \
82     && /root/clean
83
84 ENV PATH="$RUBY_INSTALLATION_PATH/bin:${PATH}"
85
86 CMD ["/bin/bash"]
```