Estudiants: Roger Arquès Vall i Marc Ribalta Gené.

Titulació: Grau en Enginyeria Informàtica

Títol de Treball Final de Grau: Web app to integrate, compare and visualize DNA sequences.

Directors: Marta Oliva Solé i Joan Fibla Palazón.

Presentació
Mes: Maig
Any: 2018
Abstract

The purpose of this project is to provide a web application to biology researchers interested in the salmo-trutta specie.

The team has developed a platform that makes possible the search and comparison of information, introduced by the Genetics and Biodiversity research team.

This document provides an explanation of how the project has been developed using agile methodologies and tools, an understanding of Spring and Angular development frameworks, a study of NoSQL and relational databases, and the definition of a REST API.
Acknowledgements

Firstly, we would like to express our sincere gratitude to our professor Marta Oliva Sole for the continuous support in our project, for her patience, motivation, and advising. Also, we would like to express our sincere thanks to Joan Fibla Palazón, not only because he gave us the opportunity to develop this project, but for his patience and continuous support during all the process.

Thank you, Juan Manuel Gimeno Illa, for your advising about software architectures and helping us to solve database problems.

Sincere thanks also to Roberto Gonzalez García, who taught us web development, continuous integration and deployment, and let us practice how to deploy using his research group server.

Thank you, Toni Granollers Saltiveri, for all the feedback you have given us about the usability of our web front-end.

Last but not the least, we would like to thank our families for supporting us and for their patience during all the project.
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2.4 Column Oriented Databases</td>
<td>28</td>
</tr>
<tr>
<td>5.2.5 Key-Value Databases</td>
<td>29</td>
</tr>
<tr>
<td>5.2.6 Polyglot persistence</td>
<td>29</td>
</tr>
<tr>
<td>5.2.7 The decision</td>
<td>30</td>
</tr>
<tr>
<td>5.3 Server side technologies</td>
<td>33</td>
</tr>
<tr>
<td>5.3.1 Java</td>
<td>33</td>
</tr>
<tr>
<td>5.3.2 Spring with Spring Boot</td>
<td>33</td>
</tr>
<tr>
<td>5.3.3 Testing with Cucumber</td>
<td>33</td>
</tr>
<tr>
<td>5.4 Client side technologies</td>
<td>34</td>
</tr>
<tr>
<td>5.4.1 Angular 5</td>
<td>34</td>
</tr>
<tr>
<td>5.4.2 Bootstrap</td>
<td>34</td>
</tr>
<tr>
<td>5.5 Version Control System (VCS)</td>
<td>35</td>
</tr>
<tr>
<td>5.5.1 Git</td>
<td>35</td>
</tr>
<tr>
<td>5.5.2 Github</td>
<td>35</td>
</tr>
<tr>
<td>5.6 Continuous Integration</td>
<td>36</td>
</tr>
<tr>
<td>5.6.1 TravisCI</td>
<td>36</td>
</tr>
<tr>
<td>5.7 Packaging and deployment</td>
<td>38</td>
</tr>
<tr>
<td>5.7.1 Docker and Docker Compose</td>
<td>38</td>
</tr>
<tr>
<td>6 Sprints</td>
<td>39</td>
</tr>
<tr>
<td>6.1 Context Diagram</td>
<td>39</td>
</tr>
<tr>
<td>6.2 Sprint 1</td>
<td>41</td>
</tr>
<tr>
<td>6.2.1 Scope of the Project</td>
<td>41</td>
</tr>
<tr>
<td>6.2.2 Roles, Tools and Technologies</td>
<td>41</td>
</tr>
<tr>
<td>6.2.3 Understanding of the information and</td>
<td>41</td>
</tr>
<tr>
<td>competitive analysis</td>
<td></td>
</tr>
<tr>
<td>6.2.4 First Prototype</td>
<td>42</td>
</tr>
<tr>
<td>6.3 Sprint 2</td>
<td>44</td>
</tr>
<tr>
<td>6.3.1 The Database options</td>
<td>44</td>
</tr>
<tr>
<td>6.3.2 The second prototype</td>
<td>44</td>
</tr>
<tr>
<td>6.4 Sprint 3</td>
<td>47</td>
</tr>
<tr>
<td>6.4.1 New changes into the prototype</td>
<td>47</td>
</tr>
<tr>
<td>6.4.2 Decided database</td>
<td>48</td>
</tr>
<tr>
<td>6.4.3 First database architecture design</td>
<td>49</td>
</tr>
<tr>
<td>6.4.4 Starting the development of the server</td>
<td>50</td>
</tr>
<tr>
<td>6.5 Sprint 4</td>
<td>52</td>
</tr>
<tr>
<td>6.5.1 Fasta upload</td>
<td>52</td>
</tr>
<tr>
<td>6.5.2 Search by trinity id</td>
<td>54</td>
</tr>
<tr>
<td>6.5.3 Angular Front-End</td>
<td>55</td>
</tr>
<tr>
<td>6.5.4 Bootstrap with Angular</td>
<td>58</td>
</tr>
<tr>
<td>6.6 Sprint 5</td>
<td>60</td>
</tr>
<tr>
<td>6.6.1 Updated MongoDB design</td>
<td>60</td>
</tr>
<tr>
<td>6.6.2 Upload Interpro Files</td>
<td>62</td>
</tr>
<tr>
<td>6.6.3 Angular file uploading module</td>
<td>63</td>
</tr>
</tbody>
</table>
6.6.4 Like Searches ........................................... 64
6.6.5 Custom Fasta files download ........................... 65
6.6.6 Front-end changes ..................................... 66
6.6.7 Other back-end issues solved ........................ 68

6.7 Sprint 6 .................................................. 69
   6.7.1 Responsive Front-end ............................... 69
   6.7.2 Adding Interpro details ............................ 70
   6.7.3 MongoDB interaction and architecture .......... 70

6.8 Sprint 7 .................................................. 74
   6.8.1 Refactoring the code ............................... 74
   6.8.2 Controller refactorization and general cleanup 75
   6.8.3 New File Manager ................................. 77
   6.8.4 Changed architecture .............................. 79
   6.8.5 Defined REST API .................................. 79
   6.8.6 Upload file component ............................ 80
   6.8.7 Family component ................................. 81
   6.8.8 Pager component .................................. 83

6.9 Sprint 8 .................................................. 85
   6.9.1 Final changes in the Back-End .................... 85
   6.9.2 Database Final Architecture ....................... 87
   6.9.3 Ending the REST API ............................. 88
   6.9.4 Front-end ......................................... 89
   6.9.5 Testing and introducing TravisCI ................. 90
   6.9.6 BLAST ............................................. 90
   6.9.7 Docker and Docker-Compose ....................... 93
   6.9.8 Deployment ....................................... 96

7 Conclusions .............................................. 98

8 Personal conclusions .................................... 99
   8.1 Marc’s conclusions .................................. 99
   8.2 Roger’s conclusion .................................. 99

9 Future Work ............................................ 101

List of Figures

1  Kanban Board Example .................................. 4
2  Scrumban Zenhub board, part 1 ....................... 8
3  Scrumban Zenhub board, part 2 ....................... 9
4  An epic issue of the project ......................... 11
5  Pull request failing .................................... 13
6  Detailed log of the failing Pull request in TravisCI 13
7  Pull request succeeded ................................. 13
<table>
<thead>
<tr>
<th>Page</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>Detailed log of the succeeded Pull request in TravisCI</td>
</tr>
<tr>
<td>9</td>
<td>Chickpea Transcriptome Database [1]</td>
</tr>
<tr>
<td>10</td>
<td>Brain Transcriptome Database [2]</td>
</tr>
<tr>
<td>11</td>
<td>Chickpea Transcriptome Database Listing</td>
</tr>
<tr>
<td>12</td>
<td>Brain Transcriptome Database Listing</td>
</tr>
<tr>
<td>13</td>
<td>Blast SalmonDB</td>
</tr>
<tr>
<td>14</td>
<td>Social Graph</td>
</tr>
<tr>
<td>15</td>
<td>Example Document Database Structure</td>
</tr>
<tr>
<td>16</td>
<td>Relational Model vs Column Oriented</td>
</tr>
<tr>
<td>17</td>
<td>Relational Model Example</td>
</tr>
<tr>
<td>18</td>
<td>Continuous Integration Workflow</td>
</tr>
<tr>
<td>19</td>
<td>Context Diagram</td>
</tr>
<tr>
<td>20</td>
<td>Prototype 1 Overview</td>
</tr>
<tr>
<td>21</td>
<td>Prototype 2 Overview</td>
</tr>
<tr>
<td>22</td>
<td>Prototype Final Overview</td>
</tr>
<tr>
<td>23</td>
<td>First database architecture diagram</td>
</tr>
<tr>
<td>24</td>
<td>Structure of the server - Sprint 3</td>
</tr>
<tr>
<td>25</td>
<td>Family repository - Sprint 3</td>
</tr>
<tr>
<td>26</td>
<td>Storage Service</td>
</tr>
<tr>
<td>27</td>
<td>Upload file, first design</td>
</tr>
<tr>
<td>28</td>
<td>Ajax request</td>
</tr>
<tr>
<td>29</td>
<td>Angular SPA Schema</td>
</tr>
<tr>
<td>30</td>
<td>Snippet of the Angular router</td>
</tr>
<tr>
<td>31</td>
<td>Angular Component annotation</td>
</tr>
<tr>
<td>32</td>
<td>Main NgModule of the application</td>
</tr>
<tr>
<td>33</td>
<td>Simplified Bootstrap Schema</td>
</tr>
<tr>
<td>34</td>
<td>New file upload page</td>
</tr>
<tr>
<td>35</td>
<td>Some of the repository methods</td>
</tr>
<tr>
<td>36</td>
<td>Result pagination</td>
</tr>
<tr>
<td>37</td>
<td>Sequence Detail</td>
</tr>
<tr>
<td>38</td>
<td>Folded Navigation Bar</td>
</tr>
<tr>
<td>39</td>
<td>ExperimentManager facade</td>
</tr>
<tr>
<td>40</td>
<td>Structure before refactoring</td>
</tr>
<tr>
<td>41</td>
<td>Structure after refactoring</td>
</tr>
<tr>
<td>42</td>
<td>Storage service classes</td>
</tr>
<tr>
<td>43</td>
<td>API table definition</td>
</tr>
<tr>
<td>44</td>
<td>Sequence GET method implementation</td>
</tr>
<tr>
<td>45</td>
<td>Wait icon when the file is being processed</td>
</tr>
<tr>
<td>46</td>
<td>Error icon and message if an error has occurred uploading</td>
</tr>
<tr>
<td>47</td>
<td>Family details screen with button</td>
</tr>
<tr>
<td>48</td>
<td>All sequences from a family</td>
</tr>
<tr>
<td>49</td>
<td>Family sequences in details page</td>
</tr>
<tr>
<td>50</td>
<td>Family search by description</td>
</tr>
<tr>
<td>51</td>
<td>rePage method code</td>
</tr>
</tbody>
</table>
1 Introduction

1.1 Context

Joan, the client of the project, is a researcher in the Genetics and Biodiversity research team, and they investigate the differential metabolic profiles associated to movement behavior of salmo-trutta specie.

During the research, a lot of information is obtained and is hard to use without computer tools. This is why a web page that can control all this information is needed. Using the web page, all the information files can be saved and the data they contain can be treated by then obtain more uses like sorting or listing by property. The main goal is to provide this information to the rest of the world, providing tools in a web page so people can interact with the research group data using their own.

When working with all this information, there will be a basic part, which will have additional data added later. Using a fasta file, the primary data about the salmo-trutta will be added, containing data about the sequence which has an identifier and a DNA sequence. After introducing this primary data, other biological known terms like Interpro, Go, or Kegg can be added. This terms group sequences that have similar properties.

1.2 Objective

The objective of this project is to build a web app, that will be used by the Genetics and Biodiversity Research Team of Lleida, to difuse DNA sequences of the Salmo-Trutta for other biology researchers to visualize and compare.

There are a lot of biological web apps in the market and all follow the same pattern. This project will try to do it better, following better HCI (Human-Computer Interaction) rules, give additional documentation, and incorporating an API making the data accessible for other web pages.

Although the project will not have many features due the lack of time, the team will try to make the structure as clean and flexible as possible. One of the core things the project has, is that all the code is open-source in github, so anyone can read and use the code, even contribute to it.

The aim is not to develop as many features as possible, but develop them well, using tools and methodologies to interact with the client and work as a team.
1.3 Document structure

The contents of this document are divided basically into 6 main blocks.

1. **Methodology** Treats how the project has been managed, which development methodology has been followed and the used work flow.

2. **Competitive analysis** Different analysis carried out in similar web sites and projects to extract possible features for this project.

3. **State of the Art** Analysis of the structure and meaning of the data, DNA sequences and other complementary data.

4. **Used Technologies** Contains an in depth analysis of the different database types in order to select the best choice for the data this project operates with. Also contains all technologies regarding development and methodology such as programming languages, frameworks, CI tools, etc.

5. **Sprints** This block contains all the development process of the project. Each sprint contains the requirements or features developed together with the problems encountered and the respective solutions.

6. **Conclusions and future work** Technical conclusions of the project and future work that can potentially be done. A personal conclusion from the developers is also included at the end.
2 Methodology

2.1 Agile methodologies

For the development of the project, the decision to use agile methodologies has been made. The methodology used is Scrumban, which is a fusion between Scrum and Kanban.

2.1.1 Scrum

The main purpose of Scrum is to work in iterations called Sprints. A sprint is a period of time with a maximum length of one month, in which the team produces different features.

Every Sprint starts with a team meeting where the work to do is decided. The result of the meeting is the Sprint Backlog, which needs to be done during the sprint.

A Sprint contains three types of meetings:

1. Sprint Planning: A discussion to determine which items from the product backlog will be done during the sprint.

2. Daily Scrum: A short meeting where the team coordinates the work for the following day or the same day, depending if the meeting is done at the end of the day or at the beginning.

3. Sprint Review: At the end of the sprint, the team, including stakeholders and client, review the result of the sprint. The purpose of this meeting is to demonstrate the new features to the stakeholders and product owner, getting feedback and new requirements.

4. Sprint Retrospective: After the Sprint Review, the team meets and tries to identify the issues during the sprint, trying to make adjustments for the next one.

A scrum team has three roles:

1. Product Owner: The team member that is responsible for managing the product backlog (a list of possible changes) and solving team conflicts in what to build.

2. Scrum Master: The responsible of ensuring the team following the agile principles and best practices. Normally, this role is assumed by an expert, who has the ability to coach other members.

3. Developers: The team members responsible of ensuring that the product increments in every sprint.
2.1.2 Kanban

When working in team, sometimes the work flow is difficult to manage and improve. Kanban \cite{4} is a methodology that provides help with good communication and easy management of the work flow.

When using kanban, a Kanban Board is normally used. A Kanban Board consists of a table with different columns, where every column has a different meaning. These columns, normally, contain user stories, which are introduced by the product owner, containing further explanation of the client requirement (also, other features can be included).

![Kanban Board Example](image)

Figure 1: Kanban Board Example

When using a Kanban Board, two roles are needed. First, the Service Request Manager, who introduces the requested expectations of a customer and helps to understand them. Second, the Service Delivery Manager or flow manager, who controls the Kanban issues (when to deliver, who develops those).
2.1.3 Scrumban

Sometimes, the project needs to use Scrum and Kanban. When this happens, the team uses Scrumban instead.

Scrumban is a fusion between Scrum and Kanban which changes some of their aspects:

- There are no iterations, sprints are deleted and continuous flow is introduced.
- There is no estimation, all the issues have a similar size.
- There are no specified roles, only the team. Roles are added when needed.
- Changes can be done when necessary (in Scrum the changes are planned in the next sprint).
2.2 How is Scrumban adapted into the project?

Scrumban has been used in the project, but, some custom changes have been made ensuring control over the inexpirience when deciding work flow.

During the project, four people have been involved: Marta, Joan, Roger, and Marc. Joan is the person explaining what the project should contain or not, the role of Client. Marta, as advisor in this project and expert in requirement engineering, had the team role of Product Owner and Scrum Master (when was necessary some assistance in one of those roles). Roger and Marc acted as development team, with the addition of getting requirements and managing the work flow (getting aid from Marta when something was out of place).

As explained in 2.1.3, there are no sprints in a normal Scrumban project, but again, because of the team inexpirience, the team did a bit of Sprint Planning in some meetings and introduced Sprint deadlines. When ending a meeting, the next one was decided without knowing if was necessary. This was because the lack of coordination and time availability of the team.

In summary, the team used Scrumban, adding parts of the Sprint concept instead of using continuous flow alone.

2.3 Sprint Meetings

During the development of the project, a series of meetings were carried out at the end of each Sprint. All the members of the team were present in all of the meetings. The meetings acted as Sprint Reviews and Sprint Retrospectives, which are part of the Scrum methodology 2.1.1 and served two main purposes.

The first purpose is to collect new requirements for the following sprint. The client explains which features the application needs to have. These requirements are written down and scrutinized to figure out the features that form each requirement.

The second purpose is to review the features developed during the previous sprint and address any issue that might have appeared. The developed features are demonstrated to the client and the advisor in order to decide if the requirements are adequately fulfilled or not. If a requirement had been partially or wrongly fulfilled, the completion of the requirement is added as task in the Scrumban Board.\footnote{Scrumban Board is explained in the next section 2.5}

\footnote{Scrumban Board is explained in the next section 2.5}
2.4 Github projects

The project is open source and is developed in Github under an organization specially created for this purpose. The organization is called sttDB because of the name of the project: *Salmo-Trutta Transcriptome Data Base*, [https://github.com/sttDB](https://github.com/sttDB).

The organization contains two sub-projects that together form the whole product. The first project, [https://github.com/sttDB/sttDB-client](https://github.com/sttDB/sttDB-client), represents the graphical interface, interaction and usability of the web site. And the second project, [https://github.com/sttDB/sttDB-server](https://github.com/sttDB/sttDB-server), represents all the domain logic, interaction with the database and the REST API for developers.
2.5 Workflow with Zenhub

During the development of the project a Scrumban Board has been used. The used board consists of a Github plugin called Zenhub that integrates pipelines with the issue mechanism that Github provides. A Scrumban board consist of a series of columns each one representing a different stage of the issues that are being developed. The columns may also be called pipelines. In a typical Scrumban board, the tasks (or issues) have a time estimation, in this project no time estimation has been assigned to the tasks because of inexperience in planning.

The board used during the development consists of 6 columns or pipelines: Backlog, To Do (Low priority), To Do (High priority), In progress, Review/QA and Closed.

---

Figure 2: Scrumban Zenhub board, part 1

---

2Zenhub: [https://www.zenhub.com](https://www.zenhub.com)
2.5.1 Backlog

The Backlog pipeline contains the issues that will have to be done in the future but the moment to start developing them is not yet known and a developer hasn’t been assigned.

Issues do not appear magically in the Backlog pipeline. These issues are generated in the Sprint Meetings. For each feature, bug or enhancement discussed during the meetings, a new issue is added to the Backlog pipeline.

2.5.2 To Do Low priority and High priority

When an issue from the backlog column is fully defined, it can be moved to corresponding To Do pipelines. In the project’s Scrumban board there are two To Do pipelines which contain the issues which development can start immediately. The pipelines are 'To Do Low Priority' and 'To Do High Priority' which contain the low priority and the urging issues respectively.

The issues are prioritized by the whole team during the Sprint Meetings.
or by the developers in any point in time during the sprint basing on the dependencies between different issues or the client’s priorities.

Also, the issues in these pipelines have at least one developer assigned.

2.5.3 In progress

The 'In progress' pipeline is pretty straightforward, contains the issues that are being currently developed. During the project the number of issues in this pipelines is tried to be minimized to avoid saturation.

Since the number of issues in the pipeline is minimized, some bottleneck in the work flow may occur at this pipeline.

2.5.4 Review/QA

When the development of an issue is finished, it is moved to the Review and Quality Assurance pipeline, where the finished work is tested and reviewed.

After an issue is reviewed and tested two things can happen.

First, for some reason, changes may have to be performed, if this is the case, a new issue is created containing the changes to implement.

Second, the issue is approved and marked as completed and moved to the closed pipeline.

2.5.5 Closed

The 'Closed’ pipeline contains the issues that have been finished.

2.5.6 Epic issues

An Epic Issue is a special type of issue that Zenhub provides. It acts as goal that consists of many smaller issues. The small issues are normal Github issues. This mechanism allows to group many related issues and establish dependencies between them.
Figure 4: An epic issue of the project
2.6 Continuous Integration

2.6.1 What is Continuous Integration and why use it

The term Continuous Integration (CI) [6] [7] was originated as one of the twelve practices in the Extreme Programming [3] development process. CI is a development practice used widely by teams of developers where changes made by the team are integrated frequently during all the development process into a shared code repository, in the case of this project, Github.

According to Agile Alliance [4] CI has two main goals. The first one is to minimize the duration and effort of each integration, achieved by using a continuous integration server, automated testing and automated build systems. The second goal is to produce versions of the product that are suitable for release at any time.

Besides these two main goals, the team also receives a series of benefits from implementing Continuous Integration, such as increased Feedback, increased productivity, less debugging time, more communication between team members and rapid discovery and fixing of bugs in the product.

2.6.2 Continuous Integration implementation

The following explanation will cover how CI has been implemented by detailing a series of actions that take place whenever a new change made by one of the two developers is pushed to the Github repository. The technologies used for CI will be more detailed in the section 'Used Technologies'.

To correctly implement CI in the project the team has used Git on the Github platform as version control system, maven as build tool, various suites of unit tests with JUnit, integration tests with Cucumber and last, but not least, TravisCI [5] as continuous integration server.

TravisCI is integrated with the Github project and is programmed to start the CI process when new changes are pushed to the branch master or when a pull request from any branch to the master branch is submitted. When a developer has finished developing the current issue and is happy with the result, the new changes are committed and pushed to the appropriate branch of the Github repository.

Once a Pull Request is submitted from the developer’s branch to master and all conflicts from the possible merge have been solved, the CI process
triggers. TravisCI starts by checking out the source code of the pull request, then the project is built using the build system, Maven in this case. TravisCI then runs all tests suites, unit and integration tests using the test scripts which can be customized.

When the process finishes a notification is sent to Github telling if the build of the Pull Request succeeded or failed. In both cases the full log of the build is supplied to the team for analysis purposes. The following images are taken from Github and TravisCI respectively and correspond to a failing Pull request.

![Image ofPull request failing](image1.png)

Figure 5: Pull request failing

![Image ofDetailed log](image2.png)

Figure 6: Detailed log of the failing Pull request in TravisCI

Once the errors causing the build to fail are solved by committing a fix, the whole process is triggered again. Hopefully the build will pass and the Pull Request will be ready to merge. The following images correspond to a successful build.

![Image ofPull request succeeded](image3.png)

Figure 7: Pull request succeeded
Figure 8: Detailed log of the succeeded Pull request in TravisCI
3 Competitive analysis

Before starting to develop and during the development, some competitive analysis has been carried out with the objective of obtaining new requirements and studying the technologies used other similar web applications.

The competitive analysis had mainly two objectives. First, to discover which technologies were used in similar applications, specially in the database layer. Second, to discover useful features to include in the system.

The competitive analysis provided advantages when getting requirements, but could not provide neither useful information about the technologies nor provide good usability examples when displaying information or requesting an input.

Competitive analysis has been an important part of the methodology. During the sprint meetings, the client requirements were sometimes extracted from other web pages and, when showing some features, usually they were compared with other web pages. Without using competitive analysis, is sure that the requirement engineering would have been worse.

3.1 Used technologies

When analyzing the technologies used by other web applications of the field, the focus was set on the server technologies, specially the database. To conduct the analysis, the team studied four papers of four different web applications, which are: FlyBar [8], CarroyDB [9], ChiloDB [10] and EimeriaTranscriptDB [11].

All of these applications used the same technologies. HTML, CSS and Javascript for the client side. PHP inside an Apache web server with utility PERL scripts for the server side. Finally, a MySQL as database. Although these technologies have been out there and proved useful for many years, none of the papers provided any particular reason to use this stack of technologies. It seems like these technologies are used because everybody uses them and not because these technologies are more suitable or more efficient with the data at hand. For this reason, the team has not been able to extract any relevant information of which technologies to use and decided to select a new stack with more suitable and modern technologies.
3.2 Web design analysis

When studying the design of biological web pages, there are two possibilities of searching methods, the first and most used is a horizontal navbar on top of the page, as it can be seen in the image below [9], and the second and sometimes used is a sidenav. The two options permits the user to navigate through the different sections of the web and in the project, and the horizontal navbar is the one used in the project.

![Chickpea Transcriptome Database](image9)

Figure 9: Chickpea Transcriptome Database [1]

Regarding the styles of the webpage, almost all the biomedical webpages are old and they have almost no style or have styles that are not used anymore. The project corrects some style errors of those web pages introducing bootstrap.

3.3 Display format

One of the main objectives of our project is to have an interface that displays the database information by doing simple searches. Almost all the examples of web pages studied have a way of showing the information. Usually, when web pages request a search, they give a little description of how to query, like the example of the image below [10].

![Gene Search](image10)

Figure 10: Brain Transcriptome Database [2]

This has been a good example when developing the search features of
the project. In those features, a query example is provided with a description.

When showing the search results, all pages follow the pattern of giving a list of items with the sequence identifier and more attributes. The Chickpea Transcriptome Database, the image below [11] shows the result in a table of items, giving the sequence identifiers, a description, and download options for different attributes.

The problem of this type of showing is when the result set is big, the client goes slower and sometimes it breaks. The solution is to use pagination. Furthermore, the buttons to show different attributes of the sequence are difficult to understand and the description of what are everyone of them is shown at the final of the query set, so if the query set is big, is difficult to see the provided description.

![Figure 11: Chickpea Transcriptome Database Listing](image)

The Brain Transcriptome Database solves the problem of showing a big list of result set, providing pagination, as shown in the image below [12]. During the analysis of this component, the team has noticed a problem with the attributes and options being shown.

When listing the sequences, this web page shows eight attributes or options about every item, but, the client and the team realized that in the project, would be better to show only the important information about the sequence and make a details page that would show all the specific attributes and options for one sequence. In this way, the listing would be much more simpler.
3.4 Blast tools

When trying to introduce BLAST (*Basic Local Alignment Search Tool*) into the project, how to do the input, output, and result display were analysed from other web pages. Usually, all the inputs and outputs were more or less the same in all web pages, but the interesting part was the result display. Some pages posted the result display as text, and this text was from the BLAST tool. Then, the *SalmonDB* [12] was analysed, and the result display had a particular way of showing.

When showing the result of a blast search, normally the sequences that match appear as text, but in the *SalmonDB*, the result appears as a diagram, too, like in the following image 13.

This same result appeared later in other web page, so the team started searching for a component and finally found the *Sequence Server*. With the *Sequence Server*, a customization of the BLAST tool can be used, which displays the search results with graphics and uses an intelligent input interface (selects the type of search possible). The *Sequence Server* will be further explained in Sprint 7 and 8.
4 State of the art

At the beginning of the project, before starting the sprints, a state of the art study was carried out. The purpose of this study was to obtain the necessary knowledge about the data that the application would work with and the technologies that form the core of the project, since these two aspects are the keystone of the project.

To conduct the study about the different databases it was crucial to carefully analyze the data that was going to be stored in, the DNA sequences. Various documents containing real world data, data that would be used by the deployed production system, were supplied by the client. These documents have been the subject of the analysis.

In the following section, the study of the data from real world files will be explained. After that, in chapter 5, the different database types will be analyzed in order to find the database that better fits the studied data.

4.1 Study of the information structure and meaning

The core data of the project are DNA sequences, in this case, of the \textit{salmo-trutta} species. There are also protein sequences. However those sequences won’t be included in the resulting first version of the product to not broaden the scope excessively.

Each DNA sequences can be complemented with extra information that give the scientists of the field a better understanding of the sequence. This additional data allows, for example, to group sequences according to certain properties that those have.

There are a lot of different types of this additional data, like Interpro families, KEGG (Kyoto Encyclopedia of Genes and Genomes) terms or Gene Ontology (GO) terms. However the project will only focus in the Interpro families. The reasons for that are: First, the client considers Interpro families to be the most relevant type of extra information and second, not to broaden the project’s scope even more.

\footnote{Interpro: \url{https://www.ebi.ac.uk/interpro/}} \footnote{KEGG: \url{http://www.genome.jp/kegg/}} \footnote{GO: \url{https://www.uniprot.org/help/gene_ontology}}
4.1.1 DNA sequences

DNA sequences are the system’s basic unit of work. They are the core of any operation performed by the system and the center of all the information. Any other type of information is always associated with at least one sequence, therefore, it is not possible for any information that does not represent a sequence to exist on its own inside the system without any relation to at least one.

The DNA sequences contain the precise order of nucleotides in a strand of DNA. The nucleotides are represented with a long string of letters, which are a combination of A, G, C and T, with the letters representing adenine, guanine, cytosine and thymine respectively. Each sequence has an ID assigned along with other secondary characteristics.

To better understand what a DNA sequence is and means it is necessary to look at where they are coming from.

When the research groups want to conduct a new experiment, the first step is to extract samples from a test subject. Then, those samples are sent to the laboratory where they are treated and input into the Trinity sequencing [9] software. After the sequencing process, Trinity outputs a Fasta formatted file.

The Fasta format is a text-based format to represent sequences that has become a standard used widely in bioinformatics. Every person and software of the field is capable of understanding it. The fact that Fasta format is standard means that interoperability between different software applications and research groups is far easier.

The fasta file outputted by Trinity contains all the nucleotide sequences of the experiment’s samples. Each of this samples has an ID assigned. Up next, the different parts of sequence will be analyzed along with an example.

To analyze the different parts of a sequence the following example will be used. Below is a sequence in fasta formatted file extracted from a file output by Trinity.

```
>comp100_c0_seq1 len=201 path=[179:0-143 323:144-200]
ACACAAACACATAAGCTGTGTTCATTCACTACTAAATACTGCTCTCTGGATCACAGGGTA
AAGCTCAATGCAGCCACCAGCAGAGAGAGAGAGGAGAGAAAAGGGAGAGAAAAGAGAGAGC
GAGAGGATAAATAGCGAAACCCGAGGAGGAGAAGGGAAATGATTGGGAAATAAAGAGG
```

Each sequence in every fasta file is structured in the exact same way and contains three main blocks of information: the sequence’s identifier, the sequence itself and secondary characteristics.

The first block of information is the identifier. The character ‘>’ at the beginning of the line denotes the starting point of a sequence. Next to it is the sequence’s ID, that will be referred to as *TrinityId* for the rest of the document. In this case, the TrinityId is *comp100_c0_seq1* and it uniquely identifies the sequence inside and only inside the fasta file. This means that the same TrinityId can appear in multiple fasta files representing completely different sequences even from different species. Although the TrinityId is not a unique identifier, it contains information in its semantics and needs to be stored.

The second main block of information is the string of nucleotides. This string consists of an undefined number of lines containing the ordered sequence of nucleotides. The string of nucleotides ends at the beginning of the next sequence or at the end of file if it is the last sequence in the file. The string always begins at the second line, counting the line with the TrinityId as the first line. This block of information is critical since it is used for sequence comparing softwares like BLAST\(^{10}\).

Finally, the third block of information is a set of additional attributes that can vary from sequence to sequence. This set of attributes always begins right after the TrinityId and spans across the first line of the sequence. Each attribute consists of a name and a value and attributes are separated with a space character. For the particular example, the set of attributes would be formed of two attributes. The first one being ‘*len*’ with the value ‘201’ and the second one being ‘*path*’ with value ‘[179:0-143 323:144-200]’.

The different attributes provide additional information and have to be stored, but neither further processing nor a full understanding of their meaning is needed.

\(^{10}\)BLAST: [https://blast.ncbi.nlm.nih.gov/Blast.cgi](https://blast.ncbi.nlm.nih.gov/Blast.cgi)
4.1.2 Interpro families

Interpro families is the only type of extra information that the project will focus on. Interpro families complement sequences.

Each Interpro family represents a property that all sequences belonging to the particular family share. Moreover, each family has a unique ID, which will be referred to as InterproId for the rest of the document, capable of uniquely identifying an interpro family everywhere. Unlike the TrinityId, the InterproId is globally unique and it is impossible for two different families to have the same InterproId. One sequence can belong to zero or many families.

In this project, the assignments of an InterproId to the different sequences comes in the form of a file provided by the software TRAPID\(^{11}\) (Rapid Analysis of Transcriptome Data).

TRAPID receives as input the raw sequences obtained from Trinity and assigns to each one of them zero or more families depending on the known properties of each sequence. The result of this assignment is a file with 4 columns. Below is an example of one assignment.

```
#counter transcript_id interpro description
1 comp153003_c0_seq1 PF03165 MH1 domain
2 comp153003_c0_seq1 PF00859 CTF/NF-I family [..]
3 comp153003_c0_seq1 PF10524 Nuclear factor I [..]
```

The first line of the file is the header line and contains the names of the columns. The first column is the number of the line of the file and can be completely ignored. The second columns is the identifier of the sequence, TRAPID calls it transcript_id but it corresponds to the TrinityId of a sequence. The third columns represents the ID of the Interpro family, the InterproId. Finally, the fourth columns contains the description assigned to the family identified by the InterproId.

In this particular example, a fasta file with a single sequence with TrinityId ‘comp153003_c0_seq1’ has been input into TRAPID. After processing the file, TRAPID assigned three different families to the sequence with the InterproIds ‘PF03165’, ‘PF00859’ and ‘PF10524’ respectively and with their corresponding descriptions.

---

4.1.3 Other types of information

Although the project only focuses on DNA sequences and Intepro Families, there are many other types of information and the system must be extensible so as to treat them in the future. This means that the system has to be able to support the new information in the future without having to be rebuilt. The need for extensibility will play a big role in deciding which database to use.

These other types of information are protein sequences as another type of sequence, which come inside a fasta file with some variations, KEGG and GO terms. These types of information won’t be further discussed in the document since it goes out of the scope.
5 Used Tecnologies

The complexity of this projects relays on the actual data and how to store and process it efficiently, which means that there is a lot of work to do in this matter. Hence, when selecting the stack of technologies to be used in the project, the data layer formed by the database received all of the team’s focus and attention. Whereas the technologies of the other layers of the stack, server, client, version control and continuous integration, where selected based mainly on the amount of past experience working with those technologies.

Therefore, the team has been able to dedicate the great majority of the time on what really matters: studying the data, methods to process it and develop meaningful features that satisfy the client’s requirements.

An exception to this policy of selection is Docker, which was considered, and has proved to be, worth learning, as will be explained later in this section.

5.1 Introducing the technologies

During the project, different technologies have been used in the back-end and front-end. To get a general idea about all the development technologies, a summary will be shown up next.

Back-end technologies:

- Java: The programing language in the back-end.
- Spring: The framework used to develop the Server application.
- MongoDB: The database selected to store the information.
- Junit: The library used to create unit tests.
- Cucumber: The tool used to create and automate integration tests.

Front-end technologies:

- Typescript, css, html: The languages used in the front-end.
- Angular: The framework used to develop the Client application.
- Bootstrap: The library used to get styles for the Interface visualization.
5.2 Study of the different database types

The project works with biological data which has information that depends on the different input files introduced. In this files, some data could be unknown, so the name of the attribute is known when the column containing the name is read. This implies that the data is formed of static and dynamic information.

Also, the type of queries that will be done are specific and almost all of them are selects, implying preference on having better performance on select rather than write. One example of the specific queries is selecting the sequences that belong to the same interpro family.

Moreover, searches imply different interactions like wanting to get a big set of data, followed by a click on a part of the set that shows a more detailed view of that part, or download a part of the set as a biologic file.

When selecting the database technology to be used, a study of the advantages and disadvantages of every database has been made [14] [15] [16] [17].

5.2.1 Relational Databases

Relational Databases have been around for more than 30 years and are by far the most used databases. This is because of the big popularity, the language known by every developer and because having a Relational Database is always a solid option.

The relational model organizes information into tables of columns and rows, with a primary key identifying each row (or also called tuple). Each table represents an entity and every column represents the attributes of the entity.

When working with clearly defined data like banking systems, relational databases are always the best option, and this is because the database architecture will always be pretty much the same. The nature of data suits perfectly with tables and the relation between entities can be searched faster using joins.

The problem comes when very complex data like molecular modeling is part of the application. This type of data has multiple levels of nested data, or data that needs to be classified differently depending on the situation. In the past this data has been modeled into relational databases, showing a bad performance.
5.2.2 Graph oriented Databases

Usually, when having relationships between data, the relational model is the most used, but, when many-to-many relationships are very common in the data, a relational database can not handle it. This happens when the information contains data about people relationships, road networks, or any sort of network and the natural way to structure this data is to start modeling the data as a graph.

A graph has two basic objects: nodes (also known as vertices) and relationships (also known as edges). For example, the relationship between people can be structured using nodes as people and edges to indicate relations between people, like in the following diagram.

![Social Graph](Image)

Graph databases are generally indexed by relationships, and this means that traversing relations can be done very quickly. For example, in a genealogical tree, going through the bloodline of a family and search for males in the family members has almost no cost. The power of graph databases
is to have a quick performance when searching in a long path or a variable length path.

Now, the downside of graph databases. If a query like selecting all the products of a market or select products with concrete price needs to be implemented, graph databases are the worst in performance. Dumping information in a graph database or searching for specific attributes in all the graph, forces the database to scan all nodes, and this is not the potential of a graph database, but a relational model database.

5.2.3 Document Oriented Databases

Document oriented databases are the contrary to graph oriented databases, they have potential to represent data with no relationship or one-to-many relationship.

Unlike traditional relational databases, the data model in a document database is not structured in a table format of rows and columns. This sort of database saves data in a document, with a structure of multiple key-values inside. Document databases are a variation of key-value databases, and their schema can change naturally, providing more flexibility than relational databases.

```json
{title: 'Babylon 5',
 seasons: [
 {season_number: '1',
   episodes: [
    {ordinal_within_season: '1',
      title: 'Midnight on the Firing Line'
    reviews: [{...}],
    cast_members: [{...}]
   ]
 ]
]
}
```

Figure 15: Example Document Database Structure

One of the crucial points for the project is that document databases can dump information really fast, as well as perform efficient searches. The other one is that there is no need to modify the entire database schema as there is with a relational database. Data can simply be added by adding objects to
the database.

Document databases have downsides [18]: When two documents have linked data between documents, the design needs to be a little more specific. For example, one option could be duplicate data, duplicating part of the document one in the document two and vice versa. This will generate a waste in disk and the searches will be a little bit more expensive. Another option could be using indexes (some document databases have them implemented), but this sort of architecture could lead into a relational model expressed in a document database, which is not wanted. This leads to the conclusion that document database can only be used when the nature of data can be expressed as a document, to gain the best performance, flexibility and scalability.

5.2.4 Column Oriented Databases

Column Oriented Databases are a very use-case dependent databases. Normally a Relational Database would do the job, but when specific requirements are needed, column oriented is perfect.

This kind of database stores data in columns instead of rows (like relations), which permits the database to be more precise when trying to query because the searches do not need to discard the unwanted attributes in a tuple like in the relational model, the search is done directly on the specific attribute. In a column, every value has a key or identifier, which is also used to know the related data in other columns.

![Figure 16: Relational Model vs Column Oriented](image)

Relational databases and column databases have similar structure, but different objectives. One of the main upsides of Column Oriented Databases is the write speed offered (for example, Cassandra\footnote{Cassandra: \url{http://cassandra.apache.org/}} is one of the best databases offering write speed, using in memory writes and logs). Usually, Column databases can work with the biggest data sets of information without points of failures (instagram uses Cassandra storing more than 80 million photos and spotify uses the same database storing more than 20 million photos).
songs).

Column databases have downsides too. If an application has almost no writes, but a lot of reads, column databases are not suited, since they are not faster when the information needs to be read. When information follows nature that is not suitable in a column oriented database, or if the requirements are not known, usually is better to use a Relational Oriented Database.

### 5.2.5 Key-Value Databases

Key-Value Databases are the most simple databases, and they are mostly used as a support for other databases (one example could be facebook architecture, using a memcached\(^{13}\) supporting a mysql\(^{14}\) database).

This sort of databases have a simple structure, even simpler than a document oriented. There are values referenced by keys and this keys are the part of the database that can be accessed. Usually, Key-Value databases are in memory, improving their speed but being volatile since the information is not stored in a disk (some databases do adjustments and store some information in disk).

Key-Value databases are most frequently used for managing the session information in web applications, phone applications and multi-player on-line games. Also, social networks like twitter, use (or used) key-value databases to store tweets for every user, so the loading would be faster.

When trying to search value attributes or do expensive queries, Key-Value databases are not good and this is why their use is for storing simple data that needs to be read fast.

### 5.2.6 Polyglot persistence

Companies around the world are saving more and more data everyday. If an application is big, probably the database architecture will be big too and a lot of queries and requirements will be implemented. When this happens, the possibility to use more than one database appears.

Polyglot persistence means using more than one type of database, taking the best part of each one according to the application requirements. All the big applications in the internet use more than one database, taking a relational database as the base with the addition of NoSQL databases that could adapt some of the most specific requirements. For example, facebook once used mySQL with a memcached, controlling the most volatile data.

\(^{13}\)Memcached: [memcached.org](http://www.memcached.org)

\(^{14}\)Mysql: [www.mysql.com](http://www.mysql.com)
in the memcached and storing the important one in the mySQL. Apache Cassandra, a column oriented database, is used for the inbox searches, along with more software helping their architecture.

Polyglot persistence is by far the best way to work with high volumes of data, but when the volume of data is not big, the time consumed in creating and maintaining more than one database is not worth it.

5.2.7 The decision

After studying each database type and looking for advice from professors that know about databases, the team decided that a document database was the optimal database for the project. In the document database world, there are a lot of organizations that develop a database, for example CouchDB or MongoDB. Because of the previous knowledge and work using MongoDB, the team decided that it was the optimal database for the project. Up next is an explanation of why the other databases were discarded and MongoDB was selected.

Even though a database has been selected, the project information does not have a huge size and almost any database could fit the requirements. The database selected fits the needs and scalability of the project.

There are some kind of databases that are easy to discard since the purpose offered is different from the project objectives. This databases are Graph oriented, Column oriented, and Key-Value oriented.

Graph oriented Databases have the purpose of represent data when a lot of different relations appear between data, or there is a huge quantity of linked data \[19\]. In the project, the data has relations between sequences defined by the Blast tool, and relations between sequences and different identifiers as Interpro, Kegg or Go [4.1.3].

Obviously a graph database could be implemented, but, the problem appears with the search options. The web application will have different search options, and the result of those searches will be a dump of information. Is true that searches between all the linked sequences could be done really fast using a graph database, but when having dumps of information, the database would suffer.

Key-Value Databases are way too simple to be used as the only database in the system. Maybe having a unique way of data, this sort of database could be applied as the only one, but usually this database appears using

polyglot persistence. In the project, this sort of database would have a lot of problems when trying to construct the dynamic information or some of the specific queries.

Column oriented databases [20] have their advantage of a fast write and searching by attributes value really fast. In the project, the writes into the database are done when the administrator introduces a fasta file containing sequences or a txt file containing information about Interpro, Kegg, Go or dynamic information. The quantity of writes will be low and with no need of being fast, meaning the property of fast writes given by most of Column Databases, does not really matter.

When readying the database, the returned information will always have an object with all the attributes, meaning that in a Column Database almost all columns should be read every time a search is done [21]. This, and because of the low load of writes, is why the project does not use a Column Oriented Database.

Relational Databases are a good option for our project [22]. Suits with the reads, writes, and almost all the nature of the information. The only problem is when dealing with dynamic information.

If dynamic information is needed to be represented into a relational database, the model would be something like:

<table>
<thead>
<tr>
<th>type</th>
<th>value</th>
<th>related_sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>blast_coefficient</td>
<td>0.86</td>
<td>[sequence]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Figure 17: Relational Model Example

The problem appears when the type has more than one value (with different structure), and when a value has a relation with more than one sequence. A lot of designs could be done to repair this necessity, but the flexibility the relational databases offer is a little bit low.

Document databases have the properties of dumping data really fast and managing the data with relations one-to-many really easy. This is perfect for the web application where the main activity of the database will be showing different sequences or different interpro families, and because the nature of the project data is having sequences that are related with different Interpro, Kegg, and Go.
Document Databases have a very important property for our project. In a document, the design for dynamic information can be done easily, for example, every sequence could have an attribute for his dynamic information, changing the structure in each sequence. In the future, dynamic data will be one of the main parts of the project, and this is why *MongoDB* has been the database for our project.

The option of doing polyglot persistence has been considered. The problem is that the project does not have a lot of different kinds of information to be stored in different databases. In the future, if more kind of data is introduced, the addition of another database could be done and then introducing polyglot persistence would be the best idea.

---

MongoDB: [www.mongodb.com](http://www.mongodb.com)
5.3 Server side technologies

The server side or Back-end technologies were chosen based mainly on the previous experience the team had with them, but also on the quality of the available documentation, the size of the developer community and if a professor of the university could be able to help, which has been the case.

Basing on the mentioned criteria, the following technologies were selected to form the Back-end of the project: Java, Spring with Spring Boot and Cucumber for testing. Below is an overview of the different technologies just to provide the reader with some context, detailing these technologies is not in the scope of this document.

5.3.1 Java

Java is an object oriented language that runs on the Java Virtual Machine (JVM) making it able to run on any architecture since the JVM provides an abstraction from the underlying Operative System (OS).

Java has a very good quality documentation and an extremely broad community of several million developers worldwide. According to the Tiobe Index\footnote{Tiobe Index: \url{https://www.tiobe.com/tiobe-index/}} Java has been on the top 2 positions basically since it was created. At the time of writing this document Java occupies the top 1 position.

5.3.2 Spring with Spring Boot

Spring is an open source application framework and inversion of control container for the Java platform. It can be highly customized to fit the developer’s needs with various modules that cover different tasks and aspects. Spring is constantly growing and has become very popular among Java web developers.

Spring Boot \cite{24} \cite{25}, developed by the Spring team provides an easy way to create production ready application with minimal configuration but highly customizable while reducing the need for the developer to maintain non conflicting dependencies.

5.3.3 Testing with Cucumber

Cucumber is a tool for running automated integration tests. The tests are written with a story-like format in human readable language that both the developer and the customer can understand. Each ”story” or scenario represents a feature that can involve multiple modules of the application.
5.4 Client side technologies

The technologies for the client side of Front-end were selected with the same criteria as the Back-end technologies mentioned previously.

The stack of Front-end technologies selected are: Angular 5 and Bootstrap. Below is an overview of the technologies.

5.4.1 Angular 5

Angular 5 [26] is a framework for building Single Page Applications (SPA) in Typescript. Angular provides a lot of code generation for the most common tasks, is well integrated with various IDEs (Integrated Development Environment), has a decent documentation and easy to understand tutorials as well as a growing community.

5.4.2 Bootstrap

Since the interface and its interaction are an important part of the project, Bootstrap was selected. Bootstrap is the leading front-end framework capable of quickly building responsive applications that can be viewed with very good quality either in large monitors and small smart phone screens.
5.5 Version Control System (VCS)

From the beginning of the project Git was the only option for the version control system of the project and to be able to work as a team, the platform Github was selected.

5.5.1 Git

Git has been selected as Version Control System (VCS) not only because the decent amount of past experience the team has with it, but also because it has become a de facto standard for distributed version control that has proved to be working in project of all sizes with great speed and efficiency. Git outclasses other VCS like Subversion or Mercurial with features like having multiple workflows known as branches and nearly trivial merge processes along with many other utilities.

Moreover, Git is one of the crown jewels of Software Engineering.

5.5.2 Github

Github is a web-based platform that uses Git. It offers the developers a distributed version control system in which projects can be store and contributed to. Github also provides a very powerful system to manage issues, to perform bug tracking, to do code reviews via Pull Requests and more.

Github has almost become a social network for developers.
5.6 Continuous Integration

To implement continuous integration (CI), a CI server was needed. There were many possibilities but two of them had the most potential: TravisCI and CircleCI. The amount of experience with both tools was almost the same, but TravisCI was preferred over CircleCI because that was the system that, when the project started, was used in class at the University, therefore existed the possibility to ask for help if needed.

5.6.1 TravisCI

TravisCI is a CI server that allows to build, test and deploy public projects hosted in Github for free. To work with private projects, a TravisCI license is required, but that’s not the case of this project.

TravisCI provides many features that have been very useful for this project to result in a quality product. The most important feature is: supports many languages, Java included. Each new build runs on a new Virtual Machine (VM), the VM can be greatly customized, provides a notification service with Emails or Slack and has a great documentation and tutorials.

Also, two plugins were used to evaluate the quality of the developed code and the amount of code covered of the tests. The plugins are CODEBEAT\(^{18}\) and Codecov\(^{19}\) respectively.

\(^{18}\)CODEBEAT: https://codebeat.co
\(^{19}\)Codecov: https://codecov.io
Finally, the following diagram illustrates the work flow of the management tools (Git and Zenhub) together with TravisCI.

Figure 18: Continuous Integration Workflow
5.7 Packaging and deployment

When thinking on how to deploy the project once it was ready, the first initial option was to deploy the application as a common JAR file into a server, which Spring Boot generates. But this option had main downside. The production server was not known in advance, in fact it wasn’t known until the very end of the development process. Besides the project was formed by various modules which can be independent containers, the JAR representing the server, the Angular client and the Mongo database.

Docker was a good option to minimize the impact of not knowing the characteristics of the production server and exploiting the container-type nature of the project.

5.7.1 Docker and Docker Compose

Docker\footnote{Docker: https://www.docker.com} is a container platform that has become a de facto standard for developers and system administrators for packaging, deploying and running distributed cloud applications. It allows developers to create templates which can later be used to create lightweight virtual machines called containers. This containers host the application code. Docker makes it easier to automate infrastructure, isolate applications and improve resource utilizations. The Docker containers are the perfect options to host micro services since they are easy and cheap to deploy.

To deploy the multi-container application that is this project, Docker Compose\footnote{Docker Compose: https://docs.docker.com/compose/} has been used. Docker Compose is a tool for deploying and running multi-container applications by defining the dependencies between containers and the configuration options in a file. Then, all the containers that form the application can be deployed automatically with only one command.

\footnotesize
\bibliography{references}
6 Sprints

This section covers the whole development process of the project, Sprint by Sprint. In each Sprint there will be explained the tasks completed, how these tasks were extracted from the meetings, which problems appeared and how were they solved. However, prior to the Sprints, there is the context diagram, a diagram that defines the scope of the project and is used to provide context to the team.

6.1 Context Diagram

The scope defined during the project is represented in the following context diagram.

![Figure 19: Context Diagram](image)

In this diagram, the bubble in the center represents the project system, and the rectangles around represent the systems or people interacting with the project system.

When a user interacts with the system front-end, the Biology Researcher rectangle represents him. The client of the project or his research partners will
have the privilege of being administrators, who are represented by the \textit{sttDB Admin} rectangle. They can upload files to the system, and this files must follow some \textit{File standards}, so the system can understand them. The project contains a REST API, which can be accessed by other web applications or computers. They are represented by the \textit{Developers using the API} rectangle. Finally, the system interacts with an external tool called SequenceServer\textsuperscript{22}. This tool is a Blast application, and is represented using the \textit{SequenceServer BLAST tool} rectangle in the diagram.

The diagram contains a total of thirteen data fluxes:

1. Fasta files: The administrator can upload Fasta files to the server, where the information is parsed and stored in MongoDB.

2. Interpro files: Like Fasta files, Interpro files can be uploaded into the server by the administrator, storing their data.

3. Download raw files: The users of the web application have the option to download raw files that the server contains (like Fasta files).

4. Search information: Any user has the option to search sequences or families.

5. Search result: The results are shown listed in the user navigator.

6. Sequences to BLAST: The user sends sequences to the system, so they can be compared with the stored ones.

7. BLAST result: Show the BLAST comparison results to the client.

8. Sequences to compare: The system redirects the user sequences to the BLAST tool, where they are compared.

9. BLAST result (SequenceServer blast tool to sttDB system): The comparison results are returned to the system.

10. Sequences JSON syntax: The REST API offers different GET methods to query the stored Sequences.

11. Experiments JSON syntax: The REST API offers different GET methods to query the stored experiments.

12. Families JSON syntax: The REST API offers different GET methods to query the stored Interpro families.

13. Files format specification: Fasta or Interpro files have an specific structure that the system gets into account when parsing them.

\textsuperscript{22}SequenceServer: \url{https://www.sequenceserver.com/}
6.2 Sprint 1

The first sprint acted as the setup for the project and laid down the foundations of the project. As explained in the section [Methodology] each sprint has a sprint meeting. During the meeting of this first sprint the work focused on defining the scope of the project, setting up the tools and define the team roles, to review different similar web applications (competitive analysis) and to understand the different data sources provided by the client. Outside of the meeting the team investigated the different database types and created a first prototype of the website to show in the next meeting.

6.2.1 Scope of the Project

There are a lot of features and ground to cover in the field, however, since there is a limited development time of four months, the scope of the project had to be clearly defined in order to create a good quality product that satisfies the client’s basic needs and which development was feasible.

The result of the discussion was that the product would cover the storage, visualization, and comparison of DNA sequences. However, plain sequences aren’t much useful on their own and the client wanted to complement the sequences with some extra information, the Interpro Families, which, for the client, is the most relevant type of complementary information.

6.2.2 Roles, Tools and Technologies

During the meeting, the team roles were decided. The roles are explained in more detail in the section [Methodology 2]. As a quick reminder, Roger and Marc have the role of development team, Joan has the role of client and Marta has the role of Product Owner and occasionally Scrum Master.

Regarding the communication and project management tools. The team decided to use Github together with Zenhub for project management and Slack as a channel of communication between the four members. The stack of technologies, except for the database, was also selected in the first sprint. The stack of used technologies is explained in the section [Used Technologies 5].

6.2.3 Understanding of the information and competitive analysis

Both the analysis and understanding of the information and the competitive analysis were carried out inside the meeting, by the whole team, and outside, by the developers. Both parts are explained in more detail in the sections [State of the Art 4] and [Competitive Analysis 3]. The developers also started
investigating the different database types, explained in the section Study of the different database types 5.1

6.2.4 First Prototype

When starting the first prototype, the idea of creating one from scratch was considered, but rapidly discarded since it involved too much work. Instead, a template was used. There are mainly two types of interfaces for web applications. The ones that involve a lot of scrolling and the ones that do not. The first type is great for presentation, the information appears sequentially and ordered. However, this model becomes less useful when quick access to various functions is needed due to the heavy scrolling.

Since the website of the project is intended to be used rather frequently by other researchers, a quick access to the important functions was needed, therefore, the scrolling model was discarded.

To create the first prototype, the team used the website Tooplate[23], which is a website with free templates for websites created with HTML and CSS. Among the various different types of templates, the team selected one that was more suitable. The selected template is: Sunflower[24]. Below is an image of the main screen.

As the reader can see in the image, the template provides quick access through the top menu bar to the key features of the website, which are the search tool and Blast tool, second and third options on the menu bar.

The first thing that the user will see is the description of the project, "About the project" in the image, alongside an image. However, the image in the prototype does not represent the project, it came with the prototype and the team did not change it because the purpose of the prototype was to outline the structure of the website.

The structure of the prototype is simple, yet useful to take a first grasp on how the interface should look like and where the various functions should appear. The team decided to keep it simple so as the requirements were easier to extract in the following meetings.
6.3 Sprint 2

The second sprint started with a meeting showing the first prototype. With this prototype, the team could receive a better feedback from the client, analyzing which requirements the client prefer and how the web page should be structured.

At the first meeting, the team defined which were the basic requirements the client wanted, so the definition of the server architecture could be started and some parts of the front-end could be developed.

At the first meeting, the team noted that what the client wanted the most were the queries showing information from the uploaded files. Knowing that, the team defined as objectives to upload files and to show the information of those, starting with Fasta files, which contain primary sequence information explained in the state of the art, section 4.1).

Because of the prototype example and the competitive analysis at the first sprint, the team understood that the client wanted a navbar with dropdowns for the web application. It is worth mentioning that the client always listened the team advices and almost all the front-end visualization was decided between the client and the team.

6.3.1 The Database options

During this second sprint, the team continued investigating over three options: To use a relational database, a document oriented database, or polyglot persistence.

After speaking with some professors and studying a little bit more the structure of the information, the team discarded polyglot persistence because of the lack of data, as explained in the decided database section.

The decision of which database should be used was not done in this sprint, but the developer team started considering MongoDB the best option. Bringing the suggestion to the next Sprint meeting, the database was decided.

6.3.2 The second prototype

Even thought the team had a good prototype from the first meeting and the client accepted the style of the first web page, another prototype was searched, trying to improve some visualization aspects the first prototype lacked (like the odd navbar or the size the section About the Project occupied).

This prototype, like the first one, was download from Tooplate, and has more
or less the same HTML structure, but different styles. The structure of the first prototype was the same as other web pages, and the client wanted to maintain it, this is why the team only searched for a better style.

In the search for this second prototype, the team aimed to obtain a prototype with simpler style and simpler variety of colors than the first one.

Finally, the team encountered a prototype with a simple style, which contained only three colors (blue, white, black) and had a simple style.

![Prototype 2 Overview](image)

Figure 21: Prototype 2 Overview

Thanks to this style, the team could improve the way of remarking the important things in the future, by adding a highlight visualization.

While working on the second prototype, the first prototype got some improvements in the requirements the client specified, like how the sections should be named or how many sections the page should have. All this improvements were added into the second prototype too, so at the Sprint end meeting the team and the client would decide which was the best one.

At the end of Sprint meeting, was decided to use the second and new prototype because of the simplicity offered and its modern visualization. However, the client suggested some changes in the header and footer: The first was to introduce another image that did not overlap the header, being a smaller image surrounded by a white background, and the second suggestion was to put the image in the center of the header, between the sttDB title and a logo on the right side (the *Endesa* sponsor logo). Also, the team decided...
to introduce the other logos in the footer of the web page.
6.4 Splash 3

At the Sprint meeting, was decided that the second prototype would be the basic design for the web front-end. However, some requirements were extracted during this second meeting regarding the information shown, which were implemented during this sprint. The development of the back-end and the design of the database started, too.

6.4.1 New changes into the prototype

Even though the team would change the prototype in the future, it was decided to structure the prototype first, so the team could know all the sections the page should have.

First, as explained at the end of the Sprint 2 section, the header was changed and the sponsor logos were introduced. Also, the image about the salmo-trutta was changed, adding a correct one, shown in the image below.

Second, the client requested the download section to have the option to download all the files introduced by the administrator. Mainly, the client wanted the option of downloading Fasta files, because they are the root of the information and the one that is impossible to get without the client’s biological information.

Finally, since the web page must be accessible to all the world, it was decided to introduce English as the main language, with the possibility to introduce Spanish and more languages in the future. Also, the team decided to introduce documentation explaining the features of the web page, ensuring that anybody could understand them. Furthermore, this documentation should contain information about the REST API, explaining the available methods and the data returned so other developers can use it.
6.4.2 Decided database

In the Sprint meeting, the option of using MongoDB instead of relational model was proposed and accepted by the team. The reason was that the dynamic information, a pillar of the project, would be difficult to save, query, and maintain, in a relational database. The other reasons of why MongoDB is used are explained in the decided database section.

Some tests had been done with a MongoDB architecture in a testing back-end, created with Spring-boot. In this back-end different architectures were tried, understanding how Spring facilitated the use of MongoDB.

The Spring Framework provides an abstraction layer to interact with the database called Repository interface. The team understood that because
of the Repository interface, the query system is the same as other types of databases.

Even though tests were done in a back-end, the team also did tests over the database directly, understanding the MongoDB query language with CRUD operations\(^\text{25}\).

### 6.4.3 First database architecture design

In the last part of the sprint, the team developed the first database architecture diagram and started the back-end project based on this first database design. Even though this architecture is incomplete and even naive, it served as a good starting point for the upcoming development. All the flaws of the architecture and its evolution will be discussed throughout the next sprints, once real data starts to be stored in the system. The database design will vary and evolve throughout the whole development process.

![Figure 23: First database architecture diagram](image)

In this first design, the database consists of two types of data represented by the documents Sequence and Family. Each document has an identifier or Primary Key, represented by the \_id field, which is automatically assigned by MongoDB when a new document is created.

The Sequence document, apart from the id, has the fields \textit{trinityId} which is the ID that comes in the Fasta file, \textit{specie} that represents the specie that the sequence belongs to, \textit{transcript} which represents the sequence of DNA and finally, \textit{familyId} that indicates to which family the sequence belongs.

The Family document, apart from the id, has the fields \textit{interproId} that represents the ID of the family, and a \textit{description} field, containing the description of the family.

Notice that with design, a sequence can only belong to one family, whereas various sequences can belong to the same family. This is an error since

\(^{25}\)MongoDB manual: [https://docs.mongodb.com/manual/crud/](https://docs.mongodb.com/manual/crud/)
a sequence can belong to zero or more families. This error was due to, probably, a misunderstanding of the requirements by the developers. But, since the initial milestone was to store just the sequences, the error was hidden. However, this error will be corrected in the following sprints, when the features involving the uploading and storing of Interpro families start to be developed.

6.4.4 Starting the development of the server

Regarding the server, the project was started using the SPRING INITIALIZER tool, which is able to bootstrap a project ready to be executed with the dependencies that it should have in no time. The dependencies used can be found in the file pom.xml of the project sttdb-server. Keep in mind that during the development of the project, more dependencies will be added, for example, dependencies for testing or security.

The server in this sprint was very basic because the team didn’t implement any feature. As said earlier in this section, the focus was set on deciding which prototype was the best and improving it. The team only developed what was needed in order to initialize the project and start working in the following sprint, Sprint 4. Below is an overview of the server’s directory structure and Java classes at this point.

![Structure of the server - Sprint 3](image)

Figure 24: Structure of the server - Sprint 3

Sprint Initializr: [https://start.spring.io](https://start.spring.io)
As said, the structure is very simple. At this point the server simply consists of basically three items. The domain classes that represent the documents in the database, the Repositories that interact with MongoDB and perform the basic CRUD (Create, Read, Update, Delete) operations and the Application class, the entry point of the whole application.

The classes Family and Sequence represent the respective documents in the database and are very simple. They contain the exact same fields as the document in the database with getters and setters.

The repositories FamilyRepository and SequenceRepository are two Spring interfaces. Through them, the code is able to transparently interact with the database. The repositories already have the CRUD operations so no further implementation is needed. However, custom methods can be added to perform special queries. Below is the FamilyRepository interface.

```java
@RepositoryRestResource(collectionResourceRel = "families", path = "families")
public interface FamilyRepository extends MongoRepository<
    Family, Long>

    Family findByInterproId(@Param("interproId") String interproId);
}
```

Figure 25: Family repository - Sprint 3

The FamilyRepository is a domain specific interface which extends from MongoRepository, a Spring interface to interact with MongoDB, that operates with documents of type Family with an ID of type Long. Also, Spring automatically implements the REST API for this document under the URL entry point /families, as it can be seen in the annotation @RepositoryRestResource. Additionally, the interface has a custom method to search for a Family with a given InterproId. The SequenceRepository works similarly, but with sequences.

Notice that there is not an implementation of neither the interface nor the methods, Spring automatically provides an implementation for the interfaces during execution time.

Regarding the configuration, there is none. Rather said, there is no more configuration than the configuration Spring-Boot automatically provides. There is no database configuration or Bean\[27\] configurations, Spring handles everything. However, the domain specific configuration will be added later.

---

\[27\]Beans: [https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html)
6.5 Sprint 4

This sprint is separated into two lines of development. The first one consists of the implementation of the features to upload Fasta files and to query the sequences stored from the application’s presentation layer or front-end. These features will be implemented initially in the prototype because the angular client is not yet developed. The second line of development consists of the migration of the front-end prototype to Angular 5. The migration will not include the features of Fasta uploading and sequence querying.

6.5.1 Fasta upload

One of the first big features of the project was the Fasta file upload into the server. In summary, the client can send any Fasta file to the server, where is going to be treated and stored.

On the server, a controller is triggered when the POST method ”/uploadFasta” is performed. This controller uses a service implementing the StorageService interface, whose responsibility is to store any file into the system.

The StorageService contains two basic parts, the properties definition, and the file manager. The StorageProperties class has the responsibility to contain the management properties regarding the location of the file system where the files have to be stored.

The FileSystemStorageService is an implementation of the StorageService interface, and has the responsibility to manage (store, load, delete) any file. To create an object of this class, first, an object of StorageProperties is needed. Then, the class will work with the properties stored (like the location) to do the file management.

In the project, one of the main parts is the work done with files received from the administrator or the files that can be given to an user. This is why this service is the most used in the code.
In the fasta upload feature, this service is used by the class `FastaFileManager`. This class only responsibility is to communicate with the `StorageService`, allowing a previous treatment to the Fasta files.

In the feature, when the controller is triggered, the first thing done is to store the file using the `FastaFileManager`, and then the fasta parser is called.

Both classes, `FastaFileManager` and `FastaFileManager` will be replaced by the class `ExperimentStorageService` in the Sprint 7 due to a code refactoring.

The fasta parser reads the Fasta file stored, parses all the information in it, and stores the parsed data as `Sequence`. Thanks to the `>` character, that marks the start of sequence, the parser knows the start of a new one and knows the old sequence needs to be stored in the database.

When saving the sequence, in this sprint, the trinityID, the length of a transcript, and the transcript, were extracted from the Fasta file parameters and stored, but, other parameters were not saved. This parameters were not saved because the team considered them irrelevant, until later sprints, that requirements for those parameters appeared.

In the front-end, the feature was coded on the prototype, using Ajax to
connect with the back-end.

The team knew this feature was temporal because the front-end was being reimplemented in angular. This meant that the prototype had the only purpose of showing how the feature worked for the client while the new front-end was done.

The upload page was simple: It only contained a brief explanation of the wanted file and two buttons, one for selecting the file and the other to send it, as shown in the following image.

Figure 27: Upload file, first design

6.5.2 Search by trinity id

When the project could store sequence information in the database, the team implemented the feature to query this information.

First, because of the Repository Interface offered by Spring, the Back-end did not need modification, any request over the sequence information could be done using the GET method /sequences/search/findByTrinityId.

On the front-end, like in the upload files feature, the implementation was done because the team wanted to show the feature to the client. To request the information to the server, a simple HTTP method was done using Ajax. The obtained response was treated and shown in the same tab as the query

---

28Spring Repositories: docs.spring.io/spring-data/jpa/docs/current/reference/html/
The idea to create the Angular application was simple: To maintain the same visual structure, but implemented with Angular’s infrastructure written in TypeScript.

Angular applications are formed by components where each component represents a different part of the application. Each component has three different parts: the template that contains the HTML and CSS styles, the component class that controls the user’s interaction with the template and finally, the services that provide extra functionalities used by the components.

Before starting to code the application, some design was needed. Since Angular is a Single Page Application (SPA) framework, components can be nested inside each other and rewrite only the component needed as the user interacts. This avoids interrupting the user experience because the whole page is not reloaded.

The idea of the design is to have a root component, identified by the tag `<app-root>`, containing the header, the navigation bar and the footer, and, in the middle, a space for other components. The space to hold the components is identified by the tag `<router-outlet>`. Below is a schema depicting this design.
To facilitate the development of the front-end application, the team used Angular CLI, which is a tool that provides project initialization capabilities, code scaffolding and more. Angular CLI was used to generate the project (with one command) with the full directory structure and the various components of the application. The components, at this moment, are: the root component, the welcome component, the ‘about’ component, the fasta upload component, the file download component and the components to search and view sequences.

As said, angular works with Services and Components, which are accessed using a Router. The router redirects paths (or routes) into components, providing an easy way of changing the views.

When the user clicks on a link with the property \[\text{routerLink}="[\text{pathname}]"\] in an HTML file the Router is activated (code shown below), then, the router will redirect to the component registered with the pathname and Angular will display the new component inside the \(<\text{router-outlet}>\).  

---

29 Angular CLI: [https://github.com/angular/angular-cli/wiki](https://github.com/angular/angular-cli/wiki)

---

Figure 29: Angular SPA Schema
export const routes: Routes = [
    {path: '', redirectTo: 'welcome', pathMatch: 'full'},
    {path: 'welcome', component: WelcomeComponent},
    {path: 'about', component: AboutComponent},
    {path: 'downloads', component: FileDownloaderComponent}
]

Figure 30: Snippet of the Angular router

The components of the project are TypeScript classes annotated with @Component. This annotation indicates Angular that this class is a component. Also, the annotation provides metadata like the selector of the component to call the component with an HTML tag, the HTML template associated with the component or its own CSS style sheet.

```typescript
@Component({
    selector: 'app-sequence-detail',
    templateUrl: './sequence-detail.component.html',
    styleUrls: ['./sequence-detail.component.css']
})
```

Figure 31: Angular Component annotation

Since the components are in charge of the user interaction with the application, the business logic is delegated to the Services. Services are classes that provide a well determined functionality and can be used by any component or service. The Services are injected into the components or service through its constructor thanks to Angular’s dependency injection capabilities. In this sprint no services were developed because the focus was set on migrating from the prototype, the services will be developed in the next sprints.

All the components, services and routing are tied together in the main module of the application, the AppModule. The AppModule is a special class annotated with NgModule and acts as entry point of the application. There is a lot going on inside this annotation:

- In the declarations field, are declared all the components of the application, those developed by the team.
- In the imports tag: are declared the external modules used within the application.
- providers tag: declares all the services that the application provides. These services are also developed by the team.
bootstrap tag specifies which component is the root component, the <app-root>, in this case the AppComponent.

```typescript
@NgModule({
  declarations: [
    AppComponent,
    AboutComponent,
    WelcomeComponent
  ],
  imports: [
    BrowserModule,
    HttpModule,
    RouterModule.forRoot(routes)
  ],
  providers: [SequenceService],
  bootstrap: [AppComponent]
})
export class AppModule {
}
```

Figure 32: Main NgModule of the application

6.5.4 Bootstrap with Angular

In order to visualize everything correctly, the team used the Bootstrap framework to create a usable app compatible with multiple screen sizes. Although the front-end that resulted from this sprint is simple and didn’t seem to imply much work, it ended up consuming a fair amount of time. This was due to the lack of experience with Bootstrap and in front-end design in general.

Many problems arose during the migration to Angular. But the main problem was that the various bootstrap containers were not positioned correctly resulting in overlapped containers, and so on.

The problem was solved with a better understanding of how bootstrap works and a correct use of containers, rows and columns. Bootstrap works with a Grid Layout [29] with 12 columns and as many rows as needed. The container is what wraps the content and allow its centering and padding. The rows live inside the containers and each row can be separated into 1, 2, 3, 4, 6 or 12, all divisors of 12. With this system in mind, the team developed the interface as follows.

In the root component, the AppComponent, there are three containers, one for the header, one for the footer and one in the middle. The container in
the middle contains the `<router-outlet>`, where the other components are displayed. Then, all the HTML of the other components of the application assume that they are being placed inside a container and only have to worry about rows and columns. Below is a simplified template showing this final structure.

```html
<!-- Inside the root component -->
<div class="container">
    Components are loaded inside the `<router-outlet>` tag
    <router-outlet></router-outlet>
</div>

<!-- Inside a component-->
<div class="row">
    Content of the component here
    <div class="col-md">
        This a column
    </div>
    <div class="col-md">
        This another column of the same size
    </div>
</div>
```

Figure 33: Simplified Bootstrap Schema
6.6 Sprint 5

In this sprint, the team developed two main features requested by the client: the uploading of the Interpro files and the page to query sequences by the TrinityId and the InterproId, using what the team calls "Like Searches". Both features would imply a change in the design of the documents in the Mongo database.

Furthermore, a feature that would allow the user to download any sequence, or set of queried sequences as a single Fasta file was developed. Also, in the front-end, the pages to upload the Fasta and Interpro files were implemented in the Angular client using a module to upload files called ng2-file-upload30.

6.6.1 Updated MongoDB design

As explained in earlier sprints, in the initial MongoDB design, a sequence could only belong to one Interpro family. This means that between the sequence and the family must be a one-to-many or 1-n relationship, therefore, a change in the design was needed.

Representing this multiplicity in a document oriented database, like MongoDB, is not trivial, as it would be in a SQL database. There are many approaches, which depend on the cardinality of the many side of the relationship, the Family. According to MongoDB 30 31 32, there are basically three approaches: one-to-few, one-to-many and one-to-squillions.

One-to-few: When few documents are related to one. In this approach, the whole Family would be embedded inside the Sequence in an array as an embedded document. This is called full denormalization and would look like:

```
{ 
   _id: ObjectID('123'),
   trinityId: 'comp1234_c0',
   .
   . (other fields)
   .
   families : [
      { _id: ObjectID('456'), interproId: 'PF1234', ...},
      { _id: ObjectID('789'), interproId: 'PF5678', ...},
   ]
}
```

30ng2-file-upload: https://github.com/valor-software/ng2-file-upload
The main advantage of this approach is that the embedded entity, the families, can be retrieved with a single query. With one query, the user gets all the information of the sequence and all the information of the families it belongs to. However, the disadvantage is that the Families cannot be retrieved as standalone entities since they exist only inside a sequence.

**one-to-many:** When the 'n' side is much larger, several hundreds. In this approach, each entity is stored in a separated document and the one part of the relationship would have an array of IDs referencing the many part. This is called *referencing* and would look like:

**Sequence Document:**

```json
{
  _id: ObjectID('123'),
  trinityId: 'comp1234_c0',
  .
  . (other fields)
  .
  families : [ 
    { _id: ObjectID('456')},
    .
    .
  ]
}
```

The sequence only stores the ID of the family it belongs to.

**Family Document:**

```json
{
  _id: ObjectID('456'),
  interproId: 'PF1234',
  description: 'example description'
}
```

Using this approach, each entity can be retrieved separately, however, in order to join both documents, an application-level join would be required.

**one-to-squillions:** Suitable for when the cardinality of the 'n' part of the relationship is several hundreds. In this case the *parent-referencing* technique is used.

To select the appropriate approach, the real world data provided by the client was examined. It is shown that each sequence would belong to less than a dozen families, therefore, the *one-to-few* approach was the most suitable. However, the team knew that the families would have to be retrievable as
standalone entities for future features. For this reason and because the priority is to strengthen the rapid querying of the data, the team selected the following solution: Store the families twice. The first copy would be stored in the Sequence using the one-to-few approach and the second copy would be stored in a separate document, but without referencing the sequence. As follows:

Sequence Document:
{
  _id: ObjectID('123'),
  trinityId: 'comp1234_c0',
  . (other fields)
  families : [
    { _id: ObjectID('456'), interproId: 'PF1234', ...},
  ]
}

Family Document:
{
  _id: ObjectID('456'),
  interproId: 'PF1234',
  description: 'example description'
}

With this solution, the families can be retrieved alone, when only the information of the family is needed, or inside a sequence if the information about the sequence is needed. All of this without application-level joins. The downside of the solution is that families occupy twice the space.

6.6.2 Upload Interpro Files

After successfully uploading and processing Fasta files, the next logical step is to upload and process the files containing the Interpro families. The feature consists of two parts. The first one is to upload the file, this is very similar as uploading a Fasta file. The second one is to process the file to extract the Interpro families and link them with the corresponding sequences. The file structure is explained in the section State of the Art.

The front-end page to upload the files has been implemented directly on Angular, not on the prototype like the Fasta uploading feature. The new upload mechanism will be explained later in this sprint.
To upload the file a POST method has to be performed on the URL "/upload/interpro" containing the file to process. By doing so, the controller InterproUploadController is triggered. Once the file arrived at the controller, it is then stored in the file system through the class FastaFileManager (The class is called FastaFileManager but it is capable of storing any file, not just Fasta. It will be renamed in the future). After the file is stored, it is then parsed using the TrapidInterproParser class, which is an implementation of the interface InterproParser.

The parsing process goes as follows: The TrinityId, the InterproId and the description are extracted from each line of the file and stored in a wrapper class called LineItems. Each LineItem represents a sequence (TrinityId) belonging to a family (InterproID and description).

When all the lines of the file have been parsed, the resulting LineItems are passed to the class InterproStorer, the class in charge of storing the families in the database, following the approach discussed in the previous point, using the SequenceRepository and FamilyRepository interfaces provided by Spring.

The downside of this solution is that the process is fully sequential and consumes a fair amount of time and resources. First, the file is parsed and all the data (LineItems) is held in-memory, resulting in an application crash with very big files. Although, the Interpro file provided by the client (more than 40,000 lines and 4,000 families) did not cause any crash, its processing consumed some time. In the future, this processing could be improved relying on concurrency or streams.

6.6.3 Angular file uploading module

In this Sprint, the team finished the migration of the client to Angular by reimplementing the Fasta upload component and implementing the InterproUpload component using the module ng2-file-upload\(^{31}\), an Angular module designed to upload big files to the specified URL. This module was created to be used with Angular 2 but it caused no problems even though the project is based on Angular 5. The module also allows the possibility to track the upload process, the status of the upload, indicate if any error has occurred, manage queues of files to upload and even to add HTTP headers.

By using this module, the team managed to upload files in a better way. By facilitating the coding because there was no need to deal with Javascript, the module did everything. Providing feedback to the user by showing the

\(^{31}\text{ng2-file-upload: } \url{https://github.com/valor-software/ng2-file-upload}\)
upload process and the status of the upload with appropriate colors (green for success, red for error).

![New file upload page](image)

**Figure 34: New file upload page**

### 6.6.4 Like Searches

Until the fifth Sprint, the queries were done using a normal find, identifying the exact trinityId to return a sequence. One of the client requirements was to have the possibility to search sequences by trinityId, but, this trinityId did not need to be complete (it could be only a part of a trinityId). In this way, the result could be a list of sequences whose trinityId matches the partial trinityId provided in the query.

Furthermore, the client was notified about the introduction of Interpro files, and then, the client introduced a new requirement, demanding the search of InterproIds using a part of their description (a key word).

This is why the team arrived to the conclusion of trying to use like queries. At first, the team did not know if Spring offered this methods in the repositories interface, and a little bit of search was done.

Finally, the team encountered a list of possible methods that Spring repositories offered [35]. Not only the like method could be done, but most of the usual methods used to query.
Using the *Like* method, the team introduced a search by trinityId, and thanks to the *Containing* method, the team knew that searching a key word in a description was not a difficult task.

6.6.5 Custom Fasta files download

In the last Sprint meeting, after showing the feature that listed resultant sequences of a query, the client requested the possibility to download the resultant sequences as a Fasta file.

To trigger the new feature, a download button was shown at the top of the query by trinityId result. This button, at the beginning, collected all the trinityIds and sent them to the server. Then, the server could find all the sequences, storing them in a new fasta file and sending the file to the client.

One problem appeared during the development of this feature. When sequences needed to be downloaded, they were passed to the server using a GET method, and this method contained all the trinityIds in the url. This meant that the bigger the result of sequences obtained, the bigger the header would be. This supposed a problem, because the limit was set to 8KB per request and it was really easy to exceed.

**GET method**

```
/download/fasta/comp6_c0_seq1,comp61_c0_seq1,comp62_c0_seq1,...
```

Later on, the team noticed that the sequences did not need to be sent to the server. By only sending the query done to obtain those sequences,
the server could search them internally and then create the file, without involving the client.

Final GET method
/download/fasta?trinityId=X&experiment=Y

Using the method information, the server detected if the Fasta needed to be specific, having only one sequence, or if the Fasta needed to contain sequences searched using a like query.

Also, there was another problem. In the back-end, at the beginning, the Fasta was created using a List of sequences: First, a query to the database was done, storing all the result into a List. This List contained all the sequences, so depending on the number of sequences, the occupied memory would be greater, slowing the server performance. This problem was not changed until later Sprints, when the team decided to change the return type to Page.

6.6.6 Front-end changes

At the start of the Sprint, in the Sprint meeting, the feature of search sequences by trinityId was shown. Using this first version of the feature, the team extracted requirements from the client, used to change the visualization in this Sprint.

First of all, the team was sure that the result listing was primitive, so it was one of the main things to improve. The team started by improving the number of sequences being shown in the result. At that moment, the number of sequences shown was the same as the cardinality of the server response. This means that if the server returned 3000 different sequences, the front-end shown those 3000 sequences.

The team solved this by using a pagination system. By using bootstrap, the team wanted to show only a part of the resultant query, making it easy for the client to understand the result and improving the user’s computer performance.

On the one hand, the changes with bootstrap were using the pagination component offered by them. On the other hand, some changes were needed in the back-end. Right then, the result of the find methods was returned using a List structure, but, the performance in the server was lower when the result had a big cardinality. The team decided that the best option was to change the returned structure, using a Page object instead of a List.

32Bootstrap pagination component: getbootstrap.com/docs/4.0/components/pagination/
Page objects have X basic pillars: A part of the result defined by the size of the page (X size means only X sequences will be shown), an attribute containing the total of the elements returned by the query, the total number of pages used to store those elements and the page number used at the moment.

Using this new Page object, the front-end could adapt without problem the result into the bootstrap pagination style, obtaining the result shown below.

![Result pagination](image)

Figure 36: Result pagination

After the pagination updates, the team decided to create a component that could show all the information of a sequence. This component could be accessed by clicking on a sequence, in the result of a query.

This new component shown all the necessary information of the selected sequence, like the example provided below. This basic information is the trinityId, the InterproId families, the experiment, the transcript and the length of the transcript. Moreover, the team added the possibility to download this sequence in a Fasta file.
Finally, other minor changes were done in the front-end visualization. One to stand out would be the change of the header resolution and style, making it less sinful.

6.6.7 Other back-end issues solved

A part from the major features developed there were also minor issues to solve in the back-end.

- The sequences could not be retrieved through the REST API. The cause of this problem was that the ID of the Sequence document was not exposed by Spring through the `SequenceRepository`. This was solved by exposing the ID in the appropriate configuration file: `ExposeEntityIdRestConfiguration`.

- The module ng2-file-upload needed to be in the ”allowed-host” header of the server. Since the server and the client are two separate projects, they use different URLs meaning that they do not have the exact same domain. This was solved by adding the header in the CORS configuration of the server.
6.7 Sprint 6

The goals of this Sprint are to achieve a responsive front-end for the web application, to be viewed from different devices, to look for ways to improve the MongoDB and data design to allow the storage and processing and dynamic data, and to develop the visualization of Interpro data.

6.7.1 Responsive Front-end

It was important for the client to be able to use the web application from different devices: from a desktop computer with a big screen to a smart phone with a small screen. This implied two main issues. The first one was that the Bootstrap columns have to be able to adjust to many screen sizes. And the second one was that the navigation bar was not designed to be viewed on small screens, its options would overlap or even disappear.

The first problem was solved with a correct use of bootstrap columns. Bootstrap was designed to allow web pages to be correctly viewed in many different screens and provides a good and easy to use system to achieve that. Thanks to the many "col" classes, col-lg, col-md, col-sm among others, representing large, medium and small screens respectively, the team was able to specify how much width of the screen a column should occupy, achieving a responsive front-end.

The problem with the navigation bar was solved by implementing a foldable version of the bar using Bootstrap styles and a hamburger button. Basically, in screens were the bar could not be shown entirely, like in a mobile phone, a hamburger menu would be shown and, when clicking on it, it would unfold displaying a vertical list of the options on the navigation bar. As it can be seen in the image below.
6.7.2 Adding Interpro details

In Sprint 5, the addition of the feature to upload Interpro files was done. This feature meant the addition of a new attribute in the sequence information, the Interpro family.

The team added this new information in the sequence detail feature in the front-end, but, this attribute had a different responsibility, and it was to link a sequence with an InterproId. Because of this, the team introduced a new component named Family. In this component, the feature to show the details of a Family was created at that moment, but in the future, it would contain more features like the search of this Interpro families.

6.7.3 MongoDB interaction and architecture

During the past Sprints, the client shown more and more sort of files to the team. Usually, this files had simple information, but in some of the other files, the information did not have a defined structure or the content could vary without any defined parameter.

With the current architecture, the correct upload of these files was im-
possible, this is why the team started searching for solutions. First of all, the team knew that the knowledge of the Spring Repositories was not high enough, so one of the main objectives was to read and understand all the documentation about Spring Repositories.

On the other hand, the team looked for solutions to modify the architecture in order to be able to store all types of dynamic data, without having to modify the Java classes: To create an attribute with a specific structure like Map<String, Object> or using custom converters [33] [34].

Before studying the spring documentation or proposing a new design, the team studied a little bit more the structure of the information.

The information structure consists of two parts: Static data and dynamic data. When structuring static data, the information is represented in the same way in the Java class and in a MongoDB document.

Java: MongoDB:
Sequence {
    BigInteger _id, "id": ...,
    String trinityId, "trinityId": ...,
    String experiment, "experiment": ...,
    String transcript "transcript": ...
    Family[] interproIds "interproIds": {...} }

Even though families may not be present in all sequences, the majority of them will have at least one, so it means no problem.

The other kind of information is dynamic information. This information differs from file to file and the team does not know which attributes are shown in each one. Furthermore, in the future, one of the options will be the possibility to upload a file with an undefined structure (it means that the team needs to start thinking about dynamic architectures).

File:
Sequence blast_quo
comp6_c0_seq1 0.2

The information shown above contains the attribute blast_quo. This attribute is dynamic, because the program will not know the existence of it. The only purpose of this information is to be dumped in the future.
When represented in Java-Document, this information can only be stored in one way:
Java: MongoDB:

71
Sequence{
    BigInteger _id,
    Map<String, Object> dynamicData
}

This solution had problems, because the repositories do not have basic methods to find information like this dynamicData. This is why a heavy study was done.

In the study of the Spring repositories, the team learned its architecture and operations. Spring repositories have four basic parts:

- All the interfaces that form part of the Spring Repository must extend the Repository interface. This interface has the Marker Interface pattern[33] which has the objective to classify a group of classes and interfaces. In this case, the classified classes and interfaces will represent the Spring Repository.

- All the Repository Interfaces given by Spring follow the same strategy in the names nomenclature: find[Optional Attribute Name]By[Where clause]
  There are more kind of methods, shown in the Spring Repositories documentation, but are not as used as the find method.

- Spring offers the option to create Custom Repositories. Using this Custom Repositories, the developer can create new custom queries that Spring do not offer. Furthermore, this new Custom Repository Interface can be extended and used in other Custom Repositories, taking advantage of the functionality.

- Spring offers the Custom Query annotation @Query. Thanks to this annotation, the developers can do custom queries without implementing Custom Repositories. In mongoDB, this annotation gives flexibility to do custom queries really fast and without problems. Moreover, this functionality is one of the main reasons why the team has taken a decision over the new database architecture.

    Query annotation example:
    @Query(value = "{'attributes.eye.color' : ?0}")

Another possible solution was the use of Converters using the Converter spring interface. This interface allows the creation of custom converters to manage the mapping from a Java class to a Mongo document and vice versa.

[33] Marker Interface: en.wikipedia.org/wiki/Marker_interface_pattern
The idea was the following: In the Sequence Java class there would be a Map in which the dynamic data would be stored. And in the Sequence Mongo document each dynamic attribute would have its own field. See the example below.

Java: MongoDB:
```
Sequence{
    BigInteger _id,
    .
    Map<String, Object> dynamicData
}
```
```
"id": ...,
"dynamicAttr_1": value,
"dynamicAttr_2": value,
"dynamicAttr_3": value
}
```

In the Java class, inside the `dynamicData` Map there would be the entries for the attributes `dynamicAttr_1`, `dynamicAttr_2`, and `dynamicAttr_3`, but in the Mongo document these attributes would no be inside a Map but on their own. To achieve that, the mapping has to be described with Java code. Although it seems like a good option at a first glance, it requires some code and couples the Sequence class and the Mongo Document with the converter, meaning that at the slightest change, all three elements would have to be modified.

After studying the possibilities, the team decided that the `@Query` annotation with the Map structure was the best solution, because the flexibility to interact with the database and the fast way to implement the queries was an important objective.

By using this method, the team had an easy way to work with dynamic information:

Java:
```
Sequence{
    Map<String, Object> dynamicData
}
```
```
Accessed easily using `@Query(value = "{dynamicData.blast_quo : ?0}"`)
6.8  Sprint 7

Since almost the whole sprint fell on Christmas holidays, the sprint meeting was made in the second half, therefor, no requirements could be collected in time to be developed in this sprint. Because of that, no major features were developed during this sprint. Instead, the team focused on improving the overall quality of the code, with a refactoring process, and the database architecture to be able to work more efficiently in the following and last sprint.

Many improvements were made in the visual interface providing better usability and user experience and the team also designed and started implementing the REST API, which would be finished in the next sprint.

6.8.1  Refactoring the code

The client is satisfied with the project so far. All the requested features have been successfully developed and the resulting product is functional and will be ready for deployment in a real world environment by the end of the development process. However, the server-side code of the project needs a cleanup and a reorganization. For example, some classes have too many responsibilities, or some packages need to be split and reorganized.

The section of the code that needs the biggest amount of refactoring is the one in charge of managing the storage of the various interpro and fasta files. Two objectives have to be achieved in that module: First, to be able to store the files of an experiment isolated from the files of another experiment. Right now all the files are saved under the same root directory in the server. Second, to provide an easy way to interact with and manage the files uploaded by the web administrator and stored in the server.

To be able to work comfortably, a general cleanup of the code needs to be carried out first. Starting by correctly separating the controllers and services, such as the FastaParser in charge of parsing fasta files.

Summarizing, the first part of the refactoring process is a general cleanup and organization of the project and the second one is the refactorization of the classes in charge of managing the experiment files.
6.8.2 Controller refactorization and general cleanup

The refactoring started by correctly separating into ”Controller” and ”Services” the FastaParser. This class is located inside the package service. It is charge of serving as controller, or entry point, for uploading fasta files and parsing them. Then, the class has two big responsibilities and, therefore, violates the single responsibility principle.

In fact, parsing the files involves two steps, the actual parsing and the storage of the sequences as documents in the Mongo database. This responsibilities are separated in the Interpro services classes because those classes were developed later, separating these responsibilities directly. However, they will not be separated in the FastaParser because the goal is to separate controller and service. Nonetheless, it should be done in the future.

The result is a controller with the URL /upload/fasta that, when a POST method is performed with a fasta file attached, calls the service called FastaParser to process it.

Since the InterproUpload controller works in the exact same way, it was decided to join both methods under the same class: the UploadController. This class is mapped to the URL /upload and has two methods mapped to /fasta and /interpro respectively. In this way, all the uploaded files go through this class constituting a Facade and providing a better understanding of the code, at the cost of having two simple and small responsibilities, calling the appropriate service for each file.

The team developed a Facade called ExperimentManager that is intended to be used by the controller. It is a class that does not provide any specific functionality, it is simply a facade that encapsulates all the calls to the other services when uploading files. It provides complex operation in an easy way at the cost of having a high coupling with the other services. The ExperimentManager and its dependencies are shown in the following diagram.
Figure 39: ExperimentManager facade

All the functionality provided by the ExperimentManager is fully covered by a suite of unit and integration tests using mockito to mock the dependencies in the unit tests.

After the refactoring, the resulting classes were separated into different packages. Below is an image of the structure of the project before and after the refactoring.
The different packages are self-explanatory. The difference of organization can be appreciated by comparing both images. Although the structure after the refactoring is bigger, it is also much clearer.

Also, some refactoring to improve readability was done in other classes.

### 6.8.3 New File Manager

At this point, all the uploaded files are stored in the same directory, under the root directory. This makes it impossible to have more than one experiment stored because it is not possible to tell which experiment a file belongs to.

As a reminder, an experiment consists of one fasta file containing the sequences and zero or more additional files, such as the interpro files, associated with it. Therefore, each time a fasta file is uploaded, a new experiment is created.

To be able to store multiple experiments, the files will be stored in a specific directory tree. Under the root directory, specified by an environment variable there will be a folder for each experiment. With each fasta uploaded a new folder will be created and all the files related to that experiment (fasta, interpro, or others) will be stored in the same folder, separated from the
other experiments. The name of the folder will be the name of the fasta file without extension. Below is an example schema of this directory structure.

```
\root-directory
  \example-experiment  (folder for the experiment, notice the name)
    |--- example-experiment.fasta  (the file with sequences)
    |--- interpro-example.txt  (interpro families)
```

Since each complementary file is always related to one experiment, when uploading one of these files, the destination experiment will have to be specified by the administrator.

If the administrator uploads a fasta file, therefor a new experiment, that has the same name as another experiment, the old experiment will be deleted together with all of its data, and will be replaced by the new one. This behavior might seem odd, but the client suggested it in order to modify an experiment if the wrong data was uploaded.

The file manager provides methods to perform all of these operations in an understandable way for the developer, for the user, it is totally transparent. The classes that form the file manager can be seen in the following diagram.

![Diagram of Storage Service Classes]

Figure 42: Storage service classes
6.8.4 Changed architecture

During this sprint, various changes to the database design were made, however, none of these changes were big.

The first one is that each new experiment is stored in the MongoDB as an independent document, not only inside a Sequence. The purpose of that is to be able to list all the experiments currently stored in the server.

Furthermore, the team noticed that Fasta files can follow a different pattern from the one explained by the client. There are two things that do not differ: The trinityId and the transcript. This is why the way of saving the transcript length was changed. Before, the fasta service obtained the length from the parameter in the Fasta file, but now this length is obtained from the transcript, calculating the length of the String.

Moreover, the team decided to change the database architecture. Now, the Sequence had a String called `dynamicLine` and all the information extracted from a Fasta file that is not the trinityId or transcript is placed there. Thanks to this change, now a Fasta file parameter called `Path` is saved (used in a lot of programs about sequence treatment).

Because of the problems with the Page object, the downloadFasta service needed to be changed. After the changes, this service worked with any Page object that contained information about sequences. Even though those changes were done, the team knew that the service needed to work with any object and would need more changes in the next Sprint.

6.8.5 Defined REST API

One of the main objectives of the project was to implement a REST API that could give all the information to other computers/programs. Almost all the other biological web applications do not have an API defined, and the team wanted to promote the implementation of APIs demonstrating that they offer many good points.

The team decided to implement the API in this Sprint, starting with the API table definition shown below. The API will only expose GET methods because the POST, PUT and DELETE methods must be available only for the web administrator.
This was the first definition of the API, but later, the team modified the URLs to accept pagination parameters, such as page or size.

In the back-end, the API was created as a Controller, having a file for each type of Entity. Using Spring web annotations and Spring repositories, the implementation of the API was done without problems, as shown in the following example.

```java
@GetMapping(value = "", params = {"trinityId", "page"})
@ResponseBody
public Page<Sequence> getSequencesByTrinityId(@RequestParam String trinityId,
                                                 @RequestParam(defaultValue = "0") int page) {
    return sequenceRepository.findById(trinityId, new PageRequest(page, 20));
}
```

### 6.8.6 Upload file component

The front-end component to upload files to the server works correctly but some changes have been done in order to improve the usability and the user experience. These changes consists of notifications to notify the status of the uploading to the user. For example, notify when the file has been uploaded but it is still being processed, if there has been an error, etc. Below are examples of notifications.

---

**Figure 43:** API table definition.

**Figure 44:** Sequence GET method implementation.
6.8.7 Family component

In this Sprint, the team made more additions in the Family component. The new feature consisted in the addition of a button on the Family details screen, which gave the user the list of sequences that are part of that interpro family.

The team developed the feature adding a button in the family details. When triggered, a call was done to the back-end, using the GET method 
/families/${interproId}/sequences?page=${pageNumber} in the family service. This method returned a set of Sequences, that were listed in another result screen like the example below.
In this Sprint, the team did a meeting in the second half of the Sprint because of Christmas holidays, and, in this meeting the team shown the new feature to the client. Even though the client liked the new feature, it was preferred to change it a little bit.

The feature needed to start with the same button, but the search result should be shown in the same details screen instead of switching to another one. Furthermore, the client requested to have the download as Fasta file option again, so the front-end part about downloading sequences was added in this Sprint (the back-end part was added in Sprint 8).
Thanks to the *Containing* option given by repositories in Spring, the implementation in the back-end was as simple as creating the method in the family repository.

### 6.8.8 Pager component

With the addition of the interpro searches, more result listing was needed in the project. This is why it was decided to create a component to represent all the paging function. Thanks to the component, any sort of data listing was easier to implement.

With the component, when services received a response from the server, that response was mapped into a new Page object. This object had the basic information on a page and an array of indexes, which is used to define the page navigation in the bootstrap viewing.

Even though this component was created, every component using a Page has a method named rePage. This method communicates with a service, requesting the page number the user selects.
The team knows this method could be refactored into the Page component, being a method that requests a callback, which would be the call to the service, but, since the call would mean to create a callback function, the code would be more or less the same, so the team preferred to wait and see more result listing implementations before refactoring.
6.9 Sprint 8

The end of the project is near. During this sprint, the feature that was requested the most by the client will be developed, the integration of the BLAST system. Additionally, many small and medium sized improvements will be made both to the front-end and the back-end, for example, allowing to download the raw files stored on the server, improvements on the database queries, final tweaks to the database architecture and changes in the interface following the advice of a usability expert. Last, but not least, the application will be deployed to a production server for everyone to see.

6.9.1 Final changes in the Back-End

One feature that the client thought would be useful was to allow the user to download the raw files stored on the server in order to, for example, study the sequences in more depth or adding them to his collection.

Through the page [http://sttdb.udl.cat/downloads](http://sttdb.udl.cat/downloads), the user can view the various experiments stored in the server and the files belonging to each experiment. When clicking on a file, the server will send the whole file so the user can download and save it. Moreover, the files can also be downloaded directly from the REST API without having to navigate through the user interface.

During this last Sprint meeting, the team and the client decided that when a Fasta file is uploaded and there is one stored with the same name, the old one should be deleted with its stored data and the new one should override it.

The developer team implemented this requirement, so when the administrator of the database uploads a Fasta file that exists in the database, the old information will be wiped and the new one will be saved.

One of the other changes done to the Back-End during this Sprint is the new Download Fasta Service. This service was implemented with a poor architecture, so the team decided to refactor it.

The new architecture consisted of two basic parts: The *FileCreator* and the *FileWriter*. The FileCreator had the responsibility to create any file using the received data structure (of the type Iterable) and the also received FileWriter. Thanks to the injection of a FileWriter, the architecture only needed one FileCreator, and a FileWriter for any sort of file format (one for Fasta files, one for Interpro, etc).

The resultant architecture consisted about a strategy pattern, with the
first implementation of FileWriter called FastaFileWriter. This new class is injected to a FileCreator class whenever a Fasta file needs to be created.

In this last Sprint, the team also included the login feature into the system. The login feature was provided by the Spring framework, which gives the developer the GlobalAuthenticationConfigurerAdapter, to configure which login system and which password encoder are going to be used, the UserDetailsService to configure the login system, the BCryptPasswordEncoder to use as password encoder, and finally the UserDetails to implement the Administrator user in the system.

In the front-end part, the team developed a simple login system, to be upgraded in the future. To login, when the Administrator introduces the User and the Password, those are encoded using BtoA and send to the Server. This information is stored in the navigator LocalStore and used when needed.

Finally, the team knew that the security was not good in the system, and this is why only the system client is allowed to request a POST, PUT, DELETE, etc... The other applications that want to use the API are allowed to do GET methods only. Doing this, the team secures that even having an account of the system, to alter files the front-end needs to be used, so a wipe of all the information is impossible.

Figure 52: DownloadFile service
6.9.2 Database Final Architecture

At the beginning of the Sprint, the team decided to introduce a new change to the architecture to give a better read performance when searching sequences of a Family.

The idea of this upgrade was to duplicate part of a Sequence (the vital part) and save it in the Family document. This sort of Sequence contained a trinityId and an experiment and was called PartialSequence.

This new approach had advantages:

1. Thanks to PartialSequences, it was easier and faster to know the Sequences of a Family, improving a lot the GET methods.
2. With this, the document of Sequences was not big because did not need information about Families.

And disadvantages:

1. When uploading files, the Entities Family and Sequence needed to be created, and with the addition of PartialSequence, every time a Family was saved, multiple Sequence were duplicated into PartialSequence (one Family has 0 to * Sequence).
2. Every time a Sequence is deleted or a Sequence is added into a Family, the PartialSequence needs to be modified too. This is not a problem right now because the project is not too big, but later on, when more data is added, errors may appear. This addition makes the maintenance and evolution of the code difficult.

Later on in the Sprint, even though the PartialSequence approach was giving good results, the team did not trust it enough. Because of that, the team started studying new possibilities for the architecture that could be more simplistic.

After thinking about it, the team decided to delete the PartialSequence implementation and introduce a new design. The new design consisted of the addition of another Map in the Sequence entity and use the query system provided by Spring Repositories to evade some of the counter parts of stacking information in the Sequence document.

The new Map should contain information of the Families and the future Kegg and Go terms. To evade some future errors, the Map has a generic <String, DomainInformation>, String being the Entity name (Family,
Kegg, Go) and DomainInformation being Objects that extend an interface used as Marker Interface\[34\] that will help the team to control which Objects are stored into it.

The reason why this new design was implemented is the advantage of using custom queries in Spring, controlling the Map object with no difficulties and controlling the output that mongoDB returns.

For example, to query the sequences containing a family, the implemented query is:

```java
@Query(value = "{'domainInfo.families._id' : ?0}" ,
   fields = "{ trinityId : 1, experiment : 1 }")
Page<Sequence> findPartialByFamilyInterproId(String interproId, Pageable pageable);
```

In this query the returned value only contains the trinityId and experiment attributes, and thanks to the query system, the search in the Map attribute is easy.

Finally, some minimal changes were done during the Sprint, like the deletion of species attribute or the addition of Experiment as entity, being a new attribute in the Sequence entity.

6.9.3 Ending the REST API

In the past Sprint, the team developed a first version of a REST API. In this Sprint, some changes and additions were done in order to improve its usability.

First of all, in order to improve the query performance when using the GET method /families/id/sequences, the architecture of the database\[6.9.2\] and the controller calling the sequences containing a family were changed.

Before the changes, the controller did queries over Family and returned its PartialSequences, but at the end of the Sprint, the component did queries over Sequence, using the new optimized Repository methods provided by the new architecture. This new implementation was not as fast as the past one, but the performance was still good and the application gained points in write performance and code maintainability.

Finally, methods to recover the experiments stored in the server and the files associated with each one were added to the REST API under the entry point /experiments. These methods are also used in the front-end to allow the user to download the experiment files.

\[34\]Marker Interface: [en.wikipedia.org/wiki/Marker_interface_pattern](en.wikipedia.org/wiki/Marker_interface_pattern)
6.9.4 Front-end

During this last Sprint, the team devoted a lot of time trying to improve the front-end usability:

1. The team considered the API a useful tool for other web applications, this is why it was added as an important feature in the home page of the web.

2. In the Sequence details, the transcript did not have a good visualization, because if the transcript had a big length an scroll was provided. Now, the transcript is always shown, without a scroll, improving the visibility of it.

3. When searching by trinityId, no explanation or help were given, so the user could not know that the query was a like query instead of a select. This is why the team added a little description on how the query works.

4. In the upload file panel, a message is now provided when a file is being uploaded, indicating the user that the file treatment will take a bit of time.

5. In the panel admin, the descriptions were a link to the feature. Now the title will be a link too, improving the navigation experience.

6. A bug was found in the navbar when clicking the Search text, which navigated through the home page. This text was disabled.

7. A button to go back to the admin panel was added in the upload file screen because the user had no way to go back in a fast way.

8. In the Family details screen, the number of total sequences was not shown so the team added it.

9. In the keyword search, it was added a tab to select on which type of information to query (between Interpro, Kegg and Go). Even though in this Sprint the implementation of the Kegg and Go files was not done, the client wanted to improve the front-end part to know a first idea on how the screen would be.

10. Finally, a bug was repaired were sometimes the listing of sequences in the Family details was not correct and more than 20 sequences were shown.

Even though all this usability problems were improved, the team also introduced the API documentation in the Help screen of the web application. This documentation explains how to query our API and the meaning of the returned data, so the developer should understand how to use it.
6.9.5 Testing and introducing TravisCI

During the whole project the team put special attention to tests. Numerous suites of unit tests have been created while developing all the functionalities. These test suites cover the majority of the project and all the critical parts. However, each unit test only covers a small part of the code. Therefor, integration tests were needed in order to test whole functionalities in a higher level of abstraction and to make sure that the various classes work properly together. Moreover, with the implementation of the REST API, the refactor of some services, and the introduction of new security restrictions integration tests were even more necessary.

Using cucumber, the team added tests that cover all the methods provided by the API and all the main features offered by the application. All the unit tests and features can be found in the folder test of the sttdb-server project.

With the incorporation of the login system and the security restrictions, the team needed to change some aspects of the testing suits to make the tests work. Thanks to the Spring annotations @Test and @Profile("Test"), it is possible to run the testing suits with another configuration, so the user and password can have other characteristics.

Finally, at the start of the Sprint 8, the team introduced travisCI in the server repository so the project had Continuous Integration. This new tool helped the team to control the addition of new features (or changes of old ones) by running tests when a subbranch needed to be merged with the master branch in the github repository. Also, by introducing the bash command - bash <(curl -s https://codecov.io/bash) the team could use a tool to see the test coverage of the project, called codecov. These two new tools will help the team a lot in the future.

One thing to note is that the tests are only present in the back-end of the application, not in the front-end. This is because of the previous experience the team had regarding end-to-end testing in front-end. The tests ended up causing more trouble than good.

6.9.6 BLAST

One of the client’s most requested features was the integration of the BLAST system. BLAST is a system capable of comparing sets of sequences. The client wanted an interface, so that any a user could compare his sequences, of any species, against the sequences stored in the server and get a result.
BLAST is a CLI (Command Line Interface) tool with many different sub-programs such as, `makeblastdb`, used to create the needed files for BLAST from a fasta file; `blastn`, used to compare nucleotides against nucleotides; or `blastp`, used to compare proteins against proteins.

Since the user will interact with BLAST through the web application, an abstraction of the CLI must be created and exposed via a REST API. Since the first version of the product is intended to work just with nucleotide sequences, the team together with the client decided to integrate just the `blastn` and `makeblastdb` programs.

However, implementing an abstraction of the CLI from scratch was extremely tedious, time consuming and the result was hard to maintain because the team is unable to understand the logic behind BLAST and all the options that `blastn` provides. So, the team started researching to find a better solution.

Finally, the team found an open source project, developed by people who understand BLAST, that provided a full abstraction of the BLAST CLI, provided a very good user interface and, on top of all, provided a Docker container that could be executed with just one command. This project is the SequenceServer mentioned previously a few times in this document.

The SequenceServer not also provided a `blastn` implementation with all of its available options, but also all the other BLAST programs, resulting in a very versatile module of the application. Moreover, the results of the operations performed by BLAST are displayed to the user in a very user friendly manner.

Once the SequenceServer is started, it looks for fasta files in the specified directory recursively and creates all the files needed for BLAST to operate, using the `makeblastdb` program. After that, all that is needed is to go to the URL where SequenceServer is listening (localhost:4567 for testing and development), input a sequence and click the ”BLAST” button to get the results. Below are a couple of image of the resulting BLAST module in the web application.

---

35Wurmlab SequenceServer: [http://www.sequenceserver.com](http://www.sequenceserver.com)
Figure 53: Blast query page

Figure 54: Blast query result
6.9.7 Docker and Docker-Compose

Docker was the preferred mechanism for deployment from the beginning. Docker provides the ability to modularize the application into independent and isolated containers which can be managed separately with a large set of utility, management and analysis commands. Then, this independent containers can be linked together using Docker networks. Docker also provides many options for security.

Starting with the containers that the system has: The system can be separated into 4 containers. The MongoDB container which contains the database; the Spring Java server container which hosts the Springboot application; the Angular client container which contains the user interface and interaction and the SequenceServer, containing the BLAST. Each container has just one responsibility making them easier to maintain without affecting the other containers.

However, some data has to be persisted. MongoDB has to store the data in the hard drive, the Java server has to keep the fasta and interpro files and SequenceServer needs access to these fasta files. To achieve that, Docker volumes are used.

Docker containers are like a mini virtual machine that run isolated from the host. The Docker volumes define a space inside the host so that, the desired containers can persist data in them. Moreover, the volumes can be shared across containers if needed.

For that, 2 Docker volumes have been created. the first one is for the database and is called "sttdb-mongo-vol". The second one is a volume shared between the Java server and the SequenceServer and its name is "sttdb-vol".

Finally, the containers have to be linked. The Angular client is useless if it cannot interact with the back-end. Similarly, the back-end is useless if it cannot interact with the database. Therefor, the containers will be linked using the Docker network system.

By default, all containers are connected to the "bridge" network provided by Docker. Then, through the ports specifically exposed by each container, the various containers can interact with each other. The exposed ports are only accessible from within the Docker network, not from the outside. To be accessible from the outside, the ports have to be mapped, meaning that the port 'X' of the container can be accessed via the port 'Y' of the host machine.

In the following diagram, the containers, volumes, exposed and mapped
ports and the connection between containers are shown.

![Docker container structure schema](image)

**Figure 55: Docker container structure schema**

Each container has a "Dockerfile" associated which defines an image of the containers. This image contains the ports, volumes and connections shown in the diagram, along with an startup command for the container to run. The Dockerfile can also contain many other options, which will not be covered. The Dockerfiles for the server and the client containers have been created by the team and can be located at the root of the github projects, sttdb-server and sttdb-client. However, the Dockerfiles for MongoDB and SequenceServer were created by the MongoDB developers and SequenceServer developers respectively.

To run the containers, the images for the server and the client have to be build from the Dockerfiles, then, with the images, Docker is able to create the containers and run them. The images for MongoDB and SequenceServer do not need to be built since they are provided, they are just executed.

Images are created by executing the 2 following commands from within the sttdb-server and sttdb-client root directories respectively.

```sh
$ docker build -t sttdb-server .
$ docker build -t sttdb-client .
```

With this commands the docker images for the client and server will be created with the names "sttdb-server" and "sttdb-client" respectively. Finally, to create and run the containers execute:
$ docker run --name sttdb-server -p 8080:8080 sttdb-server

With this command, a container with the name sttdb-server will be created from the image with name "sttdb-server" (at the end of the command) with the port 8080 mapped to the port 8080 of the host machine. The container will start by running the command specified in the Dockerfile. The process is similar for the other images with similar with adequate options.

However, in order for the container to be built, the images have to be in the same machine. The server and client images are already in the machine, but MongoDB and SequenceServer have to be pulled. This is done with the commands:

$ docker pull mongo
$ docker pull wurmlab/sequenceserver

Although the containers are fairly easy to deploy and the application works, deploying a multi container application in this way, deploying containers one by one is very tedious and error prone because of the long commands. This is why Docker-Compose was used.

Docker-Compose is a software that allows the deployment of multi container applications very easily. It does not replace Docker, works with it. Docker-Compose allows to use volumes, specify networks, establish dependencies between containers, expose and map ports and much more. All this configuration is specified in the file "docker-compose.yml". There are two of these, one for each project, and can be found in the root directory of each project repository.

By executing the following command from the root directories of each project, the whole application will be deployed. Notice the simplicity of the command now.

$ docker-compose up

However, there is one problem. The application will be deployed in an external server and building the images there would be very unpractical. To solve this, the build images are uploaded to DockerHub[^6] from the computer of a member of the team, to the "STTDB"[^7] DockerHub organization. Then, the production server will pull the images and run the containers with the docker-compose command.

[^6]: DockerHub: [https://hub.docker.com](https://hub.docker.com)
[^7]: STTDB DockerHub: [https://hub.docker.com/u/sttdb/](https://hub.docker.com/u/sttdb/)
6.9.8 Deployment

The product is almost finished by the second half of the 8th sprint, only some minor tweaks to the interface have to be made. Since the project is Dockerized and the team understands how to create, push and run the containers of the application, the deployment process was started.

However, the final production server is still not known even though it was requested before Christmas. Therefore, in order to be able to perform some test deployments, the team asked a professor for help. The professor allowed the team to deploy the application on the server of the research group he is part of, the Rhizomik server.

This server uses a software called *Rancher-Compose*\(^{38}\) to remotely deploy applications. Rancher-Compose allows to remotely deploy Docker-Compose applications and supports the Docker-Compose version used in this project.

To be able to deploy the application, the professor provided the URL, where the server’s Rancher-Compose is listening, a username and a password. Then, after setting these credentials in the development computer, the back-end and the front-end could be deployed with one command each, executed from where the ”docker-compose.yml” are located

```
$ rancher-compose up
```

Deploying in this server provided a great experience about the common problems encountered in production environments such as, lack of ports or having to ask the server administrator for any modification because of the lack of root permissions. Also, with the application deployed, the client was able to view the new changes without having to wait for the next meeting, potentially improving the feedback.

While deploying to Rhizomik, the real production server requested arrived. But, with all the ports closed to the Internet and only accessible through the University network. Therefore, a request to the server administrator to open the needed ports was submitted. This consumed some time due to security reasons, after that, the needed ports were open.

Thankfully, root permissions and SSH access were provided, thus facilitating deployment and maintenance tasks. Docker and Docker-Compose were also installed.

To deploy the application, the following steps have to be performed:

1. Access the server via SSH.

2. Create the docker volumes: "sttdb-mongo-vol" and "sttdb-vol".

3. Pull the latests versions of the Github projects.

4. Execute the: `$ docker-compose up` from each root project directory.

Although the process is pretty simple, achieving it for the first time was rather difficult and time consuming because of several issues related to, for example, write permissions for Docker containers. Up next, the most important issues found during the deployment are explained.

1. Docker containers did not have write permissions on shared volumes because the server uses the SELINUX [35] Kernel security module. Thankfully, Docker supports SELINUX and the issue was solved by adding the ":z" tag at the end of each volume.

2. Since the back-end and the front-end have different URL, specifically, ports, the server was not recognizing the front-end as an allowed origin. The CORS filter of the back-end was modified adding the new URL provided by the production server.

3. Database maintenance [36]: Maintaining the database is crucial in a production environment. However, the database is not accessible from the outside and is encapsulated in a Docker container. So, in order to be able to perform maintenance tasks, Docker provides the command `docker exec` to execute a process in a running container, the Mongo shell in this case, having accessed the server via SSH first.

The result of this deployment process is a web application with the:

User interface, front-end: [http://sttdb.udl.cat](http://sttdb.udl.cat)

REST API, back-end: [http://sttdb.udl.cat:8080](http://sttdb.udl.cat:8080)
7 Conclusions

Once the project has reached its end and the product is deployed and functioning, it is time to evaluate the product and the development process.

Starting with the quality of the product. The product is perceived as a good quality product. Of course, there are things that can be improved or added, however, the mere fact that the product is functioning and being used in a real world environment is already a pretty good sign of quality.

Moreover, the client and the advisor, who are both professionals, are very pleased with the result. In the end, the ultimate measuring tool of a software product, or any product, is the satisfaction of the client or user.

Now, onto the development process. The team is very pleased with the methodology used during the whole process. It provided feedback fast enough to correct or rethink features without losing too much and time and provided enough time to work on features from one meeting to the next. If the sprints would have lasted for 1 week instead of two, then, the majority of the meetings would have been useless because of the quantity of work done. On the contrary, if they would have lasted for one month, then, there would not have been enough feedback.

As for the code, the team considers it to be of good quality. It is maintainable and capable of evolving. Some parts, especially the first to be developed, might need a refactor in the future to improve their readability or performance. But, for the most part, it is good.

Also, the great majority of the back-end code is covered by various suites of tests which provides security when adding new features. However, some other parts are not covered, mainly because those parts were developed in a hurry. Hurry hurts.

One other thing that would have been useful and will be considered in future projects, is to implement Continuous Integration and Continuous Deployment from the very beginning to improve the workflow and to not have all the deployment issues at the end.

In conclusion, the product has good quality, the code is maintainable, the client is happy and the team is happy.
8 Personal conclusions

In this section are written the personal conclusions about the experience with this project of the two of us, the developer team: Marc and Roger.

8.1 Marc’s conclusions

This project has shown me how important agile methodologies are and how important Software Engineering is working on a real project and with a real client helped me understand a lot of aspects of Scrum ban that were impossible to see during the degree, like the importance of the milestones or how important is to comprehend the client intentions and requirements.

Apart from the development methodology, I have learned a lot about web development during this project. All the technologies used were new to me and I think they will be of a great use in the future. Also, this project made me realize that the hate I had towards Front-End development was not good for me, and now I enjoy some aspects of Front-End development.

Furthermore, thanks to this project I discovered that I like a lot two aspects of data management: data processing and data storage. In the future, I will study these two aspects, to know if they are the paths of computer science to which I will dedicate my career.

Finally, I must say that writing this report has been exhausting and difficult because of my inexperience in writing this kind of documents. However, I am really proud of the final result.

8.2 Roger’s conclusion

For me, developing this project has been very rewarding and very painful at the same time. I’ve learned a lot about Spring, Angular, Mongo and specially Docker. I’ve really like Docker and I’ll definitely spend continue working with it. Also, I learned about project management and how to communicate with the client, who does not belong to the computer science field.

Also, the fact that this project is real has forced us to address problems that, in a hypothetical project, would have been evaded. Obviously, it was a great source of blood, sweat and tears. Real projects do this.

Moreover, I discovered that I might even like project management related tasks: planning the work, prioritizing issues, etc. I cannot say the same about front-end development, I still hate it, however, someone has to do it.
Finally, something that I consider especially important. I’ve learned to organize my work and estimate (more or less) the durations of the tasks.
9 Future Work

1. Introduce HTTPS into the web.
2. Introduce the upload of GO files.
3. Introduce the upload KEGG files.
4. Improve the Key-Word queries, adding the AND, NOT and OR possibilities.
5. Improve the Key-Word queries, adding the possibility to search into KEGG, GO and INTERPRO and filter between them.
6. Make the API HATEOAS.
7. Possibility to have more than one User in the system with different permissions.
8. Give more information about errors.
9. When a file is uploaded and is going to be reseted, warn about it.
10. Change the Sequence Detail screen, showing the information in a better way.
11. Introduce the upload Fasta files containing other information (not only nucleotides).
12. Make SequenceServer container in docker reset and get new Fasta files automatically when added using the upload component.
13. Refactor the FastaDownloader component, stop using the Page object and use a Stream connection to the database instead.
14. Add e2e testing in the front-end.
15. Change/revise the database architecture, study if the Sequence entity should only save the DomainInformation identificators.
17. The FastaDownloader component must create a file with a unique id, to evade concurrency errors.
18. Change FastaParser, making it able to parse any Fasta file (now only works with nucleotides).
19. Make file (interpro, fasta or other) parsers multi threaded.
20. Integrate GBrowse tool.
21. Get and develop more requirements introduced by the client.
References


[31] “6 rules of thumb for mongodb schema design part 2.”

[32] “6 rules of thumb for mongodb schema design part 3.”

[33] “Spring data mongodb - reference documentation.”

[34] “Spring data mongodb – indexes, annotations and converters.”

[35] “Using volumes with docker can cause problems with selinux.”