University of Glasgow | School of Computing Science

Level 3 Project Case Study Dissertation

Global Rugby Network FanZone (Web)

Ruxandra Bob
Marios Constantinou
Daniel Juranec
Arnas Kapustinskas
Andrew McCluskey

10 February 2017

Abstract

Report on Team Project 3 coursework for Group V. The GRN FanZone is a fan engagement web application developed to interface with the Global Rugby Network's online platform. The project tried to utilise emerging technologies ranging from AngularJS to Firebase, in addition to cutting-edge methodologies, such as Behaviour Driven Development and Agile project management. This report covers some of the considerations taken during development and explains some of the benefits and disadvantages of the way the development team operated. The report will also cover the lessons the team learned over the course of the six months.

Education Use Consent

We hereby give our permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format.

1 Introduction

This paper presents a case study of the project and software development process of Group V, which consists of five Software Engineering students at the University of Glasgow, as part of the University's Team Project 3 course.

Our customers for the project were Global Rugby Network (GRN), who tasked us with creating a web application that would allow semi-professional and amateur rugby clubs, teams, and players to interact with rugby fans. This social platform would then be integrated with the software that GRN already provides to rugby clubs. The end result is GRN FanZone, a fully functioning web application that allows people to follow their favourite clubs, see the latest posts made by them, find rugby matches happening near them, and much more.

The purpose of this document is to reflect on the achievements and challenges that arose during the development of this project. It also explores the concepts and methodologies touched upon in the concurrently running Professional Software Development (PSD) course, how we decided to implement them, and the effectiveness of our implementation. The document is structured in a way that will start from a customer-facing perspective, working through to the backend and then finally our project management techniques.

The rest of the case study is structured as follows.

Case Study Background presents the background of the case study discussed, describing the customer and project context, aims and objectives and project state at the time of writing.

Frontend Technology talks about the key technology used for the frontend implementation of the project. It contains an overview of the technology and justification for its usage in the project.

Backend Technology covers the key technologies used for the backend development of the project. This section will also describe some of the difficulties faced by the team in this area.

Testing will reflect on the difficulties faced when writing tests for the project in Karma Jasmine, a test-runner and framework combination recommended for use with our frontend technology. It will continue to give examples of how these testing practices benefitted the team.

Continuous Integration and Deployment discusses the Continuous Integration and Continuous Deployment techniques that were used in order to ensure higher quality of code and to continuously deliver new implementations. It also outlines challenges the team faced when using these techniques.

Project Planning will show how the project was planned out, what pitfalls we faced, and how we overcame them. It will also lay out some of the lessons we learned about planning a large project.

Agile Methodology explains some of the Agile practices we used in our project.

Change Management and Version Control describes how the team used their VCS and managed the project in day-to-day life. This section will describe the lifecycle of an issue ticket in depth.

Conclusions is the conclusion, where we evaluate our performance over the whole process, what we have learned, and how this experience can benefit us in following professional software projects.

2 Case Study Background

2.1 Client Background

The Global Rugby Network (GRN), is a free team management platform for rugby teams around the world. Their aim is to provide high-quality team and performance management tools to amateur and semi-professional rugby teams worldwide.

The GRN team consists of 7 members, all of whom have played rugby at a high level. As such, each of them understands the problem domain in great depth. After years in professional rugby, and spending time talking to amateur and semi-professional coaches, Dave Millard, the CEO discovered "amateur teams and emerging rugby nations... find it hardest to access professional team management, coaching and performance software." GRN's product combines video-tagging, team management and team communication to help fill this gap.

2.2 Initial Objectives

As mentioned above, our client's primary objective is to provide a platform that enables coaches to manage their teams. The client noticed that there was a lack of fan engagement opportunities on their platform. As a result, our project motivation and initial objective was to build an online social platform for rugby fans, who could be kept up to date with official information from teams.

GRN FanZone was developed as a web application and aims to provide users with a way of interacting with their favourite clubs, teams and players, inside a user-friendly environment. The two initial target audiences included the profile-owners and the profile-followers.

Profile-followers are the users who will be registering accounts with GRN FanZone to stay up-to-date with rugby news. As an initial step, followers sign in using Google or Facebook accounts. After this, users are presented with a list of clubs, teams and players that they can choose to follow. Another requirement is letting profile-followers "unfollow" a currently followed entity. There is also the opportunity to get information about upcoming fixtures.

Profile-owners represent the group of clubs, teams and players who have registered with GRN. These entities will not access the web application directly, but use their GRN account to provide information, which is subsequently pushed to the GRN FanZone and then presented to the profile-followers.

A club is the highest level entity in the hierarchy. A club can be the parent of many different teams - for example; a club could have a juniors team for under 21-year-olds, a women's team and a men's team. These teams will often have the same location. A team consists of many players. A player can play for more than one team - Zander Fagerson plays for both the Scotland team at an international level and the Glasgow Warriors at a professional level.

These entities will interact such that the profile-owners will be able to disseminate information to their profile-followers. In the future, the platform aims to include media-based posts such as video and audio, but as this introduces additional complexity to the platform, GRN asked that we focus on textual posts for now. This allows the post system to be extended later.

Breaking the objectives down into more distinct areas, we can see three primary areas for the platform to cover:

1. Club/Team/Player news in the form of posts
2. Club/Team/Player information in the form of their profile pages
3. Fixture information from fixture detail pages

Sharing news as posts is important to improve fan engagement as it allows profile-followers to be kept up to date with information that is generated quickly, and it is relevant only for a short period of time. The post system reflects that, as each post is gradually hidden under newer posts displaying fresher information.

The profile pages for each profile-owner are designed to allow static information to be easily found. This includes genders, locations, age-groups and so on. There are different sets of information for different entities, so these have to be reflected by the different pages.

Fixtures are more interesting, as they are a crossover - they will be over soon, but we also have to include a lot of team information on them, as well as the other information, such as the time of kickoff, or the score. We opted to make this a separate page, which is linked to from a team's page.

As the product is intended to be used across the world, internationalisation support was favoured by GRN when we suggested the idea. By easily integrating other languages, we can open up the application to a much wider range of people - for example, rugby is also popular in Italy, France and Japan, to name but a few.

In general terms, our aim was to use technology to encourage engagement amongst the rugby community - ranging from fans to sponsors.

2.3 Current State of Product

The GRN FanZone is currently deployed to `https://grnfanzone.firebaseapp.com`. The product allows social sign in using Facebook or Google accounts. Following login, the user is presented with a screen that allows them to follow clubs, teams or players. Once a user has done this and reloads the dashboard, they are presented with a news feed of posts. This feed uses infinite scrolling to reduce the amount loaded on launch but not impact the user experience.

Profile pages for clubs, teams and players are also all implemented. They display the key information for each of their respective entities. They also list recent posts that entity has made and give the option to follow or unfollow the entity.

Fixture pages show the kickoff time, the two teams playing and the score (if the game has kicked off already). The page also shows the location of the game, and displays it on a map, powered by the Google Maps API.

Regarding internationalisation support, we managed translations for: Romanian, Greek, Lithuanian, Russian, Japanese, Italian, Spanish and Chinese. These translations are mostly complete - Chinese and Spanish both have some phrases not translated. This was due to the fact that we approached speakers of these languages before all of the phrases were finalised.

3 Frontend Technology

At the start of the project, GRN chose to give us a set of technologies they wanted us to use. These included defining AngularJS as our frontend framework. AngularJS is a framework maintained by Google, which has been gaining popularity over the last eight years[4]. The framework uses Typescript, a type safe flavour of Javascript. AngularJS focuses on being portable, high performance and easy to write[8]. Many of the contributors to AngularJS are from Google, but the project is open source under the MIT license and anyone can commit to it[4]. AngularJS is built on top of NodeJS, providing easy access to a wide range of external components.

AngularJS embeds code in HTML and uses a "controller" to define component behaviour. This "component" object is the basis of most AngularJS development. AngularJS also tries to minimise the logic in controllers, by either abstracting it to other components or building "services" - shared libraries of code snippets. The final part of this structure is a "module" which coordinates resource injection into controllers from one file. A project may have more than one module for its subsections, but we stuck to one file. We made use of several services, and many components.

AngularJS provides a Command Line Interface (CLI) that makes it easy to start projects and provides templates for components, services and modules. This provides the `ng init` command, which prepares a skeleton project in a matter of minutes. The basis of the project was generated in under a minute, which set up testing frameworks,

a basic starter application and a `package.json` file for NPM. When generating a new component, a HTML template, CSS file, karma jasmine test file as well as the Typescript controller. These were populated with template code to allow easy starts. This helped ensure we could get to the business logic of the component or service quickly, without wasting time.

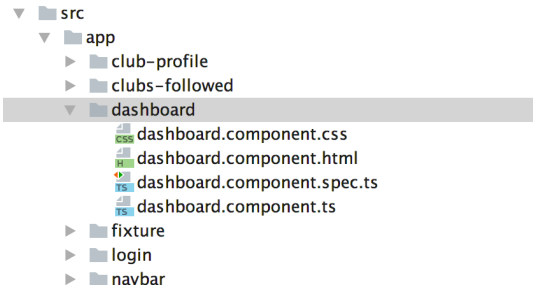


Figure 1: A selection of components used in the GRN FanZone

AngularJS gave us a huge advantage when working on this project. Components provide an easy way of separating concerns and reducing code duplication. Generating new code is easy with the CLI, and entire projects can be started and running in hours. The documentation for AngularJS is also excellent. Maintained by google, it is kept up-to-date with the regular releases AngularJS receives.

The most significant barrier to using AngularJS was our own inexperience. Thus far, our formal education has not involved developing web apps in Javascript based frameworks. Beyond this, Typescript was new to every member of the team. This meant that there was a steep learning curve for the team to deal with and adapt to. This caused some issues early on before we had a good grasp of how to use the actual features AngularJS provides.

One of AngularJS's benefits is that its developers have carefully considered testing. Software testing has long been an important aspect of software development, and this has only become more true in recent years [12]. The AngularJS CLI generates a functional karma file, which has been configured for AngularJS. The AngularJS CLI also provides an interface for running tests. Parameters range from single/constant runs of the test suite to enabling code coverage reporting. Despite this setup, we encountered some difficulties in writing good tests (please see section 5 for more on this).

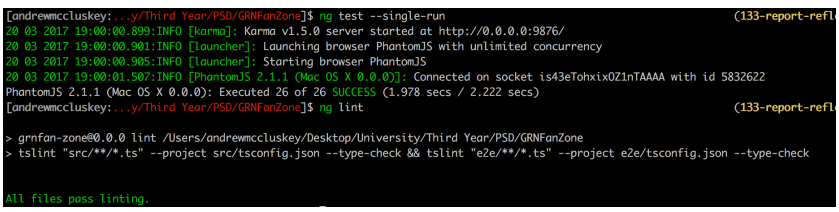


Figure 2: Using the Angular CLI to run tests and linting

4 Backend Technology

As with the frontend, our clients defined what technology they wanted the team to use for the backend of the project before we started development. We were tasked to use Firebase, which is a mobile and web application backend platform that was acquired by Google in 2014. Its initial product was a NoSQL real-time database, however, since then it has added a large range of features including static hosting services, authentication and storage.

We started off by registering our project "GRNFanZone" on the Firebase website, which provided us access to the Firebase console, as well as an API key to initialise and access the database. Afterwards, we followed the documentation to implement authentication using Facebook and Google, which was an initial requirement for the project. Finally, the team used Firebase's CLI to deploy the initial build using the provided static hosting service. This process was later refined and automated as part of our continuous deployment strategy (discussed in section 6). Overall, the ease with which Firebase allowed these features to be set up and implemented benefitted us greatly, as it allowed us to concentrate on other parts of the project.

The main benefit of Firebase for the project was the real-time cloud-hosted NoSQL database. All of the data in the database is defined using Javascript Object Notation (JSON). The database can be edited by using the editor in the online Firebase console, or by creating a JSON file and either deploying it using the CLI or importing it in the console. We used Globally Unique Identifiers (GUID) for main entity IDs, and divided our data into six main categories:

- *Users*, where every user would contain some personal information from social authentication, as well as clubs, teams and players that user follows.
- *Posts*, containing the text and title of the post, the post author ID and some information about him, the comments and likes for that post, and a timestamp.
- *Fixtures*, which contains the home and away team IDs and names, timestamps of fixture creation and kickoff, location, and the score if the fixture has already happened.
- *Clubs*, where each club has their crest, name, description, location, and the posts made by that club.
- *Teams*, which, alongside the same information as clubs, contains what club the team belongs to, the team's age group and type, alongside posts made by the team.
- *Players*, that has every players' name, age, bio, location and profile picture, as well as what team the player plays for and posts by them.

The Angularfire2 package that was used to interact with the backend used observables to provide data; information was updated on the frontend in real-time, even without refreshing the application. As the purpose of the web application was to provide a

social platform for rugby fans, all connected users being synced to the database made for a much natural and interactive user experience.

Figure 3: A snippet of the JSON used as the example database for GRN FanZone

Firebase did not, however, come without its drawbacks. As usage of NoSQL databases has only recently become more prevalent, none of the team members had previously had experience of using or modelling data for one. Perhaps the biggest issue that arose while getting acquainted with Firebase was changing our relational database data modelling mentality to one that would work well with a NoSQL database. For example, duplicated data, which is considered undesirable in an SQL database, was necessary for us to be able to retrieve information about posts efficiently. It also meant that we had more limited functionality when it came to querying the database.

Another big issue was identified later on in the project. We used the Angularfire2 module to interface between Firebase and AngularJS, which meant that all of our calls to the database had to be in the form of subscriptions to certain keys. As the application grew in complexity, more and more subscriptions had to be made, resulting in race conditions whereby the load order of the data and issues where retrieved data would be needlessly reloaded. Fixing these problems was very frustrating and took up a lot of time that could have been spent implementing new features.

5 Testing

Behaviour Driven Development is a testing methodology where the development team first describes use cases for a feature, then writes tests for it, and lastly develops the code for the feature. The team decided to use BDD as it comes recommended by

the Agile Alliance [2]. Another reason is that the AngularJS CLI sets up a Karma Jasmine testing set up by default. BDD is credited with helping to develop easily documented code - for example, by making test cases into natural language sentences (see figure 4), it becomes easier for human developers to understand what it tests, and why it is being tested [11].

Initially, the team tried to use Mocha, another testing framework which is targeted at Test Driven Development, however integrating it was difficult and was taking too long to set up, so we decided to move from TDD to BDD and just use Jasmine. BDD is an offshoot of the TDD methodology[2], that dictates that tests should be written before code. One of the issues noted with TDD is that it does not account for tests that are pointless (i.e. do not help define the required behaviour) [11].

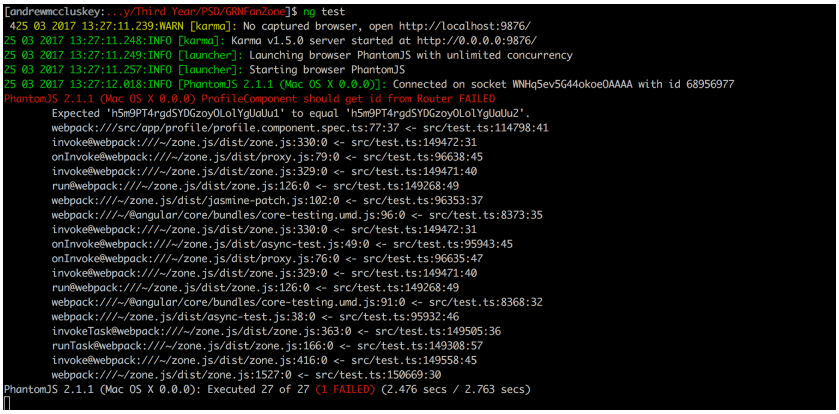


Figure 4: A failed test gives a human-readable failure message

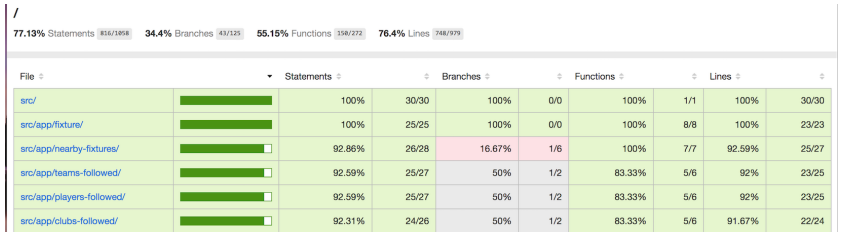
The AngularJS CLI sets up a Karma Jasmine configuration to run tests on the Typescript in the project. Karma is a test runner for Javascript that is platform agnostic and allows testing on a range of different devices and browsers[9]. Jasmine is a testing framework that aims to be as readable as possible - one of the key features of the BDD approach we wanted to adopt. One of the benefits this brought was the wealth of documentation and examples across the internet of using Karma with Jasmine on AngularJS code.

By the end of the project, a number of tests had been incorporated into the project. These tried to cover edge cases and a variety of conditions the GRN FanZone could face in the wild. Our code coverage - a metric which enumerates what proportion of the code is tested - tended to stay fairly high. We have exact figures due to our Continuous Integration, which ran the tests every time a commit was pushed(section 6), though we did not place any constraints on a minimum coverage value. Typically, the coverage ranged from 60% to 80%. Towards the end of the project, the value was just over 75%. This was representative of just under 40 tests written. To increase transparency across the project, we displayed values for our test coverage on `dev` and `master` in the readme, using icons provided by GitLab (see figure 8).

The code coverage metrics also provided a file-by-file breakdown of where coverage

was high or low. This proved to be helpful when choosing where to focus our effort - there's no point in writing tests for well-tested code, it makes much more sense to write new tests for the code that is less well tested. See figure 5 for an example of the HTML display.

Testing was not always easy. For example, getting into a habit of writing tests proved difficult for the team, showing that working in a development team on a complicated project is very different to working on individual projects. The Agile Alliance warns that "BDD requires familiarity with a greater range of concepts than TDD does, and it seems difficult to recommend a novice programmer should first learn BDD without prior exposure to TDD concepts" [2]. This turned out to be a significant difficulty for our team, demonstrating to us first hand that it is important to choose where in the project a team should focus their learning time and that too much new technology can stretch a team to the point where the effort spent to learn new things overtakes that spent on features.


/

77.13% Statements 816/1058 34.4% Branches 43/125 55.15% Functions 150/272 76.4% Lines 748/979

File	Statements		Branches		Functions		Lines	
src/	100%	30/30	100%	0/0	100%	1/1	100%	30/30
src/app/fixture/	100%	25/25	100%	0/0	100%	8/8	100%	23/23
src/app/nearby-fixtures/	92.86%	26/28	16.67%	1/6	100%	7/7	92.59%	25/27
src/app/teams-followed/	92.59%	25/27	50%	1/2	83.33%	5/6	92%	23/25
src/app/players-followed/	92.59%	25/27	50%	1/2	83.33%	5/6	92%	23/25
src/app/clubs-followed/	92.31%	24/26	50%	1/2	83.33%	5/6	91.67%	22/24

Figure 5: The HTML report on code coverage

Another struggle was getting tests to run. Due to the interconnectivity of AngularJS components, a complex data flow has to be replicated. For example, components that use the `@input()` tag to receive data don't have a trivial way of faking this. This means that the data flow there becomes difficult to test accurately. Had this not been such an issue, the post component would have been much easier to test - currently, it has the worst code coverage in the project.

There were also issues when testing backend connections. While it is allegedly possible, the team did not find a way of mocking Angularfire2, the service which provides an interface between AngularJS and Firebase. This meant that tests would affect the production database, something which could not be allowed. As a result, calls made to Angularfire2 were not tested, despite the fact that they constituted a significant amount of our business logic.

Overall, the team's testing methodologies were too optimistic. We assumed that we could make a large jump to an unfamiliar methodology, while also learning to write new tests. Another of the mistakes we made was putting testing aside and instead trying to get features out. This resulted in a code base that did not have the test coverage we desired, but also gave us a large task in writing the tests after the fact. This proved more difficult than writing them as we went along.

6 Continuous Integration and Deployment

The project used a multitude of Continuous Integration (CI) and Continuous Deployment (sometimes Delivery) (CD) techniques. A CI server's purpose "is to check the code repository for changes, check out the code if it spots any [changes], and run a list of commands to trigger the build."[10] A build is "ideally more than just compiling - it should also include a thorough test suite to help verify that the code still works with every change."[10] This gives a development team a quick, automated way of checking their code works, follows a style guide and doesn't break any other work.

One of the key concepts of CI is often phrased as: "Commit Daily, Commit Often"[10]. For our project, this was sometimes a struggle. This was due to a small number of factors, which boiled down to: "We don't work on the project every day", and "I'm not used to git". There was little we could do to remedy the former issue - all we could do was commit often *while we worked on the project*. The Gitflow branching system we used (see section 7) was unfamiliar to several members of the team, and it took time for everyone to become accustomed to the system. As the project progressed, however, more builds were made, more commits pushed, and more bugs found.

Another hurdle at which we fell was getting into the habit of writing tests for our code. I shall mention this briefly here, but for more details, please see section 5. In Fowler's 2006 paper on CI, he says: "Imperfect tests, run frequently, are much better than perfect tests that are never written at all."[7]

Continuous Deployment is the practice of continually deploying working builds to production as often as possible. It adheres to the Agile principles of:

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software. [5]
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale. [5]

In our project, as soon as a new feature is merged into the `dev` branch, tests are run, and then the changes are deployed to a staging server, hosted by firebase. Similarly, as soon as `dev` is merged into `master`, `master` is pushed to our production server, also hosted by Firebase. The `dev` branch was typically merged into `master` once a week, allowing time for any changes to be made to features that weren't quite perfect, and to iron out any bugs that were found after time in `dev`.

Our CI and CD was operated using GitLab's integrated CI system. This uses Docker to run a set of instructions defined in the 'gitlab-ci.yml' file. Instructions are separated into tasks. Tasks belong to stages - in our project, these were "test" and "deploy". The tasks are run as builds within a pipeline. The Docker instances were in some cases hosted for free by GitLab (sponsored by a cloud company). In other cases, builds were run on team computers. A common upper limit for build time is quoted as 10 minutes[7] - ours typically ranged from 5-15 minutes, with some exceptions that

were typically waiting on the GitLab CI runners to free up. The single largest time-drain was running `npm install` every time docker span up a new instance. Running builds typically took under a minute after this.

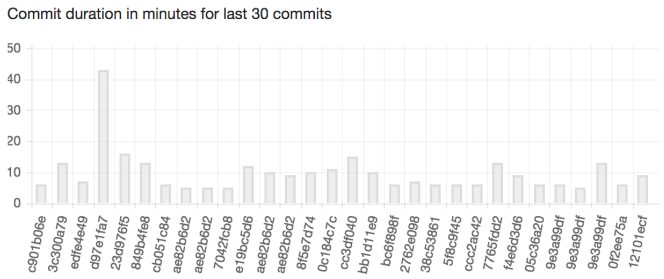


Figure 6: Build Duration for last 30 commits from 21:28 on 19/03/2017

The first task used by the project's "test" stage ran a series of lint checks over the project's source code. Linting involves performing static analysis on code to detect bugs or violations of a style guide. These kinds of checks are also performed by compilers and so on. These issues can range from missing semicolons to using a mix of double and single quotes, to whether a function is never called. The task tested CSS, JSON, Typescript, Javascript, HTML and LESS. While this was often annoying, these tests did help maintain a higher quality of code in the codebase. As our policy was to not allow a merge to `dev` take place if a branch was not passing tests, we had a method of enforcing that the standards we defined were upheld.

The second task ran the project's tests. Again, this task had to complete successfully for a branch to be merged into `dev`. This stage also generated coverage reports which were used to give the team an indication of how well our code was tested.

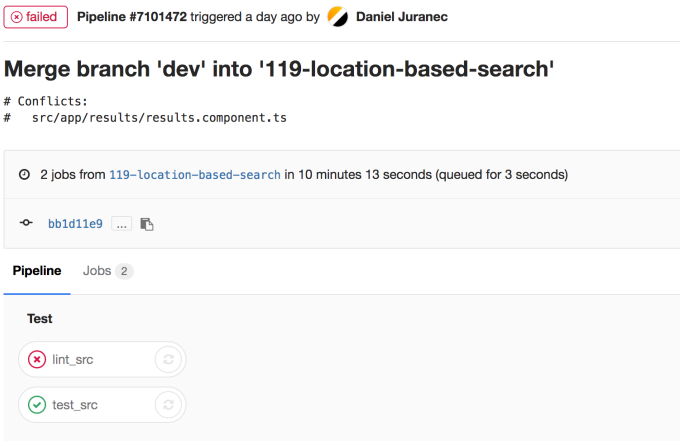


Figure 7: Example of a failed pipeline in GitLab

Merging these two tasks was considered, but left aside for now. The tasks take 5-10 minutes each to run and are run in parallel. The downside of this separation is the

fact that `npm install` is run twice. However, by leaving the tasks separate, we get a quicker, clearer indication of which part of the stage failed - actual functionality or a style issue.

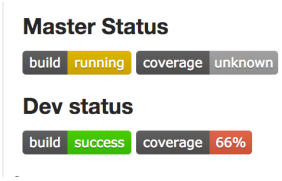


Figure 8: The buttons used to display pipeline status and test coverage

The continuous deployment tasks were both essentially the same but related to which branch was being committed to. As both `dev` and `master` can only be merged into instead of committed to, these tasks can be run only on merge commits to `dev` or `master`. The tasks run tests (to allow coverage reports for the branches) and then deploy to our live servers. The `dev` branch deploys to the staging zone, and `master` to our production site.

The fact that this is automated helps encourage rapid deployment, as the steps take some time, and are boring for people to do. This methodology takes out the human steps and means that the development team can focus on development. These rapid deployments also help by making it easy for the team to demonstrate and generate feedback from the public. This also allowed the client to continuously check our progress and give us feedback.

The CI and CD processes helped us by keeping our code of high quality, preventing broken commits and reducing manual time spent doing menial tasks. However, it took a fairly large period of time to get working and to optimise into time chunks that were consistent with the goal of 10 minutes. It is arguable that the time could have been better spent working on the actual project itself. As a counterpoint to this, it could be argued that the CI and CD have saved the development team time in fixing bugs later on, and by ensuring code is more readable, reducing time wasted understanding the code.

7 Project Planning

After the project allocation day, GRN sent us a project brief containing more details about the project itself, as well as a list of functional requirements that the final product was expected to meet. The first task we needed to complete was a detailed requirements document, that would contain non-functional and any additional functional requirements. It was decided that the focus of the first month would be allocated to finalising the document, to assure that the client approved of the direction the team decided to take in working on the project.

As part of the requirements gathering process, the team developed four personas and an epic user story of how they envisioned the project being used. This was followed by breaking down the epic into shorter, more actionable user stories centred around the following entities: team manager, sponsor, player, club supporter, rugby fan and international rugby fan. Doing this eased the process of identifying useful requirements and helped the team visualise what the final product should be offering potential users.

After the initial requirements were outlined, the team followed by designing wireframes of the web application. Initially, each team member delivered a pencil-drawn sketch of each expected application screen. Afterwards, the best design from each wireframe was merged into the main wireframes, drawn using draw.io. GRN warned us that the main wireframes would not offer useful guidance when thinking how the application should transition to a mobile format, so the team repeated the process to generate mobile wireframes.

Over the weeks dedicated to producing the requirements document, the team kept close contact with GRN, who have repeatedly offered helpful feedback and guidance regarding how the team can meet their expectations of the final product. This was in the form of a weekly progress email, and a monthly physical meeting, giving us some understanding of what the client found to be of most importance, and it helped us in the next stage in our project, which was prioritising the features of the application and separating them into achievable goals for each iteration.

There were several aspects to be taken into consideration while deciding on a concrete structure of the development process. We had to consider our lack of experience with the technologies to be used and the fact that the available time to work on the project would vary as the academic year progressed. Another thing to consider was the fact that due to the Agile methodology we decided to follow, at the end of each iteration, the product needed to be functional. Thus, it was decided to settle on having smaller, more easily achievable goals in the beginning, which would incrementally become more complex as advanced. In the end, we settled on seven iterations, excluding the initial one focused on the requirements document.

The goals for each iteration become increasingly challenging, based on the expectation that the first one or two iterations would offer all team members some hands on experience with both AngularJS and Firebase. The goals for each iteration were the following:

- First Iteration - login and sign up
- Second Iteration - dashboard and followed pages
- Third Iteration - post creation and management
- Fourth Iteration - view and edit profile, posts on comments
- Fifth Iteration - search for teams/clubs/players or fixtures
- Sixth Iteration - translation and additional tasks

- Seventh Iteration - additional features, refinement, documentation

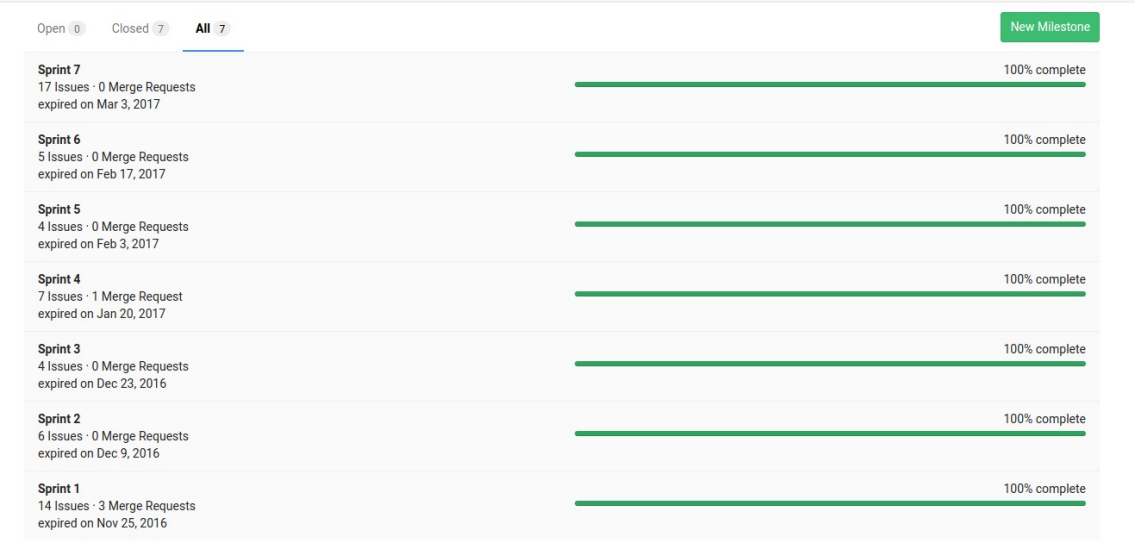


Figure 9: Project milestones on GitLab

GitLab allowed us to create project milestones. Each of them had some issues associated with it, the issues consisting of the goals for the respective iteration split into several multiple smaller issues. The deadline of each iteration would also appear on each issue, becoming bright red as the date came nearer, to alert the person assigned to the task.

8 Agile Methodology

Agile software development consists of a set of practices and methods derived from the principles described in the Agile Manifesto. Agile enforces a strong collaboration between the development team and the business stakeholders, self-organizing teams and frequent delivery of functioning software[1].

As Agile development was highly recommended as part of our software development process, in the initial team meetings, we decided to use the Scrum methodology.

Scrum is an Agile framework originally formalised for software development projects. Scrum defines several major roles within the framework: the Product Owner, the Scrum Master, the development team and the customer. The Product Owner should maintain communication with the stakeholders and create a prioritised list goals to be achieved. The Scrum Master keeps the team focused and leads team meetings[3].

Within our team, Andrew was the Product Owner and Ruxandra was the Scrum Master. After the first few team meetings, where the focus was on finalising the initial

requirements gathering, the focus was shifted to dividing the prioritised expected features. As a result of discussions between the Product Owner and the rest of the team, it was decided to split the work into seven sprints, each of them lasting two weeks. One sprint would end on Friday, and the next one would start the following Monday (Figure 10).

It was decided that instead of daily standups, the team would have two weekly meetings, one on Wednesdays, during the software development lab, and one on Fridays. The format of the meetings tried to follow the layout of a Scrum standup, including everyone describing what they had worked on since the previous meeting and the work they planned on completing before the next meeting. Each team member also discussed any difficulties they encountered, how they overcame them or whether they feel that they need help with certain issues.

The Scrum Master was tasked with closely observing these meetings. They had to determine whether the team could handle the workload of the respective sprint, if everyone in the team had something to work on and how to efficiently assign members to assist those who need help with something. The Scrum Master would also take into consideration any complaints or suggestions from the rest of the team, as well as initiate team retrospectives at the end of each sprint.

The format of the retrospectives consisted of every team member going through what they thought went well in our past iteration, what they thought went bad and what ideas they have to improve our workflow in future iterations. All these were discussed within the team and what the majority considered useful ideas were then included in the further development. The main points of each retrospective are documented on the wiki page of the project, which can be found on GitLab. One example of a point raised at a retrospective was that tickets were not being created with enough detail. In response to this, the Product Owner spent more time ensuring that the tickets were detailed. After some reading, it was discovered that it was possible to prepare templates for issues, and these were then incorporated into the software process. The retrospectives helped us air any issues we had and helped us be a better functioning team.

Another part of the software process we changed followed the review of software practices with the lab helpers. They suggested we prefix our commits with the branch number they were committed on, so that should we migrate to a new VCS we would still know which commits followed from where. This was implemented (barring forgetful moments) by most members of the team towards the end of the project.

As the project progressed, our Scrum process underwent a few changes. Once the desired goals and the predicted timelines were better defined, it was decided that one meeting during the PSD lab would be sufficient, with any additional meetings to be arranged if deemed necessary. Meanwhile, the team would communicate daily through a group chat on Facebook messenger, in order to keep everyone updated with the general progress.

Towards the end of the first semester, our sprints needed quite a few adjustments, due to the increase in additional assignments from other courses, that everyone in

the team had to complete. There was also a slight halt in communication during the winter break, as team members went home for the holidays. However, the additional time left at the end of our initial schedule allowed us to maintain the initial seven sprints and just shift everything by a few weeks.

As we became more comfortable working as a team and since at that point everyone was comfortable with the new technologies, team meetings became less structured and less formal. Along the way, the meetings became open discussions, while still making sure everyone is kept up to date. At the beginning of the second semester, our Scrum framework transitioned into a less clearly defined Agile framework, which still followed the main principles of the methodology. Sprints became of variable length (between one and three weeks) rather than a constant two weeks.

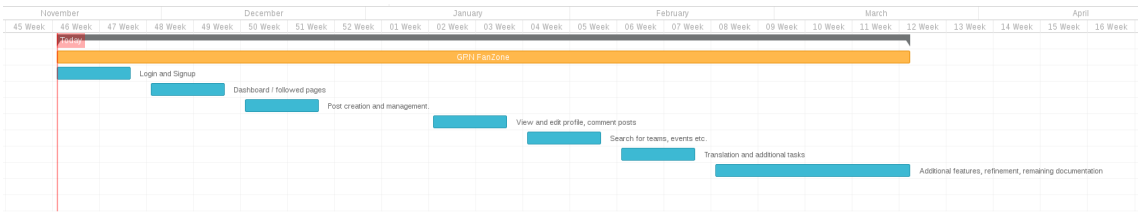


Figure 10: Gantt Chart for Development Process

At the beginning of the development process, the Product Owner split every goal into separate small issues. Over the course of the project, these were refined and split further. The members of the team were also encouraged to make small commits as often as possible and only merge one small issue at a time in a separate development branch, to avoid disrupting someone else's work by generating conflicts. Every merge request needed to be approved by at least one other team member following a code review, in order to maintain the quality of the product. These code reviews helped us to spot obvious mistakes, and to improve team understanding of how components worked.

At the end of each sprint, the product was fully or almost fully functional with respect to the features that needed to be completed up to that point. The end of most sprints coincided with the customer meetings. Thus, we were able to receive almost immediate feedback from both GRN and supervisors. This also offered us the opportunity to make additional improvements in the short time left between sprints.

To maintain the close communication with our client, as dictated by the Agile methodology, the Product Owner would communicate weekly updates by email. The team was also in close contact with the development team of our client company, which was available for questions and pointed us towards useful resources. This was helpful in assuring that our version of the product would be as close as possible to the expectations of GRN, with respect to both design and implementation.

9 Change Management and Version Control

At the very start of the project, we used a shared Google Docs folder to exchange initial ideas, such as user stories, wireframes, etc. Of course, when we started coding, some version control repository had to be selected, and our choice was GitLab. The most attractive features of GitLab for us were Git, which we were mostly familiar with at this point, issue tracking, which enabled us to submit feature requests and bug reports with ease and built-in continuous integration.

After we gathered the requirements for the project and defined the general development timeline, the issue tracker had to be filled in. This was done by the Product Owner mostly, but other team members were encouraged to report bugs and propose features too. GitLab allowed us to create the sprints (with the possibility of assigning a deadline to each of them) in the form of milestones, and afterwards, the feature requests could be assigned to a particular sprint. Additional attributes, such as assignee (the person responsible for resolving the issue), labels (feature, customer request, hotfix, non-essential, etc.), weight (1-10; decided during team meetings and representing how long they would take in hours) could also be added.

GitLab also allowed us to prioritise our tickets using a number of metrics - from the weight assigned to them to the labels. We gave `Hotfix` tickets the highest priority, as these represented bugs that existed on `master`. Following these tickets were `Bugfix` tickets, which were bugs found on `dev`. Then followed `Feature` and `Client Request` (issues raised at client meetings but not specified in the requirements document).

For our branching model, we selected one called Gitflow[6]. At its core, there are two main branches - `dev` and `master`. The `master` branch is the production-ready branch, and it reflects the state of the latest production release. Meanwhile, the `dev` branch reflects the state of our latest development efforts. We usually merged the changes from `dev` into `master` once a week to ensure the fewest bugs possible, as most of the testing was done even before merging to the `dev` branch.

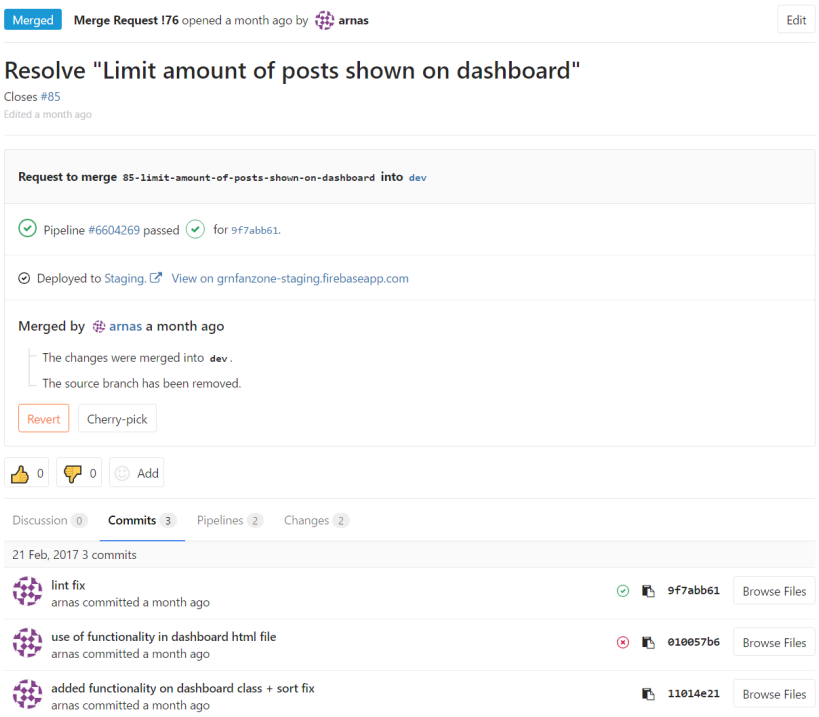


Figure 11: Example of an approved merge request

In the Gitflow model, `dev` and `master` are not committed to. Therefore all the development has to be done in separate feature branches. For each issue in the issue tracker, its assignee created a new branch via GitLab and started development in that branch. The branches were named according to this formula of:

```
$issueNumber - $issueTitle
```

Another of the useful features provided by GitLab is the ability to provide templates for feature branches. This helped make sure that feature requests and bugfixes had sufficient information in them to allow smooth development. For example, the feature template reminded the creator to add user stories and functional requirements. The bugfix template encouraged screenshots and steps for replicating the bug to be included. This meant that the reporter was not necessarily the person who resolved the ticket.

We encouraged every team member to commit often, as it encouraged transparency in the team by making completed work more visible. This policy also meant that linting and tests were run more regularly, preventing a large buildup of tidying up work towards the end of the feature branch's lifetime. By using our CI system to test every commit (section 6), we could catch bugs and linting issues early on, and fix them before they snowballed into bigger, more confusing issues. After the development of the feature had been completed, the assignee submitted a merge request for the feature branch to be merged into `dev`.

This merge request had to be approved by any other team member, following a code review. Code reviews allowed us to find mistakes in the source code not noticed by the developer, and therefore improve the quality of our project. If any mistakes were found, the reviewer could add some comments to the merge request, explaining what could be improved.

Another blocker for merge request was merge conflicts. They occurred if Git could not figure out on its own which lines of the source code to add, keep or remove, in case that files in `dev` and in the feature branch differed greatly. The assignee was responsible for this, as they would know the most about their own code. This was also often in conjunction with the rest of the team, to ensure that the merge would not result in unexpected behaviour. In some cases, the merge process did result in breaking the build. This was quickly caught by the automated tests, and could thus be promptly resolved.

Ultimately, for a branch to be merged into `dev`, it had to meet 3 criteria:

1 Passes tests and linting pipelines

2 There are no merge conflicts

3 The code passed a code review

On merge, the issue ticket was closed, the new code merged into `dev` and the feature branch being deleted. This meant that the repository was not cluttered with old branches. With over 130 branches merged, this would have been a considerable number of branches.

Encountering some issues with the workflow of GitLab was unavoidable. For example, if a feature branch was not being worked on for some time, it was getting behind `dev`, resulting in a lot of merge conflicts. The solution was either to resolve all of those manually or, if there was not much progress done on the feature branch - simply remove it and start over. Code reviews were not always performed carefully (or not at all, during crunch time), so problems were often encountered even on `dev`. On the other hand, the issue tracker did not cause any major issues; its level of convenience and robustness helped us get on the issues quickly, submit bug reports when encountered, and generally stay on top of the things.

10 Conclusions

This software project was a valuable learning experience. Each of us learned new technologies, and gained a much-improved understanding of what working in a team over a long time involves. These lessons included: teamwork, organisation, time management and communication.

One of the biggest issues we first faced was coordinating ourselves as a new team of strangers. By using icebreaker exercises to get to know one another, and discussing our strengths and weaknesses, we got to understand where each member of the team was comfortable working. There were occasional difficulties in getting everyone to meetings and ensuring work was done on time, but by communicating well, we managed to share meeting information and help each other out on difficult issues.

We also discovered that while the project's architecture is important, it is also important to remain focused on the actual product. We spent a considerable amount of time at the start of the project setting up the VCS, branching strategy, GitLab and CI configuration. While these were all valuable to the project, it would have been worth taking a step back and asking "is this the best thing for the customer?". Finding a balance is important.

As GRN already uses certain technologies (such as AngularJS and Firebase), we were asked to also use them in order to ensure compatibility. GRN also gave us feedback that differed from what we expected, resulting in us having to change our mindsets (particularly at the start of the year, before the team developed a clear vision for it). This taught us that while you may want to do things in your own way, the client is ultimately the decision-maker for many choices. By having regular contact with them allowed us to get feedback quickly and reduced the risk of a feature that would have to be completely scrapped.

Many of these technologies were new to the team (and to the industry), which meant that we had to learn them in a short period of time. This ate into development time but was also necessary to fulfil the client's needs. Learning these new technologies was interesting and challenging, but also came at the cost of time we could have spent on developing features and resulted in bugs that would not have existed if a more familiar framework had been used. Ultimately, learning to use new, different technologies is a fundamental part of being active in computer science, and the ability to adapt quickly to new ways of thinking will serve us well should we choose to go into industry.

The project gave us a chance to experience the world of software development first hand. We learnt about pitfalls to avoid in the future, but also how to work well in a team and put our education into practice. This experience will certainly be of great use to us in the future, from the technologies we learnt to the mistakes we made, to the late nights coding before client meetings.

References

[1] Agile Alliance. Agile 101. https://www.agilealliance.org/agile101/. Accessed: 2017-03-24.

[2] Agile Alliance. Behavior driven development (bdd). https://www.agilealliance.org/glossary/bdd. Accessed: 2017-03-24.

[3] Scrum Alliance. Learn about scrum. https://www.scrumalliance.org/why-scrum. Accessed: 2017-03-24.

[4] Andrew Austin. An overview of angularjs for managers. http://andrewaustin.com/an-overview-of-angularjs-for-managers. Accessed: 2017-03-20.

[5] Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Martin C. Martin, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Principles behind the agile manifesto. http://agilemanifesto.org/principles.html. Accessed: 2017-03-19.

[6] Vincent Driessen. A successful Git branching model. http://nvie.com/posts/a-successful-git-branching-model/. Accessed: 2017-03-26.

[7] Martin Fowler and Matthew Foemmel. Continuous integration. *(Thought-Works)*, page 122, 2006.

[8] Google. Angular features and benefits. https://angular.io/features.html. Accessed: 2017-03-20.

[9] Vojtěch Jína. Javascript test runner. *examensarb., Czech Technical University in Prague*, 2013.

[10] Mathias Meyer. Continuous integration and its tools. *IEEE software*, 31(3):14–16, 2014.

[11] Dan North. Introducing bdd. *Better Software*, 2006-03, June 2006.

[12] Maneela Tuteja and Gaurav Dubey. A research study on importance of testing and quality assurance in software development life cycle (sdlc) models. *International Journal of Soft Computing and Engineering (IJSCE)*, 2(3):251–257, 2012.