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 Angular to Firebase, and use cutting-edge methodologies, such as Test 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.

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 students studying Software Engineering at the University of Glasgow, done 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 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 GRNFanZone, 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 rest of the case study is structured as follows.

Section 2 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.

Section 3 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.

Section 5 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.

Section 9 is the conclusion, where we evaluate our performance over the whole process, what has been learned from it and how this experience can benefit us in following professional software projects.

2 Case Study Background

Include details of

2.1 Client Background

- The customer organisation and background.

2.2 Initial Objectives

As mentioned above, our client's primary objective is to help coaches build stronger and more successful teams, by providing them with one of the greatest and free team management systems. However, being privileged enough to have been part of the vast rugby community, our clients realized that this was not enough, and something should be done in order to try and grow this community even bigger. As a result, our project motivation and initial objective was to build an on-line social rugby platform, from where rugby lovers would be able to follow any updates concerning not only their favourite rugby Teams,Clubs and Players, but also the rugby world as a whole.

GRNFanZone, was developed as a Web Application, and aims to provide Users 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 consist of all the Users interested on keeping their knowledge about rugby up to date. As an initial step, followers should use either their Facebook account or their Google account, in order to register with FanZone. By doing so, followers are able to start following on the spot, by just clicking the follow button which appears next to each of their favourite group's resulting profile picture. In addition, included in the initial Objectives, was to give followers the ability to maintain their own profile page, where all of their followed Teams,Clubs and Players would be displayed in a good manner. In this way, Users would be able to navigate almost to every other page of the App, starting from their very own profile page. In addition, followers should have the ability to read about future events and upcoming nearby fixtures, and learn about their location by using the Google maps.

On the other hand, Profile-Owners consist of all the Clubs, Teams and Players despite of their level. Profile owners, should be able to maintain their very own profile page, where all the important information concerning the Club/Team or Player, would be released to its profile-followers. For our project's initial Objective, those information should have the form of a text Post, on which followers should be able to comment, like and read comments posted by other followers as well.

As a result, some could argue that our Project Motivation was to use technology, and help rugby community generate additional engagement not only with potential followers and supporters, but with sponsors as well.

2.3 Summary

- The final software was delivered for the customer.

3 Continuous Integration and Continuous Deployment Considerations

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."[8] 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."[8] 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"[8]. 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 *whilst we worked on the project*. The gitflow branching system we used in the VCS (see section **??**) 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 4. 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."[6]

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
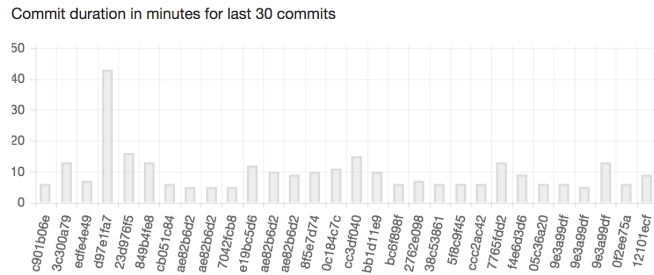
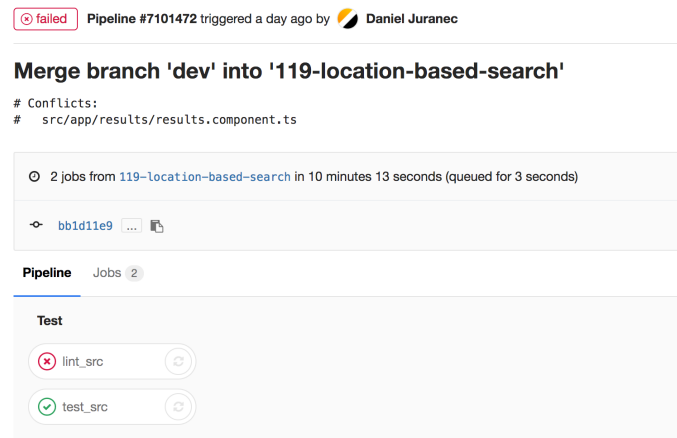Figure 1: Build Duration for last 30 commits from 21:28 on 19/03/2017

Figure 2: Example of a failed pipeline in gitlab

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[6] - 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 tests typically took under a minute after this.

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 kind 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. Whilst 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 in order 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.
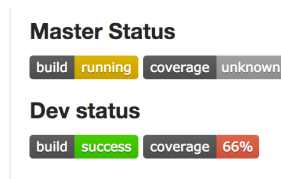
Figure 3: The buttons used to display pipeline status and test coverage

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.

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.

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 has saved the development team time in fixing bugs later on, and by ensuring code is more readable, reducing time wasted understanding the code.

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

4   Testing Considerations

Behaviour Driven Development is a development 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 reccomended by the Agile Alliance [2]. Another reason is that the angular CLI sets up a Karma Jasmine testing set up by default. BDD is credited for 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 [9].

The Angular 2 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 agonostic and allows testing over different devices and browsers. 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. Initially, we 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. One of the benefits this brought was the wealth of documentation and examples across the internet of using Karma Jasmine with Angular code.

Despite the difficulties we faced, a number of tests were written. These tried to cover edge cases and a variety of conditions they 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, though we did not place any constraints on a minimum coverage value. Typically, the coverage ranged from 80% to 60%. 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.

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.

Another struggle was getting tests to run. Due to the interconnectivity of Angular 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. Whilst it is allegedly possible, the team did not find a way of mocking angularfire, the service which provides an interface between Angular and Firebase. This meant that tests would affect the production database, something which could not be allowed. As a result, calls made to angularfire 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, whilst 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.

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

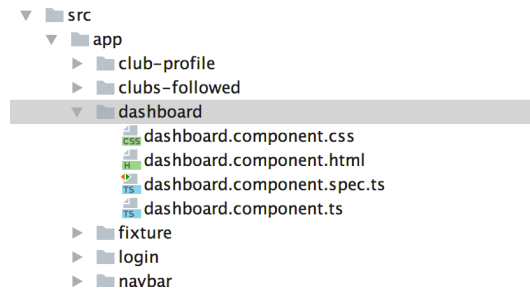Figure 5: The html report on code coverage

Figure 6: A selection of components used in the GRNFanzone

5   Frontend Technology Considerations

At the start of the project, GRN chose to give us a set of technologies they wanted us to use. These included defining Angular 2 as our frontend framework. Angular is a framework maintained by google, which has been gaining popularity over the last 8 years[4]. The framework uses typescript, a type safe flavour of javascript. It focuses on being portable, high performance and easy to write[7]. Many of the contributors to Angular are from Google, but the project is open source under the MIT license and anyone can commit to it[4]. Angular is built on top of NodeJS, providing easy access to a wide range of external components.

Angular embeds code in html, and uses a "controller" to define component behaviour. This "component" object is the basis of most Angular development. Angular 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 piece of the puzzle is a "module" which co-ordinates resource injection into controllers from one file. A project may have more than one module for it's subsections, but we stuck to one file. We made use of several services, and many components.

Angular provides a CLI that makes it easy to start projects, and provides templates for components, services and modules. This allows a "quick start" option. The basis of the project was generated in under a minute, which set up testing frameworks, a basic starter app and a `package.json` file for NPM. When generating a new component, a html template, CSS styles 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 on the configurational nightmare javascript and typescript development has become.

Angular 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 off and be running in hours. The documentation for Angular is also excellent. Maintained by google, it is kept up-to-date with the regular releases Angular receives.

The most significant barrier to using Angular was our own inexperience. Thus far, our

Figure 7: Using the Angular CLI to run tests and linting

formal education has not involved developing webapps 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 Angular provides.

One of Angular's benefits is that its developers have thoughtfully considered testing. Software testing has long been an important aspect of software development, and this has only become more true in recent years [10]. The Angular CLI generates a functional karma file, which has been configured for Angular. The Angular CLI also provides an interface for running tests. Parameters range from single/constant runs of the test suite to enabling code coverage reporting. In spite of this, it was found to be very difficult to mock our services which used angularfire to generate good tests (please see section 4 for more on this).

6 Backend Technology Considerations

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 platform that was acquired by google in 2014. Its inital product was a NoSQL realtime 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 thie initial build using the provided static hosting service. This process was later refined and automated as part of our continuous deployment strategy. Overall, the ease at which Firebase allowed these features to be set up and implemented benefitted us greatly, as it allowed us to concentrate on other needs of the project.

The main benefit of Firebase for the project was the realtime 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 inline editor in the 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 to 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 identificatory 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 him.

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 8: A snippet of the JSON used as the example database for GRNFanZone

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 to using 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 undesireable in an SQL database, was necessary for us to be able to retrieve infomation 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 into the project. We used the Angularfire2 module to interface between Firebase and Angular2, 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, due to which there arose some race conditions concerning 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.

7 Reflection on 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 have decided to use the Scrum methodology.

Scrum is an agile framework originally formalized for software development projects. Scrum defines two major roles within a development team: the product owner and the scrum master. The product owner should maintain communication with the stakeholders and create a prioritized 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 finalizing 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 6)

It was decided that instead of daily scrums, the team would have two weekly meetings, one on Wednesdays, during the software development lab, and one on Fridays. The format of the meetings included everyone describing what they've worked on since the previous meeting and the work they plan on completing until 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.

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
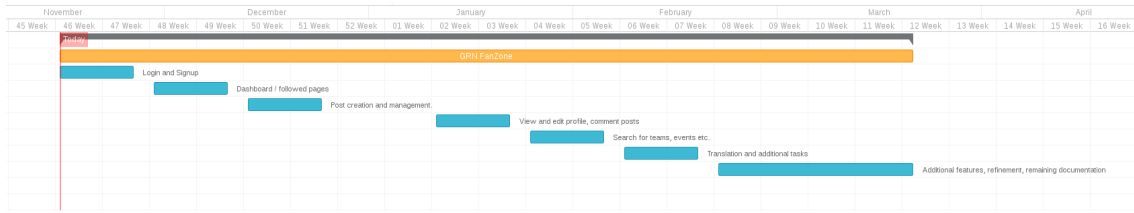
Figure 9: Gnatt Chart for Development Process

the winter break, as all 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.

At the beginning of the development process, the product owner split every goal into separate small issues. 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 the development branch, in order to avoid disrupting someone else's work by generating conflicts. Every merge request needed to be approved by at least one more team member, in order to maintain the quality of the product.

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.

8 Project Planning

After the project allocation day, GRN have sent us a project brief containing greater details about the project itself, as well as a list of functional requirements that the final product would be 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 finalizing the document, in order 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 description of how they would end up using the FanZone in their daily lives, illustrating that by documenting epic user stories. This was followed by writing down user stories centered 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 visualize 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 were 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 gave 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 expectancy that the first one or two iterations would offer all team members some hands on experience with both Angular2 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
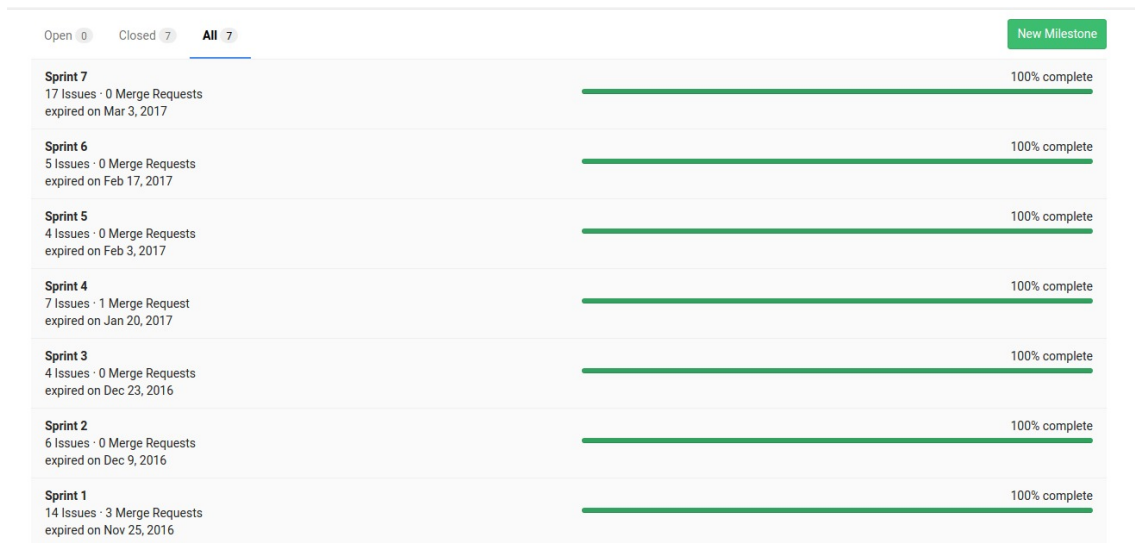
Figure 10: Project milestones

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.

Gitlab allowed us to create project milestones. Each of them had a number of 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 issues, becoming bright red as the date came nearer, in order to alert the person assigned to the task.

Throughout the project, our process was modified and improved by having team retrospectives around the end of each iteration. The format of the retrospectives consisted of every team member going through what they thought went good 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.

9 Conclusions

Explain the wider lessons that you learned about software engineering, based on the specific issues discussed in previous sections. Reflect on the extent to which these lessons could be generalised to other types of software project. Relate the wider lessons to others reported in case studies in the software engineering literature.

10 Appendices

10.1 Glossary

BDD - Behaviour Driven Development CLI - Command Line Interface TDD - Test Driven Development VCS - Version Control System

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 overvi 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] Martin Fowler and Matthew Foemmel. Continuous integration. *(ThoughtWorks)*, page 122, 2006.

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

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

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

[10] 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.