



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

Software engineering

This paper presents a case study of...

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. Sections 3 through Section 4 discuss issues that arose during the project...

2 Case Study Background

Include details of

- The customer organisation and background.
- The rationale and initial objectives for the project.
- 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." [?] 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." [?] 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" [?]. 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 ?. 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." [?]

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. [?]
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale. [?]

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

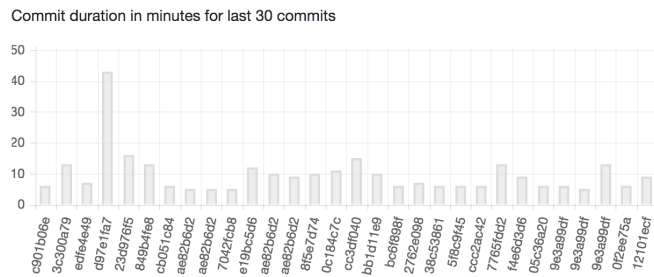


Figure 1: Build Duration for last 30 commits from 21:28 on 19/03/2017

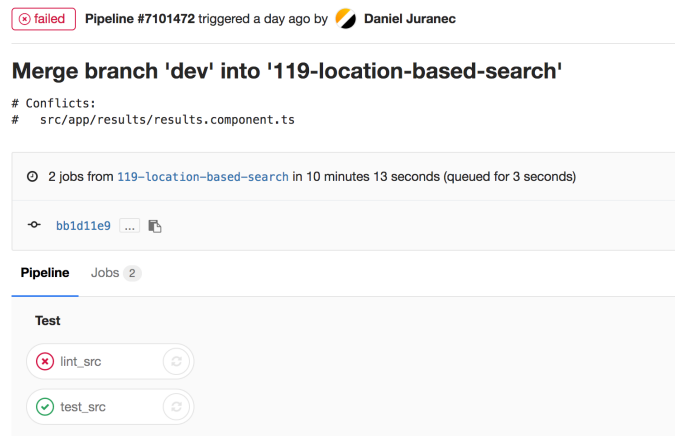


Figure 2: Example of a failed pipeline in gitlab

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[?] - 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.

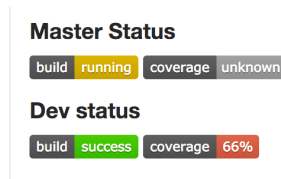


Figure 3: The buttons used to display pipeline status and test coverage

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.

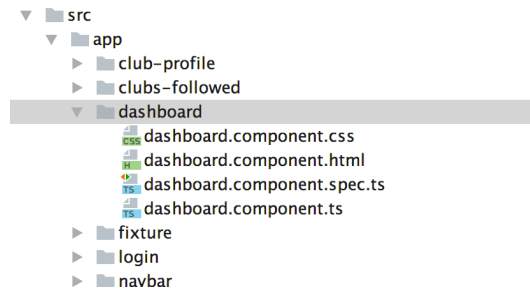


Figure 4: A selection of components used in the GRNFanzone

4 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[?]. The framework uses typescript, a type safe flavour of javascript. It focuses on being portable, high performance and easy to write[?]. 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[?]. 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

```

[andrewmccluskey:~/Third Year/PSD/GRNfanZone]$ ng test --single-run (133-report-refl
20 03 2017 19:00:00.899:INFO [karma]: Karma v1.5.0 server started at http://0.0.0.0:9876/
20 03 2017 19:00:00.901:INFO [launcher]: Launching browser PhantomJS with unlimited concurrency
20 03 2017 19:00:00.905:INFO [launcher]: Starting browser PhantomJS
20 03 2017 19:00:01.507:INFO [PhantomJS 2.1.1 (Mac OS X 0.0.0)]: Connected on socket is43eTohxix0ZinTAAAA with id 5832622
PhantomJS 2.1.1 (Mac OS X 0.0.0): Executed 26 of 26 SUCCESS (1.978 secs / 2.222 secs)
[andrewmccluskey:~/Third Year/PSD/GRNfanZone]$ ng lint (133-report-refl
> grnfan-zone@0.0.0 lint /Users/andrewmccluskey/Desktop/University/Third Year/PSD/GRNfanZone
> tslint "src/**/*.ts" --project src/tsconfig.json --type-check && tslint "e2e/**/*.ts" --project e2e/tsconfig.json --type-check

All files pass linting.

```

Figure 5: 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 [?]. 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 ?? for more on this).

5 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.[?]

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.[?]

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

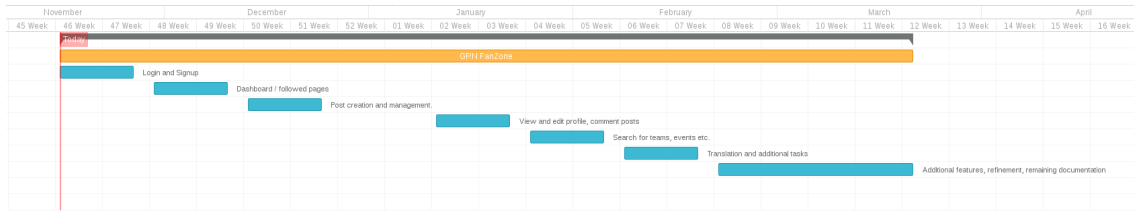


Figure 6: Gantt Chart for Development Process

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.

6 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.