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.

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 5 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."[6] 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."[6] 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"[6]. 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."[4]

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. [3]
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale. [3]

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

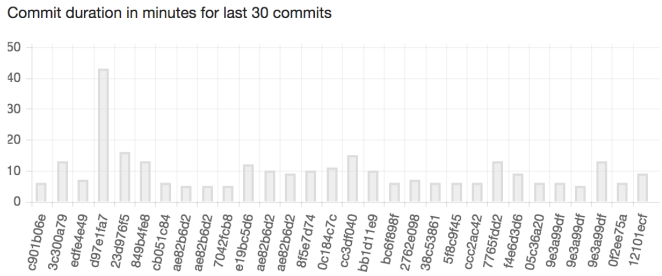


Figure 1: Build Duration for last 30 commits from 21:28 on 19/03/2017

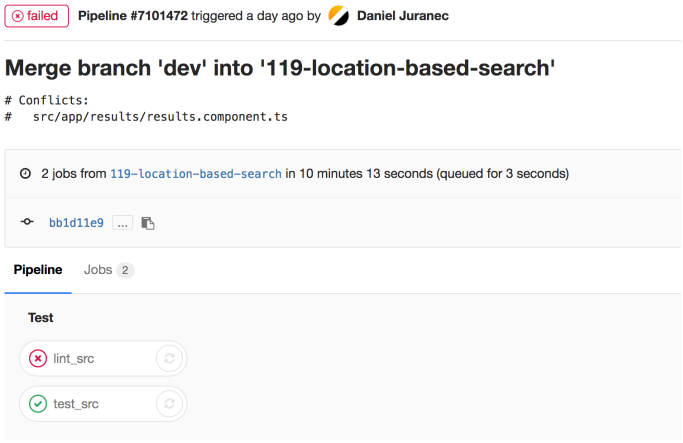


Figure 2: Example of a failed pipeline in gitlab

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[4] - 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.

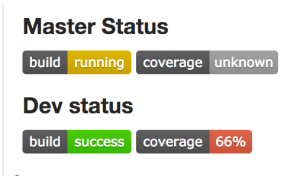


Figure 3: The buttons used to display pipeline status and test coverage

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.

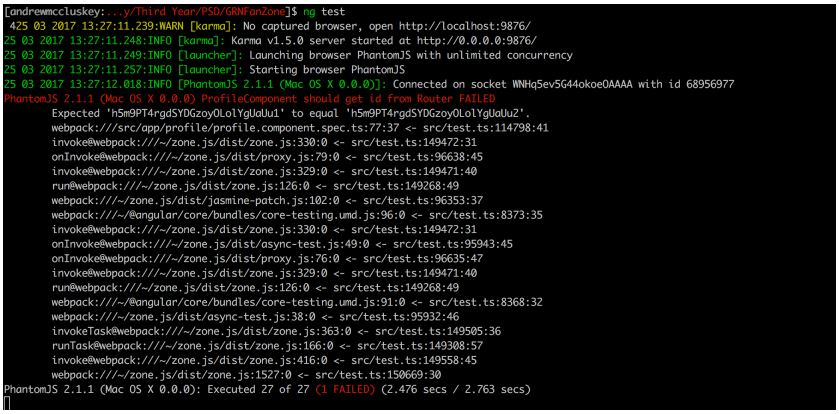


Figure 4: A failed test gives a human-readable failure message

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 [1]. 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 [7].

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.

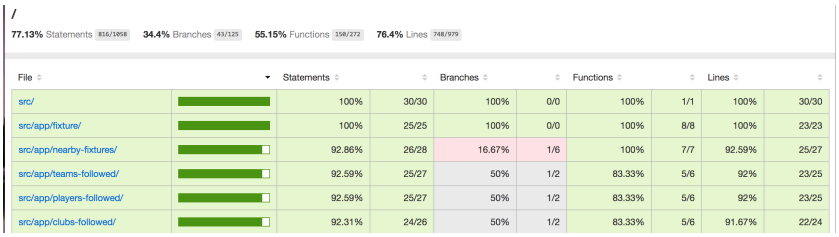
The code coverage metrics also provided a file-by-file breakdown of where coverage was high or low. This proved to be helpful when choosing where to focus our effort - there's no point in writing tests for well-tested code, it makes much more sense to write new tests for the code that is less well tested. 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" [1]. 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

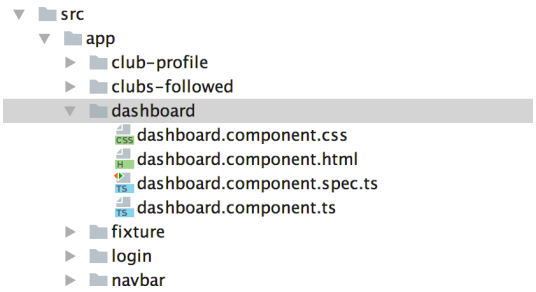


Figure 6: A selection of components used in the GRNFanzone

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[2]. The framework uses typescript, a type safe flavour of javascript. It focuses on being portable, high performance and easy to write[5]. 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[2]. 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

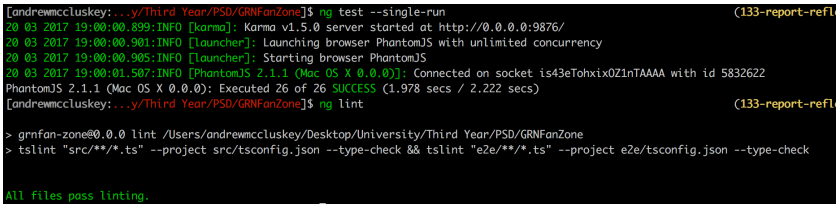


Figure 7: Using the Angular CLI to run tests and linting

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 [8]. 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 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.

7 Appendices

7.1 Glossary

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

References

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

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

[3] 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.

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

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

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

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

[8] 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.