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

The abstract goes here

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 **??** through Section **??** 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, and run a list of commands to trigger the build."[3] A build is "ideally more than just compilingit should also include a thorough test suite to help verify that the code still works with every change."[3] 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"[3]. 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

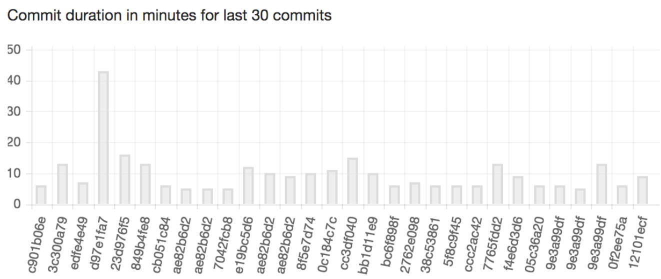

Figure 1: Build Duration for last 30 commits from 21:28 on 19/03/2017

Fowler's 2006 paper on CI, he says: "Imperfect tests, run frequently, are much better than perfect tests that are never written at all."[2]

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. [1]
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale. [1]

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[2] - 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,

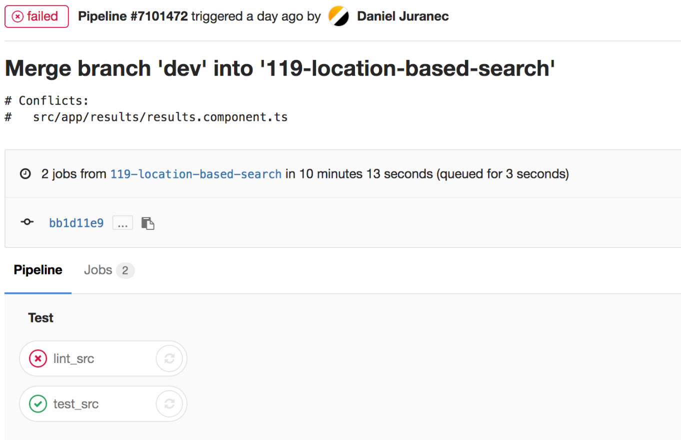

Figure 2: Example of a failed pipeline in gitlab

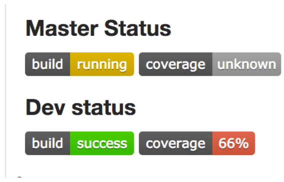

Figure 3: The buttons used to display pipeline status and test coverage

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.

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 them. 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.

4 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.

References

[1] 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.

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

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