Case Studies Critical Evaluation – Matt Stark, Dan Scott, Ross Staniland, Shane Tandy

Changes in Implementation

Throughout our project, many changes to features or planned implementation was expected and as such, we decided we would follow an Agile approach to minimise the damage of said changes.

*ASP.NET to Node.js & React*

Our original design was not very language agnostic and was based around object naming conventions and types available in .NET. When we decided to use Node.js and React for much higher portability and scalability due to the universal nature of Node, this meant that a lot of our diagrams could not be accurately created in our function names, type names and variable names. However, we attempted to keep as close to the flow of the diagram as possible and in most cases, there are one to one correlations of the functionality of the diagrams and the React/Node implementations.

*MVVM to MVC*

One of the first major changes we made was to the base architecture of the server implementation. Much of the server was based around a Service Locator pattern. Many of the services that the server used would be found using a service locator before being used and this implementation of the design pattern was great for testability.

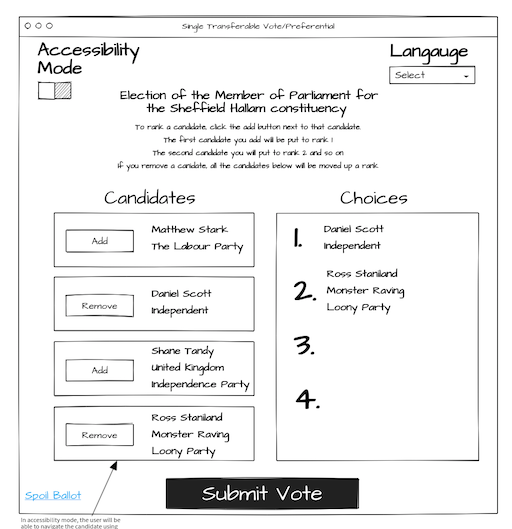
This design was changed after the decision to use Node.js was made. Instead of a service locator architecture. We used a dependency injection design pattern instead, to inject the app into the controllers so that the routes can be set up, with the concerns separated. This meant that the server configurations, database connection, database model set up and routes are all separated into services with high loose coupling from each other.



This code snippet, from our user routes configuration file, shows how routes (the main server object) is injected into the routes class and the routes are set up. This means any number of servers can be set up with the routes. This had the benefit of making the code a lot more readable and being implementable natively on Node, without the need of a heavy node module.

*Mobile First UI Changes*

Another major change to our original design is the UI design for mobile. We originally planned to create a mobile-first UI with wireframes built entirely for mobile. This, however, proved to be very difficult due to the wide range of devices that would be used in the election process. As a work around for this, we created a responsive UI to resize for different sized devices, based on the desktop wireframe. This had its pluses and minuses and meant that testing for different devices was much harder. Ultimately, these changes were made due to the wide variety of device types being used by our target audience and this meant that it wasn’t realistic to design a UI for each type of device.

This is the wireframe of the desktop voting page. We implemented a version of this page to be reactive to different sizes of devices.

*Factory pattern On Frontend*

Another major change to our technical design was to implement the use of factories for the different voting pages and voting result calculations. Originally, our design was based around an inheritance-based solution for implementing the different voting types, but this was altered to be a factory-based design including inheritance. This was changed to ensure separation of concerns. The voting page doesn’t need to know what type of vote it hosts, it is only the job of the factory to know that, and as such, it allows the same page to be used to render any and all voting types with minimal changes or hardcoded solutions.

*Single Transferable Vote and Preferential Voting*

A major change to our dynamic model was the application of the STV and PV implementations. They were planned to use an endpoint that took several votes in a list as part of the request header (through a function called SetPartiesVotedFor). This took a major change as we encountered CORS request issues on University WIFI. To fix this issue, we ended up coming up with a compromise solution of sending several requests for each chosen party. This was a very dirty solution and not the one we wanted to implement but became one of the only ways we could get the request to be accepted through to our API. In a larger scale implementation, the CORS header bug should have a proper fix in place to allow it to communicate uninterrupted.

Project Management Approach

In our project, we used a Kanban board to manage our tasks. All features and functionality were planned in advance and was placed into our “Ideas Board” column. From this, work that was accepted was placed into our “Backlog”. Our “In Progress” column has a maximum acceptance of 4 tasks (one each) to ensure that everyone is working on one task at a time. Due to this process, we used an agile Kanban Board and not a Scrum Board.

As part of our Kanban board, our testing column has a maximum acceptance of 1 as when one of the developers finished a task, it would be passed off to our dedicated tester, who could only accept one task to test at a time.

The master branch for the server GitHub repositories is set up to a Continuous Deployment server so it was important that we ensured that the master branch was always a working version, and our use of branching, testing and Kanban ensured this. In hind sight, this may have proved to be more of a problem then it added as several breaking changes that did get deployed to the live APIs because it was connected to the master branch. Fortunately, these issues were caught early and often, but it did lead to some issues where the frontend development had to be halted as the backend was down and broken.

We also implemented an expedite lane to rush any breaking changes through our development process to ensure we always had a working version on our master branch. This proved to be very successful as we never had any periods when our master branch was down for a long period of time and our implementation was also very stable. We used feature-based branching to keep new features away from the master branch until they were finished also.

Another issue we encountered was that we only had 1 dedicated tester, to 3 active developers. This led to a bottleneck in the “Awaiting Testing” column before the features could be deployed to our live version. This meant that awaiting testing became a very long backlog of work, when it should have been cleared almost immediately. In the future, to avoid this kind of backlog, the number of testers should equal the amount of active developers so features can be more quickly deployed after finishing development and testing.

In our development, we used a feature-based branching technique. This was very useful in that it stopped developers from stepping on each other’s toes while developing new features and allowed simultaneous development of features. This led to many issues though, the most problematic of which being merge errors. A lot of our time ended up being wasted fixing broken merge errors when we could have been adding new features. We learnt that this was an inherit problem with the feature-based branching as a feature could change a lot of other files needed by other people. This could have been avoided by using a branching system based on our Kanban board system. By having a branch for each column on our Kanban, all developers would be receiving commits from multiple features meaning that no major merge conflicts would occur, but this is obviously not a scalable solution to a large development team.

We believe our approach was very successful as we manged our time through the project successfully, and there was much visibility what each member was working on at each stage of the development process, and each feature moved along our process in an Agile and evolving fashion. Due to our agile approach, we dodged many issues towards new requirements and issues with our plan as the development process continued.