


ANEXD

Waqas Aslam wa40
Alex Austin aa745
James Grant jhg7
Fred Barnes
CO600: Language Selection and Dependencies

Language Selection and Dependencies

Front End

Selecting our frontend programming language took a bit of careful deliberation. As a team, we all had some varying levels of expertise in a web development environment.

Several programming languages for the front-end were proposed, some of which included:
- Java Applets & Spring MVC
- Ruby on Rails
- Variety of JavaScript Libraries
- Python Web Frameworks

We began by viewing the application from an almost ubiquitous perspective, planning and discussing the organisation of the application, while drilling towards some of the finer elements of the Anexd platform.

It had transpired that everyone sought their personal preference into how the front end should be developed. Through this, we discovered that the team had possessed some form of applied knowledge in JavaScript.

Based on our expertise and the project specification, JavaScript had been identified as a viable solution for what we needed to accomplish. From an academic judgement, we wanted to assure the complexity of the implementation was enough to satisfy our personal learning, while continuously supporting a contemporary style interface.

After some careful research, the decision led us to use Angular.JS as our preferred method for developing our frontend.

Additionally, Alex Austin and Harry Jones hold extensive knowledge of the Angular.JS framework and were confident with their understanding of the implementation mentioned above.

Strengths of Angular.JS

After carrying out a review of Angular.JS, we found a variety of strengths surrounding the Angular.JS framework and took the confinements of the language into consideration. Angular is a clever way of producing Single Page Applications (SPAs). The SPA implementation gave us the freedom to develop the HTML page once and the rest of the functionality and interaction written in JavaScript. The rest managed through Angular.JS.

Naturally we noticed that Angular is an open source application, maintained by corporations such as Google and a community of developers.

We noted that Angular provided some favourable features which would significantly improve functionality in a clean, streamlined and maintainable manner.

Some additional capabilities which we identified as beneficial towards our project are as follows:

- Creating a single page application between the model and view controllers.
- Angular code is highly testable.
- Reusable components
- A write less, get more functionality design
- Styling is simplified
- Views are in HTML, the business logic written in JavaScript.
- HTML binding capabilities for a responsive experience

To further aid us in the selection process we looked at some constraints which could cause limitations or provide focus in some areas more than others. Some things we found were:

- Since Angular is a Single Page Application, to some user configurations this could be a disadvantage. For example, if a user has JavaScript disabled, the user will only see a basic page and potentially lose out on much of the functionality.
- Since Angular.JS runs on JavaScript, we identified that we needed to focus on some of the security side of the application such as authentication and authorization.

Dependency Handling

We explored package managers to complement our frontend development environment.

Initially, Gulp was our automation tool of choice as it was quick in comparison to some of the tools we explored. However, due to some technical issues and complications, we decided that it was not working as well as we had originally hoped.

Both Bower and npm were an alternative to Gulp. We found that Bower could handle the frontend and npm dealing with the back-end dependencies. We decided that both these packages would help automatically manage dependencies, keeping all frontend components and developers on the same page respectively.

Back End

We wanted to create something consistently fast and efficient - building an application based on concurrent design was ideal. With this in mind, we knew the implementation would be intricate but scalable and very fast.

Additionally, by this point we had significant research on the connection technology we were planning on using; WebSocket connections using the popular and widely adopted Socket.IO implementation.

When looking at our project concept; the most comparable examples documented online seem to have been written with a Node.JS backend. However, these examples are only built to support a single session being run, and with Node.JS's performance in benchmarking, made it a poor choice. This is likely due to Node.JS being completely single threaded – something that is unacceptable to our concept.

We also considered languages of which we were most experienced with; Java and Python. Both of these offered some degree of concurrency, and performed reasonably well in the WebSocket benchmarks, however they still scored significantly lower than the naturally concurrent languages Erlang and Go.

Our initial choice of programming language was Erlang, for which we fortunately had experience with from our second year of university. Our research showed that Erlang had the most impressive performance for handling extreme quantities of WebSocket events.

Despite Erlang seeming like the ideal language for our solution, upon further research it was discovered that the available Socket.IO libraries in Erlang were significantly outdated, lacking much of the current specification. These findings highlighted a critical problem if we were to go ahead with development in Erlang. In order to proceed with Erlang, we would either have to choose a different connection type (such as pure WebSockets), or implement Socket.IO functionality ourselves (which would be another project in itself).

Decision to Use Go

This left the relatively new language Go as the best language available for our concept, as it performed comparably to Erlang, while also having an up to date Socket.IO library, and fortunately supported all of our other requirements, such as JSON type conversion. Having looked into Go more, it offered many other benefits that our platform would make use of, such as exposure to low level features such as pointers, some intuitive syntax, native data structure support, while also being an imperative/procedural language like others we have experience with.

No one in our group had any experience with Go, and so for our project James Grant and Alex Austin accepted the challenge of picking up the language and implementing with it; both working on their respective areas of strength and experience. James had prior experience with low level programming in C/C++, and systems programming, while having relatively strong ability when it comes to data structures, efficiency and algorithms. For this reason, James was given the job to program the core engine of the platform. Alex has a lot of experience in interfacing between multiple components in large and complex systems, so was given the role of creating the API layer of the backend, used as an access point for the frontend to obtain data through. Additionally, with Go being an increasingly adopted language, being backed by Google, and offering a combination of useful characteristics, the language seemed interesting and offers potential for future use outside of the Anexd project.