Project Title	CoderDojo Zen Projects
Author	Graham Edmond Bartley
Supervisor	Dr. Geoff Hamilton
Document Type	Technical Guide
Student Number	14541017
Completion Date	20/05/2018

Abstract

This project aims to build a project creation, management and interaction solution for the CoderDojo Foundation (CDF) and their community platform Zen which is currently used to manage over 1,000 Dojos (coding clubs) and their events across the world. This solution allows youths to share software projects they have worked on by uploading their code which is automatically stored and versioned using GitHub. Projects can then be run and interacted with directly in the browser including Python 3, Node.js, HTML5 and Java projects.

Table of Contents

Motivation

Background Information

CoderDojo is a global movement of volunteer-led programming clubs for youths aged 7 to 17 years old. The CoderDojo Foundation (CDF) exist to support and scale the movement and also develop and maintain Zen, the CoderDojo community platform (https://zen.coderdojo.com/). The current purpose of Zen is to provide a place for all types of CoderDojo community members to manage and interact with Dojos (coding clubs) and the events that they run. Dojos are created and owned by volunteers called “Champions” who must register their Dojos on Zen for them to be allowed to operate.

The Problem

While working with the CoderDojo Foundation on my INTRA placement, I was asked if I would like to work on a project for Zen after INTRA had finished. The project they proposed was to extend Zen in order to allow youths to showcase coding projects that they have worked on directly on the platform. This functionality had been highly requested by various users of Zen over the past year and would have to involve the ability to run and interact with the projects within Zen and versioning for the projects using an existing version control system. The technologies they proposed to allow youths to created projects for were Scratch 3, Python 3 and HTML5 but the solution would have to be easily extendible in order to add support for more technologies in the future (such as Java). In addition to these requirements, I would also have to handle management of the projects by the various types of users who interact with Zen.

Research

Project Technologies & Tools

In terms of technologies to use for my project I decided the best approach would be to keep them consistent with the technologies used to write Zen so that my system would be easy to integrate into Zen when it is finished. This meant writing the backend code in NodeJS and the frontend code using the VueJS framework. I also decided on using PostgreSQL for my database for the same reasons. I researched into different operating systems to use for development and found Fedora to be the most suitable for me so I used Fedora to develop for the entirety of the project. In terms of tools, I went with Atom as my text editor and Pivotal Tracker as my project tracker. These were tools I had used in the past and found to work well for me so I decided they would help with increasing my productivity throughout the project.

Runtimes

I anticipated runtimes being one of the most challenging aspects of the project. With this in mind, I did a lot of research into different methods of running code in a browser environment and carried out various different experiments into these methods. I initially wanted to run project code directly in the client's browser since this seemed like the most secure way to run code uploaded by users since it could not interact with the server at all if it was only running on the client machine. However, this is difficult to do since browsers are generally only able to run JavaScript natively but I wanted to run Scratch 3, Python 3 and possibly other languages like Java as the project went on.

One method I found of running other languages in the browser is called transpiling. Transpiling is the process of converting source code in one language to source code in another language on a similar level. I discovered that Python 3 can be transpiled into JavaScript using libraries such as Transcrypt and then the JavaScript can be run in the browser directly. I experimented with this for a while and it seemed promising until I ran into issues with dependencies and a bug with the library which I made known to the developers who said I would have to wait for a fix for it. The issue with dependencies is that libraries that people may want to use in their projects would not be available to the transpiler and so code using any kind of libraries would not transpile.

An alternative I tried was called Brython. Brython allows Python to be run in the browser in the same manner as JavaScript which seemed ideal for what I was looking for. Unfortunately, I realised that this idea of running the projects on the client-side was just not going to work for what I was trying to do. The very nature of JavaScript in a browser is event-driven whereas the code I was trying to run is blocking in nature. This subtle distinction meant that it simply would not work how I wanted it to.

I then decided I would have to run the code uploaded by the users on the server. This posed security issues which is why I initially wanted to avoid it if possible. However, after doing some research into how it could be done securely I decided that running the code in containerized environments on the server using Docker would be the best option for me. Docker is a technology which allows for images of particular operating system environments to be specified which can then be used to create containers. Containers are like lightweight virtual machines containing only what is specified in their image and nothing more. If the user's code was run inside of a Docker container on the server it would be unable to cause any harm since it operates inside it's own lightweight environment and cannot access the server filesystem.

I also noticed during my research that a release date for Scratch 3 had not yet been announced since it is a new version of Scratch. This posed a risk for the project since I was hoping to include support for Scratch 3 projects in my system. I kept a close eye on announcements of a release date for Scratch 3 throughout the project and it was eventually announced that it would be released in August 2018 which meant it would be out of scope

for my project schedule. I decided instead to implement support for NodeJS and Java projects since they are the next most popular technologies used in Dojos.

Version Control Integration

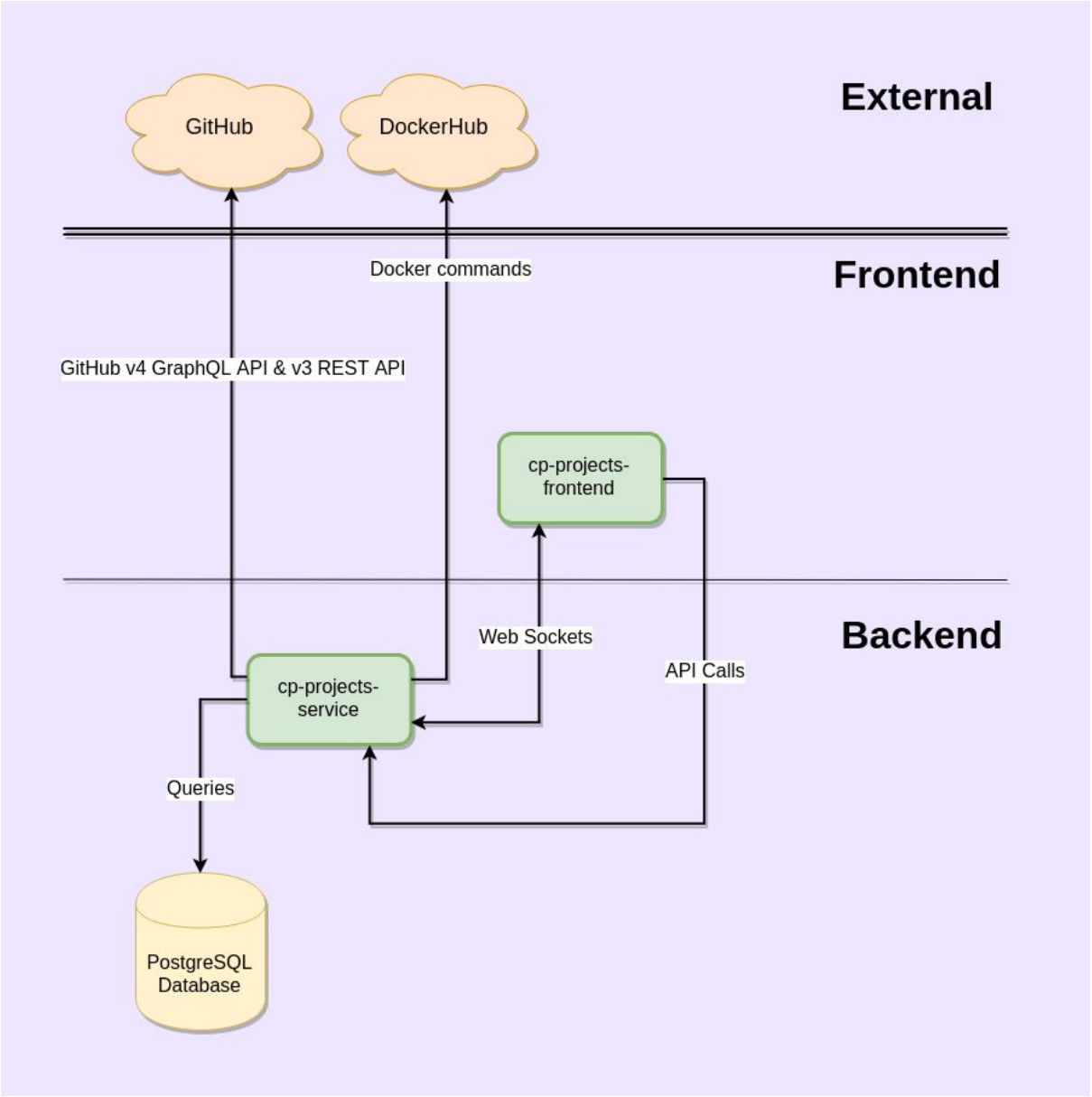
One of the requirements for my project states that it must integrate an existing version control system for storage and management of source code uploaded by users. This was another aspect of the project that I anticipated being a major challenge. I originally looked into using GitLab for this purpose but I discovered that their API is very restrictive in terms of what a general user can actually do. Since I couldn't use GitLab, I looked into using GitHub and found their version 4 GraphQL API. I was not at all familiar with GraphQL as a technology but the API seemed like it was well suited to what I wanted to do and was a new API they had just made public not long ago so it would likely be around for a long time. I decided this would be what I would use.

Design

High Level Design

System Architecture

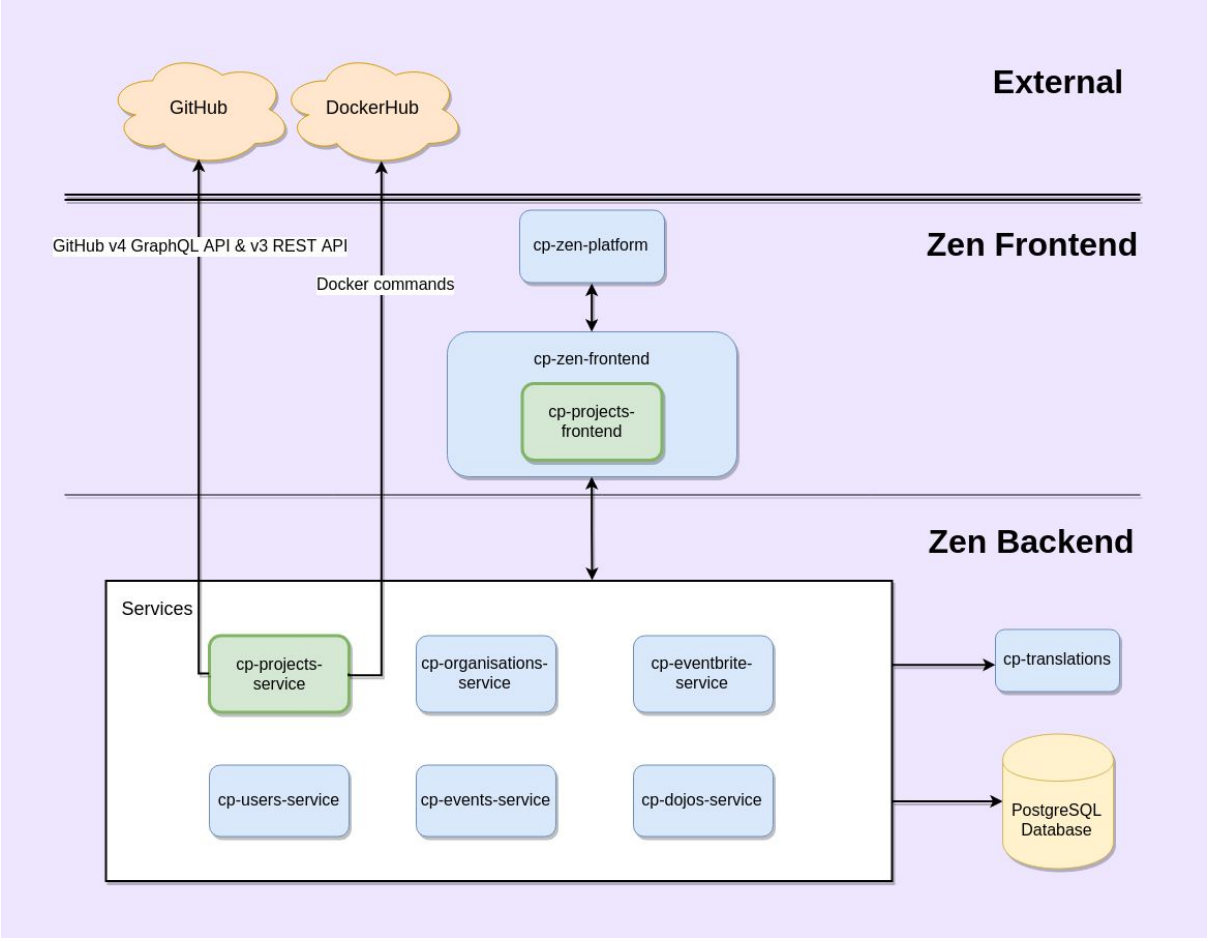
This System Architecture diagram describes the high level components of my system and how they interact without going into too much detail.



GitHub is an external entity that my system interacts with in order to store and manage user's project source code and this is achieved through the GitHub APIs. DockerHub is another external entity that my system interacts with in order to pull images for runtime containers which is done through docker commands. The frontend of my system is made up of the cp-projects-frontend entity which interacts with the backend through API endpoints defined in cp-projects-service and through websocket events. The backend of my system is made up of the cp-projects-service entity and the PostgreSQL database. The cp-projects-service entity interacts with GitHub through the GitHub APIs, DockerHub through docker commands, the frontend through websocket events and the PostgreSQL database through SQL queries.

Integrated System Architecture

This diagram has changed in a few ways since the Functional Specification since I now have a better understanding of how everything works. It describes where my system will fit into the existing Zen system when it is integrated in the future.

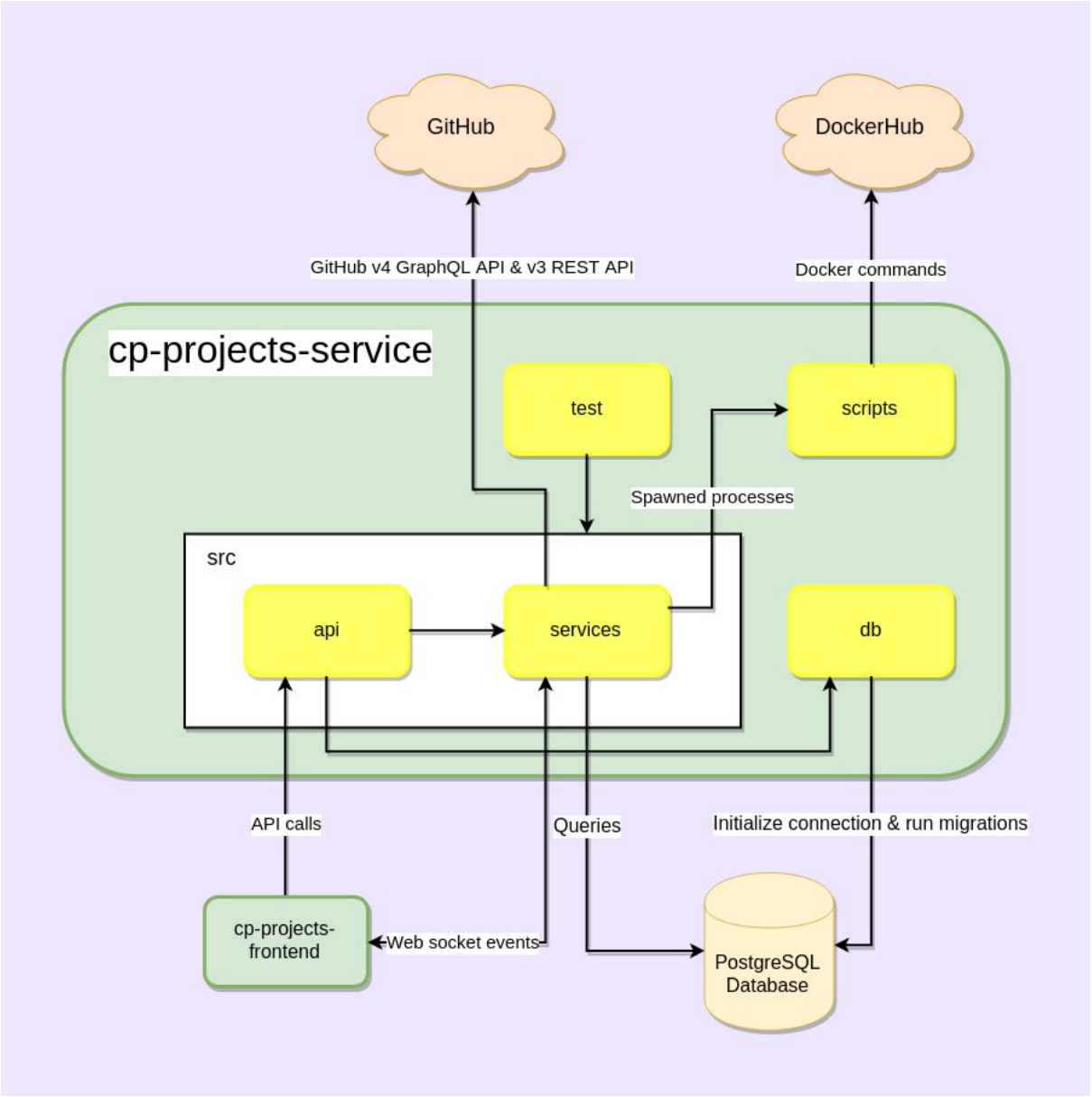


The green boxes represent my system backend and frontend and the blue boxes represent existing entities within Zen. For now the diagram gives a good perspective of the bigger picture of how my project will interact with Zen going forward.

Detailed Design

Backend

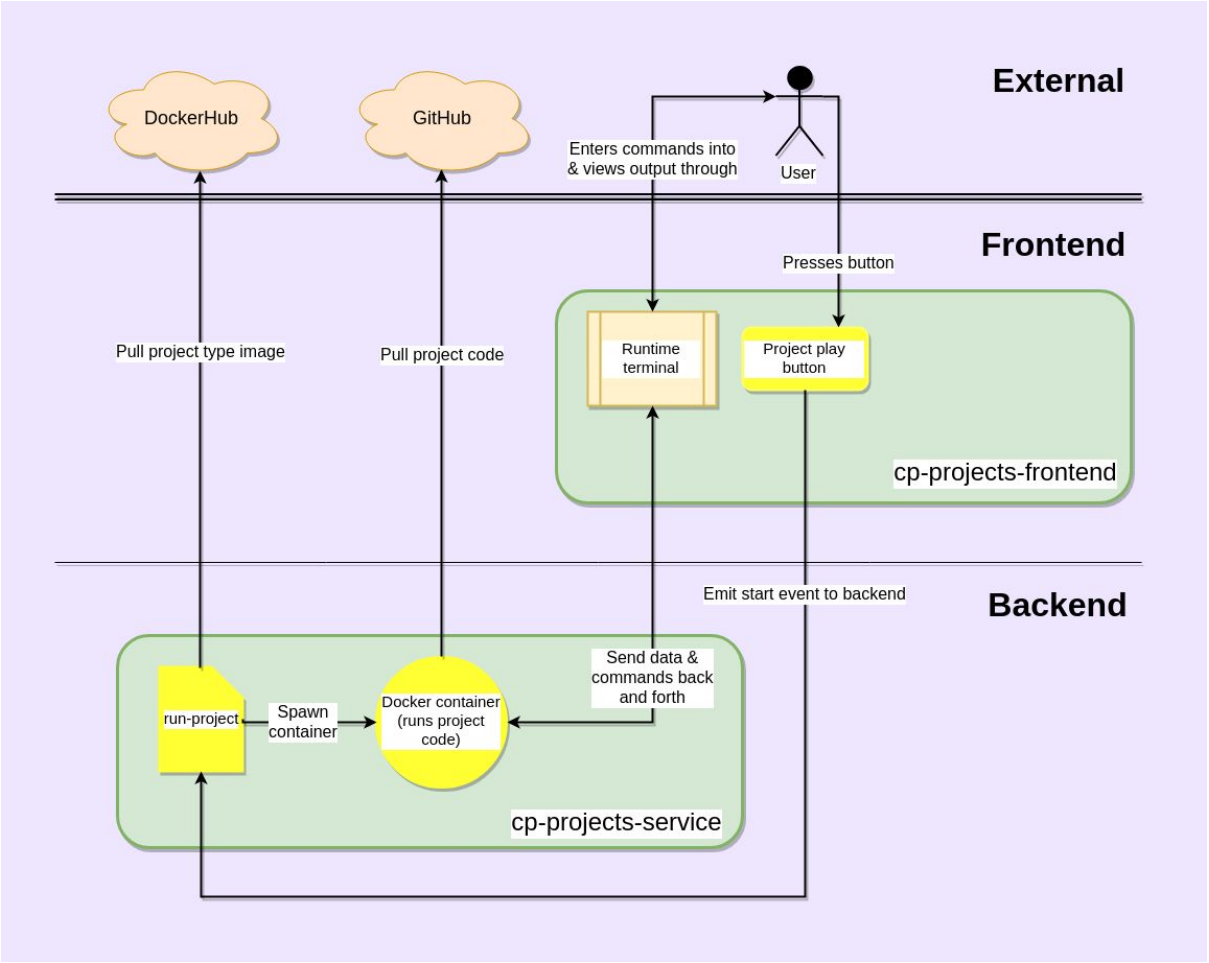
This diagram gives a more detailed view of the components that exist within the backend of my system. It shows the relationships between the components and how they interact with other entities in more detail.



The api entity is used to expose the API endpoints which can be used by other entities such as cp-projects-frontend to interact with cp-projects-service. The services entity contains several useful services which each define their own functions to be used elsewhere. These services are used to make calls to the GitHub APIs, set up and handle websockets, spawn processes to run projects and send queries to the database. The db entity is responsible for establishing connections to the database and defining and running migrations on it. Migrations are used to set up the database tables. The scripts entity is used to spawn Docker containers to run projects. The test entity is used to run unit and integration tests for the backend.

Runtime Process

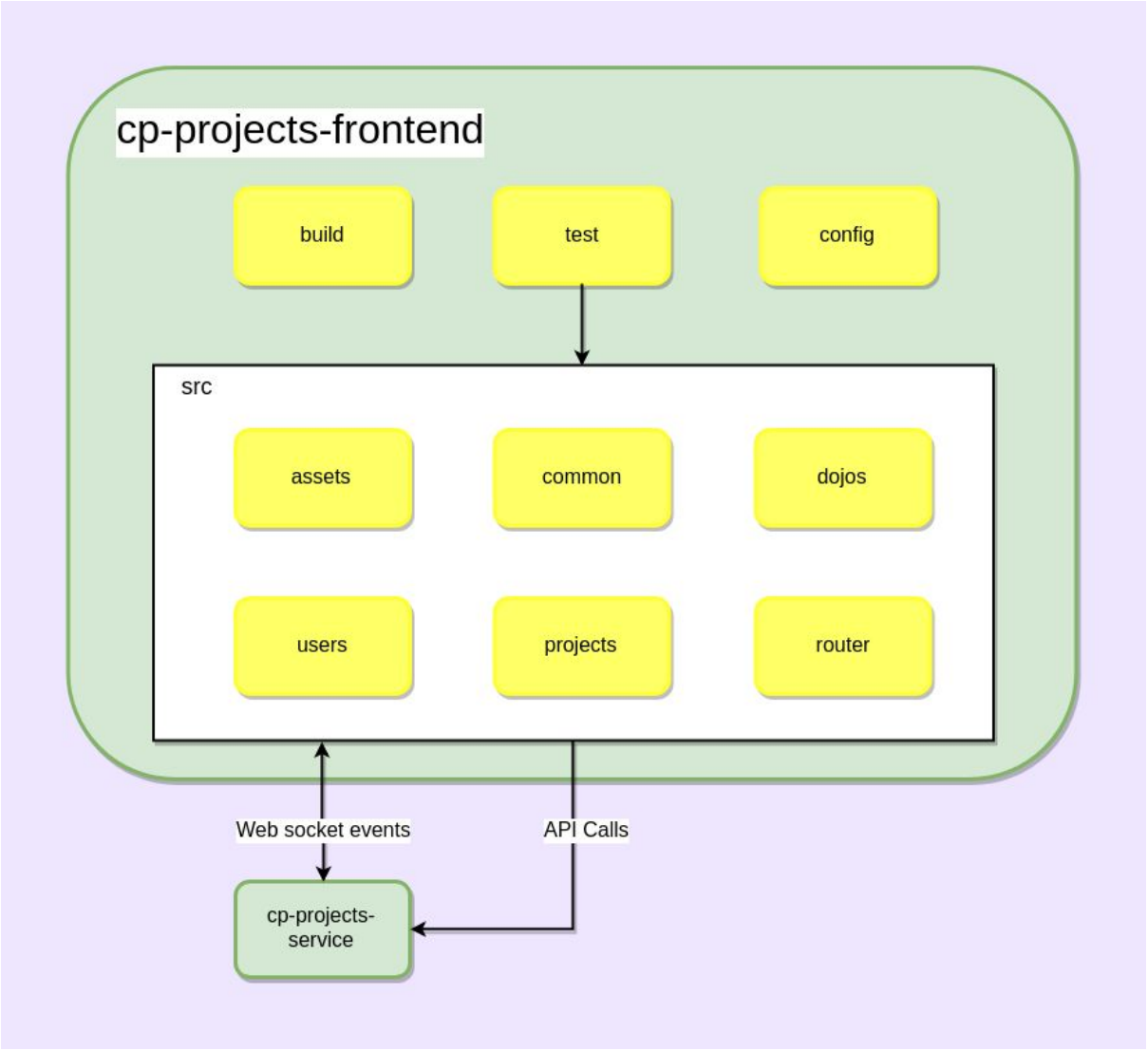
This diagram shows the process used to run projects uploaded to my system in detail.



The user starts the process by pressing the project play button on the Project Details page. This emits a start event to the backend which spawns a process to run the run-project script. The run-project script pulls the image for the project type being run from DockerHub if the image has not already been pulled and then spins up a Docker container of that image. It passes the GitHub URL and main filename of the project to the container so that it can pull the project code from GitHub and run the project main file to start running the project. The Docker container will also attempt to install dependencies for Python 3 or NodeJS projects is a requirements.txt or package.json file have been specified for them. The container then sends output from the project to the frontend terminal and executes any commands it receives back. This is done using websockets. The Docker container process is destroyed when the user leaves the Project Runtime page.

Frontend

This diagram gives a more detailed view of the components that exist within the frontend of my system. It shows the relationships between the components and how they interact with each other and the backend.



The assets entity is used to store images used in my user interface. The common, dojos, users and projects entities are used to serve components of the user interface to be loaded into the current webpage and, other than common, they all make API calls to the backend. The projects entity also uses websocket events to interact with the backend when running projects. The router entity is used to define routes and handle which components are served based on the route in the browser. The build and config entities are used to define configurations for Webpack and VueJS. The test entity is used to run unit tests for the frontend.

Implementation

All of the external libraries/packages/dependencies I have used in implementation which I have not specifically acknowledged on the “Acknowledgements” page have a licence which allows them to be freely used without mention. My implementation is designed to be entirely open source since Zen is entirely open source and uses an MIT licence.

cp-projects-service

I implemented cp-projects-service using NodeJS, SQL and shell scripts. My code is built using babel since it uses JavaScript ES6 syntax for import statements rather than using the older require syntax. The API endpoints are setup and exposed using an express server. I defined GET requests for endpoints designed to retrieve data and POST requests for endpoints designed to send data. I also setup various websocket event listeners and handlers to send and receive data using SocketIO.

In addition to initializing database connections and running migrations, the db folder contains a NodeJS file which can be used to load test data into a local PostgreSQL database which is useful for developing and for running integration tests. My unit and integration tests are run using mocha and use istanbul as a coverage reporter through nyc.

The run-project script in the scripts folder is used to pull Docker images and spin up project containers when running projects and this is done in a single line. The other folders inside the scripts folder are there so that project runtime images can be updated and rebuilt in future since they contain the Dockerfiles and docker-entrypoint scripts for each of the project runtime images on DockerHub. Having the Dockerfile and docker-entrypoint files for these images available here also helps future developers contributing to Zen who can use them as a reference point when developing new runtime images.

cp-projects-frontend

I implemented cp-projects-frontend using the VueJS framework and Webpack as a module builder. Similarly to cp-projects-service, I used ES6 syntax across the entirety of cp-projects-frontend which Webpack is able to build using babel. The build and config folders contain configuration files for Webpack and can set environment variables passed to the system based on the mode it is being built in (development, testing or production).

VueJS is a component-based framework with components being defined in files with a .vue extension which define HTML, JS and CSS for that particular component in a single file. This is used to make the application more modular. Instead of using CSS I decided to use LESS for my styling rules which allows for more hierarchical structures in defining class names and allows imports of other stylesheets and variables. Vue router is used to define routes in src/router/index.js and handle which components are injected into the page based on the current route which allows for dynamic changing of components without a page reload.

My components are divided into four folders within the src folder; common, dojos, users and projects. These each contain all components related to that particular part of the system and a service file which is responsible for making API calls to cp-projects-service for functionality related to that particular part of the system. App.vue contains the overall styling rules which are common to the entire application and the base template which other components are injected into.

Unit tests in cp-projects-frontend are run using karma and mocha with headless Google Chrome. Unit tests are carried out in files organised per component which use vue-unit-helper to access and modify data and call functions and lifecycle methods in the components within the test environment. The entire frontend can be built for production using npm run build with the appropriate environment variables. Webpack builds, optimizes and bundles all of the components, assets and modules into a single index.html file and a folder called static which can then be hosted over http:// to view.

Sample Code

Running Projects

In order to run a project a user has uploaded to my system, the following steps take place.

1. A user triggers the event to run a project through the Project Page and the **runProject()** function is called on the Project Runtime page which they are redirected to. This function emits a **start event** using websockets to the backend with the id of the project being run. This tells the backend to start up a Docker container and run that project in it. This function also **sets up a terminal** on the page which is used to take input from the user and display output from the project. Finally, this function sets up an event handler for the terminal which emits a **command event** whenever text is entered into the terminal to send the text to the backend to execute in the project container.

```
// tells the backend to spawn project container and sets up terminal
runProject() {
  // tell the backend to spawn a container for this project
  this.$socket.emit('start', this.projectData.project_id);
  // instansiate the terminal
  this.term = new Terminal({
    cursorBlink: true,
    cols: 120,
    rows: 35
  });
  // open the terminal
  this.term.open(this.$refs.terminal, true);
  // a project is now running
  this.projectRunning = true;
  // when a command is entered
  this.term.on('data', (data) => {
    // send it to the server
    this.$socket.emit('command', data);
  });
},
```

2. The backend receives the start event which was emitted and **retrieves the data for the project id** from the database. It then **spawns a process** to run the run-project shell script which **spins up a Docker container** with the appropriate image for the type of project being

run. The Docker container is given the url where the project code is stored on GitHub and the name of the main file for that project so that it knows where to pull the project code from and what file to execute first when running the project. Several event handlers are then registered to handle **sending data outputted** by the running container to the frontend and **executing commands received** from the frontend in the running container as well as **killing the process** when the frontend has emitted a **stop event** or on **socket disconnect**.

```
// when a start event is emitted by the frontend
socket.on('start', async (projectId) => {
  // get data for this project id
  const projectData = (await dbService.query({
    text: 'SELECT * FROM projects WHERE project_id=$1;',
    values: [projectId],
  })).rows[0];
  // used to signify the first output from the project process
  let firstOutput = true;
  // used to store the tag of the image to pull
  let imageTag = '';
  // set the image tag to use based on the project type
  switch (projectData.type) {
    case 'python':
      imageTag = 'python3';
      break;
    case 'javascript':
      imageTag = 'nodejs';
      break;
    case 'java':
      imageTag = 'java';
  }
  // spawn a process to run the project
  const projectProcess = pty.spawn('./scripts/run-project',
[projectData.github_url, projectData.entrypoint, imageTag]);
  // when anything is outputted by the process
  projectProcess.on('data', (data) => {
    // emit special event for first output from project process
    if (firstOutput) {
      socket.emit('firstOutput');
      firstOutput = false;
    }
    // emit it to the client to display
    socket.emit('output', data);
  });
  // when a command is received from the frontend
  socket.on('command', (data) => {
    // execute it in the running container
    projectProcess.write(data);
  });
  // when a stop event is emitted by the frontend
  socket.on('stop', () => {
    // kill the running process
    projectProcess.destroy();
```

```
  });
  // when the connection is lost
  socket.on('disconnect', () => {
    // kill the running process
    projectProcess.destroy();
  });
});
```

3. The Project Runtime page receives the **output events** emitted by the backend and **displays the text** in the terminal on that page so the user can see it. There is a **special event** received here called **firstOutput** which represents the first output of data from the Docker container and is used to know when to stop displaying the loading message to the user since the project now has some data to display.

```
sockets: {
  // when the first output has been received from the server
  firstOutput() {
    // we have now received output
    this.outputReceived = true;
  },
  // when an output event is emitted by the server
  output(data) {
    // if the terminal exists and a project is running
    if (this.term && this.projectRunning) {
      // write the data to the terminal
      this.term.write(data);
    }
  },
},
```

4. The project continues to run and the frontend and backend **send data back and forth** until the frontend emits a **stop event** when the Project Runtime component is destroyed or until the **socket connection is lost** if the browser window is closed, navigates away from my system completely or unexpectedly stops for any other reason.

```
destroyed() {
  // tell the server to stop the running container
  this.$socket.emit('stop');
},
```

Pushing Project Files to GitHub

The **pushTreeToRepo()** function makes use of five different endpoints in the **GitHub Data API** to commit and push multiple files to a GitHub repository on a specific branch and merge the new commit into the existing commits. This allows me to **store code uploaded to my system** in GitHub repositories programmatically whether they are **new or existing repositories** (so updating of project code is done through this function too since file data is merged into existing files in the repository).

The process involves first **creating a blob** for each of the files representing the content of that file in a binary format. The SHAs of the blobs (hashes which refer to the blobs) are then **added to a tree** array along with the path for the original file that their blob refers to. After the tree is posted to the repository, the **current status of the branch** that will be pushed to is retrieved which allows for the SHA of the most recent commit in that branch to be obtained. The SHA of the tree that was created is then used as the tree reference to **create a new commit** with the parent commit SHA being the SHA of the most recent commit in that branch which was just obtained. The data for the new commit is posted to the repository to create the commit and the SHA of that new commit is then used as the new HEAD for the branch, effectively **merging the commit** into the branch.

```
// pushes a tree of files to a repo
async function pushTreeToRepo(treeData) {
  await setupApiWithAccess(treeData.dojoId);

  // get the owner of the repo
  const owner = GITHUB_REST_API.defaults.headers['User-Agent'];

  // endpoints to hit
  const apiBlobEndpoint = `/repos/${owner}/${treeData.repo}/git/blobs`;
  const apiTreeEndpoint = `/repos/${owner}/${treeData.repo}/git/trees`;
  const apiBranchEndpoint =
`/repos/${owner}/${treeData.repo}/branches/${treeData.branch}`;
  const apiCommitEndpoint = `/repos/${owner}/${treeData.repo}/git/commits`;
  const apiMergeEndpoint = `/repos/${owner}/${treeData.repo}/merges`;

  // create blobs and tree array to send to GitHub API
  let tree = [];
  for (let i = 0; i < treeData.files.length; i++) {
    const file = treeData.files[i];
    const blob = await GITHUB_REST_API.post(apiBlobEndpoint, {
      content: file.content,
      encoding: 'base64',
    });
    tree.push({
      sha: blob.data.sha,
      path: file.path,
      mode: '100644',
      type: 'blob',
    });
  }

  // data to POST to tree endpoint
  const apiTreeData = {
    tree: tree,
  };

  // create the tree
  const treeResponse = await GITHUB_REST_API.post(apiTreeEndpoint, apiTreeData);
```

```
  // get current status of branch
  const branchResponse = await GITHUB_REST_API.get(apiBranchEndpoint);

  // data to POST to commit endpoint
  const apiCommitData = {
    message: 'Update',
    tree: treeResponse.data.sha,
    parents: [branchResponse.data.commit.sha],
  };

  // commit the tree
  const commitResponse = await GITHUB_REST_API.post(apiCommitEndpoint,
apiCommitData);

  // data to POST to merge endpoint
  const apiMergeData = {
    base: treeData.branch,
    head: commitResponse.data.sha,
    commit_message: 'Merge update',
  };

  // merge the tree into the branch
  return GITHUB_REST_API.post(apiMergeEndpoint, apiMergeData);
}
```

Problems Solved

Standalone vs. Integrated

During my initial planning for the project I planned to integrate my system into Zen during the course of CA400. I thought I would first build a standalone prototype of my system as a proof of concept and then integrate it into Zen to complete the project towards the end.

However, I decided after starting to develop my project that this simply would not work for CA400 since all of our project code must be stored in a central repository on GitLab in order to keep track of our changes. If I was to integrate my system into Zen during CA400 I would have to make changes to the other microservices and the frontend that exist in Zen which are on GitHub and these changes would not be tracked. For this reason I instead decided to develop my project as a standalone system with existing aspects of Zen that it will interact with in future (Zen login system, Dojo Details page, Profile page etc.) being mocked for now by pages I created myself. I think this approach has worked very well for CA400. When integrating my system into Zen in future I would see cp-projects-service running alongside the other microservices and cp-projects-frontend becoming part of the existing Zen frontend repository (cp-zen-frontend).

Runtime Efficiency

When I first implemented my project runtimes using Docker they were not as efficient as I would have liked. This was due to the process I was using to run them. Every time a project was run a new Dockerfile was created specifying the image to use. The image was then built from the specification in the Dockerfile and once it was built, a container was spawned from that image. Once the project was finished running the image was removed and would have to be created and built from scratch again for the next project. This was highly inefficient.

I decided I would improve the efficiency of the runtimes if I pre-built the images for each project type (Python 3, NodeJS and Java) since these would not change per project, only per project type. Instead of creating a Dockerfile and building an image from it every time, if the image was pre-built it could just be used to spin up a container straight away which would save a lot of time. The only issue then was how to access the pre-built image, where would it be stored? The answer to this was DockerHub which allows repositories of Docker images to be created which can then be pulled from. Since CoderDojo already have an organisation on DockerHub I was able to ask for access to create the repository under that organisation. I received access and was then able to create a repository for my images. Now the runtimes are much more efficient since they simply pull the image for the requested project type from DockerHub and spin up a container from it. The images no longer need to be specified or built. In addition, the images for each project type only need to be pulled once since they are stored on the server so subsequent plays of projects do not require the image to be pulled again.

GitHub Integration

When I initially conducted research into which version control system I would use to store the code for projects uploaded to my system, I chose GitHub and their new v4 GraphQL API because it seemed future-proof and more suitable than GitLab's very restrictive API.

Unfortunately when it came time to develop the GitHub integration for my system I realised that a lot of endpoints which exist in GitHub's v3 REST API did not yet have equivalent endpoints in the newer v4 GraphQL API. For this reason, I ended up having to use the v3 API for any GitHub functionality which the v4 API does not yet support. In the end I have used a mixture of the two which I hope will allow my system to adapt to the newer API more easily once more endpoints have been added to it since it already uses part of it.

Pushing Unzipped Source Code

When I originally tried to push code to GitHub with the GitHub v3 REST API I found that it could only accept one file per commit which did not suit my needs since I wanted to potentially push multiple source code files which a user uploaded at the same time in a single commit. I was still able to upload the source code to GitHub as a zip archive and then unzip the files when running the projects but this was not ideal since the code itself could not be viewed on GitHub and neither could the changes to individual files. In addition, the extra

step of unzipping the source code before running made the runtimes slower than they needed to be.

After some further research into this I discovered the GitHub Data API which is suitable for this purpose. It is a lower level API which deals directly with blobs, trees, commits, branches, merges etc. It was certainly a learning curve for me to familiarize myself with using this API since it operates at such a low level of Git but after some time I was able to use this API effectively for what I wanted. I now commit, push and merge all source code for projects uploaded to my system directly to their repository so that the source code can be viewed there and does not need to be unzipped when pulled anymore.

Incorrect Runtime Output

There was a bug with my runtimes for a while which I discovered when I was manually testing them. If I ran a project which waited for user input and then left that page to run another project and tried to interact with the terminal for the second project, the first project would accept my input and I would see output from the first project in the terminal of the second project. After some time spent debugging I discovered that this was because the process for the first project was still running even though I had left the page and was running a different project completely. This was only the case with projects that waited for user input.

In order to fix this I realised that I needed to kill the project process if the user left the page which I was then able to implement. However, this only worked if the user left the page to visit another page within my system since it was handled by the VueJS destroyed() hook. If the user simply closed the tab or navigated to a completely different website the process would not terminate. To fix this I decided I would need a handler for if the websocket connection was lost which would happen whenever the user left the page, regardless of if it was through closing the browser or changing website or anything else. I implemented this to kill the process when the websocket connection is lost and it fixed the issue completely.

Injection Vulnerabilities

During development I became aware of potential injection vulnerabilities in my system from either SQL injection attacks or Bash injection attacks.

The SQL injection attacks could potentially be performed on my system since I was using plaintext queries with String interpolation for variables in the Strings. If someone was to send my system a value with escape characters they could potentially execute SQL queries on my database, effectively giving them complete control over the data. In order to prevent this, I decided I would use parameterized queries rather than String interpolation. Parameterized queries allow for variables to be passed to queries in a safe way since all of the variables are sanitized before being passed to ensure they cannot include escape characters. After implementing parameterized queries across my entire system, I am no longer vulnerable to SQL injection attacks.

Bash injection attacks could potentially be performed on my system through the “Main filename” field specified by the user for any given project. This field allows the user to give a filename as the main file for their project which will be the first file executed when running their project. This is necessary for projects to be able to run since we need to know which file to execute first if there are multiple files in the project. If the user was to include escape characters in their “Main filename” then they could potentially break out of my docker-entrypoint script and execute their own bash commands since this value is used in that script. Although this is a potential risk, the damage they could do is minimized by the fact that they are still only executing commands within the Docker container for a project and not on the server itself!

Regardless, I decided it would be best to defend against this so that it would not even be possible to carry out this attack in the first place. I was able to defend against these attacks by enforcing regex for the “Main filename” value on both the frontend and, more importantly, the backend of my system so that it is effectively sanitized as it can only contain certain characters which must be in the form of a filename. Since this regex is enforced on the backend, it is no longer possible to carry out a bash injection attack on my system through the “Main filename” value.

Gathering User Feedback

For my first user evaluation I gathered feedback from the users through sheets of questions which were asked in relation to user interface mockups I had created and shown to them in person on the day of the evaluation. This worked very well for this early stage of the project but later on in the project I wanted to gather even more feedback from a wider variety of users on the project as it was at the time. Since I had a system in place at that stage I wanted the users to be able to interact with it directly and give me direct feedback on their interactions but I had to find a suitable way to make this happen.

After considering various options I decided that the best way to carry out the second user evaluation would be online since it was likely to involve a lot more participants of different age and walks of life who would be very difficult to gather in one place for a feedback session like the first one I carried out. With this idea in mind, I decided that a Google Form would be the best way to gather their input since it was easy to create, share and gather resulting data from. The only thing left to decide then was how the users would interact with my system online. The only solution here was to host my system in such a way that it would be publically accessible for them. Up until this point I had only run my project locally on my own machine for development so this would be a challenge for me to set up myself.

I spent a few days working on building and hosting my project in a small-scale production environment and was successfully able to host it at http://ca400.grahambartley.com (a domain which I own) so that my participants could access it there. I was then able to load my test data onto the system for the participants to use and it was all ready to go.

Results

I have included documents detailing the full outcomes and analysis of my two user evaluations and my code coverage in greater detail in the testing folder of my repository. I will discuss the main outcomes of my testing here.

First User Evaluation

I carried out my first user evaluation on Saturday the 10th of March 2018 in CoderDojo DCU. This session was a great way to get some early feedback on my user interface designs from one of the most significant user groups for my system - CoderDojo youths. The session was a success and resulted in some very useful feedback in relation to how I could improve my design mockups going forward into implementation.

The main feedback I got from the participants was as follows:

- **Project Creation Form:**
  - Happy with page overall
  - Some demand for Scratch support and Lua support (though Lua is not a popular language in Dojos)
- **Project Details Page:**
  - Could use brighter colouring
  - Request for play counter
  - Request for link to project author
  - Request for project comments
- **Project Runtime Page:**
  - Could be more visually appealing, more colours
- **Project List:**
  - Requests for genres and favourites

I was able to address most of this feedback going forward into development and improve on the issues raised by the participants which I think resulted in a much more usable system for those who are most likely to be using it.

Second User Evaluation

I carried out my second user evaluation from the 27th to the 30th of April 2018 via a Google Form which I sent out to participants with varying ages and technical knowledge. The participants were able to interact with my system through their web browser since I had made it available publically and were asked in the Google Form to carry out several tasks and report their feedback on these tasks.

By the 30th of April I had 18 responses to my Google Form which consisted of some really useful feedback mainly relating to navigation and usability issues in my user interface. I also

had a few participants report a bug when attempting to upload project files using Windows 10 which I was not aware of and was then able to fix shortly after!

I am happy to say that I was able to address a majority of the issues that participants were having with my system in the weeks following the evaluation which greatly enhanced my system overall, particularly in relation to navigation.

Code Coverage

I used test-driven development methodologies wherever possible throughout the development of my system to ensure a high level of coverage was achieved and constantly maintained as I added new features rather than just coding first and testing everything at the end. I also setup GitLab CI pipelines to run all of my unit tests in containerized environments whenever I pushed new code to ensure that all of the tests passed before merging any new code into master as an added precaution.

That being said, there were learning curves involved which sometimes slowed my progress when it came to testing such as trying to figure out how best to test my API endpoints, especially those which interact with GitHub since they would leave behind artifacts on GitHub after the tests were finished. I decided in the end to write integration tests for my endpoints which would interact with the rest of my system and my database directly to ensure everything worked together as well as in isolation. To overcome the issue with artifacts from tests being left behind on GitHub after tests were run I mocked the GitHub API calls in the integration tests for endpoints which use them so that nothing is actually sent to GitHub during the tests.

In the end I kept a very high level of coverage throughout the project and currently have 92.92% statement coverage on the backend from both unit and integration tests and 97.68% statement coverage on the frontend from unit tests.

Future Work

Integration into Zen

It is my ambition to integrate my system into the existing Zen stack after CA400 is complete so that it can be used by CoderDojo community members all over the world. I do not foresee many issues with integration since I have been careful to develop my system in such a way that it will easily fit into the existing stack since they use the same technologies and since Zen is built using a microservices architecture, making it easier to extend without causing harm to other areas of the system.

New Features & Improvements

Within the time given for CA400 I had to prioritise features for my system based on their value to the potential end users of the system which was a continuous process since new

features and improvements are always possible to add to software systems and I had requests for various things over the course of the project. Prioritising certain features over others was not an easy task but I used an Agile XP approach and a project task tracker to try to ensure features were prioritized based on value to users.

Going forward there are many new features and improvements I would like to see added to my system which were not prioritised during the time given for CA400. Some features and improvements I have identified are as follows:

- Images for projects
- Project report button
- Better test coverage of websocket functionality
- Comments for projects
- Filterable project lists
- Support for more project types
- Digital badges
- Custom commit messages when updating projects
- Running specific project versions

I could potentially implement many of these features myself going forward since after integration with Zen my system will be entirely open source and will accept contributions.