Using Containers to Isolate Remote Code Execution for an Online Development Environment

Joseph Fazzino
24026478

Supervisor: Dr. Hong Wei

29th April 2019

Contents

0.1 Abstract

250 - 300 words

outline aims, methods, implementation, achievements, and conclusions

0.2 Acknowledgements

I'd like to acknowledge Dr. Hong Wei for being my project supervisor. Dan Justin and Dr. Martine Magnan for their continued support through the early stages of my career. Suhail Parmar for contributing to my caffeine levels and providing help with Docker and UNIX. Dan Davis and Max Denning for helping me brainstorm the idea for the project and enduring me talking about frontend for the past year. And finally, my parents, Paul and Joanna Fazzino for being unwaveringly supportive and great role models.

0.3 Glossary of Terms and Abbreviations

The following is a list of abbreviations that are commonly used in this document:

API - Application Platform Interface

CRA - Create React App

RTC - Real-Time Communication

REPL - Read-Evaluate-Print-Loop

REST - Representational State Transfer

HCI - Human Computer Interaction

HTML - HyperText Markup Language

UX - User Experience

DX - Developer Experience

UI - User Interface

JS - JavaScript

JSON - JavaScript Object Notation

OS - Operating System

SPA - Single Page Application

P2P - Peer to Peer

PaaS - Platform as a Service

VM - Virtual Machine

VPN - Virtual Private Network

LXC - Linux Containers

SSR - Server Side Rendering

IDE - Integrated Developer Environment

CLI - Command Line Interface

STDIN - Standard Input

STDOUT - Standard Output

STDERR - Standard Error

TTY - Teletype

WS - WebSocket

CSS - Cascading StyleSheets

DOM - Document Object Model

IE - Internet Explorer

SEO - Search Engine Optimisation

LAN - Local Area Network

CPU - Central Processing Unit

PHP - Programmers Hate PHP

npm - Node Package Manager

Chapter 1

Problem Articulation & Technical Specification

1.1 Context

As computers have become more pervasive, coding has become a skill that has graduated past being something that only people who work in laboratories need to concern themselves with, to a skill that has become highly desirable commercially and is starting to be taught in the regular curriculum to children studying at a primary level education [1]. This creates new demand for beginner friendly coding tools which lends itself nicely to the promise of an online based environment where people can get started with basic coding concepts without having to trawl through documentation and technical detail about how to get running with one of the popular languages/tools available. This has led to an explosion of popularity for web applications such as `codecademy.com` which offer pre-made, executable exercises for a number of languages. A similar platform `repl.it` offers a more open and free-form experience and attempts to recreate the environment a developer may have on their machine through the web browser along with online compilation.

1.2 Problem Statement

A common pattern with the current platforms that exist is that they provide a strict sandbox within the confines of a predetermined configuration that the user selects, for example, in `codecademy` and `repl.it` you're stuck in the environment you pick when you start desired tool. An argument can be made that this makes a new developers life easier as they don't have to consider the more nuanced parts of the file system or learn any sort of terminal commands. However, it seems as though there would be value in a system that can provide both the ease of use that current existing solutions offer and also the freedom to explore a full environment with an array of tools preconfigured that encourage exploration without compromising the security and integrity of the underlying system.

1.3 Technical Specification

Based on the problem statement the potential scope for the project is very broad, there are companies and teams of developers that have the sole goal of making sure their online environments are providing users with as smooth an experience as they would expect if they had installed the tools locally.

This project will focus on the essential functionality required to behave as an online development environment while supporting a good variety of languages and offering a space which encourages exploration into different coding concepts.

With the above in mind the enumerated objectives of this project are:

1. Create a platform where users can write/execute code
2. Give every user their own personal environment
3. Eliminate the need for locally installed tooling
4. Provide a system that encourages exploration into the world of development

1.3.1 Writing and Executing Code

As an essential requirement for the development experience, the ability to edit and execute code is crucial to satisfy the overarching objective of creating an online environment. The execution of code presents a significant technical challenge however as the only code execution that can be done remotely is on a web browser which must be able to execute HTML, CSS and JavaScript. Mobile applications developed for iOS and Android are not capable of executing code other than in the language they are written in.

Functional Requirements

- Code will be able to be typed using the platform
- Code will be able to be saved
- Code will be able to be read from the platform
- Code will be able to be executed

Non-Functional Requirements

- A good variety of languages will be supported
- The basic features of a code editor will be available (i.e. syntax highlighting)
- Code that is executing will not stall the platform

1.3.2 Personal Environments

The need for the space that the user occupies to feel personal is a vital element to local development environment and therefore must be well implemented for an online equivalent.

Functional Requirements

- A personal environment will be allocated to every user

Non-Functional Requirements

- The personal environments will be isolated from the rest of the system
- The personal environments will be isolated from each other
- The personal environments will perform well and be responsive to user input

1.3.3 Local Tooling Replacement

Tooling has been through some big changes both in web browsers and locally. Web browsers have got to the point where they are so powerful that some of the most popular desktop software is being powered by them [2]. It is important to provide tools that will help those new to development, while also offering experience in tools that are of a high quality.

Functional Requirements

- High quality tools will be available to the user
- Industry standard tools will be available to the user
- The system will eliminate the need for local tooling

Non-Functional Requirements

- Popular tools will be researched and considered before being added to the system
- Tools will be standardised across the system
- Tools will behave in a responsive manner

1.3.4 Encourage Exploration into Development

Lowering the barrier to entry through the requirements stated above will inherently make it easier to explore development but more steps can be taken in order to engage users with the system such as allowing them to create short coding exercises that can be shared with friends or on social media.

Functional Requirements

- Implement exercises for users to do
- Allow creation of exercises by users

Non-Functional Requirements

- Allow any exercise to be shared
- Assign difficulty level to exercises
- Provide an open area for the user to explore their personal environment

1.4 Stakeholders

This project has a number of relevant stakeholders with various degrees of interest in the outcomes. All of them will be considered during the construction of the system.

The Developer - Joseph Fazzino

The developer of the system is responsible for making 100% of the the technical decisions and is responsible for delivering a fully functioning system adhering to the technical specification found in Section 1.3 of this report.

Project Supervisor - Dr. Hong Wei

The supervisor of this project is overseeing the development and design process that is being undertaken.

They provide guidance when it comes to essential functionality and ways that technical requirements can be implemented.

User - Beginner Level Developer

Those new to development will not have experience with the terminology and syntax that exists in programming and wider computer science. They may have an understanding of basic coding concepts taught to them during formal education.

The beginner user should be able to use the system in order to become more familiar with generic programming concepts. The exercises available through the system will likely be the area they spend the most time.

User - Intermediate Level Developer

A user more familiar with the general work flow of a developer will be able to understand certain levels of nuance of how a system might be implemented and consider how they may solve certain problems.

This kind of user would benefit more from the ability to have a playground to explore the system in so they can understand the functionality that it provides and maybe try to explore the extent to which it works.

User - Experienced Level Developer

This user will have successfully developed systems with a high level of complexity and will most likely have specialised knowledge in a certain domain/environment.

This type of developer will be difficult to convince the benefits of an online working environment when they undoubtedly have a solution that works well for them locally. Perhaps the advantage of being able to mentor developers by making exercises for them would be appealing.

1.5 Constraints

Some constraints on the development of the project exist.

- Permanent deployment - as the system is likely to be complex, deploying it will be costly and time consuming. Test deployment will be done to experiment with configuration settings in the system and perform benchmarks but a permanent live deployment will not be.
- Computer resource availability - the system will be constrained performance wise by the resources available during development meaning that any stress tests are not representative of a deployed system
- Significant testing base - as the system will not be deployed it will be difficult to adequately test the system in the manner which it would be used by end user. A different method of testing will have to be explored.

1.6 Assumptions

A number of assumptions must be made to reasonably meet the technical requirements.

- The users will have a reliable internet connection
- The users will have the necessary software/hardware configuration in order to access the system (e.g. a modern web browser)

Chapter 2

Literature Review

This chapter examines various literature around relevant subjects to the project objectives stated in Section 1.3. It looks at the various methods of **Real-Time Communication** that exist in order to create an environment where feedback is fast and frequent *(Section 2.1)*. It examines some **existing systems** that are providing some of the features listed out and critically examines the positives and negatives of some of the technical choices that are apparent in these systems *(Section 2.2)*. It also analyses some of the modern advances in **Virtualisation** technology along with how the advent of **Containers** have changed the landscape of PaaS services and virtual environments in general *(Section 2.3)*. It concludes by looking at the state of the art in **frontend** *(Section 2.5)* and **backend** *(Section 2.6)* technologies that can be used to create a modern web application.

2.1 Real-Time Communication

Real-Time Communication is an important research topic for this project as in order to create an environment for users that feels as close to a local experience as possible, the requirement for fast feedback is essential.

An experiment was done in 2012 discussing the performance of different RTC methods by Professors at the University of New Brunswick [3]. This experiment compared the different standard HTTP methods of implementing Real-Time Communication compared to the new (at the time) technology of WebSockets which are designed to create a fully duplexed bidirectional data-flow.

HTTP polling is an attempt to solve the real-time issue by repeatedly making a request to a web server at a pre determined time interval to check to see if there are any messages waiting to be read. **HTTP long-polling** is another solution that uses the HTTP protocol but reduces the number of wasteful requests by having the server intelligently not respond to the request if there is no information available and hang until a timeout or information becomes available. Both of these solutions are inadaquate for a responsive system however because the HTTP protocol is still built on top of a system not designed for real-time, fully duplexed communication channel. HTTP relies on a standard 'Request-Response' model which is only half duplex so polling was only a solution that worked for systems that were reliably sending data at a steady rate such as sensors that are being queried for an API.

A modern solution to this is the **WebSocket** protocol proposed in RFC 6455 [4] which aims to reduce latency by a factor of 3 compared to HTTP in the real-time communication aspect. It is a fully duplexed, bidirectional communication channel that provides an efficient method of communicating between several different clients using a persistent connection between the client and the server. A client may connect to a websocket endpoint on the server, send messages to it, and the server may broadcast messages back to just that client or to every client connected. Due to this behaviour it is very popular for creating text based chat communication systems.

WebSockets work by utilising a persistent TCP connection where messages can be sent back and forth without there having to be a new connection made every time. This behaviour is possible in HTTP since HTTP 1.1 however, WebSockets do not adhere to the standard, 'Request-Response' cycle that a HTTP request utilises. Any client connected to the socket is capable of broadcasting a message at any time. HTTP persistent connections also still suffer from latency due to the effort the protocol makes to control congestion [5]. WebSockets take the concept further by making it simple to embed data with each request in the form of a string in the form of a JSON schema. This makes it ideal for the transfer of small chunks of text where only text is the required form of the response. WebSockets are not appropriate for downloading resources or assets such as images.

Another new approach of RTC on the web has been developed by Google in collaboration with other browser vendors called **WebRTC** [6]. This technology is focused on streaming audio and video between different clients on the web. This new technology is aiming to be the replacement for the browser plugins that were necessary in order to use P2P video/voice chat software such as *Skype, Facebook Messenger, Google Hangouts, etcetra* [6]. WebRTC is more appropriate for applications that need a streaming based connection as it's latency is even lower than WebSockets due to it utilising the UDP protocol which has much less overhead compared to the TCP based connection of WebSockets [7]. WebRTC would not be appropriate for the use case of WebSockets as when transferring informational data between clients, such as a chat application, it is important to make sure that the data is being received in the correct order whereas UDP is less concerned so long as enough packets get transferred to create a stable audio/video connection.

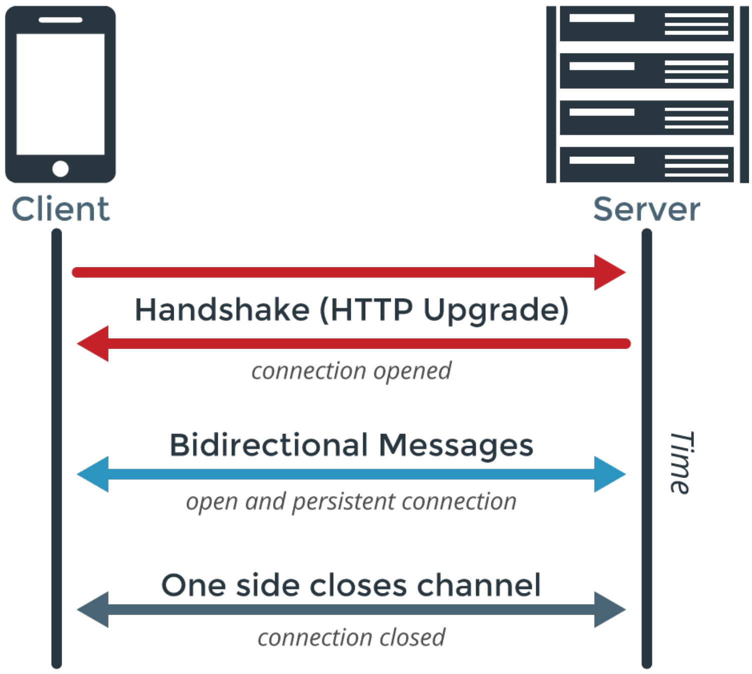

Figure 2.1: Illustration of WebSocket connection [8]

2.2 Online Developer Environments

A number of existing solutions providing online development environments exist and have been analysed for the purpose of this review.

2.2.1 Repl.it

Repl.it is very similar to the idea proposed in the Problem Statement (Section 1.2) and a lot of the requirements lined out in Section 1.3. It offers a huge array of Repl templates available for users to get started with many languages/frameworks very quickly. It also uses the Monaco Editor provided by Microsoft in order to provide a first class text editor experience.

Repl.it takes advantage of containers in order to gives users a full developer experience when visiting the system [9]. The system also uses it's own container orchestration software in order to scale the instances available to users up and down depending on demand and predicted demand.

Every code result that is available to be viewed/run is viewable through a special .repl.run subdomain. This includes long running processes like web servers which are able to be hosted from these subdomains and be always accessible. This means you could create several repls which all connect to each other like a full system.

Technically the system is very impressive, something that the system doesn't recreate quite as smoothly as a local environment would is a small amount of latency between a key being pressed and the corresponding value appearing in the REPL itself.

The system also seems to remove all previously typed entries of the REPL on every press of the *Run* button. This suggests that it is giving you a new REPL instance on every execution which isn't how a local environment works.

From a HCI point of view the website feels very smooth to use and is not frustrating to use other than the latency noted when typing directly into the running container via the REPL.

Repl.it is clearly very focused on the objective of replacing local development environments and does a good job of fulfilling that need.

2.2.2 Codecademy

Codecademy is a educational focused online environment designed to teach users how to code. Ranging in topics from beginning web development to a course on the IBM Watson API. It is a more directed experience than Repl.it as users are performing tasks for exercises but they are typing code into a similar environment, the code is executed and the result is displayed to the user.

Codecademy does not allow access directly to the REPL but if code is entered into the editor which allows for user input such as the `input()` function in Python. Then it interprets the input correctly.

The Codecademy web application is clearly a very complicated system and it shows by how unresponsive it feels when navigating from page to page. The page does a full refresh even though there are elements which do not change on the screen page to page. This leads to a frustrating wait looking a blank screen between page loads.

It is clear that Codecademy is a focused environment to encourage new developers to get into development by offering an easy to start environment and heavily directed experience. It is not concerned with the idea of replacing local development environments so much as making sure that it's not something beginners should need to think of when wanted to get to know a new tool.

2.2.3 Glitch

Glitch is a web application that is focused on trying to cultivate a social coding community that encourages developers to help each other out and build mini applications with JavaScript and Node.js. It provides an online coding environment that uses containers to isolate the users runtime.

Glitch is clearly focused heavily on the social aspect as on the homepage they have a section dedicated to users asking for help so more experienced coders can help them achieve their goals with the applications they want to build. It also showcases user made projects on the homepage which can be *Remixed* which is similar to forking a repository on GitHub for other users to modify.

In terms of design, the website has a very colourful friendly interface. A feature which is particularly notable is in each project editor there is an option to view *Container Stats* where the CPU usage in %, Memory usage in bytes and additional relevant information can be found. There is also guidelines on the technical restrictions for projects that are run in Glitch.

2.3 Virtual Machines and Containers

In order to provide as close to local experience as possible to the users of the system this project aims to create. A virtual environment for executing code and saving files is vital. Virtualisation technology is changing significantly due to the different Container solutions which attempt to promote a more disposable and lightweight type of virtual environment compared to their hypervisor powered counterparts.

2.3.1 Virtual Machines

Virtualisation is a technique in computing that, most commonly, is seen by users in the **Virtual Machine** (VM software. Virtual Machines are heavily utilised to provide virtual desktop environments on top of a users already existing desktop. The advantages of which are, a sandbox environment for potentially harmful operations, such as when penetration testers are trying to fingerprint a virus. The option of trying a different OS without needing to dedicate a partition of disk space to it or deal with a dual booting set up is another user facing benefit of virtual machines.

In the enterprise world, Virtual Machines are being used to host their customers applications in a full Platform-as-a-Service (PaaS) solution so customers no longer have to worry about hosting their own web servers or other online services.

The general way of interacting with fully virtualised environments is through a hypervisor which is a tool that is responsible for provisioning and monitoring Virtual Machines [10]. The hypervisor allocates resources such as memory and CPU cores from the host machine that the VM is allowed to consume. When the VM is shut down these resources are freed and can be used by the host system once again. The hypervisor also allows the VM to use a different base operating system than the one that is on the host machine as it provides a whole *Guest OS*.

2.3.2 Containers

Containers are a much lighter virtualisation method than Virtual Machines despite the functionality being similar. They achieve this as they are much closer to the systems 'bare metal' as any commands that are executed through a container are running on the host's hardware. This means that there is no need for a hypervisor as containers have direct access to the system resources. Usage limits can be set in the configuration of container *images*.

As Containers traditionally don't utilise a hypervisor the biggest difference between them is that the engine that powers the container provisioning software such as the *Docker Engine* isn't able to virtualise an environment based on a different OS. This is more by design however as it is what gives containers their 'lightweight' quality as they aren't having to simulate a whole kernel. Not having a full kernel to set up however means that containers can start up significantly faster than a VM.

Due to their performance benefits containers have become popular options for PaaS software. A paper was written comparing the benefits of a fully virtualised environment against a container based solution [11]. It concluded that containers have an inherent advantage over VMs due to the performance benefits and the quick start up time of them. It also mentions that few PaaS vendors are using containers for their systems so far as they are too new of a technology. It is worth nothing that the report was published in 2014 however and since then uptake will have increased.

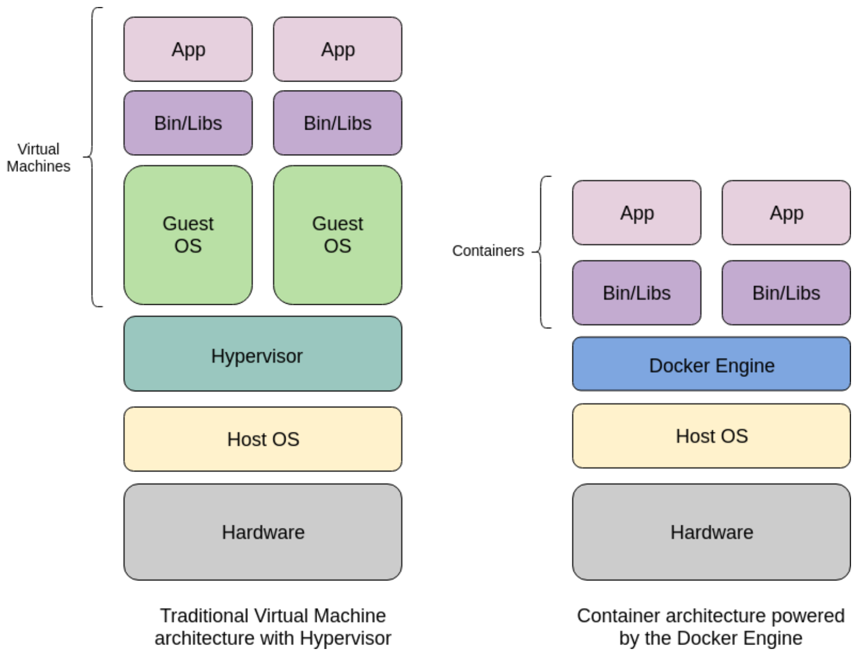

Figure 2.2: Architecture of Virtual Machines vs. Containers

2.4 Container Providers

As containers are a more modern innovation than hypervisors, recently there has been a wave of different technologies that attempt to make simple containers in varying ways.

2.4.1 Linux Containers

As briefly mentioned above, **Linux Containers** or **LXC** are the foundation of many container solutions. This is due to the fact that they offer a light kernel implementation which provides every container with some key features.

- A unique Process ID for each container
- Isolates all resources for the container by using cgroups and namespaces
- Provides each container with it's own private IP address
- Isolates all files on the container from the Host by using chroot

The features listed above are all features standard in the Linux kernel. `cgroups` [12] or control groups are a feature that is able to isolate and allocate resources from the machine to each individual process.

`namespaces` [13] is another key feature of the kernel which LXC relies on in order to provide isolation to each container. The purpose of namespaces is to wrap a process or a group of processes in an isolated instance of the global resource, changes of global resources are to other processes that are a member of a namespace but aren't visible to other namespaces.

In terms of downsides the LXC implementation is heavily tied to the Linux OS which means that it is not possible to run it on a different OS such as Windows. There are also some security concerns for LXC as all the containers share the one host kernel.

2.4.2 OpenVZ Containers

OpenVZ makes use of a modified Linux kernel with it's own set of extensions. OpenVZ is able to manage physical and virtual servers with *dynamic real-time partitioning*. It similarly offers better performance than a traditional hypervisor based system and utilises the cgroups and namespaces features of Linux to provide it's virtual environments.

On top of the advantages of LXC it also provides the following benefits.

- **Container Lifecycle** remote management can be done of containers using an API to modify the status of a container in real-time.
- **Container State** is able to create checkpoints during the container's lifecycle so that it may be recovered from that point should anything go wrong.

These mean that there is greater user access over the state of containers and that restoration points can be created to store users progress with a container.

2.4.3 Docker

The Docker process is a daemon which can provide and manage Linux Containers as *images*. It uses LXC for the container implementation and then adds on top an image management system and a *Union File System*.

Using the daemon, Docker manages to provide similar functionality as the OpenVZ containers in relation to lifecycle and state. The state of a container at any time can be saved to a new image which can then be reloaded by the daemon to that same point

Unlike OpenVZ, Docker can be run with the standard Linux kernel and therefore is more suited for PaaS software. It also has a thriving ecosystem of pre-made images which offer a huge array of different starting points and tools.

2.5 Frontend Web Technologies

The client side of web applications is based on three fundamental technologies, `HTML`, `CSS` and `JavaScript` which declare both the layout of the application and the functionality.

Web development has changed a lot since its inception, new technology has been released that gives developer greater flexibility in how they build their modern web apps.

Since the creation of jQuery [14] a number of JavaScript based frameworks/libraries have come about that try to solve some of the problems that are inherent to the web platform.

2.5.1 React

React is developed by Facebook and attempts to simplify the process of creating interactive UIs by providing a declarative way of writing UI code and encouraging the reuse of *components* which are composed HTML elements with the ability to provide interaction through JavaScript.

The need for a library such as React comes from the difficulty involved with maintaining data synchronisation between what is displayed on the screen and variables that exist in the JavaScript code. React also offers a high amount of code reuse with its component architecture.

React is able to use JavaScript functionality inline with HTML style layout syntax by providing it's own version of HTML called JSX.

React has a very strong community with over 80,000 packages listed on the npm package registry [15].

2.5.2 Vue.js

Vue.js is a JavaScript Framework that offers a lot of the same functionality as React but offers it in a way that is more akin to the traditional way that web development is done. Where React blends the 3 key technologies of the web into a JavaScript focus. Vue.js maintains a separation of these concepts.

Vue.js offers more than React out of the box such as an official routing solution for single page applications, global state management and server side rendering.

Vue uses ordinary HTML as it's view templating however it is able to inject it's own directives and JavaScript functionality by making use of the popular handlebars syntax.

It has grown very quickly since it came out in 2014 and has just over 25,000 packages published on npm [16].

2.6 Backend Technologies

There are numerous tools in use both professionally and recreationally to create web servers, databases, and other tools that a system might need to utilise.

Traditionally, the backend of an application was written in a large framework such as `PHP` or `.NET`. In more recent times however, there has been a rise of lighter scripting languages such as `Python` or `JavaScript` which are powering infrastructure for some of the biggest technology companies in the world.

2.6.1 Node.js

Node.js [17] is a JavaScript runtime built on Google Chrome's V8 engine. As JavaScript is a single threaded language many thought it was unsuited to hosting backend applications due to the lack of concurrency available to it. Node.js uses the `libuv` C library [18] in order to create an event loop which has the task of offloading external processes such as database calls and when a response come back the event loop can pass the result back to the JavaScript environment. This is how Node.js is made to be a viable option for hosting servers and running in a browser-less environment.

Node.js is in wide use in the industry with companies such as Netflix [19] which is a testament to how powerful it can be even at a huge scale like Netflix are using it. Node.js is also appropriate for use where the application is smaller in scale and perhaps only needs to interact with external processes and host some endpoints.

2.6.2 .NET Core

.NET Core [20] is a cross platform runtime developed by Microsoft which can be used to develop large scale web applications.

.NET Core is able to provide a fullstack framework relying on Model-View-Controller architecture where the Models and Controllers are written in C#, Visual Basic or F# and the views are written in Microsoft's Razor syntax which is similar in idea to React or Vue's templating/JSX but uses C# instead of JavaScript.

Being supported and created by Microsoft means that of course it's extremely useful for large scale applications that big companies will want to create. It is less well suited for smaller projects due to the MVC pattern which carries a lot of boilerplate. This can lead to writing a lot of code for simple controllers/views.

Chapter 3

The Solution Approach

3.1 Solutions for Environment Virtualisation

In order to provide users with the most 'local' experience as possible on a remote platform it is key to analyse various technologies and techniques that are currently in widespread use in the industry. As discussed during Chapter 2 there is an consensus in the industry that containers are the better solution for PaaS type software of which this project would fall into the category of.

There are many different container solutions available currently, many have similar roots such as a backbone of using LXC but they build on top of those foundations in varying ways. Some of these ways are documented in the paper which was discussed in Section 2.3 [11]. This paper performed a comparison of the various different container technologies and stacked them against each other on key implementation features such as performance and security.

The decision to not research Virtual Machines as a potential solution for the system is due to the lightweight nature of the system that is required. Container technology is also more freely accessible compared to virtual machines software which is often licensed. This means that there are few to no downsides of using a container based solution versus a fully blown Virtual Machine set up and many benefits.

3.1.1 Container Providers

The main container providers that can be applied to the project are Docker and OpenVZ as they both provide a good foundation on top of the base LXC technology. The ability to save containers at checkpoints is useful in relation to the project as any information created on the container can be saved and referred to later. While this feature is not in the proposed scope of the project it is important to consider feature development features and technology choices at an early stage can influence how easy or difficult implementing future features will be.

A big limitation of OpenVZ is that it can't run on the standard Linux kernel so it is not a very viable solution for this project as the aim to to deploy the system and requiring a modified kernel will add complexity.

3.1.2 Virtualisation Conclusion

For the system, Docker seems to be the clear choice as it provides good tooling with its daemon. A strong foundation on top of LXC, and it doesn't require additional modification before it can be used on a computer/server which is preferable to avoid for this project.

3.2 Chosen Solution for Real-Time Communication

Based off the research performed in Section 2.1 it would seem as though WebSockets fit the requirement of the project in order to ensure rapid communication between the client and their virtual development environment.

By using WebSockets it will be possible to have incredibly low latency bidirectional messages sent from the client to the server which can give the server instructions on what to do with the container that the client is allocated. The ability to send structured data chunks in string form makes it perfect for sending code and returning the output that is executed by the container.

WebSockets are available through the native Web APIs and so no 3rd party dependency is required to interact with them on the client side.

On the server side there are a few ways to implement a WebSocket endpoint but as Node.js and Express are already being used for the REST api it using a 3rd party dependency such as `express-ws` makes the most sense.

3.3 Requirements for Frontend

In order to create an experience that emulates a local installation of tooling and a text editor it is necessary to make sure that a tooling solution is chosen for this project that can meet these needs. The key requirements for the user facing side of this project are:

1. Text Editor with essential features that users expect in a standard developer environment
2. A terminal emulator that can display output of code to users and allow input where it is appropriate

Without these two features there is no way to adequately provide users with an environment that can be anywhere near the level of quality that they would expect from a local installation of tools.

3.3.1 Text Editor

A text editor is a vital part of a developers tool-chain and a few solutions exist on that can be rendered in the web browser. The reason a simple text input cant be used is that a text editor performs actions such as automatic indentation which, while could be recreated, is very specific to languages where for some indentation doesn't matter and for some it is how the language detects what code belongs to what block or function.

With this in mind it is helpful to establish a list of features that are a key requirement for any text editor.

- Automatic indentation
- Syntax colouring
- Control + F compatibility for Find
- Bracket matching
- Copy-Paste compatibility

With these features in mind it is now worth investigating the available resources to see which one satisfies the features best and if any bonus features cna be found.

Ace - *https://ace.c9.io/*

Ace is a code editor which is used by Amazon in order to provided their cloud based IDE 'Cloud9' it offers all the features that are listed above and notably includes support for themes, multiple cursors and bracket highlighting.

It's worth noting that Ace is first and foremost a code editor for online use and isn't available for users to install locally.

CodeMirror - *https://codemirror.net/*

CodeMirror is another code editor that is exclusive to the browser, it is the code editor that both Firefox, Chrome and Safari use inside their dev tools. It offers all the same features as Ace, however it has a more modern design which more accurately resembles a locally installed text editor.

CodeMirror claims to have experimental support for mobile browsers however it isn't an experience that is good enough to consider as a bonus feature for the editor.

Monaco - *https://microsoft.github.io/monaco-editor/index.html*

The Monaco text editor is made by Microsoft and used in their very popular code editor 'VSCode'. VSCode is one of the most popular text editors at the moment for a variety of different developer communities such as web developers and people starting out with a new language that don't want to have to deal with a fully blown IDE.

Feature-wise it offers all the features that are offered by CodeMirror and Ace but also comes built in with auto complete support for TypeScript, JavaScript, HTML and CSS. Through language servers as well, any language can gain support for auto complete for the standard language syntax. It also comes with a 'diff-editor' mode which can be used to gently introduce users to the idea of version control.

Monaco does not have as many themes available as the alternatives but the features that it does offer outweighs the value that alternate themes provides.

Decision on Text Editor

Based on the details above it makes a huge amount of sense to use the **Monaco Editor**, being provided by Microsoft is a significant benefit and the fact that it powers one of the most popular code editors that is being used in the industry means that it will provide as close an experience to a local coding environment as any of the other options.

Giving users experience with this tool will mean the transition to local development tools will be less abrasive as they already know the features that are available in these industry tools. The ability to provide auto complete for some languages is a huge benefit as well as new developers can be sure that the syntax they write is correct.

3.3.2 Xterm.js Terminal Emulator - https://xtermjs.org/

For online emulation of a terminal/command line the most popular solution is **Xterm.js** which has widespread adoption across many developer tools that require emulation of a terminal. It is used with VSCode in order to give users access to their local shell. It is highly performance focused in order to provide low to no latency between keystrokes and when they render on the screen. It also provides an array of addons that mean the terminal can be connected to a WebSocket stream so the online terminal can be connected to a real terminal on a machine. This is a key feature of a local development environment and therefore Xterm.js is ideal for this project.

3.4 Solution for Building the Interface

With the solutions established above it is now appropriate to evaluate the options when it comes to how to build the whole user facing side of the system.

As the developer has the most experience building websites using the `React` library [21] to help with creating reactive, data driven, single page web applications, that is the overarching technology that will be used to build the solution.

Within the React ecosystem however, there is a number of options about how to manage certain essential features such as page routing and how to style components in a way that fits into the React methodology.

3.4.1 React Framework Options

In order to get started with React the recommended way is to use a package called `create-react-app` however a limitation of this method is that it is not configurable to the extent that is required by the Monaco code editor if syntax colouring is considered a key feature, which it is.

It is necessary then to evaluate some other options for getting started with a React application that allows for the configuration options that means syntax colouring works.

The popular options in the community are: ejecting a CRA project, Next.js or Razzle.

Eject Create-React-App

Ejecting a CRA provides the developer with full access to all the configuration options that are previously abstracted away. It also installs a lot of dependencies that need to be maintained correctly and the configurable options are overwhelming when all that's required is a few lines added to a config file. This solution is undesirable.

Next.js - https://nextjs.org/

Next.js advertises itself as a React 'framework' as it provides many additional features out of the box versus traditional CRA projects. It offers functionality for:

- Routing via the File System
- Code Splitting
- Server Side Rendering
- High Level Configuration

With these features, fewer external dependencies are required and the configuration which is needed to get syntax colourisation working is available.

Razzle - https://github.com/jaredpalmer/razzle

Razzle attempts to find a middle ground between the opinionated decisions made by Next.js and the overwhelming amount of configuration that is required after ejecting an app created by CRA. It is also agnostic to the technology that you use it with so it can be used with several other different frontend libraries such as *Vue.js*.

Due to the less opinionated nature of the project the only real benefit provided by it is the server side rendering and configuration options that is also provided by Next.js.

Despite similar configuration extensibility as Next.js trying to enable syntax colouring for the Monaco editor didn't work.

3.4.2 Conclusion on Frontend Framework

As syntax colouring has been described as a key feature there isn't much of a choice beyond choosing to use either Next.js or the Ejected CRA. As Next.js has a number of other benefits however, this makes it the most attractive and powerful option for building the frontend.

3.5 Design Prototypes

Trying to build a system that provides users with an environment where they feel as though there's not a high barrier to entry involves a design process which has to ensure that things are arranged in as friendly a way as possible.

Some sketches were done in order to try to narrow down what kind of design language should be used to give the users a sense of friendless versus the more stark, professional look that some developer focused websites aim for [22]. Based on these general requirements the following low fidelity prototypes were created presenting different ways that information could be displayed for users.

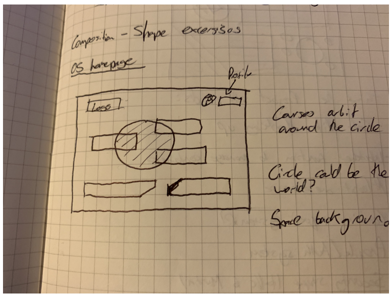

Figure 3.1: Prototype 1

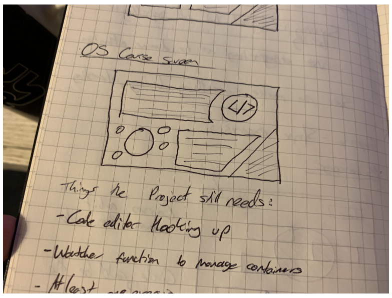

Figure 3.2: Prototype 2

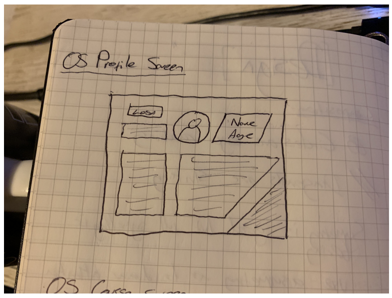

Figure 3.3: Prototype 3

3.5.1 Prototype One

Prototype design one uses small blocks of informative text which orbit around a circular shape in the middle. Not having significantly large blocks of text means the user won't be intimidated by the information being displayed to them.

The circle is a technique that designers use in order to draw the viewers attention to a specific area. Although it is not shown in the prototype image the font styles that are in use are: one serif font for the display text (the titles and headings) and one sans serif font for the body text (bulk of text).

Some navigation elements can be seen along the top representing different sections of the site.

3.5.2 Prototype Two

Design two uses circles in the same way that prototype one does in order to draw users attention however it also uses interesting borders and shape cut offs to make the website stand out against most websites which stick with the default rectangular shape of most UI elements.

This prototype contains a lot of textual information and might be more relevant for a page that needs to convey a message to the reader that requires significant blocks of text. This design wouldn't be appropriate for a landing page but maybe would for a page on an exercise the user can do.

The font styles in use are a `monospaced` font for the display text and a sans serif font for body text.

3.5.3 Prototype Three

The third design is a concept for a profile screen but could be applied to a page that represents a particular language and technology. It has lots of space for informative textual content as well as some space for image assets/other resources that might convey information in a less traditional method.

It attempts to use more shapes in order to draw attention to certain areas of the page although it does give the page a slightly skewed look on the right hand side so perhaps if this design were to be implemented it may not render on the screen in a way that looks good.

No specific font styles were decided upon for this prototype.

3.6 Overall declaration of solution chosen

Based off of the research conducted in Chapter 2 and the potential woes of implementing some of the solutions proposed in this chapter. The current solution approach for creating a full implementation of the system is to have a Next.js frontend with a backend that uses Docker containers in order to implement virtual environments for users. The frontend will make use of the Monaco code editor and the Xterm terminal emulator. Communication between the containers and the frontend will be done with the WebSockets RTC method.

A DIAGRAM WILL GO HERE

Chapter 4

Implementation

This chapter focuses on the overall implementation of the system and walks through how the several separate components interface together and interact in a way that provides the user with a positive experience.

4.1 Backend

The backend of the system is implemented in Node.js and provides the API that the frontend will interact with through REST requests and WebSocket messages. TypeScript [23] is being used rather than plain JavaScript in order to provide support for static types and catch more errors during the build time compilation rather than during run time.

4.1.1 Database

The database is a straightforward MongoDB [24] implementation with two collections, one for exercises and one for activities. An exercise can have many activities but activities can only belong to one exercise.

For the Node server to be able to make database calls the MongoDB API must be queried against, this is done by using the `mongoose` package [25].

4.1.2 REST API

To provide an API that a frontend can interact with to retrieve information from the database, a REST API is created with the Express framework [26]. Creating an API endpoint with Express is a simple process that roughly follows the formula of:

```
AppObject.RequestType("Endpoint", CallbackFunction)
```

Where *AppObject* is the variable representing the instance of the server. *RequestType* is usually one of GET or POST. *Endpoint* is a string representing the local path the handles the request and *CallbackFunction* is the function that handles the request and sends the response.

The code `app.get("/profile", callback)` is the function that would handle GET requests to the **/profile** endpoint.

The only REST endpoints in the backend code are related to the exercises section of the system as those need to be stored in a global database. Most of the communication between the front and back is implemented through WebSocket connections.

Get Exercise Endpoint

The **/exercise** endpoint is a GET request that returns the related exercise in the database that corresponds with the ID that is sent along in the query string.

The callback function that deals with the request and sends a response is shown for this endpoint and it sends a response of 404 if it can't find the exercise based on the ID passed in the request and a 500 if there was an error with processing the request (such as if the database is down). Otherwise it will send a 200 and the exercise JSON object.

Snippet 1: /exercise endpoint

```
server.get("/exercise", (req: Request, res: Response) => {
    const { id } = req.query;

    Exercise.findById(id)
        .populate("activities")
        .exec()
        .then(exercise => {
            if (exercise) {
                res.send(exercise);
                return;
            }
            res.sendStatus(404);
        })
        .catch(err => {
            res.status(500).json(err);
        });
});
```

Create Exercise Endpoint

The **/create** endpoint is a POST request used when a user makes a new exercise. Here the request object is broken down to get the parameters sent with the request and is turned into objects that can be inserted into the database.

The response is the endpoint that the frontend can use to navigate to the page for the newly generated exercise.

Snippet 2: /create endpoint

```
server.post("/create", (req, res) => {
    const { activities, title, description, language }
        = req.body;

    console.log(req.body);
    const acts = activities as IActivity[];

    {...}
});
```

4.1.3 Docker Integration

For the backend to have the ability to create and link Docker containers to a user running the application on the frontend it needs to be able to interact with the *Docker Socket*. Every machine with an installation of the Docker Engine has a Docker Socket which is what the Docker CLI uses when commands are run against it.

There is a popular package on NPM called **Dockerode** [27] which enables interaction with the Docker API via whatever socket/path is provided in it's configuration.

The following code snippet shows the instantiation of the Dockerode package using the local Docker socket and exports it for use in other files in the project's backend.

Snippet 3: Create Docker instance and point it to local Socket

```
import Docker = require("dockerode");
const SOCKET_PATH = "/var/run/docker.sock";
const options = { socketPath: SOCKET_PATH };
export default new Docker(options);
```

Provisioning a User Allocated Container

Creating a container for every user that connects to the system requires the concept of a *basic image*. This image is a Dockerfile which specifies the defaults for all user's environments. The Dockerfile is responsible for configuring the environment so that it is secure and pre-installed with all the tools that the user might need.

The basic image, comes with the following software pre-installed: **Alpine Linux Distro**, **Bash**, **Python3**, **Node.js**, **GCC** and **Git**.

Alpine Linux is the distribution that the base image of the container which is based on Ubuntu. Bash is a very common shell which is a better default than the standard *ash* or *sh* which are the shells that come with the Alpine image. Bash is important for the code execution aspect of the system which is explained further in *Executing Code - 4.1.3*. Python, Node and GCC are chosen as those are the three runtimes that are supported by the system. Git is installed so if the user writes something that they want to be able to save they can use Git through the command line.

Some additional configuration that is done in this Dockerfile is the creation of the user account that users of the system will be operating as while they're connected to the container. By default the Docker engine sets the user of a container as root but this is obviously not appropriate for a system where anyone can play with a container so a low permission user is created called *damien* who has their own home folder and ownership of that folder but everything under the root directory is protected.

The JavaScript to create the container for the user is a simple function call referencing the Docker API variable.

Snippet 4: Create container with options

```
const container = await docker.createContainer({
    Image: "basic",
    AttachStdin: true,
    AttachStdout: true,
    AttachStderr: true,
    Tty: true,
    Cmd: ["/bin/bash"],
    OpenStdin: true,
    StdinOnce: false,
    name
});
```

This tells the API to create a container using the image with the label "basic" which is what the Dockerfile described above has the label of. The various `AttachX` properties tell the container if they should allow other processes to attach to this containers Standard Input/Output/Error which, as this container is emulated on the front end is required to be `true`. Tty refers to an old way of referring to the interface for a terminal. Without this option set to true it won't display in a way that looks like a traditional command line environment. Cmd is the command that the container should run once it's been created, in this case it needs to run bash. OpenStdin allows standard input to the TTY. StdinOnce will close the STDIN connection if an attached user disconnects, this needs to be off for this system as going between an exercise and a container will detach in the way that satisfies this requirement and it needs to be able to reconnect to the STDIN. The name property is simply the labelled name of the container which is displayed to the user when they connect.

Provisioning an Exercise Container

Provisioning the exercise container is more or less the same as the user's allocated container but it has to pause the allocated container so that resources aren't being wasted and then create the container for the exercise.

Exercise containers are created when a user enters an exercise and are destroyed when a user leaves the exercise. The are created the same way with the same configuration as the allocated containers however the image they are based from is the simplest REPL image that exists that relates to the runtime that the exercise is for.

Executing Code

Getting code from the server to inside a file on the container and then executing is a fundamental requirement of this project and is achieved by taking advantage of Bash which is configured to come on every container created by the system.

The execute command in Docker (exec) is only capable of running a single command with arguments. In Bash however there is a way of chaining commands as arguments using the **-c** option. A JavaScript function called `getCodeSaveCommand` creates the command that the Docker execute command can run in order to save the file.

Snippet 5: Create command to save code to container

```
export function getCodeSaveCommand(filename, code) {
    let cmd = ["/bin/bash", "-c"];

    code = code.replace(/‘/g, "\\‘");

    cmd.push(‘echo "${code}" > ${filename}‘);

    return cmd;
}
```

This snippet will add an escape character in front of all double quotes so that the double quotes don't finish the `bash -c` command and add command that saves the code to the specified file to the cmd array. This array is what the CMD option accepts.

After the file has been saved successfully a message is sent to the client confirming the save and the client sends an attach request for the code execution so that the STDIN and STDOUT can be attached to the terminal emulator.

The execute command to run the code is more straight forward than the command to save it to a file.

Snippet 6: Creating the code execution command for the container

```
export function getCodeExecutionCommand(filename, repl) {
    let cmd = ["/bin/bash", "-c"];

    if (repl === Repl.C) {
        return cmd.concat(
            `gcc ${filename} && ./a.out && rm a.out`
        );
    } else {
        return cmd.concat(
            `${repl} ${filename}`
        );
    }
}
```

This snippet works along the same lines as the previous however more steps are involved for the C compilation step as an output file is generated which has to be executed.

4.1.4 WebSockets - Backend

As mentioned in the Solution Approach (Section 3.2) the standard WebSocket client that is available as a browser API and on the backend a middleware package `express-ws` [28] is being used to allow connections to the server using the WebSocket protocol.

Endpoint Configuration

For the server to be able to create a WebSocket connection with clients and endpoint must be created that accepts the WebSocket protocol.

Snippet 7: Setup of WebSocket Endpoint

```
    server.ws("/", (ws: WebSocket) => {
        console.log("Connection Made");
        startBasicContainer(ws)
        {...}
    }
```

The snippet above shows that a WebSocket connection can be made to the root endpoint of the server and once the connection is made it is logged to the console and the function to create the basic user allocated container is called.

Message Structure

WebSockets are only capable of sending strings of text in their messages however, as JSON is a way of representing objects through strings a template message guide can be created.

Snippet 8: WS message

```
const message = {
    type: MessageTypes.CONTAINER_STOP,
    data: { id }
};

socket.send(JSON.stringify(message));
```

This snippet shows that an example message which is a JSON object with two properties `type`, which represents the type of the message being sent, and `data` which is an object itself which contains any relevant information that might be useful for the other end of the socket. In this case the type of the message is a flag to stop a running container and the data is the ID of the container. This is sent after being stringify-ed by the built in JSON object.

Message Types

As can be seen above each message has a type. These types are processed through a `switch statement` which inspects the type, extracts the parameters from the `data` property and makes a function call.

Snippet 9: How the messages are processed by the backend

```
const { type, data } = JSON.parse(msg);
switch (type) {
    case "Container.Pause":
        // Used when focus is lost from tab
        console.log("Pausing container");
        stopContainer(ws, data.id);
        break;
    case "Container.Resume":
        // Used when focus is resumed via tab
        console.log("Resuming container");
        resumeContainer(ws, data.id);
        break;
    {...}
```

The first thing that is done when the message is received is, as the message is a string, it is parsed into JavaScript objects and the `type` and `data` properties are extracted. The `type` is switched against and based on what the value of it is. A string is logged to the console showing what action the server is performing and a function is called which will always pass the WebSocket object (so the server can reply) and then passes any relevant data that is required by that function.

WebSocket Streams

Streams is a concept in programming which quite directly means a *stream of data*. Streams are used most often to act on a huge amount of data in a more performance focused way. Streams of events are the types of streams that are used in this project however as the Docker containers are able to stream their STDIN and STDOUT. Using the Node.js Stream API it is possible to `pipe()` these streams over WebSockets.

The package `websocket-stream` is used in the server to enable the streams to be piped over the WebSocket connection.

Snippet 10: Endpoint which connects the container stream to the WebSocket

```
server.ws("/connect", (ws: WebSocket, req: Request) => {
    const stream = websocketStream(ws, { binary: true });
    console.log("Trying to connect streams");

    attachSocketToContainer(
        stream,
        req.query.id,
        req.query.bidirectional,
        req.query.logs
    );
});
```

This snippet shows that a WebSocket connection can be opened to the `/connect` endpoint of the server where a stream will be created from the WebSocket. When the connection is made a function is called to attach the WebSocket stream to the container stream and it passes the stream created by the websocket-stream package, the id of the container to attach the stream to, whether the stream is bidirectional (allows STDIN and STDOUT) and if the previous logs from the container should be allowed.

Streams are a core concept in Node.js so passing one stream to another is simple.

Snippet 11: Attachment of the container stream to the WebSocket stream

```
container.attach(
    {
        stream: true,
        stdout: true,
        stderr: true,
        stdin: isBidirectional,
        logs: showLogs
    },
    function(err: Error, stream) {
        {Error Handling here...}
        console.log("Stream Connection Established!");
        if (isBidirectional) {
            stream.pipe(wss);
            wss.pipe(stream);
        } else {
            stream.pipe(wss);
        }
    }
);
```

The snippet above is showing the Docker API making a call to attach to the running container which was calculated based off the ID passed to the function. The options show that the `stream` option is set to true, the `stdin` option is dependent on if the stream is set to be bidirectional or not and the `logs` are also determined by the parameter passed from the query string.

The callback function does error handling and then will pipe the container stream to the WebSocket stream. If bidirectional flow is enabled, it will also pipe the WebSocket stream to the container.

4.2 Frontend

The general approach for developing the frontend of the project is stated in Chapter 3 in Section 3.3 but a number of other packages were used to make the development of the solution smoother and less error prone. Namely, TypeScript [23] was used rather than plain JavaScript and Styled Components [29] was used in addition to regular CSS to make implementing the design more straightforward.

The frontend consists of 4 pages or screens all of which have varying degrees of functionality and interaction with both the backend and the user.

4.2.1 WebSockets - Frontend

The WebSocket configuration is fairly similar to the backend. The browser has the WebSocket object in it's global scope so there is no need to import a package like had to be done for the backend.

All the configuration for setting up the WebSocket connection and handling events is done in the `componentDidMount()` lifecycle of the entire application.

Starting the Connection

Snippet 12: WS connection setup

```
componentDidMount() {
    this.socket = new WebSocket('ws://localhost:4000/');

    this.socket.onopen = () => {
        console.log('Socket Opened');
    };

    {...}
}
```

This snippet is creating a new WebSocket object and passing the URL of the path that the connection will be between. In this case it's the root WebSocket path that is shown in 4.1.4 which initiates the container for the user.

Receiving Messages

As the structure of messages is predictable (see 4.1.4) the same approach to deal with messages is used on the client-side as the server-side.

Snippet 13: WebSocket event listener

```
{...}
this.socket.onmessage = (event) => {
    const { type, data } = JSON.parse(event.data);

    switch (type) {
        case MessageTypes.CONTAINER_START:
            console.log('Container Started');
            {...}
    {...}
{...}
```

Here the WebSocket is registering an `Event Listener` which performs the same basic `switch statement` that the backend is performing.

After a message comes through, depending on it's content it will modify the global state of the application so that all screens are able to inspect the current state of the socket and any response that might be relevant to the functionality of that particular page. This global state is created with the built in Context API from React.

Sending Messages

Sending messages is performed the exact same way as it is from the backend, shown in Snippet 8

4.2.2 Home Page

The home page of the web app has the goal of showing users all the functionality of the system.

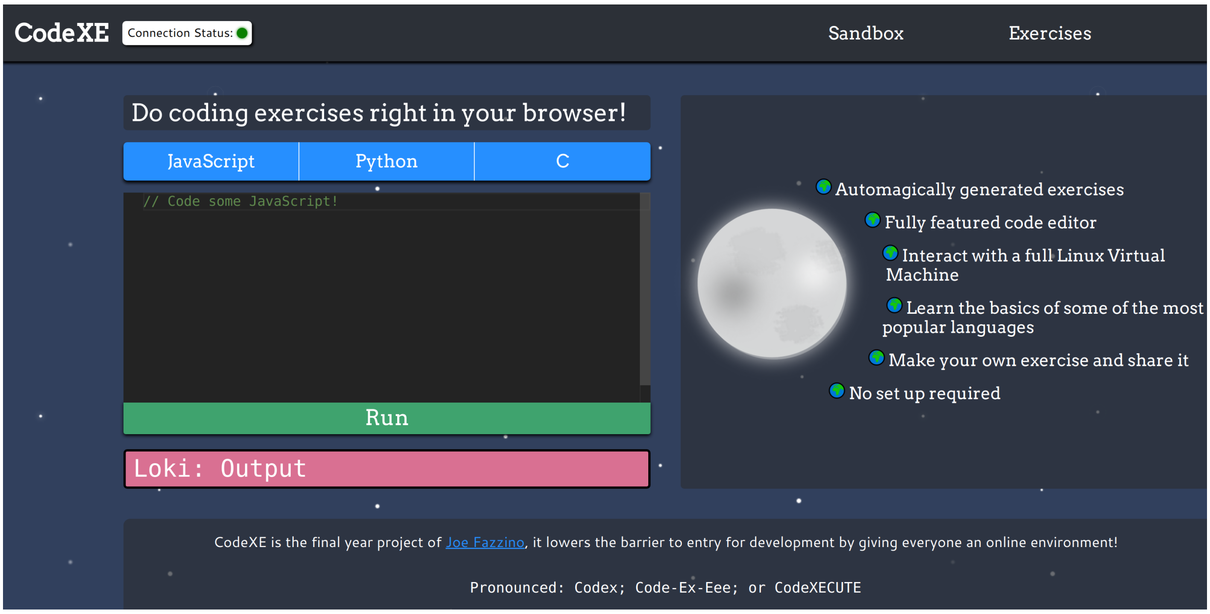

Figure 4.1: Landing Page/Home Page

At the top of the page there is a standard navigation bar which provides access to the homepage (by clicking on the name of the website), the sandbox page and the exercises page. The connection status uses a traffic light style green, yellow and red system to show the connection status to the users container. Clicking on it reveals the name of the container if connected and if there's an issue it says there's a problem.

Figure 4.2: Connection status with container name

The home page also includes the Monaco code editor and a toggle for selecting a language so the whole functionality of the site can be sampled on this page. Picking any of the languages switches to the corresponding language and pressing *Run* will execute the code and display the response in the pink output box below it which also displays the name of the container.

4.2.3 Sandbox Page

The sandbox page of the application has a simple file browser on the left which is created by passing the result of an `ls` Bash call on the container and sending the result to the

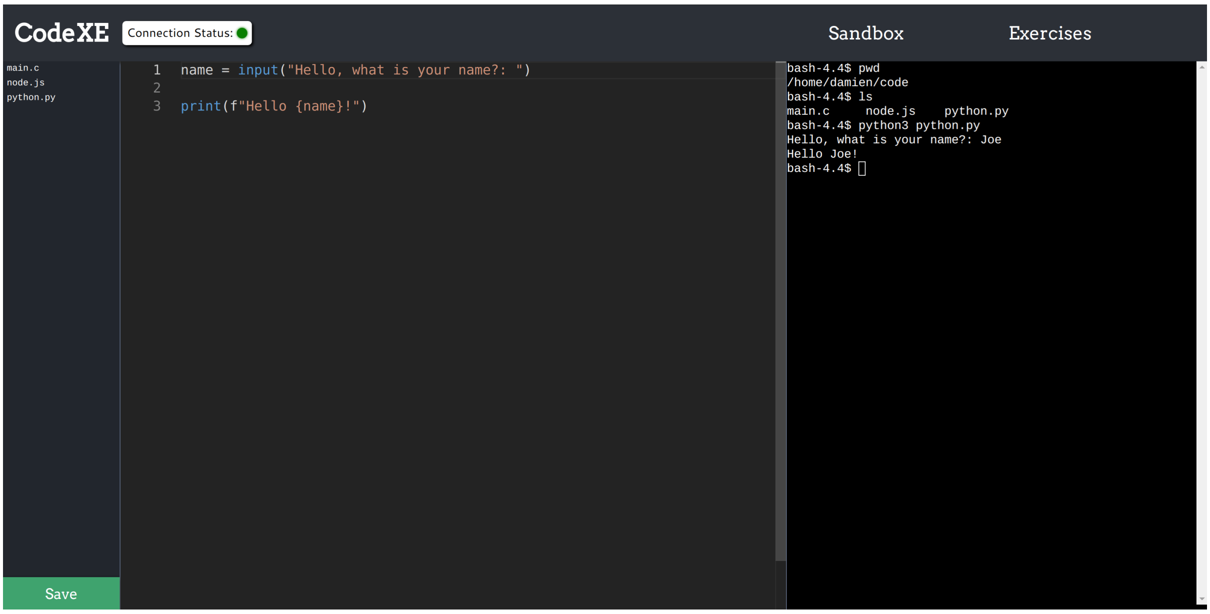

Figure 4.3: Sandbox Page

client. In the middle is the Monaco code editor and on the right is the Xterm.js terminal emulator connected to the container's stream (see 4.1.4). All three panes are resizeable.

The `Save` button will save the code to the container file that is currently open for it to be executed in the terminal emulator.

4.2.4 Exercises Page

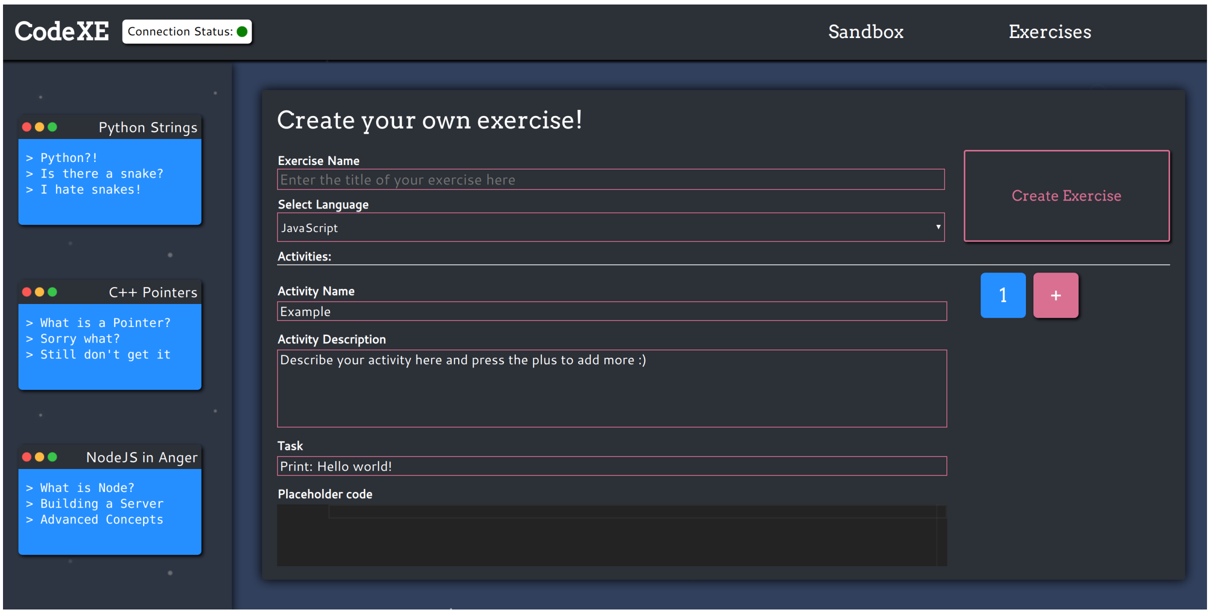

Figure 4.4: Exercises Page

The exercises page has a selection of available exercises on the left that will open the corresponding exercise screen. It also contains a form which users can create their own exercise in order to share with others or help mentor someone new to coding. Up to 15 separate activities can be made for any exercise.

4.2.5 Exercise Page

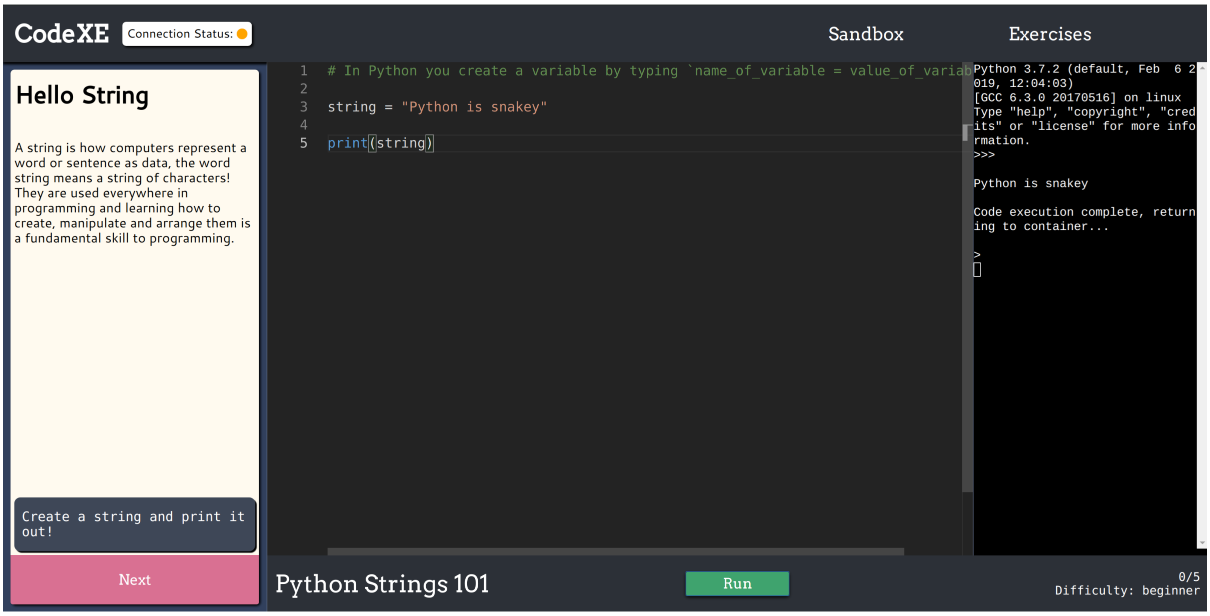

Figure 4.5: Exercise Page

The exercise page is what the user is greeted with when they interact with one of the exercises on the left of the exercises page. It displays the information for the user on the left of the specific activity they're on which is indicated by the counter in the bottom right.

This page is the most complicated of all of user facing side of the application. When the `Run` button is pressed, the best user experience is to display the result in the terminal emulator. However, this is already attached to the exercise container and having more than once connection is not a viable solution. When `Run` is pressed the message to save the code must first be sent, when that comes back as successful, it detaches the terminal from the container and makes a new request to the `/connect` endpoint of the server (see 4.1.4) to create but this time instead of passing the ID of the user's container it passes the ID of the exercise container.

When this execution finishes, the connection is re-established with the normal REPL attached stream to provide a seamless experience between playing with the REPL and executing code.

The *Connection Status* in the nav bar has changed to a yellow status, this is due to the user's main container being paused while they are interacting with an exercise container. When the user navigates away from this exercise their main container is resumed and the exercise container is destroyed.

4.3 ContainMENT - Container Management

Although containers are light weight and not very resource intensive, creating containers on demand as users make a connection with the site will lead a fairly significant scaling issue if there is no management of them in place.

Currently there is only one ContainMENT tool in place which is **Ahab** (see 4.3.1) but plans for additional tools are stated in Chapter 8.

4.3.1 Ahab

Ahab is a Python script that is responsible for making sure that any containers that have been idle for too long or exited recently are removed after a reasonable length of time so that the resources allocated to those containers can be freed for other users. The script is designed to be run every five minutes automatically as a **Cron job**

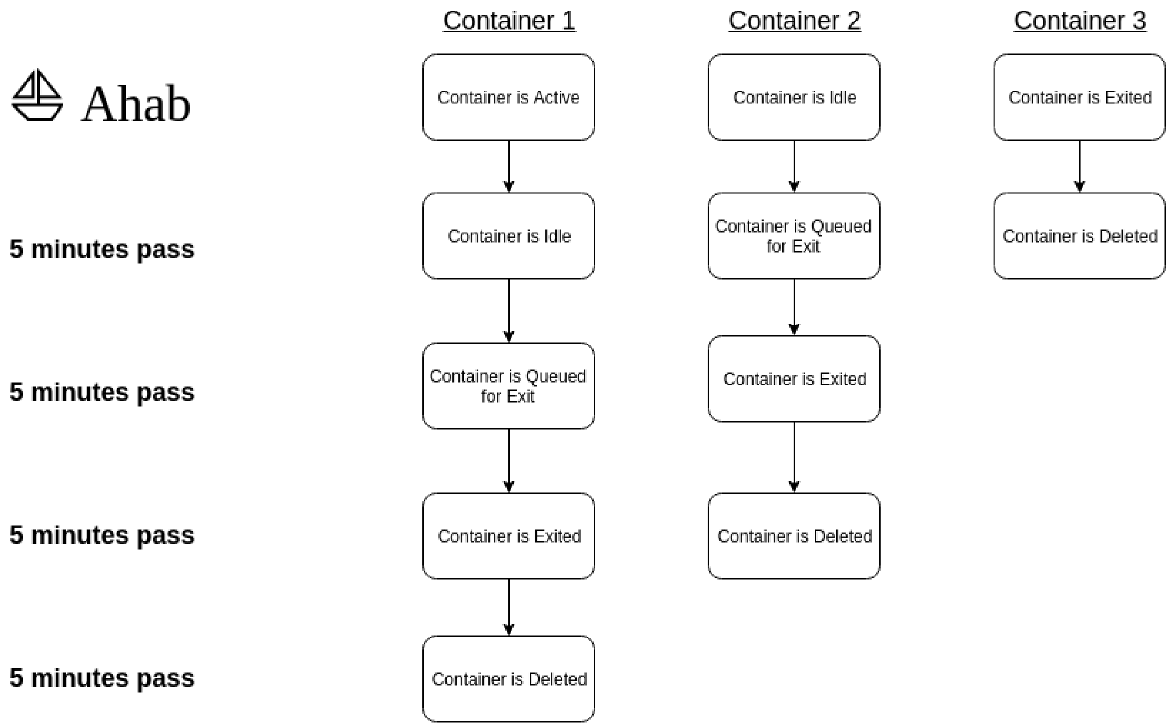

Figure 4.6: Affect of script after every execution on containers

Chapter 5

Testing: Verification and Validation

As this project has a lot of moving parts to it, testing is a necessary requirement to ensure how well the objectives stated in Chapter 1 have been met and how robust the system is generally.

Testing of the actual code that composes the system was done primarily with the Jest testing framework [30]

5.1 Usability Testing

Usability testing is the process of making sure the features that have been implemented are all working by going through them one by one and assessing their performance and quality.

Feature	**Usability Summary**
Monaco Text Editor	Feature works fully with syntax colouring, auto complete with JavaScript but not other languages
Code Execution	Fully functioning in all areas that are applicable
Terminal Emulator	Performs task well with input and output link issue with small level of latency where the messages are being buffered to only send every 10ms, leads to skipping some characters
File browsing	Opening a file and saving to it works, issue where folders aren't displayed correctly
Doing an exercise	Works, would be good for code validation to make sure the exercise output is correct but functionally works well
Creating an exercise	Works well, currently C exercises can't be made due to the lack of C REPL available
Sandbox Page	Would be good to have a run button like in the exercise page, also an issue with resizing windows going off the page
Home Page	Would be nice if any code written in the windows was reloaded when the tab to switch language is pressed rather than just putting the default comment in.

Table 5.1: Usability testing of system

5.2 Compatibility

Although web applications don't have to worry about the metal of the system that the browser is relying on, several browsers use different engines and processors in order to render DOM elements to the screen. This means it's good practice to ensure the web application being developed is functional on the different popular and modern browsers.

Browser	UI Compatibility	Functionality Compatibility
Google Chrome	Fully compatible	Fully compatible
Mozilla Firefox	Mostly compatible, only noticeable glitch is the page gains padding when the Monaco auto complete appears	Fully compatible
Microsoft Edge	Partially compatible, strange squashing of nav bar component	Fully compatible
Apple Safari	Mostly compatible, some scaling issues with text	Fully compatible
Internet Explorer 11	Not compatible, the web app uses CSS Grid for page layout which is not supported in IE 11	Un-testable

It's worth noting that Google Chrome's engine, Chromium is now powering a canary build of Microsoft Edge and is already powering Opera, this means that websites that are compatible with Chrome will be equally compatible with these browsers.

Internet Explorer 11, while having the second highest market share of browsers [31] is still only 9.83% with Firefox close behind at 9.62%. Overall coverage of the application is 84.83% of all browsers which is a significant volume of users.

Adding support for IE 11 is an option for the future however, considering Microsoft are pushing their Edge browser over IE it isn't a high priority. Most of the usage will be front enterprise machines which can't run latest versions of OS's or web browsers due to security concerns.

This compatibility test didn't test mobile devices as they aren't supported by the Monaco editor so for now a landing page is rendered saying that mobile support is coming.

5.3 Code

Testing code is a way of making sure that the end product that is created is robust to future change. Code testing can come in many forms but this project has focused on **Unit Testing**.

5.3.1 Unit Testing

Unit testing is the method of testing a component of the system as though it is a completely isolated module a benefit of this is that when writing unit tests themselves it can reveal that code that was previously thought to be modular is not.

Testing complex functionality is a high priority when thinking about writing unit tests, the WebSocket receiver functionality of the frontend is a good nomination for a test suite as it has many different outputs depending on the WebSocket event received. It also opens up the idea of Test Driven Development because if a new feature is being developed that would involve a new WebSocket event to be received, the event can be mocked (shown in Snippet 14) and the expected output can be defined. From this starting point the test will fail and the functionality can be added to the `switch statement` so that the test passes.

Snippet 14: Test for Receiving Container.Start Message

```
const MOCK_STATE = {};

test('Container Start', () => {
    const MOCK_EVENT = makeEvent(
        MessageTypes.CONTAINER_START, {
            name: 'Tester',
            info: { Config: { Hostname: 'Tester' } }
        });

    expect(handleMessage(MOCK_EVENT, MOCK_STATE))
        .toMatchSnapshot();

    const TEMP_STATE = { containerName: '', id: '' };

    expect(handleMessage(MOCK_EVENT, TEMP_STATE))
        .toMatchSnapshot();
});
```

This test is checking to see if, when a mock event is passed to the function that handles the message, the output is correct and matches the previous snapshot. If the output changes (because the function changes) then this test will fail.

The backend of the application can be tested in a very similar way by mocking inputs and snapshotting outputs, functions can be tested in a way that means they are robust to future change in the codebase so any changes that are unexpected will cause the test to fails.

5.4 Performance

Performance testing is making sure that the system is working in an acceptably fast way. Measuring the performance of the front end is done by using the built in Lighthouse tool in Google Chrome and for the backend, as it is famously difficult to measure Docker performance [32] the measuring will be done locally with the Docker CLI `docker stats` command which provides real time information on the consumption of the containers that are running.

5.4.1 Lighthouse Audits

Built into Google Chrome is a website auditing tool called *Lighthouse* [33] which can measure many different aspects of web applications such as their performance, search engine optimisation, accessibility, and best practices.

Performance is focused on things like the first meaningful paint to the screen and how quickly the web page is able to be interacted with.

Search Engine Optimisation (SEO) is simulating how well a web crawler can crawl through the page and generate a site map so the pages are visible on a search engine.

Accessibility measures important aspects such as if screen readers can interpret the elements on the page and if any colours aren't contrasting enough for those hard of sight to interpret the difference between.

Best Practices compares the website against industry standard on how to create a good modern website, this is a slightly more abstract concept to measure than the other sections however it is something Google consider important enough to include in their auditing tool.

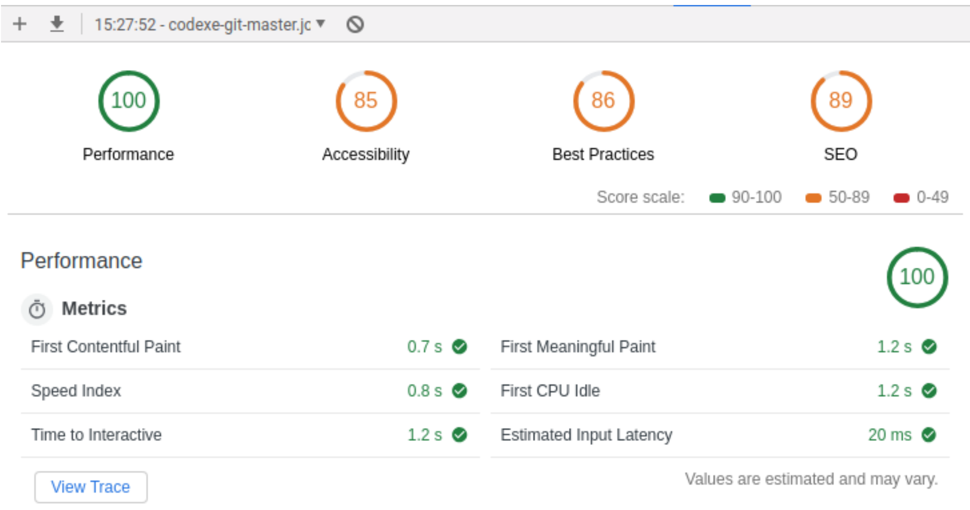

Figure 5.1: Lighthouse audit of deployed application

The figure above shows some excellent results for the different measurements. Performance being 100 is particularly notable as most of Google's own products don't meet that high. This performance result is due to how Next.js packages the application for deployment which converts all of the React layout code into normal HTML, CSS which is very fast to render. It also isolates each page into what they call a *lambda* function which means that the pages aren't always running and instead will be loaded on demand. This means there's no resource wastage from a server running 24 hours a day. The different metrics are shown at the bottom of the figure with extremely fast response times. It is worth mentioning again that this is a deployed system and is not running locally or hosted on the same LAN.

The accessibility score is only 85 as the code editor's theme has the green of the comment which has low contrast compared to the dark background, there is also an issue in the bullet point list of site features as screen readers will announce the globe images when really they shouldn't be noted. This can be fixed with an `aria-hidden` attribute on the element which will tell screen readers to ignore it.

The best practices score is due to errors being logged in the console. As only the frontend is currently deployed, the WebSocket fails to connect which results in errors logged to the console. This will be resolved once the backend of the application is deployed.

The SEO score is due to there not being a meta description of the website in the `<head>` which is what provides search engines with their summary of the website. This is very easy to resolve.

5.4.2 Docker Stats

A number of tests were performed while visually observing the running of the `docker stats` command which provides real time usage on CPU, memory, I/O and more. It is worth noting before observing the results that while the usage can say 0% that does not mean that 0% of the CPU is being occupied by the container as there is a base level allocation of usage that, while may not be consumed, is still unavailable to other processes. This is mentioned in [32].

Operation	**CPU Usage (%)**	**RAM Usage (MiB)**
Server Started	Server Idling at 0.1	Server Idling at 195.7
Container Started	No change in server. New container 'Modi' idling at 0.	Server idling at 196.3. Modi idling at 1.234
Counting to 50 million	Server jumps to 0.66. Modi peaks at 55.	No change in server. Modi peaked at 2.55. Modi idles higher post execution at 2.023
Connecting the Terminal	Server jumps to 1. No change in Modi.	No changes in either container.
Typing in the Terminal	Server peaks at 3.88. Modi peaks at 0.35.	No changes in either container.
Opening a file	Server peaks at 0.7. Modi peaks at 4.3.	No changes in either container.
Opening a Python Exercise	Server doesn't change. Modi pauses and idles at 0. Python container idles at 0.	No change in server memory use. Modi idles at 2.023 Python container idles at 5.625.
Counting to 50 million in exercise	Server doesn't change. Python peaks at 50.	Server doesn't change. Python peaks at 8.
Opening a new tab	Server behaves the same as opening container. New container idles with 0.	No change to server. Modi remains idling at 2.023. New container idles at 0.956.

Table 5.2: Results of *docker stats* through different operations

All tests were done 5 times and an average was taken. The specification of the machine that the testing was performed on is 3,500MHz CPU and 15,390MiB.

Observations

The results above show some interesting observations. As theorised in 4.1.4, the main server which creates the containers and connects them never has very high CPU usage, any high CPU usage from Node.js is when there is typing in the terminal which is when the WebSocket Streams (see 4.1.4) are connected. This shows that Node.js is a good technology to use when there isn't a heavy amount of processing needed to be done by the server such as this system where most of the work is being done by the Docker API. The memory usage isn't exactly high but is quite a bit higher than any other container, this could be due to the various components that make up Node.js such as the V8 engine and libuv.

The results also show that containers will behave like ordinary processes and take up as much available CPU as they possibly can in order to complete intensely computational tasks. If there are 10 concurrent users all attempting to perform computationally heavy tasks the CPU of the machine hosting the containers will be scheduled by the Completely Fair Scheduler which is the default scheduler in the Linux kernel [34]. Containers will take up as much CPU as they can when they are doing intense computation.

The containers don't tend to use a significant amount of memory although it depends on the kind of container and the operation running. Running the "Counting to 50 million" exercise on the homepage with JavaScript had a peak of only 2.55MiB but Python peaked at 8MiB. These results should not be taken as fully accurate metrics as the `docker stats` command only updates once every second. The containers never idled memory usage higher than 5.625MiB at any time.

These results will be very helpful in the future for when the backend can be deployed in deciding what appropriate specifications the hosting machine should have for a well performing system.

5.5 Security

Security is a big concern with a system where users have the ability to not only execute code on a remote machine but also have access to the shell of a container that will be deployed to the same space that the entire backend will be. Despite containers providing huge advantages speed/size wise over hypervisor powered virtual machines, from a security perspective there is a large problem. The containers are running on the kernel of the machine that the container provisioning engine is running on so if a user is able to exploit a kernel bug, it means that they can break out of the container and start changing things in the shell of the machine hosting the entire system. This would be terrible if the attacker has malicious intent.

A lot of kernel exploits are only possible with root access so the first step to securing the containers is making sure that the user account that the users are logged into is not the normal docker root user. This is briefly discussed in Section 4.1.3 with the Dockerfile that creates the *damien* user which is what users are occupying while they're on the system. Inside this normal user account access to sensitive information is well secured, a lot of commands that can be used maliciously are locked, the ability to install more software (like cracking tools and fingerprinting tools) is not available. One potential issue is that some of the default binaries installed which are usable, even though they aren't meant for malicious purposes could be exploited to do something malicious. The `wget` package is an example of this as on the surface it's a tool for downloading a web page but it could be used to download scripts online and then so long as they're in the users home folder they will be able to execute them.

More tools that could help prevent exploitation of the service that this project is providing would be a long running watcher which checks to see that there aren't any containers operating at 100% CPU usage for too long. This type of usage could indicate that someone is trying to break the containers/the overall system. Usage limits and tracking would aid in detecting this behaviour and providing a warning to users that this type of activity could result in being black listed from the system.

Preventing security issues is an endless task however a new tool from Google is proving effective at mitigating the risk of using containers as remote environments. gVisor [35] is compliant with the Open Container Initiative and works as a drop in runtime for the `runc` runtime that Docker uses by default and provides a small layer of security on top similar to how a traditional hypervisor provides security.

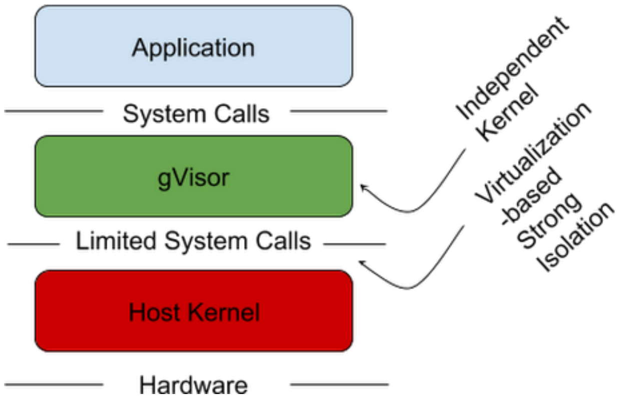

Figure 5.2: gVisor Runtime Architecture [36]

Shown in the article [36] simply replacing the `runc` runtime with the gVisor based `runsc` means that a specific kernel exploit no longer works in containers. This type of technology is inevitable as more huge industry giants adopt containerisation and provides an optimistic view of how using containers can be both well performing and secure.

Chapter 6

Discussion: Contribution and Reflection

Chapter 7

Social, Legal, Health and Safety and Ethical Issues

There are some concerns with the project's output when it comes to the social, legal, health and ethical issues. Socially, the fact that it's an online environment means that someone with limited internet connection or even intermittent internet connection will struggle to use the site reliable and adequately replace a local development environment. This issue could be mitigated if the website was converted to a progressive web app which would enable offline functionality. The project is also limited socially to people who don't speak English as a first language as currently that's the only option, internationalisation support would resolve this issue.

There are some potential legal issues with the system created. As unfettered access to a system where a user has remote access to a virtual Linux environment. Despite the precautions spoken about in *Security - 5.5* it isn't feasible to expect that all security holes have been accounted for. A user agreement might be necessary to legally separate the system and developer from the actions of the user on it. This point is also an ethical issue, a significant effort must be made by the developer to make sure that there is adequate monitoring and preventive measures in place.

A lot of the software used to implement this system is open source software meaning that the code is published in full online. The packages in use with this project are created under various open source licenses which all allow software to be used commercially [37].

Chapter 8

Conclusion and Future Improvements

This project has been an exercise in building a fullstack web application which attempts to create an experience that encourages users to experiment with programming in their own personal environment which closely emulates the experience they would get locally. It successfully incorporates industry standard tooling and familiar UI in a way that means users won't be challenged going from this system to a full integrated developer environment.

The resulting system has achieved these base objectives that were set out and the results show that it has been achieved in way that performs well when measured against modern standards and practices.

There is a lot of space in the future of this project to grow it out into a full product. One of the first priorities for the future would be to implement the gVisor sandbox runtime for the additional security benefits it provides. Deploying the application on a permanent basis would also be a priority but further testing would be required so that the correct configuration for the host of the server could be calculated for the lowest cost. A final future feature for the system in general would be implementing container snapshots and user accounts so that it's possible to save the state of a user's allocated container and come back to it, this would mean that it is escalated to a proper environment that a user can make personal and use repeatedly.

Bibliography

[1] S. Chalmers, “Why schools in england are teaching 5 year olds how to code,” Oct. 2014. [Online]. Available: https://bloomberg.com/news/2014-10-15/why-schools-in-england-are-teaching-5-year-olds-how-to-code.html (visited on 04/01/2019).

[2] T. Claburn, “Carlo has a head for apps and a body (tag) for rendering: Google takes on electron with js desktop app toolset,” Nov. 2018. [Online]. Available: https://theregister.co.uk/2018/11/02/carlo_chromium/ (visited on 04/02/2019).

[3] V. Pimentel and B. G. Nickerson, “Communication and displaying real-time data with websocket,” May 2012. [Online]. Available: https://ieeexplore.ieee.org/document/6197172 (visited on 04/03/2019).

[4] I. Fette and A. Melnikov, “The websocket protocol,” Dec. 2011. [Online]. Available: https://tools.ietf.org/html/rfc6455 (visited on 04/03/2019).

[5] V. Choudhary, “Http vs websocket (or http 2.0), which one is right for you.,” [Online]. Available: https://developerinsider.co/http-vs-websocket-or-http-2-0-which-one-for-you/ (visited on 04/06/2019).

[6] S. Dutton, “Getting started with webrtc,” Jul. 2012. [Online]. Available: https://html5rocks.com/en/tutorials/webrtc/basics/ (visited on 04/06/2019).

[7] *Tcp vs. udp*. [Online]. Available: https://diffen.com/difference/TCP_vs_UDP (visited on 04/06/2019).

[8] P. Staff, “Websockets vs rest: Understanding the difference,” *PubNub*, Jan. 2015. [Online]. Available: https://www.pubnub.com/blog/2015-01-05-websockets-vs-rest-api-understanding-the-difference/ (visited on 04/16/2019).

[9] F. Lardinois, “Repl.it lets you program in your browser,” 2018. [Online]. Available: https://techcrunch.com/2018/03/15/repl-it-lets-you-program-in-your-browser/ (visited on 04/03/2019).

[10] M. G. Sumastre, “Virtualization 101: What is a hypervisor?,” Feb. 2013. [Online]. Available: https://pluralsight.com/blog/it-ops/what-is-hypervisor (visited on 04/04/2019).

[11] R. Dua, A. R. Raja, and D. Kakadia, “Virtualization vs containerization to support paas,” Mar. 2014. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6903537 (visited on 04/04/2019).

[12] *Cgroups - linux control groups*. [Online]. Available: http://man7.org/linux/man-pages/man7/cgroups.7.html (visited on 04/16/2019).

[13] *Namespaces - overview of linux namespaces*. [Online]. Available: http://man7.org/linux/man-pages/man7/namespaces.7.html (visited on 04/16/2019).

[14] *Jquery*. [Online]. Available: https://jquery.com/ (visited on 04/03/2019).

[15] *Npm search react*. [Online]. Available: https://npmjs.com/search?q=react (visited on 04/04/2019).

[16] *Npm search vue*. [Online]. Available: https://npmjs.com/search?q=vue (visited on 04/04/2019).

[17] *Node.js*. [Online]. Available: https://nodejs.org/en/ (visited on 04/10/2019).

[18] *Libuv documentation*. [Online]. Available: http://docs.libuv.org/en/v1.x/ (visited on 04/16/2019).

[19] K. Trott and Y. Xiao, "Debugging node.js in production," *Netflix Technology Blog*, [Online]. Available: https://medium.com/netflix-techblog/debugging-node-js-in-production-75901bb10f2d (visited on 04/16/2019).

[20] *About .net core*. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/core/about (visited on 04/16/2019).

[21] *React documentation*. [Online]. Available: https://reactjs.org (visited on 04/16/2019).

[22] *Stack overflow*. [Online]. Available: stackoverflow.com (visited on 04/16/2019).

[23] *Typescript*. [Online]. Available: https://typescriptlang.org/ (visited on 04/10/2019).

[24] *Mongodb*. [Online]. Available: https://mongodb.com/ (visited on 04/10/2019).

[25] *Mongoose*. [Online]. Available: https://mongoosejs.com/ (visited on 04/10/2019).

[26] *Express*. [Online]. Available: https://expressjs.com/ (visited on 04/10/2019).

[27] *Dockerode*. [Online]. Available: https://npmjs.com/package/dockerode (visited on 04/10/2019).

[28] *Express-ws*. [Online]. Available: https://npmjs.com/package/express-ws (visited on 04/10/2019).

[29] *Styled components*. [Online]. Available: https://styled-components.com/ (visited on 04/11/2019).

[30] *Jest*. [Online]. Available: https://jestjs.io (visited on 04/13/2019).

[31] *Net market share*. [Online]. Available: https://www.netmarketshare.com/browser-market-share.aspx?qprid=2&qpcustomd=0 (visited on 04/14/2019).

[32] E. Casalicchio and V. Perciballi, "Measuring docker performance: What a mess!!!" [Online]. Available: https://research.spec.org/icpe_proceedings/2017/companion/p11.pdf (visited on 04/14/2019).

[33] *Google lighthouse*. [Online]. Available: https://developers.google.com/web/tools/lighthouse/ (visited on 04/14/2019).

[34] C. S. Pabla, "Completely fair scheduler," *Linux J.*, vol. 2009, no. 184, Aug. 2009, ISSN: 1075-3583. [Online]. Available: http://dl.acm.org/citation.cfm?id=1594371.1594375 (visited on 04/16/2019).

[35] *Gvisor*. [Online]. Available: https://gvisor.dev/ (visited on 04/14/2019).

[36] N. Lacasse, "Open sourcing gvisor a sandboxed container runtime," [Online]. Available: https://cloud.google.com/blog/products/gcp/open-sourcing-gvisor-a-sandboxed-container-runtime (visited on 04/14/2019).

[37] *Open source licenses*. [Online]. Available: https://opensource.org/licenses (visited on 04/17/2019).