University of Sussex

# A Web IDE for WebAssembly

**Sammy Hass - 231006**

Submitted for the degree of Bachelor of Science

University of Sussex

April 2023

https://www.overleaf.com/project/63ef1225e74b0442683b1177

UNIVERSITY OF SUSSEX

Sammy Hass, Bachelor of Science

# A Web IDE for WebAssembly

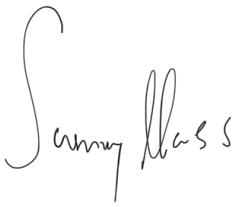
This report outlines the planning, design and development of a web-based IDE that allows developers to build new websites using JavaScript and WebAssembly via either Go or AssemblyScript as the source language for compilation. The IDE aims to allow users to develop prototypes using WebAssembly without the need to deal with the complexities of the WebAssembly workflow.

We discuss WebAssembly, why and where it is used. We will then discuss the existing landscape for WebAssembly development. Then, by analysing and comparing existing web-based solutions to WebAssembly and web development as a whole, discuss their strengths and pitfalls and subsequently describe in detail the requirements of a new tool with the knowledge we have gained through this research. We then discuss the design process of the application that was required to suitably develop it according to the requirements we specified. After this, the report discusses the development process as well as implementation details for the final system. The report then discusses the approaches taken to testing the application and the ways in which these were used to aid with the development of the system. This report will then provide a demonstration of how the developed application can be operated by users to build websites using WebAssembly.

# Declaration

This report is being submitted as a requirement for the completion of the degree in Computer Science at the University of Sussex. I hereby declare this report will not be submitted in whole or in part to another University for the award of any other degree. It is the product of my own labour except where indicated in the text. The report may be freely copied and distributed provided the source is acknowledged. I give permission for a copy of this report to be loaned out to students in future years.

Signature:



Sammy Hass - 27th April 2023

# Acknowledgements

I would like to take this opportunity to express my sincere gratitude to my supervisor, Hsi-Ming Ho, for his invaluable guidance and support throughout the duration of my project. Hsi-Ming has been an indispensable resource, always available to discuss and answer any questions I had about the process of creating this report.

His insight, expertise, and attention to detail have been instrumental in shaping and improving the quality of my work. His encouragement and enthusiasm have been a constant source of motivation, and his constructive feedback has helped me to stay on track and make progress towards the completion of this project.

I am also grateful for the patience and understanding he demonstrated during the challenging times of this project. His unwavering commitment to my success has been truly inspiring, and I feel privileged to have had the opportunity to work with him.

Once again, thank you, Hsi-Ming, for your unwavering support and guidance throughout this project. Your contributions have been invaluable, and I am deeply appreciative of all that you have done for me.

# Contents

# List of Figures

# Chapter 1

# Introduction

## 1.1 Project Background

### 1.1.1 WebAssembly

WebAssembly (WASM) [38] is a stack-based binary instruction format designed for developing isolated, highly optimised and highly-portable applications. It is supported as a standard by all web browsers and maintained by the WebAssembly Working Group [99], making it the only programming language other than JavaScript to be natively supported and standardised by all modern browsers. WebAssembly is designed to complement and not replace JavaScript [105].

As a language, WebAssembly is generally not written, but instead, is used as a compilation target for another source language such as Go [90], Rust [104], C++ [32] or AssemblyScript [84].

WebAssembly is highly portable and therefore is not only limited to running in client-side web applications. There are many different runtimes [1] that exist so that WebAssembly may be executed on different devices and compiled from a number of different source languages. This means that WebAssembly has a wide variety of applications [106], not only in the browser but also in server-side and embedded applications.

This project will focus on the use case of building powerful client-side applications that leverage inter-operation between JavaScript and WebAssembly. This is currently one of the most common and fastest-growing use cases of WebAssembly. A recent micro survey conducted by the Cloud Native Computing Foundation (CNCF) [76] found that 52% of respondents were planning to use or were currently using WebAssembly for web development. (Figure 1.1)

Examples of successful applications using WebAssembly for client-side applications include the design tool Figma [100], complex 3D fluid simulations [44] and the newly released WebContainers API [74]. The WebContainers API makes it possible to run native Node.js processes entirely inside of the browser, allowing for the creation of web servers using ‘npm‘ packages such as ‘express‘ without having to rely on anything other than a modern web browser and a secure connection [75].

These examples would have an extremely difficult time running in JavaScript alone, there are several reasons for this. JavaScript is a dynamically-typed interpreted language which [in Chrome-based browsers] uses just-in-time optimizing compilation with the V8 Engine [91] to dynamically and incrementally optimize code during its runtime, this results in a generally very good performance. However, initially, the code is not optimized when runtime begins, this makes JavaScript less ideal in situations where complex computation is required. In addition, since JavaScript is a dynamically typed language, it would not be efficient to compile it directly to byte-code due to how memory must be allocated, it must first be compiled to an intermediate language, where it can be optimized gradually

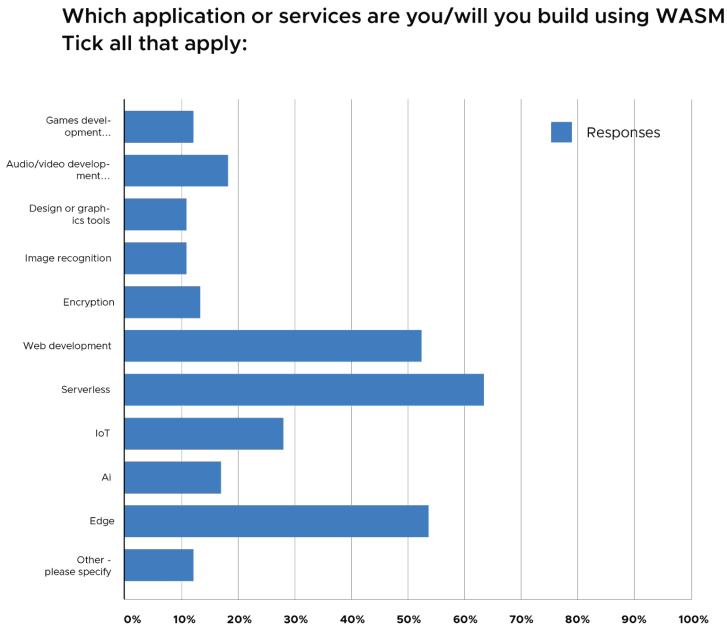



Figure 1.1: Source: CNCF WASM Microsurvey 2022 [76], WASM Use Cases

during runtime. This contrasts with WASM which can produce near-native performance due to its statically typed nature and therefore its ability to produce heavily optimized and memory-efficient binaries ahead of time.

For the above reasons, WebAssembly has become a popular choice for developing high-performance web applications, however, the ecosystem surrounding WebAssembly is not very mature and therefore setting up a developer environment is difficult for those who haven't done it before. This can be noted by the CNCF WebAssembly Microsurvey (Figure 1.2). The most common concern with WASM was that: "Developer tools are incomplete or too hard to use" with 56% of respondents selecting this as a concern.

This project aims to simplify that process by providing a fully functioning WebAssembly IDE that doesn't require any complex setup but allows the user to get started experimenting and building web experiences with WebAssembly straight out of the box.

### 1.1.2 Integrated Development Environments (IDEs)

Integrated Development Environments (IDEs) are applications that provide programmers with a suite of tools for software development. IDEs generally consist of tools for code editing, syntax highlighting, build automation, testing and debugging [13]. IDEs are generally more specialised in comparison to simpler text editors and provide centralised utilities for commonly needed developer utilities in addition to simply just editing text.

IDEs are designed to improve developer productivity and developer experience when building applications, this is done by providing a centralised suite of tools for setting up, developing, running, testing and debugging programs. Thus abstracting away the complexities of needing to manually set up complicated tools that are invaluable to software development such as a syntax highlighter, code completion, build automation, debugging features and version control.

IDEs can be classified into two types: Local IDEs and Web (Cloud) IDEs. Local IDEs are applications that are made to be downloaded onto a developer's computer and use the resources of the computer that they are running on. Meanwhile, Cloud IDEs are

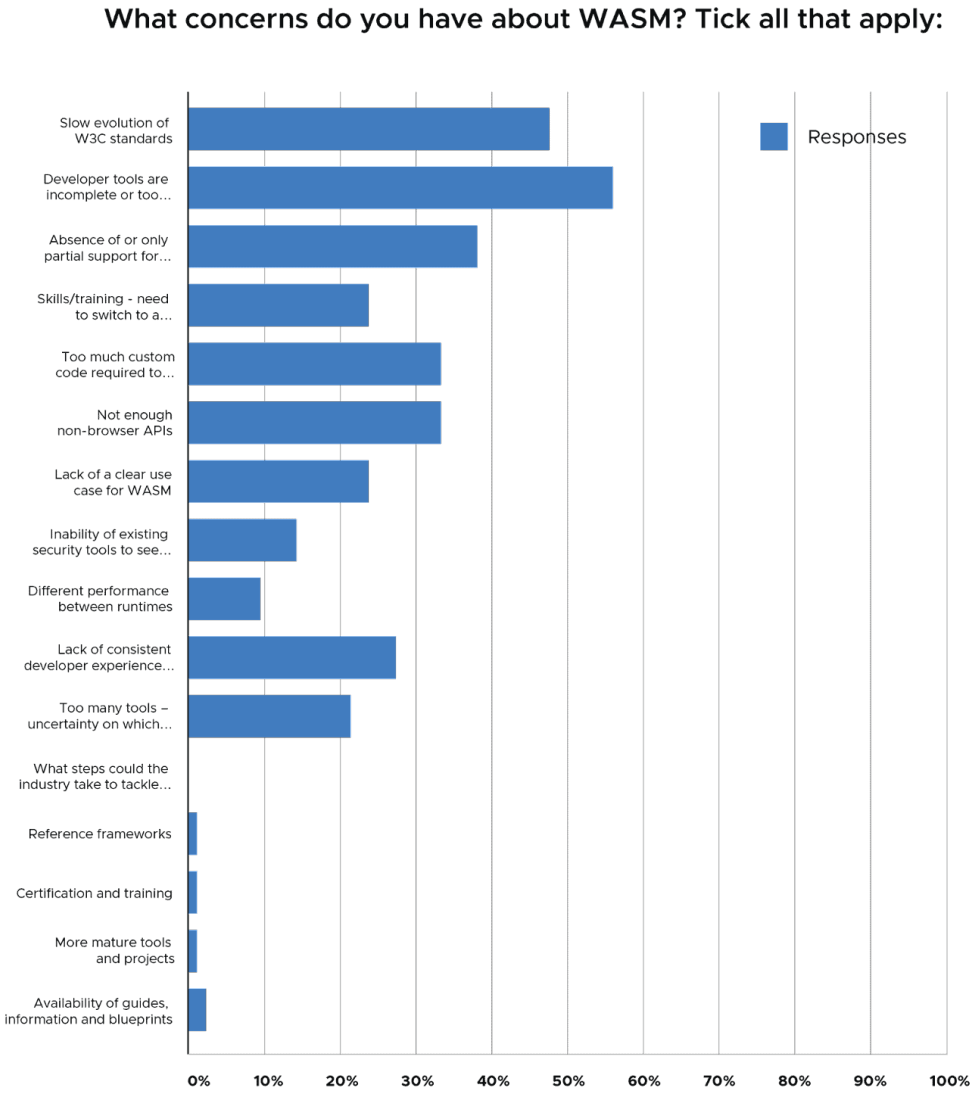



Figure 1.2: Source: CNCF 2022 [76]: Most Common Concerns

designed to allow developers to get started without needing to download anything and are completely independent of the platform they are being run on.

Web IDEs are often used for educational purposes, such as is the case with the p5.js editor, which is used widely as an educational tool [58]. These IDEs provide a standardised environment that requires zero configuration on the part of the developer. This is appealing since setting up a brand new developer environment on a new machine is often a complicated process with a variety of possible points of failure and bugs that require fixing, a process which may take days. Web IDEs completely mitigate this problem by setting up the environment for the developer, allowing the learner to focus on learning instead of first needing to take the time required to correctly set up a complete development environment locally.

## 1.2 Project Description

WebAssembly is a relatively new technology and for this reason, using it and setting up a development environment is tricky for newcomers. It is far more complicated to set up than a JavaScript environment due to the multiple steps required in writing, compiling and then actually using the compiled binary in a project. Therefore, I plan to create a fully functioning Web IDE to simplify the process of scaffolding and prototyping WebAssembly projects. The IDE will allow the user to create web pages with HTML, JavaScript and CSS as is traditional but will also allow the user to write either Go or AssemblyScript [84] code, which will be compiled (using either TinyGo [90] or the AssemblyScript compiler) to WebAssembly and subsequently be able to inter-operate with the JavaScript code the user has written via exported functions or the loaded WebAssembly module's memory buffer. The user will then be able to preview their project in their browser.

My aim in developing this project is to make it simpler for developers to work with WebAssembly by giving users a controlled environment whereby they don't need to know about the underlying complexities of how to set up WebAssembly and get it to inter-operate with JavaScript. I will place particular importance on developer experience, type safety, and ease of use to ensure that the site can be helpful as a learning tool. In addition, users will be able to view a version of their compiled binary in WebAssembly Text Format (WAT) [55] which will give users an idea of how their compiled module operates.

## 1.3 Project Relevance

This project will require knowledge across a variety of modules. It will require me to design, develop and deploy a front-end web application and an API. I will use the knowledge gained from the 'Software Engineering module (Year 2) and 'Human Computer Interaction' (Year 3) in order to design and architect the system. I will be required to use knowledge from 'Computer Networks' in building an HTTP server. I will also need to programmatically call child processes from within the server, which will utilise concepts from 'Operating Systems' (Year 2). Furthermore, I will require a database to keep track of users and projects so I will need to utilise the 'Databases' module (Year 2) and need to create a secure way for users to log in, for which I will use the experience gained from 'Introduction to Computer Security' (Year 3). In addition to all this, I will need to familiarize myself with the mechanisms of WebAssembly, its compilation process and the ways it can be utilised within a web application.

# Chapter 2

# Professional & Ethical Considerations

This Code of Conduct:

- sets out the professional standards required by BCS as a condition of membership.
- applies to all members, irrespective of their membership grade, the role they fulfil, or the jurisdiction where they are employed or discharge their contractual obligations.
- governs the conduct of the individual, not the nature of the business or ethics of any Relevant Authority

## 2.1 Public Interest

You shall:

- have due regard for public health, privacy, security and wellbeing of others and the environment.
- have due regard for the legitimate rights of Third Parties.
- conduct your professional activities without discrimination on the grounds of sex, sexual orientation, marital status, nationality, colour, race, ethnic origin, religion, age or disability, or of any other condition or requirement.
- promote equal access to the benefits of IT and seek to promote the inclusion of all sectors in society wherever opportunities arise.

## 2.2 Professional Competence and Integrity

You shall:

- only undertake to do work or provide a service that is within your professional competence.
- **NOT** claim any level of competence that you do not possess.
- develop your professional knowledge, skills and competence on a continuing basis, maintaining awareness of technological developments, procedures, and standards that are relevant to your field.

- ensure that you have the knowledge and understanding of Legislation* and that you comply with such Legislation, in carrying out your professional responsibilities.
- respect and value alternative viewpoints and, seek, accept and offer honest criticisms of work.
- avoid injuring others, their property, reputation, or employment by false or malicious or negligent action or inaction.
- reject and will not make any offer of bribery or unethical inducement.

## 2.3 Duty to Relevant Authority

You shall:

- carry out your professional responsibilities with due care and diligence in accordance with the Relevant Authority's requirements whilst exercising your professional judgement at all times.
- seek to avoid any situation that may give rise to a conflict of interest between you and your Relevant Authority.
- accept professional responsibility for your work and for the work of colleagues who are defined in a given context as working under your supervision.
- **NOT** disclose or authorise to be disclosed, or use for personal gain, or to benefit a third party, confidential information except with the permission of your Relevant Authority, or as required by Legislation.
- **NOT** misrepresent or withhold information on the performance of products, systems or services (unless lawfully bound by a duty of confidentiality not to disclose such information), or take advantage of the lack of relevant knowledge or inexperience of others.

## 2.4 Duty to the Profession

You shall:

- accept your personal duty to uphold the reputation of the profession and not take any action which could bring the profession into disrepute.
- seek to improve professional standards through participation in their development, use and enforcement.
- uphold the reputation and good standing of BCS, the Chartered Institute for IT.
- act with integrity and respect in your professional relationships with all members of BCS and with members of other professions with whom you work in a professional capacity.
- encourage and support fellow members in their professional development.

# Chapter 3

# Related Works

Web-Based IDEs are not a new phenomenon. There are a number of examples of existing online IDEs. In addition to this, there are a few existing solutions for Web-Based WebAssembly Development. In this chapter, I look into the inner workings of existing Web IDEs, with a specific focus on Web-Based IDEs for developing applications using WebAssembly. I analyse these solutions in order to develop concrete requirements for this project that will be used to guide the development of the final system.

## 3.1 WASM Fiddle

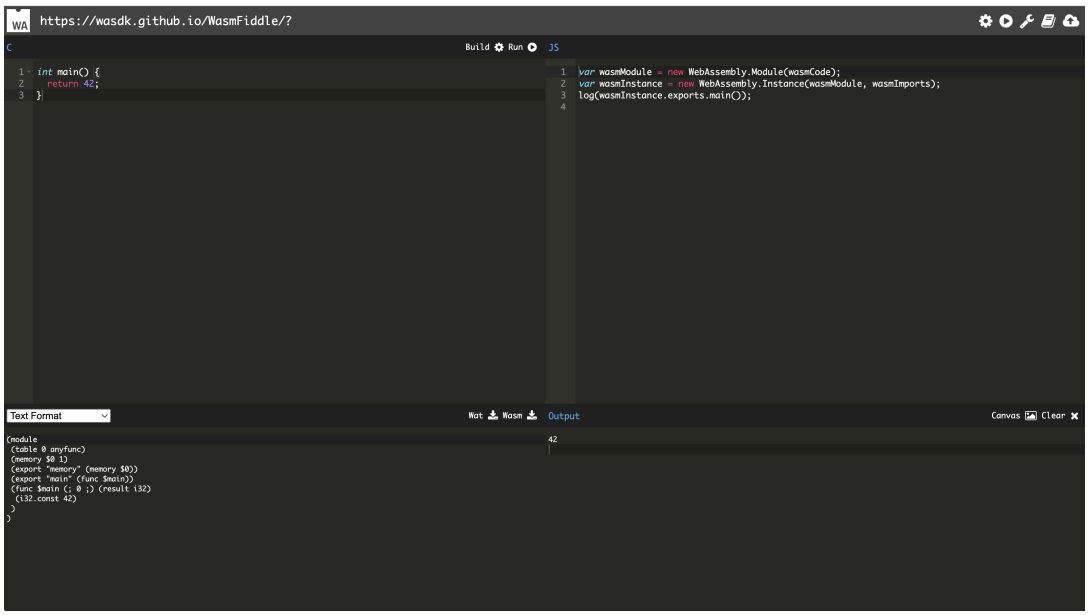



Figure 3.1: Screenshot of WASM Fiddle

### 3.1.1 Overview

WASM Fiddle [102] allows users to write source code in C and compile it to a WebAssembly module for subsequent use via JavaScript inter-operation. It has no ability to create full-fledged websites but does include the ability to access an HTML Canvas element on the page which can be used to create interactive experiences with WebAssembly. The choice to use a canvas instead of providing the user with a full HTML page is likely down to the developers only accounting for the use case in which WebAssembly is used to create

complex, high-performance graphics by manipulating pixels on the canvas. Whilst this is a valid use case, it is limiting and not particularly friendly to those who aren't familiar with HTML Canvas or those who are looking to create a website and not just graphical demonstrations.

### 3.1.2 Features and Limitations

WASM Fiddle allows users to download the compiled WASM file in the form of either WASM Text Format (WAT) [55] or as a WASM binary. This approach assumes that the user is aware of how to use use a WebAssembly module within a website. This approach to downloading and exporting projects means WASM Fiddle cannot be used to develop more complex applications, furthermore, the lack of any ability to save your work inside of the online editor makes it impossible to continue work on a project after the session has been closed. This is a limitation I seek to solve in my project by allowing users to create accounts and save their work.

In addition to providing a text editor, WASM Fiddle allows the user to change compiler options for their programs. This includes being able to choose the version of the C compiler that is used as well as the target optimisation level for the compiled binary.

WASM Fiddle has since been deprecated in place of the more complex WASM Studio, however, as I describe in the following section, WASM Studio fails to solve many of the problems with WASM Fiddle and it is now also not actively maintained.

## 3.2 WebAssembly Studio

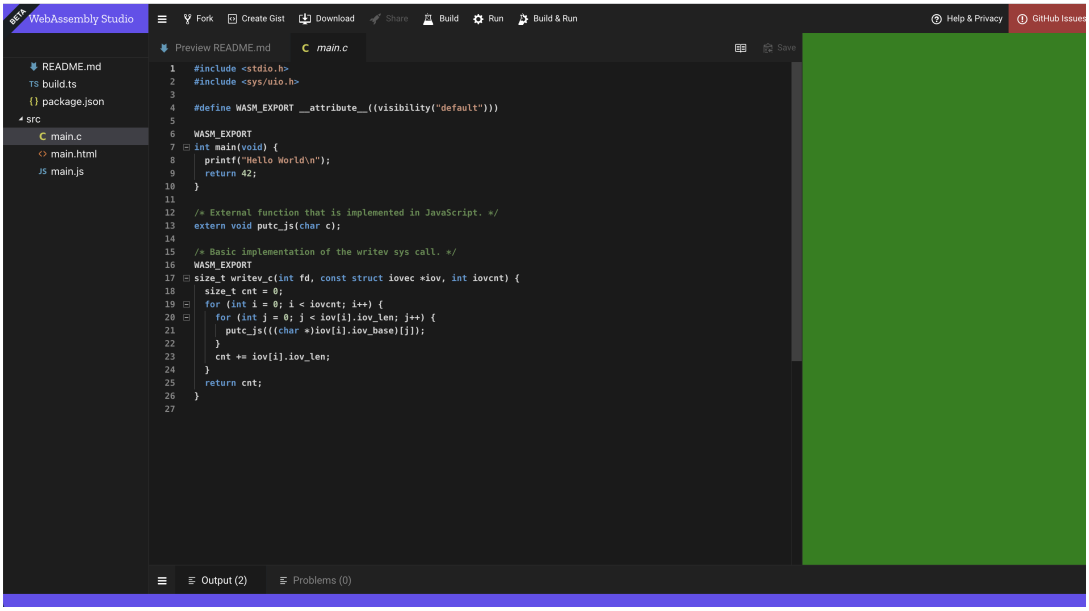



Figure 3.2: Screenshot of a C project in WASM Studio

### 3.2.1 Overview

WebAssembly Studio [103] allows the user to create a new WebAssembly project using C, C++ or Rust, then scaffolds the entire project, allowing the user to see the project structure and edit it as they please from within the browser. This is great for more experienced users; however, this makes for a difficult experience for new users who may find the number and purpose of all files confusing, furthermore, the site lacks comprehensive

documentation that one would expect from an IDE. WASM Studio is no longer actively maintained and has been archived on GitHub by the authors [2] [101].

### 3.2.2 Features and Limitations

WebAssembly Studio is great for developers who are already familiar with WASM and provides users who aren't familiar with documentation in a `README.md` file when they start a project so that they can get familiar with the WebAssembly workflow. It provides the user with substantial amounts of control over how their source code gets compiled and how the WASM file is run. In addition, users who start projects on the site, are able to download the project and set up a local environment. The ability to download projects would allow my project to appeal not only to new developers but also to experienced ones by giving them the choice to download and work on their projects in a less controlled environment.

WASM Studio does have a particularly good editor experience, the site uses the Monaco editor [48] which is the same editor used by VSCode and provides a rich, native-feeling editor experience that developers who use VSCode will be familiar with.

WASM Studio lacks user authentication, therefore, if a user wants to continue working on a project once their session ends, they must download the project or export it to Github. If a user wants to work on the project using WASM Studio after this point, they must manually upload the files. This is a significant pitfall of WASM Studio which limits its usability as a platform to write real applications over longer periods of time. I will ensure in my application that users are able to save what they work on in a sensible way on the site, the user should be able to have multiple WASM projects that they can revisit and work on.

### 3.2.3 Advanced IDE Functionality

WASM Studio does allow you to see somewhat under-the-hood, by giving the ability to view compiled WASM modules in the WebAssembly Text Format (WAT), this is an important feature, both for debugging purposes and also for emphasising that the source language is not actually being run directly on the browser but is instead being compiled to WebAssembly which can then be executed on the browser. WASM Studio also provides users with a read-only 'output' terminal that can be used to view and debug compile time errors.

WASM Studio lacks the ability to infer type definitions from a compiled WebAssembly module. This feature would be extremely helpful for developers to be able to see suggestions for the signatures of functions exported from their WebAssembly module and one that would improve the developer experience of interacting with a WASM module using JavaScript.

## 3.3 AssemblyScript Editor

### 3.3.1 Overview

The AssemblyScript Editor [3] is a simple online playground which allows users to create WebAssembly websites using the AssemblyScript language [84]. AssemblyScript is a simple, TypeScript-like programming language which is exclusively used to compile to WebAssembly. The AssemblyScript editor allows users to create simple projects using WebAssembly, JavaScript and HTML. In addition, it allows for sharing of projects via links but lacks the ability for users to easily keep track of multiple projects. The AssemblyScript editor is commonly used for creating example projects for documentation

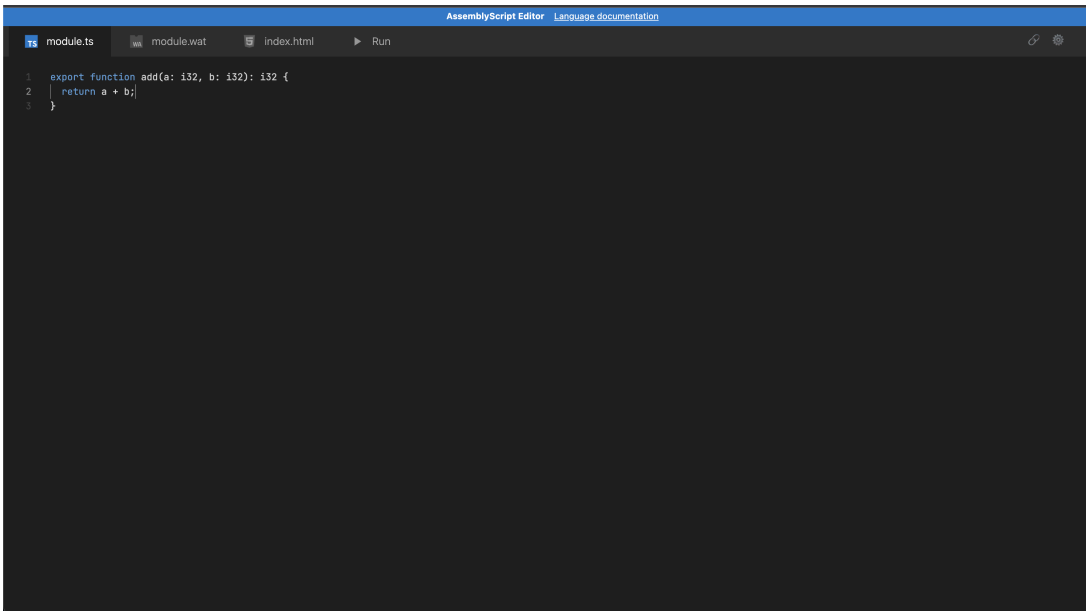



Figure 3.3: Screenshot of the AssemblyScript Editor with the `add` function written in AssemblyScript

purposes, as can be seen in the documentation of the AssemblyScript language itself. However, the editor itself comes with no examples and gives little indication to the user of what they are meant to do. Instead, a link to the language documentation for AssemblyScript is provided, which shows how to use AssemblyScript locally, but shows nothing about operating the editor.

### 3.3.2 Features and Limitations

The AssemblyScript editor does provide a great editor experience via the Monaco editor [48]. It does this by utilising the language's similarity to TypeScript to essentially extend the Monaco Editor's built-in TypeScript support to AssemblyScript. By including the type definitions for the entire AssemblyScript language and standard library, the user is able to see suggestions via IntelliSense for standard library functions such as `load` and `store` as well as for AssemblyScript types such as `i8` and `i16` [4]. This allows the user to move much faster than they would without these types being defined since this reduces the need to check the documentation for every standard library function because the signature for each standard library export is already shown in the editor. This project aims to provide a utility similar to this to improve the developer experience of using the IDE with intelligent suggestions.

The editor allows the user to work in a controlled environment whereby they may edit an AssemblyScript file and an HTML file, upon clicking 'Run', the AssemblyScript is compiled, and a preview window is shown with the HTML. In this way and others, it is somewhat unintuitive and difficult to use the AssemblyScript editor. Upon initially loading the editor, no example files are present to show the user how they can link their WebAssembly module to their HTML file, the editor relies on the user knowing how to do this. It is not immediately apparent when the editor is opened how you can call the compiled WASM module from JavaScript. In addition, it does show compiler errors if there were any but only after the user selects the `module.wat` file, no indication of compiler errors is given unless the user has selected their `module.wat` file.

The AssemblyScript editor does not provide the option to export a project to be worked on locally, this means it is helpful for toy projects or for demonstration purposes

but cannot be used to create more complex applications over a longer period.

The site does provide the ability to generate a link to a project so that it may be shared with others. This is a good feature for sharing small snippets, but it is unclear how long these URLs last so there is a possible risk of losing work.

The AssemblyScript editor provides users with a form with which they may choose and play with different options for the compiler, such as optimisation level and runtime type. This feature allows users to experiment with different compilation options and approaches to using and running their compiled WASM module. This feature provides users with a lot of flexibility to experiment and test out different ways of using AssemblyScript and WebAssembly in applications, in doing this the editor encourages the user to learn about the inner workings of AssemblyScript. This project aims when possible, to provide the user with adjustable compilation options as it will be extremely helpful as a learning experience for users to be able to experiment with different compilation options.

## 3.4 Stackblitz

### 3.4.1 Overview

Stackblitz [71] is a fully featured web-based IDE platform targeted at developing Node.js applications. Stackblitz is an industry leader in the online-IDE space, primarily due to the unique approach it took to solve the problem of creating full Node.js applications in the browser. As opposed to using a remote Node.js run-time that is interacted with by the client, Stackblitz created WebContainers [66], a WebAssembly-based operating system that enables Node.js to run natively from within the browser. The WebContainers project represents a huge step forward in the web-based IDE space, the ability to run Node.js inside the browser at native or faster-than-native speeds (up to 20% faster according to Stackblitz) is a completely novel concept and one that has only come about as WebAssembly has matured.

### 3.4.2 Relevance

Stackblitz is not oriented towards WebAssembly development, instead being mainly focused towards building Web Applications or APIs using JavaScript and Node.js, however, the advent of WebContainers is a clear and enormous step forward with a wide variety of applications. The relevance here, in addition to WebContainers themselves only made possible through the use of WebAssembly, is that it is possible to use tools such as Vite [98] to create a development server for the user to use and preview in the browser, furthermore, the AssemblyScript compiler is itself an NPM package that can be called on from within a Node.js environment. This presents the opportunity whereby users can create full websites that incorporate WebAssembly modules that are compiled from AssemblyScript, all from within a WebContainer's Node.js environment running inside the browser. This is an exciting prospect given that this would provide a complete environment with zero configuration for the user to develop a website using WebAssembly.

The Vite build tool provides a wide variety of features [97] that are incredibly helpful whilst developing web applications, these include Hot-Module-Replacement (HMR), which means Vite watches the project for file changes, and when a change does occur, the changed modules are updated at runtime so that the latest changes are represented, without needing to restart the entire server. Vite also provides out-of-the-box solutions for common front-end problems such as TypeScript [53] support and the ability to import CSS files (and many other types of static files) straight into your JavaScript and allow Vite's module bundler to ensure that these files are correctly included in build outputs. All these features, and many more make Vite a well-equipped build tool for new sites.

The WebContainers API was made publicly available [72] in February 2023. It utilises WebAssembly to run Node.js natively in the browser and relies on Stackblitz infrastructure to operate. Notably, utilising hosted proxies which expose running Node.js applications to clients, in addition to providing servers which resolve NPM packages for clients and cache commonly used ones.

For this project, we aim to analyse the use case for WebContainers in the context of a Web-Based IDE for WebAssembly and compare that to the more traditional approach taken by online IDEs of compilation and execution using an API on a remote server.

# Chapter 4

# Requirements Analysis

This project aims to solve the pitfalls of the current WebAssembly development process (as described in the Introduction and Related Works sections). The core aim is to provide users with a straightforward and fully-functioning IDE experience that is useful to both developers familiar with WebAssembly and those not.

## 4.1 Mandatory Requirements

1. Users must be able to easily create a new WebAssembly project which initially contains sample code to get the user started.
2. Users must be able to decide on a source language to use for compilation to WebAssembly. They may use either Go (which will use the TinyGo compiler [90]) or AssemblyScript [84]. Regardless of which language the user chooses, the experience of interacting with the compiled WebAssembly module from within JavaScript should be the same.
3. The website must be equally functional on all modern browsers (Chrome, Firefox, Safari, Edge)
4. The user must have the ability to edit HTML, CSS and JavaScript, alongside their chosen source language to create an interactive website.
5. Using JavaScript, users must be able to inter-operate with the compiled Web Assembly module by calling functions in JavaScript that are defined in their either Go or AssemblyScript source file.
6. The user must be able to preview the website they are creating from within a preview window in the IDE.
7. The IDE must include syntax highlighting for all relevant languages.
8. Users must be able to sign in securely to their account on the website.
9. Users must have the ability to save their work and come back to it later.
10. A project must by default only be viewable and editable by the user that created it.
11. If errors occurred in compilation to WebAssembly the user must be able to see what errors occurred and be alerted that compilation failed.
12. Compiled WebAssembly files must not be kept in storage any longer than they are needed by the user to prevent excessive storage costs.

13. The website must allow users to have multiple projects at once.
14. The website must give the user the ability to view the WebAssembly Text format [55] version of the compiled WASM file to give the user the ability to see how the compiled Web Assembly file operates.
15. The IDE must have a simple and intuitive user experience like that of a text editor such as that Visual Studio Code.

## 4.2 Desirable Requirements

1. The IDE may provide full language support for relevant source languages (Go, AssemblyScript, HTML, CSS, JavaScript).
2. The IDE may leverage the WebAssembly-based WebContainers Operating System [74] to provide the ability to create entire web applications using Vite [98] and AssemblyScript [84] without the need to interact with a remote server for compilation.
3. The IDE may allow users to download their entire project and start working on it in their own environment. The downloaded project should provide the user with a web server that serves their application locally.
4. The IDE may provide the ability to view the typed signatures of exported functions from a WebAssembly Module.
5. The IDE may allow users to share their projects with others.
6. The IDE may allow users to customize the interface to an extent, with options for colour scheme and font size.
7. The IDE may allow users to see JavaScript output from their application without the need to open their browser's developer tools. This would be done through an in-IDE 'console' window.

# Chapter 5

# Planning & Design

## 5.1 Use-Case Diagram

Using the requirements developed in the previous chapter (Chapter 4), a use-case diagram was created in order to define actions that users must be able to perform when using the IDE.

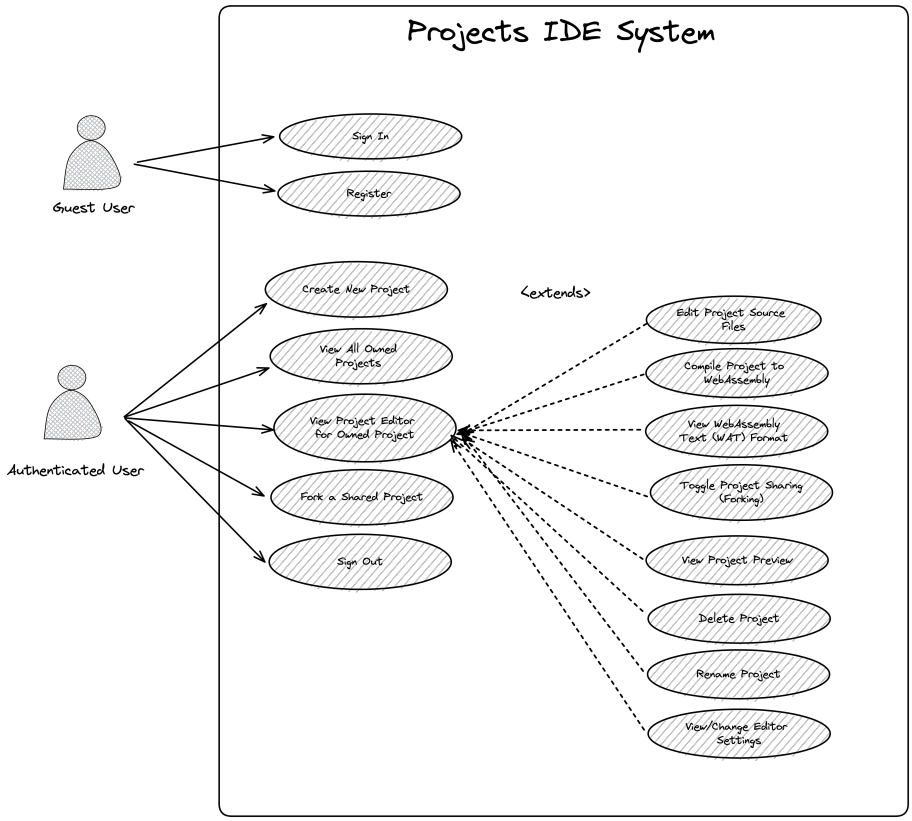



Figure 5.1: Use-Case Diagram for the WebAssembly IDE

In the use-case diagram (Figure 5.1), we distinguish between a 'guest user' and an 'authenticated user', whereby the only action that a guest user may perform is to authenticate themselves to the platform. Upon doing this, guest users become authenticated users and gain access to the actions available to authenticated users.

As illustrated in the diagram, an authenticated user is able to use the IDE system in its entirety. We can see that when an authenticated user performs the 'View Project Editor for Owned Project' action, a number of other project-editor-specific actions become

available to the user.

By developing this use-case diagram from the requirements we have created, we are able to begin to form a picture of how the application should operate from a user's perspective, this will help with the design and development of the application since we have defined the necessary actions that users should be able to perform from within a given context of the application.

## 5.2 User Interface Design

### 5.2.1 IDE Design

In designing the IDE for this project, we take inspiration from related (Chapter 3) tools and popular editors such as Visual Studio Code [54] and Stackblitz [71]. The default layouts of most popular, modern graphical IDEs tend to be quite similar and therefore it makes sense to follow a similar layout in order to avoid unnecessarily 'reinventing the wheel'. The following mockup (Figure 5.2) shows an example of a generic web-based IDE.

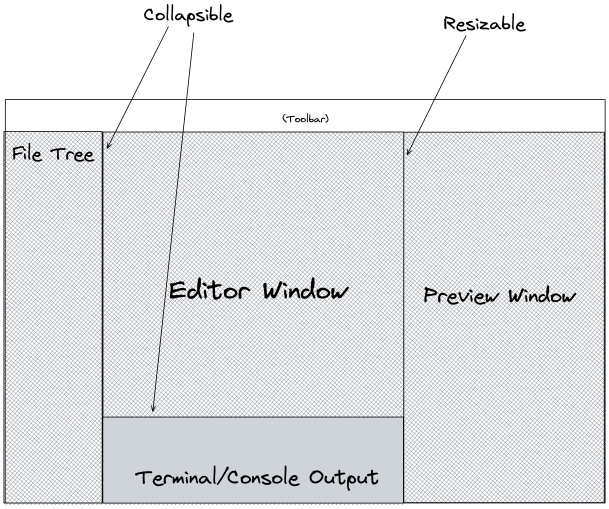



Figure 5.2: Mockup layout for a Generic Web-Based IDE for developing web applications

The mockup depicted (Figure 5.2) includes:

- A File Tree - Where users may select which file in their project they are viewing and editing in the editor window.
- An Editor Window - Where the user may view and edit the source code of the selected file.
- An Output Window - Where users can see the console output of their running application.
- A Preview Window - Where users can view and interact with their running web application.
- A Toolbar - Where users may quickly access actions and options provided by the IDE.

We adapt this generic web-based IDE mockup in order to produce a more detailed version that more directly meets the requirements (Chapter 4) we set out for the WebAssembly IDE.

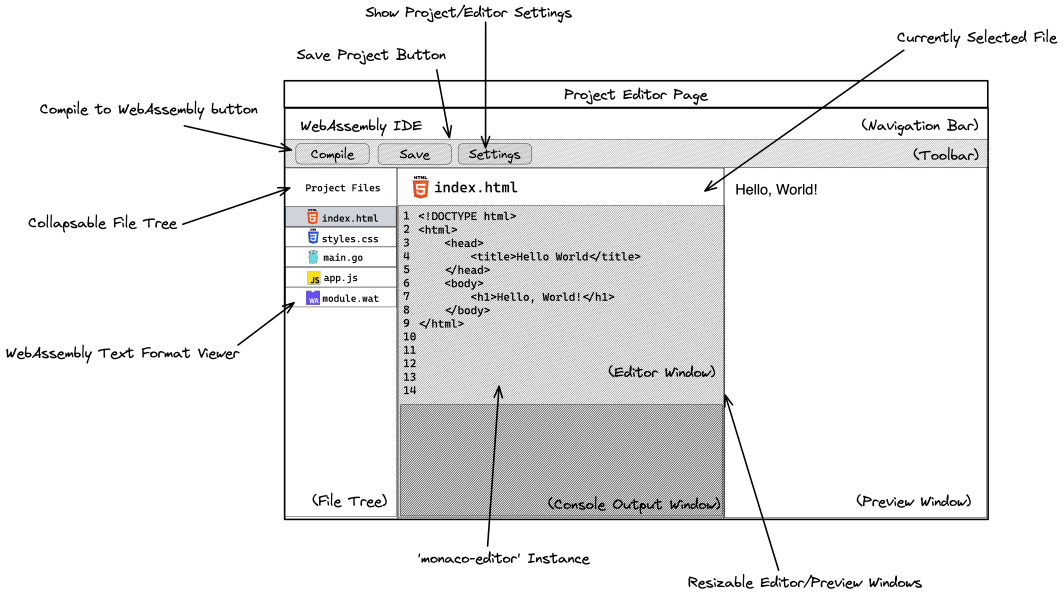



Figure 5.3: Detailed mock-up for the IDE developed to fit the requirements of the project

This mock-up (Figure 5.3) specifies our use of Monaco Editor [48] for the core editor experience. The Monaco editor provides a number of common utilities that text editors require such as search functionality, syntax highlighting and language support for a number of languages (including JavaScript, HTML, CSS and TypeScript), all of which will improve the productivity of developers operating the IDE.

### 5.2.2 User Interface Mock-ups

Using our user-case diagram, and with our editor's design in place, we can begin to visualise the entire application and its pages in mock-up form (Figure 5.4).

This diagram (Figure 5.4) illustrates how a user may authenticate to and begin using the WebAssembly IDE by creating new projects or editing existing projects.

The diagrams we have developed in this chapter will aid greatly in developing the implementation for this project. We now have properly defined the actions users of the

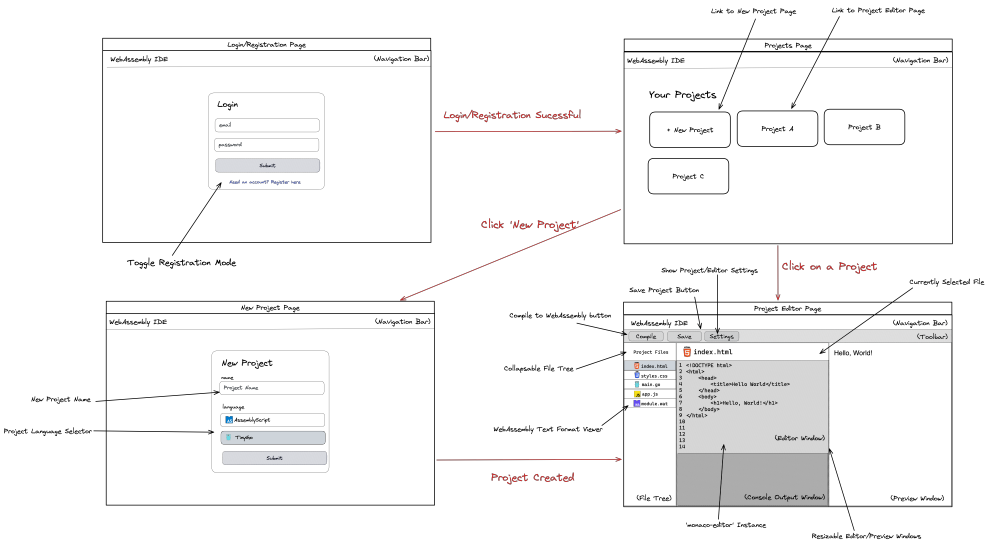



Figure 5.4: User Interface mock-ups for pages in the application

application must be able to perform in our use-case diagram (Figure 5.1). We have decided on key details for the layout of the IDE (Figure 5.3), and text editor. In addition, we have developed an idea of the exact pages that our application will need in order to fulfil the requirements we have set out. All of this will allow us to develop the system in a user-facing way. We will then incrementally improve and iterate upon [43] the implementation to adapt to new requirements that will be found by testing and using the developed application to create projects with WebAssembly.

# Chapter 6

# Implementation

## 6.1 Implementation Overview

In this section, we discuss the implementation details and architecture for the WebAssembly IDE. This section is separated into three parts, the first two will discuss the 'projects' system for the IDE, whereby users sign in, create a project and interact with an API that will perform compilation to WebAssembly for them. The last section will discuss an alternative IDE implementation, the 'WebContainer Playground' which makes use of the WebContainers API [74] to be able to run Node.js in the browser and thus provide the user with a full development experience without the need to perform compilation on a remote server.

## 6.2 Projects API

### 6.2.1 Overview

The Projects API, written in the Go programming language is responsible for Authentication, Project Storage and compilation to WebAssembly. In this section, We discuss in depth the details of the API and how it is designed. The API uses the 'gin-gonic' [35] web framework as a basis for handling requests and registering middleware.

### 6.2.2 Authentication

Authentication in the API uses an email and password scheme, whereby passwords are hashed and salted in storage to prevent database intruders from accessing them. To authenticate clients, such as the web application to the API, a Bearer Authentication scheme was used [77]. Upon registering for an account or logging in to their account using the `/auth/login` or `/auth/register` routes, the client is provided with a JSON Web Token (JWT) [40] set to expire 24 hours after creation. The client is subsequently expected to use the 'Authorization' header on each subsequent request they make to the API. The Authorization header sent by the client on new requests must be of the form: `Authorization: Bearer <JWT Generated at Login/Registration>`

Authentication for protected pages is guaranteed using the `Protected` middleware found in the `auth` module, which is used to wrap routes that require a user to be logged in to use them. Furthermore, service level operations in the API perform checks to ensure that the user requesting to read/write data is responsible for (i.e. has created) the resource that they are attempting to read or write to.

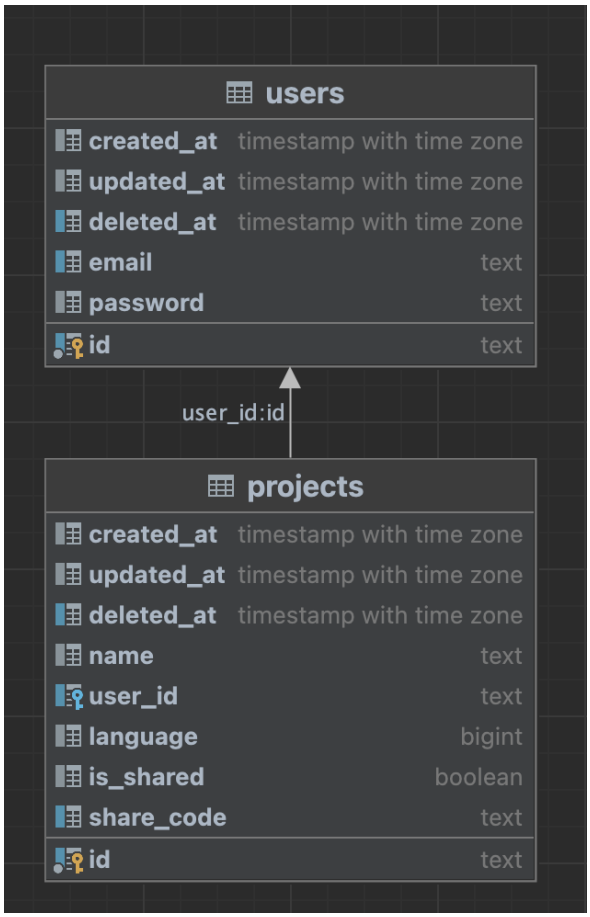



Figure 6.1: Entity-Relationship-Diagram for Postgres Database used by the API

### 6.2.3 Storage

The storage layer of the API uses a few different tools to facilitate the storage of projects and project files. The first of these is PostgreSQL [37] using the 'GORM' Object Relation Mapper [41]. The database schema itself is extremely simple, with only two tables: for storing user information for authentication and for storing references to projects made by users.

The files related to a project itself are stored in an AWS-hosted S3 bucket [11] accessed with the AWS Go SDK [9]. At the service level, when a project is created using the `CreateProject` method, a corresponding folder [10] will be created in the bucket with the default files for the compiler they have chosen (TinyGo or AssemblyScript). The folder will be given the unique key `<user_id>/<project_id>/`, this ensures there is no overlap in projects since these fields are guaranteed to be unique within their respective tables by Postgres.

Within each project folder in the S3 bucket, two subfolders are used, the first being `src/`, which is the location in which the user's source code for the project is stored (i.e, HTML, CSS, JavaScript and Go/AssemblyScript source file). When a project is created, the source files are stored here. The second folder is the `build/` folder which stores build artifacts generated by WebAssembly compilation such as the WASM module and the WAT [55] file for the project.

When a user saves their project, the changes are uploaded to the `src` folder of their project in S3, with each file being updated concurrently using goroutines [7]. Once a user requests their source code be compiled, the source code within the S3 Bucket for the user's project is compiled and a WASM module is stored for the project inside the `build` folder

of the project, alongside any other auxiliary files related to compilation (e.g. WAT file). These files are also uploaded concurrently to the bucket. After successful compilation, the server responds with a presigned URL [12] generated for the user pointing to their compiled WASM module. This means the WASM module may be accessed by the client without the need to send the entire WASM module first back to the API and then to the client.

In order to inject these storage-level operations into the service level of the API, dependency injection was used alongside the repository pattern [51], which, by injection of singleton instances of the database connection and the S3 Service allows services to interact with the storage layer.

### 6.2.4 Compilers

In order to give the user the ability to compile a source language to WebAssembly, the API requires the ability to call on different compilers that can compile source languages to WebAssembly. For this project, there are two source languages that the user may compile to WebAssembly, therefore two different compilers were required: TinyGo [90] and AssemblyScript [84]. The relevant compiler for a given project is called when a client makes a `POST` request to the `/projects/<id>/compile` endpoint on the server. Upon successful compilation, this endpoint always responds with a generated pre-signed URL pointing to the newly stored compiled WASM module in the project's S3 `build` bucket. The compiled WASM module stored in the S3 folder is always named `main.wasm`, this prevents the use of more than one WASM module per project.

#### TinyGo Compiler

The TinyGo compiler is used so that users may compile a Go program to WebAssembly for execution in the browser. TinyGo [90] is an alternative to the standard Go compiler for WebAssembly [63] that aims to produce extremely small and portable binaries from Go source code, it is popular for use on embedded devices but also supports WebAssembly as a compilation target. In compiling to WebAssembly, we aim to produce as small a binary as possible, in order to reduce storage costs as well as to create a quick feedback loop for developers. For this reason, the TinyGo compiler was chosen over the traditional Go compiler for compilation to WebAssembly. The TinyGo compiler is capable of producing far smaller binaries than that of the standard Go compiler, it does this by removing certain features that aren't always needed when writing Go programs, such as the intermediate linking step of compilation [89]. Through research and exploration, TinyGo was found to produce a substantially smaller baseline WebAssembly binary of ~30kB compared to the standard Go compiler [using flags for WebAssembly] (~1MB) when using the `syscall/js` library to create a JavaScript alert in the browser. This research showed that TinyGo was the obvious choice since it will reduce the amount of storage required to keep these compiled binaries. In addition, it would be simply impractical to send 1Mb of data over the wire every single time a user wishes to compile/recompile their project to WebAssembly.

To make the compiler work from within the server's environment, it had to be installed on the server at which the API is running, this is done at deployment with a `RUN` command inside the Dockerfile, which allows this installation step to be cached.

Since the compiler is installed on the server, we can now call on TinyGo for compilation by spawning a child process. When the endpoint for compilation is hit, the Go code relating to the project is pulled down from the S3 bucket inside `<project_id>/src/main` and stored in a temporary directory. It is then compiled using the `tinygo` command-line tool inside of a child process on the server.

```
## Run tinygo install script
RUN wget https://github.com/tinygo-org/tinygo/releases/download/v0.26.0/tinygo_0.26.0_amd64.deb && \
  dpkg -i tinygo_0.26.0_amd64.deb && \
  rm tinygo_0.26.0_amd64.deb && \
  tinygo version
```

Figure 6.2: Docker RUN command used to install TinyGo in the APIs deployment environment.

The TinyGo compiler, given Go source code, is able to detect exported functions with the `//export <name>` comment above function definitions. These functions are exported by TinyGo into a WASM module where they may be called by JavaScript. In addition to this, the `main` function, which is a required function for all Go programs (that compile to a binary) is automatically exported by TinyGo into the module. The compiled WebAssembly module tends to be quite large even when used for basic tasks, this is due to the need to include parts of the Go runtime inside the binary so that the module can operate in an isolated context with WebAssembly.

The process exit code and standard error output can be read in order to check for compile-time errors and send them back to the user accordingly. Upon successful compilation, the output file(s), which are created in a temporary directory on the server get read into a `CompileResult` struct, and the original temporary directory is deleted. The `CompileResult` struct may then be further manipulated, or as in the case for compilation to WASM on the server, stored in S3 files inside the `<user_id>/<project_id>/build` folder, in order for the files to be fetched by the client using a generated pre-signed URL.

### AssemblyScript Compiler

AssemblyScript [84] is a TypeScript-like language [53], designed to be compiled into WebAssembly. It allows developers to write low-level, memory-efficient and high-performance code, which, when compiled to WebAssembly can be safely run in the browser, server or embedded devices at native or near-native speeds.

When compared to other low-level languages used for WebAssembly, such as Go, C++ and Rust, AssemblyScript provides developers with a relatively simple TypeScript syntax and type system in order to develop for the platform. This is ideal for those initially unfamiliar with high-performance web development in WebAssembly since many of these lower-level languages can have a higher learning curve. In addition, the AssemblyScript compiler provides a number of tools to make the development process for WebAssembly more straightforward, this includes the ability to output both WebAssembly (`.wasm`) binary files as well as WebAssembly Text Format (`.wat`) files at compile-time.

In order to be able to use the AssemblyScript compiler on the server, it had to be installed within the Docker image which is used to build and run the server. This was done in much the same way as how the TinyGo compiler was installed, however, since AssemblyScript is available as an NPM [28] package, this also required installing Node.js on the server so that it was then possible to install the AssemblyScript compiler.

Once installed, the server can now call on the installed `asc` command-line tool as a child process. When compilation is requested, the tool is called on as a child process and used to generate both the WASM module file (using the `outFile` flag) and the corresponding WAT file (using the `textFile` flag). We check the process exit code and the standard error output to determine if compilation errors occurred and send them back to the user in a sanitized format if they did.

```
RUN curl -fsSL https://deb.nodesource.com/setup_lts.x | bash - && \
  apt-get install -y nodejs &&\
  node -v && \
  npm -v && \
  npm install -g assemblyscript && \
  asc --version
```

Figure 6.3: Dockerfile RUN command used to install the AssemblyScript on the server, this required installing Node.js first and then installing the compiler using NPM.

### 6.2.5 WebAssembly Text Format

WASM Text Format (WAT) [55] is a textual representation of the WASM binary format used mostly for debugging and demonstration purposes. The corresponding WAT file is generated after compilation is completed for a given project and uploaded concurrently alongside the compiled WASM module in the `<user_id>/<project_id>/` folder in the S3 bucket.

As noted in the previous section, the AssemblyScript compiler supports generating WAT files out of the box using the `textFile` flag. We use this feature with AssemblyScript projects and upload WAT and WASM files concurrently to S3 upon successful compilation.

TinyGo does not support compilation to both WASM and WAT out of the box, therefore, the server required a way to translate WebAssembly modules to the WAT format. This was accomplished by making use of the WebAssembly Binary Toolkit [107], which was installed on the server using a `RUN` command inside the Dockerfile for the server with the `apt-get` package manager. This process is then spawned as a child process after the successful compilation of a TinyGo file to WebAssembly and then subsequently uploaded to S3 as described.

A successful `GET` request to `projects/<project_id>/wat` from an authenticated user owning the project referenced will return a generated pre-signed URL which points to the WAT file uploaded at compile time. This URL may subsequently be used on the client side to retrieve the contents of the file.

### 6.2.6 Sharing

The API allows users to share their projects with others by creating a 'fork' of a project. Users may opt to share their project by sending an authenticated `PATCH` request to the `/projects/<project_id>/share` endpoint. This endpoint will create a unique 8-digit 'share code' which will enable other users to find and fork the project on their own account.

Any user may retrieve a shared project by sending a `GET` request to `/projects/fork/<share_code>`. A shared project can then be forked to create a copy of it by sending a `POST` request to `/projects/fork/<share_code>`. When this happens, the original project that the share code refers to will be cloned into a new project owned by the user making the fork request, this new project will contain all the files from the original forked project and will allow the user forking the project to update and make changes to the project as if it were there own. A forked project shares the name of the project that was copied to create the fork, except with '(fork)' appended to the name.

### 6.2.7 Deployment

The Projects API and Postgres Database are both deployed on the Railway [61] platform. Railway allows for the provisioning of PostgreSQL databases with zero configuration [60], using this platform allowed the focus during development to stay on creating features and

fixing issues, as opposed to needing to maintain the infrastructure for the storage layer and API, which would have been a considerable task in itself.

Deployment of the Go server required ensuring that a number of compilers and tools were installed on the server (TinyGo, AssemblyScript Compiler, WebAssembly Binary Toolkit). In order to make the process of deploying the application simple, a Dockerfile was created to represent the build step of the application, this Dockerfile is then subsequently used by Railway to build a binary for the API, and then to run it. In addition, efforts were made to ensure the size of the built Docker image was relatively small [42], this was difficult due to the need to include both the Node.js and TinyGo runtimes inside the Docker image so they may be used by the API at runtime, however, by disabling the `CGO` package at build time, we are able to dramatically reduce the size of the server's Docker image from approximately 2GB to approximately 750MB.

```
FROM amd64/golang:1.19.2-bullseye


# Installing Node.js and AssemblyScript
RUN curl -fsSL https://deb.nodesource.com/setup_lts.x | bash - && \
    apt-get install -y nodejs &&\
    node -v && \
    npm -v && \
    npm install -g assemblyscript && \
    asc --version

## Installing TinyGo
RUN wget https://github.com/tinygo-org/tinygo/releases/download/v0.26.0/tinygo_0.26.0_amd64.deb && \
    dpkg -i tinygo_0.26.0_amd64.deb && \
    rm tinygo_0.26.0_amd64.deb && \
    tinygo version

## Installing the Web Assembly Binary Toolkit
RUN apt-get update && \
    apt-get install wabt


WORKDIR /app
COPY go.mod .
COPY go.sum .

COPY . ./

RUN CGO_ENABLED=0 go build -ldflags="-s -w" -o=./api


CMD ["./api", "serve"]
```

Figure 6.4: Complete Dockerfile used to containerise the API into a Docker image

Railway's Github Triggers [62] are used to automatically deploy the API whenever a change was made to the repository hosting the API on Github. Once the Dockerfile was correctly set up, this allowed us to never need to worry again about the infrastructure or deployment process of the API, since a continuous deployment pipeline had been established. As well as the features mentioned above, once the deployment pipeline was successfully set up so that the server was listening for connections inside its containerised environment, Railway provides the application with an HTTPS-only domain on their platform which allows the API to be used by clients.

In order to secure the API from requests that it shouldn't accept, a `CORS` policy [21][34] was used whereby the only client allowed to send requests to the deployed API is the web application for the IDE. If any other client attempts to make requests to it, they will be rejected due to an incorrect `Access-Control-Origin` header when the 'preflight' CORS `OPTIONS` request is sent.

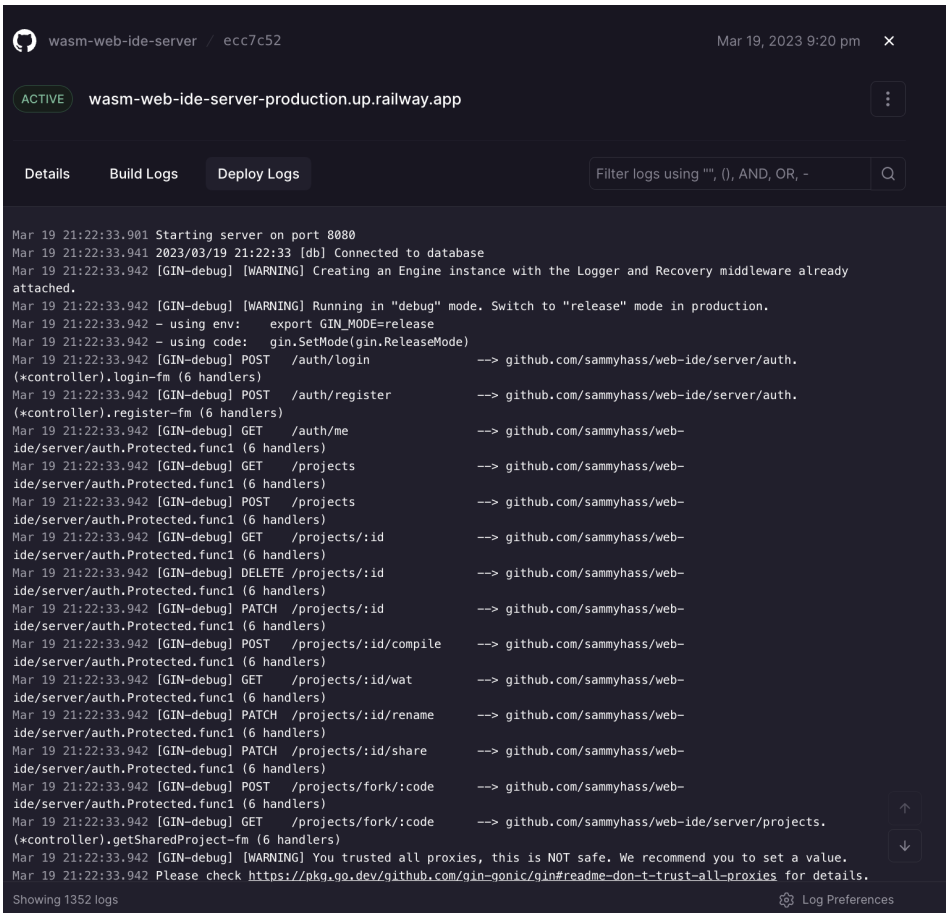



Figure 6.5: Deployment logs in Railway. The deployment was created after pushing an update to the GitHub repository hosting the API code

## 6.3 Projects Web Application

### 6.3.1 Overview

This project uses the Next.js [94] framework with TypeScript as the basis for the IDE web application. Next.js is a full-stack, production-grade React [30] [86] meta-framework. Next.js utilises SWC [95], an extremely fast Rust-based compiler, transpiler and bundler for TypeScript and JavaScript for its compilation process. SWC allows Next.js to produce builds extremely quickly as well as create a quick feedback loop for development due to the speed of the hot-module-replacement system. Next.js has a wide variety of features which make it the perfect choice for building performant, modern web applications using React and TypeScript, furthermore, being built on top of React allows us to draw from the vast existing ecosystem of packages and libraries built for React.

In order to style the application, we make use of TailwindCSS [31] and DaisyUI[65]. TailwindCSS was used as a tool for rapid prototyping during the design phase. DaisyUI was used as an extension to Tailwind to provide some base components that the web application could use and customise (such as buttons and form inputs). DaisyUI was used instead of having to create an entirely new UI library using TailwindCSS which would have required the creation of an entire-formalised design system.

### 6.3.2 Data Fetching

In order to allow the web application and backend API (Section 6.2) to communicate over HTTP(S), we made use of the 'axios' [16] npm package. 'axios' is a promise-based HTTP client which can be used from within client-side applications. The 'axios' package is used throughout the application to facilitate communication with the API. In addition, we make use of the 'zod' [14] validation library to ensure that responses from the API are in the expected form and are type-safe.

In order to keep track of the user operating the site and respond appropriately to their requests, we use interceptors [15] to inject the user's JWT, (which is stored in `localStorage` upon successful login/registration) into the 'Authorization' header for requests from the 'axios' client.

In order to perform caching and manage asynchronous state in the web application, we opt to use the 'TanStack Query' [82] package. This package is used to wrap our Axios API requests, as well as all other asynchronous operations in the web application and allows us to call these operations from within custom React hooks [87]. 'TanStack Query' provides us with a `QueryClient` [83] that maintains a cache of all asynchronous operations we perform in the web application, such as requests sent to the API. In using 'TanStack Query' with its `useQuery` [80] and `useMutation` [79] hooks for handling asynchronous operations and state, we are provided with a caching layer that ensures that we don't over-fetch from the API, in addition to ensuring that components are always provided with the most up-to-date version of the data they require. Furthermore, 'Tanstack Query' allows us to have fine-grained control over cache invalidation [81] and cache updates. This gives us the ability to mutate the query cache optimistically or respond to successful/failing mutations by updating the cache for queries that may not be related to that specific mutation. 'TanStack Query' also provides us with developer tools [78] that can be used to view and debug the asynchronous state stored inside the `QueryClient` cache.

### 6.3.3 Project Management and Creation

Once a user is succesfully authenticated, they are redirected to the `/projects` page. Here users can see all of their existing projects, which are retrieved using 'axios' and cached with the `QueryClient` described in the previous section. Users may click on any of their existing projects to open the project editor so they may view a preview and make changes to it.

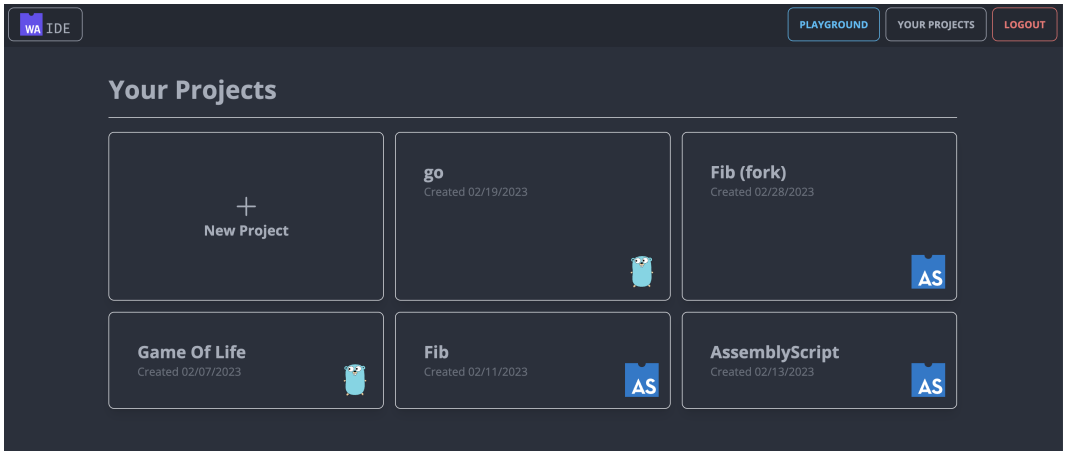



Figure 6.6: Projects page showing the current user's existing projects. The user can click on any of their projects to create a new one or click the 'New Project' button to navigate to the project creation page

Authenticated users may create a new project by navigating to the `/projects/new` page. Here they must name their new project as well as choose which compiler they want to use for compilation to WebAssembly, either TinyGo or AssemblyScript (Section 6.2.4).

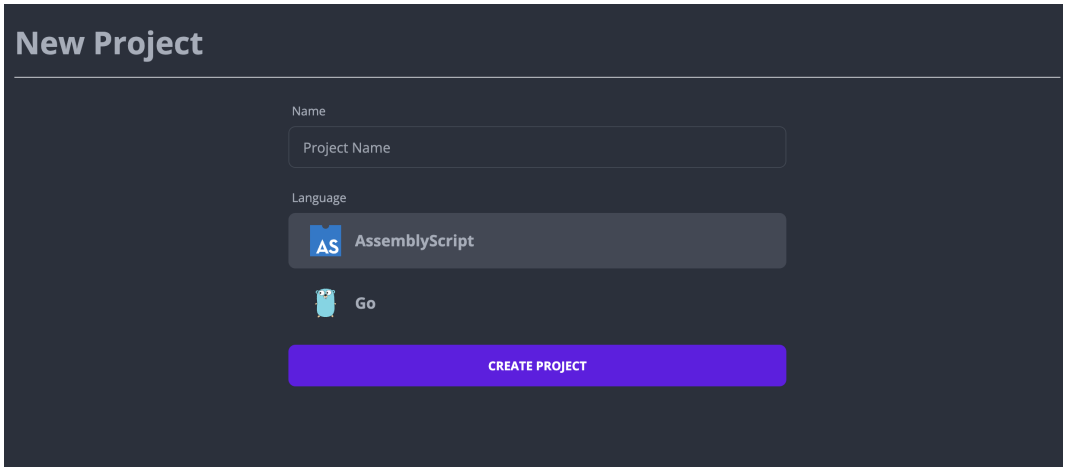



Figure 6.7: (Cropped) Project creation page, the user must name the project (they may change the name later), as well as select a language to use as a source for compilation to WebAssembly

Upon form submission on the project creation page, a `POST` request is sent to the API on the `/projects` route using our data fetching (Section 6.3.2) client, which in turn, creates a record in the Postgres table to represent our project, as well as scaffolding out a default template project based on the selected project language inside the S3 bucket (Section 6.2.3). Once the project has been created, the new source files created for the project are sent back to the client, as well as relevant storage information, this data is subsequently cached by the `QueryClient` under a unique key so it may be accessed by the editor and projects page. Once this process has been completed, the user is redirected to the editor page at the `/projects/<id>` route where they can edit and preview their project.

### 6.3.4 Editor

The project editor page on the `/projects/<id>` route provides a user with everything they need to edit, compile and preview their WebAssembly project. When the editor page is requested, the relevant project is fetched from the web applications cache, or from the API (provided the project belongs to the currently signed-in user) and is loaded into a 'Zustand' [59] store (the project editor store) where it can be accessed and updated using the `useProjectEditor` hook.

**Monaco Editor** The project editor makes use of the Monaco editor for the core editor experience [48]. Here we use the '@monaco-editor/react' [6] NPM package to provide us with a React-based editor instance for Monaco.

Monaco was the clear choice for the editor due to its popularity amongst existing web-based IDE tools as well as its ease of use. By using the Monaco Editor, we are providing users with the same familiar editor that runs Visual Studio Code [54]. Monaco also provides us with a variety of options to augment or alter the editor experience, this includes a wide range of settings that may be configured by the developer.

Users can navigate between the four different source files in their project using the 'file tree' located to the left of the text editor.

**Editor Customisation** Users are provided with options for editor customisation within the Settings modal, which can be revealed by clicking the 'Settings' button in the editor toolbar. Users can opt to change the font size in the editor, toggle the theme between light and dark, choose whether they want word wrap to be enabled, and choose whether they want a mini-map to be shown. These editor settings are stored in a store using 'Zustand', separate from the project editor store, these settings are persisted by the web application `localStorage` [26] and are used for all editors on the site.

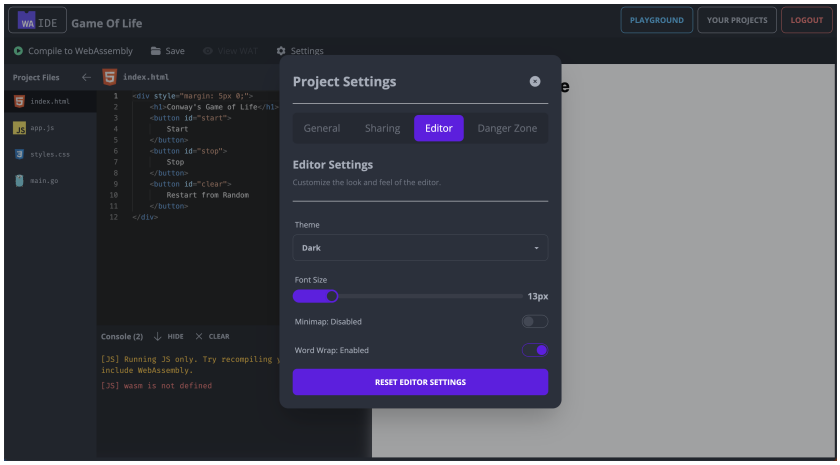



Figure 6.8: Customisation options for the editor

In addition to these settings, the editor itself is horizontally re-sizable, this means that users can adjust the size of the editor and preview windows relative to the other depending on the user's specific focus at a given point. The provided editor options and resizing functionality provide users with a substantial amount of freedom to customise the way the editor looks and feels.

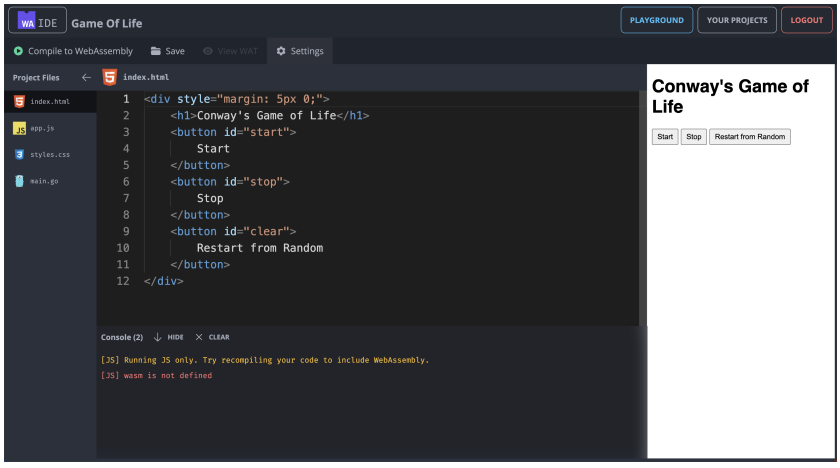



Figure 6.9: Editor page with resized editor window and larger-than-default font

**Editing, Saving and Compiling Projects** When a file is updated inside of a project, the editor's state is marked as being 'dirty', meaning that it is not up-to-date with the project inside the S3 bucket. If a user attempts to click the 'Compile to WASM' button inside of a dirty project, they will be prompted to save their project before compilation can be performed so that the project's source code is up-to-date with S3.

When clicking the 'Save' button in the toolbar, the user's project source files will be synchronised with those in their S3 project folder via a `PATCH` request to the API

(Section 6.2.3). Once the user's source files are saved and synchronised with S3, the user may request their project's source code (either AssemblyScript/TinyGo) be compiled using the relevant compiler for their project (Section 6.2.4) by sending a `POST` request to the compilation endpoint on the API.

**Language Support** The Monaco editor provides full out-of-the-box language support for JavaScript, TypeScript, HTML, CSS and JSON. This means users are provided with auto-completions and type definitions when using these languages. We make use of this language support to provide users with auto-completion and in-editor documentation for JavaScript, HTML and CSS files in the editor.

In order to support syntax highlighting and completions for the AssemblyScript language from within the editor. We make use of the fact that AssemblyScript itself is a variant of the TypeScript language, which is supported by the Monaco editor. This means that when editing an AssemblyScript file, we must ensure that the editor knows that the AssemblyScript file is also a TypeScript file so we can make use of the TypeScript support.

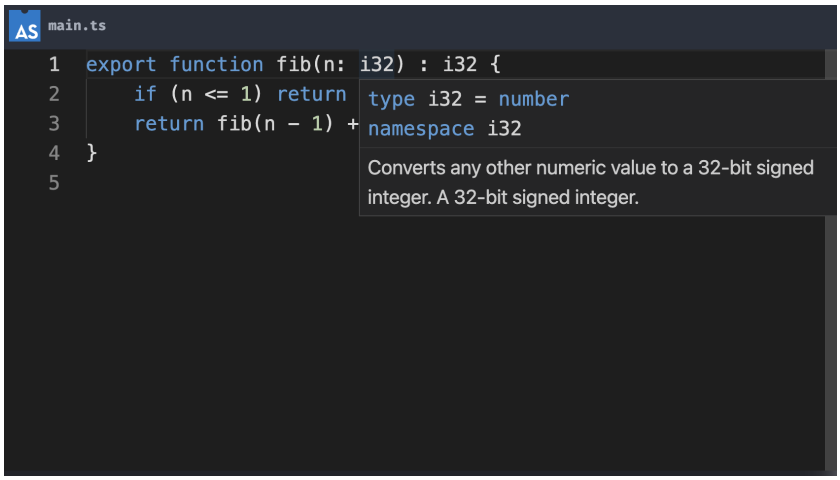



Figure 6.10: Users can hover over AssemblyScript types or standard library functions, such as the `i32` type to view the documentation for that type/function.

AssemblyScript types and standard library functions do not exist in TypeScript. In order to ensure that all AssemblyScript types and standard library functions would be easily viewable by users via type-hints and auto-completions we utilise a TypeScript type declarations file [45] which contains the TypeScript type definitions for all AssemblyScript types [4] and standard library functions. This declarations file was found by inspecting the open-source code that operates the AssemblyScript Editor [3] discussed in the Related Works chapter (Chapter 3). By using the Monaco editor API, we are able to add the type declarations file for AssemblyScript to the editor's TypeScript environment. This allows users to view documentation for AssemblyScript methods and types when they hover over them (Figure 6.10), as well as providing users with type hints for standard library functions (e.g. `load` and `store`) and AssemblyScript types (e.g. `i32` and `i8`) within the editor via auto-complete. Furthermore, with these type declarations, we are able to provide a type-safe editor experience for AssemblyScript. AssemblyScript type errors will be spotted by the editor's built-in TypeScript client and shown to the user with red underlines in the editor without the user needing to compile their AssemblyScript to WebAssembly.

The Monaco editor provides basic syntax highlighting for projects written in Go, however, full-language support for Go would require the use of an externally hosted server [47] running the 'gopls' [85] language server. This option was considered but proved to

be too complex of an undertaking due to time constraints and changing requirements for this project.

In addition to AssemblyScript type declarations, we provide additional type declarations for the editor's JavaScript environment. These are the `wasm` and `memory` variables. The `wasm` variable represents the loaded WebAssembly instance [24] that the user may call on from JavaScript. The `memory` variable contains the raw memory as an unsigned integer array that is used by the WebAssembly instance during its runtime. These definitions are provided regardless of the chosen project language and the variables that these types reference are instantiated by the preview window, which will be discussed in the following section. Loading these type declarations into the editor allows the user to see the types of the important WebAssembly-related variables (`wasm` and `memory`) when hovering over them in the editor, or when using auto-complete. These types were added in much the same way as the AssemblyScript types were added, with the only difference being that these types are added to the editor's JavaScript configuration, whilst the AssemblyScript types are added to the editor's TypeScript configuration.

### 6.3.5 Previews

**Overview** The preview window for the project editor uses an HTML iframe tag [22] to embed the user's HTML into a nested browsing context contained within the project-editor HTML page.

The HTML content of the preview window is set using the iframe `srcDoc` attribute. The content contained in the preview is defined by the editor's save state, meaning that once a user saves their project, the preview window will be automatically updated by the client. Taking the user's source files and composing them into a preview is performed entirely client-side. Composing previews on the client prevents the need to perform any other network calls following a user successfully saving their project.

**Preview Source Document** We inject the user's client-side source code (HTML, CSS, JavaScript) into a simple template HTML document for the iframe to show the user the preview of their application.

As can be seen in the image (Figure 6.11), we create the HTML preview document by interpolating the content of the user's HTML file into the body of the template. If the user is using Go (TinyGo) [90] as their project's source language for compilation, we inject a script tag that references the bindings needed to ensure that the user's WASM module produced by TinyGo can properly interact with JavaScript inside of the browser context. It is important to note that all source code provided as strings are escaped automatically by React, which will help prevent the risk posed by Cross-Site Scripting.

**WebAssembly Module Instantiation** As can be seen in the `iframeContent` function (Figure 6.11), we utilise a `runWasm` function (Figure 6.12) which is responsible for fetching and instantiating the WebAssembly module stored at `wasmPath`.

The `runWasm` function (Figure 6.12) is used to generate the JavaScript for the preview window. If provided a `src` argument for a WebAssembly module, we instantiate the `memory` variable with a value that the WASM module will understand at run-time (based on its source language for compilation). We then fetch and instantiate the module using `WebAssembly.instantiateStreaming` [25]. Once the module has been successfully instantiated, we set the value of the `wasm` variable to the WebAssembly instance [24] produced by `instantiateStreaming`. We then allow the user's JavaScript to run (which may contain references to the in-scope `wasm` and `memory` variables) thus allowing the user to interact with their WebAssembly module using JavaScript. In the scenario that no

```
export const iframeContent = ({
  html,
  css,
  js,
  wasmPath,
  nonce,
  useGo = true,
}: {
  html: string;
  css?: string;
  js?: string;
  wasmPath?: string;
  useGo?: boolean;
  nonce?: string;
}) => `
  <!DOCTYPE html>
  <html lang="en">
    <head>
      <meta charset="UTF-8">

      <meta name="viewport" content="width=device-width, initial-scale=1.0">
      <style data-testid="preview-css">
        ${css}
      </style>
      ${useGo ? `<script src="/wasm_exec_tiny.js"></script>` : ``}

      <script type="module" defer data-testid="preview-script">
        (() => {
          const __nonce = '${nonce}';
        })()
        ${consoleReassign}
        ${runWasm(js, wasmPath, useGo)}
      </script>
    </head>
    <body data-testid="preview-body">
      ${html}
    </body>
  </html>
`;
```

Figure 6.11: `iframeContent` produces an HTML document using a provided WebAssembly path and HTML, CSS and JavaScript code.

WebAssembly module path was provided to the function, the JavaScript code provided is all that is returned and then run in the preview window.

Now that the user's saved files are correctly included and used by the preview window, users can view the preview of their project from within the IDE (Figure 6.13) and view changes after saving and compiling it.

```
const runWasm = (js?: string, src?: string, isGo = true) =>
  !!src
    ? `

  let wasm = null;

  ${instantiateMemory(isGo)}

  WebAssembly.instantiateStreaming(fetch('${src}'), ${
        isGo ? 'memory.importObject' : getAssemblyScriptImports()
      }).then(___result => {
    wasm = ___result.instance;
    ${isGo ? 'memory.run(wasm);' : ''}
    ${runJS(js || '')}
  }).catch(e => {
    console.error(e)
  });
`
    : `
    console.warn('Running JS only. Try recompiling your code to include WebAssembly.')
    ${runJS(js || '')}
    `;
```

Figure 6.12: `runWasm` function for fetching and instantiating WebAssembly modules in preview windows.

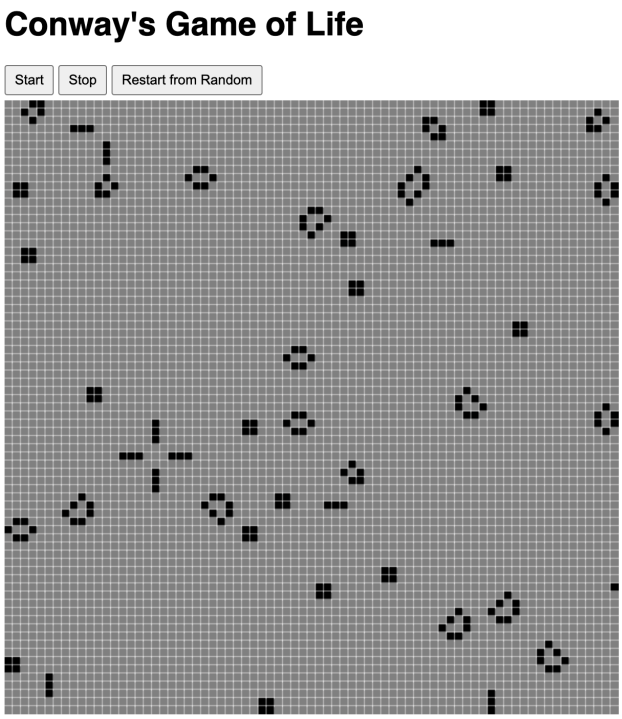



Figure 6.13: Preview window for a TinyGo project implementing the Game of Life.

**Redirecting Console Output** In the `runWasm` function, we use the preview frame's `console` to log messages out to the browser. In order to show these messages to the user in the IDE's console window, we must implement a secure method of communication between our application's main frame and our nested preview frame. This can be achieved using the `window.postMessage` [27] method which enables frames to securely communicate by message-passing. To enable this behaviour, we must first override the console in the preview frame so that it sends a message to the parent window when run (Figure 6.14).

For our main application frame to be able to receive the messages sent from the preview window, we must add an event listener that listens for the 'message' event and pushes the console message contained in received events to the console display in the IDE. With this event listener implemented, we can view console messages sent by the preview window from within our editor's console window.

```
export const consoleReassign = `
const postMessageToParent = (type, data) => {
  window.parent.postMessage({ type, data }, '*');
};

${['log', 'error', 'warn', 'info', 'debug']
  .map(
    method =>
      `console.${method} = (...args) => {\n\n\tpostMessageToParent('console', ['${method}', '[JS]', args]);};`
  )
  .join('')}
`;
```

Figure 6.14: Console methods are reassigned in the preview frame to direct them to the console window in the frame's parent window.

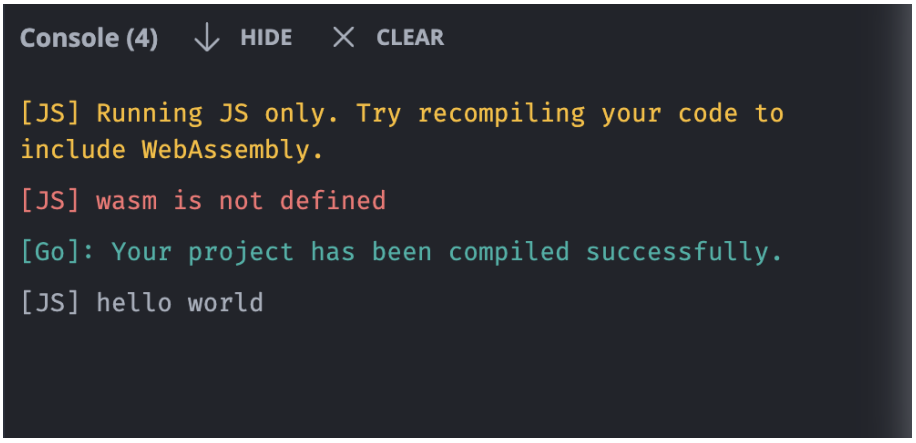



Figure 6.15: Console Window displaying combined output for processes run in the IDE.

In the console window (Figure 6.15), we distinguish where each message comes from with the language preceding the log. '`[JS]`' indicates the message is from the preview window's JavaScript console output. '`[AssemblyScript]`' or '`[Go]`' indicate that the message is related to the compilation of the given source language to WebAsssembly, e.g. a compilation error.

### 6.3.6 WAT Viewer

We can retrieve a presigned URL [12] pointing to the WebAssembly Text Format (WAT) representation of a project's WebAssembly module by sending a `GET` request to the API's `/projects/<project_id>/wat` endpoint. Then, in order to retrieve the content of the file itself, we can send a `GET` request to the pre-signed URL returned by the previous request. The results of both of these requests are stored in the query cache [83] maintained by our data-fetching layer.

Provided the presigned URL stored in the query cache [83] is valid, there is no need to fetch the presigned URL pointing to the WAT file after initially fetching it. This is because the URL only points to a location in the S3 bucket so changes to the content of the file will be seen when the file is fetched with the pre-signed URL.

Once a project is compiled for the first time in a session, the option to view the WebAssembly Text (WAT) format version of the compiled module becomes available to the user via the toolbar (Figure 6.16). The 'View WAT' button may be clicked to open a dialogue modal containing a read-only Monaco editor [48] instance displaying the WebAssembly module's textual representation (Figure 6.17). By using the Monaco editor here, we give the user the ability to easily navigate around this file using the built-in search functionality.

After a user has successfully compiled their project to WebAssembly, we invalidate [81] the query sent to retrieve the contents of the WAT file from the S3 bucket, this means that the next time the contents of the WAT file are fetched by clicking the 'View WAT'

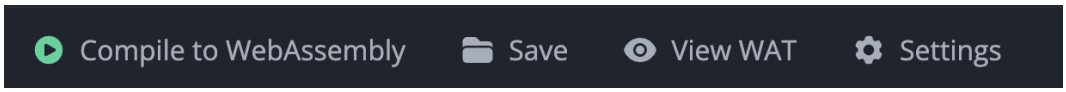



Figure 6.16: Editor Toolbar showing 'View WAT' Button.

button in the toolbar, there will be nothing present in the cache for this query meaning a new request will be sent to retrieve the contents of the WAT file itself to ensure the contents of the file are up-to-date.

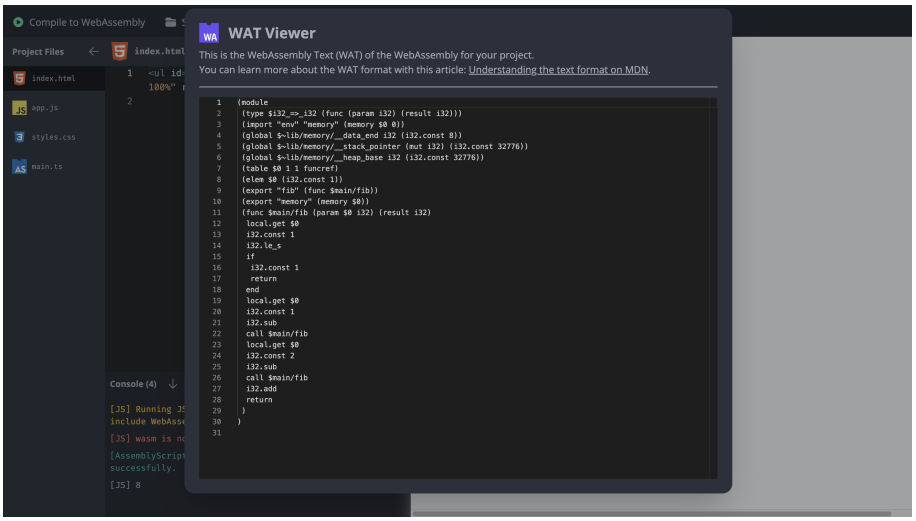



Figure 6.17: WebAssembly Text Format (WAT) Viewer displaying the textual representation of a module with a `fib` function exported.

### 6.3.7 Sharing

**Sharing Projects** We allow users to share their projects by sending a request to the `/projects/<project_id>/share` endpoint, this toggles whether to enable sharing for the project. When the project is toggled to be 'shareable', this request will return a 'share-code' which will allow any other user to 'fork' the project.

Project Sharing settings can be found in the settings dialogue, which may be opened by clicking 'Settings' from the toolbar and selecting the 'Sharing' tab.

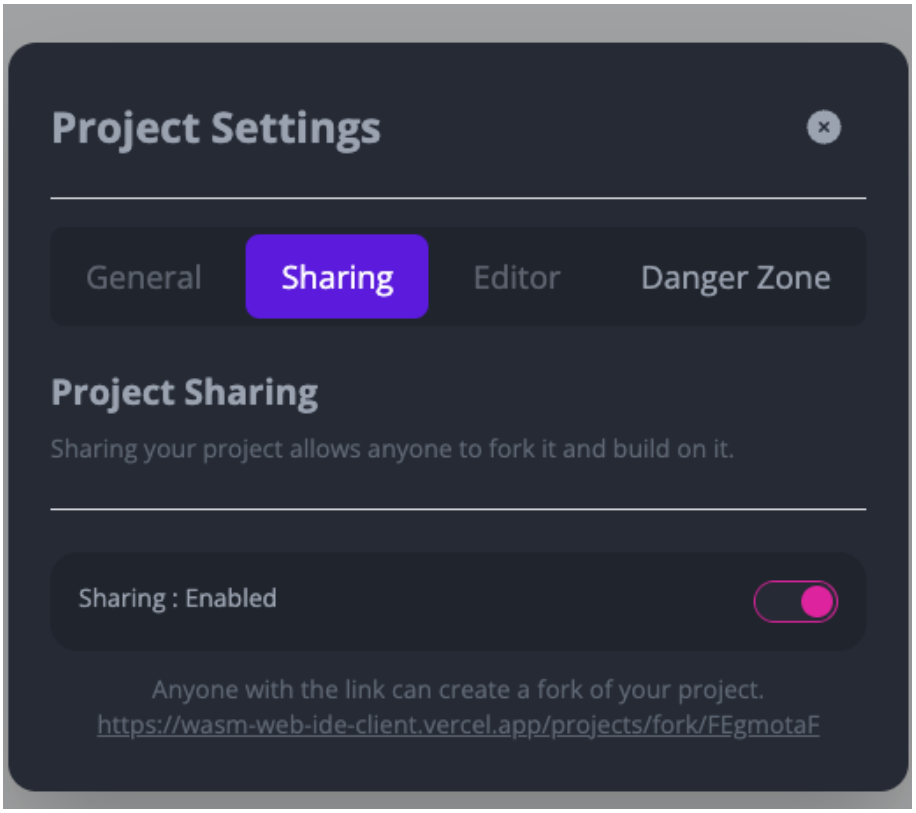



Figure 6.18: Project Sharing Settings - the user may toggle sharing for their project and can view the URL from which their project can be forked.

**Forking Projects** Any authenticated user may 'fork' a shared project by navigating to the `/projects/fork/<share_code>` page of the web application and clicking 'Fork this Project'. When this button is clicked, a `POST` request is sent to the `/projects/fork/<share_code>` endpoint on the API which will create a clone of the shared project referenced by the share-code. The newly created 'fork' project is owned by the user that created the fork and therefore (by default) not accessible to others. Once this process has been completed, the user is redirected to the project editor page for their newly forked project.

### 6.3.8 Deployment

The Web Application is deployed using the Vercel [96] platform. We use Vercel's automatic deployments feature [93] which integrates with GitHub to build and deploy the project on each push to the repository (Figure 6.19). This continuous-deployment pipeline proved extremely useful for improving productivity during development by allowing us not to worry about our web application's infrastructure, and by providing us with a way to quickly deploy and then gain feedback on new versions of the application.

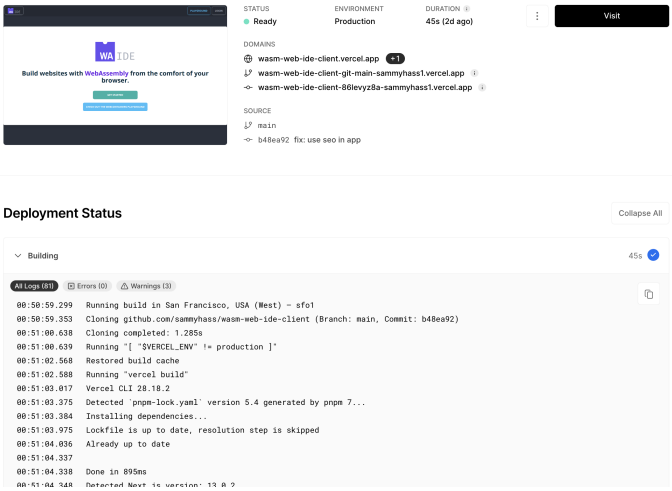



Figure 6.19: Web Application Deployment in Vercel [96], deployed to `https://wasm-web-ide-client.vercel.app/`.

Vercel provides us with a free domain (`https://wasm-web-ide-client.vercel.app/`) to host the web application using their infrastructure. We whitelist the provided domain with CORS on the API (Section 6.2.7) so that the deployed versions of the API and the web application can securely communicate over HTTPS.

## 6.4 WebContainer Playground

### 6.4.1 Overview

The 'WebContainer Playground' was created to address the issues that arose from using the 'Projects IDE' feature (Section 6.3), namely: the need for developers to be able to have proper access to a file system, the difficulty in operating the `syscall/js` package when using the TinyGo compiler [88] and the often extremely slow compilation times produced by the client-server model. We address these issues through the use of WebContainers [74], which, as discussed in the Related Works chapter (Section 3.4.2) is a WebAssembly based operating-system (localised to a browser tab) that allows us to safely run Node.js directly in the browser. The 'WebContainer Playground' targets users wishing to build web applications using HTML, CSS, JavaScript and AssemblyScript [84] (via WebAssembly).

The 'WebContainer Playground' itself operates within the `/playground` route of the Next.js application created for the 'Projects IDE' web application. Keeping the playground in the same application as the 'Projects IDE' allowed us to reuse certain React components during development, saving time when developing the playground, whilst also ensuring that the design language was kept consistent site-wide. Users do not need to be signed in to be able to use the playground.

The playground utilises the same Monaco Editor [48] instance used by the 'Projects IDE', meaning, that when editing AssemblyScript files, users will be shown type hints, and auto-completion for the AssemblyScript language, in the same way as is done in the 'Projects IDE' (Section 6.3.4). In addition, the 'editor settings' store is used across both the 'WebContainer Playground' and the 'Projects IDE' (and persisted in `localStorage`), meaning that editor customisation will be consistent across the two separate editors.

### 6.4.2 WebContainer Setup

**Security Requirements** The WebContainer API requires `SharedArrayBuffer` [70] support to allow objects from one agent to be safely cloned to another agent [23]. To allow this capability for workers to securely communicate with one another using `SharedArrayBuffer`, the Web Application must be in a 'Cross-Origin-Isolated' context. This can be done by setting the `Cross-Origin-Embedder-Policy` [19] and `Cross-Origin-Opener-Policy` [20] to `'require-corp'`, and, `'same-origin'`, respectively. We set these headers in our Next.js configuration file (`next.config.js`). It should be noted that `SharedArrayBuffer` is currently only stable in Chromium-based browsers, therefore the WebContainer playground will only function properly from within these browsers (e.g. Chrome, Edge, Brave). .

**Booting the WebContainer** When the playground page is reached, we use the `boot` method provided by the WebContainer API to initialise and boot the WebContainer. Once the booting process has been performed, the WebContainer instance is attached to the global `window` object, so that it may easily be accessed by other parts of our application with `window.webcontainer`. The `boot` method is only called when `window.webcontainer` is undefined, if the container is defined, then we already have a WebContainer booted on the window, which should not be overridden by a new instance.

**Mounting Files to the WebContainer** Once the WebContainer has been booted, we must initialise the file system for the WebContainer using the `mount` method. The files that are mounted to the WebContainer take the form of a `FileSystemTree` object [68] and represent a basic Node.js project using a Vite development environment [98] and the AssemblyScript compiler.

Node.js projects require a `package.json` file (Figure 6.20) to exist at the root of the project, this file is used to track scripts, dependencies and other metadata related to the project. The `package.json` mounted to the container specifies `assemblyscript` and `vite` as development dependencies. Furthermore, our `package.json` defines the scripts that will be needed for compilation to AssemblyScript (`npm run build-assemblyscript`), and for starting the Vite development server (`npm run dev`).

Provided the user is not using a generated playground link to mount their files (Section 6.4.7), we mount a simple default project which demonstrates how to export and use a function from a WebAssembly module built with AssemblyScript. In addition to the `package.json` file, we also mount the following files to the WebContainer:

- `index.html` The HTML entry point for the Vite web server.
- `main.js` The JavaScript entry point used by `index.html`.
- `styles.css` The style sheet used by the `index.html` file.
- `assemblyscript/index.ts` The AssemblyScript file which will be compiled by the `build-assemblyscript` script defined in `package.json`.
- `asconfig.json` An AssemblyScript configuration file [5] utilised by the `build-assemblyscript` script which allows users to modify compilation options for their AssemblyScript playground.
- `lib/setup-preview.js` A script used by the preview frame to redirect console output from the preview's frame to its parent frame, so that it may be shown within

```
export const packageJson = {
  name: 'wasm-ide-assemblyscript-playground',
  private: true,
  version: '0.0.0',
  type: 'module',
  scripts: {
    dev: 'vite',
    build: 'vite build',
    preview: 'vite preview',
    'build-assemblyscript': 'asc --config asconfig.json',
  },
  devDependencies: {
    vite: '^4.1.0',
    assemblyscript: 'latest',
  },
};
```

Figure 6.20: The `package.json` file mounted to the WebContainer by default after it has booted.

the IDE. This is the same script used by the 'Projects IDE'. (Figure 6.14) to override the console in the preview window.

Once the file system has been mounted, we use the WebContainer's `spawn` method to install project dependencies defined in the `package.json` file using the `npm install` command. In new playgrounds, this will install Vite and AssemblyScript as development dependencies (Figure 6.20). We pipe the output of this process, as well as all spawned processes in the WebContainer to the Console Window display in the IDE. This allows the user to easily monitor output from running processes in the WebContainer. Once the dependencies required for the playground have been installed, we may start the development server.

**Starting the Development Server** We set up an event listener on our WebContainer which listens for 'server-ready' events. These are emitted when a running server is listening for incoming connections. When this event is triggered it will provide us with a proxy URL from Stackblitz, which will give us the ability to use the Node.js application running in the browser's WebContainer as if it were running on its own website. These URLs are only proxies pointing to the server running in the window's WebContainer and are not publicly accessible.

We spawn [69] a new process to start the Vite development server inside of the container with `npm run dev`. Once the development server has started, the 'server-ready' event will fire on the WebContainer and we will receive a proxy URL from which we can safely access the application.

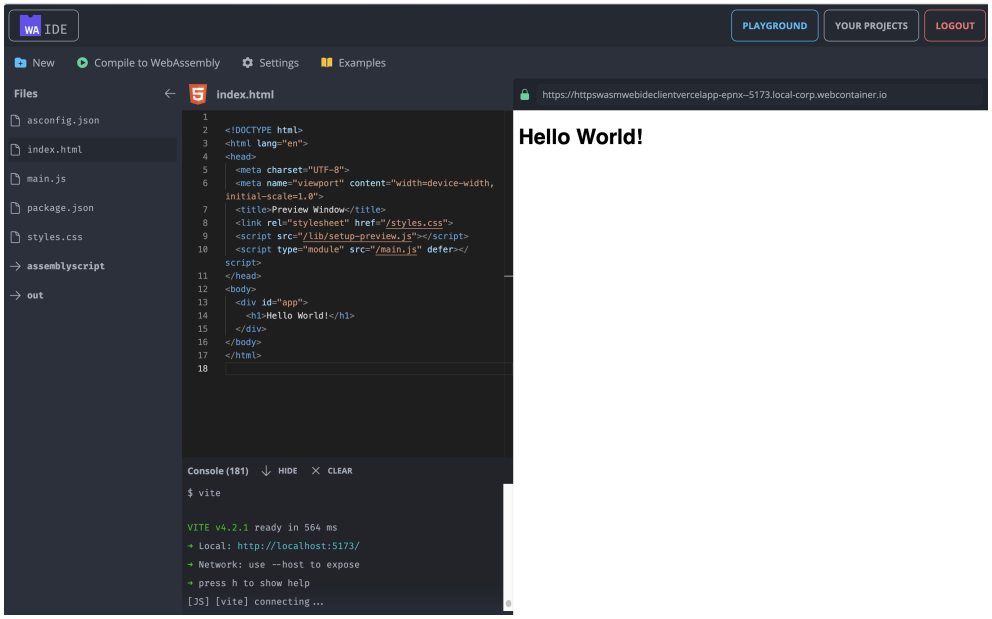



Figure 6.21: 'WebContainer Playground' after mounting project files, installing NPM dependencies and starting the development server. Note that the output from processes in the WebContainer is shown in the console.

### 6.4.3 File Management

The WebContainer's virtual file system can be managed with the WebContainers API [73] to allow users to perform reads and writes to the file system after it has initially been mounted.

**File System Tree** Once the files for the playground have been initially mounted, the user is shown a 'File Tree' to the left of the editor window. The file tree is a recursive React component which will display the files and folders stored in the WebContainer in a hierarchical structure. This component will be updated whenever the shape of the file system changes (e.g. folders/files added or deleted) to ensure the file hierarchy displayed in the file system tree is in sync with the WebContainer's file system. The user may click on any of the files shown in the file system tree to select that file and show its contents in the editor window.

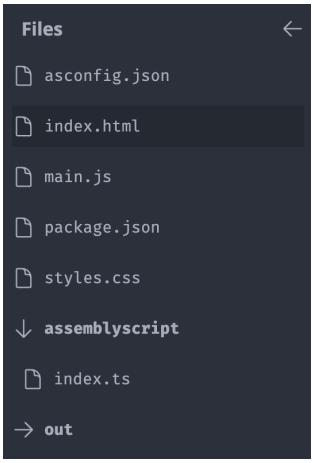



Figure 6.22: File Tree component displaying the structure of the WebContainer's file system. Note that users may 'expand' the folders displayed to show the files within them.

The File System Tree also provides the ability to handle context menu events [18] when any part of the component is right-clicked on. We use this event to show a context menu on right-click that overrides the default-right click functionality and provides users with contextual actions that may be performed on the file/folder that was right-clicked on (Figure 6.23).

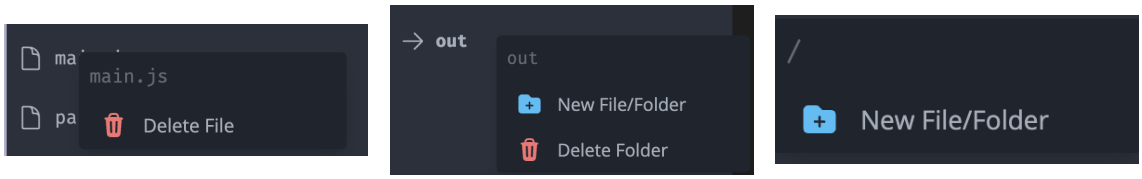



Figure 6.23: Context Menus shown when right-clicking on (from left-to-right) a file, folder and root of the file system tree.

**Editing and Saving Files** The selected file (chosen in the file tree) may be edited by the user from within the editor window. When a file is selected from the file tree, its contents are collected from the WebContainer (or from the query cache [83] if present) so that it may be shown in the editor. To update files in the WebContainer following an edit being made, we use 'debouncing' [108] to write the new contents of the file to the WebContainer's filesystem after 500 milliseconds of inactivity in the editor. This approach aims to minimise the number of times we perform writes to the WebContainer's filesystem, which is a relatively expensive operation and not one that should be performed on each keystroke. When the write to the file system is performed after the 'debounce' timer has expired, we also update the query cache to reflect the new state of the edited file so that it may be retrieved without the need to perform a new read operation on the WebContainer.

**Creating and Deleting Files** There are two ways users can create new files or folders (nodes) in the WebContainer: the toolbar, or through a context menu (Figure 6.23). When a user clicks the 'New' button in the toolbar or in the context menu, a dialogue modal will be shown which will allow the user to create a new file or folder inside of the WebContainer.

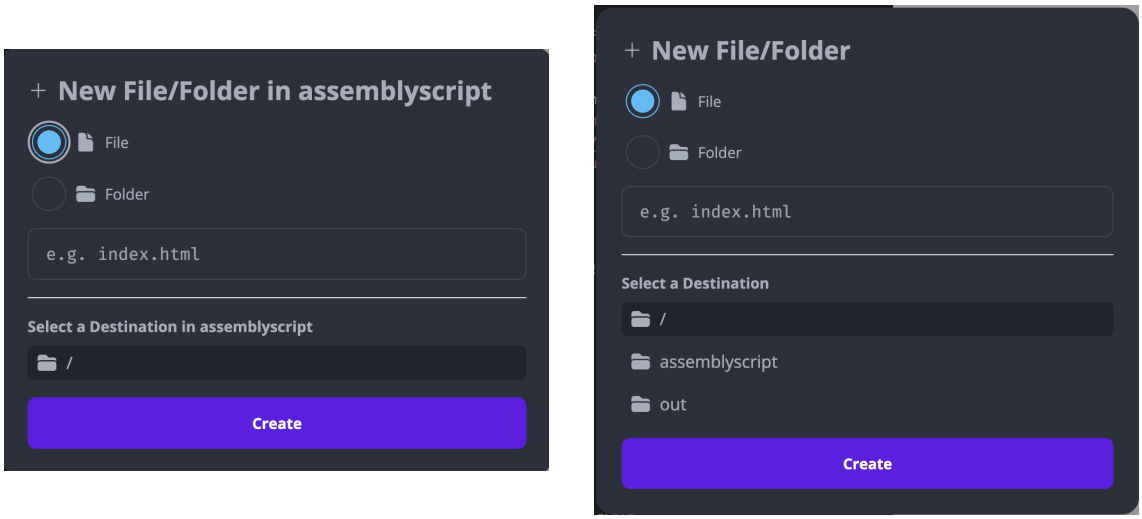



Figure 6.24: Node Creation Forms. Left: New file/folder form opened using the `assemblyscript` folder's context menu. Right: Form opened by clicking the 'New' button in the toolbar.

If using a folder's context menu to open the node creation form, the folder related to the context menu is used as the 'parent' directory for the form, meaning the created node will be stored in that folder. In contrast, if using the toolbar, we simply use the root of the WebContainer as the 'parent' directory. The form allows users to name their new node and gives users the option to choose which sub-directory within the form's 'parent' directory they want it to be stored in.

Upon form submission, if the user opted to create a new file, we call the `writeFile` [67] method on the WebContainer API's file system instance with the path defined by the user and an empty string for contents. This causes our WebContainer to create a new file in the location chosen by the user. If the user chose to create a new folder, we use the `mkdir` method on the WebContainer API's file system instance to create a new folder in the correct location. After the new node has been created, we update our application's cache (Section 6.3.2) to reflect the new node so the updates are displayed in the file system tree (Section 6.4.3). Finally, if the created node is a file, we select the newly created file so that is it immediately shown inside the playground's editor window.

Users can delete a file or folder in their WebContainer by bringing up the context menu for the file/folder they wish to delete. When the delete option is selected in the context menu, the user will be shown a confirmation dialogue to ensure that they wish to delete the file/folder they selected. Upon deletion, the application's cache is updated so that we no longer store the contents of the deleted file/folder in memory, and so that the deleted file/folder is no longer shown in the file tree.

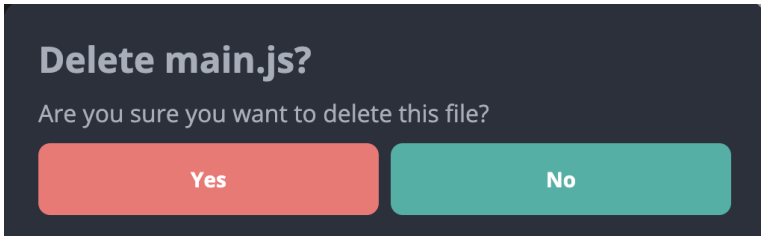



Figure 6.25: Deleting a file in the WebContainer playground. This confirmation dialogue is shown when selecting the 'Delete' option on a file/folder's context menu.

### 6.4.4 Compilation to WebAssembly

A user may compile their AssemblyScript file (by default stored in `assemblyscript/index.ts`) to WebAssembly, by clicking on the 'Compile to WebAssembly' button in the toolbar. When this is clicked, we run the `npm run build-assemblyscript` script defined in the WebContainer's `package.json`. This script results in our WebContainer running `asc --config=asconfig.json`, which will use the AssemblyScript compiler configuration file, `asconfig.json` [5] to find and compile the user's AssemblyScript file to WebAssembly with the AssemblyScript compiler.

The `asconfig.json` file (Figure 6.26) tells the compiler the location of the AssemblyScript file(s) to be compiled in the `entries` array (by default `assemblyscript/index.ts`). The configuration file also defines the AssemblyScript runtime we wish to use (`"stub"` by default) and specifies the location to which files should be output upon successful compilation. As can be seen in the configuration file (Figure 6.26), we define the `outFile` option to specify the location to store the compiled WebAssembly module, as well as the `textFile` option, which will tell the compiler where to output the WebAssembly Text Format (WAT) version of the compiled module (Figure 6.27).

The compiler configuration also specifies that we wish to generate EcmaScript Module (ESM) bindings for the WebAssembly module produced. This will result in the compiler

```
{
  "entries": [
    "assemblyscript/index.ts"
  ],
  "options": {
    "runtime": "stub",
    "bindings": "esm",
    "textFile": "out/module.wat",
    "outFile": "out/module.wasm"
  }
}
```

Figure 6.26: Default AssemblyScript Compiler Configuration file for WebContainer playground.

also generating a `module.js` (Figure 6.28) and `module.d.ts` (Figure 6.29) file upon compilation (both of these are stored in the same directory as the `outFile`). The `module.js` file will handle instantiating the WebAssembly module for the user and provide the JavaScript 'glue' needed to make the module function properly. The `module.js` file generated will export any functions that were exported from the instantiated WebAssembly module so that they may be easily used by simply importing the `module.js` from within another JavaScript module file. The `module.d.ts` file generated by the compiler is a TypeScript type declarations file which will contain TypeScript definitions for the WebAssembly module that can be used to ensure type safety and provide type hints to the user. Upon successful compilation to WebAssembly, the contents of the `module.d.ts` type declaration file generated by the compiler are loaded into the Monaco editor instance's JavaScript environment. This allows users that import the `module.js` file (to use their WebAssembly module) to view auto-completions and type hints for exports they defined in their AssemblyScript file.

It should be noted that the AssemblyScript compiler configuration may be changed by the user to adjust compilation settings for their playground.

JS out/module.js

```
1  async function instantiate(module, imports = {}) {
2    const { exports } = await WebAssembly.instantiate(module, imports);
3    return exports;
4  }
5  export const {
6    memory,
7    fib
8  } = await (async url => instantiate(
9    await (async () => {
10     try { return await globalThis.WebAssembly.compileStreaming(globalThis.fetch(url)); }
11     catch { return globalThis.WebAssembly.compile(await (await import("node:fs/promises")).readFile(url)); }
12   })(), {
13   }
14 ))(new URL("module.wasm", import.meta.url));
15
```

Figure 6.28: `out/module.js` bindings file generated by the AssemblyScript compiler. Users can import this file from their JavaScript files to interact with the instantiated module.

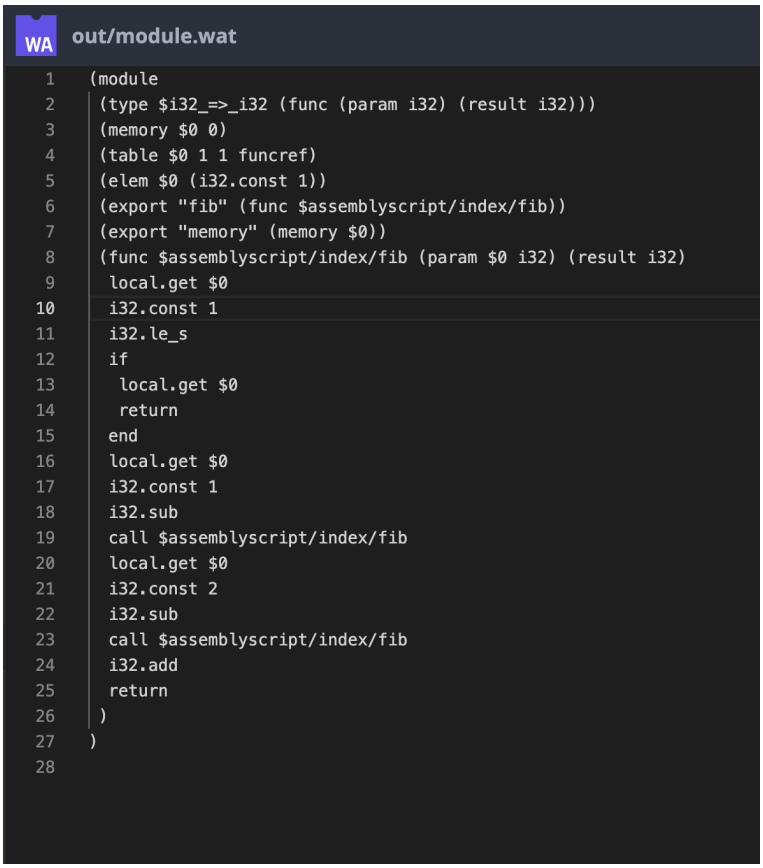



Figure 6.27: WebAssembly Text Format (WAT) representation of WebAssembly module defining an exported `fib` function shown in the playground.


Figure 6.29: TypeScript declaration file generated by AssemblyScript compiler. Note that there are two exports from the module, the `fib` function, and the WebAssembly module's `memory`.

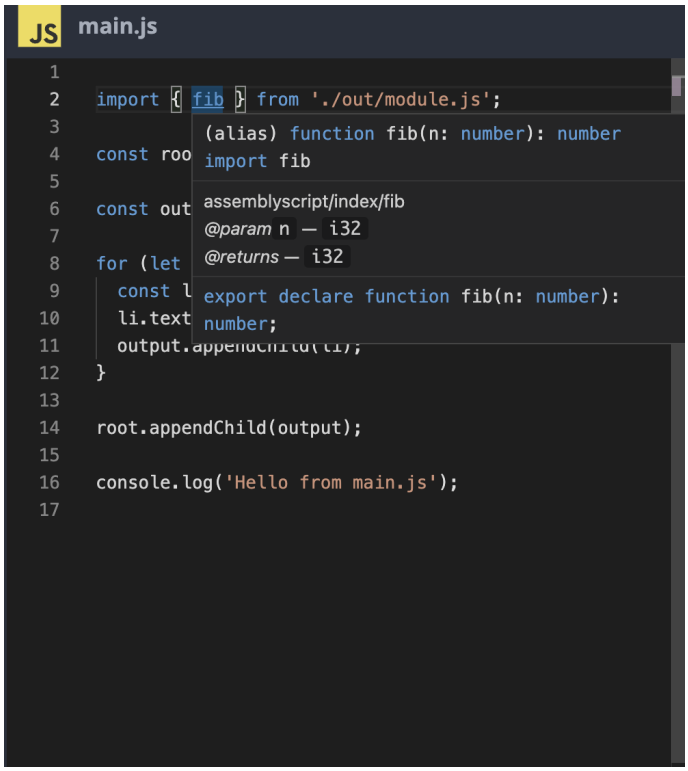



Figure 6.30: Type hints are shown when importing functions that have been exported by the compiled WebAssembly module. These type hints are shown by utilising the type declarations produced by the AssemblyScript compiler (Figure 6.29) in the Monaco editor's JavaScript environment.

Once compilation to WebAssembly has been successfully completed, we update the cache of our application to ensure the new files generated in the WebContainer at compilation are shown in the file tree so that they may be selected and viewed by the user.

### 6.4.5 Dependency Management

We provide users with the ability to manage NPM [28] dependencies related to their project by providing them with a dependency manager they may use to install and delete NPM packages. The dependency manager can be found by clicking on the settings button in the playground's toolbar, and then choosing the 'Dependencies' tab from within the settings dialogue.

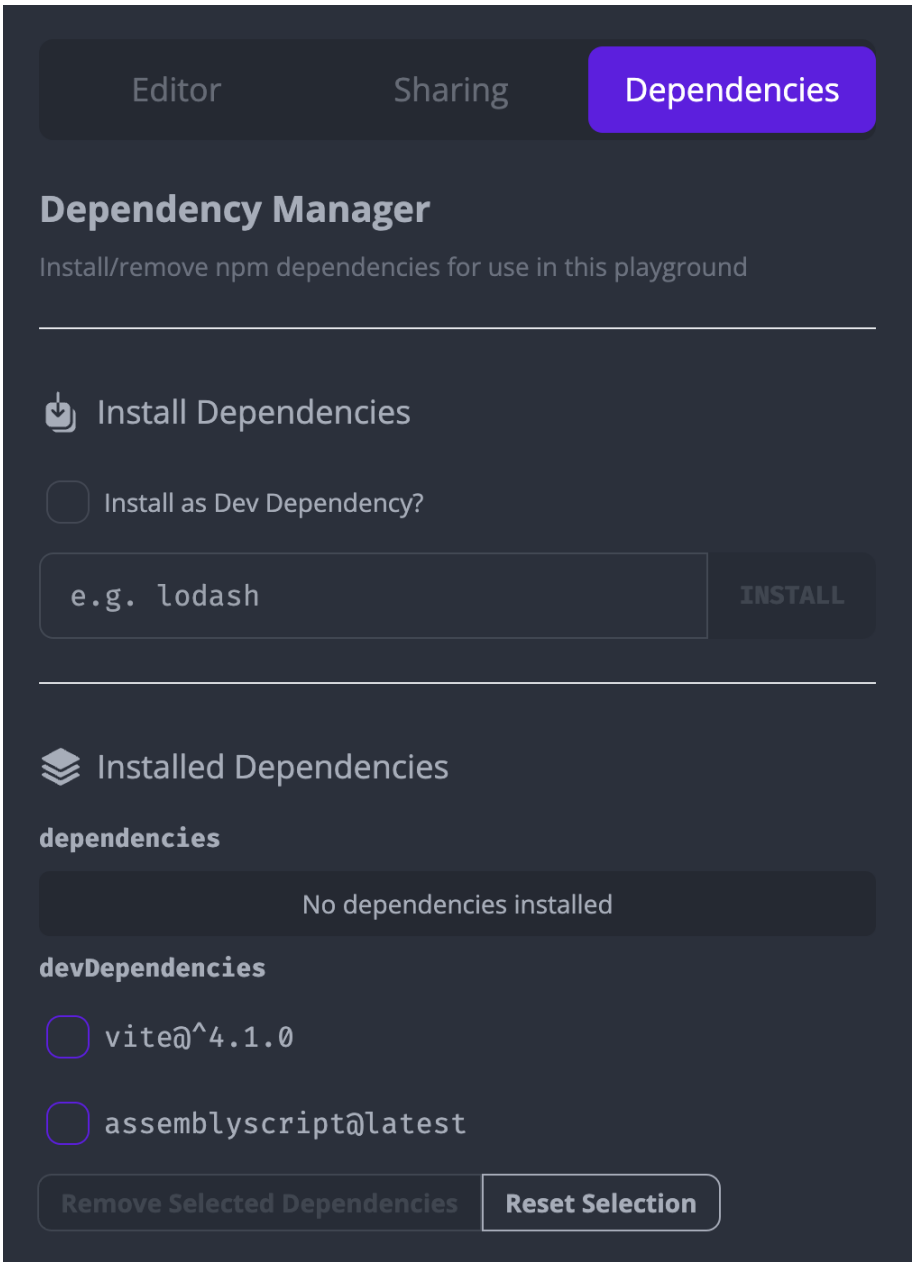



Figure 6.31: Dependency Manager for the WebContainer Playground. By default, the playground requires `vite` and `assemblyscript` to be installed as development dependencies

The dependency manager operates by inspecting the contents of the `package.json` file to find the dependencies and development dependencies installed within the WebContainer playground.

Users can install dependencies using the 'Install Dependencies' form. Here, users can add a dependency to their WebContainer playground, either for development purposes or for use in the web application itself. Upon submission of this form, we spawn a new process, '`npm install <dependency>`' which installs the dependency to the user's project in the WebContainer with NPM.

Users can remove dependencies from the WebContainer playground by selecting the ones they wish to remove from the 'Installed Dependencies' list and clicking on 'Remove Selected Dependencies', when this is done, we spawn a process in the WebContainer, `npm uninstall <dependencies>`, to uninstall the selected dependencies from the user's

project.

When the dependencies of the playground are updated, either by installing a new dependency or removing existing ones, the `package.json` file is re-read from the WebContainer to ensure the dependencies shown in the dependency manager are up-to-date.

By allowing users to install and manage NPM dependencies for their project, we allow users to extend the playground in any way they like. For example, with the ability to install NPM dependencies, it becomes possible to create an AssemblyScript/Vite application that uses the React [30] UI framework to render its content (Figure 6.32).

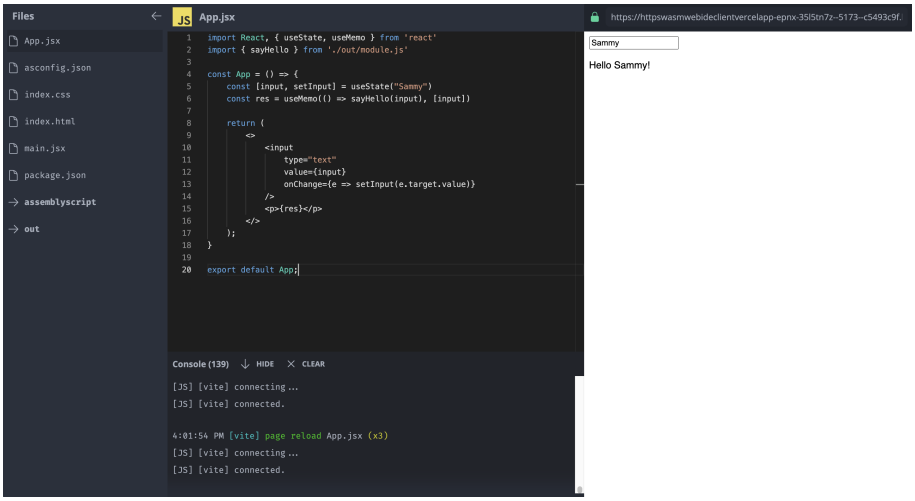



Figure 6.32: Example Playground using `react` and `react-dom` [30] as installed dependencies to render the preview window and interact with the compiled WebAssembly module. Playground Link: https://wasm-web-ide-client.vercel.app/playground?code=25030ed0aae2

### 6.4.6 Previews

As discussed previously, once the Vite development server has been started, a 'server-ready' event is fired by the WebContainer API which provides us with a proxy URL from which we can access the server running in the browser's WebContainer (Section 6.4.2). The URL provided by the 'server-ready' event is used as the `src` attribute for the iframe [22] element responsible for rendering the preview window in the WebContainer playground (Figure 6.33).

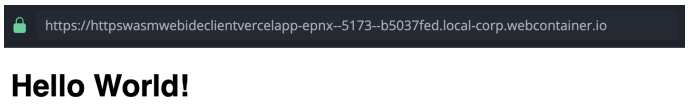



Figure 6.33: Playground Preview Window showing the user's running web application (inside a secure iframe) and the proxy URL pointing to the development server running inside of the browser's WebContainer.

The 'Vite' [98] development server used by the WebContainer provides live updates when files are changed through Hot-Module-Replacement. This means that whenever a file is updated (Section 6.4.3) in the WebContainer, the Vite development server reloads the changed module so that changes made are immediately reflected in the preview window.

The preview window's JavaScript console output in the playground's preview window is redirected to the IDE's console window using a similar approach used by the 'Projects IDE' (Section 6.3.5), whereby the console methods are overridden within the preview window to point to the `window.parent.postMessage` method.

### 6.4.7 Sharing

**Sharing Playground Files** Users are able to share their playgrounds with others. This is done by navigating to the 'Settings' dialogue by clicking 'Settings' in the toolbar and then selecting the 'Sharing' tab in the dialogue.

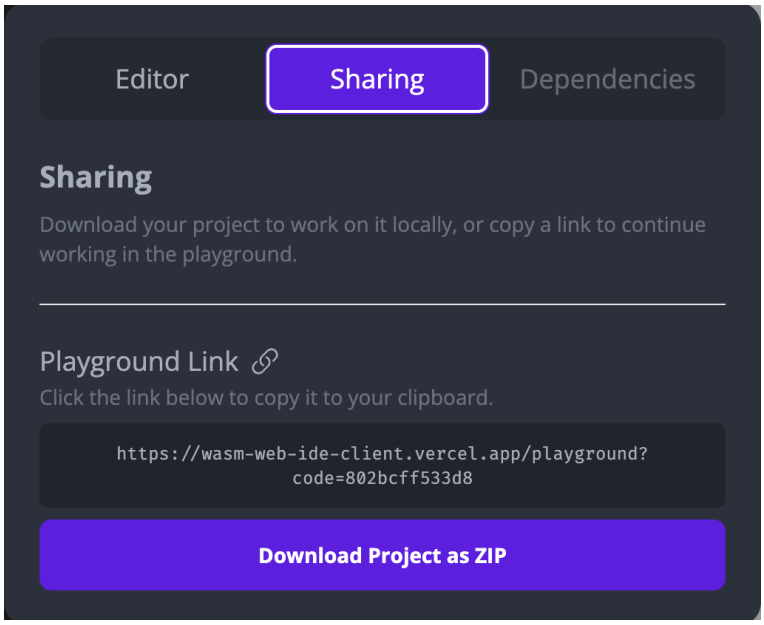



Figure 6.34: Sharing tab found in the 'Settings' dialogue. Sends a POST request to store the current state of the `FileSystemTree` in Redis so it may be retrieved in the future.

When a user reaches this dialogue (Figure 6.34), a share code will be generated for the playground. This is done using a POST request to an API route inside our Next.js application, `/api/link`. This route accepts as its body a `FileSystemTree` object [68] which represents the file system of the user's playground. The API route encodes the provided `FileSystemTree` object into a base64 string and generates a unique hexadecimal code which can be used to retrieve the project. These generated strings are stored in a Redis key-value store [64] hosted on Railway infrastructure [61]. The API route will return the share code related to the playground files once the project has been successfully stored/found in Redis.

```ts
const redis = new Redis(env.REDIS_URL);

const createCode = () => randomBytes(8).toString('hex');

const POST: NextApiHandler = async (req, res) => {
  const tree = req.body;

  const base64 = Buffer.from(JSON.stringify(tree)).toString('base64');

  const shortcode = await redis.get(`tree:${base64}`);
  if (shortcode) {
    res.status(200).json({ code: shortcode });
    return;
  }
  await redis.set(`tree:${base64}`, createCode());
  const newShortcode = await redis.get(`tree:${base64}`);
  await redis.set(`shortcode:${newShortcode}`, base64);

  res.status(200).json({ code: newShortcode });
};

const handler: NextApiHandler = async (req, res) => {
  switch (req.method) {
    case 'POST':
      return POST(req, res);
    default:
      res.status(405).end();
  }
};

export default handler;
```

Figure 6.35: API route used to generate and relate base64 encoded file system tree objects to random hexadecimal codes using Redis as a storage layer.

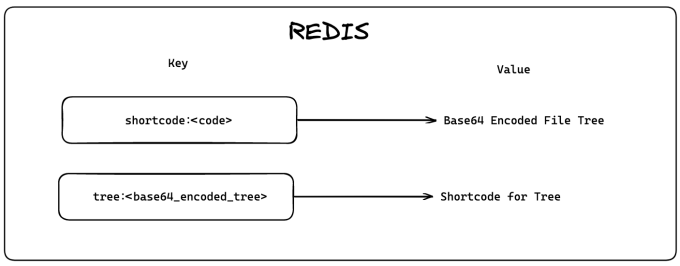



Figure 6.36: Redis Key-Value Scheme used to store share codes and base64 encoded `FileSystemTree` objects, and relate them to each other.

Prior to the generation of a new share code, we check that the `FileSystemTree` object provided is not already stored in Redis. If the `FileSystemTree` already exists under the `tree:<base64_tree>` key, we simply return the short-code stored referenced by the `tree:<base64_tree>` key in Redis.

Note that we avoid storing large files such as WebAssembly module files (stored in `out`) in Redis, in addition, we must prevent the `node_modules` folder from being saved as this folder can be extremely large and will be created by an `npm install` when the WebContainer is started, which will install any relevant dependencies found in the `package.json` file.

**Retrieving Shared Playgrounds** Users can retrieve 'shared' playgrounds by providing the generated share code as the 'code' query parameter for the playground page, (e.g. `/playground?code=<share_code>`). If a short code is provided as a query parameter for the playground page, the base64 encoded `FileSystemTree` will be retrieved from the Redis store, and, if found, will be parsed to a JavaScript object so that it represents the `FileSystemTree` object to be mounted to the WebContainer (instead of mounting the default files). Once the file-system tree object has been parsed, we mount it to the WebContainer (instead of mounting the default files) (Section 6.4.2).

The retrieval of `FileSystemTree` objects from the Redis store is performed server-side in the Next.js application using `getServerSideProps` on the playground page. This `getServerSideProps` [92] function provides the playground page with a `FileSystemTree` object as 'props' [29], which will be mounted to the WebContainer on the client side. If no `code` query parameter is provided to the `playground`, we simply return the default file system structure (Section 6.4.2) from the `getServerSideProps` (Figure 6.37) function so it is mounted after the WebContainer is booted.

```
export const getServerSideProps: GetServerSideProps<PageProps> = async ({
  query,
}) => {
  const redis = new Redis(env.REDIS_URL);
  const shortcode = query.code as string;

  if (!shortcode) {
    return {
      props: {
        files: filesystem,
      },
    };
  }

  const base64 = await redis.get(`shortcode:${shortcode}`);

  if (!base64) {
    return {
      notFound: true,
    };
  }

  const tree = JSON.parse(Buffer.from(base64, 'base64').toString());

  return {
    props: {
      files: { ...tree, lib: filesystem.lib },
    },
  };
};
```

Figure 6.37: The `getServerSideProps` function will be run server-side when the playground page is requested by a client. It retrieves the `FileSystemTree` for a shared playground or returns the default file system if no '`code`' query parameter is provided.

### 6.4.8 Downloading Files

```ts
const exportProject = async (
  container: WebContainer,
  w?: WritableStream<string>
) => {
  const allFiles = await container.fs.readdir('/');
  const processOut = await container?.spawn('npx', [
    'bestzip',
    './project.zip',
    ...allFiles.filter(f => f !== 'node_modules'),
  ]);

  w && processOut?.output?.pipeTo(w);
  const exit = await processOut.exit;
  if (exit !== 0) {
    throw new Error('Failed to export project');
  }

  const zip = await container.fs.readFile('/project.zip');

  const blob = new Blob([zip], { type: 'application/zip' });

  await container.fs.rm('/project.zip');

  return blob;
};
```

Figure 6.38: `exportProject` function responsible for creating a ZIP file containing the files in the playground and then returning the ZIP file as a blob URL so it may be downloaded.

Users can download their playground files as a ZIP file from within the 'Sharing' tab of the 'Settings' dialogue. Upon a user clicking the 'Download Project as ZIP' button (visible in Figure 6.34) a new process is started in the WebContainer which invokes the 'bestzip' [56] npm package (using the `npx` [57] utility) in order to create a zip file containing all of the files present in the user's playground (except for those stored in the `node_modules` folder). Once this process has been completed, we read the new `project.zip` file that was created by the `bestzip` process and convert the contents of the ZIP file to a JavaScript blob [17] which allows us to export the contents of the file outside of the WebContainer. This blob object is set as the `href` attribute of a hidden download link, which is then programmatically clicked, resulting in the file being downloaded.

# Chapter 7

# Testing

Testing is an extremely important part of the development of any complex system. We use two techniques to test the functionality and correctness of the 'Projects IDE' and the 'WebContainer Playground': unit testing and end-to-end testing. We use these automated tests to ensure consistent correctness in our implementation for both the API and the Web Application and to demonstrate that requirements for the IDE have been met.

## 7.1 Unit Testing

Unit Testing is used by the API (Section 6.2) to test individual components of functionality in an isolated setting. We create these unit tests with the Go `testing` package [8] and use the `go test` command to run them.

```
=== RUN   TestGenerateJWT
--- PASS: TestGenerateJWT (0.00s)
=== RUN   TestGenerateValidJWT
--- PASS: TestGenerateValidJWT (0.00s)
=== RUN   TestGenerateExpiredJWT
--- PASS: TestGenerateExpiredJWT (0.00s)
=== RUN   TestCompile_ValidAssemblyScript
--- PASS: TestCompile_ValidAssemblyScript (0.42s)
=== RUN   TestCompile_InvalidAssemblyScript
--- PASS: TestCompile_InvalidAssemblyScript (0.40s)
=== RUN   TestCompile_WorksWithValidGoFile
--- PASS: TestCompile_WorksWithValidGoFile (1.60s)
=== RUN   TestCompile_ReturnsErrorWithInvalidGoFile
--- PASS: TestCompile_ReturnsErrorWithInvalidGoFile (1.22s)
=== RUN   TestCompile_BeforeDelete
--- PASS: TestCompile_BeforeDelete (1.64s)
=== RUN   TestTinyGoWasm2Wat
--- PASS: TestTinyGoWasm2Wat (1.30s)
PASS
ok      github.com/sammyhass/web-ide/server/wasm    6.720s
```

Figure 7.1: Output produced when unit-testing our API with the `go test ./...` command

We use unit testing (Figure 7.1) for specific WebAssembly-compilation and authentication-related tasks on the API. Unit tests are not used for testing the storage layer of the API, since we mostly rely on well-tested external packages for this, notably, `gorm` [41] and the AWS Go SDK [9].

The unit tests for the API are run on each push to the GitHub repository using GitHub Actions [36] for continuous integration (CI). To be able to run all of these tests in a CI environment, we were required to set up the testing environment so that it can use the compilers (Section 6.2.4) and WebAssembly binary toolkit by spawning sub-processes. This is done in the `.github/workflows/test.yml` file which is responsible for setting up

our GitHub actions workflow (Figure 7.2).

```yaml
name: Test

on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Set up Go
        uses: actions/setup-go@v3
        with:
          go-version: 1.18
          check-latest: true
          cache: true
      - name: Install dependencies
        run: go mod download
      - name: Install TinyGo
        run: sudo bash ./scripts/install-tinygo.sh
      - name: Install WABT
        run: sudo bash ./scripts/install-wabt.sh
      - name: Install AssemblyScript
        run: sudo bash ./scripts/install-asc.sh
      - name: Build
        run: go build -v ./...

      - name: Test
        run: go test -v ./...

```

Figure 7.2: `.github/workflows/test.yml` file responsible for setting up the testing environment for the API and ensuring that the tests are run on each push to the GitHub repository hosting the API.

## 7.2 End-to-End Testing

End-to-end testing is the main form of automated testing used to ensure the entire application (back-end and front-end) works as expected from a user-facing perspective. To implement end-to-end tests, we use the Playwright Testing Library [46] which allows us to programmatically use our web application in order to test that it functions as expected. The end-to-end test suite can be found in the Next.js web application's repository in the `tests` folder and uses the Page-Object-Model [49] paradigm to model the pages of our application as classes that we can interact with in our tests.

End-to-end tests are run after each deployment of the Next.js web application to the Vercel platform [96]. In addition, the tests are run every 24 hours by a scheduled 'CRON' job to ensure that the application is consistently working as expected. This automation of test runs is accomplished using GitHub Actions [36] (Figure 7.3, Figure 7.4). A test report is generated and published for each run of the end-to-end test suite. This test report allows us to inspect the results of the tests, and view 'traces' [52] of each test to ensure they run properly and to provide proof that the entire application works as expected and according to the requirements we have specified. The test report is published to GitHub pages automatically following a test run and is deployed to `https://sammyhass.github.io/wasm-web-ide-client/`. By continuously running the tests and analysing the 'traces' (Figure 7.5) produced by them, we can guarantee that all requirements have been met and that the application consistently functions exactly according to our expectations. The 'traces' for each test can be found in the 'attachments' section located at the bottom of each 'test detail' page on the published Playwright report.

```yaml
name: Playwright Tests
on:
  deployment_status:
  schedule:
    - cron: '0 0 * * *'

jobs:
  test:
    if: github.event_name == 'deployment_status' && github.event.deployment_status.state == 'success' || github.event_name == 'schedule'
    timeout-minutes: 60
    runs-on: ubuntu-latest
    container:
      image: mcr.microsoft.com/playwright:v1.32.3-focal
    steps:
      - uses: actions/checkout@v3
      - uses: pnpm/action-setup@v2
        with:
          version: 7
      - uses: actions/setup-node@v3
        with:
          node-version: 16
      - name: Install dependencies
        run: pnpm install
      - name: Run Playwright tests
        run: pnpm exec playwright test
        env:
          HOME: /root
          TEST_URL: https://wasm-web-ide-client.vercel.app
          TEST_USER_EMAIL: ${{ secrets.TEST_USER_EMAIL }}
          TEST_USER_PASSWORD: ${{ secrets.TEST_USER_PASSWORD }}

      - uses: actions/upload-artifact@v3
        if: always()
        with:
          name: playwright-report
          path: playwright-report/
          retention-days: 3

      - name: Deploy report to gh-pages
        if: always()
        uses: peaceiris/actions-gh-pages@v3
        with:
          github_token: ${{ secrets.GITHUB_TOKEN }}
          publish_dir: ./playwright-report
          publish_branch: gh-pages
```

Figure 7.3: The `.github/workflows/e2e-test.yml` is responsible for setting up the GitHub action workflow which will run our end-to-end tests after each deployment to Vercel [96], and every 24 hours.

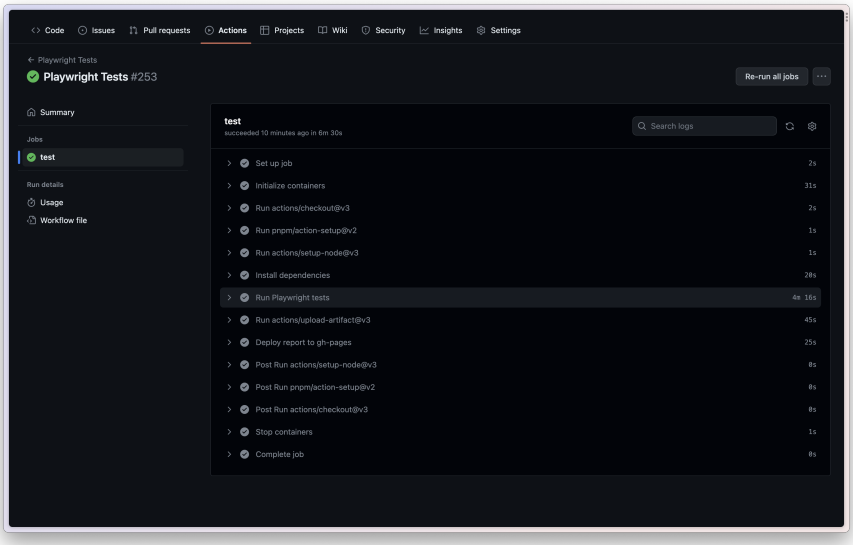



Figure 7.4: GitHub Action [36] used to run end-to-end tests.

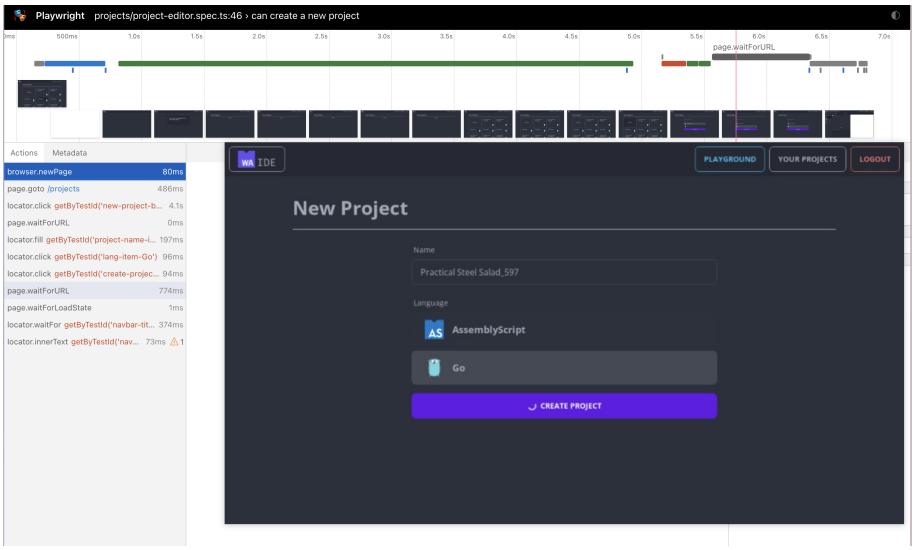



Figure 7.5: 'Traces' produced by each test in our end-to-end test suite allow us to verify that the user interface and functionality for the application are as expected. This test checks that a user can create a new project.

### 7.2.1 Projects IDE

**Authentication** We use end-to-end tests to ensure our authentication system functions correctly (Figure 7.6). These tests use the web application to guarantee that users can register new accounts, log in to an account and log out of their account.

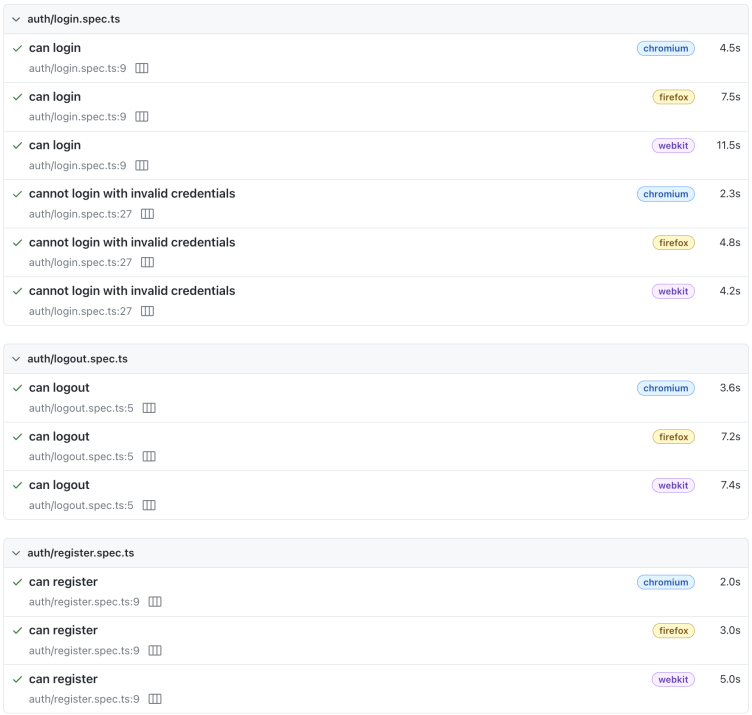



Figure 7.6: Test report for authentication tests

**Project Management** We use end-to-end tests to use the project management system and test that users of our application can create a new project, view their new project

on the `/projects` page, rename their project and subsequently delete the project (Figure 7.7).

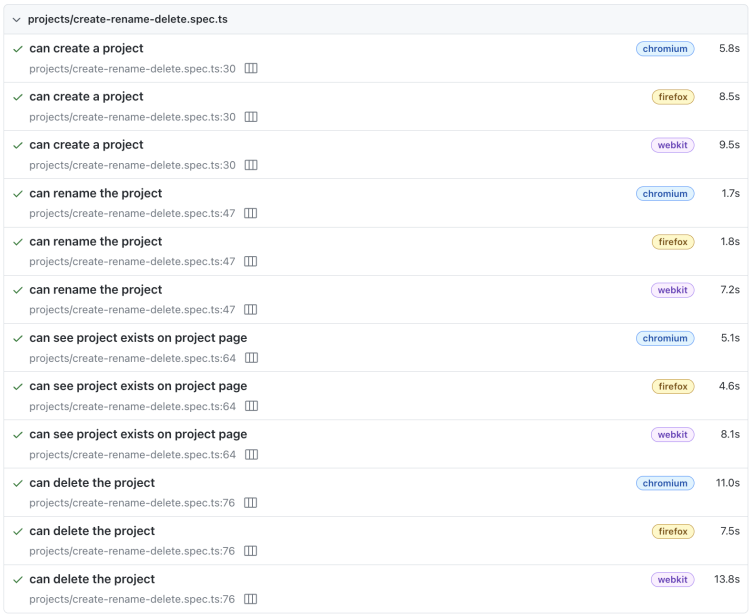



Figure 7.7: Test report for project management tests

**Previewing and Editing Projects** We test the project editor page to ensure that users can edit HTML, JavaScript and CSS files in their project, then subsequently save these files and have their edits appear in the preview window. We also test the ability of the IDE to display console output from the preview window by checking that we can view `console.log` calls in the console window, in addition to ensuring that any errors thrown in the user's JavaScript will be visible in the console window.

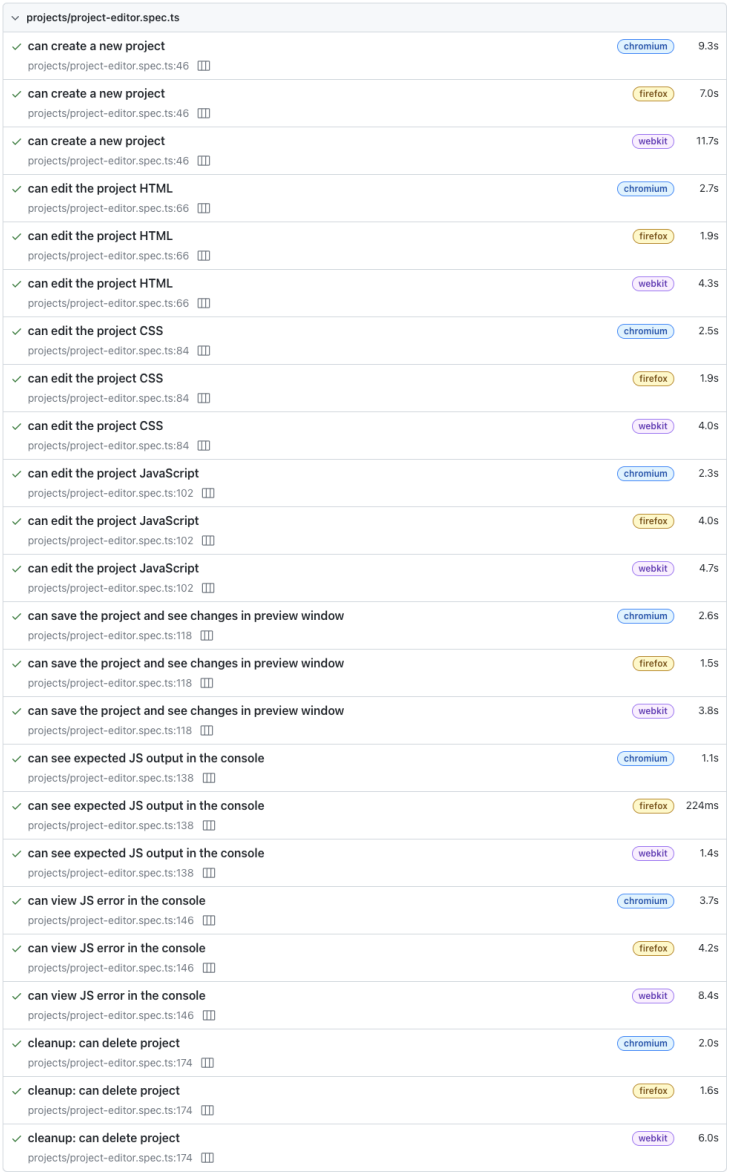



Figure 7.8: Test report for project editor and previews tests.

**Compilation to WebAssembly** We use end-to-end testing to ensure that with the 'Projects IDE', a user is able to compile a project's source code to WebAssembly and subsequently view the WebAssembly Text Format (WAT) representation of the WebAssembly module produced at compilation (Figure 7.9). These tests are implemented for both possible project source languages (Go and AssemblyScript) using parameterized testing [50] to ensure that the experience of compiling source code to WebAssembly is the same regardless of the source language used.

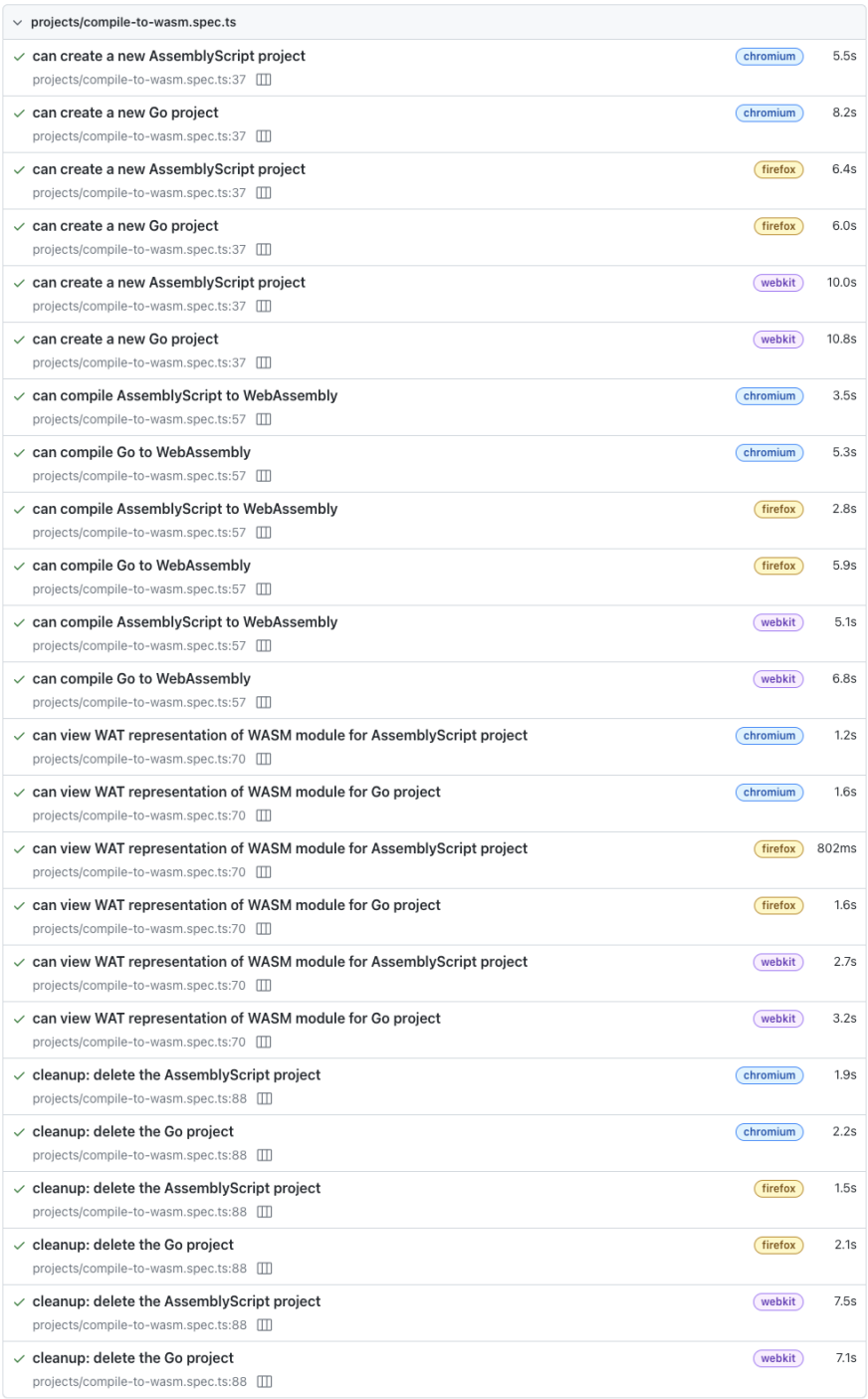



Figure 7.9: Test report for WebAssembly compilation tests in the 'Projects IDE' for both AssemblyScript and Go.

### 7.2.2 WebContainer Playground

We test the 'WebContainer Playground' (Figure 7.10) by ensuring that it correctly sets up the Node.js environment for the user within a WebContainer. We check that the files mounted to the WebContainer when it is initially set up are correct and are shown in the file tree so that users may navigate between them.

We ensure that the preview window is shown correctly once the development server has started and that console output is displayed when running processes in the WebContainer. We test that compilation from AssemblyScript to WebAssembly works as expected within

the playground and finally ensure that users can download their project as a ZIP file. These tests are only run in a Chromium browser instance due to WebContainers only currently being supported in Chromium-based browsers (Section 6.4.2).

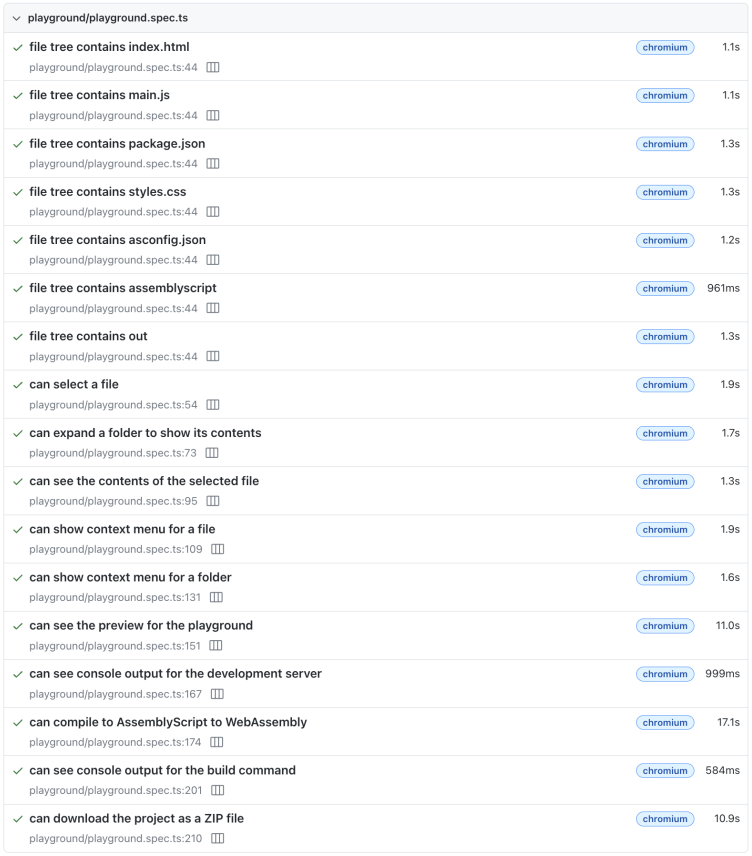



Figure 7.10: Test report for the 'WebContainer Playground' tests

# Chapter 8

# Application Demonstration

In this chapter, we discuss how the application we have created may be used to build websites using JavaScript, HTML, CSS and WebAssembly. We first describe how a user would use the 'Projects IDE' to create a new project and use TinyGo for WebAssembly compilation. We then discuss how a user can use the 'WebContainer Playground' to create a website using a WebAssembly module compiled from AssemblyScript source code.

## 8.1 Projects IDE

We will demonstrate how we can use the Projects IDE to create a simple website which calculates Fibonacci numbers using a WebAssembly module produced with TinyGo. We will then demonstrate how we can use JavaScript to interact with the WebAssembly module produced by the TinyGo compiler.

### 8.1.1 Creating an Account

To begin using the Projects IDE, we must create a new account using the `/register` page.

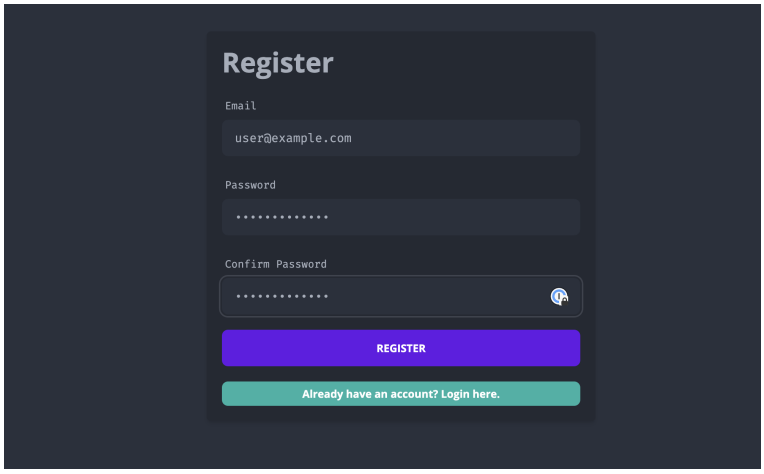



Figure 8.1: Registration Form for the Application

Upon successful registration, we will be redirected to the `/projects` page (Figure 8.2) where we can click the 'New Project' button to show the form for project creation.

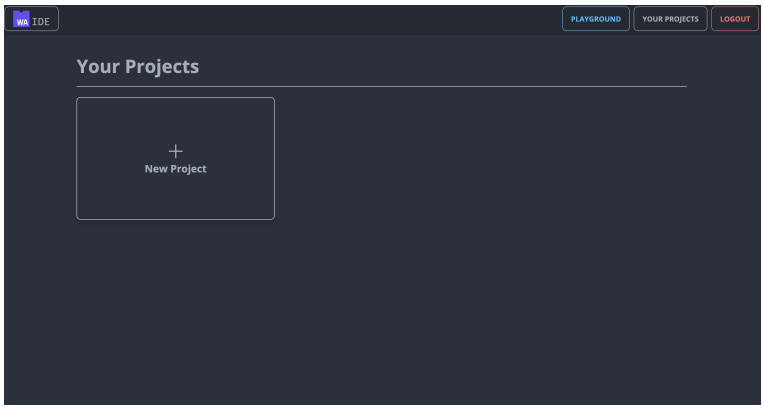



Figure 8.2: Projects Page for the Application

### 8.1.2 Creating a New TinyGo Project

We create a new TinyGo project on the `/projects/new` page (Figure 8.3). Upon form submission, we will be taken to the editor page for the newly created project (Figure 8.4).

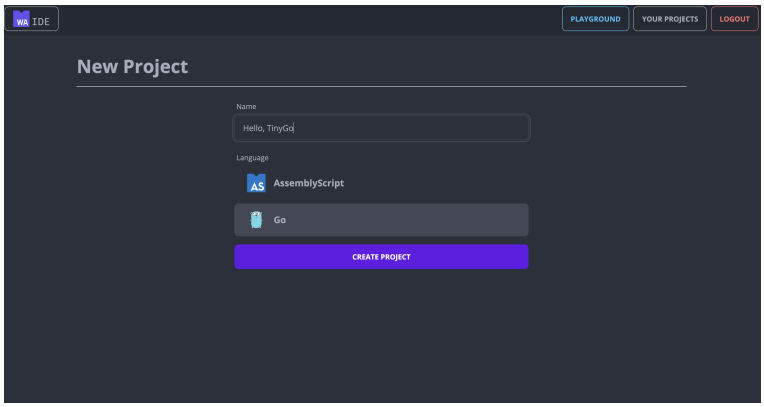



Figure 8.3: New Project Form

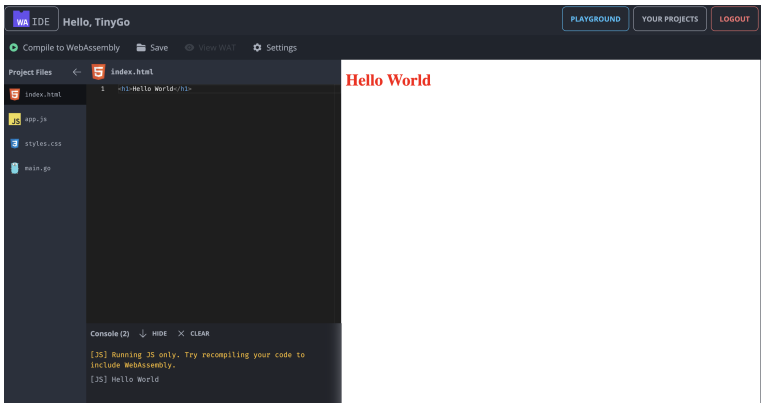



Figure 8.4: Project Editor Page shown upon project creation

### 8.1.3 Writing and exporting a function with Go

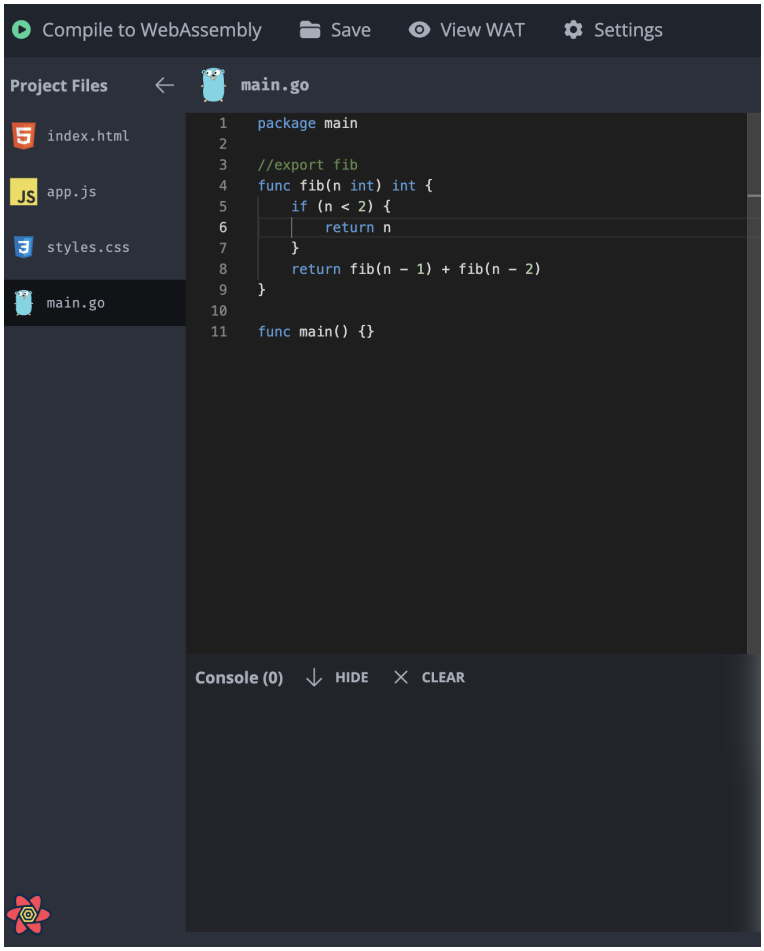



Figure 8.5: `main.go` file in the editor with an exported `fib` function.

We will now write the Go code which will be compiled into WebAssembly using the TinyGo compiler [90]. We will begin by selecting the `main.go` file from the file tree and remove the default contents of the file, we will keep the `package main` declaration at the top of the file to ensure the compiler knows that this is the entry point to our Go program.

We will now write a single function, `fib`, which will receive an integer, $n$, as its only parameter, and will return the $n$th number in the Fibonacci sequence. To export this function so it is available in our WebAssembly module for use, we must add a comment above the function "`//export fib`". This will tell TinyGo to export this function in the compiled WebAssembly module so it may be used from JavaScript.

It is important to note that we must add a `main` function to our Go code, this is a requirement for all Go `main` packages. This `main` function will be run automatically when the WebAssembly module is loaded into JavaScript, but for the purpose of this demonstration, we will leave it empty (Figure 8.5).

We will now save our project's code and compile the Go code we have written to WebAssembly. First, we click the "Save" button in the IDE's toolbar. Once the project has been successfully saved (as indicated by an alert which will appear in the lower right-hand corner of the IDE window) we can click the 'Compile to WebAssembly' button in the toolbar. The Go code we have written will then be compiled to WebAssembly using TinyGo [90], and we will see an alert and a console message (Figure 8.6) when compilation is completed successfully. We will now be able to use the exported `fib` function from

within our JavaScript code.

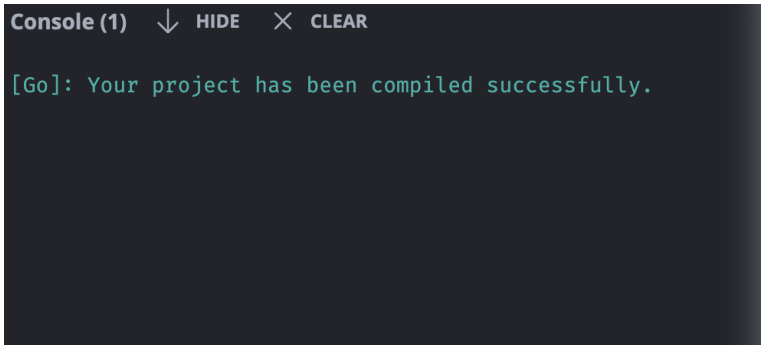



Figure 8.6: Compilation Success Message displayed in the editor console

### 8.1.4 Using the WebAssembly Text Format Viewer

Now we have successfully compiled the Go code to WebAssembly, we can view the compiled WebAssembly module in the WebAssembly Text Format (WAT) representation. We can click the 'WAT Viewer' button in the toolbar to view this file. The WAT representation for TinyGo projects will be quite a large file as parts of the Go runtime (including the Go scheduler) need to be included in the WebAssembly module. We can find the exported `fib` function in the WAT representation by searching for it within the 'WAT Viewer' (Figure 8.7).

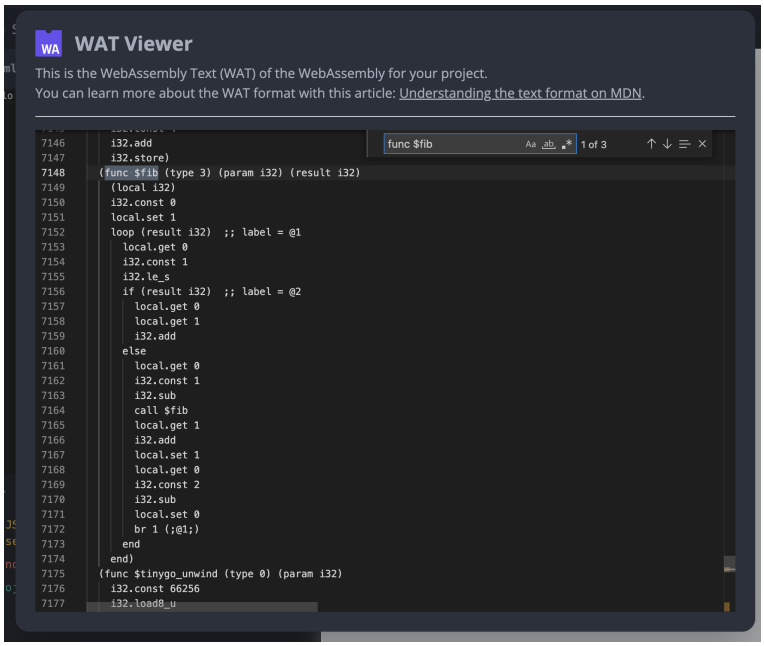



Figure 8.7: WebAssembly Text Format representation of the exported `fib` function

### 8.1.5 Using the WebAssembly Module from JavaScript

We can now use our exported `fib` function in our `main.js` file. To demonstrate the `fib` function, we will simply use a JavaScript `alert` to show the output of calling our `fib` function. We can use our `fib` function by calling the `wasm.exports.fib` function from within JavaScript (Figure 8.8) (any functions we export from WebAssembly will be accessible on the corresponding field of `wasm.exports` in JavaScript). We will provide

the number 10 as an argument for the `fib` function. We can then click 'Save' and our updated JavaScript will be run in the preview window, subsequently producing an alert containing the number 55 (Figure 8.9).

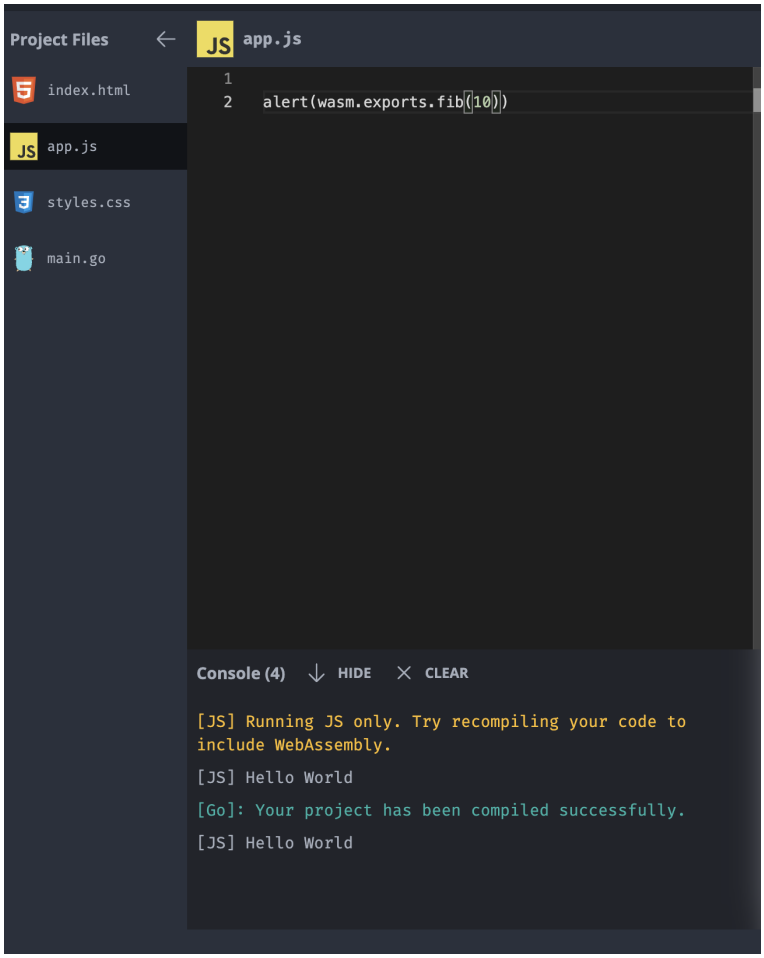



Figure 8.8: Using the `fib` function exported from WebAsssembly from JavaScript

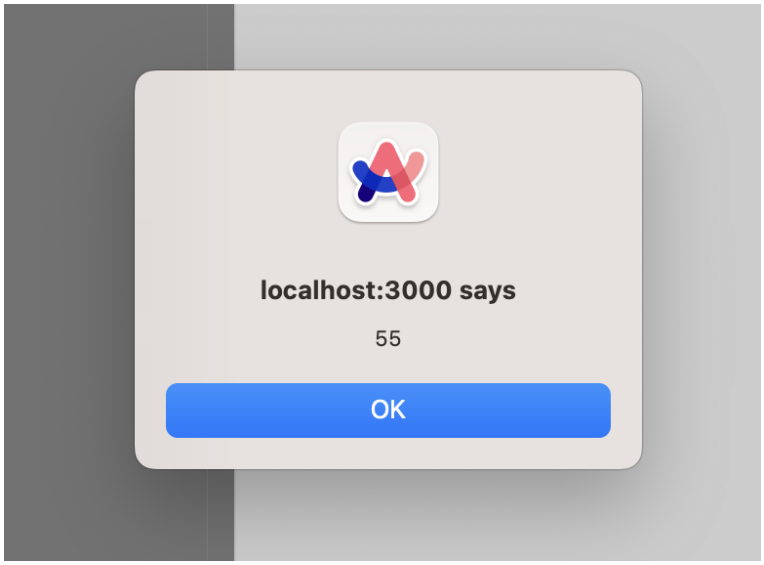



Figure 8.9: The resulting alert calls shows the output of the `fib` function defined by our WebAssembly module, and displays the result of `fib(10)`, which is 55.

### 8.1.6 Conclusion

Using the principles discussed here for using the Projects IDE it is possible to use TinyGo to create far more complicated projects than just the one we have demonstrated. A more complicated project which implements 'Conway's Game of Life' using the principles discussed in this section can be forked (Section 6.3.7) from `https://wasm-web-ide-client.vercel.app/projects/fork/FEgmotaF`.

## 8.2 WebContainer Playground

We will demonstrate how we can use the 'WebContainer Playground' to create a simple website with a form that adds and subtracts two numbers using a WebAssembly module produced by the AssemblyScript compiler. We will first navigate to the `/playground` page and wait for the Vite development server to start (Figure 8.10).

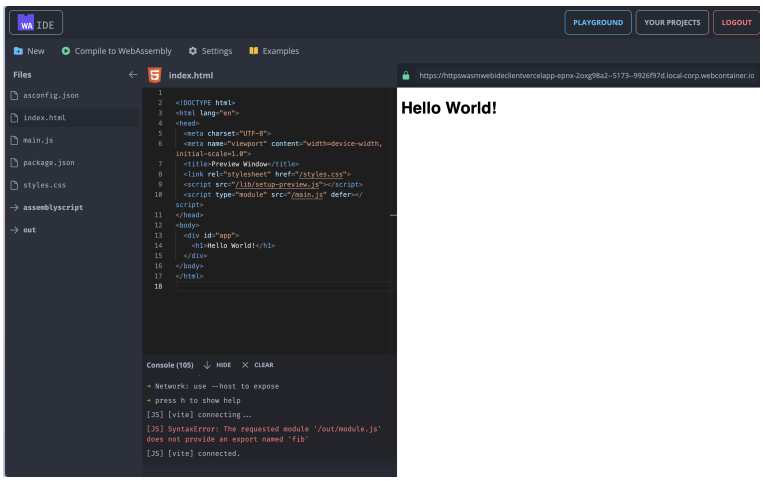



Figure 8.10: WebContainer Playground once the development server has started.

### 8.2.1 Writing and exporting functions with AssemblyScript

To begin, we will write the AssemblyScript code we need for two functions: `add`, which will add two numbers, and `subtract`, which will subtract two numbers. We will expand the `assemblyscript` folder in the file tree to reveal the `index.ts` file inside of it - this is where we will write our AssemblyScript code. We clear the default content of this file and replace it with the `add` and `subtract` functions which we will export for use in JavaScript by preceding our function definitions with the `export` keyword (Figure 8.11).

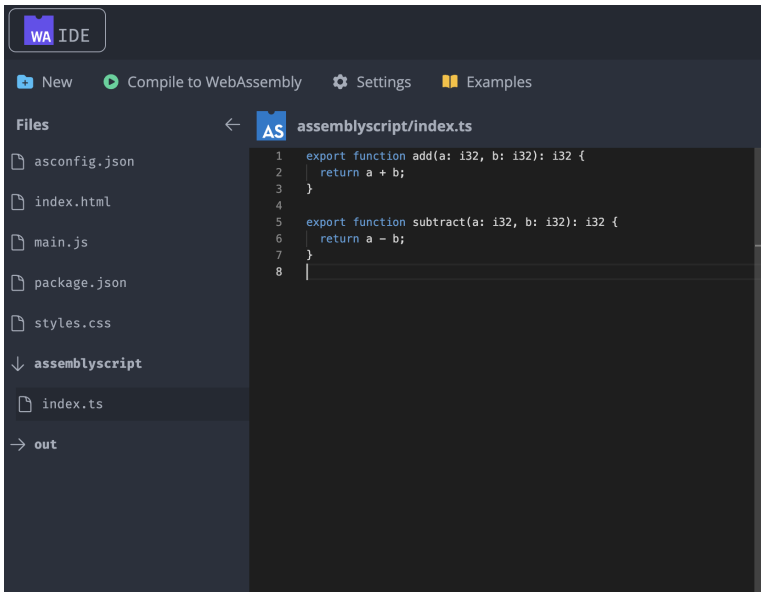



Figure 8.11: `assemblyscript/index.ts` file containing the exported `add` and `subtract` functions

### 8.2.2 Compilation to WebAssembly

We can now compile the AssemblyScript code we have written to a WebAssembly module. This can be done by clicking the 'Compile to WebAssembly' button visible in the toolbar. Upon clicking this button, we will see in our console window (Figure 8.12) that the '`asc --config=asconfig.json`' command is run to compile the AssemblyScript code to a WebAssembly module using the compiler configuration file defined in `asconfig.json`.

Once compilation has been completed, we will see that the `out` directory in our file tree has been populated with four files:

- `module.wasm`, the WebAssembly module produced by the AssemblyScript compiler
- `module.wat`, a WebAssembly Text Format (WAT) representation of the WebAssembly module.
- `module.js`, a JavaScript file responsible for instantiating our WebAssembly module (with any required bindings) so it may be utilised by inter-operation with JavaScript in our application.
- `module.d.ts`, a TypeScript type declarations file detailing the types of each function/value exported by our WebAssembly module.

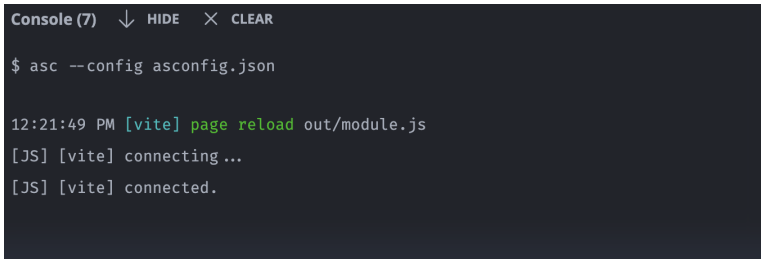



Figure 8.12: Console window after clicking the 'Compile to WebAssembly' button.

We can view the WebAssembly Text Format (WAT) representation of our WebAssembly module by selecting the `out/module.wat` file in the file tree (Figure 8.13).

out/module.wat

```
(module
 (type $i32_i32_=>_i32 (func (param i32 i32) (result i32)))
 (memory $0 0)
 (table $0 1 1 funcref)
 (elem $0 (i32.const 1))
 (export "add" (func $assemblyscript/index/add))
 (export "subtract" (func $assemblyscript/index/subtract))
 (export "memory" (memory $0))
 (func $assemblyscript/index/add (param $0 i32) (param $1 i32) (result i32)
  local.get $0
  local.get $1
  i32.add
  return
 )
 (func $assemblyscript/index/subtract (param $0 i32) (param $1 i32) (result i32)
  local.get $0
  local.get $1
  i32.sub
  return
 )
)
```

Figure 8.13: WebAssembly Text Format (WAT) representation of the WebAssembly module showing the functions `add` and `subtract` as exports.

### 8.2.3 Showing a form using HTML

We will now write the HTML required to show a form to the user. The form will have two input fields where the user can enter numbers. We will replace the contents of the 'body' in our `index.html` file with this form (Figure 8.14) and a placeholder `code` element where we will show the result of the calculations performed.

When we update the `index.html` file, we will see our development server will refresh meaning that we will be able to see the changes we make in real-time in the preview window. (Figure 8.15).

index.html

```
1
2   <!DOCTYPE html>
3   <html lang="en">
4   <head>
5     <meta charset="UTF-8">
6     <meta name="viewport" content="width=device-width,
    initial-scale=1.0">
7     <title>Preview Window</title>
8     <link rel="stylesheet" href="/styles.css">
9     <script src="/lib/setup-preview.js"></script>
10    <script type="module" src="/main.js" defer></script>
11  </head>
12  <body>
13      <form>
14          <input name="a" type="number" placeholder="a"/>
15          <input name="b" type="number" placeholder="b"/>
16          <input type="submit" value="Calculate" />
17      </form>
18      <code id="result" />
19  </body>
20  </html>
21
```

Figure 8.14: `index.html` file responsible for showing a form where users can enter two numbers and click 'Calculate'.

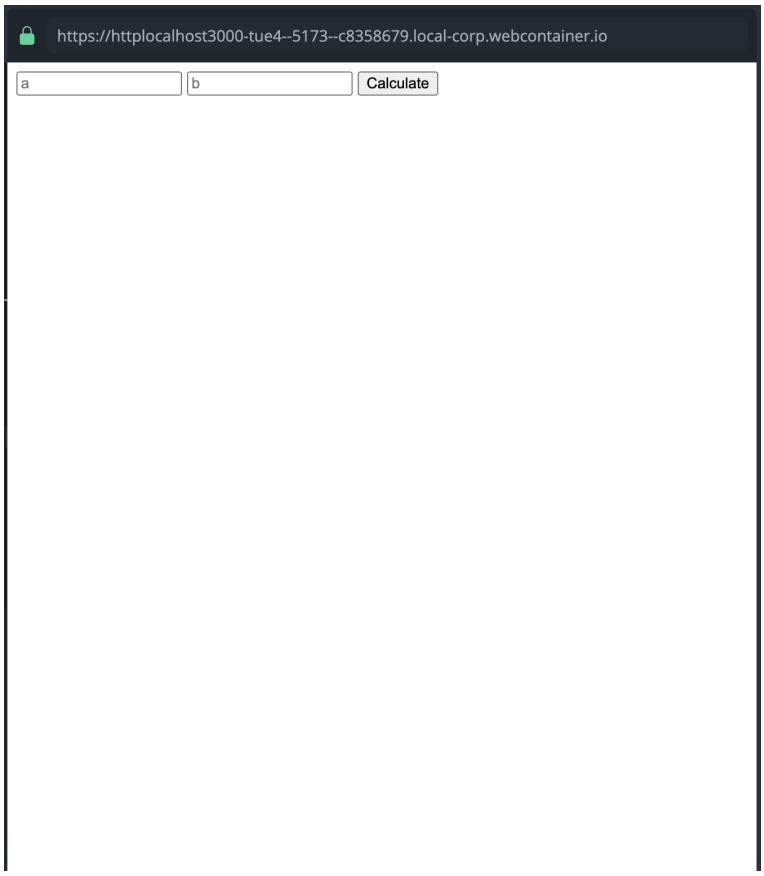



Figure 8.15: Updated preview window showing the form we created in the `index.html` file.

### 8.2.4 Handling form submission using JavaScript

Now we have added the HTML responsible for showing a form to users, we must handle the form's 'submit' event using JavaScript in our `main.js` file so that we may call the functions we have exported from our WebAssembly module.

After removing the default JavaScript code in `main.js`, we will import the `add` and `subtract` functions that we exported from the WebAssembly module from the `out/module.js` file produced when we compiled our AssemblyScript code to WebAssembly. Note that when we import `out/module.js` in `main.js` we will see auto-completion suggestions for the functions we exported. Furthermore, we can hover over these imported functions to view the type definitions for them that were defined in `out/module.d.ts` (Figure 8.16).

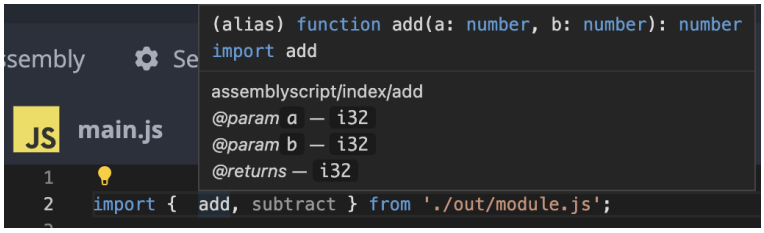



Figure 8.16: Type definition for `add` function exported from the WebAssembly module shown when we 'hover' over the imported function.

We will now write the rest of the JavaScript code responsible for handling the form's

'submit' event and showing the result of adding and subtracting these two numbers (Figure 8.17).

main.js

```
1
2   import {  add, subtract } from './out/module.js';
3
4   const output = document.getElementById('result');
5
6   const form = document.querySelector('form')
7
8   function onSubmit(event) {
9     event.preventDefault();
10
11    const formData = new FormData(form)
12
13    const a = parseInt(formData.get("a"))
14    const b = parseInt(formData.get("b"))
15
16    const addResult = `add(${a}, ${b}) = ${add(a, b)}`
17    const subResult = `subtract(${a}, ${b}) = ${subtract(a, b)}`
18
19    output.innerHTML = `${addResult}<br />${subResult}`
20  }
21
22  form.addEventListener('submit', onSubmit)
23
```

Figure 8.17: Updated `main.js` file used to handle the form's 'submit' event and call the `add` and `subtract` functions exported from the WebAssembly module and subsequently show the result to the user.

Now we have implemented the JavaScript responsible for handling the form submission, when a user enters two numbers into the form and presses the 'Calculate' button, they will see the result of performing addition on subtraction with these two numbers (Figure 8.18).

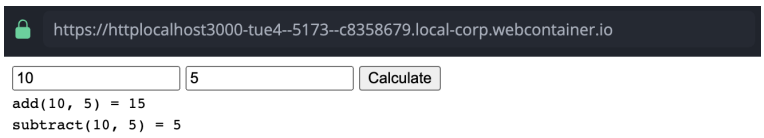



Figure 8.18: [
Preview Window showing the result of calling the `add` and `subtract` functions]Preview window showing the result of adding and subtracting the two numbers entered by the user using WebAssembly.

### 8.2.5 Conclusion

We have demonstrated how we can create a simple website which utilises WebAssembly to perform mathematical operations. The 'WebContainer Playground' itself is extensible so that users can create any web application that they would normally create using Node.js [33] and utilise AssemblyScript and WebAssembly within these applications. Note that we haven't demonstrated here that users can create new files in their playground using the WebContainer file system, as well as install external dependencies from NPM [28]. Additional more complex examples for use-cases of the WebContainer playground can be found on the examples page for the playground (`https://wasm-web-ide-client.vercel.app/playground/examples`).

# Chapter 9

# Evaluation & Future Work

## 9.1 Evaluation

### 9.1.1 Reflection

This project aimed to build a web-based IDE for WebAssembly whereby users of the IDE do not require knowledge about the underlying complexities of the WebAssembly development workflow in order to be able to build client-side applications that leverage inter-operation between JavaScript and WebAssembly. The project has successfully achieved this goal. The 'Projects IDE' provides users who may be unfamiliar with the WebAssembly development workflow a way to quickly get started creating simple websites using WebAssembly from within a controlled development environment. The 'WebContainer Playground' provides a WebContainer-based [74] development environment targeted towards web applications that utilise WebAssembly. It allows both newer developers and more experienced ones to create web applications using WebAssembly inside of an extensible Node.js environment.

The majority of the requirements (Chapter 4) we set out to accomplish at the start of this project have been successfully met. Future work (Section 9.3) will be carried out to fulfil the requirements we have not yet met for the 'WebContainer Playground', the most important of which is the project management system as already exists in the 'Projects IDE'.

We note that the implementation for both the 'WebContainer Playground' and 'Projects IDE' is thoroughly tested with end-to-end tests (Chapter 7) to ensure consistency and correctness in our implementation. These tests provide proof of requirements for the project being met in the form of screenshots and recordings attached to the published Playwright report.

### 9.1.2 Peer Review

We have received feedback for the project from several professionals in the field, including developers working at Stackblitz [71] and members of the community associated with their open-source projects. This feedback has been overwhelmingly positive and especially focused on the novelty and utility of this specific use-case of WebContainers [74] in providing a web-based IDE focused on the WebAssembly development workflow. Moving forward, we plan to use this feedback to further refine our WebContainers-based web IDE, with a focus on improving its usability and extending its functionality to better support the WebAssembly development workflow. Additionally, the positive feedback we received will be used to support future development opportunities for this project. The feedback received from professionals in the field will be invaluable in shaping the future direction of this project and its impact on the wider industry.

## 9.2 Lessons Learned

Building this project has taught me a variety of important and invaluable skills that will aid in the future development of both this project and all future projects I undertake as a software engineer.

### 9.2.1 WebAssembly Development and Workflow

To successfully meet the requirements for this project, I had to gain a comprehensive understanding of the WebAssembly development workflow, its use cases, and integration processes in web applications. This involved learning how to compile AssemblyScript and Go programs into WebAssembly, as well as finding the most optimal methods for doing so. Additionally, I had to familiarize myself with how WebAssembly modules are loaded into the browser for inter-operation with JavaScript. The knowledge I have gained in this field has been invaluable in guiding the development of this project and will continue to guide all future work (Section 9.3) related to it.

### 9.2.2 Project Management & Planning

Managing a project from start to finish requires a lot of planning and organisation. I learned how to break down a project into smaller tasks, estimate the duration of tasks, and create a timeline for completion. I also became familiar with agile development methodologies and learned how to adapt to changing requirements and priorities as new technologies and solutions arose.

One of the most important lessons learned from this project is the critical role of continuous feedback from my supervisor. Throughout the project, I made a concerted effort to engage with my supervisor on a regular basis, seeking feedback on my progress and incorporating their input into my decision-making process. This approach allowed me to identify and address issues early on, ensuring that the project stayed on track and met the expectations set out for its success. Moving forward, I recognize the importance of maintaining this feedback loop and will continue to prioritize regular check-ins with stakeholders in all future projects.

### 9.2.3 IDE Design & Development

In building this project, I have had to familiarize myself with the complex subject of IDE design and development. Specifically, I have gained an understanding of the design and development of web-based IDEs and the ways in which these solutions should emulate the look and feel of traditional local IDEs so they are familiar to developers.

The text editor for the IDE uses the 'monaco-editor' npm package [48] which provides an extremely familiar experience to users whilst also being extensible to the IDE developer. I was required to develop a firm understanding of this package and how it could be utilised to create web-based editors similar to that of Visual Studio Code [54]. Furthermore, I was required to discover how the functionality of the editor could be altered and extended in order to fit the requirements of a web-based IDE focused on WebAssembly development.

My expertise in IDE design and development will be highly valuable for all upcoming future work (Section 9.3) related to this project.

### 9.2.4 Maintenance of Large Distributed Applications

Developing this application gave me valuable experience in managing all the moving parts of a complex system. Since I was responsible for both the front-end and back-end, I had to juggle and maintain numerous components, all of which posed the risk of introducing

possible points of failure. This experience taught me the importance of understanding how all the pieces fit together to create a cohesive and reliable system.

One critical lesson I learned from this project was the importance of testing, specifically end-to-end testing from the start to catch regressions and provide proof of a working system. I made sure to conduct regular automated and manual tests throughout the development process, which helped ensure the reliability of the system. These tests allowed me to catch issues early on before they could escalate into larger problems.

Overall, building and deploying this large application taught me extremely useful lessons about managing complex systems and continuously testing for reliability. These lessons will undoubtedly prove useful in future projects as I continue to develop my skills as a software engineer.

### 9.2.5 Continuous Integration & Deployment

In order to set up a continuous integration and deployment pipeline for this project, I learned how to utilise GitHub Actions and develop workflows [36] which would be run on each push to the GitHub repository. Using GitHub Actions, I was able to create systems to automate the testing and deployment steps of the project for both the web application and API.

As mentioned we made use of Docker and containerisation as part of our continuous integration and deployment pipelines. This a vital skill which will allow me to develop and containerise applications so that they may be easily deployed to any environment.

Learning how to use services such as Railway and Vercel to automate the deployment processes for this project saved me significant time and effort. These tools allowed me to easily set up pipelines which could be integrated with GitHub Actions. This streamlined the development process and helped ensure the stability of the system. Automating the deployment process was particularly beneficial since it allowed me to focus more on developing new features and fixing issues, rather than worrying about deployment logistics.

The skills I've developed surrounding 'DevOps' and continuous integration and deployment pipelines will prove extremely useful to me in the future as I progress into professional environments where continuous integration and deployment are a key part of the workflow of teams working in agile environments.

## 9.3 Future Work

In implementing this project, we have successfully met most of the requirements we initially set out to accomplish. However, there are a variety of improvements that should be made to improve the user experience and functionality of the application as a whole.

After evaluating both the 'Projects IDE' and the 'WebContainer Playground' it is clear the 'WebContainer Playground' is a far more complete IDE experience than the 'Projects IDE' in addition to being both more secure and more performant. For this reason, future work will be focused on improving the 'WebContainer Playground' and making it more useful for building larger and more complicated projects.

### 9.3.1 Authentication and Project Management

The 'WebContainer Playground' provides a simple way for users to continue working on their projects using a 'share' link, however, this is not appropriate for building larger projects over longer periods of time. For this reason, the 'WebContainer Playground' should be adapted to integrate 'projects' similar to the 'Projects IDE' whereby users can sign in, create new WebContainer-based projects and come back to them at a later time for continued work.

### 9.3.2 Documentation Website

Users should be provided with a documentation website with relevant links to AssemblyScript documentation, as well as information regarding all possible actions that they may be performed within the IDE and instructions on how to perform them. At this point, the implementation and demonstration chapters (Chapter 6, Chapter 8) of this report are the most complete form of documentation for the application, these chapters should be adapted and condensed into a user-facing documentation website to show users how to use and create websites using AssemblyScript and WebAssembly in the IDE. In addition, we should provide examples to users on the documentation website for use cases of the IDE, some example projects using the 'WebContainer Playground' are provided on the playground's examples page (`https://wasm-web-ide-client.vercel.app/playground/examples`).

### 9.3.3 Improved Language Support

As discussed, the editor for the 'WebContainer Playground' currently supports type hints and auto-completion for AssemblyScript and JavaScript. In addition to providing type-hints for exported functions in the user's compiled WebAssembly module. However, JavaScript language support requires some improvements to allow users to receive type hints when using external packages from NPM, in addition to when importing files other than the `out/module.js` file containing the JavaScript code to instantiate the user's WebAssembly module. We will improve the JavaScript language support in the editor to provide type support for all installed NPM packages, as well as any file in the WebContainer environment that may be imported by a user in their JavaScript file. In addition, we should implement syntax-highlighting support for the WebAssembly Text Format (WAT).

### 9.3.4 File Operations

The IDE allows users to create new files and delete them but lacks support for renaming files, uploading files to the IDE and moving files between folders. This should be implemented to give users more freedom about how their projects are structured.

### 9.3.5 Terminal Improvements

The IDE currently supports an output-only terminal which prints standard output/error from all processes running in the WebContainer. The terminal does not provide the user with a way to run arbitrary commands within the WebContainer. This should be remedied through the use of a package such as `jshell` [39] which allows the user to interact with the file system and run commands in the Node.js WebContainer environment.

We should allow users to have multiple terminal 'tabs' running at once. By default, the Vite development server should be started in the first terminal 'tab' when the WebContainer is first set up, but users should be able to open a new tab and run commands using `jshell` inside of their WebContainer. Users should then be able to navigate between these terminal 'tabs'.

### 9.3.6 User Testing

User Testing has not thus far been possible during the development of the application due to frequently changing requirements, the niche field chosen for this project, and the existence of the two separate IDEs in the 'WebContainer Playground' and the 'Projects IDE'. User testing is essential and should be carried out with the 'WebContainer Playground' so that feedback may be gathered regarding the user experience and usability of the application in order for further iteration on design and functionality to take place.

### 9.3.7 Outlook for Future Work

In building this project, it has become clear that the WebAssembly-based 'WebContainer' operating system represents a paradigm shift in the Web-Based IDEs space. The 'WebContainer' project and those using similar approaches will likely, over time, make the need for the client-server model as present in the 'Projects IDE' and other more traditional web-based IDEs somewhat obsolete. This is due to the impressive performance of WebContainers, the cost associated with running them in comparison to the cost of running servers performing expensive computations such as compilation, as well as the security and isolation they provide. It's important to note that 'WebContainers' themselves represent a novel use-case of WebAssembly and would not be possible to implement securely and efficiently without the use of WebAssembly. WebAssembly itself is still a relatively new technology when compared with much more mature web technologies such as JavaScript. Recent developments in WebAssembly (Chapter 3) such as the WebContainer project indicate that novel-use cases of WebAssembly will continue to arise as WebAssembly and the developer ecosystem surrounding it matures and improves. Based on our experience building and deploying the 'Projects IDE,' we recommend that developers consider leveraging WebContainers for their own applications to benefit from the efficiency and security they provide. Overall, developing the 'Projects IDE' and 'WebContainer Playground' has been a rewarding project that has greatly expanded our understanding of cutting-edge technologies and we look forward to further exploring the possibilities of WebAssembly and web-based IDEs for WebAssembly in future work.

# Bibliography


[1] appcypher. Awesome WASM Runtimes. https://github.com/appcypher/awesome-wasm-runtimes. 1

[2] ashleygwilliams. Webassembly studio issues: clarify maintenance status of this project. https://github.com/wasdk/WebAssemblyStudio/issues/381. 9

[3] AssemblyScript. Assemblyscript editor. https://www.assemblyscript.org/editor.html. 9, 29

[4] AssemblyScript. Assemblyscript types. https://www.assemblyscript.org/types.html. 10, 29

[5] assmeblyscript. using the compiler — assemblyscript - configuration files. https://www.assemblyscript.org/compiler.html#configuration-file. 37, 41

[6] Suren Atoyan. @monaco-editor/react. https://www.npmjs.com/package/@monaco-editor/react. 27

[7] Go Authors. A tour of go — goroutines. https://go.dev/tour/concurrency/1. 20

[8] The Go Authors. testing package - testing - go packages. https://pkg.go.dev/testing. 52

[9] AWS. aws/aws-sdk-go. https://github.com/aws/aws-sdk-go. 20, 52

[10] AWS. Organizing objects in the amazon s3 console using folders. https://docs.aws.amazon.com/AmazonS3/latest/userguide/using-folders.html. 20

[11] AWS. S3 cloud storage. https://aws.amazon.com/s3/. 20

[12] AWS. Sharing objects using presigned urls. https://docs.aws.amazon.com/AmazonS3/latest/userguide/ShareObjectPreSignedURL.html. 21, 33

[13] AWS. What is an ide? https://aws.amazon.com/what-is/ide/. 2

[14] colinhacks. colinhacks/zod. https://github.com/colinhacks/zod. 26

[15] Axios Contributors. Interceptors. https://axios-http.com/docs/interceptors. 26

[16] Axios Contributors. Promise based http client for the browser and node.js. https://axios-http.com/. 26

[17] MDN Contributors. Blob - web apis — mdn. https://developer.mozilla.org/en-US/docs/Web/API/Blob. 51

[18] MDN Contributors. Context menu event. https://developer.mozilla.org/en-US/docs/Web/API/Element/contextmenu_event. 40

[19] MDN Contributors. Cross-origin-embedder policy. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cross-Origin-Embedder-Policy. 37

[20] MDN Contributors. Cross-origin-opener policy. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cross-Origin-Opener-Policy. 37

[21] MDN Contributors. Cross-origin resource sharing (cors). https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS#what_requests_use_cors. 24

[22] MDN Contributors. iframe: The inline frame element. https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe. 30, 46

[23] MDN Contributors. Shared array buffer. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer. 37

[24] MDN Contributors. Webassembly.instance. https://developer.mozilla.org/en-US/docs/WebAssembly/JavaScript_interface/Instance. 30

[25] MDN Contributors. Webassembly.instatiatestreaming(). https://developer.mozilla.org/en-US/docs/WebAssembly/JavaScript_interface/instantiateStreaming. 30

[26] MDN Contributors. Window.localstorage - web apis. https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage. 28

[27] MDN Contributors. Window.postmessage(). https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage. 32

[28] NPM contributors. npm. https://www.npmjs.com/. 22, 44, 70

[29] React Contributors. Passing props to a component - react. https://react.dev/learn/passing-props-to-a-component. 50

[30] React Contributors. React. https://react.dev/. ix, 25, 46

[31] TailwindCSS Contributors. Tailwind css - rapidly build modern websites without ever leaving your html. https://tailwindcss.com/. 25

[32] Contributors Emscripten. Emscripten website. https://emscripten.org/index.html. 1

[33] OpenJS Foundation. Node.js. https://nodejs.org/en. 70

[34] gin contrib. gin-contrib/cors: Cors gin's middleware. https://github.com/gin-contrib/cors. 24

[35] gin gonic. Gin web framework. https://gin-gonic.com/. 19

[36] GitHub. Github actions. https://github.com/features/actions. 52, 53, 54, 73

[37] The PostgreSQL Global Development Group. Postgresql: The world's most advanced open source relational database. https://www.postgresql.org/. 20

[38] WebAssembly Working Group. Webassembly. https://webassembly.org/. 1

[39] Brady M. Holt. jsh - npm. https://www.npmjs.com/package/jsh. 74

[40] IETF. Json web tokens. https://jwt.io/. 19

[41] jinzhu. go-gorm/gorm: The fantastic orm library for golang, aims to be developer friendly. https://github.com/go-gorm/gorm. 20, 52

[42] Andrew Klotz. Smallest golang docker image. https://klotzandrew.com/blog/smallest-golang-docker-image. 24

[43] C. Larman and V.R. Basili. Iterative and incremental developments. a brief history. *Computer*, 36(6):47–56, 2003. 18

[44] Magnum. 3D Fluid Simulation Example. https://magnum.graphics/showcase/fluidsimulation3d/. 1

[45] Microsoft. Declarations file: Introduction. https://www.typescriptlang.org/docs/handbook/declaration-files/introduction.html. 29

[46] Microsoft. Fast and reliable end-to-end testing for modern web apps — playwright. https://playwright.dev/. 53

[47] Microsoft. Language server protocol. https://microsoft.github.io/language-server-protocol/. 29

[48] Microsoft. Monaco Editor. https://github.com/microsoft/monaco-editor. 9, 10, 17, 27, 33, 36, 72

[49] Microsoft. Page object models — playwright. https://playwright.dev/docs/pom. 53

[50] Microsoft. Parameterize tests — playwright. https://playwright.dev/docs/test-parameterize. 57

[51] Microsoft. The repository pattern. https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design. 21

[52] Microsoft. Trace viewer — playwright. https://playwright.dev/docs/trace-viewer. 53

[53] Microsoft. Typescript: Javascript with syntax for types. https://www.typescriptlang.org/. 11, 22

[54] Microsoft. Visual studio code. https://code.visualstudio.com/. 16, 27, 72

[55] Mozilla. Understanding WebAssembly text format. https://developer.mozilla.org/en-US/docs/WebAssembly/Understanding_the_text_format, 09 2022. 4, 8, 14, 20, 23

[56] nfriedly. bestzip - npm. https://www.npmjs.com/package/bestzip. 51

[57] npm. npx - npm. https://www.npmjs.com/package/npx. 51

[58] p5js. p5.js editor. https://editor.p5js.org/. 3

[59] Poimandres. @monaco-editor/react. https://github.com/pmndrs/zustand. 27

[60] Railway. Postgresql. https://docs.railway.app/databases/postgresql. 23

[61] Railway. Railway. https://railway.app/. 23, 48

[62] Railway. Using github actions with railway. https://blog.railway.app/p/github-actions. 24

[63] Naveen Ramanathan. Webassembly: Introduction to webassembly using go. https://golangbot.com/webassembly-using-go/. 21

[64] Redis. Redis. https://redis.io/. 48

[65] Pouya Saadeghi. daisyui components: Use tailwind css but write fewer class names. https://daisyui.com/. 25

[66] Eric Simons. Introducing webcontainers: Run node.js natively in your browser. https://blog.stackblitz.com/posts/introducing-webcontainers/, 05 2021. 11

[67] Stackblitz. Api reference — webcontainers. https://webcontainers.io/api. 41

[68] Stackblitz. Api reference — webcontainers - filesystemtree. https://webcontainers.io/api#filesystemtree. 37, 48

[69] Stackblitz. Api reference — webcontainers - spawn. https://webcontainers.io/api#%E2%96%B8-spawn. 38

[70] Stackblitz. Configuring headers. https://webcontainers.io/guides/configuring-headers. 37

[71] Stackblitz. Stackblitz. https://stackblitz.com/. 11, 16, 71

[72] Stackblitz. Webcontainer api is here. https://blog.stackblitz.com/posts/webcontainer-api-is-here/. 12

[73] StackBlitz. Working with the file system — webcontainers. https://webcontainers.io/guides/working-with-the-file-system. 39

[74] Stackblitz. Webcontainers api. https://webcontainers.io, 2 2023. 1, 14, 19, 36, 71

[75] Stackblitz. Webcontainers api tutorial. https://webcontainers.io/tutorial/1-building-your-first-webcontainers-app, 02 2023. 1

[76] CNCF Staff. wasm microsurvey: a transformative technology, yes, but time to get serious. https://www.cncf.io/blog/2022/10/24/cncf-wasm-microsurvey-a-transformative-technology-yes-but-time-to-get-serious/. viii, 1, 2, 3

[77] Swagger. Bearer authentication. https://swagger.io/docs/specification/authentication/bearer-authentication/. 19

[78] TanStack. Devtools. https://tanstack.com/query/v4/docs/react/devtools. 26

[79] TanStack. Mutations. https://tanstack.com/query/v4/docs/react/guides/mutations. 26

[80] TanStack. Queries. https://tanstack.com/query/v4/docs/react/guides/queries. 26

[81] TanStack. Query invalidation. https://tanstack.com/query/v4/docs/react/guides/query-invalidation. 26, 33

[82] TanStack. Tanstack query. https://tanstack.com/query/latest. 26

[83] TanStack. Tanstack query. https://tanstack.com/query/v4/docs/react/reference/QueryClient. 26, 33, 40

[84] AssemblyScript Team. AssemblyScript. https://www.assemblyscript.org/. 1, 4, 9, 13, 14, 21, 22, 36

[85] Go Team. gopls command. https://pkg.go.dev/golang.org/x/tools/gopls. 29

[86] React Core Team. Next.js by vercel - the react framework for the web. https://nextjs.org/. 25

[87] React Core Team. Reusing logic with custom hooks. https://react.dev/learn/reusing-logic-with-custom-hooks#extracting-your-own-custom-hook-from-a-component. 26

[88] The Go Authors. syscall/js - Go Packages. https://pkg.go.dev/syscall/js, 10 2022. 36

[89] TinyGo. Differences from go. https://tinygo.org/docs/concepts/compiler-internals/differences-from-go/. 21

[90] TinyGo. Using WebAssembly — TinyGo. https://tinygo.org/docs/guides/webassembly/, 02 2023. 1, 4, 13, 21, 30, 62

[91] V8 Team. Firing up the Ignition interpreter. https://v8.dev/blog/ignition-interpreter, 08 2016. 1

[92] Vercel. Data fetching: getserversideprops — next.js. https://nextjs.org/docs/basic-features/data-fetching/get-server-side-props. 50

[93] Vercel. Deploying github projects with vercel. https://vercel.com/docs/concepts/deployments/git/vercel-for-github. 35

[94] Vercel. Next.js by vercel - the react framework for the web. https://nextjs.org/. 25

[95] Vercel. Next.js compiler. https://nextjs.org/docs/advanced-features/compiler. 25

[96] Vercel. Vercel: Develop. preview. ship. for the best frontend teams. https://vercel.com/home. 35, 36, 53, 54

[97] Vite. Features — vite. https://vitejs.dev/guide/features.html. 11

[98] Vite. Vite: Next generation frontend tooling. https://vitejs.dev/. 11, 14, 37, 47

[99] W3C. Webassembly working group. https://www.w3.org/wasm/. 1

[100] Evan Wallace. Webassembly cut figma's load time by 3x. https://www.figma.com/blog/webassembly-cut-figmas-load-time-by-3x/, 06 2017. 1

[101] wasdk. Github — webassembly studio. https://github.com/wasdk/WebAssemblyStudio/. 9

[102] wasdk. Wasm fiddle. https://wasdk.github.io/WasmFiddle/. 7

[103] wasdk. Webassembly studio. https://webassembly-studio.kamenokosoft.com/. 8

[104] wasm-pack contributors. https://github.com/rustwasm/wasm-pack. 1

[105] WebAssembly. Is WebAssembly trying to replace JavaScript? FAQ - WebAssembly. https://webassembly.org/docs/faq/. 1

[106] WebAssembly. Use cases. https://webassembly.org/docs/use-cases/. 1

[107] WebAssembly. Webassembly/wabt: The webassembly binary toolkit. https://github.com/WebAssembly/wabt. 23

[108] Gavin Wright. What is debouncing? - techtarget. https://www.techtarget.com/whatis/definition/debouncing. 40