Gizual Data Layer:
Enabling Browser-Based Exploration of Git Repositories

Stefan Schintler

Gizual Data Layer:
Enabling Browser-Based Exploration of Git Repositories

Stefan Schintler B.Sc.

Master's Thesis

to achieve the university degree of

Diplom-Ingenieur

Master's Degree Programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Ao.Univ.-Prof. Dr. Keith Andrews
Institute of Interactive Systems and Data Science (ISDS)

Graz, 09 Dec 2024

Gizual Datenschicht:
Browserbasierte Verarbeitung von Git-Repositories

Stefan Schintler B.Sc.

Masterarbeit

für den akademischen Grad

Diplom-Ingenieur

Masterstudium: Informatik

an der

Technischen Universität Graz

Begutachter

Ao.Univ.-Prof. Dr. Keith Andrews
Institute of Interactive Systems and Data Science (ISDS)

Graz, 09 Dec 2024

Diese Arbeit ist in englischer Sprache verfasst.

Abstract

Gizual is a state-of-the-art, open-source, web application for visually exploring the source code and metadata of files in a Git repository. This thesis describes Gizual's data layer, which is implemented in TypeScript, Rust, and WebAssembly. Operations are performed directly in the web browser, rather than server-side, to ensure data privacy.

File access and handling is achieved through a custom File I/O layer, which leverages a mixture of modern web browser File APIs to simulate a native file system and expose it to WebAssembly. Git operations are performed using a Rust wrapper around the libgit2 library, which itself is compiled to run inside the browser with WebAssembly. An innovative software architecture is implemented to distribute workloads across pools of web workers, which are synchronised using a custom communication protocol.

Kurzfassung

Gizual ist eine moderne, Open-Source Webanwendung zur Visualisierung von Quellcode und dessen Metadaten aus einem Git-Repository. Diese Arbeit beschreibt die Datenschicht von Gizual, die in TypeScript, Rust und WebAssembly implementiert ist. Die Operationen werden direkt im Webbrowser und nicht serverseitig ausgeführt, um Datenschutz zu gewährleisten.

Der Datenzugriff und die Verarbeitung erfolgen durch die Nutzung moderner Web-APIs, um ein natives Dateisystem zu simulieren und es für WebAssembly verfügbar zu machen. Git-Operationen werden mit der libgit2-Bibliothek unter Zuhilfenahme einer in Rust implementierten Zwischenschicht durchgeführt, die im Webbrowser durch WebAssembly nutzbar wird. Eine innovative Softwarearchitektur wird eingesetzt, um Arbeitslasten über eine Vielzahl von Web Workern zu verteilen, die mithilfe eines benutzerdefinierten Kommunikationsprotokolls synchronisiert werden.

Contents

List of Figures

List of Tables

List of Listings

Acknowledgements

This thesis could not have been completed without the assistance of several people. I sincerely thank my thesis advisor, Prof. Keith Andrews, for his mentorship and expertise, which significantly enhanced the quality of this work beyond what I could have achieved independently. I am also thankful to my colleague and friend, Andreas Steinkellner, for collaborating on the Gizual project and motivating me to finish my studies. I am especially grateful to my parents, Birgit and Michael, who made this journey possible through their unwavering support, allowing me to pursue an academic education in a field of my choice. I also extend my heartfelt gratitude to my extended family and friends, whose continuous encouragement has been invaluable throughout this endeavour.

Stefan Schintler

Graz, Austria, 09 Dec 2024

Credits

I would like to thank the following individuals and organisations for permission to use their material:

- The thesis was written using Keith Andrews' skeleton thesis [Andrews 2021].
- Chapter 5 and Appendix A were written jointly with Andreas Steinkellner.

Chapter 1

Introduction

The modern web serves as a platform for building and sharing applications across a multitude of devices, allowing users to access and interact with information and services from anywhere at any time. Web browsers are the frontend to the web. They act as gatekeepers, ensuring device security, maintaining privacy, and providing a seamless interaction with the web. Browser vendors aim to protect users from potentially untrusted code that could compromise their systems. However, these security measures can pose challenges for modern web-based applications that require underlying operating system features.

This thesis explores the challenges, workarounds, and opportunities in creating the data layer for Gizual, an innovative web-based application for visualizing Git repository source code and its metadata. The data layer provides data access for the frontend, handling data processing and interacting with the Web APIs. Unlike traditional backends, Gizual's data layer runs locally on the client side, inside the web browser next to the user interface. This local execution enhances user experience by performing data processing locally, minimising computational resource demands, and preserving privacy, as no data is sent remotely.

Gizual is an open-source web application, developed with TypeScript and Rust. It is inspired by SeeSoft [Eick et al. 1992] and aims to reimagine Git code repository visualisation. The source code of Gizual is open-source and available on GitHub [Schintler and Steinkellner 2024b]. A deployed version can be accessed at `gizual.com`. Whereas this thesis [Schintler 2024] focuses on implementing Gizual's data layer to enable browser-based exploration of Git repositories, the companion thesis by Andreas Steinkellner [Steinkellner 2024] focuses on Gizual's user interface.

Beneath the user interface, Gizual's data layer utilises advanced WebAssembly technology to execute Git operations directly in the browser. A combination of the C library libgit2, custom implementations for file I/O, and efforts to parallelise these operations across web workers yields a highly performant in-browser data layer for Gizual. Gizual's innovative methods of leveraging browser performance while preserving privacy are key focuses of this thesis, setting the stage for further development and improvements in local-first web applications.

Chapters 2 to 4 provide a comprehensive overview of the fundamental web technologies. Chapter 2 introduces the technical foundation of the programming languages and protocols used. Chapter 3 outlines modern web browser functionalities. Chapter 4 discusses the Git version control system and related tools. Chapter 5 details Gizual's architecture. Chapter 6 explores techniques for processing large local repositories with browser APIs and required workarounds. Chapter 7 addresses executing native Rust code in browsers and Gizual's solutions. Chapter 8 examines Gizual's multi-threading approach and performance bottlenecks. Finally, Chapter 9 provides insights into potential opportunities for future work on Gizual's data layer.

Chapter 2

Web Technologies

Gizual performs data extraction and processing entirely on the client-side, eliminating the need for additional software installations or for uploading any user data to a server. This functionality is made possible by leveraging modern web technologies to craft a seamless, web-based application. This chapter describes the various web technologies upon which Gizual is based.

2.1 Hypertext Markup Language (HTML)

The Hypertext Markup Language (HTML) defines the structure of a web page to be rendered by a web browser. It was first described in the proposal for Information Management by Berners-Lee [1989]. The first versioned release of HTML, Version 2.0, was announced by Berners-Lee and Connolly [1993]. Since Version 3.2 of the HTML standard, the World Wide Web Consortium (W3C) has been the only publisher of the HTML standard, continuously releasing new versions of the specification. After the release of HTML5 on 28 Oct 2014, conflicts arose between the Web Hypertext Application Technology Working Group (WHATWG) and the W3C, leading to an agreement on 28 May 2019, for WHATWG to take over the stewardship [W3C 2019]. Since then, HTML has continued as a *living standard* under WHATWG's guidance, with continuous updates and extensions introducing new features and capabilities. Figure 2.1 shows some of the more commonly used HTML elements.

Web browsers use HTML to render web pages by interpreting the tags and their attributes to display text, images, links, and other media. It can also be used to create forms, tables, lists, and other items to structure the content of a web page semantically. Styling and layout of the content are typically done using Cascading Style Sheets (CSS), and interactivity is added using JavaScript (JS). A simple example of HTML5 code is shown in Listing 2.1.

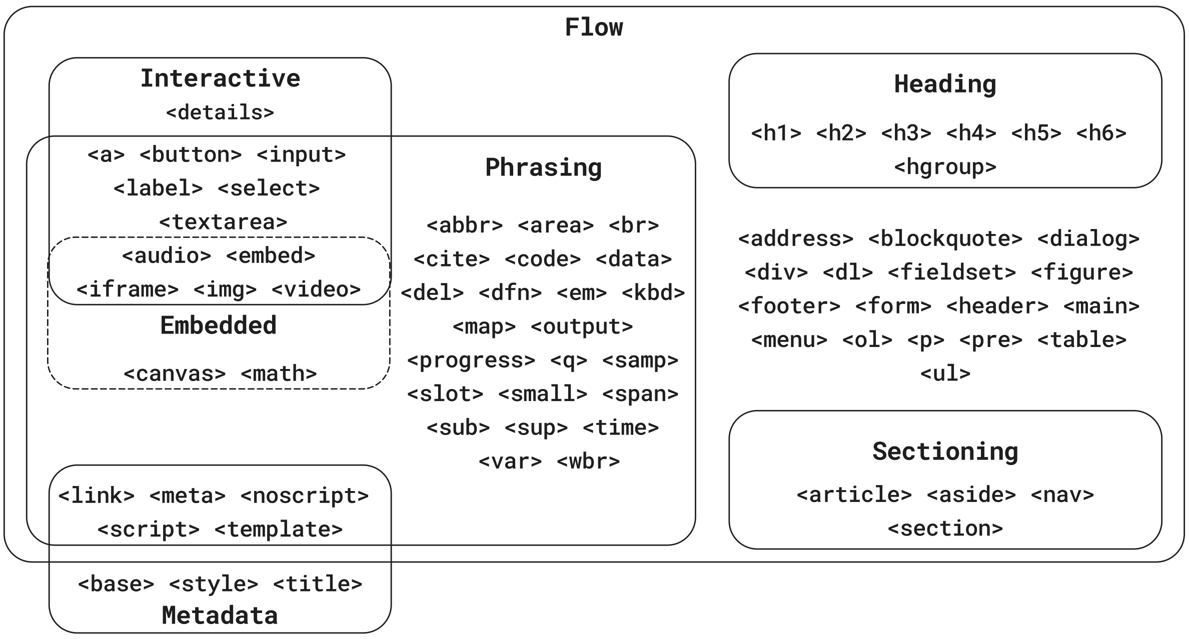


Figure 2.1: Some common HTML5 elements in their respective categories. [Diagram drawn by the author of this thesis, based on the diagram by Pe [2022].]

```
<!DOCTYPE html>
<html>

<head>
  <title>Hello World</title>
</head>

<body>
  <h1>This is a Title</h1>
  <p>This is an example paragraph.</p>
</body>

</html>
```

Listing 2.1: Simple HTML5 example code, specifying the structure and content of a web page.

2.2 Cascading Style Sheets (CSS)

Cascading Style Sheets (CSS) is a language initially proposed in 1994 by Håkon Wium Lie while working at CERN with Tim Berners-Lee [Lie 1994]. By 1996, the World Wide Web Consortium (W3C) released the first standardised version, CSS1, developed by Lie and Bert Bos. CSS is used to describe the presentation and styling of a document written in HTML. It enables control over layout, fonts, colours, and overall aesthetics and helps to maintain a consistent look and feel across multiple web pages, making it an essential tool for web design [Lie and Bos 2005].

A CSS style declaration comprises a selector, a property, and a value, as can be seen in Figure 2.2. Selectors have varying specificity to target elements in an HTML document and apply styles to them. The selectors can be based on element names, class names, IDs, or a combination thereof. The property and value pairs define the styling of the elements selected by the selector. For example, the CSS code in Listing 2.2 sets the font family, size, and colour of all paragraphs in a document to Arial, 12px, and blue.

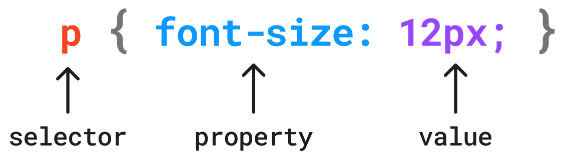


Figure 2.2: A CSS style declaration comprises a selector, a property, and a value. [Image created by the author of this thesis.]

```css
p {
  font-family: Arial;
  font-size: 12px;
  color: blue;
}
```

Listing 2.2: Simple CSS example code, specifying properties for the <p> element.

2.3 JavaScript (JS)

JavaScript is a high-level, interpreted scripting language introduced in 1995 by Netscape, initially known as LiveScript [Wirfs-Brock and Eich 2020]. On 04 Dec 1995, a joint press release by Netscape Communications Corporation and Sun Microsystems formally announced JavaScript [Netscape 1995]. It was described as an object scripting language designed to enhance web pages by allowing scripts to modify Java objects and complement Java for web application development. This branding strategy aimed to link JavaScript with Java to increase its popularity, even though Java and JavaScript do not share many similarities.

JavaScript was initially developed to incrementally enhance the user experience of web pages by adding lightweight interactivity and dynamic content after the page has been loaded. Due to trademark issues, the standardized version of JavaScript is called ECMAScript and is maintained by the organization Ecma International. The latest version is ECMAScript 2024 (also known as ES15) [Ecma 2024]. Today, JavaScript has become a versatile language that can be used for server-side programming, desktop applications, and mobile apps, as well as web applications.

In web browsers, JavaScript is embedded in the HTML code using the `<script>` element either inline or as a reference to an external JavaScript file. JavaScript can be used to manipulate the HTML content of a web page, handle user input, create animations, fetch data from a server, and much more. To accomplish this, JavaScript has access to a wide range of APIs provided by the web browser. JavaScript is a runtime type-safe language and as such does not support static type checking out of the box [Flanagan 2020]. A simple example of JavaScript code is shown in Listing 2.3.

```javascript
// This is a constant variable which cannot be reassigned
const receiver1 = "reader!";

// This is an arrow function
const sayHello = (receiver) => {
  // Using a template string to create a string with a variable
  let stmt = `Hello ${receiver}!`;
  console.log(stmt);
};

class ExampleClass {
  // This is a class variable
  memberVariable = "Hello!!";

  // This is a class method
  screamHello() {
    console.log(this.memberVariable);
  }
}

// This is a function call
sayHello(receiver1);

const instance = new ExampleClass();
instance.screamHello();

// Log Output:
// Hello reader!
// Hello!!
```

Listing 2.3: Simple JavaScript example code, showing the use of variables, classes, methods, and the `console.log()` function.

2.4 TypeScript (TS)

TypeScript is a statically typed superset of JavaScript which transpiles to pure JavaScript code. Version 1.0 of TypeScript was released in 2014 by Microsoft [Turner 2014]. It adds optional static typing to JavaScript, which aids in catching errors at transpile time, as well as providing better code completion and navigation in modern Integrated Development Environments (IDEs) like Visual Studio Code [Microsoft 2024c]. TypeScript is designed to be compatible with existing JavaScript code and libraries, which makes it easy to integrate into existing projects. To get started with TypeScript, the official web site provides an interactive tutorial and a playground to test TypeScript code online [Microsoft 2024b]. For a more in-depth introduction to TypeScript, the book by Baumgartner [2023] is recommended. An example of TypeScript code is shown in Listing 2.4. The transpilation of TypeScript code to JavaScript can be done using one of the following tools:

- *tsc*: The default TypeScript compiler provided by Microsoft.
- *babel*: A JavaScript compiler mostly known for transpilation of modern JavaScript features to older versions of JavaScript, though it can also transpile TypeScript code using the *@babel/preset-typescript* preset [Babel 2024].
- *swc*: The Speedy Web Compiler (SWC) created by @kdy1 [2024] is written in Rust and is used by popular frameworks like Next.js.
- *esbuild*: A transpiler focused on performance, created by Wallace [2024].

```
interface Greeter {
  createGreeting(receiver: string): string;
}

class EnglishGreeter implements Greeter {
  createGreeting(receiver: string): string {
    return `Hello ${receiver}!`;
  }
}

const greeter: Greeter = new EnglishGreeter();

const greeting: string = greeter.createGreeting("TypeScript");

console.log(greeting); // Hello, TypeScript!
```

Listing 2.4: Simple TypeScript example code, showing the use of interfaces, classes, and type annotations.

```
// This is a comment

// Function definition with a single string argument
fn say_hello(receiver: &str) {

  // Printing a message to the console
  println!("Hello, {}!", receiver);
}

fn main() {

  // Calling the function with a string argument
  say_hello("world");
}
```

Listing 2.5: Simple Rust example code, showing the use of variables, functions, and the `println!()` macro.

2.5 Rust

Rust is a systems programming language initially developed as a personal project by Graydon Hoare in 2006, while he was working for Mozilla [Thompson 2023]. Rust is designed with a focus on safety, concurrency, and performance. It is a statically typed language which offers memory safety without garbage collection. Rust is also known for its strict borrow checker, which enforces strict rules on how memory is used and shared between different scopes [Bugden and Alahmar 2022]. Rust is compiled into machine code and can be used for a variety of applications, including web applications, embedded systems, and operating systems. It also features a foreign function interface to enable interoperability with libraries written in C and C++ [Klabnik and Nichols 2023]. For web applications, Rust can be compiled to WebAssembly (WASM) to run in a web browser. To get started with Rust, the official web site provides a comprehensive guide and documentation, as well as the required software [RF 2024]. A simple example of Rust code is shown in Listing 2.5.

2.6 WebAssembly (WASM)

WebAssembly (WASM) is a modern binary instruction format for compiled languages like C, C++, Rust, and Go. It is designed to replace existing machine code for specific architectures with a portable, safe, and efficient format, which can be executed in a variety of environments with vastly different hardware and operating systems, as well as within modern web browsers [WACG 2024a]. As stated by Haas et al. [2017], WebAssembly was designed with the following goals in mind:

- *Safe*: WebAssembly is designed to be safe and sandboxed.
- *Fast*: WebAssembly is designed to be fast and efficient by leveraging compile-time optimisations.
- *Portable*: WebAssembly is designed to be a single compile target that can be executed on a variety of devices with different architectures and operating systems.
- *Compact*: WebAssembly is designed to be compact to reduce loading overhead, since one of the primary use cases is to be executed inside a web browser.

To get started with WebAssembly, the book by Sletten [2022] offers a great introduction. Typically, the software is first written in a low-level language like C or C++ and then compiled to WebAssembly using a compatible toolchain like `llvm` [LLVM 2024]. It can then be executed in a WebAssembly runtime, like an embedded runtime in another language, via the command line, or in a web browser. However, aside from the binary format, there is also a WebAssembly text format [WACG 2024b], which offers a human-readable representation of the binary format. A simple example of the WebAssembly text format is shown in Listing 2.6.

2.7 RPC Protocol

Remote Procedure Call (RPC) is a protocol that allows software to call procedures on a remote process or machine and receive the result. While the encoding and format of RPC messages can vary, the idea is to abstract the communication between different entities and software stacks in a structured way. JSON is the most common format for RPC communication between client and server in web-based applications. The specification for JSON-RPC can be found at JSON-RPC [2013]. A simple example of a JSON-RPC request and response is shown in Listing 2.7.

```
(module
  ;; Define the function signature with two parameters and a return type
  (func $add (param $a i32) (param $b i32) (result i32)

    ;; load the first parameter onto the stack
    local.get $a

    ;; load the second parameter onto the stack
    local.get $b

    ;; add the two values on the stack
    i32.add

    ;; return the result of the addition operation
    return
  )
  (export "add" (func $add))
)
```

Listing 2.6: Simple WebAssembly example code in text format, showing the definition of a function to add two integers and return the result.

```
// Request
{
  "jsonrpc": "2.0",
  "method": "add",
  "params": [30, 12],
  "id": 1
}

// Response
{
  "jsonrpc": "2.0",
  "result": 42,
  "id": 1
}
```

Listing 2.7: Simple JSON-RPC example for a procedure named `add` with two parameters `30` and `12`.

Chapter 3

The Modern Web Browser

Gizual uses modern web technologies to provide a performant experience. This chapter describes some basic concepts within a web browser, which are essential building blocks for complex applications with heavy local computation.

A web browser is a software application designed to interact with web servers over the internet. This is done by connecting to a server using the Hypertext Transfer Protocol (HTTP), downloading the corresponding page as Hypertext Markup Language (HTML) and displaying it to the user. Since the introduction of JavaScript in 1995, web browsers became not only a tool to display static content, but also an application platform for dynamic content and, indeed, entire applications [Wirfs-Brock and Eich 2020]. While many different web browsers are available, they are almost always based on one of three web browser engines:

- *Chromium and Blink*: The browsers Google Chrome, Microsoft Edge, Opera, Brave, Arc, and many others are based on Chromium and Blink [Chromium 2024b].
- *Firefox and Gecko*: The browsers Waterfox, LibreWolf, SeaMonkey, Tor Browser, and many others are based on Firefox and Gecko [Mozilla 2024a].
- *WebKit*: Apple Safari uses its own WebKit engine [Apple 2024].

As stated by Tamary and Feitelson [2015], since 2013 the Chromium project has become the most used web browser engine, with Google Chrome being the most popular web browser. Therefore, this chapter will focus mostly on the architecture of a Chromium-based web browser. Chromium consists of multiple processes, as shown in Figure 3.1. The main process is called the *browser process* and manages all other processes via Inter-Process Communication (IPC). This process also spawns a sub-process for each tab, called a *renderer process*. The renderer process is responsible for rendering the web page and executing any client-side code. Within the renderer processes, a rendering engine handles page layout and rendering. It is also responsible for executing client-side code. In all modern Chromium-based browsers, the rendering engine used is Blink [Chromium 2024a].

3.1 The Blink Rendering Engine

The Blink rendering engine was forked from WebKit by Google in 2013 [Seidel 2014]. It is used in all Chromium-based browsers, including Google Chrome, Microsoft Edge, and Opera. The rendering engine parses HTML and CSS, is responsible for the layout and rendering of web pages, and executes client-side code in a sandboxed environment [Hara 2018]. Blink uses the Skia Graphics Engine to render web pages to the user's screen [Google 2024]. While the name rendering engine suggests that it is only responsible for rendering web pages, it is also responsible for executing JavaScript and WebAssembly code, as well as handling user input events.

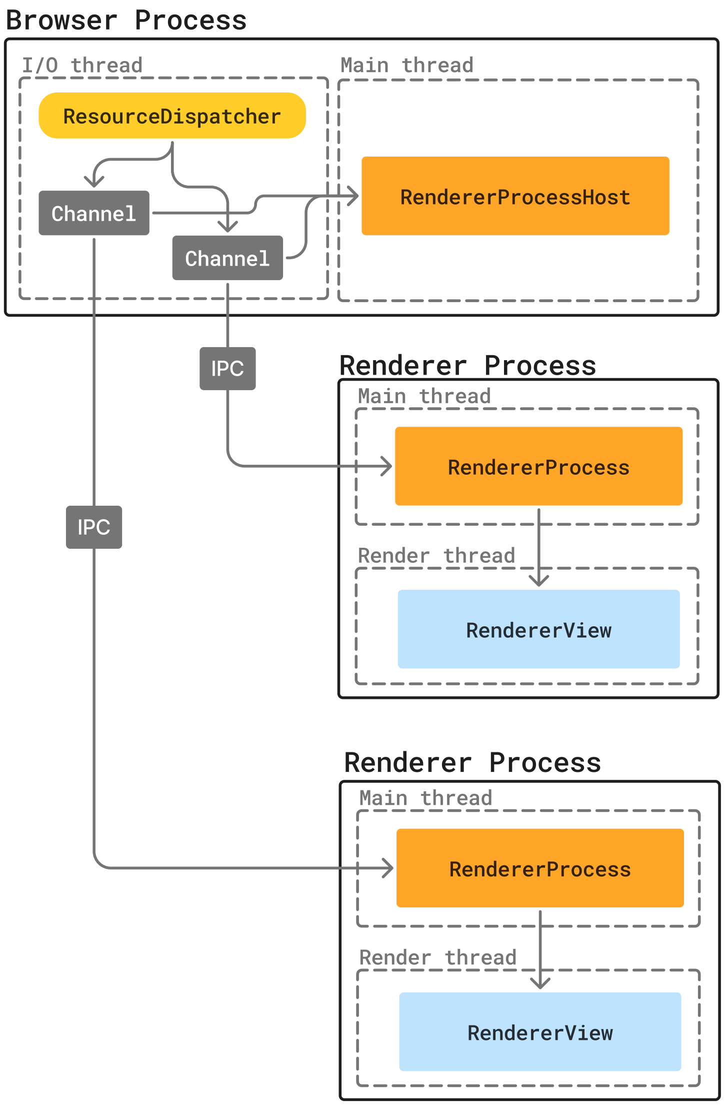


Figure 3.1: The simplified architecture of the Chromium web browser. [Diagram drawn by the author of this thesis, based on the original by Chromium [2024c] and used under the terms of the CC BY 2.5 license.]

3.2 Web APIs

Every web browser provides a set of APIs to call from JavaScript, in order to interact with the underlying operating system to access sensors, storage, and the network. A detailed list of available and upcoming APIs can be found at WHATWG [2024f]. Not all APIs are available in every browser, so feature detection is necessary to ensure compatibility, as shown in Listing 3.1. Only those APIs relevant for the implementation of Gizual's data layer will be discussed in the following sections.

3.2.1 Document Object Model (DOM)

The Document Object Model (DOM) describes a set of JavaScript interfaces to interact with a web page. The DOM represents the Hypertext Markup Language (HTML) as a hierarchical tree structure of nodes [WHATWG 2024a, Chapter 4.1]. Each node can be accessed and manipulated using JavaScript. The DOM also defines how events are handled and provides means to abort operations and activities [WHATWG 2024a, Chapter 3.2].

```
if ("storage" in navigator) {
  // Storage API is supported
  navigator.storage
    .getDirectory()
    .then(function (directory) {
      console.log("Storage directory:", directory);
    })
    .catch(function (error) {
      console.error("Storage error:", error);
    });
} else {
  console.error("Storage API is not supported");
}
```

Listing 3.1: A typical JavaScript feature detection clause to check for the availability of the `navigator.storage` API. If the API is available, the `navigator.storage.getDirectory()` function is called. Otherwise, an error message is logged to the console.

```
// main.js

const worker = new Worker("worker.js");

worker.onmessage = (event) => {
  // Will log "Message from worker: ABCD"
  console.log(`Message from worker: ${event.data}`);
  worker.terminate();
};

worker.postMessage("AB");

// worker.js

self.onmessage = (event) => {
  console.log(`Message from main: ${event.data}`);
  self.postMessage(`${event.data}CD`);
};
```

Listing 3.2: A simple example of how to create a web worker and communicate with it using the `postMessage()` function.

3.2.2 Web Workers

Within a tab in a browser, there is only one thread executing the main JavaScript code. Only JavaScript code running within this main thread can interact with the DOM. To enable parallelism, the Web Worker API was introduced within the HTML5 specifications [WHATWG 2024d, Chapter 10]. A web worker is a separate thread executing a JavaScript file defined during its creation, and represents a separate realm with its own global environment and scope. It can communicate asynchronously with other web workers or the main thread using the `postMessage()` function. Structured data like strings, numbers, or arrays are copied to the receiving scope, while resources containing transferable objects like ArrayBuffer and MessagePort are transferred without cloning [WHATWG 2024d, Section 2.7.2]. Alternatively, a SharedArrayBuffer can be used to share memory between multiple realms. A simple example of how to create a web worker and communicate with it is shown in Listing 3.2.

```
fetch("https://demo-url.com/data.json", {
  method: "GET",
  headers: {
    "Content-Type": "application/json",
  },
})
  .then((response) => response.json())
  .then((data) => {
    console.log(data);
  });
```

Listing 3.3: A simple example of how to use the Fetch API to request a resource from the network.

3.2.3 Fetch API

The Fetch API provides a means to request resources from the network using the HTTP protocol. It is a replacement for legacy APIs like `XMLHttpRequest` and provides a flexible way to handle responses. The Fetch API is promise-based and allows the chaining of multiple requests and responses. The API specification is defined by WHATWG [2024b]. A simple example of how to use the Fetch API is shown in Listing 3.3.

3.2.4 APIs for File I/O

File Input and Output (File I/O) is a common task in many applications, yet because of security concerns, browsers do not allow direct access to the user's file system. Instead, a set of APIs and conventions have been developed to provide a means to interact with data in a user's storage from within a web browser. The first approach was to use the `<input type="file">` element within a `<form>` element. It provides a way to select files from the user's storage to be uploaded to a server. The `type="file"` attribute for the `<input>` element is part of RFC 1867, which was proposed by Nebel and Masinter [1995] and later standardized in the HTML 3.2 specification [Raggett 1997]. The `type="file"` attribute is still widely used to upload files to a server. However, this mechanism alone does not provide the ability to read or write files on the client side. Accessing file data on the client side has several advantages, such as:

- Validating file data before uploading.
- Reducing server load by (pre-)processing data on the client side.
- Providing offline capabilities.
- Increasing privacy by not sending data to a server.

Over the years, various APIs have been developed to enable client-side file and folder access, as illustrated in Figure 3.2. The following sections describe seven different APIs for accessing the user's local file system.

3.2.4.1 File API

Initially proposed as the File Upload API by Berjon [2006], the File API is designed to provide a way to read file data from an `<input type="file">` element. This can be done by listening to the `change` event of the `<input>` element and accessing the `target.files` property contained in the event object. The API provides a `File` object which represents a file on the user's storage and provides access to the file's metadata like `name` and `size`. The `File` object also provides a means to read the file's content, since it inherits from the `Blob` object. A `Blob` object represents raw data and provides methods like `text()` and `arrayBuffer()` to read the data asynchronously. The File API is still widely used and is supported by all modern browsers.

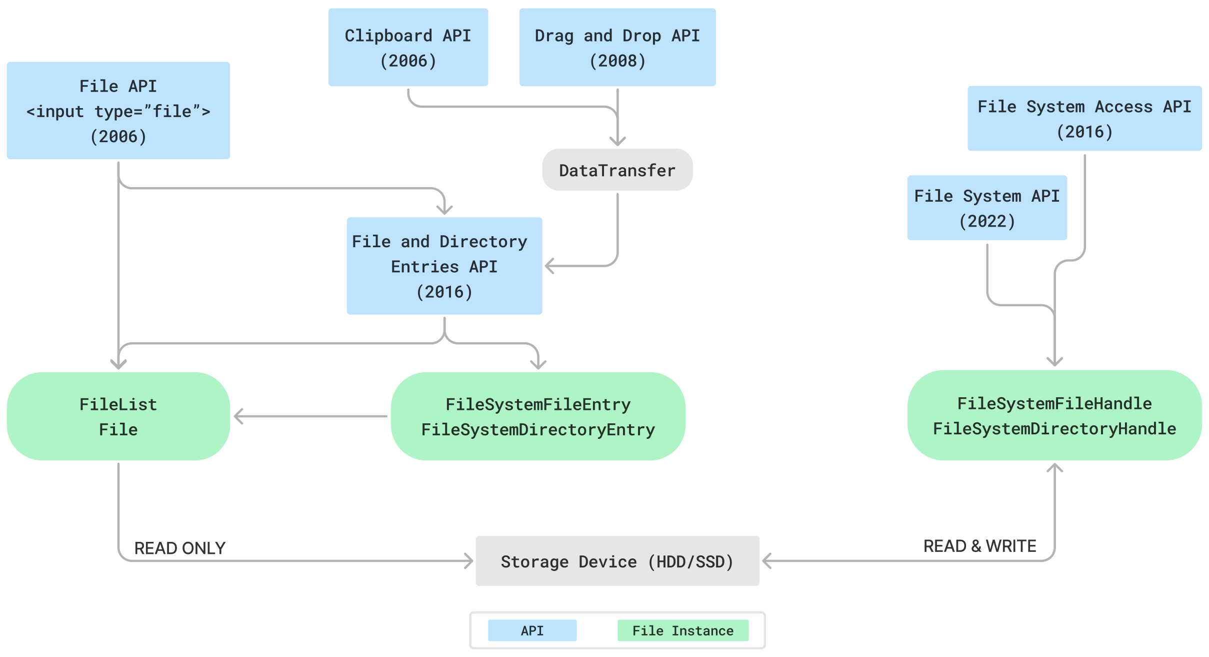

Clipboard API (2006)
Drag and Drop API (2008)
File API <input type="file"> (2006)
File System Access API (2016)
DataTransfer
File System API (2022)
File and Directory Entries API (2016)
FileList File
FileSystemFileEntry FileSystemDirectoryEntry
FileSystemFileHandle FileSystemDirectoryHandle
READ ONLY
Storage Device (HDD/SSD)
READ & WRITE
API
File Instance

Figure 3.2: The different File I/O APIs available in modern web browsers. [Diagram created by the author of this thesis.]

However, it is limited to reading one or multiple files from the user's storage and does not provide a way to write files or access directories by itself. A simple example of how to use the File API is shown in Listing 3.4.

3.2.4.2 File and Directory Entries API

The File and Directory Entries API is a continuation of the discontinued File API: Directories and System proposal. It was initially proposed by Bell [2016a] and first implemented in Chromium-based browsers, before it was added to the other major browsers for compatibility reasons [Bell 2016b]. Most of the API is still considered experimental and is not yet standardised.

The API extends the File API and provides a way to read files and directories recursively using the `webkitdirectory` attribute of the `<input>` element. If an `<input>` element with the `webkitdirectory` attribute is used, the user can select a directory and the browser will provide a `FileList` object containing all files and directories within the selected directory recursively. A demonstration of this feature is shown in Listing 3.5. For large directories with many deeply nested files, using the `webkitdirectory` attribute can lead to performance issues and can even crash the browser.

3.2.4.3 Drag and Drop API

The Drag and Drop API is primarily designed to provide a means to drag and and drop elements within a web page. It is defined in the HTML5 specification [WHATWG 2024d, Chapter 6.11] and allows the definition of drag sources and drop targets. However, the API can also be used to access files and directories by dragging and dropping them into the browser window. This uses some of the functionality of the File and Directory Entries API from Section 3.2.4.2. The Drag and Drop API provides a single FileSystemDirectoryEntry object when a directory is dropped into the browser window. The FileSystemDirectoryEntry represents a directory on the user's storage and provides a means to recursively browse through nested directories. It is supported by all major browsers and does not have the same performance implications as the File and Directory Entries API. A simple example of how to use the Drag and Drop API to read a directory is shown in Listings 3.6 (HTML and CSS) and 3.7 (JavaScript).

```html
<!DOCTYPE html>
<html>
  <body>
    <h1>File API - 1. Reading a File</h1>
    <label for="file-picker">Select a file:</label>
    <input id="file-picker" type="file" accept=".txt,.md" />
    <pre><code id="output"></code></pre>
    <script>
      const picker = document.querySelector("input");
      picker.addEventListener("change", async (e) => {
        const file = event.target.files[0];
        const text = await file.text();
        document.querySelector("#output").innerHTML = text;
      });
    </script>
  </body>
</html>
```

Listing 3.4: File API: A simple example to read a text file on the client-side.

```html
<!DOCTYPE html>
<html>
  <body>
    <h1>File API - 2. Scanning a Directory</h1>
    <label for="file-picker">Select a directory:</label>
    <input id="file-picker" type="file" webkitdirectory />
    <pre><code id="output"></code></pre>
    <script>
      const MAX_FILES = 1000;

      const picker = document.querySelector("input");
      const output = document.querySelector("#output");

      picker.addEventListener("change", async (e) => {
        output.innerHTML = "";
        printFile(event.target.files, 0);
      });

      function printFile(files, index) {
        if (index > MAX_FILES) {
          return;
        }
        const file = files[index];
        const text = file.webkitRelativePath + "\n";
        output.innerHTML += text;

        if (index < files.length - 1) {
          // setTimeout is used to prevent the browser from freezing
          // while processing a large number of files
          setTimeout(() => printFile(files, index + 1), 2);
        }
      }
    </script>
  </body>
</html>
```

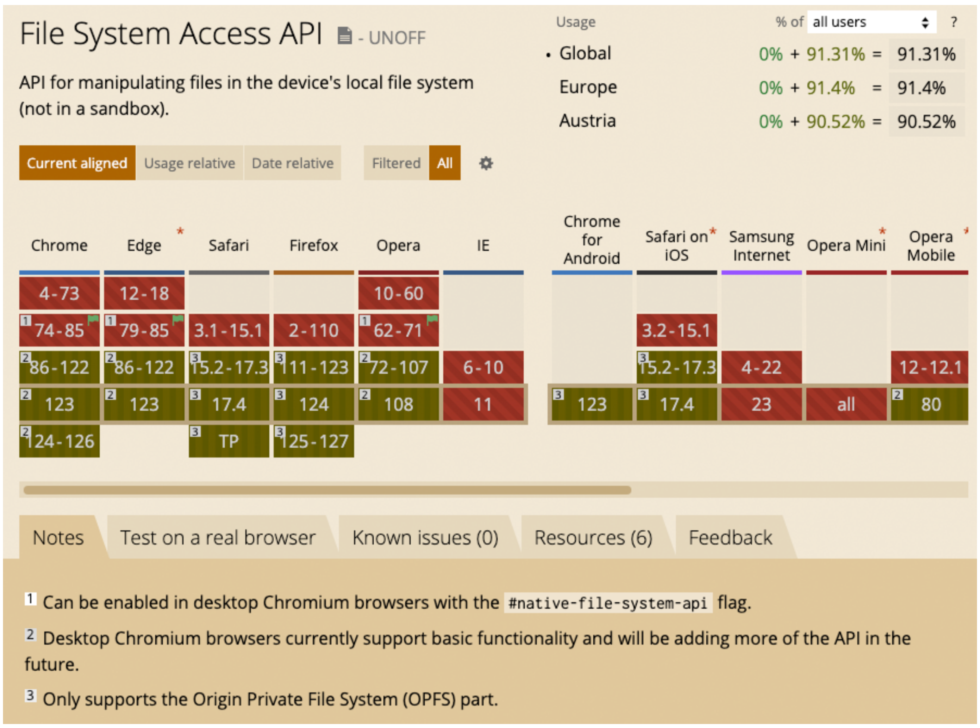
Listing 3.5: File and Directory Entries API: A simple example to read a directory on the client-side using the `webkitdirectory` attribute of the `<input>` element.

```
 1 <!DOCTYPE html>
 2 <html>
 3   <body>
 4     <h1>Drag and Drop API - Scanning a Directory</h1>
 5     <div id="drop-area">
 6       <span>Drop files here</span>
 7     </div>
 8     <pre id="output"></pre>
 9     <style>
10       #output {
11         display: block;
12         width: 100%;
13         background-color: #efefef;
14         min-height: 30px;
15       }
16 
17       #drop-area {
18         width: 400px;
19         height: 100px;
20         border: 2px solid #ccc;
21         display: flex;
22         justify-content: center;
23         align-items: center;
24         font-size: 24px;
25         color: #ccc;
26       }
27 
28       #drop-area span {
29         color: #ccc;
30       }
31 
32       #drop-area.dragging {
33         border: 2px dashed rgb(80, 80, 80);
34         color: #333;
35         span {
36           display: block;
37           color: black;
38         }
39       }
40     </style>
41     <script src="./drag-drop-api-read-directory.js"></script>
42   </body>
43 </html>
```

Listing 3.6: Drag and Drop API: The HTML and CSS of a simple example to read a directory on the client-side. The JavaScript code from Listing 3.7 is imported in line 41.

```javascript
const MAX_DEPTH = 3;

const dropArea = document.querySelector("#drop-area");
const output = document.querySelector("#output");

dropArea.addEventListener("dragover", (event) => {
  event.preventDefault();
  dropArea.classList.add("dragging");
});

dropArea.addEventListener("dragleave", (event) => {
  event.preventDefault();
  dropArea.classList.remove("dragging");
});

dropArea.addEventListener("drop", async (event) => {
  event.preventDefault();
  dropArea.classList.remove("dragging");
  output.innerHTML = "";
  const item = event.dataTransfer.items[0].webkitGetAsEntry();

  scanFiles(item, document.querySelector("#output"));
});

// Taken from https://developer.mozilla.org/en-US/docs/Web/API/DataTransferItem/
    webkitGetAsEntry#javascript
function scanFiles(item, container, depth = 0) {
  if (depth > MAX_DEPTH) {
    return;
  }
  let elem = document.createElement("li");
  elem.textContent = item.name;
  container.appendChild(elem);

  if (item.isDirectory) {
    elem.textContent += "/";
    let directoryReader = item.createReader();
    let directoryContainer = document.createElement("ul");
    container.appendChild(directoryContainer);
    directoryReader.readEntries((entries) => {
      entries.forEach((entry) => {
        setTimeout(() => {
          scanFiles(entry, directoryContainer, depth + 1);
        }, 1);
      });
    });
  }
}
```

Listing 3.7: Drag and Drop API: The JavaScript code of the simple example to read a directory on the client side. It is used in conjunction with the HTML and CSS code from Listing 3.6.



File System Access API - UNOFF

API for manipulating files in the device's local file system (not in a sandbox).

Usage	% of all users
Global	0% + 91.31% = 91.31%
Europe	0% + 91.4% = 91.4%
Austria	0% + 90.52% = 90.52%

Current aligned | Usage relative | Date relative | Filtered | All

Chrome	Edge *	Safari	Firefox	Opera	IE	Chrome for Android	Safari on iOS *	Samsung Internet	Opera Mini *	Opera Mobile *
4-73	12-18			10-60						
[1] 74-85	[1] 79-85	3.1-15.1	2-110	[1] 62-71			3.2-15.1			
[2] 86-122	[2] 86-122	[3] 15.2-17.3	[3] 111-123	[2] 72-107	6-10		[3] 15.2-17.3	4-22		12-12.1
[2] 123	[2] 123	[3] 17.4	[3] 124	[2] 108	11	[3] 123	[3] 17.4	23	all	[2] 80
[2] 124-126		[3] TP	[3] 125-127							

Notes | Test on a real browser | Known issues (0) | Resources (6) | Feedback

[1] Can be enabled in desktop Chromium browsers with the `#native-file-system-api` flag.

[2] Desktop Chromium browsers currently support basic functionality and will be adding more of the API in the future.

[3] Only supports the Origin Private File System (OPFS) part.

Figure 3.3: Browser support for the File System Access API. [Screenshot taken by the author of this thesis from caniuse.com [2024].]

3.2.4.4 Clipboard API

The Clipboard API was initially drafted in 2006 and is designed to provide a way to interact with the user's clipboard [McCathieNevile and Schepers 2006]. The API provides a way to read and write data to the clipboard using the functions `navigator.clipboard.read()` and `navigator.clipboard.write()`, respectively. If a user copies a directory or file from the file system and pastes it into the browser, the API can be used to access these parts of the filesystem through the `paste` event and its `clipboardData` property, which is a `DataTransfer` object defined in the Drag and Drop API. As in the Drag and Drop API, a `webkitGetAsEntry()` function can then be used to access the `FileSystemEntry` object to read the file or directory. A simple example of how to use the Clipboard API is shown in Listing 3.8.

3.2.4.5 File System API

The File System API is documented at WHATWG [2024c]. It defines a modern interface to interact with a file system from within the sandbox of a browser window. The API is designed to be secure and mostly asynchronous. While the specification defines an interface, the implementations and file systems provided to the sandbox are up to the browser vendor. The two most common implementations nowadays are the Origin Private File System and the File System Access API.

3.2.4.6 Origin Private File System (OPFS)

The Origin Private File System implements a subset of the File System API to provide read and write access to a sandboxed file system for a specific origin on the user's storage device. The file system is persistent, although application code cannot ensure that data might not be deleted by the browser at any time. The API does not provide ways to access the file system directly, nor does it allow the user access to the sandboxed file system. It is supported by many modern browsers, including Chromium, Firefox, and Safari, as shown in footnote 3 of Figure 3.3. A simple example of how to use the Origin Private File System API is shown in Listing 3.9.

```
<!DOCTYPE html>
<html>
  <body>
    <h1>Clipboard API - Paste a folder into the page</h1>
    <pre id="output"></pre>
    <style>
      #output {
        display: block;
        width: 100%;
        background-color: #efefef;
        min-height: 30px;
      }
    </style>
    <script>
      window.addEventListener("paste", (e) => {
        const output = document.querySelector("#output");
        output.innerHTML = "";
        const item = e.clipboardData.items[0].webkitGetAsEntry();

        scanFiles(item, document.querySelector("#output"));
      });

      // Taken from https://developer.mozilla.org/en-US/docs/Web/API/
          DataTransferItem/webkitGetAsEntry
      function scanFiles(item, container) {
        let elem = document.createElement("li");
        elem.textContent = item.name;
        container.appendChild(elem);

        if (item.isDirectory) {
          elem.textContent += "/";
          let directoryReader = item.createReader();
          let directoryContainer = document.createElement("ul");
          container.appendChild(directoryContainer);
          directoryReader.readEntries((entries) => {
            entries.forEach((entry) => {
              scanFiles(entry, directoryContainer);
            });
          });
        }
      }
    </script>
  </body>
</html>
```

Listing 3.8: Clipboard API: The JavaScript code of a simple example to read a directory on the client-side.

```
const opfsHandle = await navigator.storage.getDirectory();
for await (let [name, handle] of opfsHandle) {
  if (name === "dir_to_delete") {
    handle.remove({ recursive: true });
  }
}
const newDir = opfsHandle.getDirectoryHandle("new_dir", { create: true });
const fileHandle = await newDir.getFileHandle("file.txt", { create: true });
const writable = await fileHandle.createWritable();
await writable.write("Hello World");
await writable.close();
```

Listing 3.9: Origin Private File System API: JavaScript code to manipulate files and directories within a sandboxed file system. First, a directory handle is retrieved using `navigator.storage.getDirectory()`, then the directory entries are iterated through to find and remove a specific directory. Afterwards, a new directory and a file within it is created, before content is written to the file.

3.2.4.7 File System Access API

The File System Access API is designed to provide a way to read from and write to a user's local file system directly, using the following asynchronous functions:

- `showOpenFilePicker()`: Opens a file picker dialogue to select a file.
- `showSaveFilePicker()`: Opens a file picker dialogue to save a file.
- `showDirectoryPicker()`: Opens a directory picker dialogue to select a directory.

The API is designed to be secure and user-friendly. As such, there are conditions that need to be met before these functions can be called:

- The functions must be called in response to a user action like a mouse event.
- The page must be served from a secure context.
- The functions must not be called from cross-origin sub frames (iframes).

The FileSystemHandle object returned by the API is defined in the File System API, and as such can be used interchangeably with the Origin Private File System. The specification is still in a draft state and can be found at WICG [2024]. Only some Chromium-based browsers support this API, as shown in footnote 2 of Figure 3.3. An example of how to use the File System Access API is shown in Listing 3.10.

3.2.5 WebAssembly JavaScript Interface

The WebAssembly JavaScript Interface is an API designed to provide a way to execute and interact with WebAssembly modules from within JavaScript. The interface is standardised by WAWG [2024] and describes how to instantiate and execute WebAssembly modules. The API also defines how JavaScript can interact with memory provided to a WebAssembly module. A simple example of how to execute a WebAssembly module of Listing 2.6 from JavaScript is shown in Listing 3.11.

```
<!DOCTYPE html>
<html>
  <body>
    <h1>File System Access API</h1>
    <button id="run-demo">Run Demo</button>
    <pre><code id="output"></code></pre>

    <script>

      if (!"showDirectoryPicker" in window) {
        console.error("This browser does not support the File System Access API.");
        return;
      }

      const output = document.querySelector("#run-demo");
      const button = document.querySelector("#open-file");

      button.addEventListener("click", async () => {
        output.textContent = "";
        const dirHandle = await window.showDirectoryPicker();

        const newDir = await dirHandle.getDirectoryHandle("new_dir", {
          create: true,
        });
        const fileHandle = await newDir.getFileHandle("file.txt", {
          create: true,
        });
        const writable = await fileHandle.createWritable();
        await writable.write("Hello World");
        await writable.close();
        output.textContent = "File written to new_dir/file.txt";
      });
    </script>
  </body>
</html>
```

Listing 3.10: File System Access API: A directory handle is requested after a user action. Then, a new directory called `new_dir` and a file within it named `file.txt` is created with the content "Hello World". This code can only be executed in a browser which supports the File System Access API (Chromium-based).

```
const wasmCode = fetch("wat-example.wasm").then((response) =>
  response.arrayBuffer()
);
const module = new WebAssembly.Module(wasmCode);

const instance = new WebAssembly.Instance(module);

const result = instance.exports.add(1, 2);

console.log(result); // 3
```

Listing 3.11: WebAssembly JavaScript Interface: Instantiating the WebAssembly module of Listing 2.6 and calling the `add` function from it.

3.3 The Browser Event Loop

JavaScript is single-threaded, only one piece of code can be executed at a time. Within a browser, this single thread executes all JavaScript code, handles user input, and updates the user interface. Especially for longer-running tasks, this can lead to a poor user experience, since the main thread may become blocked and unable to handle user input or update the user interface in time. To prevent this, the browser uses the concept of an event loop to handle tasks in a non-blocking way, as specified in the HTML specification [WHATWG 2024d, Chapter 8.1.7]. This concept splits work into small tasks and schedules them to be executed in the future. The event loop is responsible for executing these tasks in a prioritised order so as to reduce the risk of blocking the main thread.

3.3.1 Event Loop Components

The event loop consists of several components that work together to handle tasks in a non-blocking way. The three main components are task queues, the microtask queue, and the call stack.

3.3.1.1 Task Queues

Task queues store tasks that are scheduled to be executed in the future, for example callbacks or events. The HTML specification defines several task sources that are used to group tasks as they are added to their task queue. Even though the name suggests task queues are a First-In, First-Out structure, the specification [WHATWG 2024d, Chapter 8.1.7.1] states that “Task queues are sets, not queues, because the event loop processing model grabs the first runnable task from the chosen queue, instead of dequeuing the first task”. In this context, a task is considered runnable, if it can be executed immediately and has no unmet dependencies.

Browser vendors can implement task sources as they see fit. However, the following task sources are commonly used:

- *Events*: Tasks related to user input like mouse movements, clocks, keyboard inputs, or touch events.
- *Callbacks*: Tasks scheduled for execution by functions like `setTimeout()` or `requestIdleTask()`.
- *Resources*: Tasks related to loading resources like images, scripts, or stylesheets.
- *Parsing*: Tasks related to parsing HTML and CSS files after they have been downloaded.

3.3.1.2 Microtask Queue

The microtask queue is a special queue that has a higher priority than the main task queue. It is used to handle tasks that are executed immediately after the current task has finished. This is useful for tasks that need to be executed as soon as possible, like updating the user interface after a change in the DOM. The microtask queue is also used to schedule `then()` callbacks of Promises.

3.3.1.3 Call Stack

The call stack in JavaScript is a Last-In, First-Out structure for managing function calls. It maintains a stack of frames, each representing an individual function call. Initially, the call stack is empty. As a task is picked from the task queue for execution, the call stack is only responsible for executing this task and is blocked until the task is completed. During execution, each function call adds a new frame to the top of the stack. Once a function completes and returns, its corresponding frame is removed, and the information about the next frame on the stack is used to continue execution. This mechanism ensures that the JavaScript engine can track and manage the sequence and order of function executions accurately. Only when the call stack is empty, can the event loop continue to execute the next task from the task queues.

```javascript
const taskSourcePriority = ["event", "callback", "parsing", "resource"];
const macroTaskQueues = {
  event: [],
  callback: [],
  parsing: [],
  resource: [],
};
const microTaskQueue = [];

function getNextTask() {
  for (const source of taskSourcePriority) {
    const task = macroTaskQueues[source].shift();
    if (task) {
      return task;
    }
  }
  return undefined;
}

function executeAllMicroTasks() {
  const tasks = microTaskQueue;
  microTaskQueue = [];
  tasks.forEach((task) => task.execute());
}

function eventLoop() {
  while (true) {
    const nextTask = getNextTask();
    if (nextTask) {
      nextTask.execute();
    }
    executeAllMicroTasks();

    if (renderingRequired()) {
      render();
    }
  }
}
```

Listing 3.12: The event loop algorithm in a very simplified form, ignoring many edge cases and optimizations for the sake of brevity and clarity.

3.3.2 The Event Loop Processing Model

The event loop processing model describes how an event loop has to process the task queues and is defined in the HTML specification [WHATWG 2024d, Chapter 8.1.7.3]. An event loop has to continuously execute these steps to ensure that tasks are executed in a non-blocking way. The simplified example in Listing 3.12 represents the processing model of the event loop as a JavaScript function. The main takeaway is that longer running tasks should be split into smaller chunks or offloaded to a web worker, so as to prevent blocking the main thread and avoid an unresponsive user interface.

3.3.3 Task Precedence

The event loop has to handle tasks in a specific order to ensure that the most important tasks are executed first. As a developer, it is important to understand the precedence of tasks to ensure that the application behaves as expected. A simple example of how the event loop handles tasks with different priorities is shown in Listing 3.13. It consists of a series of log statements that are executed in a specific order.

```
function log(s) {
  console.log(s);
}

log("A");
setTimeout(() => log("B"), 0);
new Promise((resolve) => {
  log("C");
  resolve();
}).then(() => {
  log("D");
});
log("E");

// Output: ACEDB
```

Listing 3.13: Simple example of how the event loop handles tasks with different priorities. Initially, the log statement "A" is executed synchronously. Subsequently, a `setTimeout` function is called with a delay of 0 milliseconds, which schedules the log statement "B" to be executed asynchronously as a callback. Following this, a promise is created with a `resolve` function that logs "C" synchronously. The promise is then resolved, triggering the execution of the callback that logs "D". Finally, the log statement "E" is executed synchronously.

The log output of the code is "ACEDB", since synchronous log statements, including promise callbacks, are executed first, followed by the microtask `then()` callback, and finally the asynchronous `setTimeout()` callback.

Chapter 4

Git Version Control System

A version control system is a software tool that helps developers manage changes to source code over time. It allows developers to track changes, collaborate with others, and revert to previous versions of the source code. Git is a very popular distributed version control system [Git 2024]. Its command-line tool `git` is often used by developers when interacting with a Git repository. This chapter provides an overview of the main concepts of Git and introduces some details of its inner workings, based on the book by Chacon and Straub [2024]. This chapter also provides an overview of other popular libraries and tools for programmatically interacting with Git repositories, highlighting their capabilities and limitations.

4.1 Introduction to Git

Git is a distributed version control system originally developed in 2005 for the linux kernel development community by Linus Torvalds [Chacon and Straub 2024, Section 1.2]. Unlike centralised version control systems, such as Subversion or CVS, Git is a distributed version control system, giving each developer a full copy of the repository and its history on their local machine [Spinellis 2012]. Only when changes are pushed to a remote repository are they shared with other developers. This allows developers to work offline and independent of a central server, until they are ready to share their changes with others. Initially designed for the Linux kernel, Git has since become the most popular version control system, according to Stack Overflow's 2022 Developer Survey [SO 2022], which asked the question "What are the primary version control systems you use?", as can be seen in Figure 4.1. The question was not asked in 2023 or 2024.

4.2 Git Fundamentals

When a repository is managed by Git, its history is stored as commits. Each commit stores at least one reference to its parent commit, as well as metadata such as the author, a unique commit id, and a commit message. An example of a Git commit graph, showing the history of a repository, is shown in Figure 4.2. The following sections provide an overview of the main entities of Git repositories and their relationships.

4.2.1 Setting up a Repository

A repository is a collection of source code that is managed by Git. It contains the history of the project, including all the changes that have been made to it. To create a new repository, the command `git init` is used. This command creates a repository in the current directory and initialises the necessary files and directories to manage the source code with Git. All data related to the repository history is stored in a hidden directory called `.git/` within the root directory of the repository.

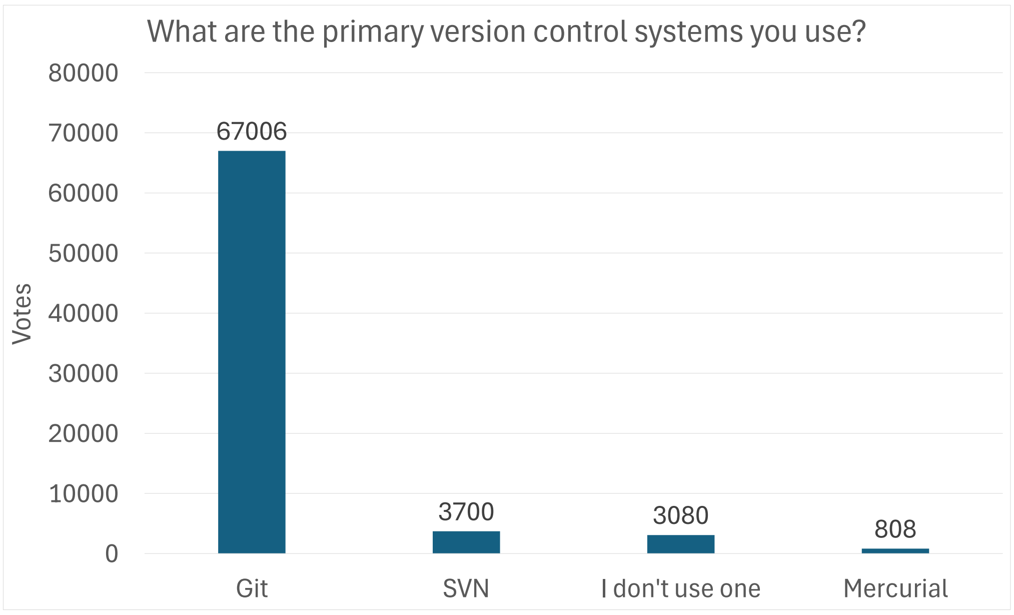


Figure 4.1: Survey results for the question “What are the primary version control systems you use?” from the Stack Overflow Developer Survey 2022. The question was not asked in 2023 or 2024. [Diagram created by the author of this thesis, based on Data of the Public 2022 Stack Overflow Developer Survey Results, licensed under the Open Database License (ODbL) [SO 2022].]

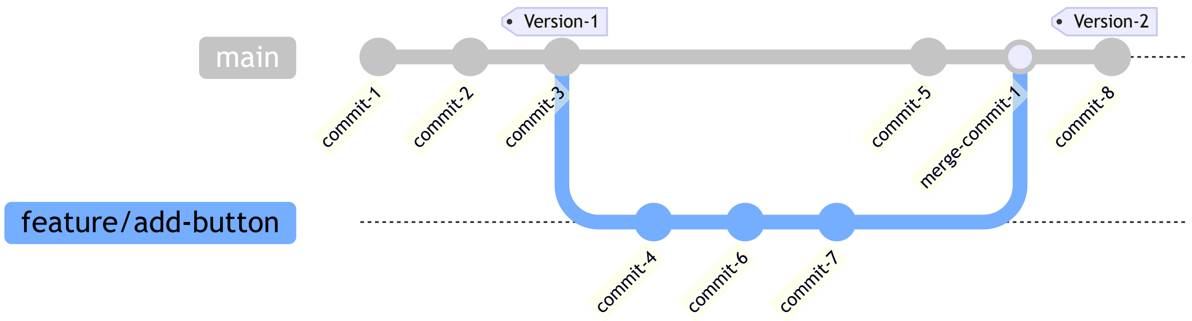


Figure 4.2: In this example of a Git commit graph, the repository starts with three sequential commits on the main branch, until the third commit, `commit-3`, is tagged as `Version-1`. A feature branch called `feature/add-button` is created and worked on separately, resulting in three additional commits `commit-4`, `commit-6`, and `commit-7`. Within the same time frame, `commit-5` is added to the main branch. Subsequently, the feature branch is merged into the `main` branch, creating a merge commit called `merge-commit-1`. Finally, a new commit, `commit-8`, is made on the main branch, and is tagged as `Version-2`. [Image created by the author of this thesis.]

4.2.2 Committing Changes

A commit is a snapshot of the repository at a specific point in time. It contains the changes that have been made to the repository since the last commit. After making some changes in the working directory, say by editing the file `README.md`, these changes would be added to the staging area with the command:

```
git add README.md
```

The changes are committed to the repository with the command:

```
git commit -m "Initial commit"
```

The commit message “Initial commit” is a short description of the changes made in the commit. The commit also records the author of the changes, a timestamp, and a unique commit id. The author of a commit is usually determined by the user’s name and email address configured in the Git configuration settings, either during the installation of Git or by running the commands:

```
git config --global user.name "Max Mustermann"
git config --global user.email "maxmuster@gmail.com"
```

To check if the commit was successful, the command `git log` can be used to show the commit history of the repository.

4.2.3 Branching

A branch is a separate channel of development in a repository. It enables developers to work on different tasks in parallel without interfering with each other. Initially, a repository has a default branch called `main`. A new branch is created with a command similar to:

```
git branch feature/add-button
```

This creates a new branch called `feature/add-button` based on the main branch’s current commit. To switch to the new branch, the command:

```
git checkout feature/add-button
```

is used. The modifications made in the new branch are independent of the main branch until they are merged back into the main branch.

To merge the changes of the new branch back into the main branch, the command:

```
git merge feature/add-button
```

is used. This creates a new commit, which combines the changes of the new branch with the main branch. Afterwards, the new branch can be deleted with the command:

```
git branch -d feature/add-button
```

4.2.4 Merge Conflicts

A merge conflict occurs when two branches with different changes can only be merged automatically with the risk of losing data. This can happen when two branches diverge significantly, and the changes made on one branch conflict with those made on the other. To resolve a merge conflict, the developer must manually merge the changes made on the two branches. This is typically done in an editor like VS Code, or on the command line with `git merge`.

4.2.5 Tagging

A tag is a reference to a specific commit in the repository. It is mainly used to mark releases or other points of interest in the history of the project with a custom user-defined label. A tag is created with a command like:

```
git tag Version-1
```

This creates a tag called `Version-1`, which points to the current commit. To list all tags in the repository, the command:

```
git tag
```

can be used.

4.2.6 Git Revision

A Git revision is a reference to a specific commit. This can be a commit id, a branch name, a tag name, or a relative reference to a commit, such as `HEAD`. The first few characters of the commit id are usually sufficient to uniquely identify the commit. For example, the commit id `commit-3` in Figure 4.2 can be referenced as `commit-3` or `Version-1`.

4.2.7 Using a Remote

A remote is a reference to Git server accessible over the internet. Common remote repository hosting services include GitHub [GitHub 2024b], GitLab [GitLab 2024], and Bitbucket [Bitbucket 2024]. The remote name is used as a handle to synchronise the local repository with the remote repository.

A remote repository is defined by the command:

```
git remote add origin <remote-url>
```

This creates a remote called `origin`, which points to the remote repository at the given URL. To push changes to the remote repository, the command:

```
git push origin main
```

is used. This pushes the changes made in the main branch to the remote repository. To pull changes from the remote repository, the command:

```
git pull origin main
```

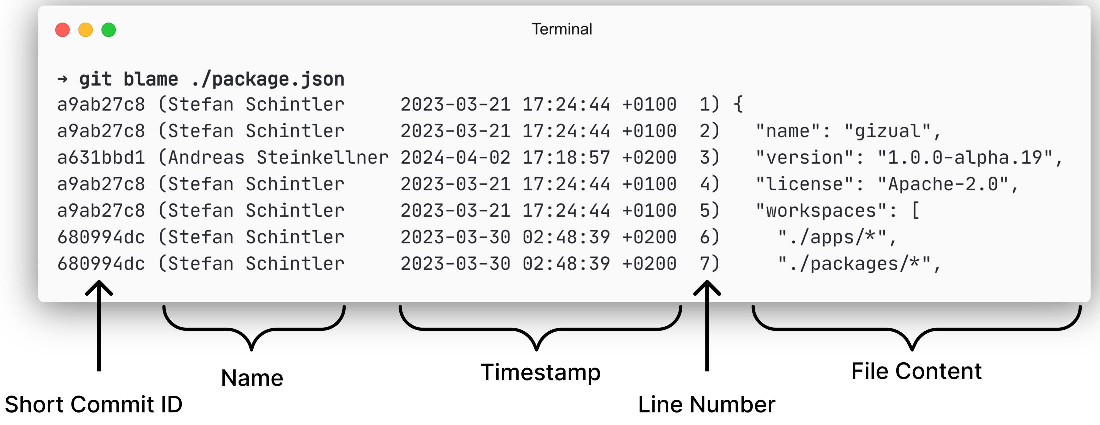
is used. This fetches the changes from the remote repository and merges them into the main branch. A remote URL can either be an HTTP or an SSH URL, depending on the protocol used.

4.2.8 Blame

A blame shows the author of each line in a file at a given point in time. It can be used to track down which author is responsible for a specific change within the source code, and as such is a valuable tool for debugging and code review. An example of the output of the:

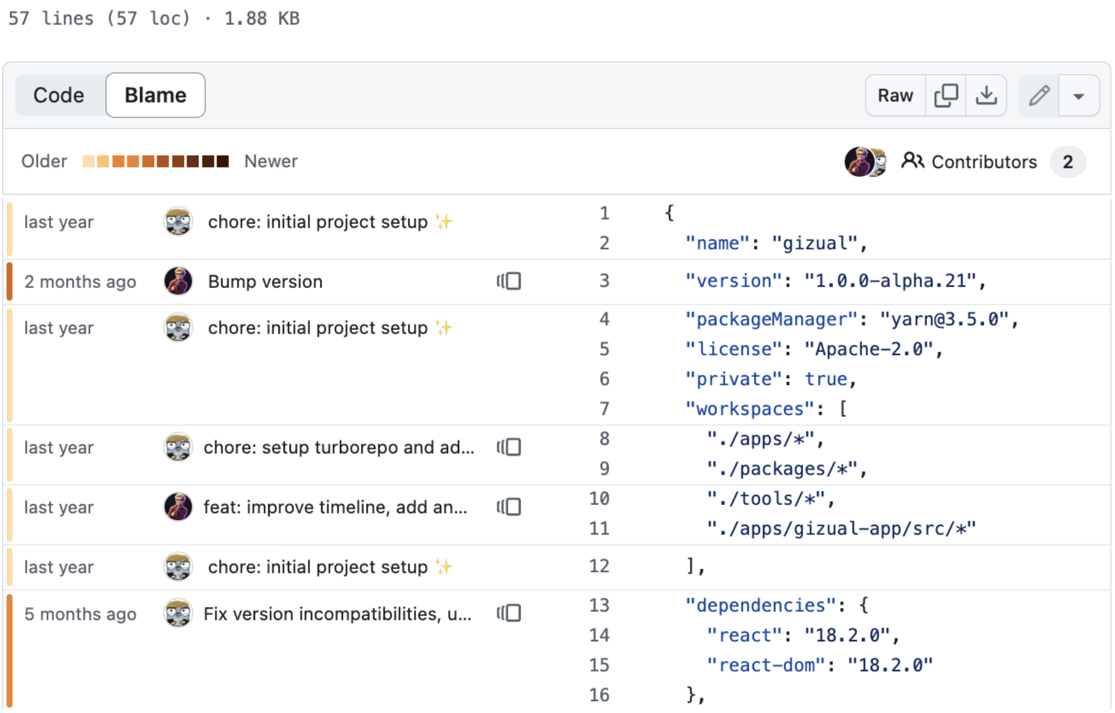
```
git blame
```

command in a terminal is shown in Figure 4.3. The Blame view on hosting platforms like GitHub is often more elaborate, as shown in Figure 4.4.



```
→ git blame ./package.json
a9ab27c8 (Stefan Schintler     2023-03-21 17:24:44 +0100  1) {
a9ab27c8 (Stefan Schintler     2023-03-21 17:24:44 +0100  2)   "name": "gizual",
a631bbd1 (Andreas Steinkellner 2024-04-02 17:18:57 +0200  3)   "version": "1.0.0-alpha.19",
a9ab27c8 (Stefan Schintler     2023-03-21 17:24:44 +0100  4)   "license": "Apache-2.0",
a9ab27c8 (Stefan Schintler     2023-03-21 17:24:44 +0100  5)   "workspaces": [
680994dc (Stefan Schintler     2023-03-30 02:48:39 +0200  6)     "./apps/*",
680994dc (Stefan Schintler     2023-03-30 02:48:39 +0200  7)     "./packages/*",
```

Figure 4.3: The output of the `git blame` command in a terminal window for the Gizual repository, showing the commit id, author, and last change timestamp of each line of the file `package.json`. [Image created by the author of this thesis.]



```
last year      chore: initial project setup ✨           {
                                                            "name": "gizual",
2 months ago   Bump version                                 "version": "1.0.0-alpha.21",
last year      chore: initial project setup ✨              "packageManager": "yarn@3.5.0",
                                                            "license": "Apache-2.0",
                                                            "private": true,
                                                            "workspaces": [
last year      chore: setup turborepo and ad...               "./apps/*",
                                                              "./packages/*",
last year      feat: improve timeline, add an...              "./tools/*",
                                                              "./apps/gizual-app/src/*"
last year      chore: initial project setup ✨              ],
5 months ago   Fix version incompatibilities, u...          "dependencies": {
                                                              "react": "18.2.0",
                                                              "react-dom": "18.2.0"
                                                            },
```

Figure 4.4: GitHub's Blame view is more elaborate, using colour-coding and showing the commit message for each line of code. [Screenshot created by the author of this thesis from GitHub [2024b].]

4.3 Git Internals

The Git version control system stores the history of a repository as an acyclic graph of commits. This is accomplished by storing a set of objects representing the different entities in the repository, such as commits, trees, and blobs. All of these objects are stored in a hidden directory called `.git/`, within the root directory of the repository.

4.3.1 Object Filenames

Each object in a Git repository is identified by a unique SHA-1 hash. This hash is calculated using the content of the object as input to the SHA-1 algorithm. The resulting hash is a 40-character hexadecimal string that uniquely identifies the object. For example, the SHA-1 hash of the blob object for the `README.md` created in Section 4.2.2 would be `29658341f39210201ff7f72a4be83937cf2288c5` and would therefore be stored at `.git/objects/29/658341f39210201ff7f72a4be83937cf2288c5`.

4.3.2 Object Types

A Git object is a compressed text file that represents a specific entity in the repository. The object has different fields depending on its type:

- commit: size, reference to the main tree root, author, committer, commit message, and timestamp.
- tree: 1 to n references to files (blobs) or nested trees.
- blob: size, content of the file.

To distinguish between the different types of objects, the object is prefixed with a header containing the object's type and size. The size of the object is the size of the uncompressed object data excluding its header. Following the instructions to create an initial commit, as described in Section 4.2.2, the commit object would look like this:

```
commit 187
tree 112e80d883d32420393873531b61a7ae4f43a3d2
author Max Mustermann <maxmuster@gmail.com> 1720343396 +0200
committer Max Mustermann <maxmuster@gmail.com> 1720343396 +0200

Initial commit
```

4.3.3 Graph Structure

Git stores the history of a repository as a graph of commits. Each commit contains at least one reference to its parent commit, representing the previous state of the repository. This allows Git to reconstruct the history of the repository by traversing the graph of commits.

Each commit contains metadata about the author, a timestamp, and a reference to the root tree object, which represents the state of the repository at the time of the commit. The root tree object contains references to blobs and other sub-tree objects, representing the files and nested directories within the repository at the point in time of the specific commit it belongs to. A blob represents a file in the repository and contains the content of the file.

A simplified graph structure of a repository is shown in Figure 4.5. In this example, the repository contains three commits, each with a reference to its parent commit. The root tree object of each commit contains references to blobs that represent the files in the repository. In the first commit, a single file `README.md` was added to the repository. In the second commit, a new file `test.txt` was added. The tree of the second commit still contains the reference to the original `README.md` file, since it has not changed.

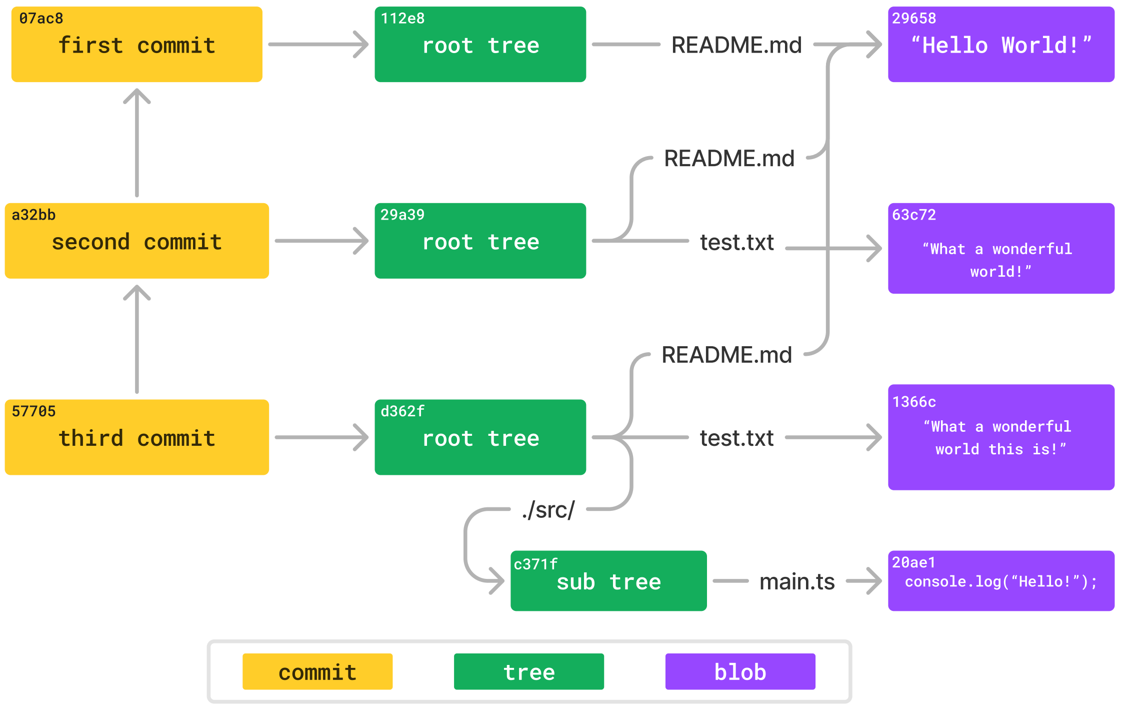


Figure 4.5: Three commits with their trees and blobs in a Git repository. [Adapted from Figure 9 of Chacon and Straub [2024, page 64] and Figure 175 of Chacon and Straub [2024, page 423]. Used under terms of a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License.]

In the third commit, the content of the `test.txt` was updated, while the `README.md` did not change. As such, the tree of the third commit contains references to both the original `README.md` file and the updated `test.txt` file. The third commit also contains a reference to a new file `src/main.ts`. This is represented within the Git graph as a sub-tree for the nested directory `src/` as well as a blob for the file `main.ts`.

4.3.4 Packfiles

As the history of a repository grows, the number of objects stored in the repository also increases, since each time a file is altered, the whole file is stored as a new blob object. To reduce the size of the repository and improve performance, Git uses packfiles to store the objects in a compressed format. A packfile is a single file that contains multiple objects that have been compressed and delta-encoded. This reduces the size of the repository and speeds up operations on the Git history. Packfiles are created by running the `git gc` command, or automatically while using the Git command-line tool when the repository has collected enough loose objects. This process compresses the objects in the repository and creates a packfile to store them in the repository's `.git/objects/pack/` directory, while the original loose objects are deleted.

4.4 Git Libraries and Tools

To interact with a repository managed by Git, several libraries and tools have been developed. These provide an API to interact with the repository programmatically, either via the command line or via a graphical user interface. The following sections provide an overview of popular Git libraries and tools. They highlight their abilities and limitations in the context of the extraction of metadata from the blame information of Git repositories within a web browser environment, which is the focus for this thesis.

4.4.1 Git Command Line Interface

The Git command line interface (CLI) is the default way to interact with a Git repository [Git 2024]. It was developed in conjunction with the original Git version control system for the Linux kernel and remains the default implementation to interact with Git repositories. Programmatic usage of the Git CLI can be achieved using wrapper libraries like Simple Git [King 2024b], which provides a JavaScript API to interact with Git repositories via the Git CLI. However, this approach is not suitable for the intended use in Gizual, as it requires Node.js and the installation of Git on the system. While the Git CLI is fairly self-contained, it requires a suite of dependencies to build and execute it, such as `zlib`, `Perl`, and a POSIX compliant shell [GitHub 2024a, lines 110–167]. This makes it unfeasible to run the Git CLI in a web browser environment.

4.4.2 Libgit2 Library

The libgit2 library is an implementation of the most common Git operations in a C library [libgit2 2024]. Since it is designed to be portable, many bindings to other programming languages have been created, such as for Ruby, .NET, and Python. With some modifications to the build process, Salomonsen [2024] managed to compile the libgit2 library to WebAssembly, which allows running the library in a web browser environment. This makes it a suitable candidate for the intended use in Gizual, even though the file system used by Salomonsen is backed by memory only. The source code of libgit2 is licensed under the GPL2 license. Some example C code using the libgit2 library to extract blame information about the last authors of each line in a file is shown in Listing 4.1.

4.4.3 Isomorphic-git Library

The isomorphic-git library is an implementation of the most common functions of Git written entirely in JavaScript [Hilton 2024a]. It is designed to behave similarly to the Git CLI by traversing the data structures of a repository within the `.git/` directory. The library can be used in both Node.js and browser environments. Since no native file system exists in browser environments, the library suggests using lightning-fs [Hilton 2024b] to provide a virtual filesystem for the library to interact with. However, since this virtual file system is held entirely in memory, it is not suitable for large repositories. Furthermore, isomorphic-git also does not support generating blame, which makes it unsuitable for the intended use in Gizual.

4.4.4 Gitoxide Library

Gitoxide is a open-source library written in Rust that aims to provide similar functionality to the Git CLI [Thiel 2024]. It is still in early development and does not yet provide features like `git blame`. It is a promising project that could potentially be used in future by Gizual, since it is written entirely in Rust and as such could simplify the integration within the web browser environment.

```c
#include "git2.h"
#include <stdio.h>

#define MAX_LINES 100

void catch_error(int error_code, int exit_code)
{
  if (error_code != 0)
  {
    fprintf(stderr, "Error: %s\n", git_error_last()->message);
    exit(exit_code);
  }
}

int main(int argc, char **argv)
{
  git_libgit2_init();
  char *file_path = "README.md";
  char *git_repo_path = "/home/stefan/Projects/gizual/gizual";

  git_repository *git_repo = NULL;
  catch_error(git_repository_open(&git_repo, git_repo_path), 1);

  git_blame_options blame_opts;
  git_blame_options_init(&blame_opts, GIT_BLAME_OPTIONS_VERSION);

  git_blame *blame = NULL;
  git_object *obj = NULL;

  catch_error(git_blame_file(&blame, git_repo, file_path, &blame_opts), 2);

  size_t lineno = 1;
  char oid[64] = {0};

  while (lineno < MAX_LINES)
  {
    const git_blame_hunk *hunk = git_blame_get_hunk_byline(blame, lineno);
    if (!hunk)
      break;

    git_oid_tostr(oid, 64, &hunk->final_commit_id);
    printf("line %ld originated from commit %s\n", lineno, oid);
    lineno++;
  }

  git_object_free(obj);
  git_repository_free(git_repo);
  git_libgit2_shutdown();
  return 0;
}
```

Listing 4.1: Simple example code written in C, showing how to use the Libgit2 library to extract blame information for the `README.md` file in a Git repository.

Chapter 5

Gizual Architecture

“ I feel that it’s lovely when, as a user, you’re not aware of the complexity. ”

[Jonathan Ive; British and American designer, former Chief Design Officer at Apple.]

Gizual is a browser-based web application for visualising Git repositories. It is implemented with modern web technologies including HTML, SCSS, React, TypeScript, MobX, WebAssembly, Rust, web workers, and Node.js. All repository data processing is done in the browser, no external servers or databases are required. To bring all of these technologies together, a sophisticated software architecture was evolved.

This chapter was written jointly by Stefan Schintler and Andreas Steinkellner.

5.1 Architectural Requirements

Being a web application, Gizual can harness some unique benefits of the web as a platform. From a development standpoint, web libraries are available in abundance, reducing the complexity of building an application from scratch. For end users, the web generally provides a simple and easy-to-use platform for utilities that they can trust, since browser security policies are generally strict enough to protect from malicious actors. Building applications for the web does, however, come with a unique set of challenges, many of which are irrelevant for desktop applications. During the initial project ideation phase of Gizual, the team agreed upon some non-negotiable architectural requirements that would ensure that the application would match the perceived performance of a native tool.

5.1.1 Non-Blocking

By default, all JavaScript code within a web application is processed on the main thread. In addition to being responsible for layout and reflow operations, custom JavaScript execution is also part of the main thread’s responsibilities. For simple websites, this behaviour is usually not problematic, since JavaScript execution engines have become increasingly performant over the years. Users’ device performance is also steadily increasing, making performance optimisations less relevant for a typical web development workflow. Gizual, however, needs to process large quantities of data rapidly, usually in bursts that occur when users zoom or pan around on the visualisation canvas. Batching file manipulation and calculation operations into the same thread handling user interactions creates a performance choke point, leading to an unresponsive user interface or jittery animations. One of the agreed upon architectural requirements was to implement a solution which would reduce the computational complexity on the main thread by assigning pools of web workers [Surma 2019; WHATWG 2024e] to handle time-consuming tasks such as rendering and git exploration. With less work on the main thread, user interactions and animations would not be impeded by rendering tasks, and the application would always feel responsive.

5.1.2 Asynchronicity

In synchronous contexts, instructions are processed sequentially. In the context of web applications, one JavaScript thread runs through its set of instructions and processes them in order. Unfortunately, this can lead to long and inefficient wait times, drastically reducing the user experience and slowing down the application. Asynchronicity solves this problem by providing developers with functionality to let their code run outside the boundaries of the synchronous execution context. Writing asynchronous functions creates a chain of asynchronicity within the application, since any function which calls an asynchronous function must itself be asynchronous. This is why asynchronous functions are often referred to as being contagious [Haverbeke 2024, Chapter 11]. For Gizual, doing all calculations sequentially in a synchronous manner would not be feasible. Asynchronicity was deemed necessary, despite the additional complexities this concept invariantly introduces.

5.1.3 Parallel Execution

Complex functions within Gizual are generally expected to be executed in parallel. In order to achieve this, the project heavily relies on the use of asynchronous code and web workers. Without parallel execution of simultaneous file operations, the Git exploration performance would be too slow at providing repository analytics, slowing down the performance of the entire application. Additionally, parallel execution unlocks the power of modern multicore processor architecture, with different tasks being scheduled across many threads or even processor cores. However, the process of spawning and coordinating multiple threads (web workers) creates a slight computational overhead. Gizual relies on a defined limit of concurrent web workers, adhering to best practice recommendations in order to achieve maximum performance.

5.1.4 Separation of Concerns

The Gizual architecture consists of many modules of varying complexity, linked together via common bridges and interfaces. This loose coupling of segregated functions adheres to the general software engineering concept of *separation of concerns*. The term was initially used in a blog post by famous computer scientist Edsger W. Dijkstra [Dijkstra 1974], who used it to describe the process of clustering and ordering thoughts. In software engineering, the term is often used in conjunction with programming concepts like *model-driven development* [Kulkarni and Reddy 2003]. It describes the strategic separation of self-contained functions into smaller modules, which can be used across the entire application. This module-based approach invites the use of abstract interfaces, which greatly enhance modularity and simplify code maintenance.

Separation of concerns is a prevalent concept in web development too, where the three distinct technologies HTML, CSS, and JavaScript are used in tandem to create a single application. Modern frameworks, however, enable developers to forego this distinction, if desired. The Gizual architecture required more separation of concerns than a typical web application, since the architectural requirements defined in the previous three subsections already necessitate additional layers of abstraction and workload distribution.

5.2 Architectural Overview

To fulfil the architectural requirements, the project is split into six packages. A single, central management component, called Maestro, is responsible for managing the application, including global application state, user input, and data processing. Maestro controls the explorer and renderer pools of web workers, manages global state for the three UI Controllers and various higher-order UI Components, and populates an SQLite database with indexable metadata such as the names of authors (developers) for efficient querying. Figure 5.1 provides a structural overview of the interconnected packages and their communication pathways.

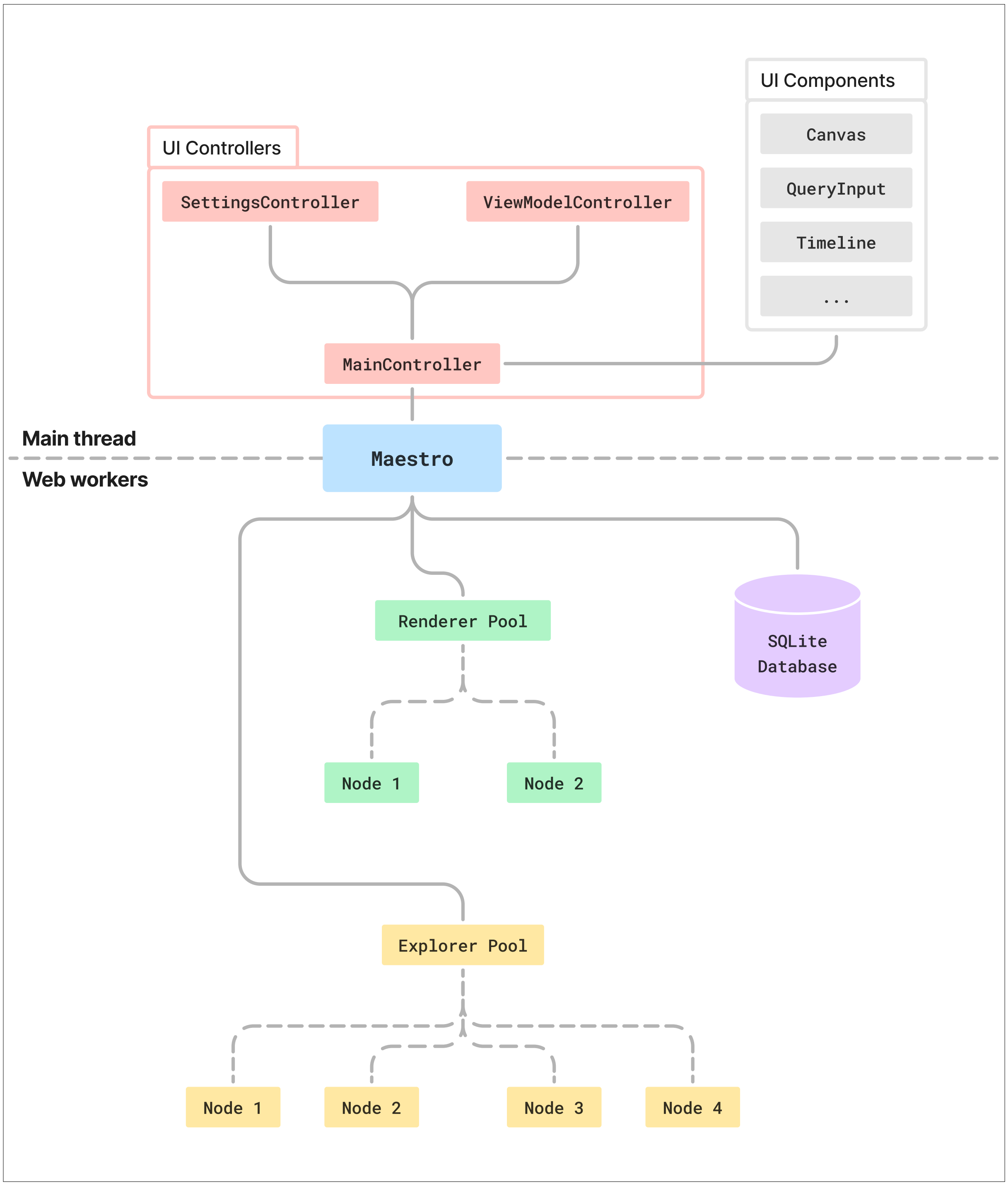


Figure 5.1: The software architecture of Gizual. UI controllers and UI components are instantiated in the main thread. The explorer pool, renderer pool, and SQLite database are executed asynchronously in web workers inside the browser. The Maestro provides a unified interface across the different realms. [Diagram created by the authors of this chapter.]

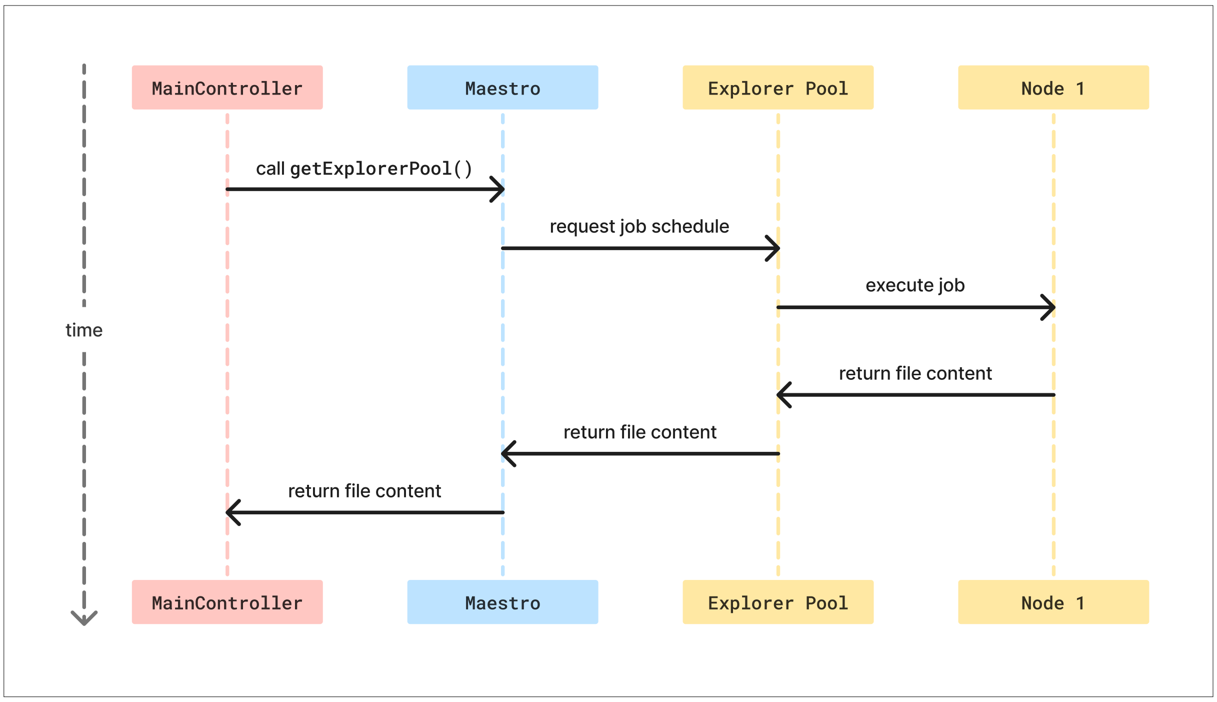


Figure 5.2: Sequence diagram for the `getFileContent` function. [Diagram created by the authors of this chapter.]

5.2.1 Explorer Pool

The explorer pool is a managed pool of web workers, responsible for handling data analysis and interactions with Git repositories. A set number of individual self-contained explorer workers are spawned by a central pool master. Maestro provides new workloads, which are distributed evenly across the worker pool, as illustrated in Figure 5.2. Each worker node is a continuously running process, awaiting a new job when idle. Every node is written in WebAssembly [WACG 2024a] and leverages the open-source library libgit2 [libgit2 2024] to interact with the Git repository chosen by the user.

5.2.2 Renderer Pool

The renderer pool is a collection of web workers which generate bitmaps for visualisation tiles. Distribution is handled through a central pool master, which schedules jobs and assigns them to idle workers. Each job results in a single bitmap image (tile) representing one file in the repository, which is used in the masonry grid on the main visualisation canvas.

5.2.3 SQLite Database

For performance reasons, an SQLite [SQLite 2024] database is used to store certain indexable metadata such as the names of authors (developers), so they can be efficiently queried. The SQLite database runs inside the web browser in its own web worker through WebAssembly. It is not persisted between sessions.

5.2.4 UI Controllers

Higher-Order UI Components use view models based on MobX [MobX 2024] to store state. These view models are centrally managed through the ViewModelController, which handles their assignment and keeps a reference for garbage collection. This controller is exposed to the Maestro via the MainController, responsible for grouping common UI functionality, variables, and controllers. User settings are handled

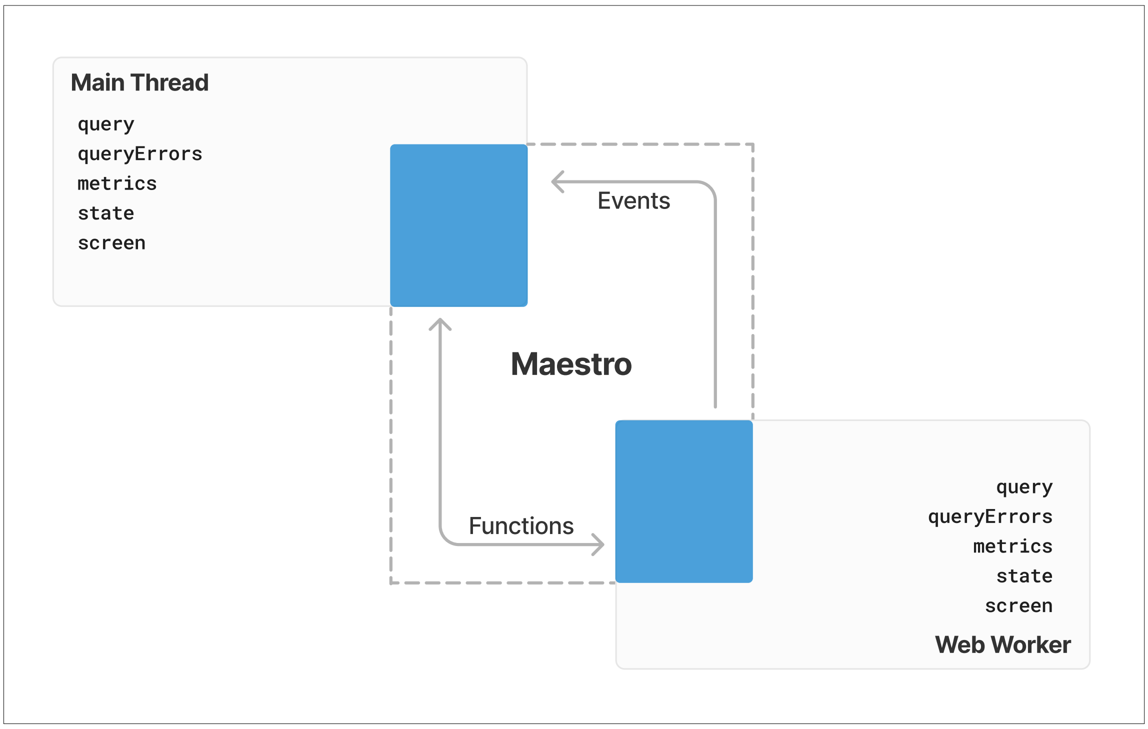


Figure 5.3: The Maestro controller has two parts: MaestroMain and MaestroWorker. MaestroMain is created in the main thread. MaestroWorker is invoked within a worker thread to manage longer-running asynchronous tasks. [Diagram created by the authors of this chapter.]

through the SettingsController, which exposes all settings as a single JavaScript object, and manages their state and persistence.

5.2.5 UI Components

The project has two distinct types of UI component: Primitive UI Components, which keep track of state internally in typical React fashion, and Higher-Order UI Components, which have a view model assigned to them to access global state. The three most notable Higher-Order UI Components are:

- Canvas: The Canvas component provides an interactive way to navigate the visualisation. Tiles are arranged in a masonry grid, and users can interact with the canvas through mouse and touch.
- QueryInput: The QueryInput component aggregates all supported query modules in a single component. It is directly attached to the Maestro and the query interface.
- Timeline: The Timeline component aggregates functionality to navigate time within a repository. It displays commits and the selected start and end dates in an interactive way.

5.2.6 Maestro

Gizual's architecture is based upon a single main controller called Maestro. Maestro is responsible for managing the overall state of the application, including loading a repository, selection of files, and the visualisation chosen by the user. To accomplish this while reducing the chance of blocking the main thread, Maestro actually consists of two entities, MaestroMain and MaestroWorker, as shown in Figure 5.3.

The MaestroMain instance is created within the main thread and is responsible for handling user input and maintaining global state for the user interface in a MobX store, to allow UI components to subscribe to changes and update themselves accordingly. This allows the user interface to remain responsive, while the application works in the background. The MaestroWorker instance is invoked within a worker thread and is responsible for handling longer-running asynchronous tasks. Global application state is synchronised between Maestro and MaestroWorker using a custom, event-based protocol.

5.3 State Management

Managing state in Gizual across different realms requires additional tooling compared to a standard single-threaded web application. In order to synchronise state globally, Gizual uses a set of functions and events across different threads. The underlying state management between realms is handled by Maestro. It uses MobX to store a reactive representation of the global application state in the main thread. UI controllers and components subscribe to this state representation and update accordingly.

5.3.1 Introduction to MobX

MobX [MobX 2024] is a state management library that aims to simplify the process of managing UI state. It does so by providing an easy way for components to automatically derive their state from the general application state. In the context of React, it ensures that only the components that are affected by a state update are re-rendered, providing a substantial boost to perceived speed and performance. MobX solves state management issues through *state*, *actions*, and *reactions*, which are all part of the library.

Actions in MobX are functions that modify state. Executing an action will task MobX with investigating which state variable was changed during this action, such that it can propagate a state update to all functions that “observe” that piece of state. Through that interface, components can all share a common similar state, stored in a MobX class, and re-render events will only propagate to those components that require them.

5.3.2 State Management in React

Typically, the proposed way of managing state within React is to propagate state updates down the rendering hierarchy. This goes hand in hand with the paradigm of “lifting state up”, a term used to describe the process of moving the assignment of state to the parent component if a sibling component might also need access. For smaller applications, this process is quite intuitive and easy to grasp. As applications grow in scale, business requirements often create a need to extend component state to encapsulate more design variants or general functionality. At some point, managing state manually can become quite cumbersome and error-prone.

The traditional way of managing React state uses the following syntax:

```
[state, setState] = React.useState(undefined)
```

As soon as a variable is stateful in the context of a React component, every usage of this state variable is tracked by React automatically, and changing the state by calling the `setState` function re-renders the component. Sometimes, state updates need to be conditional and based on other variables, like function properties or calls to an API. This is usually handled with the `useEffect` function, which allows developers to specify an optional set of tracked variables that trigger a function execution. One of the more common problems that React developers frequently face is unnecessary component re-rendering, usually triggered by non-idiomatic or erroneous usage of the `useEffect` function, which trigger unnecessary state updates.

To illustrate this concept, the example app in Listing 5.1 uses two implementations of a simple ToDo list, called in lines 21 and 22. The first function component, `TodoReact`, shown in Listing 5.2 uses plain React code and state management logic, while the second implementation, `TodoMobX` shown in Listing 5.3, uses MobX to manage state. The button that adds a ToDo to the list is deliberately placed as a sibling to the components that display the items. With simple React state management code, it would be an expected

```
 1 import React from "react";
 2 import "./App.css";
 3 import { TodoMobX, TodoStore } from "./TodoMobx";
 4 import { TodoReact } from "./TodoReact";
 5
 6 const store = new TodoStore();
 7
 8 function App() {
 9   const [todos, setTodos] = React.useState([
10     "Example Todo 1",
11     "Example Todo 2",
12   ]);
13
14   const addTodo = () => {
15     store.addTodo();
16     setTodos([...todos, ‘Example Todo ${todos.length + 1}‘]);
17   };
18
19   return (
20     <main>
21       <TodoReact todos={todos} />
22       <TodoMobX store={store} />
23       <button onClick={addTodo}>Add Todo</button>
24     </main>
25   );
26 }
27
28 export default App;
```

Listing 5.1: A simple React application with two alternative components to manage a ToDo list. The first component, `TodoReact` called in line 21, uses vanilla React to store state. It is shown in Listing 5.2. The second component, `TodoMobX` called in line 22, uses MobX and is shown in Listing 5.3.

procedure to lift state up, which means transferring it to the `App` component. This would allow for both the list and the button to access the state value and update it accordingly. Doing so works perfectly fine, but adds an unnecessary re-render for all components within the `App` component as soon as the state changes. The MobX implementation, on the other hand, only re-renders the components that are affected by the state change, which is only the `TodoList` component in this example.

The key strength of MobX lies in the automatic tracking of state changes by observing access to the values of *observable* properties within *tracked functions*. Instead of tracking the specific *values* of observable objects, MobX tracks the *property access*. In principle, this means that once the proper decorator annotations are placed to differentiate *observable*, *action* and *computed* objects appropriately, MobX automatically handles state propagation. In practice, this is enough for most application code, but sometimes reactivity needs to be defined on a more granular basis.

5.3.3 Advanced Reactivity Within MobX

When automatic reactivity tracking is not sufficient for the specific needs of an application, MobX provides custom functions to attach reactions to specific observable values. With the `autorun` function, it is possible to define an encapsulated function block, in which all used *observable* values are tracked automatically. Additionally, an optional delay can be specified to introduce a wait time before the initial function execution.

For more granularity, a custom `reaction` function can be used to define the *observable* objects that

```
function TodoReact({ todos }: { todos: string[] }) {
  return (
    <div className="container">
      {todos.map((t) => (
        <div>{t}</div>
      ))}
    </div>
  );
}

export { TodoReact };
```

Listing 5.2: A simple ToDo list implemented with basic React state management.

```
import { makeAutoObservable } from "mobx";
import { observer } from "mobx-react-lite";

class TodoStore {
  _todos: string[]; // Internal representation of state.

  constructor() {
    this._todos = ["Example Todo 1", "Example Todo 2"];

    /* Automatically let MobX track variables as state, functions as actions. */
    makeAutoObservable(this, undefined, { autoBind: true });
  }

  addTodo() {
    this._todos.push(`Example Todo ${this._todos.length + 1}`);
  }

  get todos() {
    return this._todos;
  }
}

const TodoMobX = observer(({ store }: { store: TodoStore }) => {
  return (
    <div className="container">
      {store.todos.map((t) => (
        <div>{t}</div>
      ))}
    </div>
  );
});

export { TodoStore };
export { TodoMobX };
```

Listing 5.3: A simple ToDo list implemented with React and MobX. An additional store is introduced to efficiently manage state updates and actions.

MobX needs to track manually. A supplied callback function will then automatically be executed once any of the supplied dependencies update. This behaviour is similar to the functionality React provides with its `useEffect` hook.

Regardless of using `autorun` or `reaction`, both functions exist outside the scope of MobX's automatic garbage collection and cleanup pipeline, so they need to be disposed of manually when no longer needed. Within Gizual, both `autorun` and `reaction` are used across parts of the application to fine-tune the reactivity of components. The architectural split between View and ViewModel, explained in Section 5.2, often requires a granular approach to state management to reduce unnecessary re-renders whilst keeping state consistent and synchronous across the entire application.

5.3.4 MobX Usage in Gizual

In Gizual, Maestro encapsulates the application state into specific objects, called observable boxes in MobX terminology. UI components can individually subscribe to specific parts of state they depend on by observing these boxes. Listing 5.4 shows a truncated example of the observable boxes used by Maestro.

5.4 Query Interface

Users can customise the output of the visualisation through a defined set of input parameters. These parameters are grouped into a single query object. This object is a crucial part of the application state, and is encapsulated into a separate reactive box within Maestro.

The query interface allows communication between the user interface and the data layer. It consists of a strict data schema to represent the user's input and desired output. To make this data accessible to the main thread and other realms, the query object is JSON-serialisable. It satisfies the `QuerySchema` interface shown in Listing 5.5. The query is configured using three scopes: `commit-range`, `files`, and `visualisation`.

5.4.1 Scope `commit-range`

The number of files inside a repository usually varies over time. Each commit can add or remove files, thereby changing the available file list. To specify the desired range of commits, two options are available:

- `branch` and `rangeByDate`: Select commits from a specific branch within a given time range.
- `rangeByRev`: Select commits between two Git revisions (e.g. a commit id or a tag).

By specifying either of these options, a start commit and an end commit are selected. The end commit defines the available file list, which is then narrowed down by the `files` scope. The start commit saves time by limiting the backward extent of the Git blame operation on each file.

5.4.2 Scope `files`

The `files` scope defines the files the user would like to see. Gizual supports the following file selection types:

- `path:string`: Select files by their path using a glob pattern.
- `path:string[]`: Select files by their distinct paths using a file picker.
- `changedInRev:string`: Select files which were changed in the specified Git revision (e.g. a commit id or a tag).

```typescript
import { observable } from "mobx";

class Maestro {
  globalState = observable.box<State>(
    {
      screen: "welcome" as "welcome" | "initial-load" | "main",
      queryValid: false,
      repoLoaded: false,
      authorsLoaded: false,
      commitsIndexed: false,
      filesIndexed: false,
      error: undefined,
      currentBranch: "",
      tags: [],
      branches: [],
      remotes: [],
      // ... truncated
    },
    { deep: false }
  );

  metrics = observable.box<Metrics>(
    {
      numExplorerJobs: 0,
      numExplorerWorkersBusy: 0,
      numExplorerWorkersTotal: 0,
      numRendererJobs: 0,
      numRendererWorkersBusy: 0,
      numRendererWorkers: 0,
      numSelectedFiles: 0,
    },
    { deep: false }
  );

  query = observable.box<Query>(
    { branch: "", type: "file-lines" },
    { deep: false }
  );

  queryErrors = observable.box<QueryError[] | undefined>(undefined, {
    deep: false,
  });

  // ... truncated
}
```

Listing 5.4: Partial implementation of the global state management defined in Maestro. State is encapsulated into observable boxes, which leverage the MobX reactivity ecosystem.

```
export interface QuerySchema {
  branch: string;
  time:
    | {
        rangeByDate: [string, string] | string;
      }
    | {
        rangeByRev: [string, string] | string;
      };
  files:
    | {
        path: string[] | string;
      }
    | {
        lastEditedBy: string[] | string;
      }
    | {
        changedInRev: string[] | string;
      };
  type:
    | "file-lines"
    | "file-lines-full"
    | "file-mosaic"
    | "author-mosaic" /* Future work */
    | "author-contributions" /* Future work */
    | "file-bar" /* Future work */
    | "author-bar" /* Future work */;
  preset:
    | {
        gradientByAge: [string, string];
      }
    | {
        paletteByAuthor: [string, string][];
      };
}
```

Listing 5.5: TypeScript interface for the query schema. The `files`, `time` and `branch` properties define the target input for the visualisation. The `type` and `preset` properties define the desired output shape.

5.4.3 Scope `visualisation`

The `visualisation` scope defines two properties of the resulting visualisation: the visualisation type and its visual encoding. Gizual supports three visualisation types, shown in Figure 5.4:

- `file-lines`: Displays file contents as a series of coloured lines. Each line represents one line of code.
- `file-lines-full`: Displays file contents as a series of full-width coloured lines. Each line represents one line of code.
- `file-mosaic`: Displays file contents as a mosaic of coloured tiles. Each tile represents one line of code.

Gizual currently implements two different visual encodings:

- `gradientByAge`: Colours are assigned based on the age of the line of code.
- `paletteByAuthor`: Colours are assigned based on the author of the line of code.

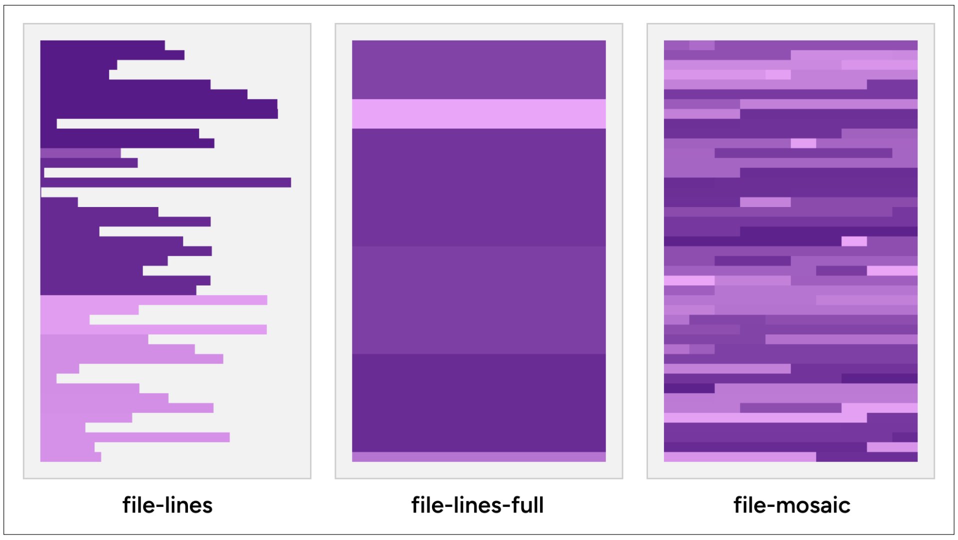


Figure 5.4: The three visualisation types provided by Gizual. [Image created by the authors of this chapter.]

Chapter 6

File I/O in Gizual

One of the main features of Gizual is the ability to explore repositories from a user's local file system, without any need to access them over the internet. To accomplish this, multiple files from within a repository's `.git/` directory have to be accessed and parsed. This chapter first provides an overview of the requirements around File I/O in Gizual. Then, each of the available web browser APIs for File I/O (see Section 3.2.4) are considered in turn. Finally, the chosen approach is discussed, depending on which browser is being used.

6.1 Requirements

Several characteristics had to be weighed in terms of File I/O, so as to be able to implement all of the features necessary in Gizual's data layer. These involved repository size, memory usage, browser support, usability, and support for web workers.

6.1.1 Repository Size

The amount of data stored within a repository is an important characteristic, since it represents the amount of data that needs to be processed by Gizual. In this context, the repository size refers to the size of the repository's `.git/` directory and all of its files and sub-directories. Its size is heavily influenced by the following factors:

- The number of commits within the repository.
- The frequency of changes within the repository.
- The size of each of the changes.
- The size of files within the repository.
- When the repository was last optimised using packfiles.

During the development of Gizual, four repositories of varying sizes were selected on GitHub to act as test repositories, with the goal of ensuring that Gizual could process as large a repository as possible:

- *Vue 2*: Small sized repository [You 2024].
- *React*: Medium sized repository [Meta 2024a].
- *VS Code*: Large sized repository [Microsoft 2024a].
- *Linux Kernel*: Very large sized repository [Torvalds 2024].

Repository	GitHub Path	Size	`.git/` Size	First Commit	# Commits
Vue 2	`vuejs/vue`	small	35 MB	10 Apr 2016	6,694
React	`facebook/react`	medium	657 MB	29 May 2013	25,873
VS Code	`microsoft/vscode`	large	1,025 MB	13 Nov 2015	134,419
Linux Kernel	`torvalds/linux`	very large	5,836 MB	16 Apr 2005	1,310,317

Table 6.1: The four GitHub repositories used to test Gizual. These statistics were collected on 29 Oct 2024 after cloning the repositories locally.

Each repository was freshly cloned on 29 Oct 2024. Table 6.1 gives an overview of the four test repositories.

6.1.2 Memory Usage

Each application running within the confines of a browser sandbox has its own memory space. This memory space is used to store the data, DOM elements, and the application's state. The memory space is limited and grows as the application runs. If too much memory is used, the browser will terminate the application or crash. Therefore, the memory usage of the application has to be considered when designing a data heavy local application like Gizual. As a result, it is not feasible to store the entire repository in memory.

6.1.3 Browser Support

Gizual aims to be compatible with all modern web browsers, so the following browsers are used to regularly test Gizual's data layer:

- *Google Chrome*: Versions 120 to 132 on MacOS and Windows.
- *Mozilla Firefox*: Versions 120 to 132 on MacOS and Windows.
- *Safari*: Versions 17 and 18 on MacOS.

6.1.4 Usability

While the data layer of Gizual is not directly tied to the user interface, it must still prioritise usability, as ease of use is a key project objective. Consequently, selecting a local repository for processing by Gizual should be straightforward, requiring minimal clicks, with short loading times, and intuitive interactions familiar to users.

6.1.5 Support for Web Workers

Gizual's data layer relies on a complex interplay of various optimisations, including parallel data processing using a pool of web workers. Therefore, the data layer must be capable of handling data processing across multiple web workers simultaneously.

6.2 File API

The File API is the most common method for accessing files within a browser sandbox. However, while this API allows users to select multiple files, it does not support accessing an entire directory with its files and sub-directories. This limitation is particularly problematic, since the `.git/` directory is hidden by default on most operating systems, making it impractical to require users to manually select all files within it. Consequently, the File API is unsuitable for Gizual's data layer.

Repository	GitHub Path	Commit ID	# Files	Load Time
Vue 2	`vuejs/vue`	9e88707	16.827	2.93 s
React	`facebook/react`	0bc3074	81.290	7.95 s
VS Code	`microsoft/vscode`	6c3a714	87.837	9.54 s
Linux Kernel	`torvalds/linux`	e42b1a9	86.733	9.14 s

Table 6.2: The number of files and the load time for a given commit for each of the four chosen test repositories when using the File and Directory Entries API. The load time refers to the time it takes between requesting the directory picker and being able to access the repository data from within JavaScript. Time spent in user interactions is not included. The measurements were taken using screen recording on a MacBook Pro 2023, M3 Max with 48 GB RAM using Chrome v132. They were collected using the most recent commit to the main branch on 29 Oct 2024.

6.3 File and Directory Entries API

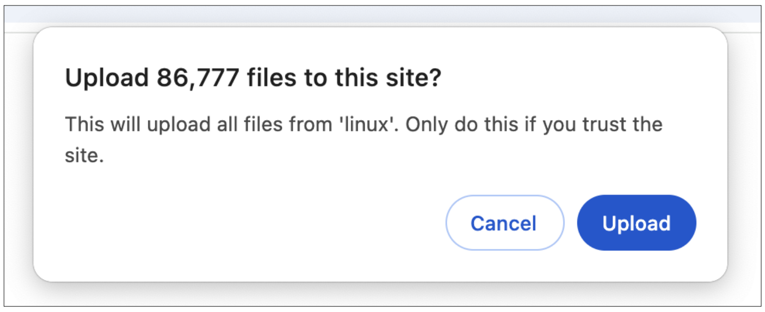
The File and Directory Entries API extends the capabilities of the File API. Though considered experimental, it is widely supported by browsers and allows users to select entire directories, making all files and sub-directories available in the browser environment. However, one limitation is that hidden directories, such as `.git/`, are not automatically visible in the file picker of the browser. Therefore, users must select the root directory of a repository to include everything within it.

This approach works well for smaller repositories, but as the size of a repository increases, performance issues can arise. Large numbers of files and directories can lead to the browser freezing, which severely affects the user experience. Table 6.2 shows the number of files for a specific commit in each of the four chosen test repositories, as well as the load time between selecting the directory and being able to access the data from within JavaScript. The browser needs to traverse the entire directory structure when a user selects a directory, which can be time-consuming for large repositories, especially for web-based repositories having deeply nested `node_modules/` directories.

In terms of usability, when using the File and Directory Entries API, the browser may ask the user for permission with a misleading dialogue, like those shown in Figure 6.1. Although the `.git/` directory can technically be accessed with the File and Directory Entries API, its effectiveness is limited by repository size and usability concerns, making it at best a suboptimal fallback option for Gizual.

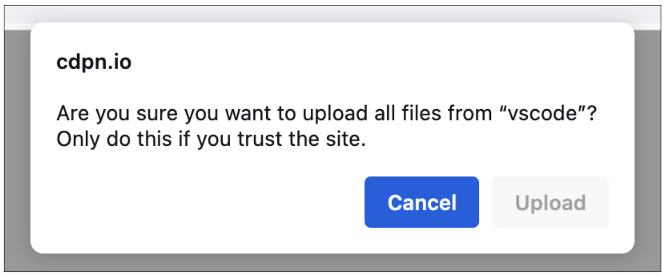
6.4 File System Access API

The File System Access API is designed to provide a means to access a directory of a user's local file system from within a web browser sandbox directly. As such, it is independent of the size or number of files within a repository. However, the API is not widely supported by modern web browsers and only works in Chromium-based browsers. In terms of usability, the browser prompts the user for permission, properly informing the user that the data will be readable only within the browser sandbox, as illustrated in Figure 6.2. Furthermore, since FileSystemHandle objects are transferable objects, they can be passed to web workers to parallelise the process of data extraction. As such, the File System Access API is the best solution for accessing the `.git/` directory of a repository for Chromium-based browsers.


Upload 86,777 files to this site?

This will upload all files from 'linux'. Only do this if you trust the site.

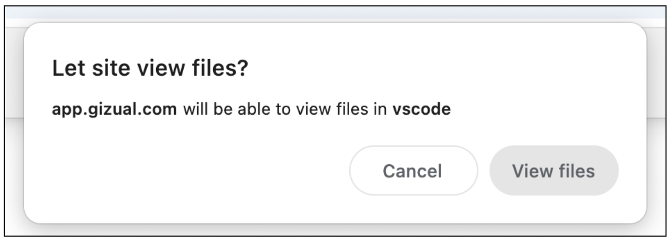
(a) Chromium v132.


cdpn.io

Are you sure you want to upload all files from “vscode”? Only do this if you trust the site.

(b) Firefox v132.

Figure 6.1: The dialogues displayed by browsers when requesting explicit permission to access a repository’s main directory with the File and Directory Entries API. The wording is misleading, because it uses the word *upload*. Users might assume this means the data is being transmitted to a server, but in reality, the data is only accessed within the browser sandbox. [Images created by the author of this thesis.]


Let site view files?

app.gizual.com will be able to view files in **vscode**

Figure 6.2: The message displayed by the browser when requesting explicit read permission to access a repository’s main directory with the File System Access API using Chrome v132. [Image created by the author of this thesis.]

6.5 Drag and Drop API

The Drag and Drop API enables users to drag the root folder of a repository into the browser window, providing a method to access the folder and traverse all its files and nested directories without delay or freezing. However, because the FileSystemDirectoryEntry object provided by the API is not a transferable object, it cannot be passed to other parts of the application. Consequently, the Drag and Drop API alone is inadequate for accessing a repository’s `.git/` directory. While loading all files and directories into memory might be feasible for small repositories, it is not practical for larger ones.

6.6 Combining Drag and Drop and Origin Private File System APIs

The Drag and Drop API is not sufficient for the purpose of accessing the repository data from within one or multiple web workers. However, while the Origin Private File System API does not provide a way to directory access local data from a user's local file system, it does allow parallel reading of files from multiple directories. Therefore, a combination of the Drag and Drop and the Origin Private File System APIs is feasible for the use case of Gizual. It is implemented as follows:

- The user selects the root directory of the repository and drags it into the browser window.
- The browser creates a FileSystemDirectoryEntry object for the selected directory.
- The required data from the `.git/` directory is copied into the Origin Private File System.
- A reference to the directory within the Origin Private File System can now be used interchangeably with the File System Access API.

To combine the Drag and Drop and Origin Private File System APIs, an import function was implemented in TypeScript. It is shown in Listing 6.1.

6.7 Chosen Solution

The chosen solution for File I/O in Gizual leverages a combination of several of the File APIs discussed previously. For Chromium-based browsers, the recommended approach is to use the File System Access API for direct interaction with the user's local file system, as it provides the most effective and efficient solution. For other browsers, Gizual uses a combination of the Drag and Drop API and the Origin Private File System API. This approach increases loading time, since data must be transferred from the user's local file system to the browser sandbox. However, since data within the Origin Private File System is not stored in memory, repository size is not a limitation. Importantly, since the Origin Private File System shares the same API as the File System Access API, there is no need to implement separate processing infrastructures.

As a result of the chosen solution, the following limitations are present:

- For Chromium-based browsers, embedding Gizual in an `<iframe>` is not possible, since the File System Access API is not available in iframes.
- For Safari and Firefox browsers, incognito mode (private browsing mode) is not supported, since the Origin Private File System is not available in this mode.

```
export async function importDirectoryEntry(
  rootEntry: FileSystemDirectoryEntry
): Promise<FileSystemDirectoryHandle> {
  let directory = await navigator.storage.getDirectory();
  await clearDirectory(directory);

  directory = await directory.getDirectoryHandle("repo", { create: true });
  await clearDirectory(directory);

  const importEntry = async (
    source: FileSystemEntry,
    target: FileSystemDirectoryHandle
  ) => {
    if (isFileSystemDirectoryEntry(source)) {
      if (shouldIgnoreFilePath(source.name)) return;

      const dirName = source.name;
      let targetHandle: FileSystemDirectoryHandle;
      try {
        targetHandle = await target.getDirectoryHandle(dirName, {
          create: false,
        });
      } catch {
        targetHandle = await target.getDirectoryHandle(dirName, {
          create: true,
        });
      }
      const reader = source.createReader();
      await new Promise<void>((resolve, reject) => {
        const parseEntries = async (entries: FileSystemEntry[]) => {
          for (let i = 0; i < entries.length; i++) {
            const entry = entries[i];
            await importEntry(entry, targetHandle);
          }
          if (entries.length > 0) {
            reader.readEntries(parseEntries, reject);
          } else { resolve(); }
        };
        reader.readEntries(parseEntries, reject);
      });
    } else if (isFileSystemFileEntry(source)) {
      const fileName = source.name;
      const fileHandle = await target.getFileHandle(fileName, { create: true });
      const writable = await fileHandle.createWritable();
      await new Promise((resolve, reject) => {
        source.file((file) => {
          writable.write(file).then(resolve).catch(reject);
        }, reject);
      });
      await writable.close();
    }
  };
  await importEntry(rootEntry, directory);
  return directory;
}
```

Listing 6.1: Function to import the `.git/` directory of a repository from a FileSystemDirectoryEntry provided by the Drag and Drop API into the browser's Origin Private File System.

Chapter 7

Native Code in Gizual: git-explorer

Gizual utilises libgit2 [libgit2 2024], a C implementation of Git [Chacon and Straub 2024], to extract metadata and blame information from repositories directly within the browser sandbox. To accomplish this, libgit2 is compiled to WebAssembly [Sletten 2022], which is executed within the browser sandbox, and is wrapped in JavaScript for ease of use. The whole module is called `git-explorer`. This chapter explores the difficulties of compiling libgit2 to WebAssembly and interacting with it from JavaScript, as well as the necessary workarounds to overcome these challenges.

7.1 Compiling libgit2 to WebAssembly

Compiling libgit2 to WebAssembly involves several steps to ensure it operates smoothly within the browser environment. To facilitate this process, the Rust programming language [Klabnik and Nichols 2023] was chosen, due to its strong focus on safety, concurrency, and performance.

7.1.1 Rust Target: wasm32-wasi

WebAssembly is a binary instruction format for compiled languages like C, C++, and Rust. However, WebAssembly itself does not provide a standard library to interact with the host environment to perform common tasks such as file I/O, networking, and other system calls. To overcome this limitation, the WebAssembly System Interface (WASI) Version 0.1 (preview1) was chosen as the standard library for WebAssembly [WASI 2024]. WASI provides a set of API specifications for WebAssembly modules to interact with the host environment in a standardised way. Rust can target WASI as the standard library, enabling Rust code to be compiled to a WebAssembly module that expects a WASI-compliant host environment.

To create a WASI-compliant host environment, the WebAssembly module must be instantiated in the browser with a set of importable functions according to the WASI API specifications. There are libraries available that offer basic implementations of these APIs, such as Wasmer [Wasmer 2024]. Unfortunately, due to specific requirements concerning file I/O and interoperability with JavaScript, these implementations did not provide the required customisations necessary for Gizual's use case. Therefore, a set of APIs was created specifically for Gizual's data layer. Listing 7.1 illustrates the most important function definitions for the WASI APIs used by Gizual. The complete list of definitions can be found within the Gizual repository [Schintler and Steinkellner 2024c]. To compile Rust code to a `wasm32-wasi` target, the following commands can be used:

```
rustup target add wasm32-wasi      // install dependencies
cargo build --target wasm32-wasi  // compile the code
```

```
interface GizualWasiFunctions {
  // Handling of main arguments
  args_sizes_get(argc: number, argv_buf_size: number): number;
  args_get(argv: number, argv_buf: number): number;

  // Environment variables
  environ_sizes_get(environ_count: number, environ_size: number): number;
  environ_get(environ: number, environ_buf: number): number;

  // Time
  clock_time_get(id: number, precision: bigint, time: number): number;

  // Access to randomness
  random_get(buf: number, buf_len: number): void;

  // File I/O
  fd_close(fd: number): number;
  fd_filestat_get(fd: number, filestat_ptr: number): number;
  fd_fdstat_get(fd: number, fdstat_ptr: number): number;
  fd_fdstat_set_flags(fd: number, fdstat_ptr: number): number;
  fd_prestat_get(fd: number, prestat_ptr: number): number;
  fd_prestat_dir_name(fd: number, path_ptr: number, path_len: number): number;
  fd_read(fd: number, iovs_ptr: number, iovs_len: number,
    nread_ptr: number): Promise<number> | number;
  fd_pread(fd: number, iovs_ptr: number, iovs_len: number, offset: bigint,
    nread_ptr: number): number;
  fd_write(fd: number, iovs_ptr: number, iovs_len: number,
    nwritten_ptr: number): Promise<number> | number;
  fd_pwrite(fd: number, iovs_ptr: number, iovs_len: number,
    offset: bigint, nwritten_ptr: number): number;
  fd_advise(fd: number, offset: bigint, len: bigint, advice: number): number;
  path_open(fd: number, dirflags: number,path_ptr: number, path_len: number,
    oflags: number, fs_rights_base: bigint, fs_rights_inheriting: bigint,
    fd_flags: number, fd_out_ptr: number): Promise<number> | number;
  path_filestat_get(fd: number, flags: number, path_ptr: number,
    path_len: number, filestat_ptr: number): Promise<number> | number;
  fd_readdir(fd: number, buf: number, buf_len: number, cookie: bigint,
    bufused_ptr: number): Promise<number> | number;

  // 18 additional functions have been excluded because they are unused by Gizual
  // and exist solely as stubs to meet the WASI interface requirements.
}
```

Listing 7.1: The most important function definitions for the WASI APIs used by Gizual. These functions are used to interact with the host environment and are required to instantiate WebAssembly modules targeting a WASI-compliant host.

```
function syncReadFile(file: FileSystemFileHandle): string {
  let result = "";
  let finished = false;

  file.getFile().then((file) => {
    // this is never called, since the event loop is stuck within the while loop
    file.text().then((text) => {
      result = text;
      finished = true;
    });
  });

  while (!finished) {
    // busy wait until the asynchronous call has completed
    // will result in a deadlock within the browser event loop
    console.log("waiting for file read to complete");
  }

  return result;
}
```

Listing 7.2: Naive and simplified approach to implementing a synchronous function to read a file. Since the Origin Private File System and File System Access API are asynchronous, busy wait is used to wait for the asynchronous call to complete. This approach would result in a deadlock and is therefore not a viable solution.

7.1.2 Asynchronous APIs and Asyncify

Within the WASI API, all functions are defined to be synchronous. However, some of these functions need to be implemented using asynchronous browser APIs, such as those responsible for reading files. In synchronous functions, it is not possible to await a promise using the `await` keyword. Therefore, a naive approach might be to simply call the asynchronous browser API and busy wait until the asynchronous call has completed, as shown in Listing 7.2. However, this approach would result in a deadlock within the browser event loop, since the event loop would be blocked waiting for the asynchronous call to complete and would never yield control to the browser due to the synchronous busy wait.

To overcome this limitation, the compiled WebAssembly module must be modified to facilitate asynchronicity within imported WASI functions. This approach builds upon approaches articulated by Alon Zakai, notably in his blog post [Zakai 2019] and presentation [Zakai 2022]. These resources provide an in-depth analysis of employing Binaryen's `wasm-opt` tool to effectively transform WebAssembly modules, thereby enabling asynchronous functions to be invoked from native code.

In practical terms, when a WebAssembly module is asyncified, additional code is injected wherever a WASI function is called. This is done to snapshot the current state of the WebAssembly state machine and unwind its stack, allowing the module to return control back to the JavaScript code earlier. Within the JavaScript context, the asynchronous browser API is then called, and the result is awaited. Subsequently, the snapshot data is restored, and the call stack is returned to its original state so that the WebAssembly module can resume execution.

To apply the asyncify transformation to the WebAssembly module, the following command can be used:

```
wasm-opt -O3 --asyncify -o asyncified.wasm original.wasm
```

Unfortunately, the additional injected code leads to significantly increased size of the WebAssembly module, and a degradation of runtime performance. Performance issues are discussed later in Section 8.1.

```
(module
  (import "env" "async_func" (func $async_func))
  (export "memory" (memory 0))
  (export "main" (func $main))
  (memory 1 1)
  (func $main
    (call $async_func)
  )
)
```

Listing 7.3: Original WebAssembly module using an imported function call `async_func`. If this external function was actually an asynchronous function, this example would not work since it has not been transformed. Listing 7.4 shows the asyncified WebAssembly module.

Listing 7.3 shows an original WebAssembly module, and Listing 7.4 shows its asyncified counterpart. The asyncified version has around 6 times the number of instructions. Listing 7.5 provides the JavaScript code used to run the transformed WebAssembly module.

7.1.3 libgit2 Rust bindings using git2-rs

To interact with libgit2 from Rust, the `git2-rs` crate (package) was chosen, because it provides a high-level, idiomatic interface for working with Git repositories [git2-rs 2024]. This crate is designed to be easy to use and offers a simple API for performing common Git operations, such as blaming and traversing Git history.

Since Gizual runs entirely within the browser sandbox, certain adaptations were necessary to ensure compatibility of the git2-rs crate with the `wasm32-wasi` target. Consequently, the crate was forked and adapted to the `wasm32-wasi` target by disabling features such as `pthread` and `mmap`, among other modifications. The forked crate is available in a separate GitHub repository [Schintler and Steinkellner 2024a].

7.2 Rust to JavaScript Interoperability

When interfacing JavaScript with Rust through WebAssembly, several challenges arise, particularly concerning the sharing of data structures and memory management. Therefore, a wrapper is required to support interoperability between JavaScript and Rust. This architecture is shown in Figure 7.1.

libgit2 is first wrapped in a layer of custom Rust code, which is then compiled to the `wasm32-wasi` target. Within the JavaScript module, a corresponding WASI-compliant ABI (Application Binary Interface) bridge handles interactions with the file system, exposing stdin and stdout to control execution as well as FSA-FS, which is a custom file system implementation converting native file I/O interactions to browser-based File System Access API calls.

7.2.1 Rust to JavaScript File I/O Interface

As shown previously in Listing 7.1, the WASI API defines a set of functions to interact with a local file system, prefixed with `fd_` and `path_`. During the development of Gizual, two interchangeable implementations of file systems were developed to support different use cases:

- *MEMORY-FS*: A simple in-memory file system that can be used for testing purposes with no asynchronous operations.

```
(module
 (type $0 (func))
 (type $1 (func (param i32)))
 (type $2 (func (result i32)))
 (import "env" "async_func" (func $async_func))
 (global $__asyncify_state (mut i32) (i32.const 0))
 (global $__asyncify_data (mut i32) (i32.const 0))
 (memory $0 1 1)
 (export "memory" (memory $0))
 (export "main" (func $main))
 (export "asyncify_start_unwind" (func $asyncify_start_unwind))
 (export "asyncify_stop_unwind" (func $asyncify_stop_unwind))
 (export "asyncify_start_rewind" (func $asyncify_start_rewind))
 (export "asyncify_stop_rewind" (func $asyncify_stop_rewind))
 (export "asyncify_get_state" (func $asyncify_get_state))
 (func $main
  (local $0 i32)
  (local.set $0
   (block $__asyncify_unwind (result i32)
    (if
     (i32.or
      (i32.eqz (global.get $__asyncify_state))
      (i32.eqz
       (if (result i32)
        (i32.eq (global.get $__asyncify_state) (i32.const 2))
        (then
         (i32.store
          (global.get $__asyncify_data)
          (i32.sub (i32.load (global.get $__asyncify_data)) (i32.const 4))
         )
         (i32.load (i32.load (global.get $__asyncify_data)))
        )
        (else (local.get $0))
       )
      )
     )
     (then
      (call $async_func)
      (drop
       (br_if $__asyncify_unwind (i32.const 0)
        (i32.eq (global.get $__asyncify_state) (i32.const 1))
       )))
    )
    (return)
   )
  )
  (i32.store (i32.load (global.get $__asyncify_data)) (local.get $0))
  (i32.store
   (global.get $__asyncify_data)
   (i32.add (i32.load (global.get $__asyncify_data)) (i32.const 4))
  )
 )
 ;; The following function definitions have been truncated for brevity.
 ;; $asyncify_start_unwind
 ;; $asyncify_stop_unwind
 ;; $asyncify_start_rewind
 ;; $asyncify_stop_rewind
)
```

Listing 7.4: WebAssembly module after applying the asyncify transformation. The transformation adds additional code before and after every function call to allow asynchronous APIs to be used. The original WebAssembly module can be found in Listing 7.3. The corresponding JavaScript code to execute this module can be found in Listing 7.5.

```
const wasmCode = await fetch("./asyncify-showcase-transformed.wasm").then(
  (response) => response.arrayBuffer());
const wasmModule = new WebAssembly.Module(wasmCode);
let inAsyncCall = false;

// The location where the asyncify data is stored. (Hardcoded for brevity)
// This as a memory address within the WebAssembly module.
const DATA_ADDR = 16;

// A view to interact with the continuous memory space
const view = new Int32Array(exports.memory.buffer);
const instance = new WebAssembly.Instance(wasmModule, {
  env: {
    async_func: () => {
      if (!inAsyncCall) {
        // The function is called for the first time
        // pointer to start of stack is stored in first 4 bytes
        // DATA_ADDR >> 2 represents DATA_ADDR / 4, since stack aligned to 4 bytes
        view[DATA_ADDR >> 2] = DATA_ADDR + 8;

        // The end of the stack is stored in the second 4 bytes of memory
        view[(DATA_ADDR + 4) >> 2] = 1024;

        // start unwind process, saving defined stack for later rewind
        moduleExports.asyncify_start_unwind(DATA_ADDR);
        inAsyncCall = true;

        // asynchronous function can now be called,
        // such as Origin Private File System
        navigator.storage.getDirectory().then((directory) => {
          // The stack is then rewound, and execution continues by calling the
          // original main function again.
          moduleExports.asyncify_start_rewind(DATA_ADDR);
          moduleExports.main();
        });
      } else {
        // After rewind, execution ends here and unwinding process is stopped,
        // allowing execution to continue normally
        moduleExports.asyncify_stop_rewind();
        inAsyncCall = false;
      }
    },
  },
});
const moduleExports = instance.exports as Record<string, Function>;

// Run the main function of the WebAssembly module
moduleExports.main();
moduleExports.asyncify_stop_unwind();
```

Listing 7.5: JavaScript code to run the transformed WebAssembly module of Listing 7.4. For every function call that needs to support asynchronous APIs, a significant amount of additional code is required to unwind and rewind the WebAssembly state machine. This is a simplified example, the actual implementation in Gizual uses dynamically generated memory addresses.

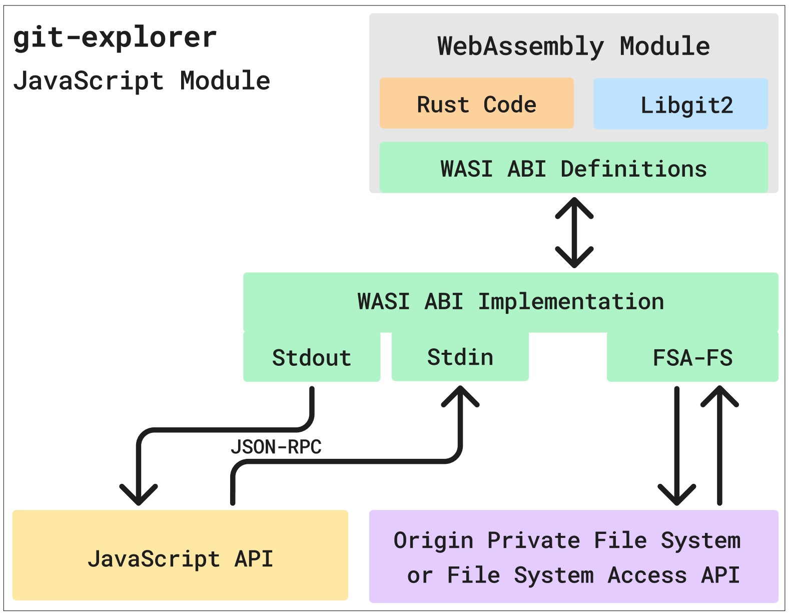


Figure 7.1: Gizual's `git-explorer` module enables libgit2 to be run from JavaScript within the browser sandbox. It consists of a WebAssembly module with embedded libgit2 and a custom JavaScript wrapper to enable bidirectional communication and file I/O. [Diagram created by the author of this thesis.]

- *FSA-FS*: A file system bridging C-style file I/O calls with asynchronous operations, provided either by the File System Access API (for Chromium-based browsers) or the Origin Private File System (fallback for other browsers). It leverages the transformations described in Section 7.1.2 to support asynchronous functions in WASI.

7.2.2 JavaScript to Rust Command Interface

Typically, a Rust module targetting `wasm32-wasi` acts in a similar way to a command line application. The module is executed with a specific set of command line arguments and environment variables. After execution, the module exits and the standard output is made available to the JavaScript code. However, this process requires the module to be reinstantiated and the repository data parsed for each execution, since no state is preserved between executions.

To circumvent this, Gizual maintains the WebAssembly module instantiated in memory between jobs. A job represents a single operation that needs to be executed, such as blaming a file or looking up a commit id within a repository. To accomplish this, JavaScript code communicates with the WebAssembly module through a bidirectional JSON-RPC interface using file descriptors 0 (stdin) and 1 (stdout). The communication is enabled by a custom file system implementation based upon WASI function calls. The JavaScript wrapper is called `WasiRuntime` and takes care of supporting the asyncify transformations, the custom file system implementation, and the command interface via stdin and stdout. The main function of the WebAssembly module is shown in Listing 7.6. An example of how this module can be used from JavaScript is provided in Listing 7.7.

```rust
pub fn main() {
  let mut exp = Explorer::new();

  // The repository is opened. It is always located at /repo
  exp.cmd_open_repository(&OpenRepositoryParams {
    path: "/repo".to_owned(),
  });

  // This loop is called continuously until the shutdown flag is set to true
  while !exp.shutdown.load(std::sync::atomic::Ordering::Relaxed) {
    let line = std::io::stdin().lock().lines().next();
    if line.is_none() {
      continue;
    }
    let line = line.unwrap();

    if line.is_err() {
      eprintln!("Failed to read from stdin: {}", line.err().unwrap());
      continue;
    }
    let line = line.unwrap();

    // The ping command is used to check when the module is ready
    if line == "PING" {
      println!("PONG");
      continue;
    }
    // Each line is a single JSON-PRC payload and deserialized before it is handled
    let request = Explorer::deserialize_request(line.clone());

    match request {
      Ok(request) => exp.handle(request, explorer_callback),
      Err(err) => {
        eprintln!("Failed to deserialize request {}: {}", line, err);
      }
    }
  }
}

fn explorer_callback(response: Response) {
  let result = Explorer::serialize_response(response);

  match result {
      Ok(result) => {
          println!("{}", result);
      }
      Err(err) => {
          panic!("Failed to write to stdout: {}", err)
      }
  }
}
```

Listing 7.6: This main function of the WebAssembly module in Rust. Its main loop runs continuously, awaiting requests from stdin, until the `shutdown` flag is set to true. These requests are deserialised and handled. The `explorer_callback` function returns the response to the JavaScript code.

```javascript
async function setup() {
  // This example utilises the File System Access API to open a directory picker.
  // This is only supported by Chromium-based browsers
  const handle = await window.showDirectoryPicker();

  // The WasiRuntime is a custom implementation of the WASI API
  // specifically designed for Gizual and spawns a WebWorker which
  // then executes the WebAssembly module
  const runtime = await WasiRuntime.create({
    moduleUrl: "./explorer-main.wasm",
    folderMappings: {
      "/repo": handle,
    },
  });

  await runtime.run({ env: {}, args: [] });

  // Awaiting till the module is ready
  await runtime.writeStdin("PING\n");
  const stdout = await runtime.readStdout();
  if (stdout !== "PONG") {
    throw new Error("unable to start module");
  }
  return runtime;
}

async function main() {
  const runtime = await setup();

  const payload = {
    method: "get_branches",
  };

  const payloadString = JSON.stringify(payload) + "\n";
  this.runtime.writeStdin(payloadString);

  const stdout = await runtime.readStdout();
  const result = JSON.parse(stdout);
  console.log(result);
  /*
  Output:
  {
    "data": {
      "branches": [
        "main",
        "feature/add-button"
      ]
    },
    "end": true
  }
  */
}
```

Listing 7.7: The JavaScript code to use the WebAssembly module. The `setup` function is used to open a directory picker and instantiate the WebAssembly module and WasiRuntime. This runtime is the custom JavaScript wrapper for the WebAssembly module. Afterwards, within the `main` function, a job is executed by sending a JSON-RPC request to the WebAssembly module. The response is then parsed and printed to the console.

Chapter 8

Performance and Multi-Threading

Gizual's data layer is designed to be highly scalable and capable of processing large repositories locally. It must also generate blames for multiple files simultaneously. Since reaction and loading times are critical for the user experience, the data layer must be able to handle data exploration and processing as quickly as possible. This chapter discusses performance challenges, describes a performance comparison of the same Git operation performed across several environments, and gives insights into how multi-threading was leveraged to keep performance under control.

8.1 Performance Challenges

Executing Git operations within the browser sandbox presents several performance challenges. This section discusses the reasons for these performance degradations.

8.1.1 Libgit2 Limitations

Outside of the browser sandbox, the Git command line interface (CLI) is the most common way to interact with Git repositories and execute blame commands. However, the CLI is not designed to be built in WebAssembly and therefore cannot be executed in a web browser environment. The pure C implementation of Git, libgit2, was therefore chosen as the basis for the data layer of Gizual.

Unfortunately, the performance of libgit2 is known to be inferior to the Git CLI [Libgit2 2015]. David Kastrup implemented a patch for the Git CLI in 2014 to improve the performance of Git blame [Kastrup 2014a]. The previous system utilised a single linear list for organising blame entries, resulting in inefficient quadratic runtime, since every subtask had to scan through mostly irrelevant data. His new method assigns each subtask its own dedicated data, linked through "struct origin" chains to specific commits. By organising commits in a priority queue based on commit dates, this approach significantly improves processing efficiency, often achieving speedups of three times or more for large files with diverse histories. Unfortunately, due to licensing debates, he did not give permission to relicense his contribution for use in libgit2 [Kastrup 2014b].

8.1.2 WebAssembly Limitations

Gizual's Git module, `git-explorer`, is executed within the browser sandbox, using the libgit2 library compiled to WebAssembly. However, WebAssembly has significant performance limitations compared to native code. Research conducted by Jangda et al. [2019] concluded that code compiled for WebAssembly can be up to two times slower than native code, due to the limited instruction set and lack of optimisations in browser implementations.

8.2 Performance Analysis

To compare the performance of Git operations across different environments, an experimental comparison of a Git blame operation was conducted. The following sections describe the methodology used to conduct the performance comparison and the results of the comparison.

8.2.1 Methodology

To assess the performance limitations described in the previous section, benchmarks were conducted to compare the Git CLI, libgit2, and Gizual's `git-explorer` module. These benchmarks measured the time taken to execute a single blame operation within a repository. The Vue 2 and React code repositories were chosen for testing, similar to Chapter 6. Since both repositories are web-based projects containing a `package.json` file, this file was selected for the blame operation. Given its frequent modifications and existence since the initial commit, the package.json file is presumed to represent a worst-case scenario. All benchmarks were conducted on a MacBook Pro 2023, M3 Max, with 48 GB RAM running macOS Sonoma 14.1.1.

8.2.1.1 Git CLI

The benchmark was conducted using the Git CLI version v2.39.3. By default, the Git CLI uses a terminal-based editor like `vim` to paginate the output of the `blame` command, which makes measurements of the blame duration more difficult. To avoid this, the Git CLI was started with the `--no-pager` option, which disables pagination and prints the output directly to the terminal. To measure the actual runtime duration of the the terminal command, `hyperfine` [Peter 2024] was used with a warmup of 5 runs and a measurement of 10 runs. To conduct the benchmark, the following command was used:

```
hyperfine --warmup 5 --runs 10 "git --no-pager blame ./package.json"
```

8.2.1.2 Libgit2

libgit2 is a C library, which means there is no native command line interface. However, libgit2 provides examples for how to implement popular Git CLI commands using libgit2. Therefore, the libgit2 repository version v1.8.0 was cloned and compiled. After compilation, the `lg2` binary within the build output at `build/examples/lg2` was used to measure the performance of the blame command. Similar to the Git CLI, `hyperfine` was employed to measure the actual runtime duration. The following command was used:

```
hyperfine --warmup 5 --runs 10 "lg2 blame package.json"
```

8.2.1.3 Gizual's Git-Explorer module

To evaluate the performance of Gizual's `git-explorer` module, a browser-based benchmark was conducted. To accomplish this, a simplified version of Gizual's data layer was used with a distinct build target. The exact implementation can be found in the Gizual repository at `apps/benchmark-app/`. Each of the test repositories was copied into the browsers Origin Private File System, before 10 runs of blames were performed and their execution time was measured using the `console.time()` and `console.timeEnd()` functions.

8.2.2 Results

The results of the performance comparison are shown in Figure 8.1. The Git CLI outperforms the other two options by about a factor 10. Using libgit2 natively is somewhat faster than Gizual's `git-explorer` module, where a single Git blame can take over 2 seconds in some cases. Despite this, the only feasible solution for Gizual's data layer is to implement libgit2 with WebAssembly in the browser environment.

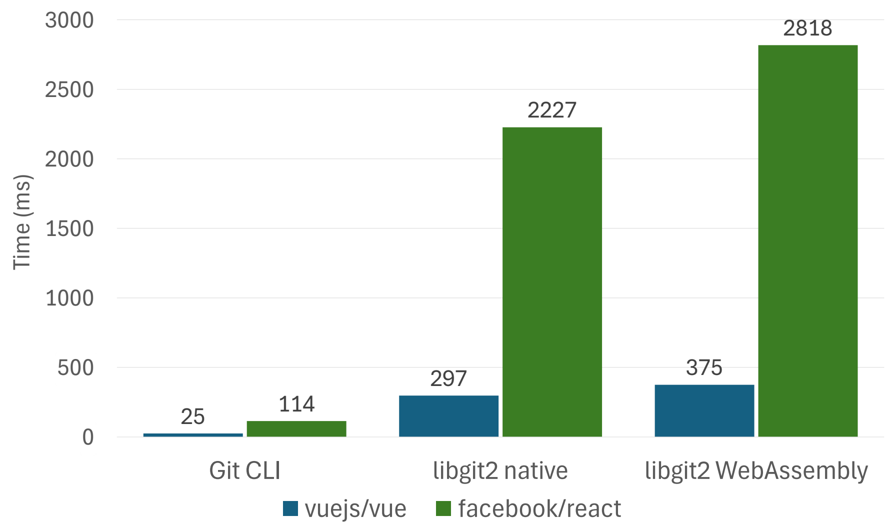


Figure 8.1: Comparison of blame performance of Git CLI, libgit2, and Gizual's `git-explorer` module (libgit2 in WebAssembly). Blame was calculated for the single file `package.json` in both the Vue 2 and React repositories. [Chart created by the author of this thesis.]

8.3 Multi-Threading Approach

Based on the results of the performance analysis, it is apparent that the blame performance of libgit2 in WebAssembly is dramatically slower than the Git CLI. While some improvements could be made by applying different kinds of caching optimisations within the File I/O layer, the overall performance is mediocre at best. However, while a single blame request can not be sped up without significant extra effort, like writing a more performant blame algorithm similar to the one used in the Git CLI, it is possible to execute multiple blames in parallel using web workers.

8.3.1 Pool Architecture

Gizual uses a pool architecture to efficiently distribute jobs across multiple web workers, abstracting the complexities of spawning and managing workers, scheduling tasks based on priority, and tracking progress. As depicted in Figure 8.2, a Pool Master coordinates a set of Pool Nodes. Jobs are collected from the Maestro and organised into a queue by the Pool Master. Every 50 milliseconds, the state of Pool Nodes and the job queue is assessed. When a Pool Node is idle, the Pool Master assigns the highest priority job available. This design ensures efficient resource utilisation and responsive processing within Gizual's data layer.

8.3.2 Job Definition

To distribute tasks across the pool of workers, a job definition is created for each task. An example job definition is shown in Listing 8.1. The job definition is a serialisable object containing the following properties:

- *id*: A unique identifier for the job, required to keep track of the job in the queue and match a response to a job.
- *priority*: The priority of the job, used to sort the jobs in the queue based on relevance.
- *method*: The method to call within the WebAssembly module.
- *params*: The parameters to pass to the method.

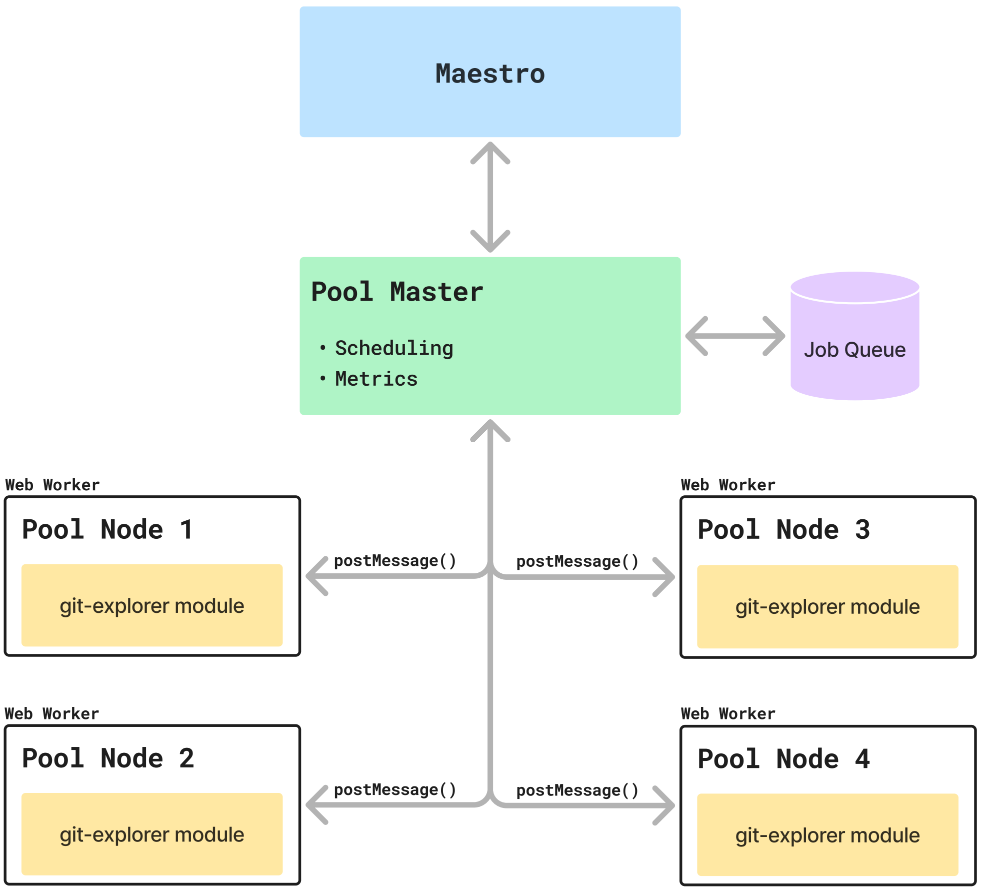


Figure 8.2: A simplified diagram of the pool architecture used by Gizual to distribute jobs to web workers. The Pool Master manages a queue of prioritised jobs and a pool of workers, each of which is responsible for executing a single job at a time. [Diagram created by the author of this thesis.]

```
{
  "jsonrpc": "2.0",
  "id": 5,
  "priority": 1,
  "method": "get_blame",
  "params": {
    path: "./package.json",    // path to the file to blame
    rev: "v2",                 // git revision to start the blame from
    sinceRev: "v1"             // git revision to stop the blame at
  },
}
```

Listing 8.1: Example of a job used within Gizual's pool of web workers to distribute tasks. The job requests a blame of the file `package.json` at revision `v2`, and the blame should stop at revision `v1`.

8.3.3 Job Prioritisation

The data layer of Gizual is optimised to enhance Gizual's frontend performance. This optimisation is achieved by using a prioritised job queue.The priority of blame requests is dynamically determined based on their position relative to the viewport. When a file is visible in the viewport, its blame requests receive the highest priority. If a file is near the currently visible area, its blame request is assigned medium priority. Otherwise, the blame request is deferred to ensure efficient resource utilisation.

8.3.4 Job Cancellation

Gizual's frontend frequently updates the data layer with information about which files are currently visible in the viewport. This visibility data is used to update the priority of blame requests. If a file is no longer visible and is not close to the viewport, any associated blame request is canceled. This strategy prevents unnecessary processing, allowing resources to be reallocated to more critical tasks, thereby optimising performance and responsiveness.

8.3.5 Pool Size

The pool size defines the number of parallel jobs that can be executed simultaneously. It must be scaled according to the available hardware resources to ensure optimal performance. Spawning too many workers can lead to resource contention and potential performance degradation. Consequently, the pool size in Gizual is dynamically set based on the `navigator.hardwareConcurrency` property [MDN 2024b], which provides an estimate of the number of logical processors available.

Chapter 9

Outlook and Future Work

The web browser has become a ubiquitous application platform, yet achieving parity with natively installed applications still needs to be improved. Browser vendors must prioritise security and privacy to protect users from malicious websites. Nonetheless, new features and capabilities are needed to interact with the underlying operating system, such as accessing the local file system, to close the gap between web and native applications. The File System Access API offers a promising solution for local file system interactions. However, its support is limited to Chromium-based browsers, with Firefox declining implementation due to security concerns [Mozilla 2024b]. Although Gizual has employed a workaround for this limitation, a more robust and widely supported solution is necessary.

In terms of executing native code within the browser sandbox, the use of WebAssembly shows potential, yet the absence of a standardised ABI necessitates workarounds and leads to inconsistencies. The WebAssembly System Interface (WASI) aims to provide a standardised ABI, but much functionality still relies on simplified or stubbed implementations, which might work for small concepts but do not scale to the full range of use cases. Gizual has managed to circumvent this by proxying WASI APIs to browser-native APIs using custom code and asyncify transformations. However, a more robust and standardised solution for native file system interaction remains elusive. Native code typically expects synchronous interaction with underlying hardware, while browser-based abstractions within the browser sandbox are asynchronous. This fundamental limitation will only be resolved if browser runtimes are adapted. Efforts to achieve this are underway: JS Promise Integration [WebAssembly 2024] is currently being implemented and may soon be standardised. This feature aims to bridge the gap between synchronous native code in WebAssembly and asynchronous browser APIs, eliminating the need for asyncify transformations and enabling seamless interaction between native code and the browser sandbox. Currently, Chromium has implemented this experimental feature behind a flag. Once standardised and adopted by all major browsers, it might facilitate more robust interaction with local resources through WebAssembly modules.

While Gizual's core features are in place, further enhancements could improve its data layer. One significant limitation is memory usage and runtime performance, as discussed in Chapter 8. Various performance bottlenecks have been identified, some stemming from the use of the libgit2 library and its lack of blame operation optimisation. The Rust-based gitoxide library [Thiel 2024] offers a promising alternative, although it is still under development and does not yet support blame operations. Transitioning to gitoxide as it matures may result in a more performant and robust data layer.

Additionally, Gizual's file I/O layer is implemented naively, with each explorer worker repeatedly reading the same file. A more efficient approach could involve using a SharedArrayBuffer to cache frequently accessed files in the browser sandbox, enhancing file I/O performance while potentially introducing new memory usage challenges.

Beyond technical improvements, Gizual could benefit from additional features, such as gathering more information beyond file blames, including overall statistics on contributions and commits.

Chapter 10

Concluding Remarks

This thesis described the development of the data layer for Gizual, a modern web-based application designed to visualise source code from Git repositories. Numerous challenges were encountered and solved, including the execution within the browser of native Rust code in conjunction with a C library. A custom file system was implemented to bridge the gap between browser APIs and traditional native code. Performance considerations were meticulously addressed by provisioning pools of synchronised web workers, to ensure that the data layer could deliver a responsive and efficient user experience.

With Gizual, developers and managers are now equipped to easily explore and visualise Git repositories, gaining valuable insights into code evolution, frequency of changes, and their own impact on the codebase. The data layer of Gizual represents a significant advance towards creating in-browser, privacy-preserving, web applications, demonstrating the potential for future innovations in this domain.

The source code of Gizual is open-source and available on GitHub [Schintler and Steinkellner 2024b]. A deployed version can be accessed at `gizual.com`.

Appendix A

Developer Guide

This guide explains the structure of the Gizual code base and the development and build processes in the Gizual project. It is aimed at developers who might wish to modify or extend Gizual. The most current version with general contribution guidelines is available online in the Gizual repository [Schintler and Steinkellner 2024b].

This appendix was written jointly by Stefan Schintler and Andreas Steinkellner.

A.1 Development Stack

The Gizual project requires the following dependencies to be installed on the local development system:

- *Node.js v18 or higher*: Used for building and bundling the application and running the server-side code [OpenJS 2024b].
- *Yarn Berry v3.5.0*: Used as a package manager because of its workspace support [yarn 2024a].
- *Rust v1.79.0 or higher*: Used for building the native parts of Gizual [RF 2024].
- *Git*: Used for version control of the source code [Git 2024].

A.2 Project Structure

Gizual uses the workspace feature of yarn [yarn 2024b] to group packages and their dependencies into self-contained folders. Although technically self-contained, these workspaces can co-depend on other workspaces within the repository. Listing A.1 shows the structure of the Gizual code repository.

A.2.1 Gizual API

The Gizual API workspace is located in the `apps/gizual-api/` folder and contains the source code for the server-side service, which provides a proxy for cloning Git repositories. This is necessary since GitHub and other Git hosting services do not allow cross-origin requests (CORS) [MDN 2024a] from external domains. However, since Gizual also offers the ability to explore local Git repositories, this service is entirely optional. It is written in TypeScript, based on the Express v4 framework [OpenJS 2024a], and makes use of the simple-git [King 2024a] package to interact with Git hosting services.

```
gizual/
├── apps/
│   ├── gizual-api/
│   │   ├── src/
│   │   ├── package.json
│   │   └── ...
│   ├── gizual-app/
│   │   ├── src/
│   │   ├── package.json
│   │   └── ...
│   └── ...
├── packages/
│   ├── explorer/
│   │   ├── src/
│   │   ├── package.json
│   │   └── ...
│   ├── explorer-web/
│   │   ├── src/
│   │   ├── package.json
│   │   └── ...
│   └── maestro/
└── ...
```

Listing A.1: The overall structure of Gizual's source code repository, showing the most important workspaces. Each workspace defines its dependencies in its own `package.json` file.

A.2.2 Gizual App

The Gizual App workspace is located in the `apps/gizual-app/` folder and contains the source code for the frontend web application. It is written in TypeScript and based on the React framework [Meta 2024b]. For state management, it uses the MobX [MobX 2024] library. Most of its UI components are based on the Mantine [Mantine 2024] library, and component styles are written in SCSS. This package aggregates all other dependencies and serves as the main build target of the Gizual application. Building and bundling is done with Vite [Vite 2024].

A.2.3 Maestro

The Maestro workspace is located in the `packages/maestro/` folder and contains the source code for the main controller of the application. It is written in TypeScript and designed as the main gateway between the user interface and the data layer. It is responsible for managing global application state and data processing. A detailed description of this package can be found in Section 5.2.

A.2.4 Explorer

The Explorer workspace is located in the `packages/explorer/` folder and contains the source code for the `git-explorer` module. Its web bindings are located the `packages/explorer-web/` folder. The Explorer is written in Rust [RF 2024] and is based on the libgit2 [libgit2 2024] library. This module is responsible for exploring local Git repositories and providing a list of files and their blame information. It is used by the Gizual App to display the visualisation via the Maestro package. It also provides the pooling mechanism to use multiple instances of explorer workers in parallel. The package exposes a simple TypeScript API to interact with the underlying native module.

A.2.5 Renderer

The Renderer workspace is located in the `packages/file-renderer/` folder and contains the source code for the file visualisation module. Developed in TypeScript, it uses the OffscreenCanvas API to render images off the main thread, enabling parallel processing. It contains rendering functions for each available visualisation type, by leveraging the blame information provided by the Explorer. Communication between the Gizual App and the Renderer is handled via Maestro.

A.3 Data Flow

Gizual depends on many interconnected components, running in separate threads. Figure A.1 provides a structural overview of the interconnected packages and their communication pathways. Data flow in Gizual begins with the `updateQuery()` function, which is executed on initial load, and then each time the user modifies the query. This triggers Maestro to update its states. Maestro determines the set of selected files off the main thread. It estimates the file lengths and generates information for each visualisation tile. This information is then passed back to main thread and triggers a MobX state update. This state is used to render the visualisation tiles in the Gizual App.

A corresponding component is created for each tile. Each component is coupled with a block in Maestro, and an attached IntersectionObserver keeps track of its position in the viewport. This information is provided to Maestro, which in turn determines if a re-render of this tile is necessary. Each time a tile is rendered, Maestro passes the cached blame information to the renderer pool and executes the rendering function. If a tile's blame information is not yet cached, it is retrieved from the explorer pool first. The rendered image is stored in an Object URL within the browser environment, and its URL is passed back to Maestro, which in turn triggers a MobX reaction to update the block's state and image. This two-way reactive binding between the tile representation on the canvas and its internal state in Maestro is the foundation of Gizual's data flow.

A.4 Build and Deploy

The Gizual project consists of multiple distinct build targets, which all need to be built in the correct order. Usually, web-based applications rely on a single build and bundling tool like Vite. However, the Rust-based parts of Gizual require a more complex build process, which cannot easily be represented in conventional build scripts, especially with hot module reloading during development.

To address this issue and streamline the overall build process, the Gizual project contains its own designated build tool, called `please`. This tool is based on yarn workspaces and allows a workspace to reference local dependencies, input source files, and necessary build commands. This information is defined within the `package.json` file of each workspace. Based on this information and a target workspace, `please` automatically traverses all referenced workspaces recursively and builds a dependency graph. Recursive dependencies are detected and resolved. This graph is then used to determine the correct order of building and bundling. The building process for each workspace is only executed, if the source files of the corresponding workspace have changed.

To simplify the deployment process, Gizual is bundled into a single docker image. After `please` has built the application, the build artefacts are copied into the docker image. The docker image is then pushed to a container registry and is ready to be deployed.

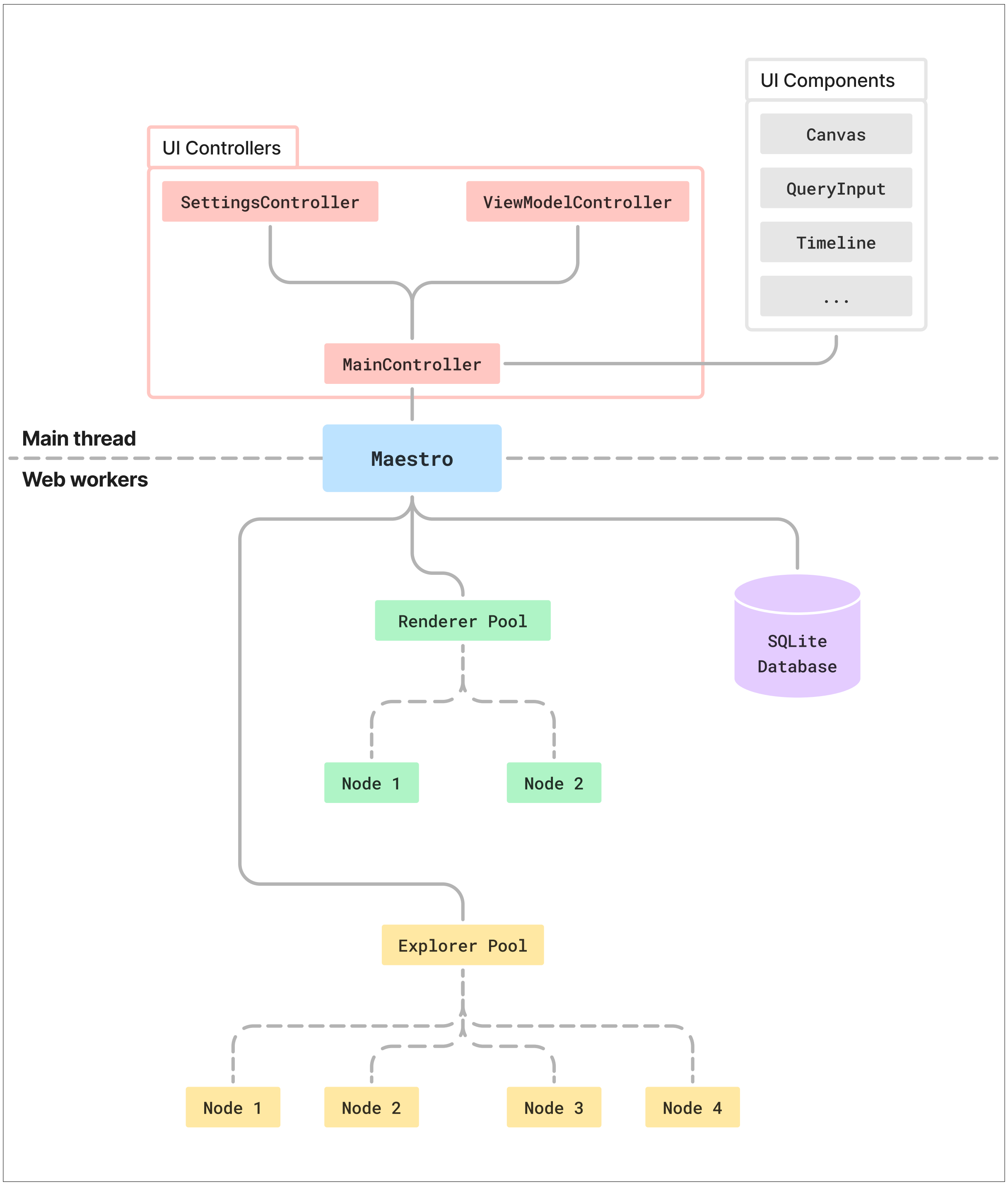


Figure A.1: The software architecture of Gizual. UI controllers and UI components are instantiated in the main thread. The explorer pool, renderer pool, and SQLite database are executed asynchronously in web workers inside the browser. The Maestro provides a unified interface across the different realms. [Diagram created by the authors of this chapter.]

Bibliography

@kdy1 [2024]. *Speedy Web Compiler*. https://esbuild.github.io/ (cited on page 6).

Andrews, Keith [2021]. *Writing a Thesis: Guidelines for Writing a Master's Thesis in Computer Science*. Graz University of Technology, Austria. 10 Nov 2021. https://ftp.isds.tugraz.at/pub/keith/thesis/ (cited on page xiii).

Apple [2024]. *WebKit*. 25 Aug 2024. https://webkit.org/ (cited on page 11).

Babel [2024]. *Babel*. Babel Contributors. https://babeljs.io/ (cited on page 6).

Baumgartner, Stefan [2023]. *Typescript Cookbook: Real World Type-Level Programming*. 12 Sep 2023. ISBN 1098136659 (cited on page 6).

Bell, Joshua [2016a]. *File and Directory Entries API - Initial Commit*. 17 May 2016. https://github.com/WICG/entries-api/commit/e37f6a2e745cf73161488536ef9447dcfb4e1cb7 (cited on page 15).

Bell, Joshua [2016b]. *Intent to Ship: global names for FileSystemFileEntry, FileSystemDirectoryEntry*. 16 Aug 2016. https://groups.google.com/a/chromium.org/g/blink-dev/c/xTfeHGlcIQg/m/X0Q4iDJHAgAJ?pli=1 (cited on page 15).

Berjon, Robin [2006]. *File (Upload) API Specification*. 18 Oct 2006. https://w3.org/TR/2006/WD-file-upload-20061018/ (cited on page 14).

Berners-Lee, Tim [1989]. *Information Management: A Proposal*. May 1989. https://cds.cern.ch/record/369245/files/dd-89-001.pdf (cited on page 3).

Berners-Lee, Tim and Dan Connolly [1993]. *Hypertext Markup Language - 2.0*. RFC 1866. 15 Nov 1993. https://tools.ietf.org/html/rfc1866 (cited on page 3).

Bitbucket [2024]. *Bitbucket*. 26 Aug 2024. https://bitbucket.org/ (cited on page 30).

Bugden, William and Ayman Alahmar [2022]. *Rust: The Programming Language for Safety and Performance*. Proc. 2nd International Graduate Studies Congress (IGSCONG'22) (Virtual). 11 Jun 2022. doi:10.48550/arXiv.2206.05503 (cited on page 7).

caniuse.com [2024]. *File System Access API*. 25 Aug 2024. https://caniuse.com/native-filesystem-api (cited on page 19).

Chacon, Scott and Ben Straub [2024]. *Pro Git*. Version 2.1.434. 04 Sep 2024. https://github.com/progit/progit2/releases/download/2.1.434/progit.pdf (cited on pages 27, 33, 55).

Chromium [2024a]. *Blink*. 25 Aug 2024. https://chromium.org/blink/ (cited on page 11).

Chromium [2024b]. *Chromium*. 25 Aug 2024. https://chromium.org/Home/ (cited on page 11).

Chromium [2024c]. *Multi-Process Architecture*. 14 Oct 2024. https://chromium.org/developers/design-documents/multi-process-architecture/ (cited on page 12).

Dijkstra, Edsger W. [1974]. *On the Role of Scientific Thought*. In: *Selected Writings on Computing: A Personal Perspective*. Edited by David Gries. EWD 447. Springer, 30 Aug 1974. ISBN 0387906525. doi:10.1007/978-1-4612-5695-3_12. https://cs.utexas.edu/~EWD/transcriptions/EWD04xx/EWD447.html (cited on page 38).

Ecma [2024]. *ECMAScript 2024 Language Specification*. 2024. https://ecma-international.org/wp-content/uploads/ECMA-262_15th_edition_june_2024.pdf (cited on page 5).

Eick, Stephen G., Joseph L. Steffen, and Eric E. Sumner Jr. [1992]. *Seesoft: A Tool for Visualizing Line Oriented Software Statistics*. IEEE Transactions on Software Engineering 18.11 (Nov 1992), pages 957–968. doi:10.1109/32.177365. http://www.sdml.cs.kent.edu/library/Eick92.pdf (cited on page 1).

Flanagan, David [2020]. *JavaScript: The Definitive Guide*. 7th Edition. O'Reilly, 23 Jun 2020. ISBN 1491952024 (cited on page 5).

Git [2024]. *Git*. 11 Jun 2024. https://git-scm.com/ (cited on pages 27, 34, 75).

git2-rs [2024]. *git2-rs*. 05 Nov 2024. https://github.com/rust-lang/git2-rs (cited on page 58).

GitHub [2024a]. *Git Installation*. 07 Jul 2024. https://github.com/git/git/blob/v2.45.2/INSTALL#L110-L167 (cited on page 34).

GitHub [2024b]. *GitHub*. 26 Aug 2024. https://github.com/ (cited on pages 30–31).

GitLab [2024]. *GitLab*. 26 Aug 2024. https://gitlab.com/ (cited on page 30).

Google [2024]. *Skia: The 2D Graphics Library*. https://skia.org/ (cited on page 11).

Haas, Andreas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien [2017]. *Bringing the Web up to Speed with WebAssembly*. Proc. 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017) (Barcelona, Spain). 14 Jun 2017, pages 185–200. doi:10.1145/3062341.3062363. https://www.cs.tufts.edu/~nr/cs257/archive/andreas-rossberg/webassembly.pdf (cited on page 8).

Hara, Kentaro [2018]. *How Blink works*. 14 Aug 2018. https://bit.ly/how-blink-works (cited on page 11).

Haverbeke, Marijn [2024]. *Eloquent JavaScript*. 4th Edition. 2024. ISBN 1593279507. https://eloquentjavascript.net/Eloquent_JavaScript.pdf (cited on page 38).

Hilton, William [2024a]. *isomorphic-git*. 06 Jul 2024. https://isomorphic-git.org/ (cited on page 34).

Hilton, William [2024b]. *lightning-fs*. 06 Jul 2024. https://github.com/isomorphic-git/lightning-fs (cited on page 34).

Jangda, Abhinav, Bobby Powers, Emery D. Berger, and Arjun Guha [2019]. *Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code*. Proc. 2019 USENIX Annual Technical Conference (ATC '19) (Renton, Washington, USA). 10 Jul 2019, pages 107–120. https://usenix.org/conference/atc19/presentation/jangda (cited on page 65).

JSON-RPC [2013]. *JSON-RPC 2.0 Specification*. JSON-RPC Working Group, 04 Jan 2013. https://jsonrpc.org/specification (cited on page 8).

Kastrup, David [2014a]. *[PATCH 1/2] blame: large-scale performance rewrite*. 26 Apr 2014. https://public-inbox.org/git/1398470210-28746-1-git-send-email-dak@gnu.org/ (cited on page 65).

Kastrup, David [2014b]. *Re: A few contributor's questions*. 03 Feb 2014. https://public-inbox.org/git/87vbwwxfol.fsf@fencepost.gnu.org/ (cited on page 65).

King, Steve [2024a]. *Simple Git*. 19 Nov 2024. https://npmjs.com/package/simple-git (cited on page 75).

King, Steve [2024b]. *simple-git*. 07 Jul 2024. https://github.com/steveukx/git-js (cited on page 34).

Klabnik, Steve and Carol Nichols [2023]. *The Rust Programming Language*. 2nd Edition. No Starch Press, 28 Feb 2023. 560 pages. ISBN 1718503105. https://doc.rust-lang.org/book/ (cited on pages 7, 55).

Kulkarni, Vinay and Sreedhar Reddy [2003]. *Separation of Concerns in Model-Driven Development*. IEEE Software 20.5 (Sep 2003), pages 64–69. ISSN 0740-7459. doi:10.1109/MS.2003.1231154. https://citeseerx.ist.psu.edu/document?doi=5b27c112f72fa5bc8e6f412412674ac0968a3c0b (cited on page 38).

Libgit2 [2015]. *Git Blame is slow*. 01 Apr 2015. https://github.com/libgit2/libgit2/issues/3027 (cited on page 65).

libgit2 [2024]. *libgit2*. https://libgit2.org/ (cited on pages 34, 40, 55, 76).

Lie, Håkon Wium [1994]. *Cascading HTML Style Sheets – A Proposal*. 10 Oct 1994. https://w3.org/People/howcome/p/cascade.html (cited on page 4).

Lie, Håkon Wium and Bert Bos [2005]. *Cascading Style Sheets: Designing for the Web*. 04 Apr 2005. ISBN 0321193121. https://archive.org/details/cascadingstyles000lieh/mode/2up (cited on page 4).

LLVM [2024]. *LLVM Compiler Infrastructure Project*. LLVM Developer Group, 04 Jul 2024. https://llvm.org/ (cited on page 8).

Mantine [2024]. *Mantine*. 19 Nov 2024. https://npmjs.com/package/@mantine/core (cited on page 76).

McCathieNevile, Charles and Doug Schepers [2006]. *Clipboard Operations for the Web 1.0: Copy, Paste, Drag and Drop*. 15 Nov 2006. https://w3.org/TR/2006/WD-clipboard-apis-20061115/ (cited on page 19).

MDN [2024a]. *Cross-Origin Resource Sharing (CORS)*. MDN Web Docs, 19 Nov 2024. https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS (cited on page 75).

MDN [2024b]. *Navigator: hardwareConcurrency property*. MDN Web Docs, 12 Nov 2024. https://developer.mozilla.org/en-US/docs/Web/API/Navigator/hardwareConcurrency (cited on page 69).

Meta [2024a]. *React*. 29 Oct 2024. https://github.com/facebook/react (cited on page 49).

Meta [2024b]. *React*. Meta Platforms, 19 Nov 2024. https://npmjs.com/package/react (cited on page 76).

Microsoft [2024a]. *Visual Studio Code*. 29 Oct 2024. https://github.com/microsoft/vscode (cited on page 49).

Microsoft [2024b]. *TypeScript*. https://typescriptlang.org/ (cited on page 6).

Microsoft [2024c]. *Visual Studio Code*. https://code.visualstudio.com/ (cited on page 6).

MobX [2024]. *MobX: Simple, Scalable State Management*. MobX Contributors, 28 Jun 2024. https://mobx.js.org/ (cited on pages 40, 42, 76).

Mozilla [2024a]. *Firefox Browsers*. 25 Aug 2024. https://mozilla.org/en/firefox/ (cited on page 11).

Mozilla [2024b]. *Mozilla Standards Positions: File System Access API*. 01 Dec 2024. https://mozilla.github.io/standards-positions/#native-file-system (cited on page 71).

Nebel, Ernesto and Larry Masinter [1995]. *Form-Based File Upload in HTML*. RFC 1867. Nov 1995. https://datatracker.ietf.org/doc/html/rfc1867 (cited on page 14).

Netscape [1995]. *Netscape and Sun Announce JavaScript*. 04 Dec 1995. https://web.archive.org/web/20070916144913/https://wp.netscape.com/newsref/pr/newsrelease67.html (cited on page 5).

OpenJS [2024a]. *Express - Node.js Web Application Framework*. OpenJS Foundation, 19 Nov 2024. https://expressjs.com/ (cited on page 75).

OpenJS [2024b]. *Node*. OpenJS Foundation, 19 Nov 2024. https://nodejs.org/ (cited on page 75).

Pe, Maksim [2022]. *HTML Element Content Categories*. 07 Aug 2022. https://de.wikipedia.org/wiki/Hypertext_Markup_Language#/media/Datei:HTML_element_content_categories.svg (cited on page 4).

Peter, David [2024]. *Hyperfine*. 11 Nov 2024. https://github.com/sharkdp/hyperfine (cited on page 66).

Raggett, Dave [1997]. *HTML 3.2 Reference Specification*. 14 Jan 1997. https://w3.org/TR/2018/SPSD-html32-20180315/ (cited on page 14).

RF [2024]. *Rust Programming Language*. Rust Foundation. https://rust-lang.org/ (cited on pages 7, 75–76).

Salomonsen, Peter [2024]. *wasm-git*. 07 Jul 2024. https://github.com/petersalomonsen/wasm-git (cited on page 34).

Schintler, Stefan [2024]. *Gizual Data Layer: Enabling Browser-Based Exploration of Git Repositories*. Master's Thesis. Graz University of Technology, Austria, 09 Dec 2024. https://ftp.isds.tugraz.at/pub/theses/sschintler-2024-msc.pdf (cited on page 1).

Schintler, Stefan and Andreas Steinkellner [2024a]. *git2-rs Fork*. 05 Nov 2024. https://github.com/gizual/git2-rs (cited on page 58).

Schintler, Stefan and Andreas Steinkellner [2024b]. *Gizual*. 18 Nov 2024. https://github.com/gizual/gizual (cited on pages 1, 73, 75).

Schintler, Stefan and Andreas Steinkellner [2024c]. *Gizual WASI Functions*. 01 Dec 2024. https://github.com/gizual/gizual/blob/v1.0.0-alpha.24/packages/wasi-shim/src/wasi.ts#L235 (cited on page 55).

Seidel, Eric [2014]. *Blink's First Birthday*. 03 Apr 2014. https://blog.chromium.org/2014/04/blinks-first-birthday.html (cited on page 11).

Sletten, Brian [2022]. *WebAssembly: The Definitive Guide: Safe, Fast, and Portable Code*. O'Reilly, 04 Jan 2022. 332 pages. ISBN 1492089842 (cited on pages 8, 55).

SO [2022]. *Stack Overflow Developer Survey 2022*. Stack Overflow, 01 Jun 2022. https://survey.stackoverflow.co/2022/ (cited on pages 27–28).

Spinellis, Diomidis [2012]. *Git*. IEEE Software 29.3 (01 May 2012), pages 100–101. doi:10.1109/MS.2012.61. https://www.dmst.aueb.gr/dds/pubs/jrnl/2005-IEEESW-TotT/html/v29n3.pdf (cited on page 27).

SQLite [2024]. *SQLite*. 10 Aug 2024. https://sqlite.org/ (cited on page 40).

Steinkellner, Andreas [2024]. *Gizual User Interface: Browser-Based Visualisation for Git Repositories*. Master's Thesis. Graz University of Technology, Austria, 09 Dec 2024. https://ftp.isds.tugraz.at/pub/theses/asteinkellner-2024-msc.pdf (cited on page 1).

Surma, Das [2019]. *Use Web Workers to Run JavaScript Off the Browser's Main Thread*. web.dev, 05 Dec 2019. https://web.dev/articles/off-main-thread (cited on page 37).

Tamary, Jonathan and Dror G. Feitelson [2015]. *The Rise of Chrome*. PeerJ Computer Science (28 Oct 2015). doi:10.7717/peerj-cs.28 (cited on page 11).

Thiel, Sebastian [2024]. *gitoxide*. 07 Jul 2024. https://github.com/Byron/gitoxide (cited on pages 34, 71).

Thompson, Clive [2023]. *How Rust Went From a Side Project to the World's Most-Loved Programming Language*. 14 Feb 2023. https://doc.rust-lang.org/stable/rust-by-example/ (cited on page 7).

Torvalds, Linus [2024]. *Linux*. 29 Oct 2024. https://github.com/torvalds/linux (cited on page 49).

Turner, Jonatahn [2014]. *Announcing TypeScript 1.0*. 02 Apr 2014. https://devblogs.microsoft.com/typescript/announcing-typescript-1-0/ (cited on page 6).

Vite [2024]. *Vite*. 19 Nov 2024. https://npmjs.com/package/vite (cited on page 76).

W3C [2019]. *Memorandum of Understanding Between W3C and WHATWG*. 28 May 2019. https://w3.org/2019/04/WHATWG-W3C-MOU.html (cited on page 3).

WACG [2024a]. *WebAssembly*. WebAssembly Community Group, 30 Aug 2024. https://webassembly.org/ (cited on pages 8, 40).

WACG [2024b]. *WebAssembly Text Format*. WebAssembly Community Group. https://webassembly.github.io/spec/core/text/index.html (cited on page 8).

Wallace, Evan [2024]. *esbuild*. https://esbuild.github.io/ (cited on page 6).

WASI [2024]. *WebAssembly System Interface*. 31 Oct 2024. https://wasi.dev/ (cited on page 55).

Wasmer [2024]. *Wasmer*. 05 Nov 2024. https://wasmer.io/ (cited on page 55).

WAWG [2024]. *WebAssembly JavaScript Interface*. WebAssembly Working Group, 02 Jul 2024. https://webassembly.github.io/spec/js-api/ (cited on page 21).

WebAssembly [2024]. *JavaScript-Promise Integration Proposal*. 22 Nov 2024. https://github.com/WebAssembly/js-promise-integration/blob/main/proposals/js-promise-integration/Overview.md (cited on page 71).

WHATWG [2024a]. *Document Object Model (DOM)*. Web Hypertext Application Technology Working Group, 17 Jun 2024. https://dom.spec.whatwg.org/ (cited on page 12).

WHATWG [2024b]. *Fetch - Living Standard*. 06 Jun 2024. https://fetch.spec.whatwg.org/ (cited on page 14).

WHATWG [2024c]. *File System - Living Standard*. Web Hypertext Application Technology Working Group, 24 Jan 2024. https://fs.spec.whatwg.org/ (cited on page 19).

WHATWG [2024d]. *HTML - Living Standard*. Web Hypertext Application Technology Working Group, 10 Jun 2024. https://html.spec.whatwg.org/ (cited on pages 13, 15, 23–24).

WHATWG [2024e]. *Web Workers. Chapter 10 of HTML Living Standard*. Web Hypertext Application Technology Working Group, 29 Aug 2024. https://html.spec.whatwg.org/#toc-workers (cited on page 37).

WHATWG [2024f]. *WHATWG Standards*. Web Hypertext Application Technology Working Group. https://spec.whatwg.org/ (cited on page 12).

WICG [2024]. *File System Access*. Web Incubator Community Group, 20 Mar 2024. https://wicg.github.io/file-system-access/ (cited on page 21).

Wirfs-Brock, Allen and Brendan Eich [2020]. *JavaScript: The first 20 years*. Proc. ACM on Programming Languages 4.HOPL (12 Jun 2020). doi:10.1145/3386327. https://wirfs-brock.com/allen/jshopl.pdf (cited on pages 5, 11).

yarn [2024a]. *Yarn*. 11 Jun 2024. https://yarnpkg.com/ (cited on page 75).

yarn [2024b]. *Yarn Workspaces*. 19 Nov 2024. https://yarnpkg.com/features/workspaces (cited on page 75).

You, Evan [2024]. *Vue 2*. 29 Oct 2024. https://github.com/vuejs/vue (cited on page 49).

Zakai, Alon [2019]. *Pause and Resume WebAssembly with Binaryen's Asyncify*. 16 Jul 2019. https://kripken.github.io/blog/wasm/2019/07/16/asyncify.html (cited on page 57).

Zakai, Alon [2022]. *Asyncify in Pracitce*. Jan 2022. https://kripken.github.io/talks/2022/asyncify.html#/ (cited on page 57).