AMP-JS: A LIGHTWEIGHT VIEW-LAYER LIBRARY FOR BUILDING SCALABLE WEB APPLICATIONS

Project Report Submitted

In Partial Fulfillment of the Requirements

For the Degree Of

BACHELOR OF ENGINEERING

IN

COMPUTER SCIENCE AND ENGINEERING

Submitted By

Mohammed Ozair Khan (1604-16-733-086)

Arish Rahil Shah (1604-16-733-088)

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

MUFFAKHAM JAH COLLEGE OF ENGINEERING & TECHNOLOGY

(Affiliated to Osmania University)

Mount Pleasant, 8-2-249, Road No. 3, Banjara Hills, Hyderabad-34

2020

CERTIFICATE

This is to certify that the project dissertation titled "**AMP-JS: A Lightweight View-Layer Library for Building Scalable Web Applications**" being submitted by

1. Mohammed Ozair Khan (1604-16-733-086)
2. Arish Rahil Shah (1604-16-733-088)

in Partial Fulfillment of the requirements for the award of the degree Of **Bachelor of Engineering** in **Computer Science and Engineering**, **Muffakham Jah College of Engineering and Technology, Hyderabad** for the academic year 2019-20 is the bonafide work carried out by them. The results embodied in this report have not been submitted to any other University or Institute for the award of any degree or diploma.

Signatures:

Internal Project Guide
(Dr. Krishna Keerthi Chennam)

Head CSED
(Dr. A. A. Moiz Qyser)

External Examiner

DECLARATION

This is to certify that the work reported in the major project entitled "**AMP-JS: A Lightweight View-Layer Library for Building Scalable Web Applications**" is a record of the bonafide work done by us in the Department of Computer Science and Engineering, Muffakham Jah College of Engineering and Technology, Osmania University. The results embodied in this report are based on the project work done entirely by us and not copied from any other source.

1. Mohammed Ozair Khan (1604-16-733-086) _____
2. Arish Rahil Shah (1604-16-733-088) _____

ACKNOWLEDGEMENTS

We are very thankful to our project guide, Dr. Krishna Keerthi Chennam, Assistant Professor, Department of Computer Science Engineering, MJCET, Hyderabad for her invaluable encouragement, comments and suggestions throughout the course of the project. Her insights and instructions have benefited its every aspect.

We would like to thank Dr. A. A. Moiz Qyser, Professor and Head of Department, Computer Science Engineering, MJCET for providing necessary information and guidance for the completion of our project work.

A special thanks to Mr. Akbar Hashmi, Assistant Professor for teaching us Web Programming in third year; much of our learning outcomes are directly used in this project.

We would like to express our sincere gratitude to the Open Source Community for the software and tools without which the development would have been impossible.

ABSTRACT

JavaScript has been synonymous with building modern web applications for the past twenty years. Event-driven architecture has predominantly been followed to achieve native app-like experience on the browser. The problem with this architecture is scalability. Increase in the size of the application is directly proportional to the amount of code written. Frameworks built on the top of JavaScript have solved a lot of problems. But they are complex and have a substantial cost on the produced size of the application due to their build tooling, dependencies, and polyfills. Amp.js provides a declarative approach for creating data-driven applications through separation of concerns. It uses HTML Templates with Tagged Template Literals to build reactive User-Interfaces. The View layer is bound efficiently to the Modal of the system creating a two-way data binding for efficiently updating the DOM.

Keywords: Library, DOM, Templates, Bundle size, User Interface.

CONTENTS

List of Figures

List of Tables

1 INTRODUCTION

In 2020, the World Wide Web is ubiquitous with over a billion websites accessible from billions of Web-connected devices. Each of those devices runs a Web browser or a similar program which can process and display pages from those sites. Most of such pages embed, or load source written in the JavaScript programming language[27]. According to the Stack Overflow Developer Survey 2019[22], the JavaScript programming language is used by over 67.8% of programmers all around the world, making it the most widely used programming language. The language has evolved tremendously over the period of 1995-2020, especially in the domain of front-end web development. Although, creation of a hassle-free system for generating user interfaces has always been a difficult task, libraries built on the top of JavaScript have solved a lot of them.

1.1 History of JavaScript Libraries

The release of jQuery[8] in August 2006 with its "write less, do more" philosophy abstracted the tasks of DOM traversal, animations, and ajax under its easy-to-use API. This was soon met with the challenge of scalability as event-driven architecture made code disarrange itself quickly. Google solved this problem by adopting an MVC-architecture design pattern in its framework AngularJS[17] which was released in October 2010 where model defined the data to be displayed inside views. Its popularity was lost due to the size of the library itself as browsers had to download large amount of JavaScript code, parse, and then execute it. Since internet wasn't as fast and browsers weren't as powerful at the time, many large-scale websites faced issues. In addition, the dependency injection system introduced different entities which were at their core, same.

In 2013, Facebook released React[15] along with which the concept of Virtual DOM was introduced. Virtual DOM maintains a tree structure of the actual DOM in the memory, which performs efficient diff computations on the nodes. It enabled re-construction of the DOM faster and pushed only those changes to the DOM that had occurred. Many other client-side JavaScript frameworks such as Vue.js[28] and Mithril.js[13] also adopted this strategy. In 2015, Redux[4] library was created and became revolutionary data-flow architecture that was inspired by Facebook's Flux architecture. The benefits and shortcomings of creating applications using vanilla JavaScript, AngularJS and React are discussed in detail later in the report.

1.2 Bridge HTML & JavaScript

The language of the web is HTML. JavaScript on the client-side is used to create and manipulate HTML elements programmatically after they have been generated and parsed by the browser. But HTML is a declarative language whereas JavaScript is an imperative

language. With the release of template tag in the HTML specification and template literal in ES2016, a much more declarative system can be built in JavaScript for creating web applications that requires a reactive user interface. Such a system should be more efficient and performant than Virtual DOM in updating only those parts that change.

1.3 Project Aims

The aim of the project is to design and implement a view-layer library in JavaScript for building user interfaces. The core motivation is to expose the good underlying features of JavaScript and provide a layer of abstraction for its weakness and idioms. The syntax should be designed, as much as possible, with the programmer in mind and not the language itself. It should be *terse* and *declarative*, allowing the programmer to write less while achieving more. It should make use of state-driven interface development architecture. Since the language is evolving, application built using the library should be able to use latest syntax and features without worrying about the browser compatibility.

1.4 Report Structure

The rest of the report covers the theory behind the system to be produced, its design, and evaluation of how well the produced system meets the goals of this project.

Chapter-2: Discusses the background theory behind the project, highlighting related work and system.
Chapter-3: Introduces a high-level design of the system to be produced.
Chapter-4: It Does a requirement analysis of the proposed system.
Chapter-5: Describes in detail the system to be produced.
Chapter-6: Contains the implementation of the system along with sample code.
Chapter-7: Includes the various test cases, the error faced and recovered during testing and the intended results.
Chapter-8: Concludes the report, discussing achievements made, any major problems overcome, where this project fits into the bigger picture, and potential future work.

2 LITERATURE SURVEY

This section gives a brief overview of the technologies used today to build web applications in JavaScript. The core philosophy behind these technologies, their syntax, performance and shortcomings are also discussed.

2.1 React

React is undoubtedly the most widely used library for building interactive user-interfaces. This fact is evident from over 150K stars on GitHub and over 7M weekly downloads. Developed at Facebook and released in 2013, it drives most of the widely used apps, powering Facebook and Instagram among many other applications. React is different from any other framework in that it makes use of components instead of templates or HTML directives[14]. These components are created using JavaScript programming language; hence, instead of compiling templates and then rendering them, React enables the developer to make use of the features and syntax of the language itself. This makes code run much faster as additional stages of parsing and generation are skipped. A component is a small reusable chunk of code that is responsible for one job, usually to render some HTML.

2.1.1 Virtual DOM

React makes use of Virtual DOM for creating and rendering elements. React defines Virtual DOM as "The Virtual DOM (VDOM) is a programming concept where and ideal, or "virtual", representation of a UI is kept in memory and synced with the "real" DOM by a library such as ReactDOM. This process is called reconciliation"[21]. This is the approach behind the declarative API of React.

The reconciliation process consists of VDOM diffing where on every subsequent render, a new VDOM tree is generated. This newly generated tree is diffed against the previous one which determines the nodes in the real DOM that needs to be updated using browser-level APIs by the React to match the new VDOM tree. This process is followed repeatedly for every update to the DOM. This may seem tedious at first, handling large amounts of objects is particularly fast in JavaScript. As we learn from Pete Hunt's talk 'Rethinking Best Practices' at JSConfEU 2013, "This (Virtual DOM) is actually extremely fast, primarily because most DOM operations tend to be slow. There's been a lot of performance work on the DOM, but most DOM operations tend to drop frames." The following figure illustrates the working of VDOM and how the changes are reflected on the real DOM.

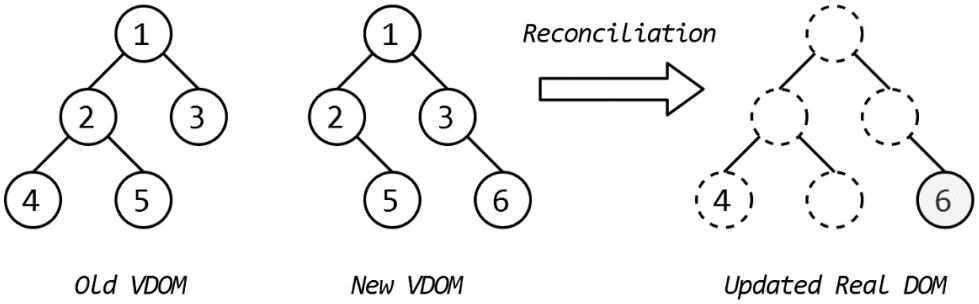

Figure 2.1: Virtual DOM

2.1.2 JSX

JSX (stands for *JavsScript XML*) is an extension of JavaScript that serves as an HTML equivalent in React[20]. React uses Babel's plugin `preset-react` that compiles JSX to nested `React.createElement()` function calls passing attributes to JSX as parameter to function. In React, the JSX elements can be treated as JavaScript expressions, or they can be saved as variable, passed to function or stored in object or array. Each JSX expression must have exactly one outermost element which is usually wrapped in `<div>`. Curly braces mark the start and end of JavaScript expression in JSX.

Code Snippet 1 JSX Compilation by Babel

```
// Input
const App = (props) => {
  return (
    <div className="greet">
      <h1>Hello {props.name}!</h1>
    </div>
  );
};
ReactDOM.render(<App />, document.body);

// Output
const App = (props) => {
  return React.createElement("div", {
    className: "greet"
  }, React.createElement("h1", null, "Hello ", props.name, "!"));
};
ReactDOM.render(React.createElement(App, null), document.body);
```

`ReactDOM`'s render method is responsible for mounting the component 'App' on the body of the HTML document. `React.createElement()` method builds VDOM nodes.

2.1.3 Adaptable React Design Patterns

- **JavaScript:** React uses the language itself rather than templating engine which enables the programmers to use the common syntax of the language over an abstraction. Babel makes it easier to write code in the latest version of JavaScript (`ESNext`) without worrying about browser support.
- **Components:** React is component-based. Encapsulated components are built that manage their own state, then compose together to make complex UIs. These components are like HTML tags, although they are understood by only by React. Since these components are written in JavaScript instead of templates, rich data can easily be passed through the app keeping state out of the DOM. Nested components can be created syncing state between them through *props*.
- **Focus on UI:** Instead of becoming a framework, React focuses solely on developing UIs. Community maintained projects such as React-Router and Redux can be added progressively as per the requirement.

2.2 Web Components

Web components[6] are a set of platform APIs that allows creation of new custom, reusable, encapsulated tags in HTML to be used in web pages and web apps. Custom components and widgets built on Web Components standards work across all modern browsers and can be used with any JavaScript library or framework that works with HTML. Web components aims to solve the problems of code reusability and writing complex HTML (and associated style and script).

2.2.1 Custom Elements

The HTML Specification defines Custom Elements as "a way for authors to build their own fully-featured DOM elements. Although authors could always use non-standard elements in their documents, with application-specific behaviour added after the fact by scripting or similar, such elements have historically been non-conforming and not very functional. By defining a custom element, authors can inform the parser how to properly construct an element and how elements of that class should react to changes"[1]. Custom Elements are used to declare and reuse elements that are actual HTML tags. Associated with these tags are properties which can be observed and efficiently update the content inside these tags. It has the ability to respond to certain occurrences:

- When defined, its constructor is run without any arguments.
- When being inserted in the DOM, its `connectedCallback()` is called without any arguments.
- When being removed from the DOM, its `disconnnectedCallback()` is called without any arguments.
- When any of its attributes are changed, appended, removed or replaced on the DOM, its `attributeChangedCallback()` is called with attribute's name, old value, new value and namespace as arguments.

Code Snippet 2 Custom Elements definition

```
class FlagIcon extends HTMLElement {
  constructor() {
    super();
    this._countryCode = null;
  }

  static get observedAttributes() { return ["country"]; }

  attributeChangedCallback(name, oldValue, newValue) {
    // name will always be "country" due to observedAttributes
    this._countryCode = newVal;
    this._updateRendering();
  }
  connectedCallback() {
    this._updateRendering();
  }
  get country() { return this._countryCode; }
  set country(v) { this.setAttribute("country", v); }

  _updateRendering() {
    // update the content of element
  }
}
customElements.define("flag-icon", FlagIcon);

/* The above defined custom element can be consumed in HTML as such */
<flag-icon country="IN"></flag-icon>
```

2.2.2 Shadow DOM

An important aspect of web components is encapsulation— being able to keep the markup, structure, style and behaviour hidden and separated from other code from the page so that different parts of the page do not clash. The Shadow DOM[2] provides a way to attach a hidden separate part of DOM to an element. It allows hidden DOM trees to be attached to elements in the regular DOM tree— this shadow DOM tree starts with shadow root, underneath which can be attached to any elements, in the same way as normal DOM. The contents of a shadow DOM are isolated and cannot be accessed by querying the document. All the CSS written inside the scope of shadow DOM are not leaked outside and the page styles don't bleed in. Hence, a truly custom, isolated and self-contained HTML elements can be generated using them.

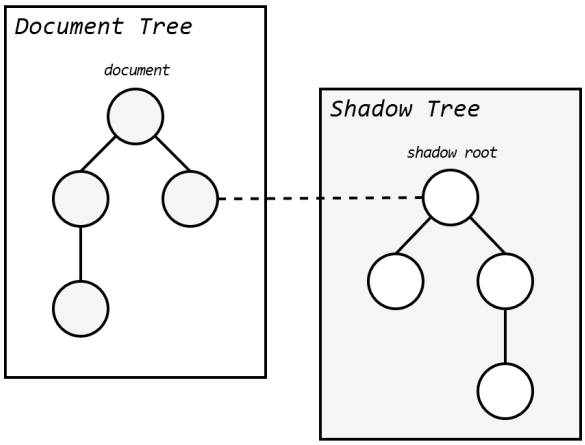

Figure 2.2: Shadow DOM tree

2.2.3 Template Element

In order to use some markup structure repeatedly on a web page, it is better to use some form of template rather than repeating the same structure over and over again. This was possible before but is made a lot easier with the `<template>` element[3]. This element and its content are not rendered in the DOM but can still be referenced using JavaScript. Template elements are mostly used with web components in order to generate custom elements faster as their content can be cloned and used as content of the element. Template element is now supported across all major browsers.

2.2.4 Shortcomings of Web Components

- **Constraints:** In his *criticism of Web Components*, Michael Haufe explains that, "custom CSS pseudo selectors cannot be used with web components, they don't work seamlessly with native elements and their associated APIs and non-trivial elements have to extend

`HTMLElement`"[12]. Additionally, web components have to be defined with ES2015 class which means they cannot be transpiled to give more people the enhanced experience.

- **JavaScript required:** Web Components simply don't work without JavaScript enabled. If that matters, a lot of fallback content needs to be created for graceful degradation of each element, or for each page. Hence, more amount of code is needed to be written for handling conditions when the initial code doesn't work.
- **Polyfills required:** In order to have browser-support and consistent rendering across all browsers, polyfills are needed[11]. Theses polyfills are potentially 100kb+ in size based upon the support requirements. They also suffer being slowed on scale and are deprecated quickly.

2.2.5 Adaptable Web Components Design Patterns

- **HTML Elements:** Instead of working like React and putting abstraction on JavaScript, web components generate HTML Elements that can be created and dropped into existing HTML files. This makes them more versatile as they can even be used with different libraries and frameworks without conflicts.
- **`<template>` Element:** Web Components make use of the Content Template element available in HTML to create custom elements. Since the content inside this element is only parsed and not rendered, it can be instantiated subsequently during runtime using JavaScript.

2.3 Event-driven vs State-driven Architecture

Every single web application that is developed today follows one of the two architectures: event-driven or state-driven (also called data-driven). The former has been in usage since the development of JavaScript itself but it soon met with challenges while the latter is popularized by modern frameworks.

2.3.1 Event-driven Architecture

Libraries that were used before React helped building the user interface applying an event-driven architecture. In this architecture, the UI is depended upon various browser events such as clicks, double-clicks and scrolls which were handled by the browser. User-defined custom events are also supported in their latest versions. Whenever an event occurs, the browser is told exactly what to do in a procedural or imperative manner. JavaScript's implementation of event-driven application is exemplified in Code Snippet 3 and illustrated in Figure 3.

Code Snippet 3 Example of Event-driven architecture in jQuery

```
<body>
  <input placeholder="Enter your name" />
  <p>Hello, <span class="name"></span>!</p>
  <script>
    $(document).ready(function() {
      $("input").on("input", function(event) {
        $(".name").text(event.target.value);
      });
    });
  </script>
</body>
```

The `on()` method attaches the event `"input"` to the textbox. This event is triggered whenever text is entered through a text input device like keyboard. As a result, the callback function is executed which sets the text content of element with `class` as `"name"` to the textbox value. This value is provided through the `Event` object which is generated and bubbled automatically until being handled.

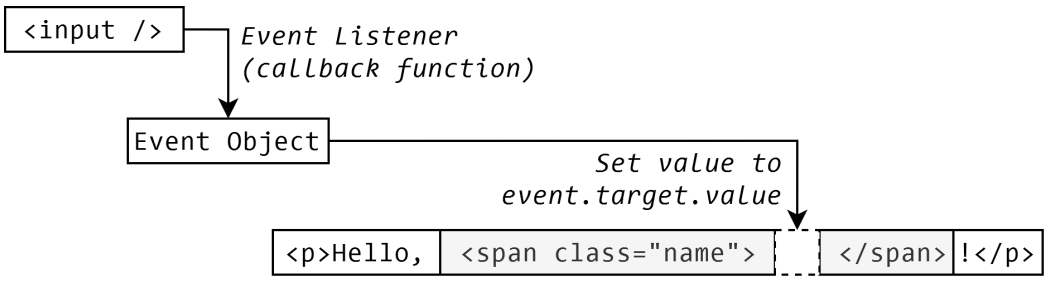

Figure 2.3: Event-driven architecture in jQuery

2.3.2 State-driven Architecture

In contrast, the state-driven architecture has user interface being dependent upon the state/-data. The change in state is informed to the framework which in turn communicates with the browser. This architecture has been widely popularized by React. Instead on being dependent upon events, UIs could now be made as a pure function of state. The challenging part to implement state-driven architecture is detecting *change*. Different libraries use different approach, but *mutation* in the state is a recurring pattern in all of them. Implementation of a state-driven application in React is exemplified in Code Snippet 4 and illustrated in Figure 4.

Code Snippet 4 Example of State-driven architecture in React

```
import React from "react";
import ReactDOM from "react-dom";

function Greeting() {
  const [name, setName] = useState("stranger");
  const handleInput = (event) => setName(event.target.value);
  return (
    <div>
      <input onInput={handleInput} />
      <p>Hello, {name}!</p>
    </div>
  );
}

ReactDOM.render(
  <Greeting />,
  document.body
);
```

`useState()` hook [added in React 16.8] is used to manage the state `name` in our application. The magic of React is hidden behind its `render` function. This method is responsible for construction and reconstruction of the virtual DOM, performing diffing, and communicating with the browser to efficiently update the real DOM.

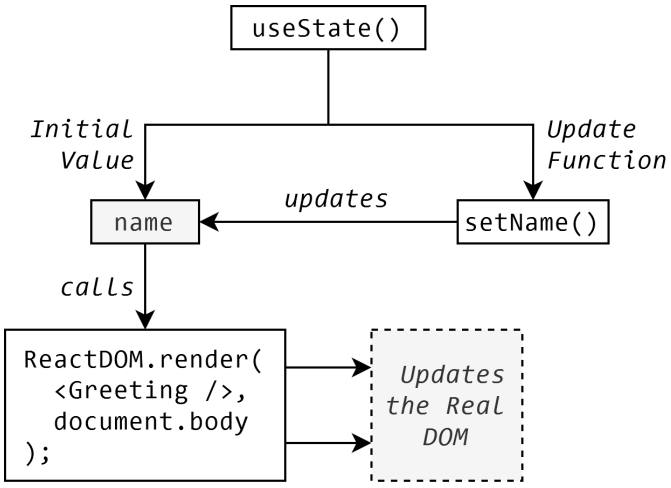

Figure 2.4: State-driven architecture in React

2.3.3 Scalability Differences

For a small-scale application that is composed of a large amount of static markups and stylesheets, events are a better approach. Once the application scales and grows larger in size, shifting to state for managing interfaces is easier. Scaling event-driven interfaces requires you to declare more event listeners, their respective handlers and sync data between them. Whereas scaling state-driven interfaces only requires increase in the state, rest of the code is generally left untouched. As per the current trend of native app-like experience on the web, state-driven approach is popular as it allows the creation of *Single Page Application* very easily, and is easier to understand than its counterpart.

3 SYSTEM ANALYSIS

3.1 Existing Systems

DOM manipulation is the most important aspect of the modern, interactive web. Unfortunately, it is a lot slower than most JavaScript operations. This slowness is made worse by the fact that frameworks update the DOM much more than they have to. To address this problem, people at React have popularized *virtual DOM*.

In React, for every DOM object, a corresponding "virtual DOM object" exists. This object is a representation of the DOM object, like a lightweight copy. The virtual DOM object has the same properties as a real DOM object, but it lacks the real thing's power to change what's on screen. Manipulating the DOM is slow, but manipulating the virtual DOM object is much faster as nothing gets drawn on screen. When a JSX element is rendered, every single virtual DOM object gets updated. Once this is done, React compares the virtual DOM with a virtual DOM *snapshot* that is taken right before the update. By comparing them, React figures out exactly which virtual DOM objects have changed. This process is called "diffing". Once React knows which virtual DOM objects have changed, then React updates those objects, and *only those objects*, on the real DOM.

Shadow DOM is another DOM feature that allows hidden DOM trees to be attached to elements in regular DOM tree. They allow components to have their own "shadow" DOM trees, that can't be accidentally accessed from the main document. Shadow DOM has own stylesheets, style rules from outer DOM don't get applied. This allows creation of a component that can only be referenced by itself and is kept hidden from the document even though it exists inside it.

3.2 Problems with Existing System

The Virtual DOM is usually fast enough, but with certain cavets. The original promise of React was that the whole app could be re-rendered multiple times on every state change without causing performance issues. If it was the case, there would have been no need for optimizations like `shouldComponentUpdate` and `useMemo`. React v16.0 introduced React Fiber that allowed the update to be broken down into chunks. This meant that the update didn't block the main thread for long period of times, though it didn't reduce the amount of work or the time an update takes. As Rich Harris mentions in his article *Virtual DOM is pure overhead*, "It's important to understand that virtual DOM isn't a feature. It's a means to an end, the end being declarative, state-driven UI development. Virtual DOM is valuable because it allows you to build apps without thinking about state transitions, with performance that is generally good enough. That means less buggy code, and more time spent on creative tasks instead of tedious ones"[10].

In addition, shadow DOM and its encapsulation feature isn't free from problems. They break accessibility, i.e., applications built with shadow DOM makes it difficult or impossible to be used by people with disabilities. Progressive enhancement (or Server-Side Rendering) is also broken since it requires the *Browser Object Model* (or BOM) for its creation which is unavailable on the server. They don't support SVGs hence, charts and illustrations cannot be created using web components and shadow DOM[11].

3.3 Proposed System

The proposed system is an HTML templating library. These templates must be written in JavaScript by mixing static HTML strings and dynamic JavaScript values using template literals. This enables a functional / UI-as-a-data programming model, fast initial rendering, and fast updates that minimally updates the DOM when state changes. The key that enables this is separating static parts of templates from the dynamic parts with template literals, and never traversing or updating the static parts after the initial render.

Amp-js should be comparable in many ways to virtual-DOM approaches, but work without storing a separate representation of DOM in the memory, or computing DOM diffs like virtual-DOM libraries must do.

- Where VDOMs work at the level of DOM representation of a UI, amp-js should work at the *value* level.
- Amp-js UIs should be values, like VDOM, but instead of each DOM node being represented as a single value, a reference to a template should represent the DOM structure.
- The tree of values that is used to create DOM is more sparse than the associated DOM tree, so traversing it when necessary is faster.
- Amp-js does not need to track changes.

3.4 Feasibility Study

3.4.1 Economic Feasibility

Economic feasibility is the cost-benefit analysis of the project, which assesses whether it is possible to implement it. If the benefits cut cost then the system is implemented, otherwise alternatives are proposed. Since amp-js is a library that will help build applications, economic feasibility of amp-js is dependent directly on the application. Although, it is safe to say that amp-js is not going to add an extra overhead cost to the application and is going to be open-sourced so that it can be reused, modified and published without any permission.

3.4.2 Technical Feasibility

In recent years, developing a web application is a comparatively easier task. But this task is quickly met with challenges as the size of the application increases. Hence, using a framework or a library is generally preferred over writing lower-level modules by hand. But, a complex workflow setup is required in order to use these frameworks. Amp-js should be created in such a way that would allow developers to just drop-in on an existing application and *amplify* them without additional configuration.

3.4.3 Operational Feasibility

Operational Feasibility is the assessment of to what degree the project would benefit an organisation and how efficiently the system would work. As stated in the proposed system, amp-js makes use of the latest JavaScript features and allows the transpiler to take care of the browser support. This helps the developer using this library to develop the application based upon their target audience instead of the library. The modules that are used in amp-js are *tree shaken*, or the dead code has already been eliminated making the size of the library reasonable small. Using the library is going to add only a few kilobytes of additional scripts to the application bundle size, which can then be further minimized by using `gzip` or `brotli` compressions. Code Splitting and dynamic imports can also be used by the developer which is supported by amp-js to reduce the size of the initial script which can make the first paint faster.

4 SOFTWARE REQUIREMENT SPECIFICATION

4.1 Introduction

This chapter gives a scope description and overview of everything included in the SRS section. Also, the purpose for this project is described and a list of technologies and dependencies is provided.

4.1.1 Purpose

The purpose of this project is to build a client-side templateing engine that can be used to render the user interface dynamically. This task should be comparable in perfomance and in most cases better than the systems existing today to build modern web applications. Moreover, it should provide an interface for generating custom reusable components that have encapsulated data and properties but can be rendered efficiently to *amplify* the HTML elements. The behaviour should mimic Web Components without the requirement of polyfills and browser support dependencies.

4.1.2 Scope

Amp-js is a JavaScript library that allows us to build user interfaces as a pure function of state. It uses the syntax and features of the latest version of ECMAScript and supports their usage for application development too. It will distribute as an ES6 module since all the major browsers support the syntax. For a dedicated workflow, installing it as a dependency is a better approach. The system should be available as open source that can be changed as per the developer's requirement. Applications that have already been developed should also be able to adopt amp-js incrementally.

4.2 Overall Description

This section will give an overview of the whole system. The system will be explained in its context to show how the system interacts with other systems and introduces it's basic functionality.

4.2.1 Project perspective

The system consists of two parts: generating a DOM representation and rendering them on the screen. The representation consists of the static and dynamic parts of the template separated alongwith a description can easily be cached and re-rendered. The rendering takes the DOM representation and a container as parameters, generates and appends real

DOM on the container. This process can be repeated multiple times, only updating the part that changes every time.

This method should be more efficient than React's Virtual DOM and Angular's Incremental DOM. Instead of repeating the same process and finding out what's changed, amp-js is going to keep the parts that are going to change separated from the parts that are not. When something does change, only changeable parts needs to be checked and not the whole template. It improves the performance of the system by a greater margin. Since the foundational piece of this system is `<template>` element whose content can be cloned easily, there is no need for a dedicated renderer like React Fiber as initial rendering should be fast and take below 160ms.

4.2.2 Project Functions

Amp-js is a client-side HTML templating library that both boots and updates fast. The size of the library is very small when compared to major frameworks. The API is easy-to-use and is extensible. Since adding JavaScript to a web application increases the load on the browser as it has to be downloaded, parsed and then executed, the size of the library being small is beneficial. This makes the application faster when it boots for the first time as well as when it updates.

The library is going to benefit developers in terms of the developers' experiences too. Most developers want their templates to resemble the final DOM. Since template literals are strings, templates written inside strings can come closer in their syntax to the actual DOM. They can span across multiple lines and have expressions embedded as interpolation strings.

4.3 Specific Requirements

This section contains all system requirements of the project. It gives a detailed description of the system and all its features.

4.3.1 Hardware Requirements

Since the library does not have any designated hardware, there is no hardware requirements. The parsing and execution of the code is dependent on the browser. The application using it however, can have its own requirements that include hardware interface.

4.3.2 Software Requirements

Software requirements refers to the tools used for developing the project. Since the project is a JavaScript library, the only major requirements is the JavaScript programming language. In addition, a web browser is needed to actually make use of the library.

JavaScript

JavaScript, often abbreviated as JS, is a programming language that conforms to the ECMAScript specification. It is used as a client-side scripting language, which means the source code is processed by the client's web browser rather than on the web server. Thus, JavaScript functions can run after a webpage that has loaded without communicating with the server. Like server-side scripting languages, such as PHP and ASP, JavaScript code can be inserted anywhere within the HTML of a webpage. However, only the output of server-side code is displayed in the HTML, while JavaScript code remains fully visible in the source of the webpage.

Web Browser

Web browsers have in-built JavaScript engines that allow them to download and execute JavaScript. The Browser Object Model (BOM) allows JavaScript to interact with the browser. It consists of the objects navigator, history API, screen, location and document which are children of the global object `window`. In the `document` node is the **DOM** (Document Object Model), which represents the contents of the page. Since the project is developed to interact with the DOM, it requires a browser to do so. All of the major browsers including Explorer, Firefox, Netscape, and Safari have the DOM built in and hence should be able to execute both the development and production versions of amp-js.

Git

Git is designed for coordinating work among programmers, but it can be used to track changes in any set of files[5]. Git is used in the development of our project as it allows us to code incrementally, testing each increment and committing them if the tests pass. If the increment does not meet the project requirements, the commit can be reverted. For changing an existing code, a new branch can be created, tested and then merged to *master* upon passing them.

Node.js

Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine[7]. Node.js is primarily used for its package manager, *npm* (Node Package Manager). It allows us to install libraries and develop our project using them.

4.4 Development Dependencies

Development dependencies, or `devDependencies` are packages that are consumed by requiring them in files or run as binaries, during the development phase. These are packages that are only necessary during development and not necessary for the production build. The user of this library will not have to install them while using it.

NPM (Node Package Manager)

npm is a command-line utility that aids in package installation, version management, and dependency management[16]. A node project is initialized by generating a `package.json` file. Now dependecies can be added to this project by using the command `npm install`. This enables us to install the packages already developed for handling lower level tasks and build our project on top of them. They serve as foundational pieces for our project. `package.json` file also contains meta information about the project, scripts that can be run using the `npm run` command and the entry point of the project through its bundled file name.

Rollup.js

Rollup is a module bundler that is used to compile small pieces of code separated over multiple files into something larger, more complex, such as a library or application[9]. It uses the new standardized ES6 module sytax of JavaScript which is not supported natively everywhere, but can be used with rollup. An important feature of rollup is *Tree Shaking* or *Dead code elimination*. Rollup statically analyzes the code that is being imported, and excludes anything that isn't actually used. This allows development of new project on top of existing tools and modules without adding extra dependencies or bloating its size.

For example, with CommonJS syntax, *the entire tool or library must be imported*.

```
// import the entire utils object with CommonJS
const utils = require( "./utils" );
const query = "Rollup";

// use the ajax method of the utils object
utils.ajax(`/api/search?q=${query}`).then(handleResponse);
```

With ES modules, instead of importing the whole `utils` object, we can just import the one `ajax` function we need:

```
// import the ajax function with an ES6 import statement
import { ajax } from "./utils";
const query = "Rollup";

// call the ajax function
ajax(`/api/search?q=${query}`).then(handleResponse);
```

Two plugins: `rollup-plugin-terser` and `rollup-plugin-babel` have been used to compile and minify the production version of the library. `rollup.config.json` file is used to specify the bundling configuration.

Babel

Babel docs defines itself as, "...a toolchain that is mainly used to convert ECMAScript 2015+ code into a backwards compatible version of JavaScript in current and older browsers or environments"[18]. It is a transcompiler that is used to convert JavaScript written in the syntax adhering to ECMAScript 2015 or above into versions of JavaScript that can be run by older JavaScript Engines. Hence, it allows the usage of newest features of the JavaScript programming language without worrying about browser support.

For example,

```
// Babel Input: ES2015 arrow function
[1, 2, 3].map((n) => n + 1);
// Babel Ouput: ES5 equivalent
[1, 2, 3].map(function(n) { return n + 1; });
```

`@babel/core` and `@babel/preset-env` plugins have been used for syntax transformations and transpilation to target environment. The configuration for babel is defined in a `.babelrc` file.

Jest

Jest is a comprehensive JavaScript testing library[26]. It is faster than many other test runners and works out of the box without any configuration. It is primarily used for **snapshot testing** which captures snapshot of large objects to simplify testing and to analyze how they change over time. By installing it as a development dependency, the tests can be started by running `npm test`.

ESLint

ESLint is a tool for identifying and reporting on patterns found in ECMAScript/JavaScript code, with the goal of making code more consistent and avoiding bugs[29]. It allows us to write better code by following standards setup by the JavaScript programming language itself. Configurations in ESLint are overwritable and defined in `.eslintrc` file. Prettier is used alongside ESLint which auto-formats the code based upon its configuration.

4.5 Modules

The proposed system has two modules: an efficient templating engine and a custom resuable components interface.

Efficient Templating Engine

The first module comprises of developing an efficient templating engine that is going to make use of JavaScript tagged template literals and HTML `<template>` element. It allows

the creation of UI following a state-driven architecture, i.e., the user interface is a pure function of the state. Whenever the state changes, the UI updates incrementally to reflect those changes on the DOM. This re-rendering process is efficient and allows multiple subsequent renders to be possible without causing a memory leak.

The syntax resembles the final DOM that will be rendered, much like JSX without the additional syntactical sugar. Beside this, event Listeners, properties, and attributes can be added directly to the template itself. Templates can be generated directly or a function that returns the template to. The former creates a static template whereas the latter allows state to be passed as a parameter to the function. In addition, templates amp-js support looping, conditional rendering and nesting.

Custom Reusable Components

The second module of the project comprises of an interface that allows us to extend basic HTML elements encapsulating reusable code. At a high level, custom elements follow the design patterns of web components but using amp-js rendering. These elements can be reused, have their state encapsulated to the scope and efficiently re-render by using JavaScript `Proxy`. Each component may or may not have data associated with it. The data can even be shared across multiple components through `props`. Updating the data from a parent component is going to update all the components dependent on it. Functions can also be used to pass data back to the parent, thus achieving a two-way data flow.

The custom components should have a specific naming convention. It must **contain a hyphen**. So `<my-element>`, `<amp-root>`, and `<custom-tag>` are all valid names, while `<element>` and `<x_tags>` are not. With this requirement, the browser's HTML parser can distinguish custom components from regular elements. It also ensures forward compatibility when new tags are added to HTML.

4.6 Release Versions

4.6.1 Development Version

The development version of the library should be used while developing the application as it involves helpful console warnings to provide information about an improper usage. This is achieved through *JavaScript Source Maps*.

4.6.2 Production Version

The production version is optimized for size and speed and should be used when the application is ready to be shipped. The code is minified and does not include console warnings.

5 SYSTEM DESIGN

5.1 Core Sections

Amp-js relies on two key platform features for its performance and ergonomics: JavaScript Tagged Template Literals and the HTML `<template>` Element. So it's helpful to understand them first.

5.1.1 Tagged Template Literals

A JavaScript template literal is a string literal that can have JavaScript expressions embedded in it[25].

```
`My name is ${name}`;
```

The literal uses backticks instead of quotes, and can span multiple lines. The part inside `${}` can be any JavaScript expressions. A *tagged* template literal is prefixed with a special template tag function.

```
let name = "Ben";
tag`My name is ${name}.`;
```

Tags are functions that take the literal strings of the template and values of the embedded expressions, and return a new value[24]. This can be any kind of value, not just strings. Amp-js returns an object representing the template, called a `TemplateResult`.
A tag function can return any object; it does not have to return a string. The first argument to the tag is a special array of strings. It retains its identity across multiple evaluations of the tagged literal.

Code Snippet 5 Tagging the Template Literals

```
// This code:
const tag = (strings, ...values) => {
  console.log("strings: ", strings);
  console.log("values: ", values);
};
let text = "abc";
tag`<div>${text}</div>`;

// prints:
strings: ["<div>", "</div>", raw: Array(2)]
values: ["abc"]
```

The key features of template tags which amp-js utilizes to make updates faster is that the object holding the literal strings of the template is *exactly* the same for every call to the

tag for a particular template. This means that the strings can be used as a key into a cache so that amp-js can do the template presentation just once, the first time it renders a template, and updates skip that work.

5.1.2 HTML Template Elements

A `<template>` element[3] contains an inert fragment of DOM. Inside the template's content scripts don't run, images don't load, custom elements aren't upgraded, and so on. However, the contents can be efficiently cloned. Template elements are normally used to tell the HTML parser that a section of the document must not be instantiated when parsed, and will be managed by code at a later time; but template elements can also be created imperatively with `createElement` and `innerHTML`.

Amp-js should create HTML `<template>` elements from the tagged template literals, and then clone them to create new DOM.

5.1.3 JavaScript Modules

Until modules werearrived, browsers did not have a standard way to import code from code, so user-land module loaders or bundlers were required. Since there was no standard, competing formats got multiplied. Often libraries were published in a number of formats to support users of different tools, but this causes problem when a common library is depended on by many other intermediate libraries. If some of those intermediate libraries load format A, and others load format B, and yet others load format C, then multiple copies are loaded, causing bloat, performance slowdowns, and sometimes hard-to-find bugs.

The only true solution is to have one canonical version of a library that all other libraries import. Since modules support is rolling out to browsers now, and modules are very well supported by tools, it makes sense for that format to be modules[23].

The browser current only accepts modules specified using a full or relative path (a path that starts with `/`, `./`, or `../`). For ease of authoring, many developers prefer to import modules by name (also known as node-style module specifiers). Since this isn't currently supported in the browser, tools that can transform these specifiers into browser-ready paths are used.

```
// Node-style module import:
import { html, render } from "amp";
```

```
// Browser-ready module import:
import { html, render } from "../node-modules/amp/amp.js";
```

5.2 Template Structure

Amp-js templates must be well-formed HTML, and bindings can only occur in certain places. The templates are parsed by the browser's built-in HTML parser before any values are interpolated.

No warnings. Most cases of malformed templates are not detectable by amp-js, so you won't see any warnings—just templates that don't behave as you expect—so take extra care to structure templates properly.

Follow these rules for well-formed template:

- Templates must be well-formed HTML when all expressions are replaced by empty values.
- Bindings ***can only occur*** in attribute-value and text-content positions.

```
<!-- attribute value -->
<div label=${label}></div>

<!-- text content -->
<div>${textContent}</div>
```

- Expressions ***cannot*** appear where tag or attribute names would be seen.

```
<!-- ERROR -->
<${tagName}></${tagName}>

<!-- ERROR -->
<div ${attrName}=true></div>
```

- Templates can have multiple top-level elements and text.
- Templates ***should not contain*** unclosed elements—they will be closed by the HTML parser.

```
// HTML parser closes this div after "Some text"
const template1 = html`<div class="broken-div">Some text`;
// When joined, "more text" does not end up in .broken-div
const template2 = html`${template1} more text. </div>`;
```

5.3 Binding Types

Expressions can occur in text content or in attribute value positions. There are a few types of bindings:

- Text:

```
html`<h1>Hello ${name}</h1>`;
```

  Text bindings can occur anywhere in the text content of an element.

- Attribute:

```
html`<div id=${id}></div>`;
```

- Boolean Attribute:

```
html`<input type="checkbox" ?checked=${checked}>`;
```

- Property:

```
html`<input :value=${value}>`;
```

- Event Listener:

```
html`
  <button
    @click=${(e) => console.log('clicked')}
  >Click Me</button>`;
```

Event Listeners

Event listeners can be functions or objects with a `handleEvent` method. Listeners are passed as both the listener and options arguments to JavaScript's `addEventListener` and `removeEventListener` methods, so that the listener can carry event listener options like `capture`, `passive`, and `once`.

```
const listener = {
  handleEvent(e) {
    console.log('clicked');
  },
  capture: true,
};
html`<button @click=${listener}>Click Me</button>`;
```

5.4 Control Flow with JavaScript

Amp-js has no built-in control-flow constructs. Instead we use normal JavaScript expressions and statements.

Ifs with ternary operators

Ternary expressions are a great way to add inline-conditionals:

```
html`
  ${user.isloggedIn
    ? html`Welcome ${user.name}`
    : html`Please log in`
  }
`;
```

Ifs with if-statements

We can express conditional logic with if statements outside of a template to compute values to use inside of the template:

```
getUserMessage() {
  if (user.isloggedIn) {
    return html`Welcome ${user.name}`;
  } else {
    return html`Please log in`;
  }
}

html`${getUserMessage()}`;
```

Loops with Array.map

To render lists, `Array.map` can be used to transform a list of data into a list of templates:

```
html`
  <ul>
    ${items.map((i) => html`<li>${i}</li>`)}
  </ul>
`;
```

Looping statements

```
const itemTemplates = [];
for (const i of items) {
  itemTemplates.push(html`<li>${i}</li>`);
}
html`<ul>${itemTemplates}</ul>`;
```

5.5 Rendering

Amp-js templates are only a description of the UI. They must the rendered to affect the DOM. While there are a few internal phases to rendering, they are abstracted over two major phases:
Template are most often written as functions that return an amp-js `TemplateResult`:

```
const f = (state) => html`<h1>${state.title}</h1>`;
```

They are invoked to obtain a description of the UI with a particular state:

```
let ui = f(state);
```

Then rendered to a specific container:

```
render(ui, container);
```

`render()` can be called multiple times with results of the same template, but different state, and the DOM will be updated to match the state. There is a lot of difference between an initial render and an update.

5.6 Phases of Template Rendering

Templates go through distinct phases when rendering. Some of the work for rendering happen the first time a template is rendered anywhere on a page, for some the first time a template is rendered to a particular container, and some other when a template instance is updated with new data.

5.6.1 Definition

Templates are defined with the `html` template tag. This tag does very little work - it only captures the current values and a reference to the strings object, and returns this as a `TemplateResult`.

Amp-js syntax is configurable via `TemplateProcessor`s. The specifics of using a `:`, `@`, and `?` prefixes for attribute names to denote property, event and boolean attribute bindings is only how the default TemplateProcessor works. It is important that the syntax of a template is fixed at definition time so that its meaning and behavior don't change if it is rendered by a different library. That is – it's the template author who needs to control the syntax, not the code that renders.
So, `TemplateResult` also contains a reference to the `TemplateProcessor` to be used.
Amp-js ships with one template tag: `html`. This tag is extremely simple, just:

```
export const html = (strings, ...values) =>
   new TemplateResult(strings, values);
```

And the `TemplateResult` constructor is also very simple:

```
constructor(strings, values) {
  this.strings = strings;
  this.values = values;
}
```

This means that evaluating a template expression can be extremely fast. It is only as expensive as the JavaScript expressions in the template, two function calls and an object allocation.

5.6.2 Preparation

When a template is rendered for the very first time on a page, it must be prepared. Preparation creates a `<template>` element that can be cloned to create new instances, and metadata about the expressions in a template so that updatable Parts can be created for the instance.

Step-1: Join the template's string literals with a marker string
This is like a string interpolation that disregards the values of the expressions. For each expression, amp-js does some backtracking on the preceding string literal to determine if the expression is in a text position (i.e., `<p>${}</p>`) or an attribute position (i.e., `<p class=${}></p>`). This marker is a comment node (`<!--$commentMarker-->`) for text positions, and a sentinel string (`{{--$attributeMarker--}}`) for attribute positions.

For attributes with expressions, we append a suffix to the attribute name. This is for special-cased attributes like `style` and `class` and many SVG attributes that are not handled by the browser at a point when they contain marker expressions. (IE and Edge parse the `style` attribute value and discard it if it is invalid CSS for instance).

The template: `<p class=${c}>${text}</p>` is transformed into the HTML string: `<p class={{-amp-}}><!--amp--></p>`. Generating the HTML is now the responsibility of `TemplateResult`.

Step-2: Create a `<template>` Element
The HTML string from step 1 is simply used to set the `innerHTML` of a new `<template>` element. This causes the browser to parse the template HTML. Text-position markers written as comments are parsed into comment nodes.

Because this step does not use any provided values (and because templates are inert), it is safe from XSS vulnerabilities or other malicious input.

Step-3: Create an amp-js `Template` object
The `Template` class does most of the heavy-lifting for preparing a template. It walks the tree of nodes in a `<template>` element finding the markers inserted in step 1. If a marker is found, either in an attribute, as a marker comment node, or in text content, the depth-first

index of the node is recorded, along with the type of expression (`"text"` or `"attribute"`) and the name of the attribute.
Template has to do different works for specific Node types:

Element: Each attribute must be checked for the presence of an expression marker. When we find an attribute that has an expression marker, we remove the attribute and create the expression metadata object that records its location and name.

Text: Usually text-position markers are comment nodes, but inside `<style>` and `<script>` tags the markup that looks like a comment (`<!--amp-->`) will just be inserted as text in the script or style. So we must search for the marker as text. If found we split the Text node and insert a comment marker.

Comment: Comment is usually either an expression marker, or a user-written comment with no expression associated with them. Occasionally a user may write an expression inside a comment. This is especially easy to do with IDEs that help comment out of the code sections and are inline-html aware. As we see a comment like `<!--<div>${text}</div>-->` in the template text, so we must scan comments for marker text.

5.6.3 Creation

The create phase is performed when rendering for the first time to a specific container or Part. A template can be rendered to a container (with `render(result, container)`) or a Part. Templates are rendered to Parts by being used as a value with another template. In this example:

```
const postBodyTemplate = (text) => html`<p>${text}</p>`;
const postTemplate = (post) => html`
  <h1>${post.title}</h1>
  ${postBodyTemplate(post.body)}
`;
```

The `TemplateResult` returned from `postBodyTemplate(post.body)` is rendered to the `NodePart` defined by the expression markers.

When a `TemplateResult` is rendered to a container or Part we get its `Template` via a template factory function that's passed to `render()` via the options argument. A template factory is responsible for creating and caching `Template` objects. This abstraction is necessary because certain extensions to amp-js may need to modify templates or cache based on additional keys. The template factory is what initiates the *prepare* step if a `Template` is not yet in the cache.

Step-1: Create a `TemplateInstance`

`TemplateInstance` is the class that is responsible for the most of the create and update steps. It's given a `Template`, `TemplateProcessor`, and `RenderOptions`.

Step-2: Clone the template

The `<template>` element is cloned by `TemplateInstance#_clone()` (private method `clone()` of class `TemplateInstance`).

Care must be taken to manage cloning and upgrade order. We need the cloned DOM to exactly match the template DOM so that we can find Part locations when we walk the cloned tree. When Custom Elements upgrade, it's possible that they could modify the DOM before we get a chance to walk it, throwing off our part indices.

In general, Custom Elements are barred from modifying their own DOM, and we don't need to worry about their ShadowRoots, except in the case of the Web Components polyfills. So with native Web Components, template contents are cloned with `document.importNode()`, which will cause Custom Elements to upgrade. With polyfilled Web Components, template contents are cloned with `template.content.cloneNode()`, which will not upgrade elements. They are then upgraded later.

Step-3: Create Parts

After cloning, `TemplateInstance#_clone()` walks the cloned contents so that we can associate nodes with expression metadata by depth-first-index. When we get to a node that has an associated expression we call the `TemplateProcessor` function with the node and expression metadata to create a Part. The `TemplateProcessor` will return either a `NodePart`, `AttributePart`, `PropertyPart`, `EventPart` or `BooleanAttributePart` based on expression position and/or attribute name.

When we encounter `<template>` nodes in the tree, we recurse into their content fragments to find any other expressions and create their Parts.

5.6.4 Updation

The update step, which is operational for initial renders as well, is performed in the private method `update()` of `TemplateInstance` class (`TemplateInstance#update()`).

Updates are performed in two phases: setting values, and commit. `update()` simply iterates through each part and value (the parts array and values array are always of the same length) and calls `part.setvalue(v)` for each part. Then it iterates through the parts again and calls `part.commit()`. This two-phase design allows attributes and properties that have multiple parts associated with them to be committed only once.

The heavy-lifting of the update phase is done by the parts themselves, in `setValue()` and `commit()`, and in `AttributeCommitter` and `PropertyCommitter`.

5.7 Component System

Now we have an efficient templating engine, its usage can be extended to create custom components system. It is going to follow the design patterns of Web Components without requiring any polyfills.

5.7.1 What are Components?

Component is one of the most powerful features of amp-js. The component system is an abstraction that allows us to build large-scale applications composed of small, self-contained and often reusable components. If we think about it, almost every type of application interface can be abstracted into a tree of components. It enables an easier maintenance of code when the interface scales.

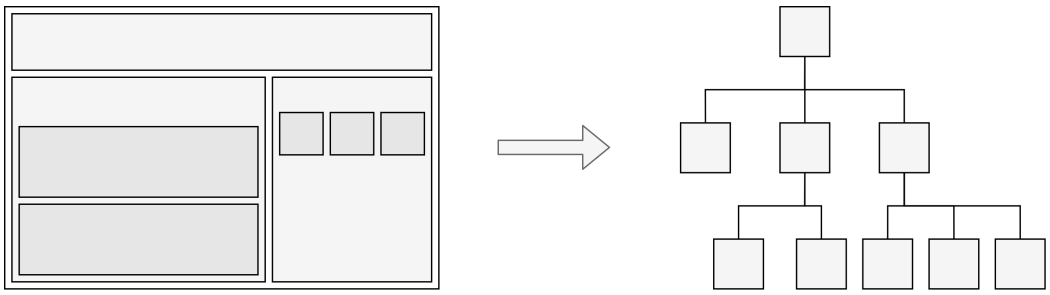

Figure 5.1: Abstraction of interface into components

5.7.2 Using Components

Registration

Before we start using components, it is important to register them. Registration is required to inform amp-js about the components and their definition when used on the interface. This is achieved by using the `Amp.component()` method provided by the default export.

```
const Root = Amp.component("amp-root", {
  // definition...
});
```

Component names should follow **W3C rules** for custom tag-names (all lowercase, must contain a hyphen). This is required so that the browser's HTML parser may understand the difference between regular HTML elements and amp-js component. It also ensures forward compatibility when new tags are added to HTML. For example, `<amp-root>` is a valid component name, whereas `<child>` and `<amp_root>` isn't.

Once registered, the instance's `generate()` method can be called to convert the custom-tag to an amp-js component. The component can now be used directly on the document or inside a parent instance's template as `<amp-root>`. Make sure that the element has been used somewhere in the application before calling its `generate()` method as amp-js renders the instance on an alerady existing tag. Here's the full example:

Code Snippet 6 Creating components in Amp-js

```
// Register the component in a JavaScript file:
const Root = Amp.component("amp-root", {
  template() {
    return html`<div>A custom component!</div>`;
  }
});
Root.generate();

// Use the component in an HTML file:
<amp-root></amp-root>

// Which will render:
<amp-root>
  <div>A custom component!</div>
</amp-root>
```

Note that the component's template does not replace the custom element, which serves as a **mounting point** but it does replace all the previously existing content inside it. Also, the components are provided a `template` function that returns the amp-js DOM representation which is rendered internally.

Component State

Components defined in amp-js are capable of maintaining their own encapsulated state. The state is isolated and is inaccessible outside the component. A component's state is defined inside the `data` option of the definition. This state should be a key-value pair. It can be accessed inside the template using `this` keyword. Internally amp-js **Proxies** the data properties so whenever one of these properties changes, the template function can be executed again and the component can be re-rendered. The `data` is bound to `template`, hence to access the state, "`this`" keyword needs to be used.

```
const Root = Amp.component("amp-root", {
  data: {
    message: "World"
```

```
  },
  template() {
    return html`<h1>Hello ${this.message}!</h1>`;
  }
});
Root.generate();
```

Component Methods

State change causes the component to re-render and update. This mutation should not be done inside the template function as it may lead to infinite renders. Instead methods must be declared inside the `methods` option of the component definition. The state is bound to methods, hence is triggered by changing the state object accessible using `this` keyword.

Methods can also be used as event listeners and amp-js automatically passes the event object as a parameter. Later we will see that amp-js also provides some built-in methods that are executed during an instance lifecycle.

```
const Root = Amp.component("amp-root", {
  data: {
    message: "World"
  },
  methods: {
    changeMessage() {
      this.message = "Amp-js";
    }
  },
  template() {
    return html`
      <h1>Hello ${this.message}!</h1>
      <button @click=${this.changeMessage}></button>
    `;
  }
});
Root.generate();
```

In the above Code snippet, the method `changeMessage` is executed as a callback function after clicking the button. This method changes the `message` property of data and thus causes a re-render updating only the part that has changed.

5.7.3 Nested Components

In amp-js a component can hold one or more components nested inside it. The instance of all the nested components are declared as an array within the `components` option. Upon re-render, amp-js automatically detects the components that have changed from the array and updates them. Their usage is exemplified below:

Define the nested component first:

```
const Child = Amp.component("amp-child", {
  // definition...
});
```

Now this component can be used inside another parent component. Amp-js is informed about this component by adding its instance to the `components` option.

```
const Root = Amp.component("amp-root", {
  // add the child components
  components: [Child],
  // use the child component inside template
  template() {
    return html`
      <div id="root">
        <amp-child></amp-child>
      </div>
    `;
  }
});
Root.generate();
```

Props

Every component instance has its own **isolated scope**. This means that we cannot make reference to the parent data inside child component's template. Date can be passed down to child components using **props**.

A "prop" is a field on a component's state that is expected to be passed down by the parent. The child component needs to declare explicitly the props it expects to receive using the `props` option. While passing props through the template, add a colon (`:`) followed by the name of the prop. For instance, `:foo=${bar}`.

```
const Child = Amp.component("amp-child", {
  // declare the props
  props: ["msg"],
```

```
  // the props can be used inside templates
  template() {
    return html`<span>${this.props.msg}</span>`;
  }
});
```

Then we pass a plain string to it like so:

```
const Root = Amp.component("amp-root", {
  template() {
    return `<amp-child :msg=${"hello"}></amp-child>`
  }
});
```

Props are accessible inside the templates through `props` object. It is noted while passing props, interpolation `${}` is required as it allows amp-js to track them.

Not only strings, any form of JavaScript data can be passed down to child components, including event listeners and Promises. A piece of state can also be passed down to the component. This is highly beneficial for the parent-child communication between components.

5.7.4 Instance Lifecycle

Each amp-js instance goes through a series of initialization steps when it is created - for example, it needs to set up proxies, generate instances for child nodes and create the necessary data bindings. Along the way, it also invokes some **lifecycle hooks**, which gives an opportunity to execute custom logic. The `onmount` hook, for example, is called after the template has been rendered intially on the interface.

```
const Root = Amp.component("amp-root", {
  data: { a: 1 },
  onmount: function() {
    console.log("a is: " + this.a);
  }
});
// -> "a is: 1"
```

There are another hooks which will be called at different stages of instance's lifecycle, for example `oncreate` and `onupdate`. All lifecycle hooks are called with `this` context pointing to the amp-js instance invoking it. There are no controllers, the custom logic for a component has to be split among these lifecycle hooks.
There are three "hooks" provided by Amp-js for adding control logic:

oncreate

Called synchronously after the instance is created. At this stage the instance has finished processing which means the following have been set up: data observation, methods, and event callbacks. However, the rendering phase has not been started yet. So any call to methods would work, but is advised not to change state as the component may behave in an undesired way.

```
oncreate: function() {
  // Code that will run before
  // the view has been rendered
}
```

onmount

Called after the instance has been mounted where the custom tag has been filled with the defined template. At this stage, props passed to the component are also defined. Since this method is only called once, it is best suited for performing asynchronous calls. The function itself can be made `async`.

`onmount` also guarantees that all the child components have also been rendered. If a component itself is a child component, `onmount` may be called multiple times based upon the parent component's template.

```
onmount: function() {
  // Code that will run only after the
  // entire view has been rendered
}
```

This function can also be declared as async which enables the usage of `async-await` for handling Promises in JavaScript.

```
onmount: async function() {
  // This http request is made only once,
  // as soon as the component mounts
  let response = await fetch("https://example.com/api/v1/users");
  let result = await response.json();
  this.users = result.data;
}
```

onupdate

Called after a data change causes the template to udpate and re-render. The component's DOM will have been updated when this hook is called, so DOM dependent operations can

be performed here. This hook guarantees that all child components have also been re-rendered. It is noteworthy that state should not be changed inside the hook, as changing state triggers it. So the component might stuck in an infinite loop of assignments and re-rendering.

```
onupdate: function() {
  // Code that will run only after the
  // entire view has been re-rendered
}
```

5.8 UML Diagrams

5.8.1 Use Case diagram

A use case is a set of scenarios tied together by a common user goal. The use case diagram models the functionality of the system as perceived by outside users, called actors. The purpose of the use case diagram is to list the actors, use cases and show which actors had participated in each use case.

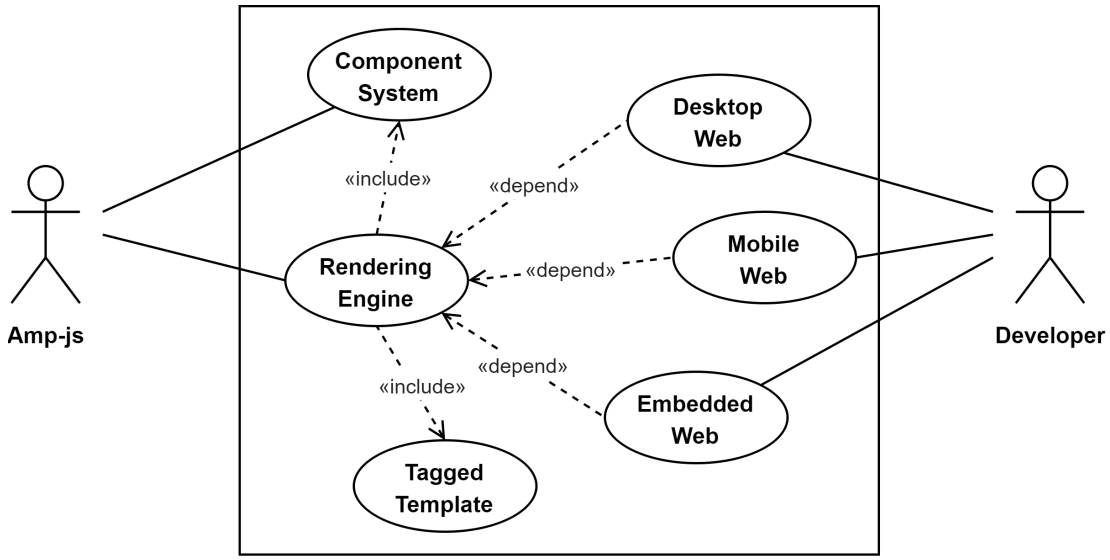

Figure 5.2: Use Case diagram

5.9 Class diagram

A class diagram describes the type of objects in the system and various kinds of static relationships that exist among them. Class diagrams also show the properties and operations of a class and the constraints that apply to the way objects are connected.

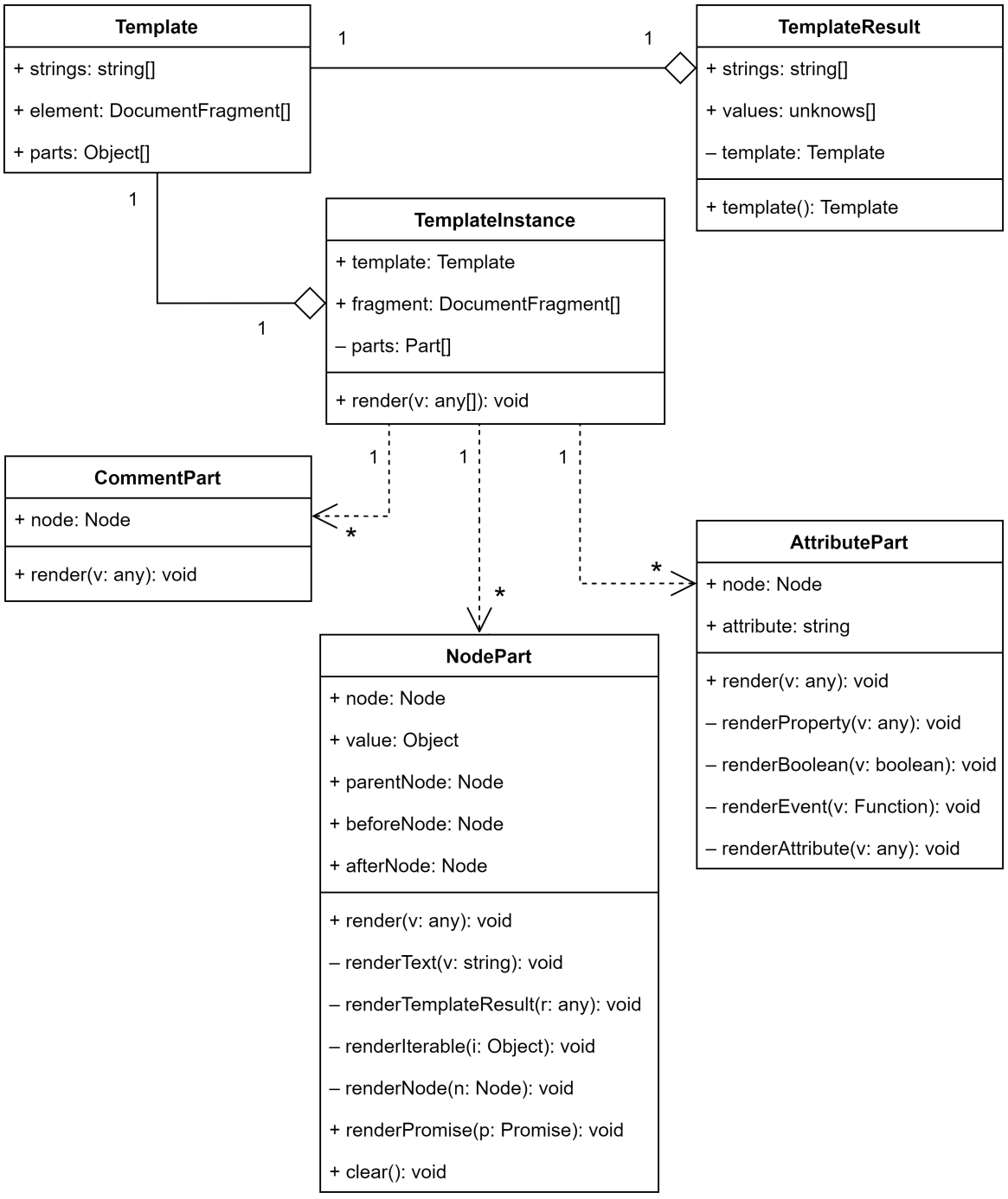

Figure 5.3: Class diagram

5.10 Activity diagram

Activity diagrams are a technique to describe procedural logic, business logic, and work flow. In many ways, they play a role similar to flowcharts, but the principal difference between them and the flowchart notation is that they support parallel behaviour.

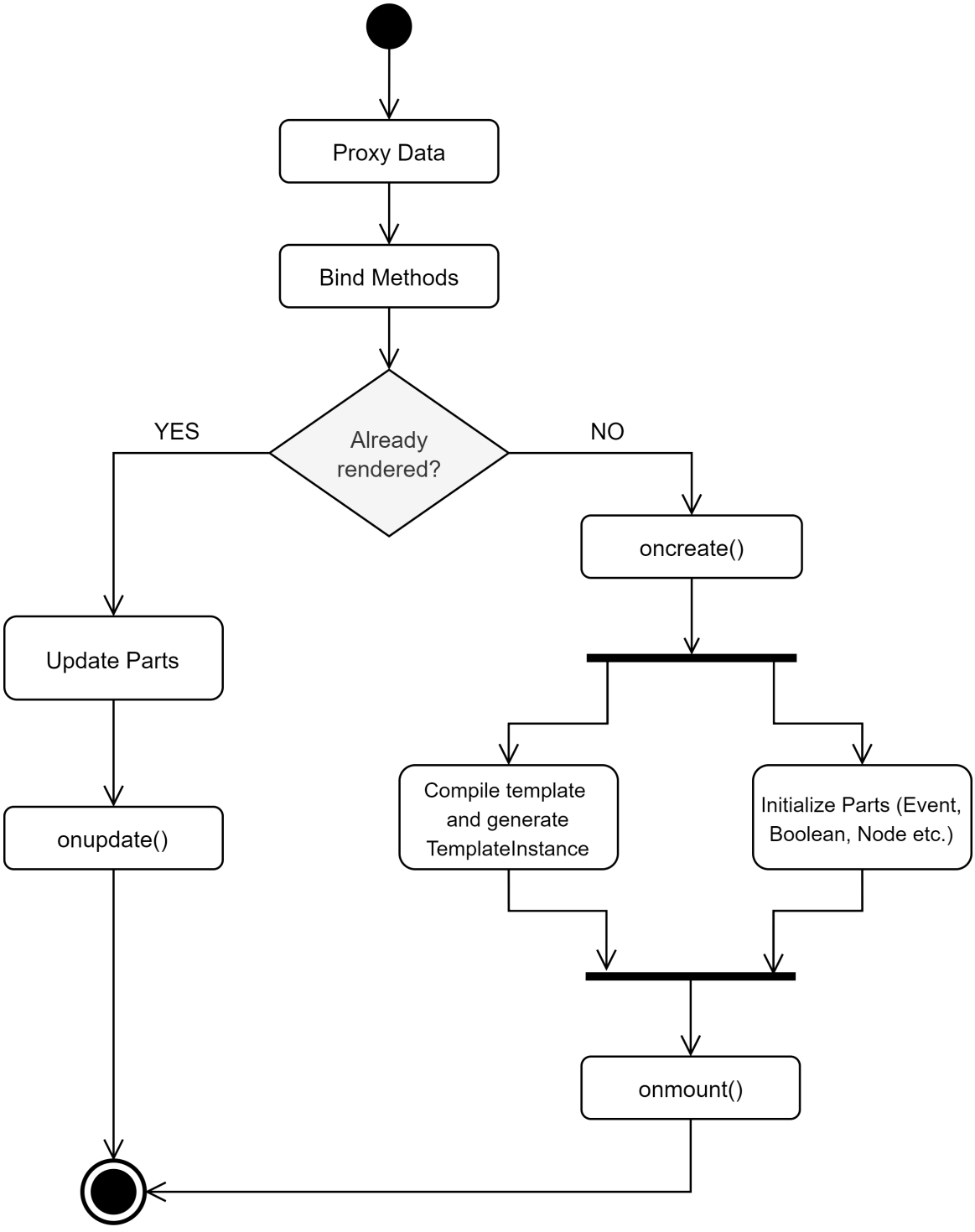

Figure 5.4: Activity diagram – Component

5.11 Sequence diagram

A sequence diagram is used to capture the behaviour of a single scenario. The diagram shows a number of example objects and the messages that are passed between these objects within the use case.

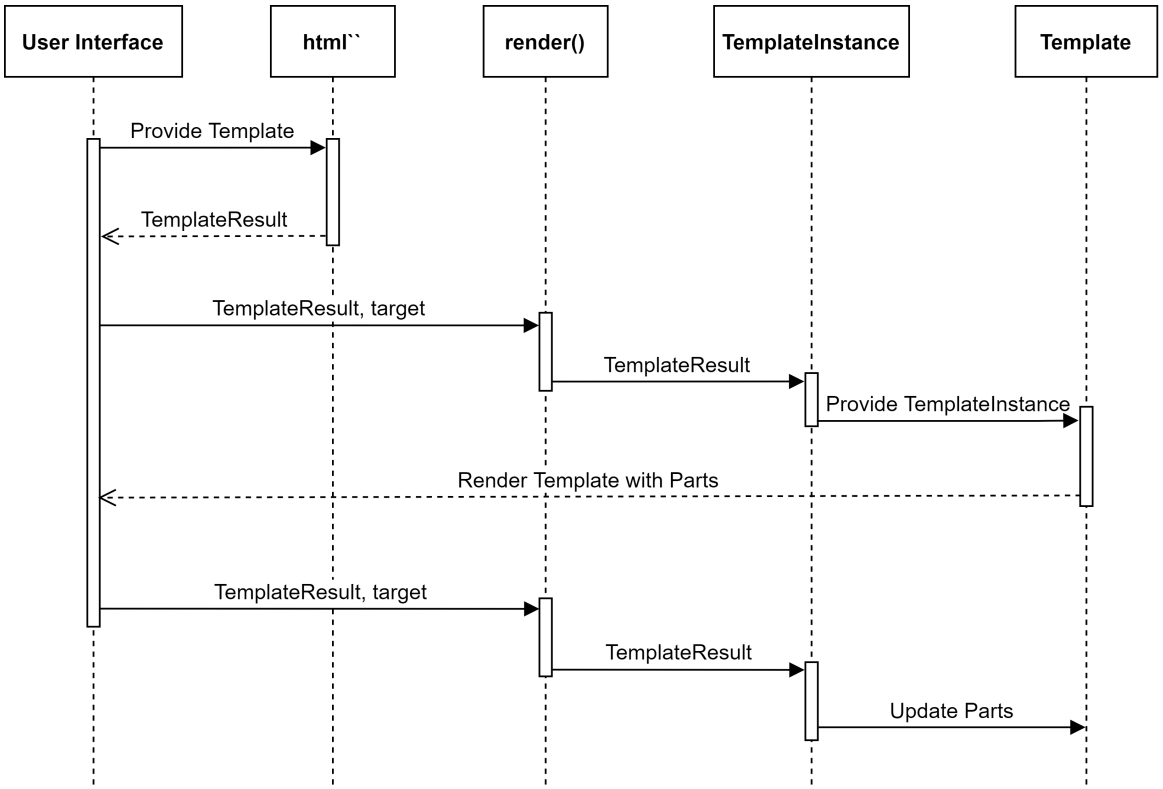

Figure 5.5: Sequence diagram

6 IMPLEMENTATION

This chapter contains the implementation of the system designed in the previous sections. The source code is written in JavaScript programming language.

6.1 Classes

Amp-js uses the ECMAScript 2015 classes which are primarily a syntactical sugar over JavaScipt's existing prototype-based inheritence. The class syntax does not introduce a new object-oriented inheritance model. In fact, **babel** transforms these classes to prototypes later during development.

6.1.1 TemplateResult

`TemplateResult` holds the strings and values that result obtaining from a tagged template string literal. It can find and return a unique `Template` object that represents its tagged template string literal.

Properties	`strings: string[]`
	`values: unknown[]`
	`template: Template`
Methods	`template(): Template`

```
class TemplateResult {
  constructor(strings, values) {
    this.strings = strings;
    this.values = values;
    this._template = undefined;
  }
  get template() {
    if (this._template)
      return this._template;
    let template = templateMap.get(this.strings);
    if (!template) {
      template = new Template(this.strings);
      templateMap.set(this.strings, template);
    }
    this._template = template;
    return template;
  }
}
```

6.1.2 Template

`Template` holds the `DocumentFragment` that is to be used as a prototype for instances of this template. When a template is to be rendered in a new location, a clone will be made from this.

Properties `strings: string[]`
`element: DocumentFragment[]`
`parts: Object[]`

```
class Template {
  constructor(strings) {
    this.strings = strings;
    this.element = buildTemplate(strings);
    this.parts = findParts(strings, this.element);
  }
}
```

6.1.3 TemplateInstance

An instance of a template that can be rendered somewhere. It is responsible for cloning the template content so that it can efficiently be appended to the DOM. Later, it iterates through all the parts of the template and fills them with content. Each part is of the type node, comment or attribute.

Properties `template: Template`
`fragment: DocumentFragment[]`
`parts: (AttributePart|CommentPart|NodePart)[]`
Methods `render(values: any): void`

```
class TemplateInstance {
  constructor(template, parent, before, after) {
    this.template = template;
    this.fragment = template.element.content.cloneNode(true);

    const parts = this.template.parts.map((part) => {
      let node = this.fragment;
      part.path.forEach((nodeIndex) => {
        node = node.childNodes[nodeIndex];
      });
      part.node = node;
```

```
    if (part.type === NodePart) {
      if (part.path.length === 1) {
        part.parent = parent;
        part.before = node.previousSibling || before;
        part.after = node.nextSibling || after;
      } else {
        part.parent = node.parentNode;
      }
    }
    return part;
  });
  this.parts = parts.map((part) => new part.type(part));
 }

 render(values) {
  this.parts.map((part, index) => part.render(values[index]));
 }
}
```

6.1.4 NodePart

`NodePart` defines the "part" that contains some textual or DOM element information. This part does not contain an attribute or a comment. If the node is defined, the `NodePart` represents the position of that node in the tree. If only a parent is defined, this `NodePart` represents the content of the parent. The type of this part is detected and is rendered using the `render()` function.

Properties
`node: Node`
`value: Object`
`parentNode: Node`
`beforeNode: Node`
`afterNode: Node`

Methods
`render(value: any): void`
`_renderText(serializable: any): void`
`_renderTemplateResult(tempRes: TemplateResult): void`
`_renderIterable(iterable: Iterable): void`
`_renderNode(node: Node): void`
`_renderPromise(promise: Promise): void`
`clear(): void`

```
class NodePart {
 constructor({ node, parent, before, after }) {
  this.node = node || emptyNode;
  this.value = noChange;

  this.parentNode = parent || (node && node.parentNode);
  this.beforeNode = before || (node && node.previousSibling);
  this.afterNode = after || (node && node.nextSibling);
 }

 render(value) {
  if (isDirective(value)) {
   value(this);
  } else if (value !== noChange) {
   // Detect the type of value and render
  }
 }

 _renderText(serializable) {
  if (this.value !== serializable) {
   if (this.node.nodeType === 3) {
    this.node.textContent = serializable;
   } else {
    this._renderNode(document.createTextNode(serializable));
   }
  }
 }

 _renderTemplateResult(templateResult) {
  this.templateInstances = this.templateInstances || new Map();
  let instance = this.templateInstances.get(templateResult.template);
  if (!instance) {
   instance = new TemplateInstance(
    templateResult.template,
    this.parentNode,
    this.beforeNode,
    this.afterNode
   );
```

```
    this.templateInstances.set(templateResult.template, instance);
  }
  if (this.node !== instance.fragment) {
    this.clear();
    this.parentNode.insertBefore(instance.fragment, this.afterNode);
    this.node = instance.fragment;
  }
  instance.render(templateResult.values);
}

_renderIterable(iterable) {
  if (this.node !== iterableNode) {
    this.clear();
    this.node = iterableNode;
    if (!this.iterableParts) {
      this.iterableParts = [];
    } else {
      this.iterableParts.length = 0;
    }
  }

  let index = 0;
  if (index === 0) {
    moveNodes(this.parentNode, this.beforeNode, this.afterNode);
  } else if (index < this.iterableParts.length) {
    const lastPart = this.iterableParts[index - 1];
    moveNodes(this.parentNode, lastPart.afterNode, this.afterNode);
  }
  this.iterableParts.length = index;
}

_renderNode(node) {
  if (this.node !== node) {
    this.clear();
    this.parentNode.insertBefore(node, this.afterNode);
    this.node = node;
  }
}
```

```
  _renderPromise(promise) {
    if (this.promise !== promise) {
      this.promise = promise;
      promise.then((value) => {
        if (this.promise === promise) {
          this.promise = undefined;
          this.render(value);
        }
      });
    }
  }

  clear() {
    moveNodes(
      this.parentNode,
      this.beforeNode,
      this.afterNode,
      this.node instanceof DocumentFragment && this.node
    );
    this.node = emptyNode;
  }
}
```

6.1.5 CommentPart

This class keeps in track the comments added with template instance.

Properties `node: Node`
Methods `render(value: any): void`

```
class CommentPart {
  constructor({ node }) {
    this.node = node;
  }

  render(value) {
    this.node.textContent = value;
  }
}
```

6.1.6 AttributePart

The attributes passed with the template are assigned to their respective element. The class is responsible for handling and updating different types of attributes. `AttributePart` can be of type property, event listener, a boolean attribute or an actual HTML attribute.

Properties	`node: Node`
	`render: Function`
Methods	`render(value: any): void`
	`_renderProperty(value: any): void`
	`_renderBoolean(boolean: boolean): void`
	`_renderEvent(listener: Function): void`
	`_renderAttribute(attr: string): void`

```
class AttributePart {
 constructor({ node, attribute }) {
  this.node = node;
  switch (attribute[0]) {
   case ":":
    this._render = this._renderProperty;
   case "?":
    this._render = this._render || this._renderBoolean;
   case "@":
    this._render = this._render || this._renderEvent;
    this.node.removeAttribute(attribute);
    this.name = attribute.slice(1);
    break;
   default:
    this._render = this._renderAttribute;
    this.name = attribute;
  }
 }

 render(value) {
  if (isDirective(value)) {
   value(this);
  } else if (value !== noChange) {
   this._render(value);
  }
 }
```

```
  _renderProperty(value) {
    if (typeof value === "undefined") {
      throw new Error("undefined cannot be assigned");
    }
    this.node[this.name] = value;
  }

  _renderBoolean(boolean) {
    if (this.value !== !!boolean) {
      boolean
        ? this.node.setAttribute(this.name, "")
        : this.node.removeAttribute(this.name);
      this.value = !!boolean;
    }
  }

  _renderEvent(listener) {
    if (typeof listener === "undefined") {
      throw new Error("undefined cannot be assigned");
    }
    if (this.value !== listener) {
      this.node.removeEventListener(this.name, this.value);
      this.node.addEventListener(this.name, listener);
      this.value = listener;
    }
  }

  _renderAttribute(string) {
    if (typeof string === "undefined") {
      throw new Error("undefined cannot be assigned");
    }
    if (this.value !== string) {
      this.node.setAttribute(this.name, string);
      this.value = string;
    }
  }
}
```

6.2 Functions

Amp-js uses the ES6 arrow functions expressions which are the syntactically compact alternative to a regular JavaScript function expression, although it lacks its own bindings to `this`, `argument`, or `super` keywords. They are transpiled to an equivalent function expression after the build phase by **babel**.

6.2.1 html

This function is available as an export. It is the tagged template function that generates the DOM representation object which is then rendered by calling `render` and passing the target element.

Parameters `strings: string[]`
`...values: unknown[]`
Returns `TemplateResult`

```
const html = (strings, ...values) => {
  return new TemplateResult(strings, values);
};
```

6.2.2 render

This function is responsible for creating DOM from its representation. The DOM then is rendered on the provided target element. It is available as an exported function. This function can be called multiple times passing different state and it finds the most efficient way of updating them without any form of *diffing*.

Parameters `content: any`
`target: Node`

```
const render = (content, target) => {
  let part = nodeParts.get(target);
  if (!part) {
    part = new NodePart({ parent: target });
    nodeParts.set(target, part);
  }
  part.render(content);
};
```

6.2.3 component

This function is exported with the default export object. It is used for generating the component instance, which can be rendered by calling its `generate` function.

Parameters	`id: string`
	`definition: Object`
Returns	`id: string`
	`generate: Function`

```
const component = (id, definition) => {
  const generate = (nodes) => {
    validate(id, definition);
    nodes.forEach((node) => {
      const proxyHandler = {
        get: function (target, key) {
          if (typeof target[key] === 'object' && target[key] !== null) {
            return new Proxy(target[key], proxyHandler);
          } else {
            return target[key];
          }
        },
        set: function (target, key, value) {
          let flag = key in target;
          target[key] = value;
          if (flag) {
            render(template.call(state), node);
            updateChildren(components);
            callLifeCycle(lifeCycle.onupdate);
          }
          return true;
        }
      };


      const updateChildren = (components) => {
        if (components && components.length > 0) {
          components.forEach((comp) => {
            const isInDOM =
              document.querySelectorAll(comp.id).length > 0;
```

```
        if (isInDOM)
          comp.generate();
      });
    }
    const childNodes = node.querySelectorAll(id);
    if (childNodes.length > 0)
      generate(childNodes);
  };
  let state = createData(data);
  const mCopy = { ...methods };
  if (mCopy) {
    const methodNames = Object.keys(mCopy);
    if (methodNames.length > 0)
      methodNames.forEach((name) => {
        state[name] = mCopy[name].bind(state);
      });
    }
  state.attr = (name) => node.getAttribute(name);
  state = new Proxy(state, proxyHandler);
  render(template.call(state), node);
 });
};
return {
 id,
 generate: () => generate(document.querySelectorAll(id))
};
};
```

The above code including some ***util*** functions are spread across multiple files. **rollup** combines them together and **babel** handles their transpilation.

7 TESTING

This chapter introduces the different testing strategies used during the development as well as the final release of the project. It also incorporates the various test cases for which the system was tested and their results.

7.1 Introduction

Testing is the process of executing a program with the intent of finding errors[19]. It is conducted in order to gather information about the quality of the system. Through software testing, we can test whether a system or its part produces the expected result or not. It detects software failures so that defects may be discovered and corrected. The scope of software testing includes the inspection of code in various environments and conditions, called as *test cases* as well as assessing the aspects of code: if it does what it is supposed to do and what it needs to do.

Unit Testing

Unit testing is a way of testing a unit – the smallest piece of code that can be logically isloated in a system. It is the first part of testing. In most programming languages, that is a function, a subroutine, a method or property. Unit testing is an important step in the development process, because if done correctly, it can help detect early flaws in code which may be more difficult to find in later testing stages. It allows the programmmer to refactor code and upgrade system libraries at a later stage, and make sure that the module still works correctly. This is achieved by writing test cases for all functions and methods so that whenever a change causes a fault, it can be quickly identified without changing the whole system.

Integration Testing

Integration testing is performed after unit testing. It determines if independently developed pieces of software work correctly when they are connected to each other. This combination can create either an entire system or a significant sub-system. There are different type of integration testing: big-bang, mixed(sandwich), top-down and bottom-up.

In big-bang approach, the individual modules are coupled together to form the complete system or its major part and then used for integration testing. Bottom-up testing approach tests lower level components firstly, and then it is used to facilitate the testing of higher level components. Top-down tests the top integrated modules and its branch is tested gradually until the end of related module. Sandwich testing combines top-down testing with bottom-up testing.

7.2 Jest

Jest is a testing framework which is used to test the code written in JavaScript. It can perform both unit and integration tests. Test suites in Jest are declared in a file with extension `.test.js`. `describe(name, fn)` creates a block that groups several related tests. **Test conditions** are specified in the `test(name, fn)` function. Here the function `fn` contains various **Test cases**. Each test case depends upon two parameters: expected value and actual value. The `expect` function along with a "matcher" function is used to assert something about a value. For example,

```
describe("block", () => {
  test("test name", () => {
    expect(value).toBe(expectedValue);
  });
});
```

7.3 Test Cases

Testing of the system is spread across four test suites which are further divided among various scopes:

- **parser » nodeWalker**

  Scope: findParts

Input	Expected Output	Actual Output	Result
Test case: Correctly detects part types **Expression:** `` html`<!--${0}--><div a=${1}>${2}</div>`; ``			
`parts[0].type`	`CommentPart`	`CommentPart`	PASS
`parts[1].type`	`AttributePart`	`AttributePart`	PASS
`parts[2].type`	`NodePart`	`NodePart`	PASS
Test case: Returns the correct path for node parts **Expression:** `` html`<div>${0}<div>${0}</div></div>`; ``			
`parts[0].path`	`[0, 1]`	`[0, 1]`	PASS
`parts[1].path`	`[0, 3, 0]`	`[0, 3, 0]`	PASS
Test case: Preserves original attribute names **Expression:** `` html`<div a=${0} a-b=${1} (a)=${3} a$=${6}></div>`; ``			
`parts[0].attribute`	`a`	`a`	PASS
`parts[1].attribute`	`a-b`	`a-b`	PASS
`parts[2].attribute`	`(a)`	`(a)`	PASS
`parts[3].attribute`	`a$`	`a$`	PASS

Test case: Considers comment nodes in parts **Expression:** `html` <!-- -->${0}<!-- -->${0} `;`			
`parts[0].path`	`[0, 2]`	`[0, 2]`	PASS
`parts[1].path`	`[0, 5, 1]`	`[0, 5, 1]`	PASS
Test case: Returns the correct path for attribute parts **Expression:** `html`<div a=${0}> <div a=${0}> `;`			
`parts[0].path`	`[0]`	`[0]`	PASS
`parts[1].path`	`[0, 3]`	`[0, 3]`	PASS
Test case: Preserves prefixes in the attribute name **Expression:** `html`<div :a=${0} ?a=${1} @a=${2}> `;`			
`parts[0].attribute`	`:a`	`:a`	PASS
`parts[1].attribute`	`?a`	`?a`	PASS
`parts[2].attribute`	`@a`	`@a`	PASS
Test case: Does not break on the "style" attribute **Expression:** `html`<div a=${0} style=${”} b=${1}> `;`			
`parts[0].attribute`	`a`	`a`	PASS
`parts[1].attribute`	`style`	`style`	PASS
`parts[2].attribute`	`b`	`b`	PASS

- **parser » templateParser**

  Scope: `parseContext`

Input	Expected Output	Actual Output	Result
Test case: Detects comment contexts			
`parseContext("<!--")`	`commentContext`	`commentContext`	PASS
`parseContext("<div><!--")`	`commentContext`	`commentContext`	PASS
`parseContext("<!-- <p")`	`commentContext`	`commentContext`	PASS
`parseContext("<!-- a=")`	`commentContext`	`commentContext`	PASS
Test case: Detects node contexts			
`parseContext("<div>")`	`nodeContext`	`nodeContext`	PASS
`parseContext("<div>text")`	`nodeContext`	`nodeContext`	PASS
`parseContext("<div> a=")`	`nodeContext`	`nodeContext`	PASS
`parseContext("<!-->")`	`nodeContext`	`nodeContext`	PASS
Test case: Detects attribute contexts			
`parseContext("<div a=")`	`attributeContext`	`attributeContext`	PASS
`parseContext("<div a =")`	`attributeContext`	`attributeContext`	PASS

Scope: `buildTemplate`

Test case: Assigns the attributeMarker value to dynamic attributes **Expression:** `buildTemplate(html` `);`			
`content.getAttribute("a")`	`attributeMarker`	`attributeMarker`	PASS

Expression: `buildTemplate(html`<div a=${0} b=${0} c=${0}> `);`			
`content.getAttribute("a")`	`attributeMarker`	`attributeMarker`	PASS
`content.getAttribute("b")`	`attributeMarker`	`attributeMarker`	PASS
`content.getAttribute("c")`	`attributeMarker`	`attributeMarker`	PASS

- **template » parts**

Scope: `isSerializable`

Input	Expected Output	Actual Output	Result
Test case: Should return a truthy value for strings, numbers, and booleans			
`!!isSerializable("")`	`true`	`true`	PASS
`!!isSerializable(0)`	`true`	`true`	PASS
`!!isSerializable(true)`	`true`	`true`	PASS
Test case: Should return a falsy value for other things			
`!!isSerializable(null)`	`false`	`false`	PASS
`!!isSerializable(undefined)`	`false`	`false`	PASS
`!!isSerializable(Symbol())`	`false`	`false`	PASS
`!!isSerializable({})`	`false`	`false`	PASS
`!!isSerializable([])`	`false`	`false`	PASS
`!!isSerializable(html``)`	`false`	`false`	PASS
`!!isSerializable(() => {})`	`false`	`false`	PASS

Scope: `AttributePart`

Test case: Render Attributes **Expression:** `part = new AttributePart({ node, attribute: "a" });`			
`part.node === node`	`true`	`true`	PASS
`part.name`	`"a"`	`"a"`	PASS
For a, `part._render === part._renderAttribute`	`true`	`true`	PASS
For :a, `part._render === part._renderProperty`	`true`	`true`	PASS
For @a, `part._render === part._renderEvent`	`true`	`true`	PASS
For ?a, `part._render === part._renderBoolean`	`true`	`true`	PASS
`part.render("one"); node.getAttribute("a")`	`"one"`	`"one"`	PASS

Scope: `CommentPart`

Test case: Render Comments **Expression:** `part = new CommentPart({ node, attribute: "" });`			
`part.node === node`	`true`	`true`	PASS
`part.textContent`	`"test"`	`"test"`	PASS
`part.render("one"); node.textContent`	`"one"`	`"one"`	PASS
`part.render("two"); node.textContent`	`"two"`	`"two"`	PASS

Scope: `NodePart`

Test case: Remembers what node it represents **Expression:** `part = new NodePart({ node });`			
`part.node === node`	`true`	`true`	PASS
`part.parentNode === parent`	`true`	`true`	PASS
Test case: Removes nodes that this NodePart represents from the DOM **Expression:** `part = new NodePart({ node }); part.clear();`			
`parent.outerHTML`	`<div></div>`	`<div></div>`	PASS
Test case: Reassigns parents for nodes that are DocumentFragment contents **Expression:** `fragment = document.createDocumentFragment(parent);`			
`fragment`	`<!-- marker -->`	`<!-- marker -->`	PASS
Test case: Moves nodes back into the DocumentFragment after rendering			
`fragmentString`	`<li></li>`	`<li></li>`	PASS
`fragmentString.outerHTML`	`<ul></ul>`	`<ul></ul>`	PASS

- **template » templates**

Scope: `TemplateResult`

Input	**Expected Output**	**Actual Output**	**Result**
Test case: Stores the strings and values **Expression:** `{ strings, values } = html` ${0} `;`			
`strings`	`["<div>", "</div>"]`	`["<div>", "</div>"]`	PASS
`value`	`[0]`	`[0]`	PASS
Test case: Lazily loads the template **Expression:** `html``;`			
`_template`	`undefined`	`undefined`	PASS
`template()`	`true`	`true`	PASS
`_template()`	`true`	`true`	PASS

Scope: `Template`

Test case: Stores the strings that constructed the template **Expression:** `new Template(htmlStrings` ${0} `);`			
`template.strings`	`strings`	`strings`	PASS
Test case: Constructs a template element that holds a DOM template			
`instanceof TemplateElement`	`true`	`true`	PASS
Test case: Constructs a template element that holds a DOM template **Expression:** `template = () => html``;`			
`template()`	`Same instance`	`Same instance`	PASS

Scope: `TemplateInstance`

Test case: Clones the template document fragment from the source Template **Expression:** `new TemplateInstance(html` ${0} `.template);`			
`template.element.content`	`fragment`	`fragment`	PASS
Test case: Constructs Part instances according to the definitions from the Template **Expression:** `html` ${0}${0} `;`			
`instance.parts[0] instanceof NodePart`	`true`	`true`	PASS
`instance.parts[1] instanceof NodePart`	`true`	`true`	PASS
`instance.parts[2] instanceof NodePart`	`true`	`true`	PASS
`instance.parts[3] instanceof AttrPart`	`true`	`true`	PASS
`instance.parts[4] instanceof AttrPart`	`true`	`true`	PASS
`instance.parts[5] instanceof CommentPart`	`true`	`true`	PASS
Test case: Calls "render" on the parts with the correct values **Expression:** `new TemplateInstance(html`${3}${3}${3}`.template)`			
`part._renderCalledWith === index)`	`true`	`true`	PASS
`instance.render()`	`[0, 1, 2]`	`[0, 1, 2]`	PASS

Note: The test cases described above are only a part of the testing performed on the system. Describing every single test case will lengthen the scope of this report.

7.4 Coverage

Code coverage is the metric which describes the degree of source code of the program that has been tested. It helps assess the quality of the test suites. The common metrics that are mentioned in a coverage report include:

- **Statement coverage:** To know how many of the statements in the program have been executed.
- **Branches coverage:** To examine how many of the branches of the control structures (if statements for instance) have been executed.
- **Function coverage:** To observe how many of the functions defined have been called.
- **Line coverage:** To evaluate how many of lines of source code have been tested.

Following is the coverage report for the test suites performed on the system:

File	Statement	Branch	Function	Line
parser » directive.js	66.76%	100%	50%	60%
parser » dom.js	100%	87.5%	100%	100%
parser » node-walker.js	96.88%	93.75%	100%	96.67%
parser » template-parser.js	100%	100%	100%	100%
parser	96.81%	95.12%	90.91%	96.59%
template » markers.js	100%	87.5%	100%	100%
template » parts.js	90.52%	82.95%	100%	90.52%
template » template.js	100%	87.5%	100%	100%
template	92.9%	85%	100%	92.81%

8 CONCLUSION

From every aspect, the project is functionally successful. The design goals are achieved, and a working, lightweight JavaScript library has been developed. More work is required before amp-js can be used as a framework, but in its current state this library can generate interactive user interfaces and perform updates faster than a virtual DOM.

The library successfully inherits the advantages of JavaScript in a clean and clear syntax, provides those features that were not available in a majority of templating engines, and makes amends for performance shortcomings through separation of concerns. Amp-js is only 4kB in size (minified + gzipped) which is a lot smaller than many UI frameworks that take up the majority of an app's JavaScript size. Smaller size means less JavaScript to download, parse and execute — leaving more time for the application code.

Template written in amp-js are efficient: only the parts that actually change will be updated, expressive: full power of JavaScript and functional programming patterns can be used to create templates, and it is extensible: interface can be broken down into reusable components, each having its own execution logic and data bindings.

9 FUTURE EXTENSIONS

Global State Management

A global state management that uses flux or state machine architecture could be implemented to share pieces of a global state through different parts of the application.

Client-side Routing

Client-side routing is an integral part of single-page applications (SPA). Although a basic hash-based routing may be achieved by listening to a `hashchange` event in the `onmount` lifecycle hook, a dedicated router using JavaScript's history API should be relatively simple to implement.

Scoped CSS

A tagged template function for writing styles could be developed which may keep the CSS rules *scoped to the component*.

REFERENCES

[1] The World Wide Web Consortium (W3C). *Custom Elements*. https://www.w3.org/TR/custom-elements/ Last Accessed: 03/29/2020.

[2] The World Wide Web Consortium (W3C). *Shadow DOM*. https://www.w3.org/TR/custom-elements/ Last Accessed: 03/29/2020.

[3] The World Wide Web Consortium (W3C). *The `template` Element*. https://www.w3.org/TR/html52/semantics-scripting.html#the-template-element Last Accessed: 04/02/2020.

[4] Dan Abramov. *Redux – A Predictable State Container for JavaScript Apps*. https://redux.js.org/ Last Accessed: 03/28/2020.

[5] Scott Chacon. *Git – fast version control system*. https://git-scm.com/ Last Accessed: 04/12/2020.

[6] Dominic Cooney. *Introduction to Web Components*. https://www.w3.org/TR/components-intro/ Last Accessed: 03/29/2020.

[7] Ryan Dahl. *Node.js – JavaScript runtime*. https://nodejs.org/en/ Last Accessed: 04/12/2020.

[8] The jQuery Foundation. *jQuery: The Write Less, Do More, JavaScript Library*. https://jquery.com/ Last Accessed: 03/27/2020.

[9] Rich Harris. *Rollup – JavaScript Module Bundler*. https://rollupjs.org/guide/en/ Last Accessed: 04/12/2020.

[10] Rich Harris. *Virtual DOM is pure overhead*. https://svelte.dev/blog/virtual-dom-is-pure-overhead Last Accessed: 04/09/2020.

[11] Rich Harris. *Why I don't use web components*. https://dev.to/richharris/why-i-don-t-use-web-components-2cia Last Accessed: 04/11/2020.

[12] Michael L. Haufe. *A Criticism of Web Components*. https://thenewobjective.com/web-development/a-criticism-of-web-components Last Accessed: 04/11/2020.

[13] Leo Horie. *Mithril – A JavaScript Framework for Building Brilliant Applications*. https://mithril.js.org/ Last Accessed: 03/28/2020.

[14] Pete Hunt. *Why did we build React?* https://reactjs.org/blog/2013/06/05/why-react.html Last Accessed: 04/17/2020.

[15] Facebook Inc. *React – A JavaScript library for building user interfaces*. https://reactjs.org/ Last Accessed: 03/28/2020.

[16] GitHub Inc. *npm – build amazing things*. https://www.npmjs.com/ Last Accessed: 04/12/2020.

[17] Google Inc. *AngularJS – Superheroic JavaScript MVW Framework*. https://angularjs.org/ Last Accessed: 03/27/2020.

[18] Sebastian McKenzie. *Babel – The compiler for next generation JavaScript*. https://babeljs.io/ Last Accessed: 04/12/2020.

[19] Glenford Myers. *The Art of Software Testing*. Hoboken, New Jersey: John Wiley & Sons, Inc., 2004, p. 11.

[20] React. *JSX In Depth*. https://reactjs.org/docs/jsx-in-depth.html Last Accessed: 03/29/2020.

[21] React. *Virtual DOM and Internals*. https://reactjs.org/docs/faq-internals.html Last Accessed: 03/26/2020.

[22] *Stack Overflow Developer Survey*. https://insights.stackoverflow.com/survey/2019 Last Accessed: 03/27/2020.

[23] Ecma TC39. *Modules*. https://tc39.es/ecma262/#sec-modules Last Accessed: 04/05/2020.

[24] Ecma TC39. *Tagged Templates*. https://tc39.es/ecma262/#sec-tagged-templates Last Accessed: 04/05/2020.

[25] Ecma TC39. *Template Literals*. https://tc39.es/ecma262/#sec-template-literals Last Accessed: 04/05/2020.

[26] David Foster Wallace. *Jest – Delightful JavaScript Testing*. https://jestjs.io/ Last Accessed: 04/12/2020.

[27] Allen Wirfs-Brock and Brendan Eich. "JavaScript: The First 20 Years". In: (2020). DOI: https://doi.org/10.5281/zenodo.3710954.

[28] Evan You. *Vue.js – The Progressive JavaScript Framework*. https://vuejs.org/ Last Accessed: 03/28/2020.

[29] Nicholas C. Zakas. *ESLint – Pluggable JavaScript Linter*. https://eslint.org/ Last Accessed: 04/12/2020.

APPENDIX I

PROJECT TITLE	CATEGORY
AMP-JS: A Lightweight View-Layer Library for Building Scalable Web Applications	Research

Abstract: JavaScript has been synonymous with building modern web applications for the past twenty years. Event-driven architecture has predominantly been followed to achieve native app-like experience on the browser. The problem with this architecture is scalability. Increase in the size of the application is directly proportional to the amount of code written. Frameworks built on the top of JavaScript have solved a lot of problems. But they are complex and have a substantial cost on the produced size of the application due to their build tooling, dependencies, and polyfills. Amp.js provides a declarative approach for creating data-driven applications through separation of concerns. It uses HTML Templates with Tagged Template Literals to build reactive User-Interfaces. The View layer is bound efficiently to the Modal of the system creating a two-way data binding for efficiently updating the DOM.

Keywords: Library, DOM, Templates, Bundle size, User Interface.

Table I.1: Mapping to POs

(Mapping Scale [1-3]: 1–Slight 2–Moderate 3–Strong)

PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
3	3	3	2	3	1	3	3	3	2	2	2

Table I.2: Mapping to PSOs

(Mapping Scale [1-3]: 1–Slight 2–Moderate 3–Strong)

PSO1	PSO2	PSO3
3	2	3

RELEVANCE DETAILS	
Implementation Details The project is implemented using Language: JavaScript (ES2015).	
PO and PSO Justification	
PO1	It is strongly mapped as the basic concepts of Mathematics, Physics and Engineering fundamentals are used for designing a solution to the complex engineering problem of constructing an efficient HTML template rendering engine.
PO2	It is strongly mapped as the concept of Separation of concerns are used to distinguish between the static and dynamic parts of HTML templates.
PO3	It is strongly mapped as efficiency and robustness are considered while generating the syntax.
PO4	It is moderately mapped as there are many strategies that can be used for templating. We selected the most efficient one for the proposed system. Critical thinking and the ability to analyze and solve complex real-world problems is done for developing the application.
PO5	It is strongly mapped as JavaScript language and Babel are modern IT tools used in our project.
PO6	It is weakly mapped as it does not add anything to the society but improving the developer experience.
PO7	It is strongly mapped as the project is sustainable to the environment and does not cause any kind of harm to it.
PO8	It is strongly mapped because the plagiarism tool can be used to check the authenticity and originality of the work done by the students.
PO9	Projects are used to inculcate group work and to manage a team for promoting knowledge, conceptualization and delivering same with varied complexity, therefore the mapping is strong.
PO10	Demonstrate versatile and effective communication skills, both verbal and written with team members and present the product to the audience in comprehensive manner. Therefore the mapping is moderate.
PO11	Project Management tools like CPM and Pert etc. can be used to propose and work accordingly, hence the mapping is moderate.
PO12	The mapping is moderate as projects are executed based on the self learning or self efforts put in by the group.
PSO1	It is strongly mapped as understanding of the principles and working of the hardware and software aspects of computer systems is required to decomposed the system into phases and workflows.
PSO2	It is moderately mapped as cases are explored to ensure reliability of the product, optimal quality and tested thoroughly to validate the ongoing process.
PSO3	It is strongly mapped as self learning skills are applied for a real world problem by using content knowledge and acquired intellectual skills.

APPENDIX II

Project Plan

Academic Year	2019-20
Project Title	Amp-js
Project Supervisor	Dr. Krishna Keerthi Chennam
Start Date	4-Apr-2019
End Date	12-Apr-2020

S.No.	Tasks	Start	End	Days	Status
1	**Project and Batch Registration**	**4-Apr-2019**	**27-Apr-2019**	**23**	**Completed**
2	Problem Definition	10-Apr-19	12-Apr-19	2	**Completed**
3	Abstract	12-Apr-19	20-Apr-19	8	**Completed**
4	Literature Survey	20-Apr-19	18-May-19	28	**Completed**
5	Requirements Analysis	17-Jul-19	10-Aug-19	24	**Completed**
6	**Project Seminar**	**18-Aug-19**	**16-Sep-19**	**29**	**Completed**
7	**Project Review 1**	**6-Oct-18**	**12-Oct-18**	**2**	**Completed**
8	Feasibility Study	24-Oct-19	6-Nov-19	13	**Completed**
9	Detailed Design/Module Identification	16-Nov-19	8-Dec-19	22	**Completed**
11	Implementation	6-Jan-20	15-Feb-20	40	**Completed**
12	Testing	15-Feb-20	2-Mar-20	16	**Completed**
13	Result Analysis	3-Mar-20	10-Mar-20	7	**Completed**
14	**Project Review 2**	**11-Mar-20**	**16-Mar-20**	**5**	**Completed**
15	Report Compilation	17-Mar-20	14-Apr-20	28	**Completed**
16	**Viva Voce**	**15-Apr-20**	**27-Apr-20**	**12**	**--**

Table II.1: Overall Progress

Gantt Chart depicting Overall Progress