Bachelor's thesis

Information and Communications Technology

2020

Duong Dinh & Zhuanyan Wang

MODERN FRONT-END WEB DEVELOPMENT

– How libraries and frameworks transform everything

TURKU AMK

TURKU UNIVERSITY OF APPLIED SCIENCES

BACHELOR'S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Information and Communications Technology

2020

Duong Dinh & Zhuanyan Wang

MODERN FRONT-END WEB DEVELOPMENT

- How libraries and frameworks transform everything

There is a large number of libraries, frameworks, and utilities for front-end web development as of 2020. Although they have helped to transform front-end web development significantly, the number of these tools is multiplying so fast that it seems impossible to keep track of it anymore. This thesis first investigates the issues traditional front-end technology stack has, then explores some of the most widely adopted front-end web development tools to examine how they approach these issues. Through these analyses, the thesis should answer the question as to why there are so many libraries, frameworks, and utilities for front-end web development.

The issues are categorized based on which part of the development process they belong to. Duong Dinh is responsible for providing historical context and background information on the issues as well as the alternatives to the technologies involved, while case studies on the issues along with code snippets are presented by Zhuanyan Wang.

Overall, this thesis discovers that there are three primary reasons behind the appearance of all the third-party front-end web development tools. Firstly, there are fundamental flaws with the way HTML, CSS, and JavaScript are designed. Secondly, the development pace of HTML, CSS, and JavaScript cannot catch up with the speed at which the web is evolving. Thirdly, there is no authority figure in the front-end web development to enforce official guidelines and to provide an official integrated development environment or developer kit like that of mobile or desktop platforms.

Since the number of front-end web libraries, frameworks, utilities, language extensions. is ever-increasing, it is becoming difficult to make sense of the situation. This thesis clarifies the current situation of front-end web development by analyzing the issues and the tools that are created to toggle them.

KEYWORDS:

Front-end web development, web development, web framework, web library, web platform

CONTENTS

FIGURES

LIST OF ABBREVIATIONS

API – Application Programming Interface

CLI – Command-line Interface

CMS – Content Management System

CSS – Cascading Style Sheets

CSS WG – CSS Working Group

DOM – Document Object Model

ECMA – European Computer Manufacturer’s Association

GUI – Graphic User Interface

HTML – Hypertext Markup Language

IDE – Integrated Development Environment

IE – Internet Explorer

JS – JavaScript

JSON – JavaScript Object Notation

MDN – Mozilla Developer Network

MVC – Model View Controller

NPM – Node Package Manager

SASS – Syntactically Awesome Style Sheets

SPA – Single Page Application

UI – User Interface

UX – User Experience

VS Code – Visual Studio Code

XML – Extensible Markup Language

WHATWG – Web Hypertext Application Technology Working Group

WWW – World Wide Web

1 INTRODUCTION

The ever-increasing number of libraries, frameworks, utilities, etc. exist in front-end web development (Greif, et al., 2019) (Greif & Benitte, 2019) has caused massive confusion to beginners and developers from other areas of software engineering alike. It is more important now than ever to understand the fundamental reasons behind their appearance.

The ultimate objective of this thesis is to explain why there are so many tools created for the front-end web. Essentially, these tools are created to toggle the issues of front-end web development. Thus, to achieve the goal of making sense of all the tools, the thesis first explores what causes all these development issues in the first place. Then, some of the most prominent libraries, frameworks, etc. are chosen to form a case study to dive deep into the relationship among the technology stack, the issues, and the third-party tools. Finally, the possible alternative solutions other than the ones presented in the case studies will be discussed to give readers a broader view of the current state of the subject involved.

Chapter 2 Background provides an overview of the past and present state of front-end web development.

Chapter 3 Methodology explains how this thesis plans to achieve its objectives.

Chapter 4 Analysis of the Primary Technology Stack dives deep into the issues of developing a website in the key front-end web technology stack and how some of the third-party tools are created to toggle these issues.

Chapter 5 Analysis of the Secondary Technology Stack examines the tools that build to improve and automate the development process of the front-end web.

Chapter 6 Results presents the findings of these studies and analyses.

Chapter 7 Conclusions section summarizes the thesis and presents the authors' perspective as well as provides advice for beginners and developers from other areas of software engineering who want to dabble into front-end web development.

2 BACKGROUND

2.1 Primary Technology Stack

Fundamentally, a web page consists of Hypertext Markup Language (HTML), Cascading Style Sheets (CSS), and JavaScript. HTML defines the structure of a web page, the CSS stylizes the page, and JavaScript enables user interactions. The three of them combined are considered front-end web development (World Wide Web Consortium, 2016).

A web page, as its name suggests, is a page of texts and images hosted on the web. In the traditional sense, a web page focuses more on merely presenting the information. However, the popularization of smart handheld devices such as smartphones and tablet computers has changed how users perceive and consume digital content forever. With much more powerful hardware across the board, what a browser can do on both mobile and desktop platforms has also evolved accordingly. As a result, a website today is expected to not only presenting information but also to provide users with responsive user interfaces, rich features, high performance, and great user experiences (UX).

In pursuit of all these, the complexity of an average modern web project has increased and the traditional way of developing a web site seems no longer sufficient to support its scale and scope. Thus many tried-and-tested concepts, patterns, and principles from the more traditional software engineering like definition of modularization, model view controllers (MVC), separation of concerns (SoC), object-oriented programming (OOP), and even static typing have all made their way to the realm of web development in hope to ensure a great developer experience (DX).

However, these ideas from software engineering are often incompatible with HTML, CSS, and JavaScript. Moreover, the speed at which the W3C working groups and Ecma International issue their new iteration of HTML, CSS, and JavaScript cannot catch up with the web's rapid evolution. Consequently, many front-end web libraries, frameworks, and utilities are created to help developers modularizing, automating, and testing their projects, but it has become increasingly challenging to keep track of all the new tools (WHATWG, 2020).

2.2 Secondary Technology Stack

As numerous of libraries, frameworks, utilities, etc. are built to solve the fundamental issues of HTML, CSS, and JavaScript, an exponential number of tools are also created to enhance the developer experience while building a front-end web project. This section will discuss the problems that lay in modern web development and their solutions. Furthermore, the section also points out other tools that are commonly used and the procedure to make a complete web application.

Initially, for a developer to write lines of codes, it is crucial to obtain a development environment, and since there are not any authentic organizations or enterprises that came up with the official development environment for web developers to handle their codebase, therefore, many editors have been created by private companies or individuals in order to fill the gap. Afterward, a package manager is a tool to help to keep track of what software is installed onto the computer and allows developers to easily perform needed actions. In order to install the node package manager, Node.js installation is a must. Originally, Node.js was designed as a server environment for applications, but developers started using it to create tools to aid them in local task automation.

The web industry is constantly evolving, browsers, however, may not support the latest technologies if users have not yet updated the browsers. This creates an irony in that the new features introduced in the latest version greatly improve the developers' experience, yet they do not support the users' software. Hence, source-to-source tools are also invented in order to compile codes from the latest version written by programmers, then produce the output to the users with their current version. There are many kinds of source-to-source compiler, which allow solving syntax compatibility (Babel), typing system (Typescript), preprocessor (Sass).

A good project is made of consistent lines of code. Modern linters and formatters help to narrow the gap by defining a clear set of rules that are required for all the developers working on that project. These tools also help developers to write better code by showing common errors and enforcing good patterns. A formatter can enforce consistent maximum line lengths, assure it does not mix single and double quotes, add ending semicolon after each ending line of code, and help with other formatting concerns. This is especially crucial when working on a crew of developers with diverse backgro

More junior developers can adopt common practices by following the rules and everyone can be certain that the code stays consistent even when the practices change over time.

The next step in the procedure is testing and debugging. They are important processes during software development and maintenance. Testing focuses on finding issues while debugging's object is at solving the problem. With debugging, developers distinguish the issues in the application. Once the developer has fixed the bug, the tester will re-test to assure that the errors/bugs will be removed.

Prior to the stage when a web project is able to be deployed, several optimizations will be taken into concern otherwise the performance of the product will be affected, and consequently, the user experience will be negative. Considering there is no authority to provide official development tools, various third-party tools are needed to solve each piece of the puzzle. Although there are even more procedures in the process, the following steps are most notable and crucial: minification, compression, code splitting, and caching.

In summary, this thesis explores some of the most widely adopted front-end web libraries, frameworks, and utilities in the market. By analyzing their histories, benefits, and shortcomings, this thesis should answer the question as to why there are so many libraries, frameworks, and utilities in the field of front-end web development.

3 METHODOLOGY

This chapter explains the methods used in this thesis to archive its objectives by analyzing the issues presented when developing a website in the traditional web technology stack, and how some of the third-party tools are created to toggle these issues.

The issues will first be studied, classified, and divided into categories. At the beginning of each analysis on a specific category, the history of the subject involved will be given by Duong Dinh to provide context, then Zhuanyan Wang will further analyze the issues by providing case studies. Afterward, alternative approaches will be discussed to give a broader view of the subject involved.

3.1 Investigating and categorizing the issues

Considering the number of topics involved in front-end web development, it is not possible to cover all of them. This thesis will focus on discussing the issues and challenges presented when developing a website using only Vanilla HTML, CSS, and JavaScript. By dividing them into various categories based on which part of the development process they belong to, the thesis can dive deep into a specific issue without being overwhelming. In terms of categorizing the issues, the thesis starts at the most fundamental problems with HTML and CSS. Then the thesis discusses the issues with JavaScript in terms of DOM and modularization. Lastly, the thesis covers the development environment of the front-end web.

3.2 Providing historical context on each issue

To understand why such a large number of third-party tools exist in front-end web development, one must first inquire about all the issues these tools are created to solve. By providing historical context for these issues, readers should be able to grasp a better understanding of the reasons behind the appearance of all the libraries, frameworks, etc.

3.3 Providing case studies and observations for each issue

There is no better way to conduct in-depth analyses of an issue than providing case studies. For every issue, codes written in HTML, CSS, and JavaScript will be first provided to highlight the issue. Then, code snippets of a library, framework, or utility will be given to demonstrate how it approaches the issue.

The authors are fully aware of the extent of web development. For every issue HTML, CSS, and JavaScript have, there will be countless solutions created by a community of web developers to address it. To acknowledge the existence of numerous solutions without going too broad and generic, other alternative approaches will be discussed.

4 ANALYSIS OF THE PRIMARY TECHNOLOGY STACK

The analysis of the primary technology stack section dives deep into the challenges and issues presented by developing a website in Vanilla HTML, CSS, and JavaScript by first dividing the chapter into three parts: Repetition, DOM management, and Modularization. Challenges brought by Vanilla HTML, CSS, or JavaScript will be first presented, which is followed by a case study with code snippets that demonstrate how a specific technology approaches the challenges. At the end of some sections, alternative approaches will be discussed if there are any.

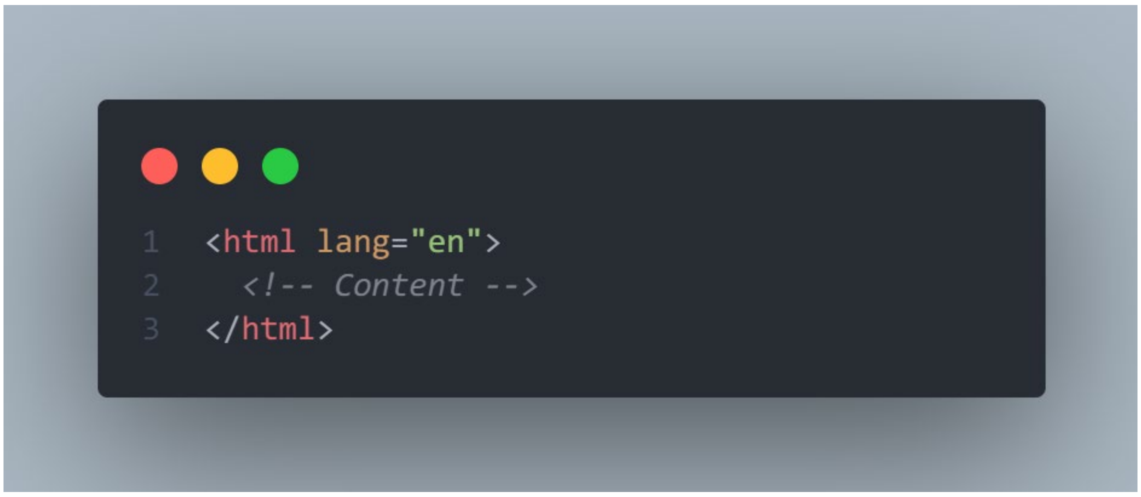
4.1 Repetition

4.1.1 HTML

HTML stands for Hypertext Markup Language which is the most fundamental building block of the World Wide Web (WWW). It was originally designed to be used as a markup language for scientific documents, but its overall design has enabled it to be adopted by other fields over the years (WHATWG, 2020).

Problems

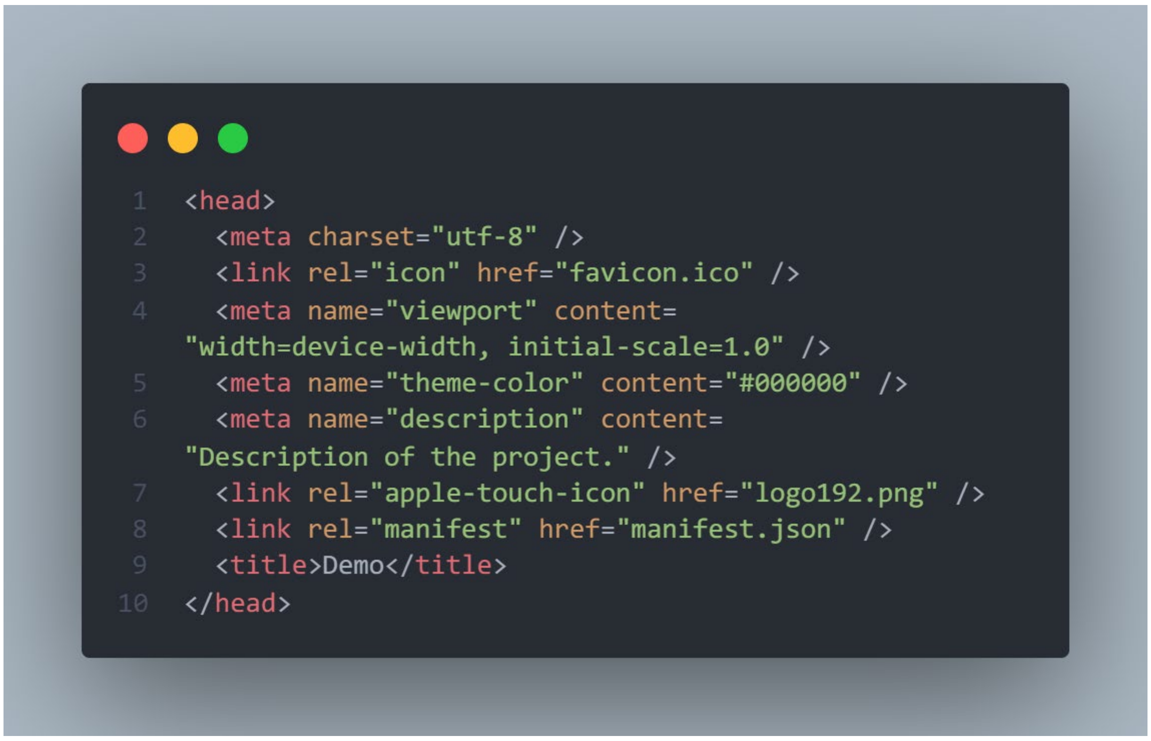
Although the syntax of HTML provides an intuitive way to inspect the structure and hierarchy of a web page's elements, the way it is designed to be used dictates that every single HTML file is served as an individual web page (WHATWG, 2020). For example, the display language of a website in most cases is consistent throughout all pages, but since one HTML file represents one page, the value of `lang` property inside `<html>` tags will be repeated in multiple HTML files.



```html
<html lang="en">
  <!-- Content -->
</html>
```

Figure 1. The display language can be defined inside the `lang` property.

As shown in Figure 1, the display language can be defined inside the `lang` property, but if the value of `lang` property is ever needed to be changed, then it is required to visit every single one of the HTML files and modify them one by one manually. Furthermore, the value of properties like `charset`, `viewport`, `theme-color` throughout a project will also likely to be the same, and thus will be repeated as well.



```html
<head>
  <meta charset="utf-8" />
  <link rel="icon" href="favicon.ico" />
  <meta name="viewport" content=
"width=device-width, initial-scale=1.0" />
  <meta name="theme-color" content="#000000" />
  <meta name="description" content=
"Description of the project." />
  <link rel="apple-touch-icon" href="logo192.png" />
  <link rel="manifest" href="manifest.json" />
  <title>Demo</title>
</head>
```

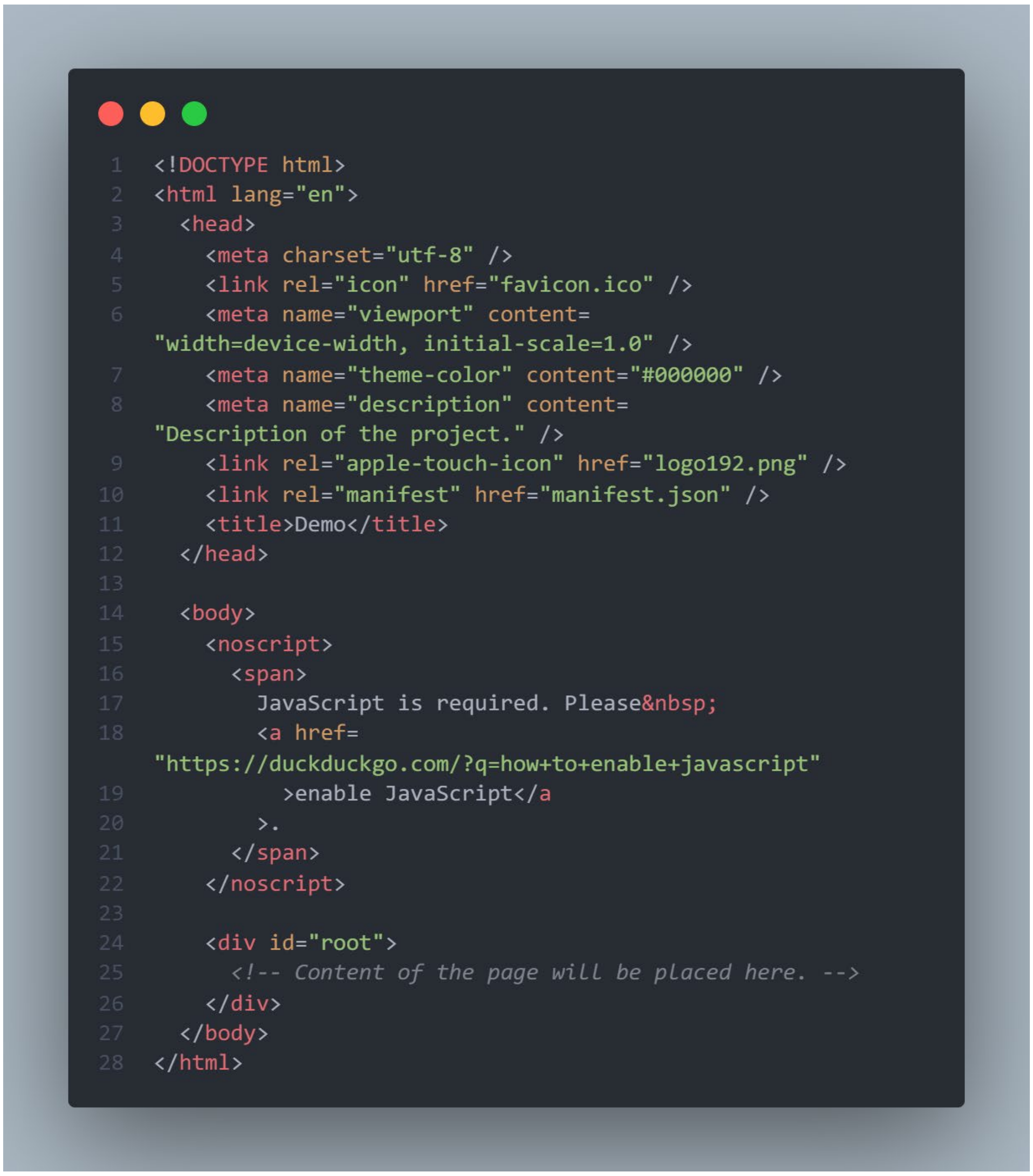
Figure 2. The content inside `<head>` tags of a HTML file.

As can be observed in Figure 2, the content inside `<head>` tags will likely to be repeated throughout all HTML files. Thus, it is not difficult to conclude that as the project grows in size, the same line of codes that will be repeated across all HTML files also increases, and the maintainability of such code base will be increasingly worsened.

Solution and case study: React

To toggle the repetition issue is to make sure that all settings for a website like `charset`, `description`, and `lang` are only required to define once in one place while ensuring only the content that needs to be changed like the `body` of an HTML file are updated. React, for instance, is one of the front-end web libraries that provide a solution to this problem.

React is a JavaScript library for building user interfaces (Facebook Inc., 2020). Its prototype, FaxJS was first created by Jordan Walke at Facebook (Walke, 2011), but later went open-sourced by the name of React in 2013 (Occhino & Walke, 2013).



```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <link rel="icon" href="favicon.ico" />
    <meta name="viewport" content=
"width=device-width, initial-scale=1.0" />
    <meta name="theme-color" content="#000000" />
    <meta name="description" content=
"Description of the project." />
    <link rel="apple-touch-icon" href="logo192.png" />
    <link rel="manifest" href="manifest.json" />
    <title>Demo</title>
  </head>

  <body>
    <noscript>
      <span>
        JavaScript is required. Please&nbsp;
        <a href=
"https://duckduckgo.com/?q=how+to+enable+javascript"
          >enable JavaScript</a
        >.
      </span>
    </noscript>

    <div id="root">
      <!-- Content of the page will be placed here. -->
    </div>
  </body>
</html>
```

Figure 3. An HTML template used by React.

As shown in Figure 3, there is only one HTML file in React and it is commonly referred to as the HTML template. How React works with HTML is to change only the parts that need to be updated while leaving other contents like `<head>` and `<noscript>` untouched. This means that users will always be on the same web page. When new content (or another web page in the traditional sense) is requested by users, instead of loading an entirely new page or HTML file from a server, the current page will update itself dynamically based on user interactions using JavaScript. This way, the repetition issue with HTML is resolved. Since this method has become so commonly adopted by web libraries and frameworks of all kinds, a new term called Single Page Application (SPA) is created to reflect their singular HTML file nature (Flanagan, 2006).

Alternatives to React

A single-page application works inside a browser and does not demand page reloading during use. Instead, it allows all essential HTML, JavaScript, and CSS code to be fetched by the browser with one single load, or only retrieves necessary resources and added to the page, normally to respond to end-user interactions. SPAs request the HTML file and data returned in independent action and later render webpage in the end-user. There are plenty of JavaScript frameworks such as Angular, Vue.js, Svelte, etc. which also provide the same infrastructure to achieve the same goal (Flanagan, 2006). Although previously mentioned frameworks allow developers to build SPAs, they have different approaches as well as principles.

Although previously mentioned frameworks have different ways of implementing their codebase, they all share a similarity; they all have a mixture of JavaScript and HTML codes. Considering React, it usually has one `index.html` file to connect with `index.js`, and `index.html` file has no contents besides containing necessary `<metadata>` and `<script>` tags. The reason is, in React, HTML lives within JavaScript file. The way React constructs make the logic and the style stay in one file. It helps to provide cleaner and more efficient codebase. However, the opposite happens with Angular: JavaScript codes live within an HTML file. Svelte and Vue also have the same way of approach and they all reach the same purpose: to remove the repetition.

4.1.2 CSS

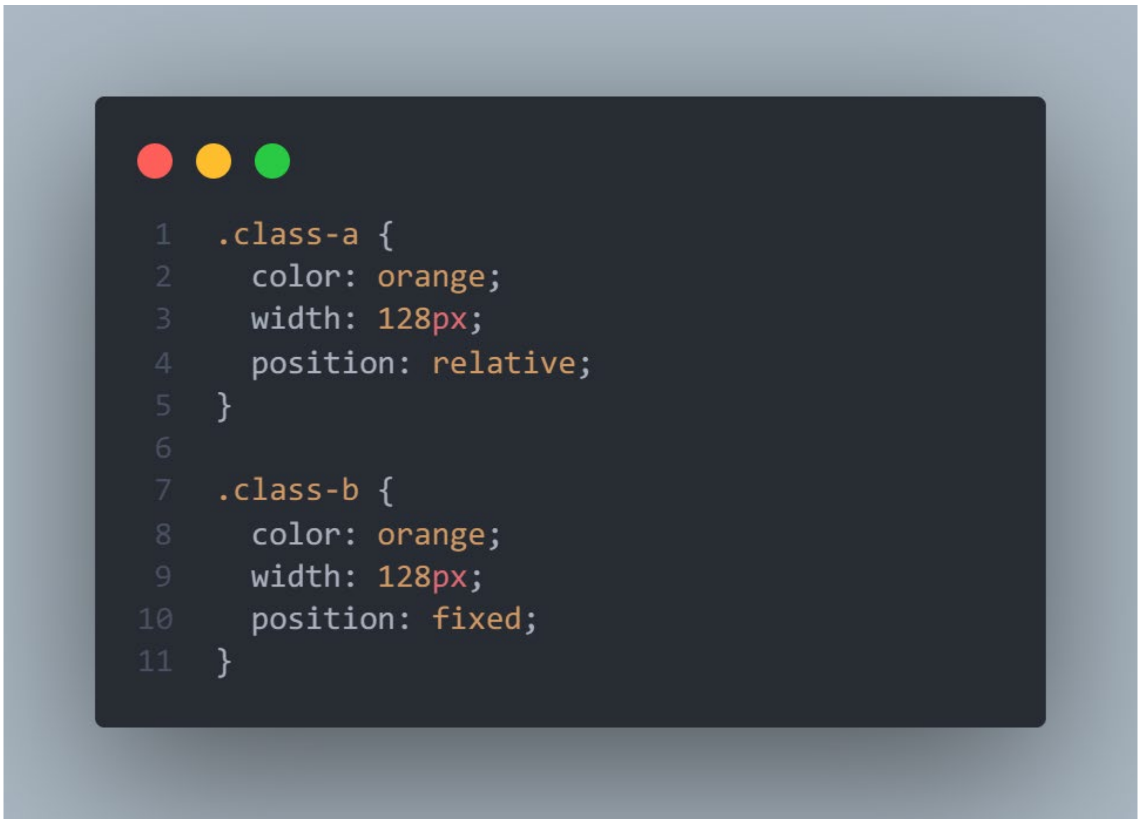
Cascading Style Sheets (CSS) was first introduced in 1996. At that moment, the current version of HTML was HTML2. HTML is responsible for the layout, the appearance, and the information input of the page. CSS is responsible for describing the presentation of a document written in a markup language like HTML. It is a fundamental technology of the World Wide Web, alongside HTML and JavaScript (CSS WG, 2020).

CSS was created to provide a tool for web designers to manage the style of HTML elements. Moreover, it was not intended to manage the visual look of webpages; it was only designated to access and control the properties of particular elements that existed in the HTML file. As the time CSS was created, most of the websites looked plain and architects could not anticipate the richness and complexity of the designs of the current state of webpages nowadays. The reason for that was clearly stated by one of the creators of CSS:

“Variables, constants, conditionals, expressions over variables, etc. are features that used most by programmers, however, they do not exist in CSS. These things give a lot of power to developers as they know how to customize and utilize these features to their best. However, for inexperienced users, they will feel more pressure, more likely, feel scared and stop using CSS” (Bos, n.d.).

Problems

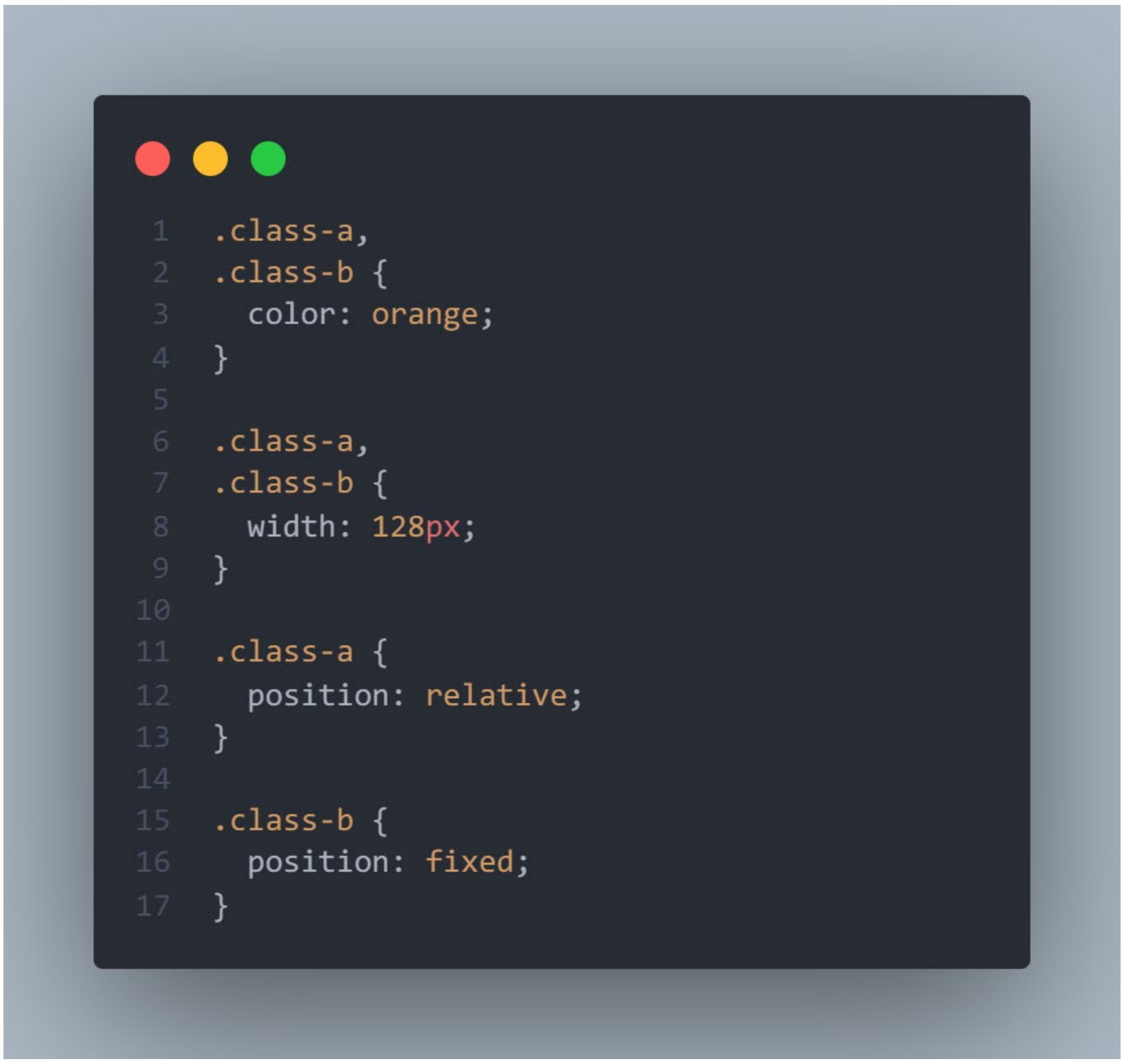
The original creators of CSS concerned with adoption. CSS was designed to be powerful enough for styling web pages and separating structure from visual while being easy to understand and use. Nevertheless, it is still rooted as a styling language, not a markup language. It offered a simple way for developers to control fonts, colors, and sizes, all in one file. Still, when it comes to styling the layout of a webpage, CSS cannot handle it adequately until the release of its 2.1 version (Mitton, 2015) (Bos, et al., 2016).



```css
.class-a {
  color: orange;
  width: 128px;
  position: relative;
}

.class-b {
  color: orange;
  width: 128px;
  position: fixed;
}
```

Figure 4. Two classes that share the same value for some properties.

As can be seen in Figure 4, there are two classes: `.class-a` and `.class-b`. These two classes share the same color and width but have a different position. Currently, codes for both color and width are one of the same thus they are repetitive. If the theme color or the width needs to be changed, the only way is to find all the `orange` and `128px`, then change them manually or use built-in features like find-and-replace from editors. Without using any third-party tools, one way to toggle this issue in CSS is to group the elements that share the same value for properties.



```css
.class-a,
.class-b {
  color: orange;
}

.class-a,
.class-b {
  width: 128px;
}

.class-a {
  position: relative;
}

.class-b {
  position: fixed;
}
```

Figure 5. Grouping classes that share the same property value.

Although the approach in Figure 5 makes the codes less repetitive, it severely worsens the readability and maintainability of the codebase. By having multiple `.class-a` and `.class-b`, it is also much harder now to keep track of which class has which property. If more elements and properties are added in the future, the situation will be further worsened.

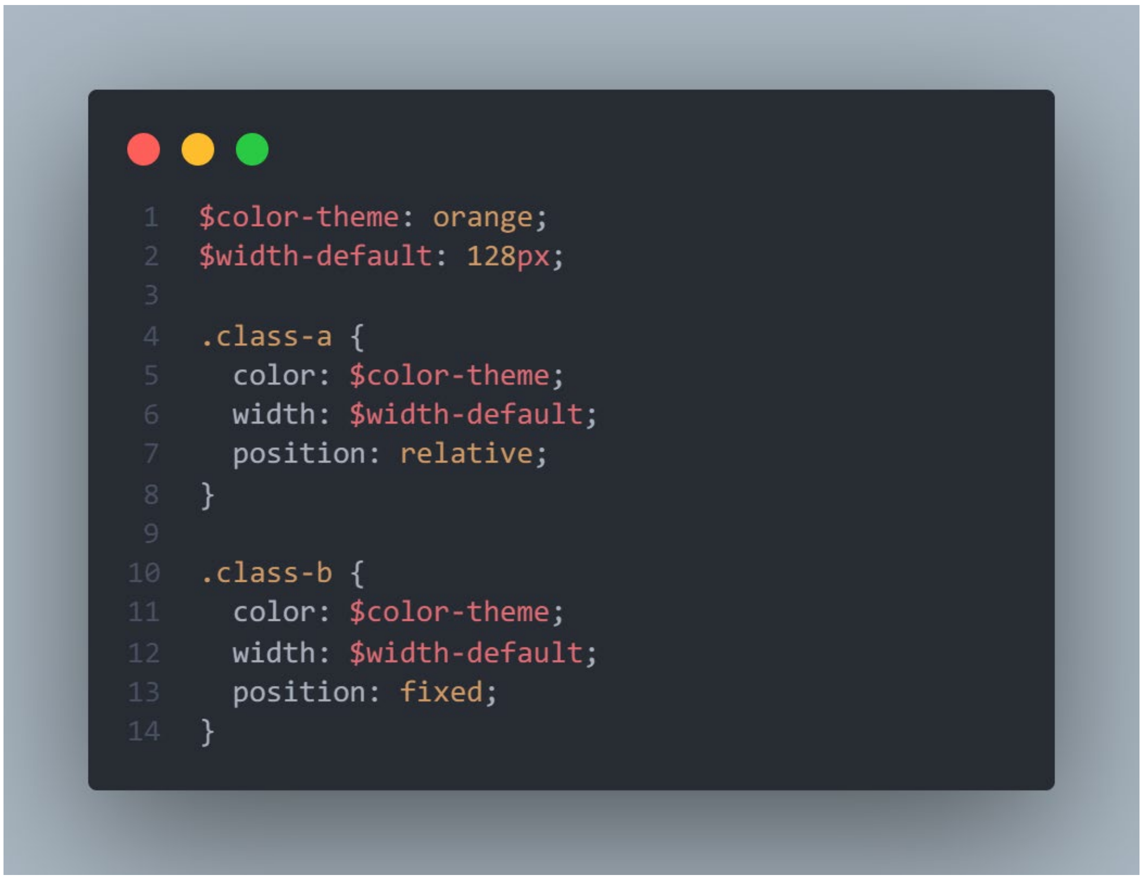
Solution and case study: Sass

To toggle this issue, the most obvious approach is to introduce the concept of variables and functions to CSS, as well as the concept of CSS preprocessors (Greif & Benitte, 2019).

A CSS preprocessor is a program that allows generating CSS from the preprocessor's unique syntax and providing features that are not currently available in CSS. Sass stands for Syntactically Awesome Stylesheets, it is a CSS preprocessor that allows developers to produce necessary lines of codes for writing CSS code. It also allows developers to make modifications more immediately since it doesn’t have to change duplicated selectors like the way CSS was doing.

Sass is one of the most commonly used CSS preprocessors in modern web applications (Greif & Benitte, 2019). A CSS preprocessor assists developers to write clearer and simpler to understand code. Sass takes input from `.scss` or `.sass` files, which are later compiled into regular `.css` files. The recently compiled CSS files are deployed to a browser to style the web application. Although CSS is becoming more robust, gradually embracing features that are currently available in preprocessors (variable is already available in CSS 3.0), it requires specific features, such as code reuse that makes it more challenging when it comes to "code maintenance" when the projects grow more significant and complex.

To achieve the same result in Figure 5 without the disadvantage of worse readability, Sass provides an easy solution.

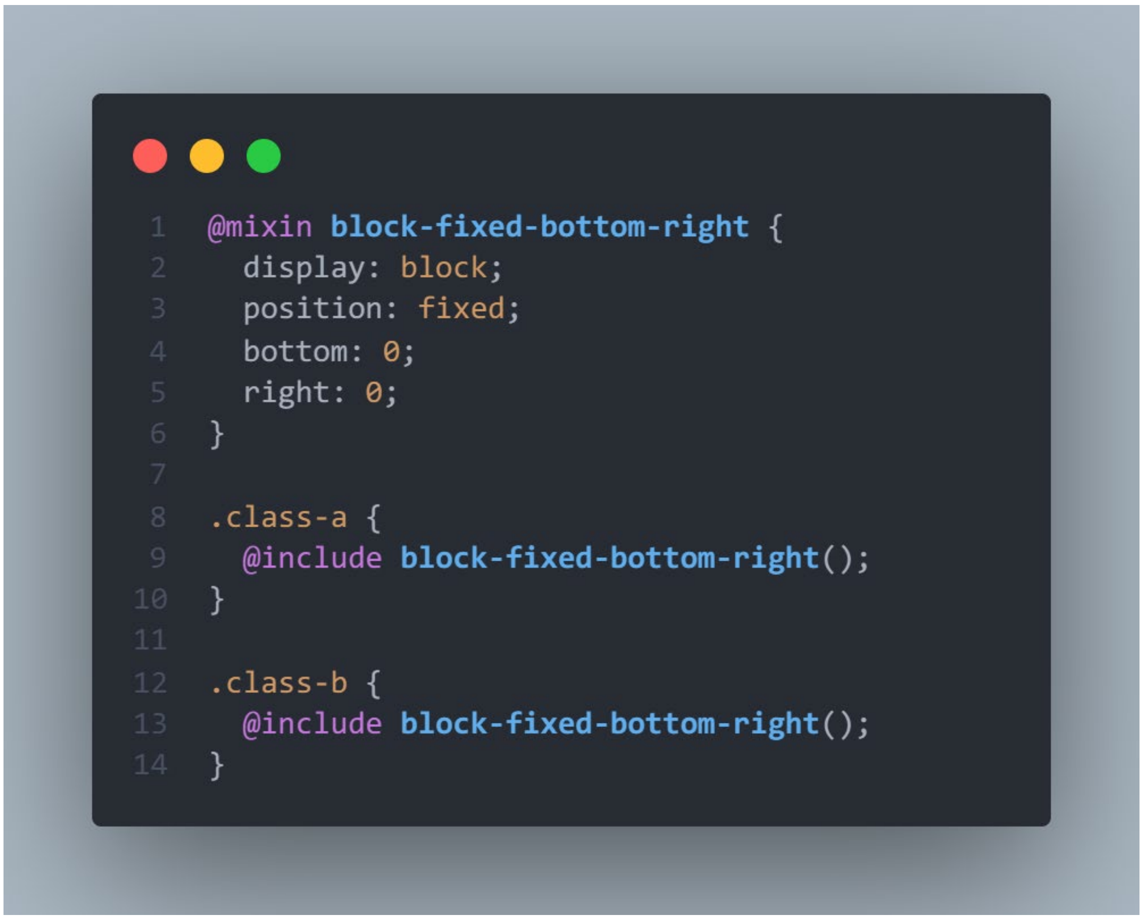


```scss
$color-theme: orange;
$width-default: 128px;

.class-a {
  color: $color-theme;
  width: $width-default;
  position: relative;
}

.class-b {
  color: $color-theme;
  width: $width-default;
  position: fixed;
}
```

Figure 6. Sass allows variables to be used in CSS.

As can be observed in Figure 6, now if a developer wants to search for `.class-a` or `.class-b`, the class name will appear only once. Similarly, if the theme color needs to be changed, one needs only to change the value of the variable `$color-theme` without the need to go through all the elements that have the theme color and change them one by one.

What if a block of CSS codes is shared by many elements? For example, what if two elements with different classes have the same settings for their positioning? SCSS can also handle such situations with ease.



```
@mixin block-fixed-bottom-right {
  display: block;
  position: fixed;
  bottom: 0;
  right: 0;
}

.class-a {
  @include block-fixed-bottom-right();
}

.class-b {
  @include block-fixed-bottom-right();
}
```

Figure 7. SCSS enables developers to reuse a block of CSS codes.

As shown in Figure 7, SCSS's `mixin` provides a way to reuse a block of CSS codes easily by using its `@mixin` and `@include` syntax. With variables and `mixin`, most of CSS's repetition issues are solved.

Alternatives to Sass

In terms of alternatives for Sass, there are plenty of CSS preprocessors available that help developers to achieve the same end. Less.js (usually referred to as a simpler form of Less), Stylus, PostCSS, Styled Components, etc. Although Sass and Stylus are the most widely adopted and versatile, there are still alternatives to be considered. These CSS preprocessors have common features and some of that vary between them. Most of the preprocessors will support variables, nesting, inheritance, conditionals, loops, functions, `@mixin` and `@include`. These innovative features make CSS preprocessors more object-oriented-like and allow programmers to perform their ideas behind efficiently. Although each of these mentioned alternatives has its advantages and disadvantages. Developers must choose the most suitable for each project. Later in the Modularization section, a case study will be given to explore how SCSS can further improve and enhance CSS.

4.2 DOM management

The DOM is an acronym for Document Object Model, it is an API for HTML and XML documents. DOM represents the structure of web documents and the way a webpage is accessed and manipulated. With the Document Object Model, developers can manipulate the document as well as modify the structure. Anything placed in an HTML or XML file can be accessed, changed, deleted, or added using the Document Object Model. Everything in the DOM also represents a node. The document is the main node. All HTML elements represent element nodes, etc. (Robie, 2020).

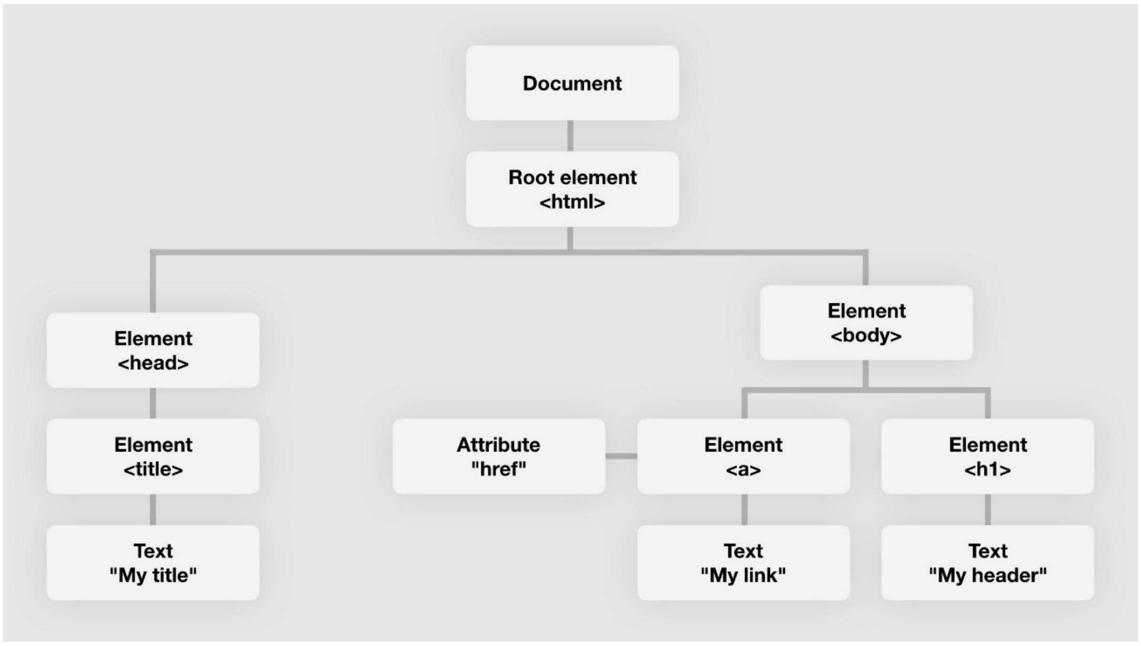


Figure 8. The tree-structure of Document Object Model.

As illustrated in Figure 8, the DOM is formed as a tree-structured - which is allowed by HTML document structure. This way, it allows traversing trees and among nodes more easily.

Problems

Nowadays in any web application, the DOM trees are huge and complicated. Since developers are more and more pushing towards dynamic web apps (SPAs), therefore a lot of modification is made with the DOM tree incessantly. This is bad in terms of performance and development. If there is a list of several items on the web page and with any user's interactions, it has to update every single one of them, the entire real DOM will be re-rendered. It's multiple times more effort than what is demanded. It does not only affect any other ongoing user's interaction on the web page but also has its defects on bandwidth.

Considering the DOM consists of hundreds of `div` elements. And there are lots of methods that handle events: clicks, submits, type-ins, etc. A typical JavaScript event handler will first target for every necessary node in an event and then update it if required. This used to be the method to access the DOM, but it arises two problems:

- It is difficult to maintain. For example, if there is an event handler that needs to be modified. If the developers who are doing the maintenance, lost the context, they have to dive deep into the codebase to even understand what those lines of code were meant to be. Both bug-risky and time-consuming.
- It is inefficient. These kinds of actions require more manual work. The way it should be doing is telling in advance that which nodes are to-be-updated.

Case study and solution: Virtual DOM

The case any application that has many of these operations in the background will render poorly. For websites like social media that need to fetch and send data frequently, the DOMs need to be able to update themselves extremely fast to keep the data displayed on the UI elements in sync with the database. For example, using the Twitter mobile site or Quora, as users scroll down the page, they will be shown a button print "show newer feeds". After a user starts scrolling for a while, they will have thousands of nodes added to the DOM. Interacting with those nodes in an efficient way is a huge problem. If one is trying to move a 1000 `div` 5 pixels to the left for instance, it probably takes more than a second. Considering the performance, this is a lot of time for the modern web application. Developers can optimize the script and apply some tricks, but in the end and for the long term, it's a mess to work with huge pages and dynamic UI. The similarity is seen on

Facebook, the advertisements, newsfeeds, stories, etc. change over time, comments update almost immediately to satisfy the interaction among all users. Needless to say, there are tons of DOM operations running behind the scene.

Although the DOM itself brought a lot of issues, however, there are also methods for developers to work things out. Currently, W3C group is working on something called the Shadow DOM.

The shadow DOM is W3C working draft standard. This specification creates a way of combining many DOM tree-structures into one hierarchy and manage how these trees interact with each other in a document, therefore it enables better composition of the DOM.

Another option is Virtual DOM. Virtual DOM is an abstract concept of the DOM. Virtual DOM solutions are built on top of standard DOM. They still utilize DOM eventually, but do it as little as possible and very efficiently. It is lightweight and for simplicity definition, the Virtual DOM is React's local and simplified version of the actual HTML DOM. Unlike the DOM nor the shadow DOM, the Virtual DOM is not an official specification, but a new method of communicate with the DOM.

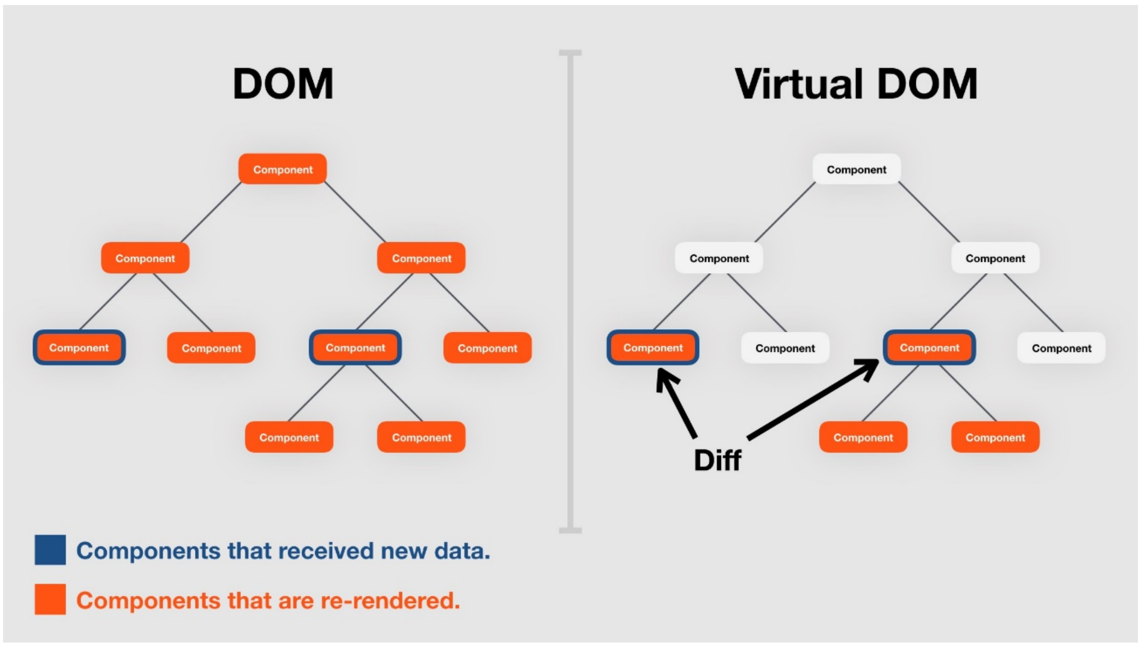

DOM
Virtual DOM
Component
Diff
Components that received new data.
Components that are re-rendered.

Figure 9. How DOM and Virtual DOM re-render a component tree.

Figure 9 shows that the difference between how DOM and Virtual DOM re-rendering updated components. Instead of re-rendering the whole DOM tree when components receive new data, Virtual DOM will re-render only the components that need to be updated.

When a page is rendered using Virtual DOM, the state of the DOM tree structure/hierarchy is stored, and when there any updates to be made to the UI, instead of writing the whole new tree, it does a “diffing” algorithm. React library is currently using the Virtual DOM which enables it to do computations within this domain and skip the real DOM operations. Each UI part is a component in React, and every component has a state. React.js will first listen and observe for state changes. When a component changes its state, React does the updates to the Virtual DOM tree, afterward it compares the current version with the former version of the Virtual DOM. This comparison process applies “diffing” algorithm. In this way, lots of DOM operations/refreshes are reduced, improving performance significantly. The whole process is called Reconciliation. (Facebook Inc., 2020)

Alternatives to Virtual DOM

There are two types of DOM, Virtual DOM, and real DOM. There are many front-end frameworks apply these two approaches as their principle. React applies the concept of Virtual DOM to update the application while Angular performs it differently using real DOM. Every component inside Angular has an associated change detector, which is generated at application startup time. When the change detection is called, for each function used in the component, it will compare the differences between the current and the previous value. The real DOM will be updated if there is a difference found. At the time being, Vue and React are the two well-known frameworks that apply the same rendering process called Virtual DOM. On the other hand, Angular and Svelte share the same philosophical technique, they change the DOM tree by accessing straight the DOM itself.

4.3 Modularization

Modularization is the process of separating a computer program into smaller, separate sub-programs. A module is a separate software component. It can often be used in a variety of applications and functions with other parts of the system. Related functions are classified in the same unit of programming code and non-related functions are developed as separate units of code so that the code can be reused by other applications.

4.3.1 Modules in ECMAScript

ECMAScript is a standard for scripting languages and JavaScript is an implementation of ECMAScript. The acronym ES refers to ECMAScript. ECMA is an abbreviation for European Computer Manufacturer’s Association; however, technically ES and JavaScript are virtually one and the same except that JavaScript may provide some additional features (Ecma International, n.d.).

JavaScript started as a client-side scripting language for developing web applications. To date, ES has published ten versions. The first three versions – ES1, ES2, ES3 were annual updates whereas ES4 was never released due to political disagreements. However, its prior version - ES5 contained limitations that did not allow developers to create classes, importing modules, etc. ES5 is referred to as the first standardized version of JavaScript. However, ES5 was not supporting for modules. The later version was introduced as ES6/ECMAScript - 2015/ES2015, it was the first update to the language since 2009. With this version released, JavaScript has made a big achievement by providing more syntaxes and reduce burdens of work for developers and partially reached the expectations of a modern programming language. The latter versions of JavaScript: ES7, ES8, ES9 ES10 also brought updates to the programming language, however, for the clarity and the scope of the topic, this part will not include it here. To perceive the importance of it, a full comprehension of the need for modules is required and what they provide (Ecma International, n.d.).

Problems

JavaScript was considered as a language for building small-scale websites; however, nowadays websites are evolving to web applications; code bases and functionality are growing to be immense. Developers demanded a certain way to distribute functionality and concurrently specify dependencies. Programming languages, like C++ and Java, rely on external libraries, for example, math library to perform a calculation with basic math equations without redefining and rewriting them. Accordingly, libraries allow programming languages to not replicate specific codes.

The similarity appears in what modules bring to JavaScript. Modules allow programmers to separate functionalities and organize the codebase more strictly. One of the biggest advantages of modules is because JavaScript's global namespace is undoubtedly easy to be polluted. In previous version, functions are the only way in JavaScript that generates scope, hence, anything not declared inside a function belongs to the global namespace. Consequently, every function declared in JavaScript is accessible globally, therefore its name cannot be declared again. This will consequently lead to a polluted namespace where names and code can be hard to locate and reuse. However, the global accessibility of a function is not always desired.

The module appears based on the idea that to make functions in a JavaScript file accessible to other JavaScript files, and vice versa. Modules allow developers to divide functionalities in applications, it helps programmers to manage dependency more efficiently. It enables code reusability, code refactoring and moreover, code extensibility.

ES5 is the backbone of every web application and supported by almost all of the current browsers. The shortage is, as it stated earlier, ES5 doesn't support modules. For that reason, various workarounds to support modularization have come up. In Vanilla ES5, there are two commons patterns to implement modules: Singleton pattern (IIFE - immediately invoked function expressions) and Constructor pattern. Besides Vanilla ES5, instead of relying upon it, third party tools brought advantages that ES5 could not.

ES6 brought various features and remove the limitations of ES5. ES6 currently has `class` syntax which makes Object-oriented programming simpler to implement. `Let` and `const` are new keywords to replace `var`. `Let` can be reassigned and `const` can only be assigned to one value once. And finally, ES6 added support for modules. Although ES6 has some new features, a lot of these are not compatible with most modern-day web browsers. Therefore, developers have come up with methods to bring features that they desired: tools so-called "trans-pilers" are being created to compile versions of ES6 code into older versions such as ES5 so that browsers can understand this newly created syntax. Most modern web applications are using these compilers so that the code will be compatible with ES6 when it is released but also compatible with current web browsers. Like other usages of modules, ES6 provides a way to export modules so that they can be specified as dependencies to other modules.

Considering the traditional web project that uses only Vanilla HTML, CSS, and JavaScript, all the files are usually placed inside their corresponding folder named after their format, but there is not an authority figure to provide an official guideline for web development project structure like that provided by mobile, or desktop platforms when developing software applications for them. This results in a lack of convention and good practices when it comes to structuring a web project.

How a traditional web project is structured varied, and the lack of convention worsens the situation. This results in different development teams have different way of structuring a web project. For new members of a team, it takes a long time to adapt to the way of doing things in the team.

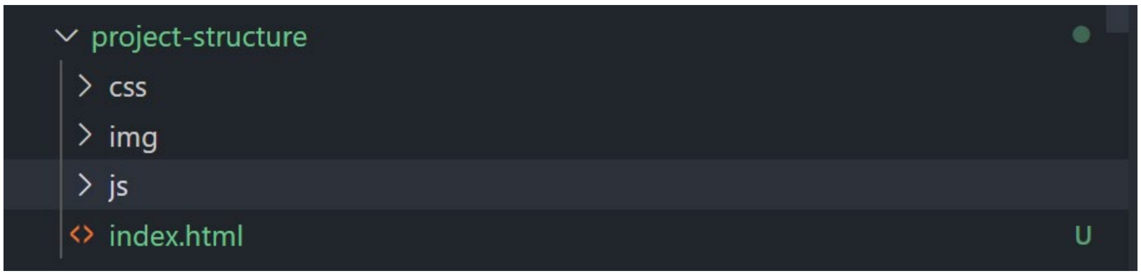


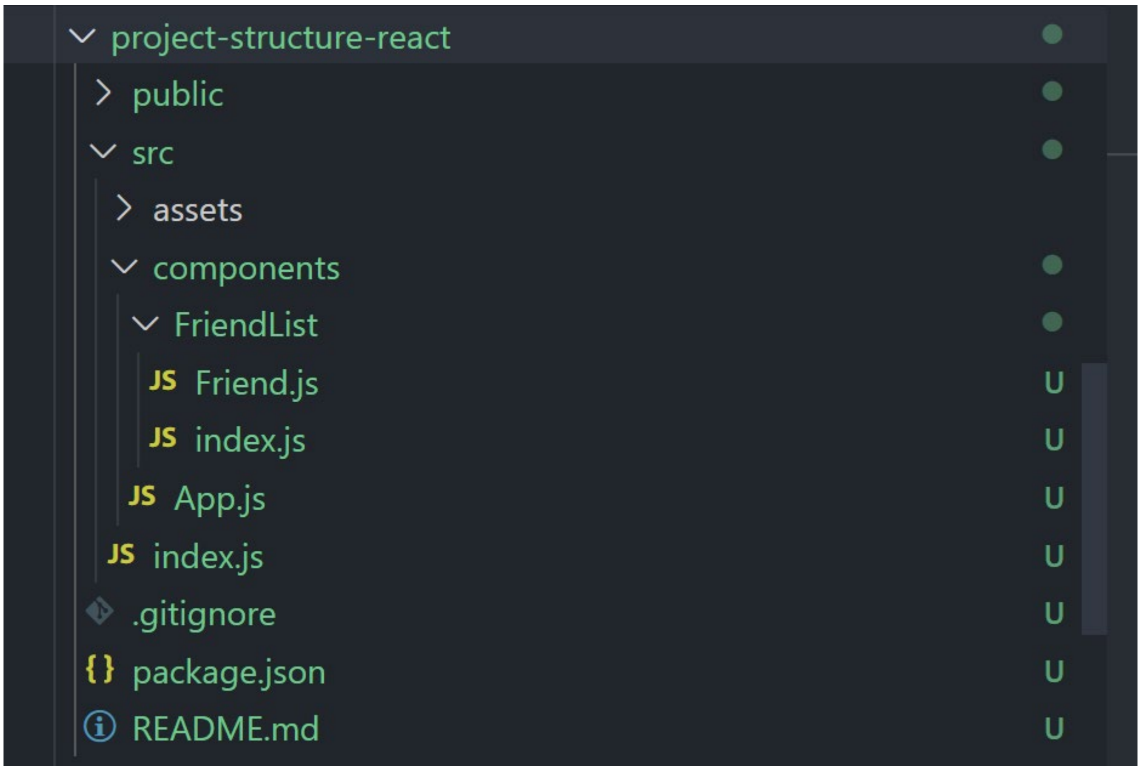
Figure 10. Grouping files based on their formats.

As presented in Figure 10, one common approach to modularizing a web project is to place the files that have the same format in their correspondent directory.

Solution and case study: React

In chapter 2, a case study of how React solves the issue with repetitive HTML codes is provided. In this section, more examples will be given to demonstrate how React organize and modularize a front-end web project.

Comparing to the traditional way of organizing files based on their formats, React.js provides a more sensible approach. In React, all elements of a web page are broken down into components that are updated dynamically based on their states. The codes for each component will be placed inside its respective file or folder, and the components are organized based on their parent and child relationship.



```
project-structure-react
  public
  src
    assets
    components
      FriendList
        Friend.js      U
        index.js       U
      App.js           U
    index.js           U
  .gitignore           U
  package.json         U
  README.md            U
```

Figure 11. Project structure of a typical React project.

In Figure 11, there are two folders `public` and `src` on the root of the project directory, the former contains the HTML template and other assets that are not dynamic such as favicons, and the latter holds all the source codes and assets that will be imported by the source codes.

On the root of the `src` directory, the entry file `index.js` that contains all components can be found. Within the `components` folder, `App.js` is the parent of all child components, and its child component `FriendList` has its folder with a corresponding name. As for `Friend`, a child component of `FriendList` is placed right next to `index.js` within the `FriendList` directory.

Implications of the React case study

Since all developers that have trained to write in React will learn this philosophy of structuring the project, it is also easier for any web development teams that use React to hire a new member. Before the age of web libraries and frameworks, a web developer does not know what to expect when applying for such a position, because a web project can be structured and organized in all kinds of ways. In some cases, the files might be placed inside folders that corresponding to their format, in other cases, the project team might have designed their internal framework to modularize the project since the philosophy behind the modularization methods of web libraries and frameworks are often universal (such as breaking down UI elements into components). Thus, it is not difficult to conclude that newly joined members of a web development team will take extra time in the past to familiarize themselves with the project in contrast to present time, if a team is using libraries like React nowadays, the project structure is much more predictable like the HTML template and static assets will be placed inside the `public` directory, and all the source codes and dynamic assets will be inside the `src` directory, etc.

Alternatives to React

Apart from React, how other libraries and frameworks modularize a front-end web project shares many things in common with the case study above. For instance, in libraries and frameworks like Vue, Svelte, UI elements are also broken down into components (You, 2019). Similarly, these approaches promote a specific way of organizing files as the best practice to provide a guideline for their adopters, but they do not force developers to comply with it blindly. Thus, instead of always grouping files that are associated with one component in the same directory, it is also common to place files that are correspondent to one route or feature under the same folder.

4.3.2 Separation of concerns

Separation of Concerns (SoC) design principle for separating a computer program into distinct sections, such that each section addresses a separate concern (Elliott, 2014).

Separation of concerns is defined as for each module or layer in a website or application should only be responsible for a certain thing and thus, should not contain codes that handle other functionalities. Separating concerns decreases code complexity by breaking down a large application into smaller units of encapsulated functionality.

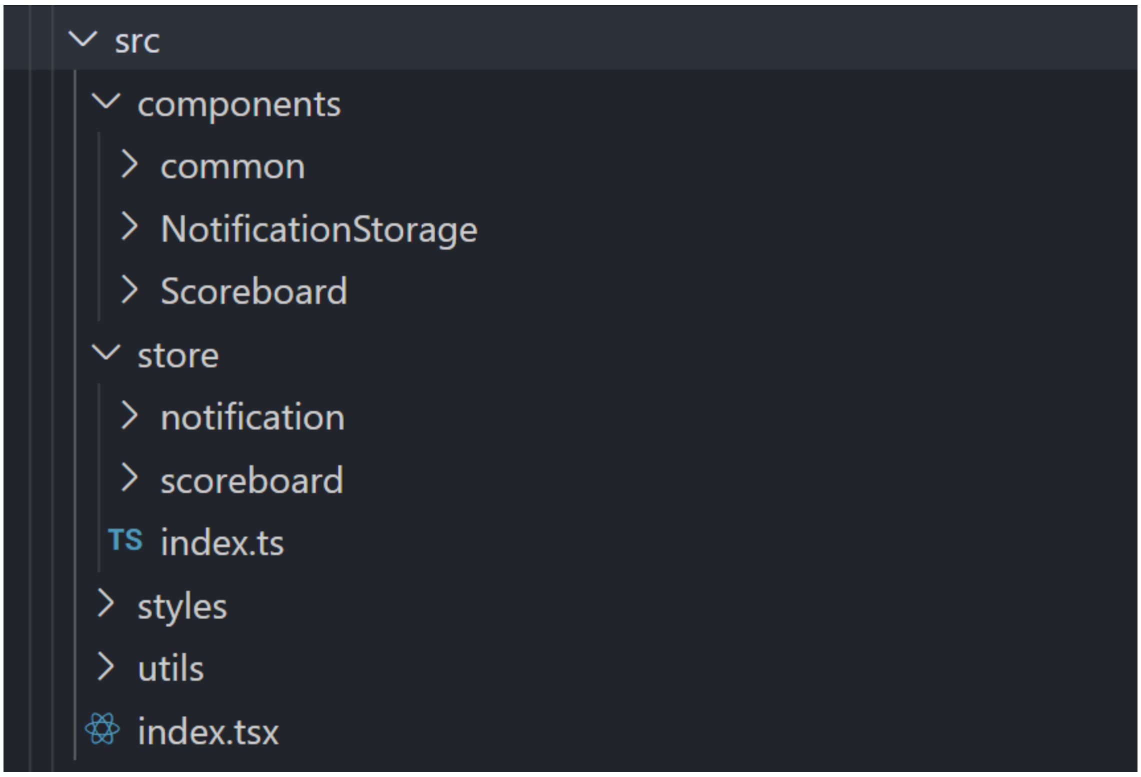
Problems

Separating presentation and logic when building web-based applications enables developers to create websites with dynamic content easier and faster. It also allows web designers who aren't especially experienced in application development to easily manipulate the appearance of a website. For websites with content that needs to modify regularly, advantage means that the updates can be achieved much more rapidly, therefore content will be brought to Web site visitors faster.

Early web-based applications were uncomplicated and often contained both presentation and business logic combined. This could lead to maintenance problems when changes occurred to either one of which. Separating two of the components apart simplifies maintenance and allows faster and easier updates.

Solution and case study: Redux

Simple application architecture in the context of a small demonstration will show how to achieve this separation and deploy and change Web applications faster.



```
src
  components
    common
    NotificationStorage
    Scoreboard
  store
    notification
    scoreboard
    index.ts
  styles
  utils
  index.tsx
```

Figure 12. A structure example of a React-Redux project.

As can be observed in Figure 12, in a typical React Redux project, the project structure is mostly the same as a React project. The difference is that within the `src` directory, the business logic which usually involves codes that dealing with how data is fetched and manipulated is placed inside the `store` directory while the components inside the `components` directory are now only in charge of presenting the data in a visually appealing way.

In such a project, the business logic is handled by Redux while the presentations are handles by React. Using React together with Redux will start to show its advantage as projects increase in size, as different parts of a project will be handled by a developer specialize in a specific area. For example, since the React components are now only concerned themselves with how UI elements are presented, so the developers that specialize in the design and visual aspect of the web can work on the React components without the need to worry about how states are managed and how data are handled.

Alternatives to Redux

A similar concept has already been widely adopted in software development, and it is referred to as the Model-View-Controller (MVC) paradigm. Business Logic is often broken further into two components in the Model-View-Controller (MVC) paradigm -- the Model, which defines how the data will be stored and retrieved; and the Controller, which controls the interaction between the Model and the View.

For simplicity and practicality, the thesis will not focus and make this distinction. In many cases, the distinction is unnecessary because the control flow pattern of the Model and the Controller is remarkably similar and conceived of at the same time. Presentation Logic means how it displays these objects to a user. This is often referred to as the View in the MVC paradigm.

It is often difficult to determine where business logic ends, and presentation logic begins. How developers model their application often dictates what is possible in the user interface. Sometimes the business and presentation logic are so intricate and dependent on each other that the two get intertwined with each other. Sometimes it is usually challenging or sometimes impossible to separate the two distinctly. In most cases, separating the two is just a matter of discipline.

5 ANALYSIS OF THE SECONDARY TECHNOLOGY STACK

The analysis of the secondary technology stack section concerns the tools that build to improve and automate the development process of the front-end web. After various libraries, frameworks, utilities, etc. are created to toggle the fundamental issues of HTML, CSS, and JavaScript, an equal number of tools are also created to improve the developer experience while building a front-end web project. In the following chapters, the missing parts will be divided into following categories: Managing dependencies, Source-to-source compiling, Code linting and formatting, Testing and debugging, Optimization for deployment.

5.1 Development environment

In software development, the development environment is a collection of processes and tools that are used to develop a source code or program. The term is synonymously used with an integrated development environment (IDE), which is the software development tool used to write, build, test and debug the program. It also provides developers with a common user interface (UI) to develop and debug in different sorts of modes. Commonly speaking, the term "development environment" would apply to the entire environment, while the IDE just applies to the local application that developers use to code. There is much overlap as they use an IDE for debugging just as use a development server to test.

Problems

Front-end web development’s key technology stack lacks a proper development environment from the very beginning because HTML is a markup language, CSS is a stylesheet language, and JavaScript is a scripting language. This means that even though browsers need to compile and execute them under the hood, but the time required is so minimal that they are more like running in real-time in practical terms. So, comparing to a traditional compiled programming language like C, they do not need to be compiled and built in a development environment before deploying to a web server or browser.

To make the matters worse, unlike developing native software applications for desktop and mobile platforms like Windows or Android, there is not an authority figure to provide an official development environment. For example, C# is mainly used for Windows platform, which is backed by Microsoft, thus Microsoft provides an official IDE called Visual Studio while Objective-C or Swift has an official IDE called Xcode which is maintained and developed by Apple.

Solution: Third-party tools

Thus, many private companies or individual developers have taken the matter into their own hands. Many of the editors like Visual Studio Code, Atom, and Subline Text are created to fill this void, even though they are technically not IDE, but with the help of various extensions and plugins, they are functionally similar to a traditional IDE like Visual Studio or Eclipse. However, even with tools like Visual Studio Code, many essential components of the whole development process are still missing such as dependency manager.

5.1.1 Managing dependencies

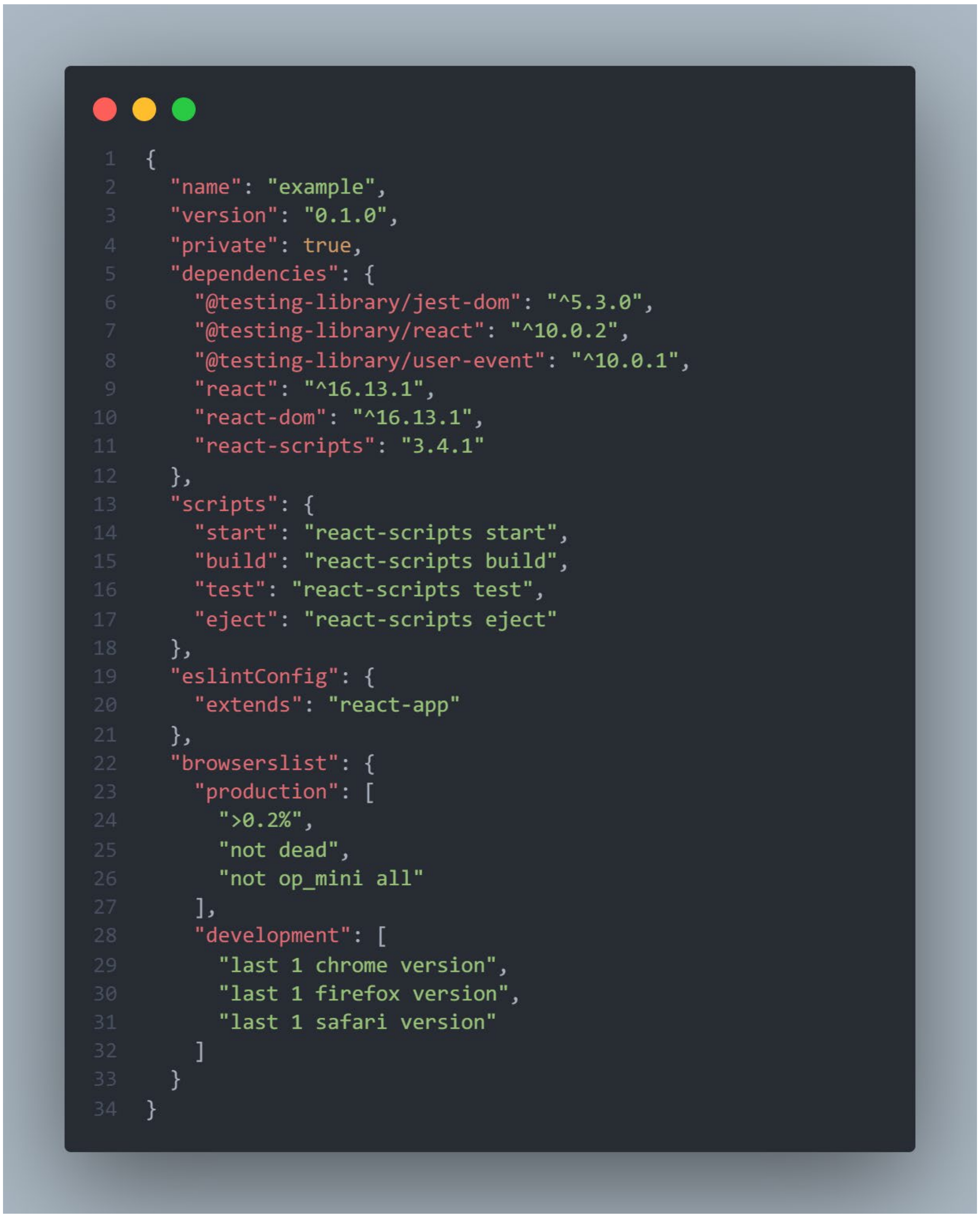
Problems

To set up a development environment that is missing from Vanilla HTML, CSS, and JavaScript template, a management tool is required to enable developers to perform basic interaction with all the dependencies quickly. Although a package manager with GUI will provide more user-friendliness, it will take much more effort and time and resources to produce and maintain, hence a package manager manages with command lines is the rational choice. Because none of HTML, CSS, or JavaScript can run in a command-line environment, something is needed to fill this gap.

Solutions: Node.js and NPM

Node.js allows JavaScript to run in command-line, and later Node Package Manager (NPM) becomes one of the most widely used dependency managers. Node.js gives the possibility to write applications in JavaScript on the server. It’s built on the V8 JavaScript runtime and written in C++ — therefore the speed is secured. Originally, it was intended as a server environment for applications, however, developers started using it to create tools to aid them in local task automation. Since then, a whole new ecosystem of Node-based tools like Grunt, Gulp, and Webpack has evolved to transform the face of front-end development. To make use of these tools (or packages) in Node.js, it is a must to be able to install and manage them in a useful way. This is where NPM becomes useful. It installs the packages developers want to use and provides a user interface to work with them. This following case studies will demonstrate how NPM and its `package.json` works.

Case study: NPM and `package.json`



```json
{
  "name": "example",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "@testing-library/jest-dom": "^5.3.0",
    "@testing-library/react": "^10.0.2",
    "@testing-library/user-event": "^10.0.1",
    "react": "^16.13.1",
    "react-dom": "^16.13.1",
    "react-scripts": "3.4.1"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "eslintConfig": {
    "extends": "react-app"
  },
  "browserslist": {
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  }
}
```

Figure 13. A `package.json` generated by `create-react-app`.

As demonstrated in Figure 13, a project that's using Node.js requires a `package.json` file which is usually generated by a library or framework. At its simplest, a `package.json` file can be represented as a manifest of the project that holds all the packages and applications it needs, information about the source control, and metadata like the project's name, description, and author, etc. Inside a `package.json`, metadata is almost always found specific to the project - no matter if it's a web application, Node.js module, or even just a plain JavaScript library. This metadata ensures the identify the project and serves as a baseline for users and contributors to get information about the project. A `package.json` file is always built in the JSON format, which allows it to be easily read as metadata and parsed.

Another important aspect of a `package.json` is that it contains a collection of any given project's dependencies. These dependencies are the modules that the project depends on to function properly.

Having dependencies in a project's `package.json` allows the project to install the versions of the modules it depends on. By running an install command inside of a project, developers can install all of the dependencies that are listed in the project's package.json - meaning never should be bundled with the project itself.

It also allows the separation of dependencies that are needed for production and dependencies that are needed for development. In production, it is likely not going to need a tool to watch CSS files for changes and refresh the app when they change. But in both production and development, developers want to have the dependencies that enable them to accomplish a certain task. If needing to format a `package.json` file manually to get the project up and running seems a bit daunting, there's a handy command that will automatically generate a base `package.json` file that is the `npm init` command.

The `npm init` command is a step-by-step tool to scaffold out a project. It will prompt out for input for a few aspects of the project in the following order:

- The project's name
- The project's initial version
- The project's description
- The project's entry point (meaning the project's main file)
- The project's test command (to trigger testing with something like Standard)
- The project's git repository (where the project source can be found)
- The project's keywords (basically, tags related to the project)

- The project's license (this defaults to ISC - most open-source Node.js projects are MIT)

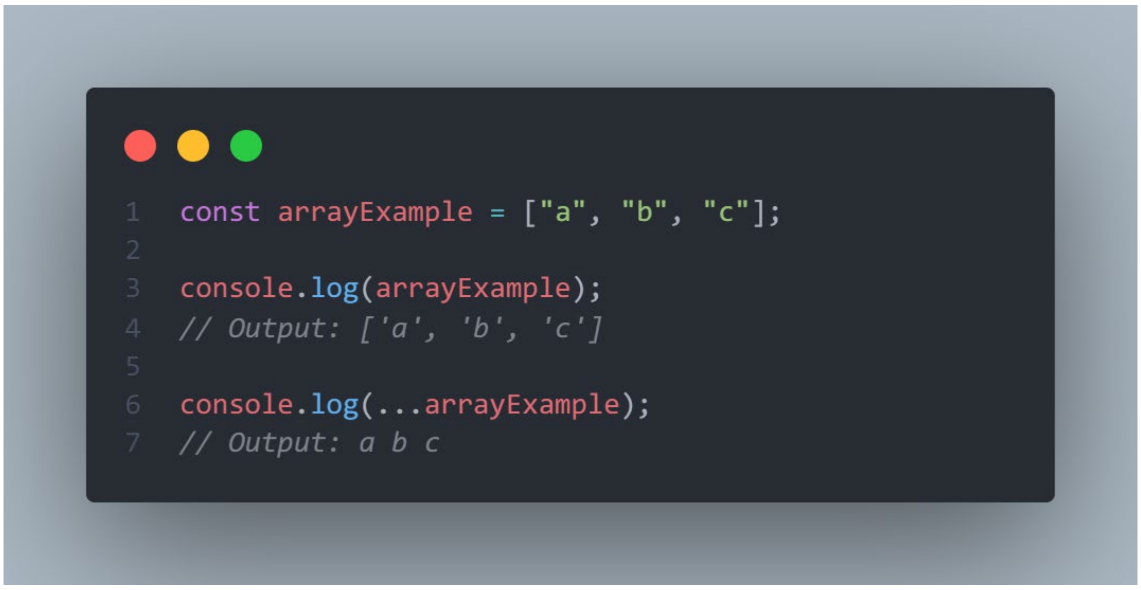
Once it runs through the `npm init` steps above, a package.json file will be generated and placed in the current directory.

5.2 Source-to-source compiling

5.2.1 Syntax compatibility

Problems with using the latest version of ECMAScript

Although HTML, CSS, and JavaScript are constantly evolving, users' browsers, however, may not support the latest iteration of those technologies if users have not yet updated their browsers. This creates a dilemma that on the one hand, the new features introduced in the latest version of HTML, CSS, and JavaScript greatly improves developers' experience while working on a project, but on the other hand, users' software may not support these new features just yet.



```javascript
const arrayExample = ["a", "b", "c"];

console.log(arrayExample);
// Output: ['a', 'b', 'c']

console.log(...arrayExample);
// Output: a b c
```

Figure 14. Spread operator introduced in ECMAScript 6.

For instance, as shown in Figure 14, the spread operator from ECMAScript 6 helps developers to iterate and clone an array or string by simply placing three dots at the beginning of the array or string without the need to write a lengthy customized function or install third-party utility libraries such as Lodash (Ecma International TC39, 2014).

Solution: Source-to-source compiling

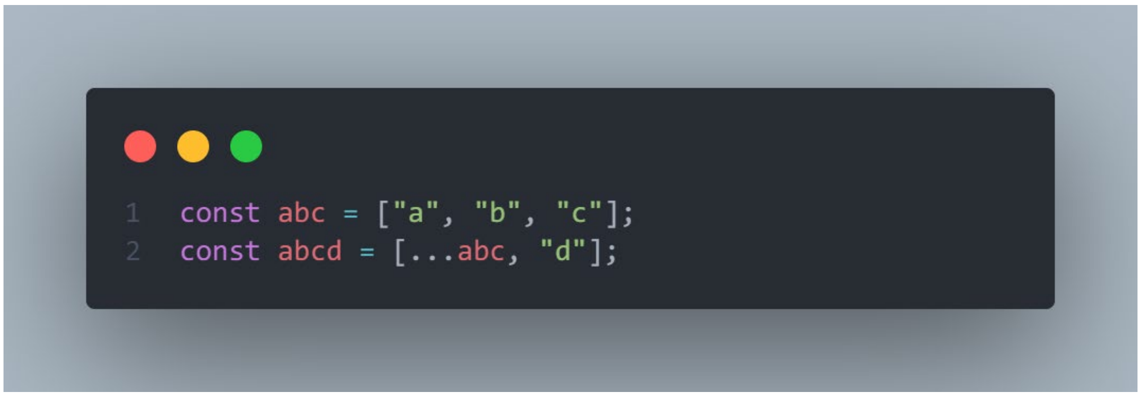
To allow developers to use these convenient new syntaxes without sacrificing the compatibility of the product, source-to-source compiling is the solution. A source-to-source compiler is a kind of compiler that receives the code of a program written in one language as its input and returns the similar source code in the same or another language to its output. A source-to-source compiler interprets between programming languages has the same level of abstraction (Aho, et al., 2007).

JavaScript keeps constantly evolving. Numerous versions of JavaScript were implemented which added updates to syntax and functionality to the language. These updates make JavaScript more comfortable to code with and reduce difficulties of developers in general. Nevertheless, these features of JavaScript have been around for a few years and steadily adopted by almost developers, therefore, browsers also need to update in order to meet these changes. For a newer version of JavaScript code runs in a not-supported-browser (Internet Explorer), the code will not function properly.

One of the problems every web developer is constantly facing is that a lot of time-saving new features from the latest iteration of JavaScript are often only supported by the latest versions of a browser. For instance, numerous features from the ECMAScript 6 standard are not supported by the legacy browsers like Internet Explorer 11 or the simplified version of browsers like Opera Mini (Zaytsev, 2020).

Case study: Babel and ECMAScript

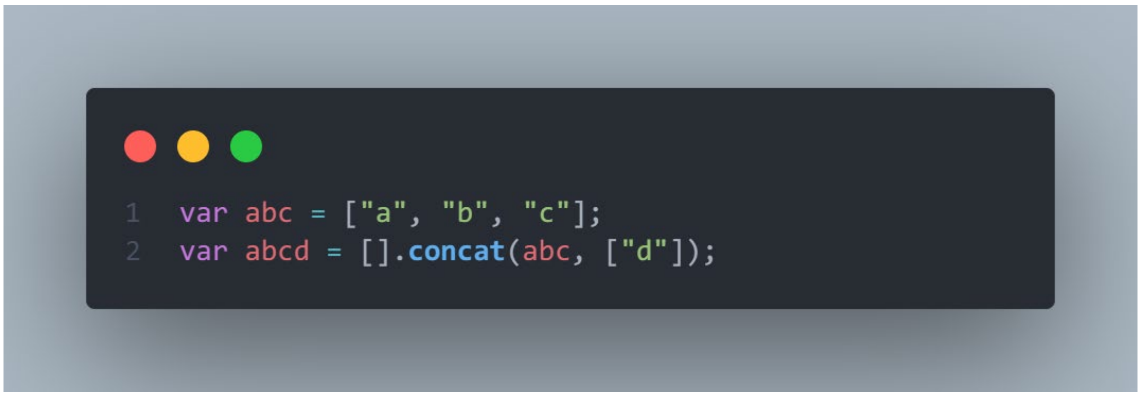
To allow developers to use the latest features of JavaScript language while keeping the legacy browsers compatibility, a source-to-source compiler is needed. Among the source-to-source compilers for JavaScript, one of the most well-known is Babel. To illustrate, the spread operator syntax from ECMAScript 6 mentioned in the previous section will be used as the input for Babel.



```
const abc = ["a", "b", "c"];
const abcd = [...abc, "d"];
```

Figure 15. Passing codes that use spread operator to Babel.

As demonstrated in Figure 15, three alphabets `a`, `b`, `c` are assigned to an array called `abc`, and then the array `abc` is concatenated with a new alphabet `d` to form a new array called `abcd`. The use of spread operator here allows developers to concatenate arrays and strings intuitively by simply adding a comma between the array and the new item, and it also ensures that the original array `abc` will never be mutated.



```
var abc = ["a", "b", "c"];
var abcd = [].concat(abc, ["d"]);
```

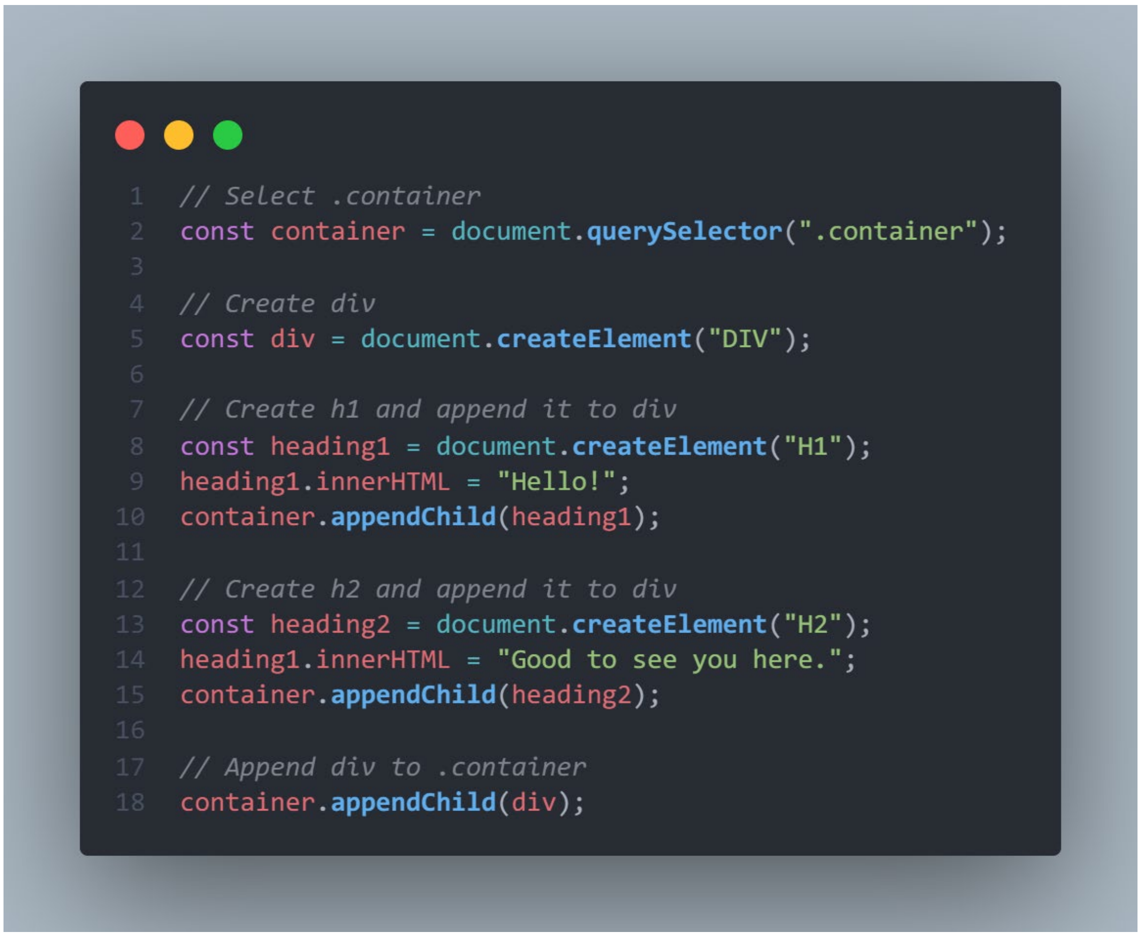
Figure 16. Babel translates new syntaxes to the old ones.

As can be depicted in Figure 16, after Babel received the codes in Figure 15, the first noticeable change happened to `const` as they are now replaced with the older syntax `var` (Deveria & Schoors, 2020). As for the spread operator, it is used for concatenation in this case, thus Babel uses its equivalent in the older version of JavaScript `concat` to achieve the same effect while keeping the codes compatible with old browsers (Mozilla and individual contributors, 2020).

Case study: Babel and JSX

Apart from translating the syntax from the latest ECMAScript, Babel can also be used to compile language extensions for JavaScript like JSX and TypeScript.

JSX is an extension to the JavaScript language syntax. Similar in appearance to HTML, JSX provides a way to structure component rendering using a syntax familiar to many developers. React components are typically written using JSX, although they do not have to be (components may also be written in pure JavaScript).

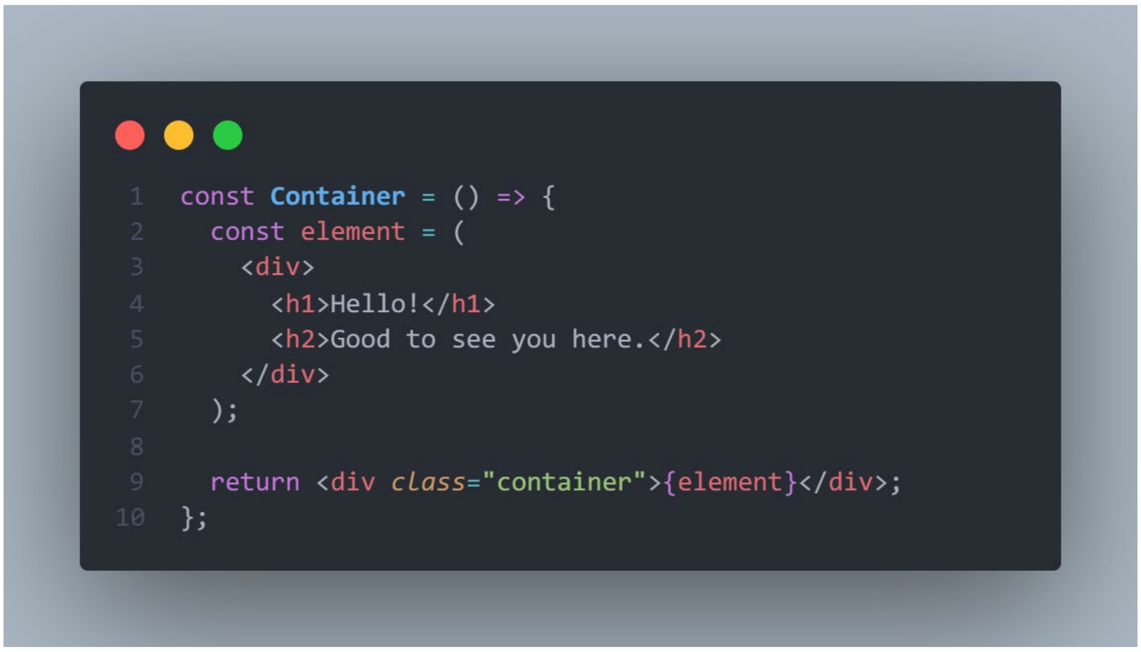


```javascript
// Select .container
const container = document.querySelector(".container");

// Create div
const div = document.createElement("DIV");

// Create h1 and append it to div
const heading1 = document.createElement("H1");
heading1.innerHTML = "Hello!";
container.appendChild(heading1);

// Create h2 and append it to div
const heading2 = document.createElement("H2");
heading1.innerHTML = "Good to see you here.";
container.appendChild(heading2);

// Append div to .container
container.appendChild(div);
```

Figure 17. DOM manipulations in Vanilla JavaScript.

Figure 17 demonstrates how to create a `<div>` element that contains a `<h1>` and a `<h2>`. It clearly shows that without the intuitive syntax of HTML, the codes that are required to create these simple elements are difficult to read and maintain.

With React JSX, an HTML element can be assigned directly to a JavaScript variable while keeping the intuitive HTML syntax and proper syntax highlighting. To place this HTML element to its intended location, one only need to wrap the variable it has been assigned to inside brackets, then placing the variable directly inside HTML. When compiling React projects, Babel will translate all codes to Vanilla JavaScript.



```
const Container = () => {
  const element = (
    <div>
      <h1>Hello!</h1>
      <h2>Good to see you here.</h2>
    </div>
  );

  return <div class="container">{element}</div>;
};
```

Figure 18. DOM manipulations in React JSX.

As can be seen in Figure 18, creating a `<div>` element with a `<h1>` and `<h2>` is so much simpler in React, because React JSX allows HTML syntax to be used alongside JavaScript syntax.

5.2.2 Tying system

Problems with not having a tying system in JavaScript

When JavaScript was developed, the development team introduced JavaScript as a client-side programing language. However, when using JavaScript, developers get to know that it can be handled as a server-side programming language also. However, when the language began to grow, the code of JavaScript became complex and heavy. Due to this, JavaScript was not ready to fulfill the requirement of Object-oriented programing language. This limits JavaScript's capability to reach the enterprise level as a server-side technology.

Solutions: Source-to-source compiling and TypeScript

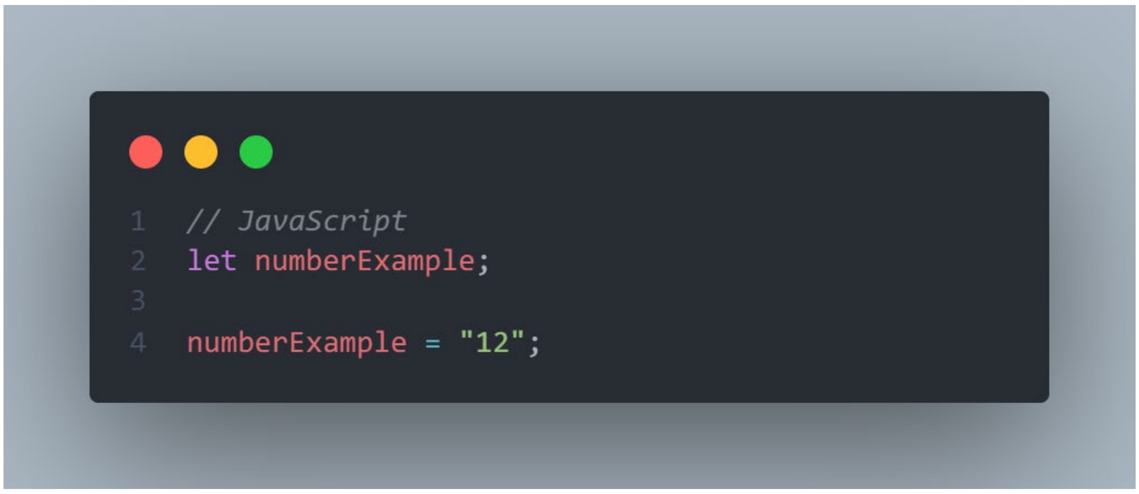
TypeScript was developed by the development team to bridge this gap (Microsoft Corporation, 2020) and it will not be made possible without the support of source-to-source compiling. TypeScript is used for creating large applications. When Microsoft first created TypeScript, it came with a motto: “JavaScript that scales.” It is simpler to manage a large codebase with TypeScript for it simplifies JavaScript code. TypeScript also consists of tools to increase the efficiency of developers and it has a strong type system.

TypeScript is very useful for debugging efforts. With JavaScript, it is time-consuming to find bugs for it is an interpreted language meaning that developers have to actually run the code to discover bugs, which requires a lot of time. But inside TypeScript, there is a compiler that automatically checks the code for any problems. If it sees any errors, it will then create compilation error messages, which helps programmers to look into the bugs before the code is run.

JavaScript is dynamically typed. This means JavaScript does not know what type a variable is until it is instantiated at run-time. TypeScript adds type support to JavaScript. Bugs that are caused by false assumptions of some variable being of a certain type can be completely eradicated (how strict developers type the codes is fully their options).

Case study: TypeScript

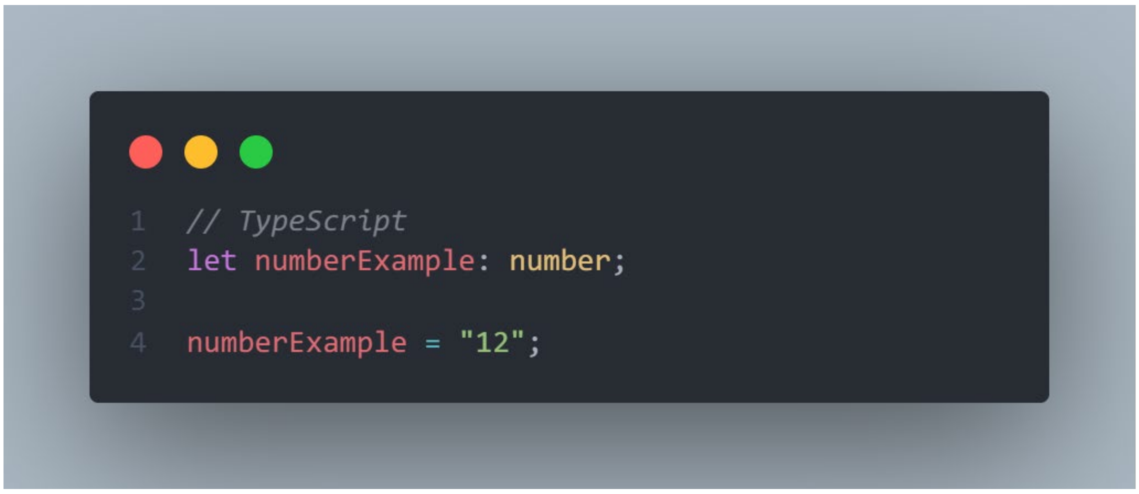
TypeScript makes the typing process becomes simpler and less explicit with type inference. The "type" in TypeScript is directly referred from its use. Even though if it does not type the types explicitly, they are still there to keep developers from doing something which consequently will result in a run-time error.



```
// JavaScript
let numberExample;

numberExample = "12";
```

Figure 19. JavaScript does not provide a type system.

The case shown in Figure 19 does not seem to matter that much as there are only four lines of codes, but as the project grows in size, hundreds of thousands of variables will be added to the project and they will be used by people other than their creators. When one developer misuses a variable, the worst-case scenario is that all UI elements might still be rendered without errors, but some of which might not be rendered in a way that it is supposed to. In this case, it will be extremely difficult to debug which specific line of codes is at fault because JavaScript will only throw errors when there is something wrong with compilation. To avoid this, TypeScript has brought a static typing system to JavaScript.



```
// TypeScript
let numberExample: number;

numberExample = "12";
```

Figure 20. TypeScript will throw errors if a variable is misused.

If `numberExample` is assigned to a string as demonstrated in Figure 20, TypeScript will immediately warn developers of this mistake. Because by simply adding `: number` after the `numberExample` variable declaration, TypeScript will remember that this variable is intended to be a number.

5.2.3 Preprocessor

Source-to-source compiling enables other possibilities

Lastly, apart from making TypeScript or using the latest ECMAScript syntax possible, source-to-source compiling is also used in other parts of front-end web development such as CSS. The benefits of having a CSS preprocessor like Sass are already discussed in the previous sections, similar to Babel and JavaScript, these preprocessors will also translate the codes written in CSS language extensions like SCSS to Vanilla CSS when compiling a project.

5.3 Code linting and formatting

Problems with codes consistency and accuracy

Since the developers may use different editors to work on the same project especially if the project is an open-source one, there needs to be a solution to ensure that no matter what text editors or IDEs a team member is using, their codes will always be properly checked and formatted whenever they are committing the codes to a version control system. To achieve this, tools for code linting and formatting are required.

Solutions: Linter and formatter

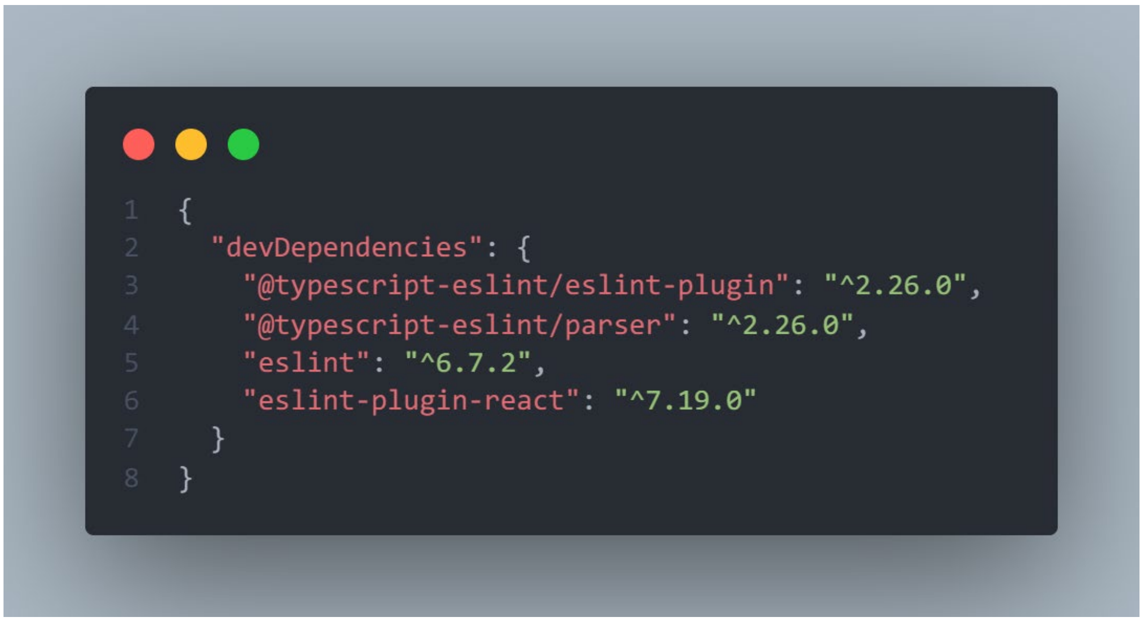
Linting, or a linter, is a tool that examines the source code to flag potential programming errors like bugs, stylistic errors, and peculiar constructs. This term originally comes from a Unix utility that examined C language source code (Johnson, 1978).

Code formatting or Formatter helps application remove all original code styling input and secure that all outputs follow to a consistent style as well as increase the readability (Prettier.io, 2020).

Case study: ESLint

The following case study will demonstrate how certain technologies approach linting and formatting for a web project. Linting for a specific programming language is simple, but when there are various technologies involved, things become much more complex. For example, in this particular case study, React, TypeScript, and Sass are used.

One of the most widely used linter for JavaScript in the market currently is ESLint and the case study is using TypeScript, a language extension for JavaScript and together with React which has its syntax like JSX. Luckily, the ESLint provides plugins to allow all these technologies to work together. Inside `package.json`, a file that allows NPM or its alternatives to keep track of which dependency is needed, there are four dependencies.

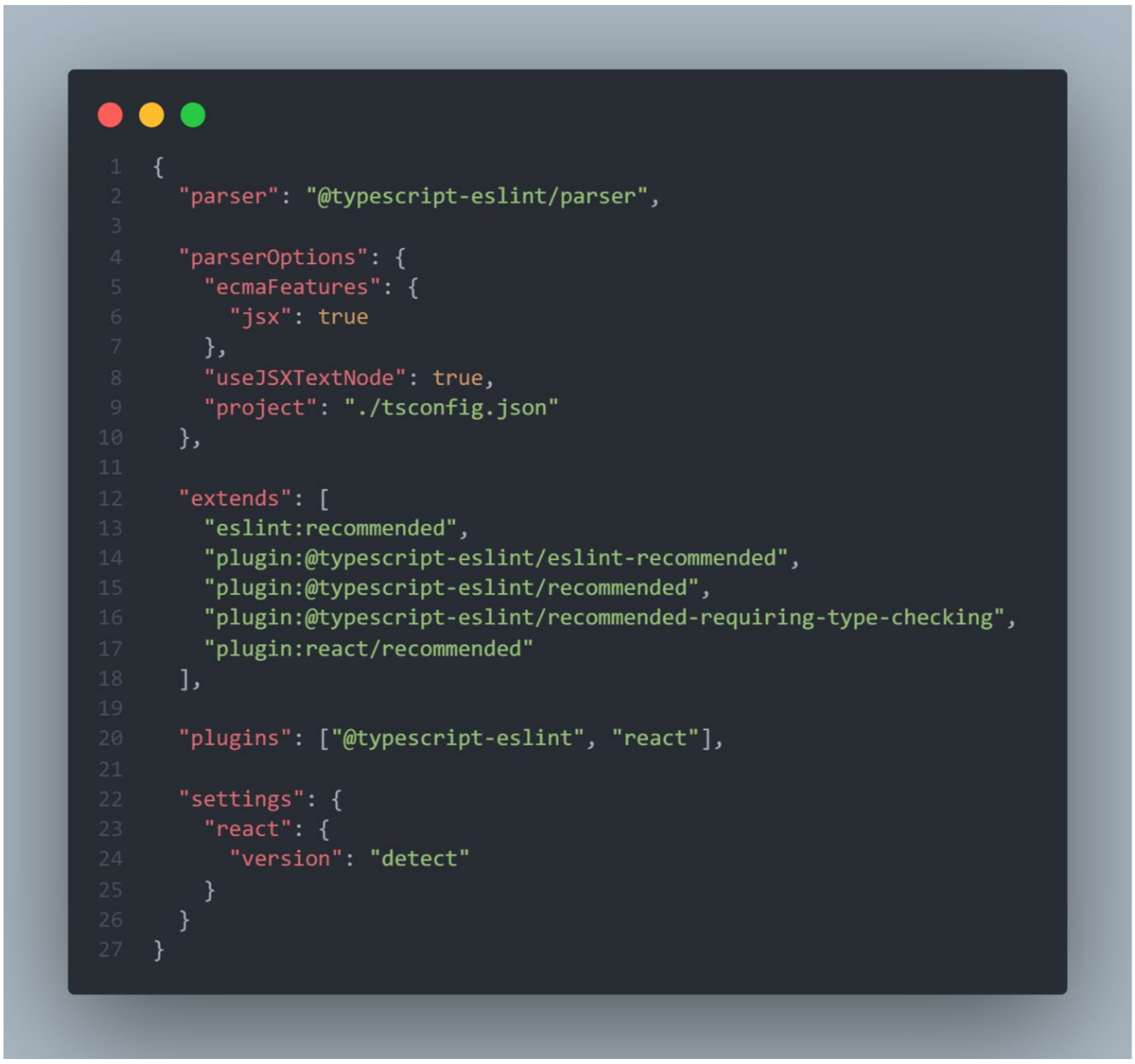


```json
{
  "devDependencies": {
    "@typescript-eslint/eslint-plugin": "^2.26.0",
    "@typescript-eslint/parser": "^2.26.0",
    "eslint": "^6.7.2",
    "eslint-plugin-react": "^7.19.0"
  }
}
```

Figure 21. ESLint plugins as `devDependencies` in `package.json`.

Figure 21 shows the plugins used by ESLint to support React JSX and TypeScript. Both `@typescript-eslint` are there to enable ESLint to support TypeScript, and the `eslint-plugin-react` is an ESLint plugin that adds React specific linting rules for ESLint.

The default configuration file for ESLint is called `.eslintrc`, customized rules and settings can be added to tweak ESLint to meet the specific needs of a project.



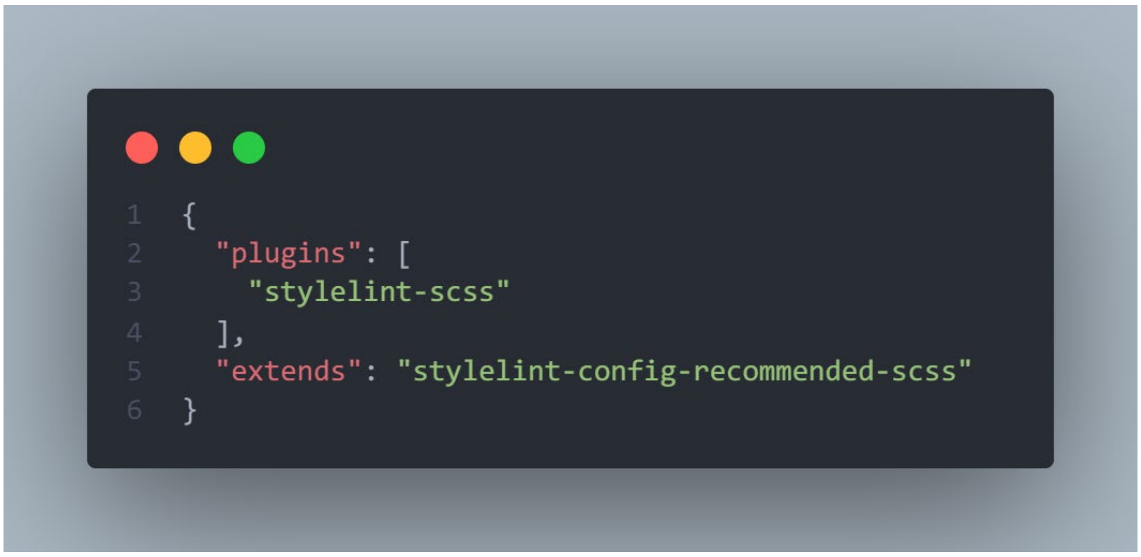
```json
{
  "parser": "@typescript-eslint/parser",

  "parserOptions": {
    "ecmaFeatures": {
      "jsx": true
    },
    "useJSXTextNode": true,
    "project": "./tsconfig.json"
  },

  "extends": [
    "eslint:recommended",
    "plugin:@typescript-eslint/eslint-recommended",
    "plugin:@typescript-eslint/recommended",
    "plugin:@typescript-eslint/recommended-requiring-type-checking",
    "plugin:react/recommended"
  ],

  "plugins": ["@typescript-eslint", "react"],

  "settings": {
    "react": {
      "version": "detect"
    }
  }
}
```

Figure 22. The configuration file for ESLint by default is called `.eslintrc`.

Figure 22 showcases some of the configurations supported by ESLint. The parser from `@typescript-eslint` is used to interpret TypeScript syntax, and inside the options of the parser, JSX is set to true to support linting for React. Inside `extends`, linting rules are defined to allow ESLint to check all JavaScript, TypeScript, and JSX codes based on specific preferences. The two ESLint plugins discussed previously, `@typescript-eslint` and `react` are specified in the `plugins` section. The `settings` object allows shared settings to be added which will be supplied to every rule defined in the `extends` object. Lastly, inside the `rules` object, it is allowed to overwrite some of the rules that do not fit the project.

Case study: StyleLint

While ESLint is used for JavaScript, TypeScript, and JSX, a linter called StyleLint is used in this case study to support linting for all stylesheets like CSS, Sass, Less, etc. Similar to the configuration file of ESLint, for StyleLint, it is called `.stylelintrc`.



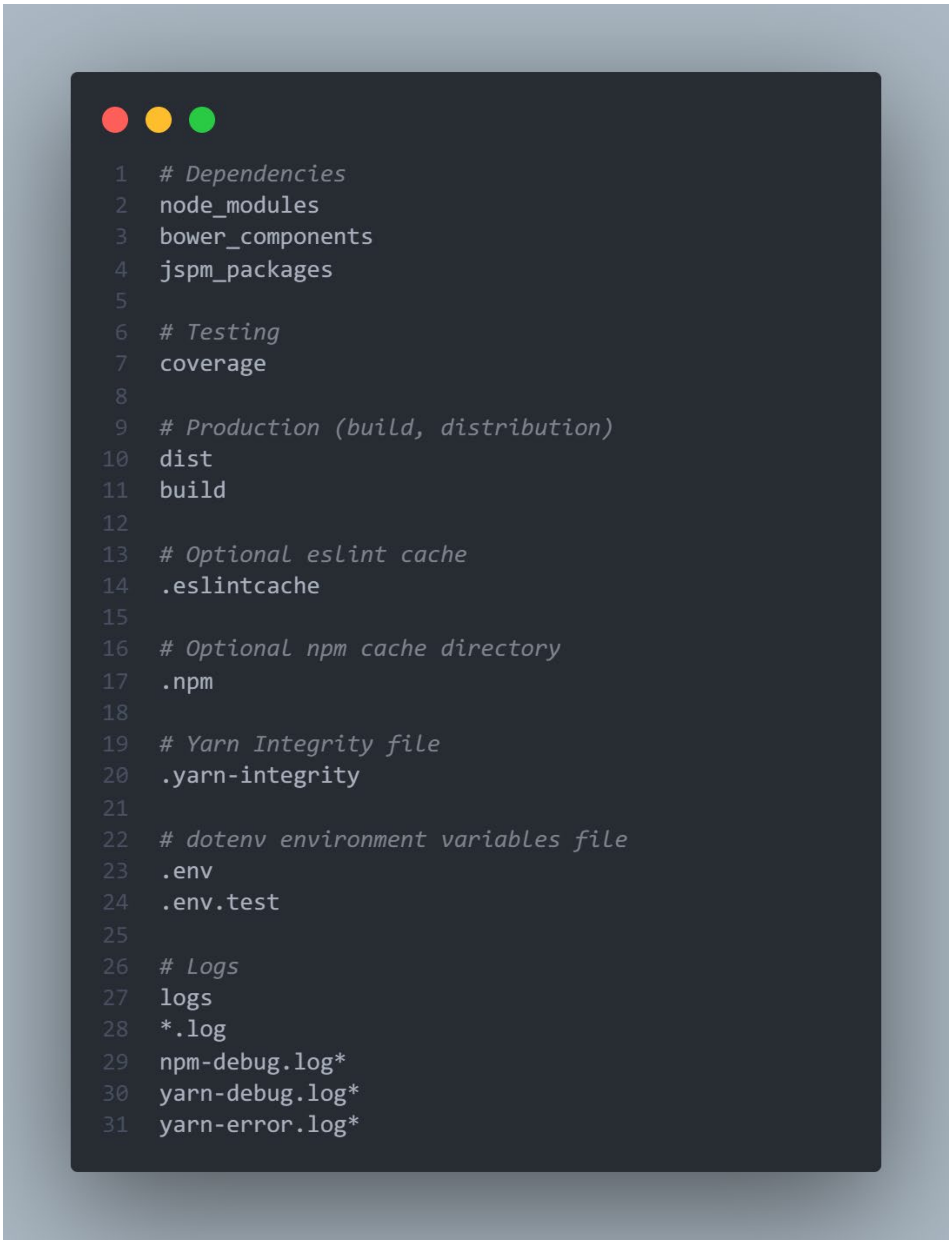
```
{
  "plugins": [
    "stylelint-scss"
  ],
  "extends": "stylelint-config-recommended-scss"
}
```

Figure 23. The configuration file for StyleLint by default is called `.stylelintrc`.

Figure 23 depicts some of the configurations supported by StyleLint. A StyleLint plugin `stylelint-scss` allows StyleLint to work with SCSS, or more commonly known as Sass, a language extension of CSS. Inside `extends`, the customized linting rule is defined to ask StyleLint to check all SCSS codes based on the specific preferences.

Linters and formatters also support ignore rules

All the linting and formatting tools support ignore rules, similar to Git. Inside these `.ignore` files, directories, and files can be added to the ignore rules, so they can be left untouched by the tools.



```
# Dependencies
node_modules
bower_components
jspm_packages

# Testing
coverage

# Production (build, distribution)
dist
build

# Optional eslint cache
.eslintcache

# Optional npm cache directory
.npm

# Yarn Integrity file
.yarn-integrity

# dotenv environment variables file
.env
.env.test

# Logs
logs
*.log
npm-debug.log*
yarn-debug.log*
yarn-error.log*
```

Figure 24. Ignore rules can be defined inside these `.ignore` files.

Figure 24 shows an example `.ignore` files. The most common directories to be ignored are usually the ones that contain dependencies, testing results, configuration files, and build files. It is no exception in this case study, `node_modules` and `bower_components` are the folders that hold dependencies, `coverage` for the testing results from Jest, `dist` and `build` and `dist` contains the build files generated by Webpack that are ready to be served to clients. Standard regular expressions are also supported in these `.ignore` files as can be seen in the last part of the code snippet. In the `#Config` section, `webpack.*.js` allows tools to ignore all files that started with `webpack`, and the same rule also applied to `jest.*.js`. As for `*.config.js`, it means that all files that ended with `config.js` will be ignored.

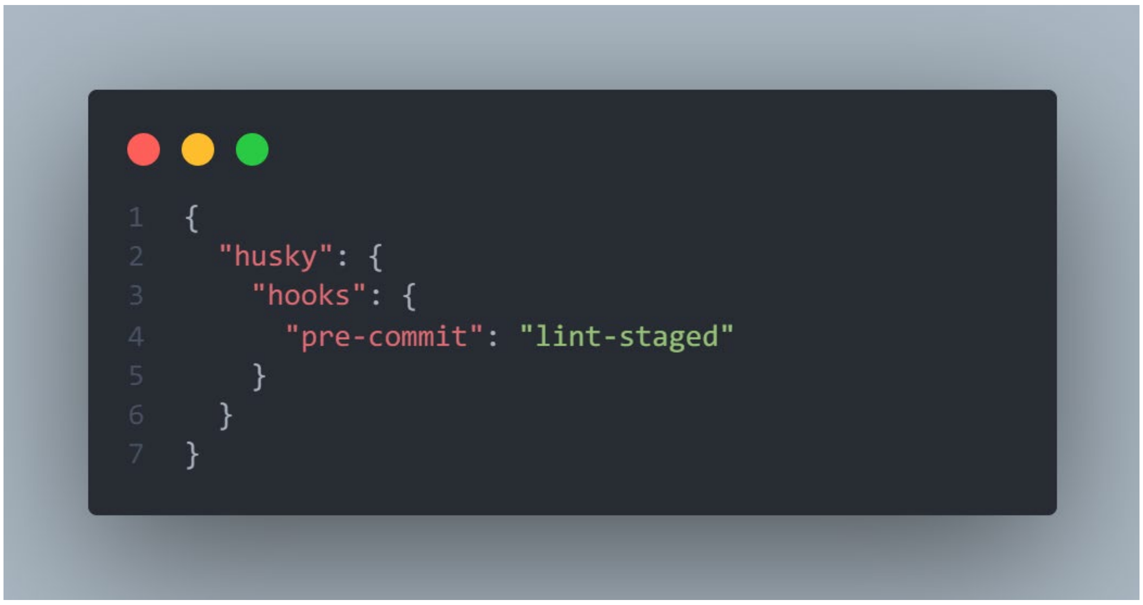
5.3.1 On-demand processing

Problems with running linter and formatter on every file

After all the linting and formatting tools are properly set up, this proof-of-concept project still has one thing missing, to automatically run all the staged files against linters and formatters on every code commit. The reason to check only the staged files instead of running the full linting or formatting command is that as the scale of projects gets larger, the amount of time that is required to check all files also scales with that. However, to achieve this, Git or other version control systems need to be able to communicate with all the linting and formatting tools.

Solutions and case studies: Git Hooks, Husky, and Lint-staged

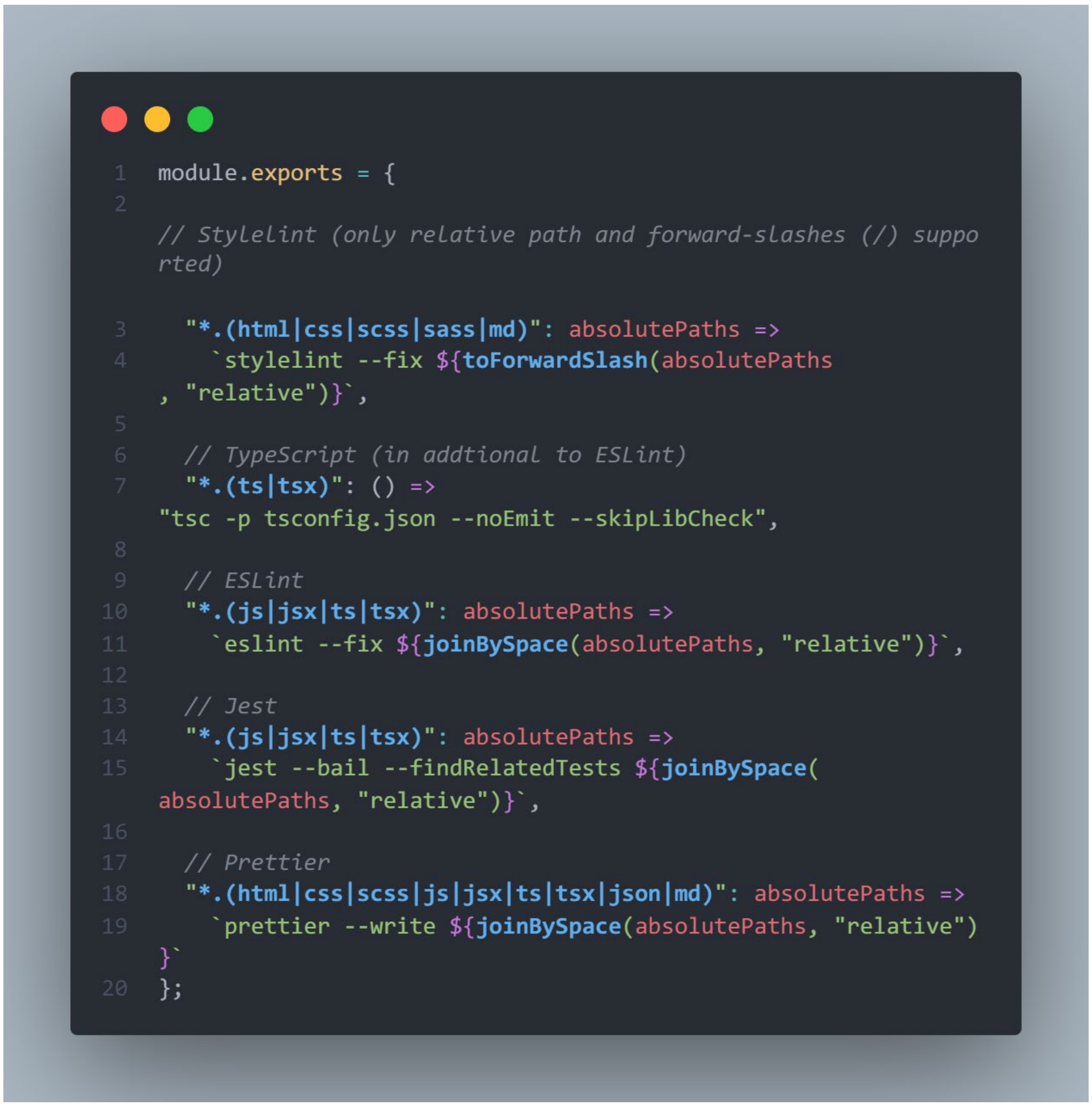
Fortunately, Git Hooks can do exactly that. In this case study, a Git Hooks called “Husky” is installed to enable custom scripts to be run on every Git command, while `lint-staged` is employed to extract the names from the list of staged files controlled by Git and feed the names to all the linting and formatting tools.



```
{
  "husky": {
    "hooks": {
      "pre-commit": "lint-staged"
    }
  }
}
```

Figure 25. \`lint-staged\` will be run on every \`git commit\`.

Figure 25 demonstrates how to allow Git Hooks to run Lint-staged on every Git commit. However, different linting and formatting CLIs often support different kinds of file paths, it is needed to customize the paths \`lint-staged\` extracted from Git to meet the requirements of a specific linting or formatting CLI. \`lint-staged\` has taken that into consideration, and thus supports writing JavaScript functions to customize the paths.



```
module.exports = {

// Stylelint (only relative path and forward-slashes (/) supported)

  "*.(html|css|scss|sass|md)": absolutePaths =>
    `stylelint --fix ${toForwardSlash(absolutePaths, "relative")}`,

  // TypeScript (in addtional to ESLint)
  "*.(ts|tsx)": () => "tsc -p tsconfig.json --noEmit --skipLibCheck",

  // ESLint
  "*.(js|jsx|ts|tsx)": absolutePaths =>
    `eslint --fix ${joinBySpace(absolutePaths, "relative")}`,

  // Jest
  "*.(js|jsx|ts|tsx)": absolutePaths =>
    `jest --bail --findRelatedTests ${joinBySpace(absolutePaths, "relative")}`,

  // Prettier
  "*.(html|css|scss|js|jsx|ts|tsx|json|md)": absolutePaths =>
    `prettier --write ${joinBySpace(absolutePaths, "relative")}`
};
```

Figure 26. The `module.exports` of `lint-staged` configuration file.

Figure 26 describes how Lint-staged can be configured to pass down relative or absolute paths depending on the requirement of a tool.

5.4 Testing and debugging

Problems

When writing in compiled programming languages, an IDE is required to be able to run the codes. Such IDEs are often shipped with built-in testing frameworks and debuggers like Microsoft Visual Studio's built-in NUnit and Xunit for C# testing (Poole & Prouse, 2019) (.NET Foundation, 2020). As discussed at the beginning of the Development Environment section, the lack of proper IDEs for web development results in the appearance of a large number of third-party toolsets, it is no exceptions when it comes to testing and debugging. For front-end web development, traditionally, the testing heavily relied on end-to-end testing which required developers to test all elements and functionalities of all levels of user interface manually by hands. As for debugging, it was often done by using the browsers' built-in inspection tools. With the birth of numerous libraries and frameworks for the front-end web, customized testing frameworks and debuggers are also developed to specifically work with each of those libraries or frameworks (Greif & Benitte, 2019).

Types of testing

Web testing is a software testing method to examine web applications for potential bugs. Any web-based applications should complete this testing before putting them into production. By implementing website testing, organizations can assure that the system is functioning accurately and can be taken by real-time users (Nguyen, et al., 2003).

Unit testing is a type of software testing where singular units or components of the software are tested. This process is usually done throughout the development stage of an application. Unit testing isolates a segment of the codebase and verifies its correctness. Unit Testing is the most fundamental testing and usually done by the developer.

End-to-end testing is a method used to assess the flow of an application from start to end to observe the behavior of a web application. The purpose of this process is to ensure that data integrity and to identify system dependencies are maintained within multiple system components and systems. A whole application is tested for crucial functionalities like communicating with other interfaces, databases, systems, networks, and other applications.

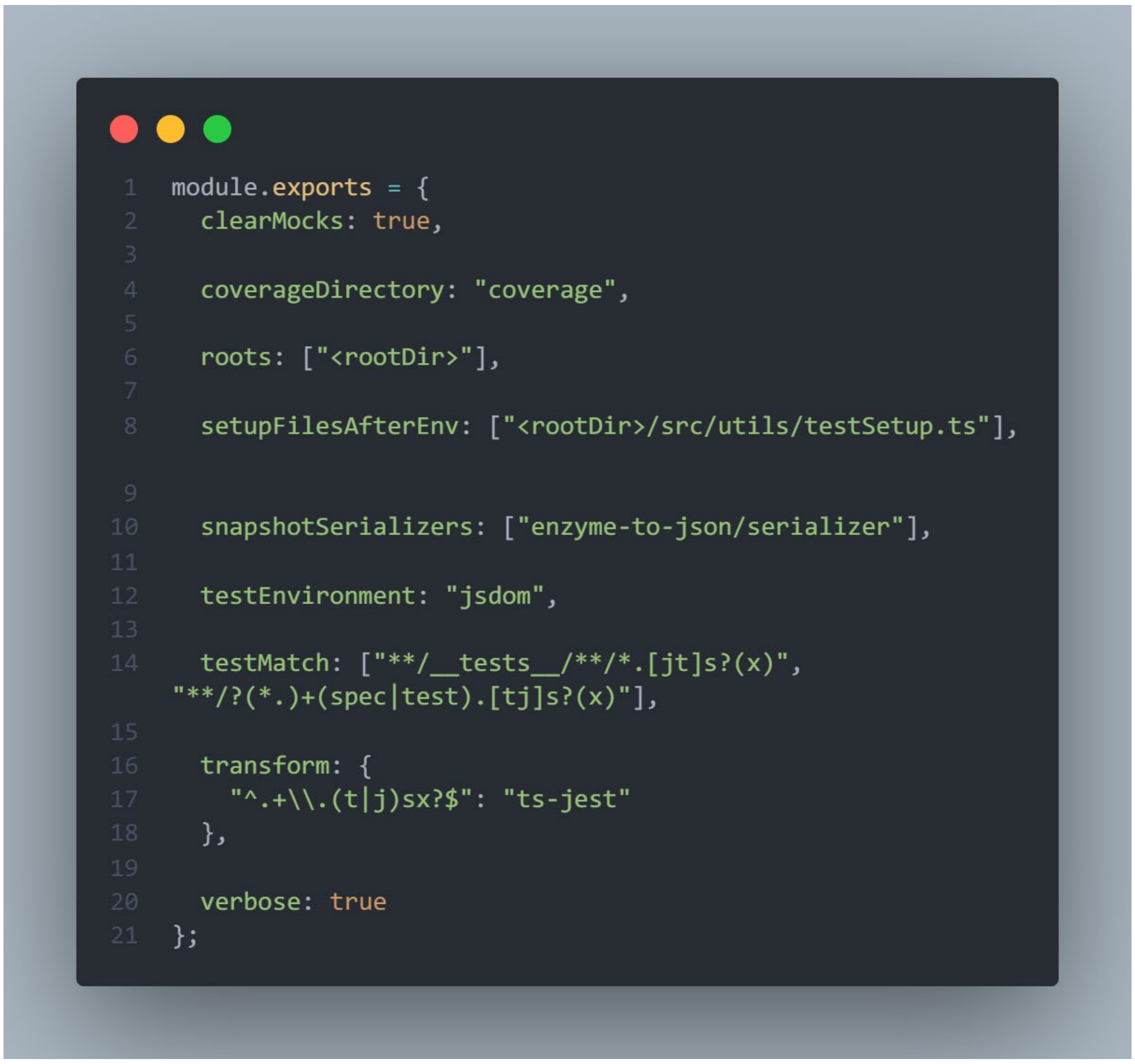
Solutions: Third-party testing frameworks and debuggers

In the following case study, Jest and Enzyme will be employed to write tests for React, Redux, TypeScript, and JavaScript in general. React Developer Tools and Redux DevTools alongside with browsers' built-in inspection tools are used for debugging.

Case study: Jest

Jest is a JavaScript testing framework created by developers at Facebook that provides easy to use APIs and features like snapshot testing (Facebook Inc., 2020). Currently, Jest is the most commonly adopted JavaScript testing framework in the industry (Greif & Benitte, 2019).

Similar to other libraries and frameworks, Jest also supports using configuration files to customize its preferences and features. In this case, a `jest.config.js` can be automatically generated by first installing Jest CLI globally and then running `jest –init` in the terminal.

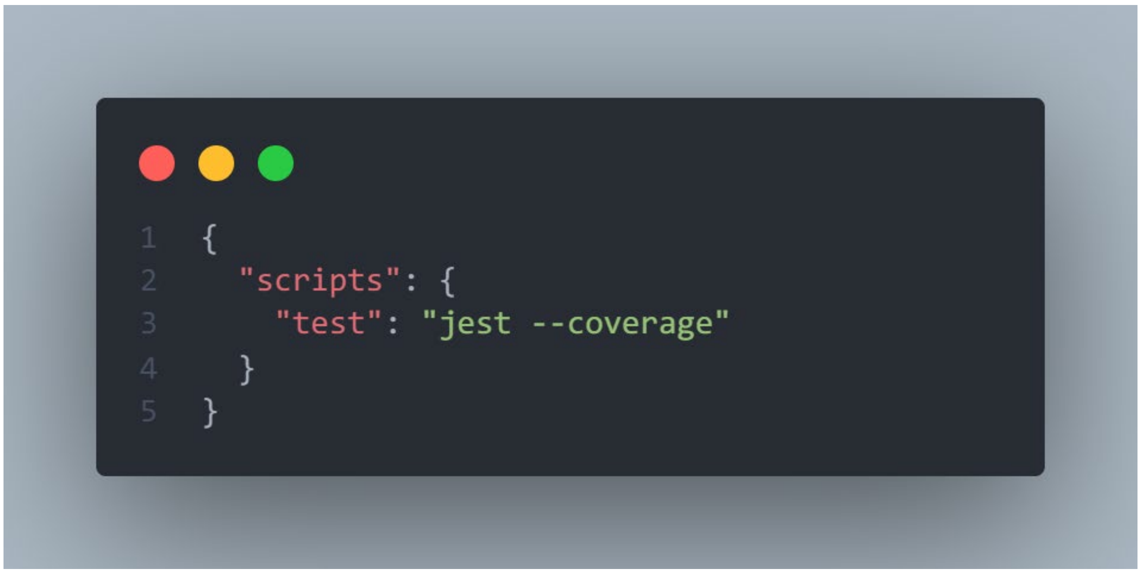


```js
module.exports = {
  clearMocks: true,

  coverageDirectory: "coverage",

  roots: ["<rootDir>"],

  setupFilesAfterEnv: ["<rootDir>/src/utils/testSetup.ts"],


  snapshotSerializers: ["enzyme-to-json/serializer"],

  testEnvironment: "jsdom",

  testMatch: ["**/__tests__/**/*.[jt]s?(x)",
"**/?(*.)+(spec|test).[tj]s?(x)"],

  transform: {
    "^.+\\.(t|j)sx?$": "ts-jest"
  },

  verbose: true
};
```

Figure 27. The configuration file for Jest.

Figure 27 showcases some of configurations supported by Jest. By default, `jest.config.js` will have all the Jest options available inside the file in the form of comments. To change a certain option, just un-comment the line of codes and modify it accordingly. A `jest.config.js` with all kinds of helpful comments can be generated by running `jest --init` in the terminal.

Inside `package.json`, custom scripts can be added. In this case, running `npm test` (or `yarn test` if Yarn is used) in the terminal will execute `jest –coverage`. The flag `--coverage` is one of the Jest CLI options that generate a detailed testing coverage report of the project.



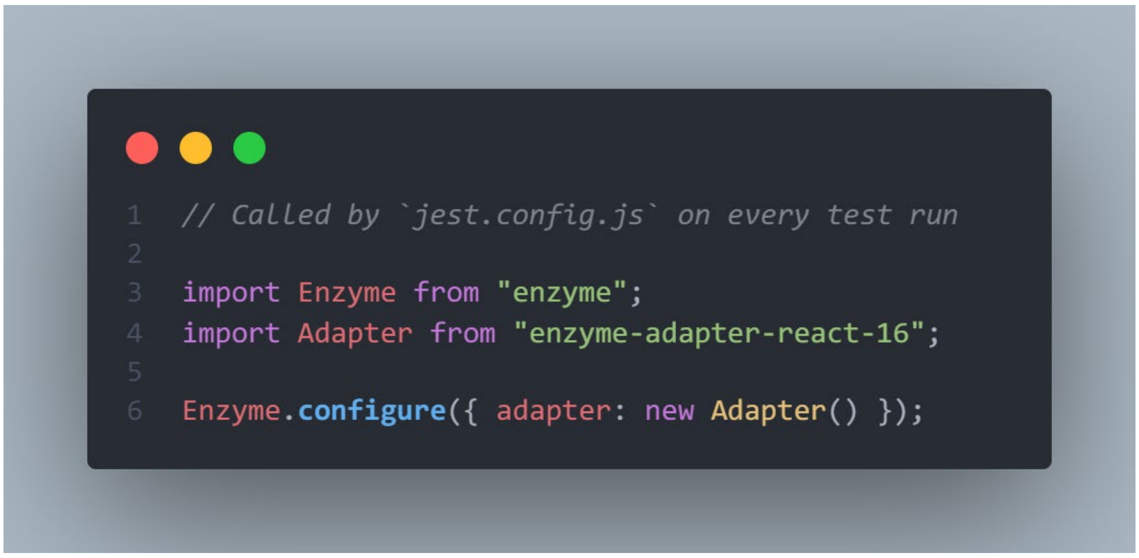
```
{
  "scripts": {
    "test": "jest --coverage"
  }
}
```

Figure 28. Customized scripts can be defined inside the `scripts` object.

As can be seen in Figure 28, running `npm test` will execute `jest --coverage` so that Jest is not required to install globally. `--coverage` will generate a `coverage` folder on the root directory of the project which contains files like HTML that can be opened to inspect how many lines of code have been covered by the tests.

Case study: Enzyme

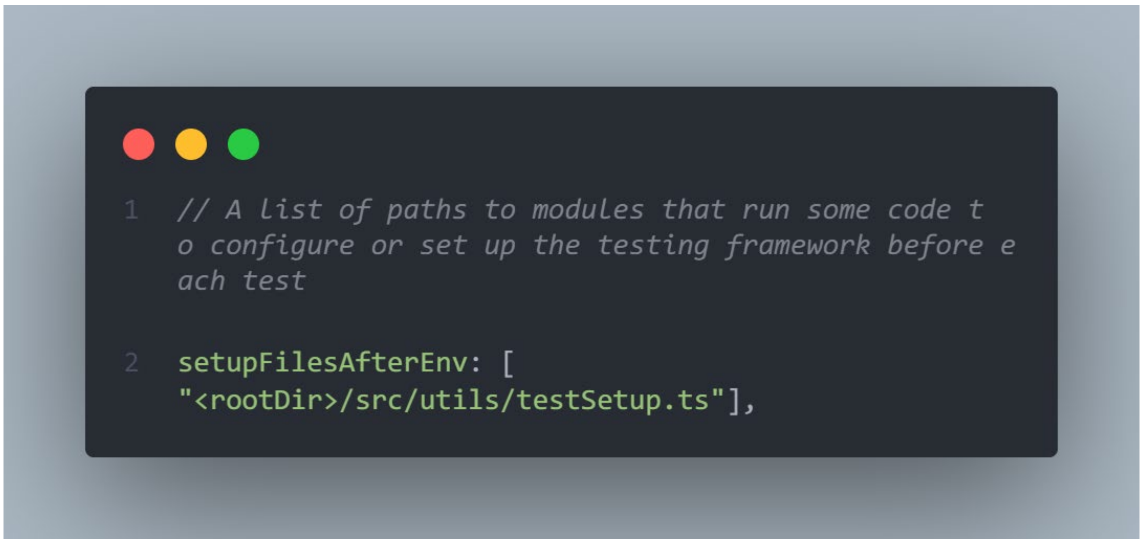
Apart from Jest, Enzyme is a testing utility that is often used alongside with other testing frameworks for testing a React component's output. Enzyme is described as unopinionated when it comes to working with bundlers and assertion libraries, so it can be used together with most of the popular web project dependencies like Webpack Mocha, Browserify and more (Airbnb, Inc., 2020).



```
// Called by `jest.config.js` on every test run

import Enzyme from "enzyme";
import Adapter from "enzyme-adapter-react-16";

Enzyme.configure({ adapter: new Adapter() });
```

Figure 29. Enzyme and Adapter will be imported and configured on every test run.

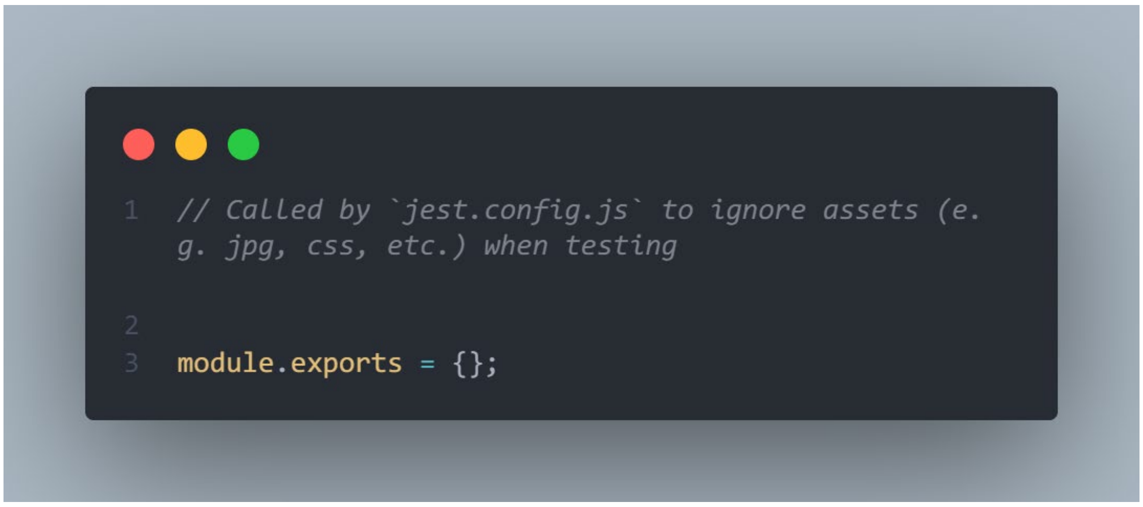
Figure 29 demonstrates how to import Enzyme and its adapter automatically on every test run. By default, it is required to import Enzyme, Adapter, and configure the Adapter for every test written. Fortunately, the `setupFilesAfterEnv` option from `jest.config.js` can solve this problem by simply moving the repetitive codes needed for every test to an individual file and then add the path to this file inside the `setupFilesAfterEnv` option. In this case, a `testSetup.js` is created to handle the Enzyme and its Adapter. Moving the repetitive codes needed for every test to a standalone file so that should these lines of code need to be changed, we do not need to use IDE features like find-and-replace. Instead, we can simply change them once only.



```
// A list of paths to modules that run some code t
o configure or set up the testing framework before e
ach test

setupFilesAfterEnv: [
"<rootDir>/src/utils/testSetup.ts"],
```

Figure 30. Adding the path of the file above to the Jest configuration.

As depicted in Figure 30, adding the file path to the `setupFilesAfterEnv` option will allow Jest to run the codes inside that file before each test.



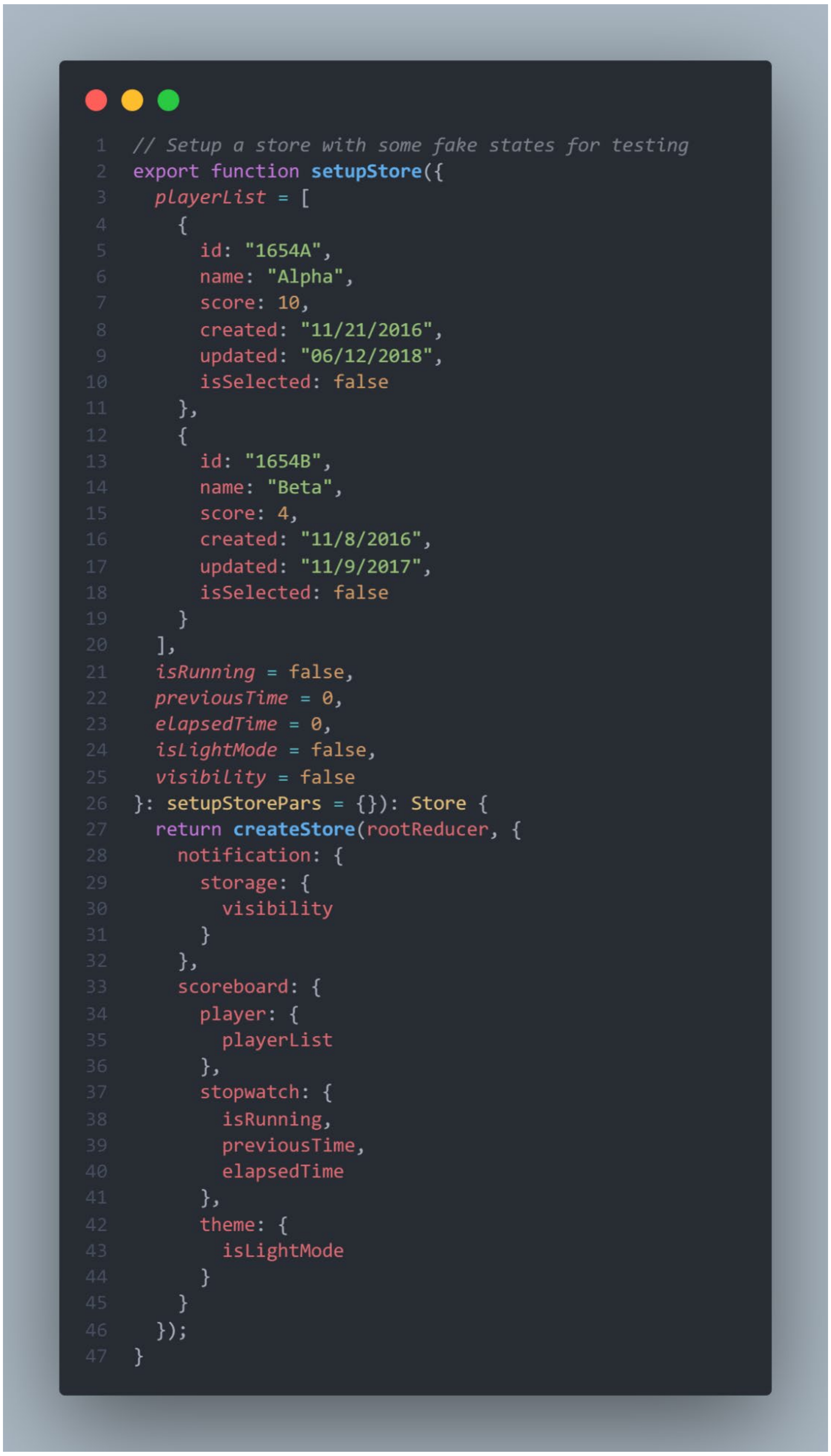
```
// Called by `jest.config.js` to ignore assets (e.g. jpg, css, etc.) when testing

module.exports = {};
```

Figure 31. An empty module is exported to represent the assets.

Figure 31 shows how to configure Jest skip the assets when running tests, because assets like images and other formats like CSS will not be recognized by Jest. Thus, it is required to use a stub to simulate the behavior of these files. In this case, since the files themselves are needed to be tested, and the style of UI elements are not a concern of Jest or Enzyme, an empty module is exported to represent these files.

Enzyme and React-Redux Hooks

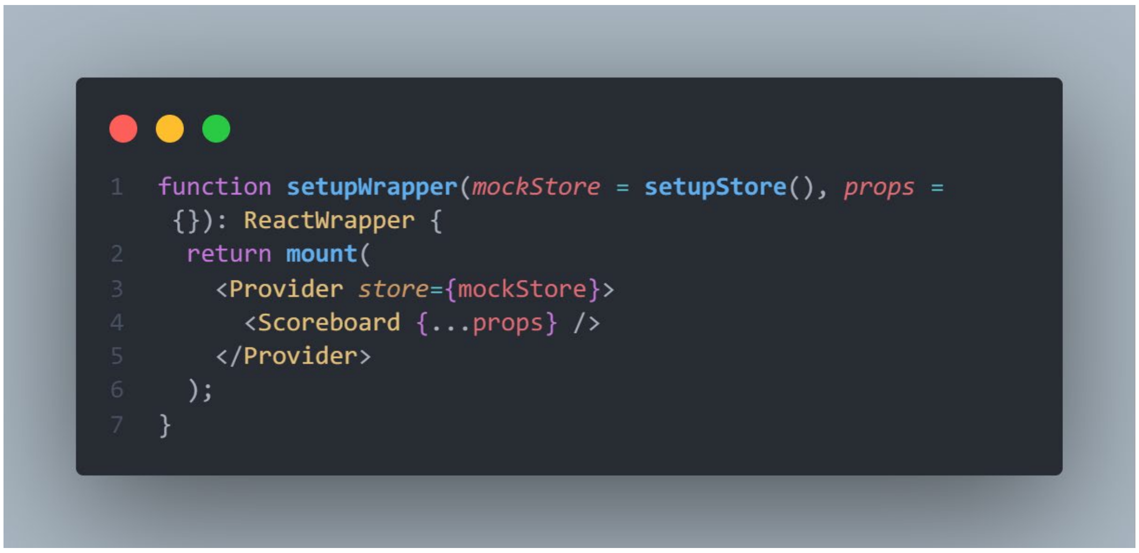
While the way Jest and Enzyme working with React, Redux, JavaScript, or TypeScript is straight forward enough, the new Hooks like React Hooks and React-Redux Hooks has introduced extra complexity to the matter. Without Hooks, Jest and Enzyme can access the states of a component directly to test how it is changed and manipulated, and the use of Redux means that some of the component states are stored inside the Redux store while some are stored within the components. While the use of React-Redux Hooks does not prevent us from accessing the states inside the Redux store, all the internal states of a component remain inaccessible. Thus, instead of testing the components' states themselves, now the tests must be focusing on how the UI elements of a component changed if the states are updated, and to test how states are updated, we need to simulate the interactions like `onClick` event to trigger the updates. In other words, we must shift from the mindset of traditional unit testing a.k.a. testing the individual functionalities to test how users interact with the UI elements and what those interactions mean for the states of a component.



```
// Setup a store with some fake states for testing
export function setupStore({
  playerList = [
    {
      id: "1654A",
      name: "Alpha",
      score: 10,
      created: "11/21/2016",
      updated: "06/12/2018",
      isSelected: false
    },
    {
      id: "1654B",
      name: "Beta",
      score: 4,
      created: "11/8/2016",
      updated: "11/9/2017",
      isSelected: false
    }
  ],
  isRunning = false,
  previousTime = 0,
  elapsedTime = 0,
  isLightMode = false,
  visibility = false
}: setupStorePars = {}): Store {
  return createStore(rootReducer, {
    notification: {
      storage: {
        visibility
      }
    },
    scoreboard: {
      player: {
        playerList
      },
      stopwatch: {
        isRunning,
        previousTime,
        elapsedTime
      },
      theme: {
        isLightMode
      }
    }
  });
}
```

Figure 32. A helper function that set up a mocked Redux store.

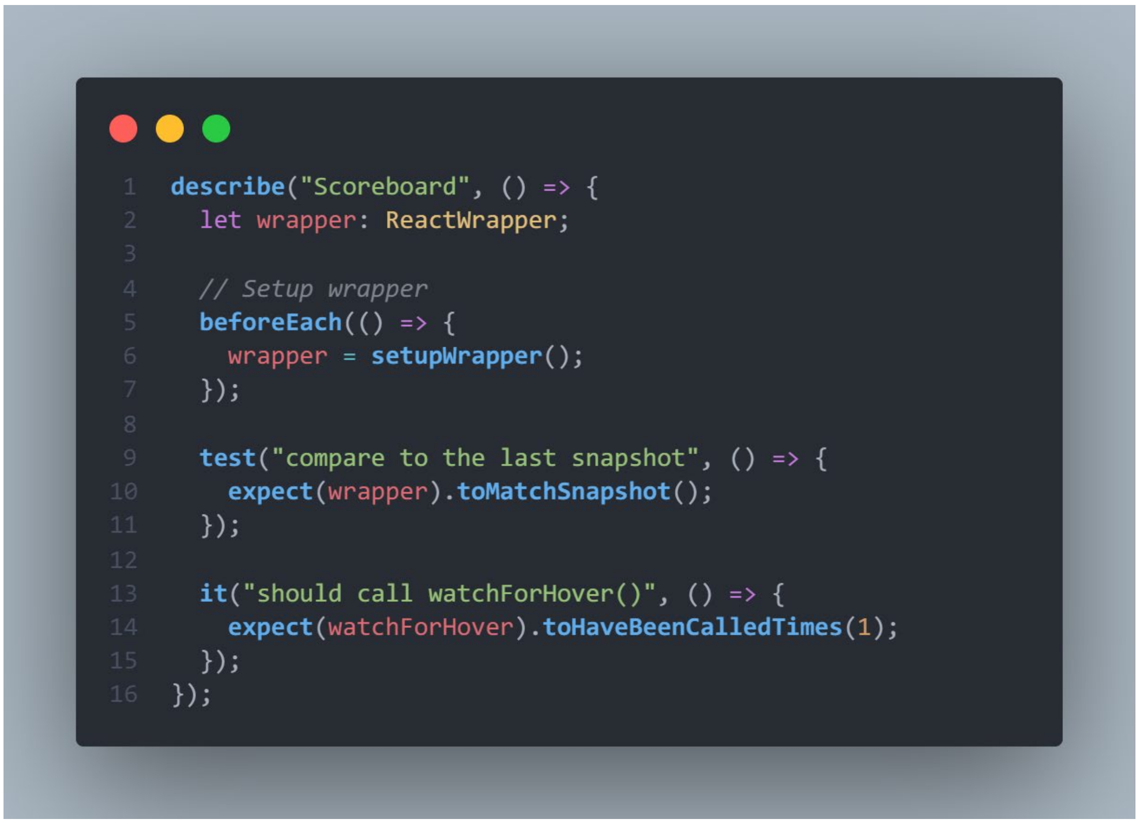
As can be observed in Figure 32, since some of the states are stored inside Redux, hence a utility function to set up a mocked store (a store that simulates the real one) is written to help the process. A function to set up a mocked Redux store to simulate the real one, a user of this function can pass optional arguments to modify the default store settings.



```
function setupWrapper(mockStore = setupStore(), props =
 {}): ReactWrapper {
  return mount(
    <Provider store={mockStore}>
      <Scoreboard {...props} />
    </Provider>
  );
}
```

Figure 33. How the Redux store setup function is used.

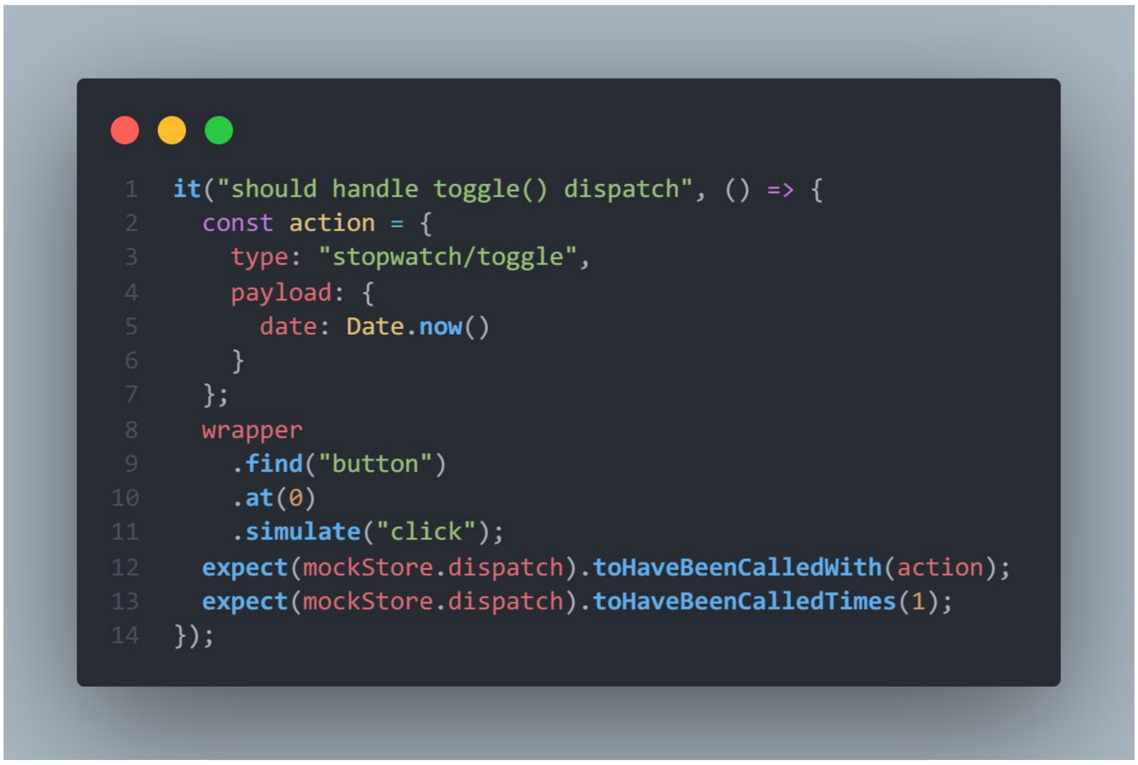
Figure 33 demonstrates how the codes from Figure 32 can be used. The mocked Redux store is first passed to the Provider, and then the Provider feeds the content of this store to the component.



```
describe("Scoreboard", () => {
  let wrapper: ReactWrapper;

  // Setup wrapper
  beforeEach(() => {
    wrapper = setupWrapper();
  });

  test("compare to the last snapshot", () => {
    expect(wrapper).toMatchSnapshot();
  });

  it("should call watchForHover()", () => {
    expect(watchForHover).toHaveBeenCalledTimes(1);
  });
});
```

Figure 34. An example of testing a React component.

Figure 34 demonstrates how a React component is tested in Jest and Enzyme in which snapshot testing is used to ensure the UI does not change unexpectedly without the need to write test for each UI element.



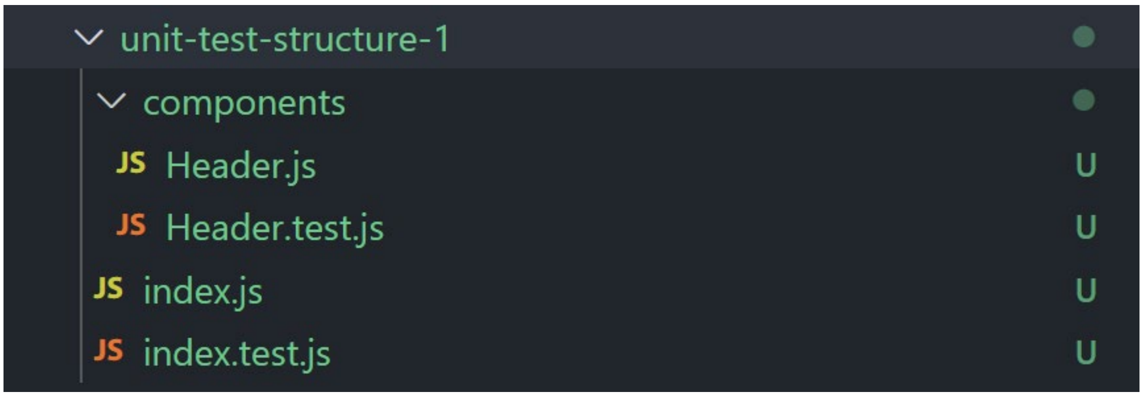
```javascript
it("should handle toggle() dispatch", () => {
  const action = {
    type: "stopwatch/toggle",
    payload: {
      date: Date.now()
    }
  };
  wrapper
    .find("button")
    .at(0)
    .simulate("click");
  expect(mockStore.dispatch).toHaveBeenCalledWith(action);
  expect(mockStore.dispatch).toHaveBeenCalledTimes(1);
});
```

Figure 35. An example of how functions related to Redux are tested.

In Figure 35, when testing Redux related functions of a component, how these functions impact the actual UI elements is tested to better reflect how users will actually interact with the application.

Two ways to organize the test files

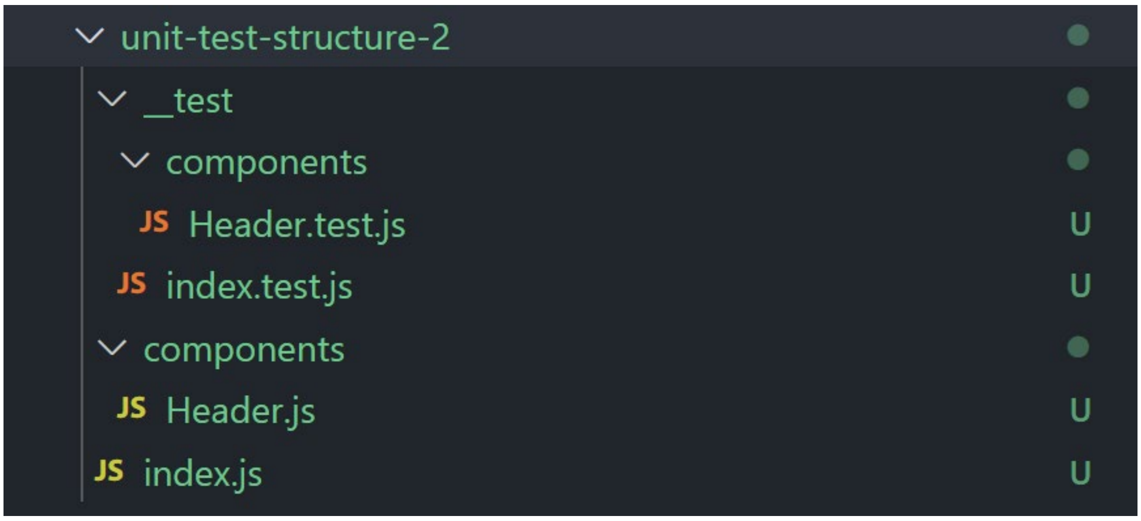
In general, there are two ways to organize the test files. One method is to place the test file right next to file it is supposed to test, this approach emphasizes that the tests should be an extension to the codes they are testing. Another common method is to place all the tests to a `__test` folder and organize the tests to mirror the actual project structure.



```
∨ unit-test-structure-1          ●
  ∨ components                   ●
    JS Header.js                 U
    JS Header.test.js            U
  JS index.js                    U
  JS index.test.js               U
```

Figure 36. The test files are placed next to the file they are testing.

As shown in Figure 36, one method to organize the test files is to place the test file right next to file it is supposed to test, this approach implies that the testing codes are the extension of the codes they are testing.



```
∨ unit-test-structure-2          ●
  ∨ __test                       ●
    ∨ components                 ●
      JS Header.test.js          U
    JS index.test.js             U
  ∨ components                   ●
    JS Header.js                 U
  JS index.js                    U
```

Figure 37. The test files are all placed inside a `__test` folder.

As illustrated in Figure 37, another method is to place all the tests to a `__test` folder and organize the tests to mirror the actual project structure, this approach emphasizes on making each project's module less overwhelming, and easier to navigate through all the directories. By separating the tests from the actual codes, the folder that holds a module will contain only the files that concern the functionality of the module.

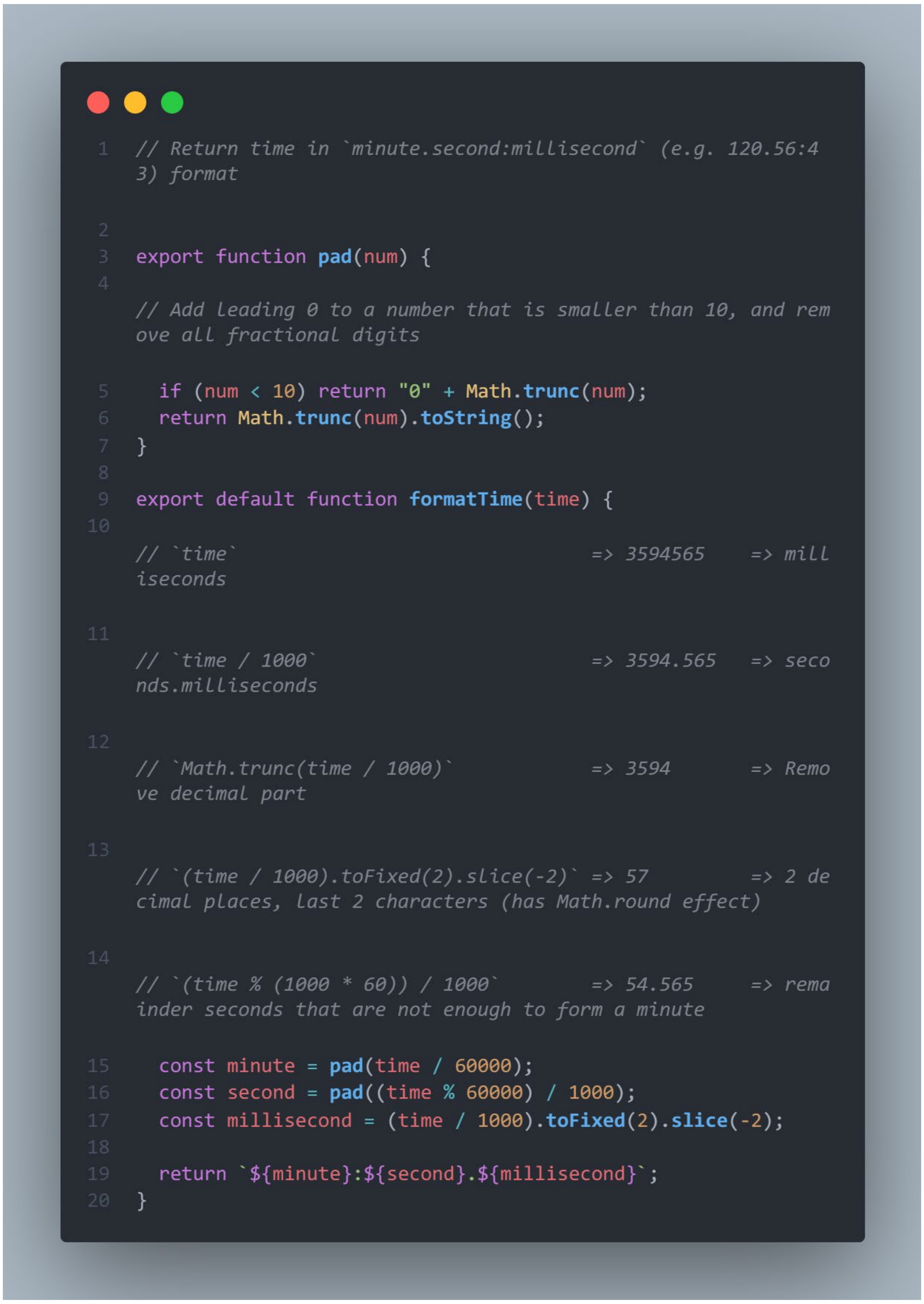
5.5 Optimization for deployment

Problems

Before a front-end web project is ready to be deployed, there are several optimizations should be taken into consideration otherwise the performance will suffer, and ultimately, the user experience will be worse. Since again, there is no authority figure to provide official development tools, various third-party tools are needed to solve each piece of the puzzle.

Solution and case study: Minification

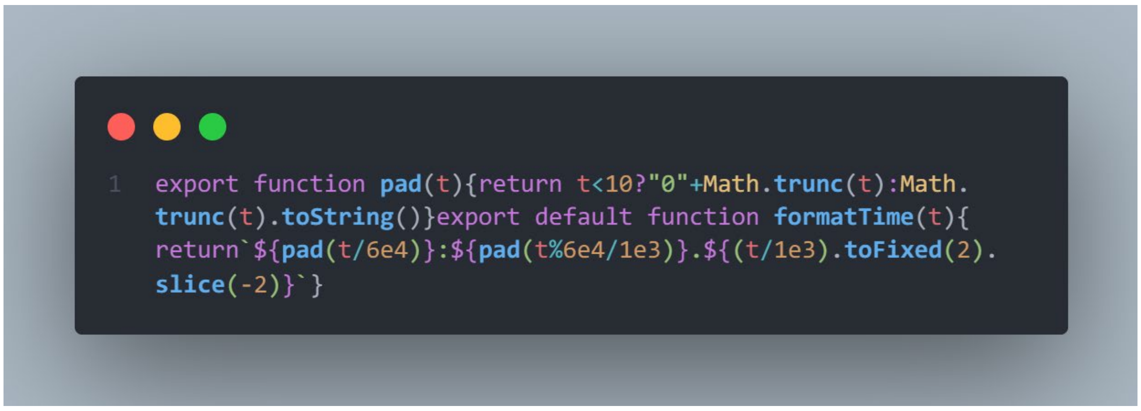
To begin with, minification is a process of eliminating comments, unnecessary white spaces and similar redundancies in source codes to reduce file size and improve loading speed (Google LLC, 2018).



```javascript
// Return time in `minute.second:millisecond` (e.g. 120.56:43) format

export function pad(num) {
// Add leading 0 to a number that is smaller than 10, and remove all fractional digits
  if (num < 10) return "0" + Math.trunc(num);
  return Math.trunc(num).toString();
}

export default function formatTime(time) {
// `time`                               => 3594565     => milliseconds
// `time / 1000`                        => 3594.565    => seconds.milliseconds
// `Math.trunc(time / 1000)`            => 3594        => Remove decimal part
// `(time / 1000).toFixed(2).slice(-2)` => 57          => 2 decimal places, last 2 characters (has Math.round effect)
// `(time % (1000 * 60)) / 1000`        => 54.565      => remainder seconds that are not enough to form a minute
  const minute = pad(time / 60000);
  const second = pad((time % 60000) / 1000);
  const millisecond = (time / 1000).toFixed(2).slice(-2);

  return `${minute}:${second}.${millisecond}`;
}
```

Figure 38. JavaScript source codes before minification.

As can be depicted in Figure 38, the JavaScript source codes before minification are usually consist of many empty spaces and comments to improve readability.

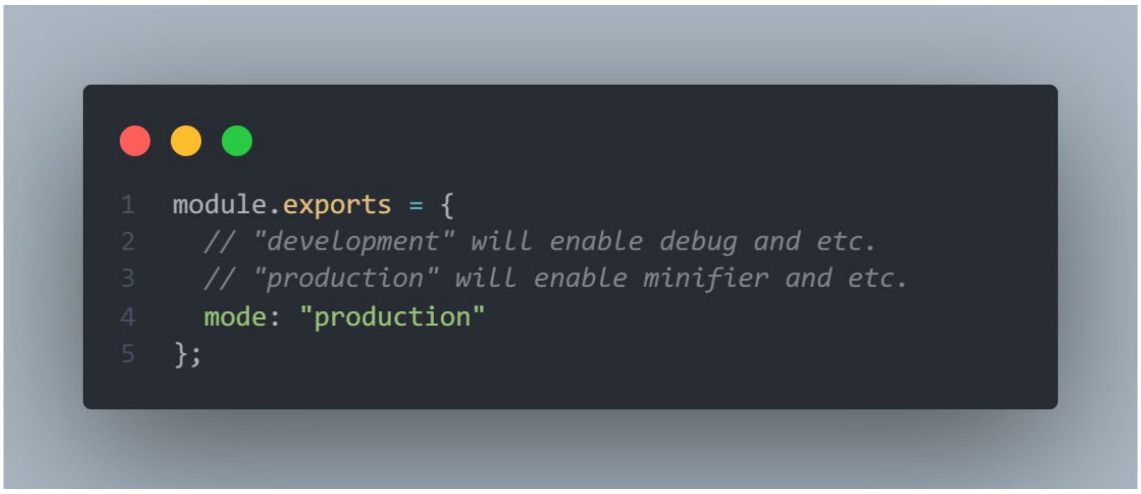


```
export function pad(t){return t<10?"0"+Math.trunc(t):Math.
trunc(t).toString()}export default function formatTime(t){
return`${pad(t/6e4)}:${pad(t%6e4/1e3)}.${(t/1e3).toFixed(2).
slice(-2)}`}
```

Figure 39. JavaScript source codes after minification.

Figure 39 illustrates how codes from Figure 38 look like after minification. As can be observed by comparing Figure 38 and Figure 39, after the source codes are minified, they are no longer human-readable, but the difference in file size is significant. The file size of the source codes in Figure 36 is 1.28 KB (1,318 bytes) while the size of the codes in Figure 37 is mere 188 bytes (188 bytes), the difference of the two is 1130 bytes or 86%. The larger the project, the greater the gain from minification.

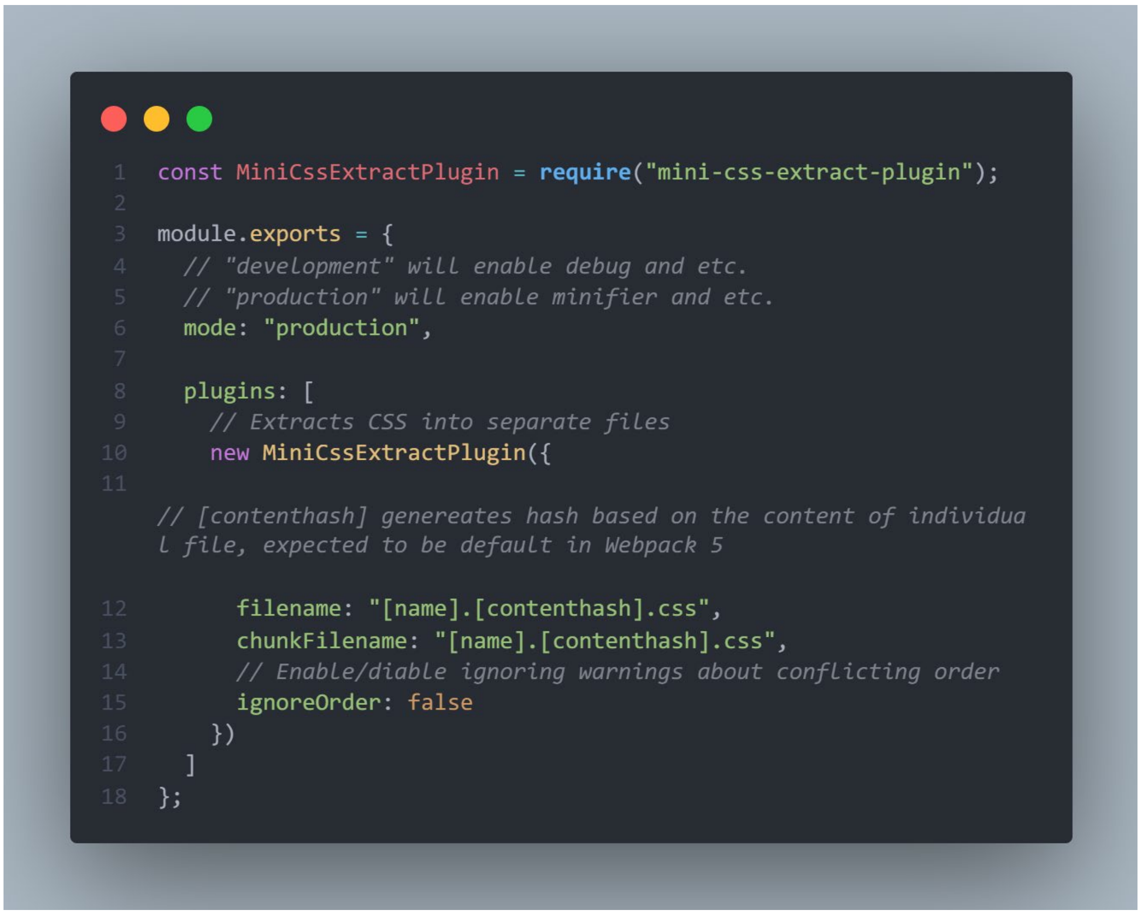
The bundler used in the case study is Webpack and it will use `TerserWebpackPlugin` to minify JavaScript source codes in production mode by default (Webpack, 2020).



```
module.exports = {
  // "development" will enable debug and etc.
  // "production" will enable minifier and etc.
  mode: "production"
};
```

Figure 40. Setting mode to production in `webpack.config.js` to enable minification.

As demonstrated in Figure 40, by setting `mode` to `production`, Webpack will enable built-in minification. However, Webpack does not support minification for CSS by default, so plugins are needed to support that.



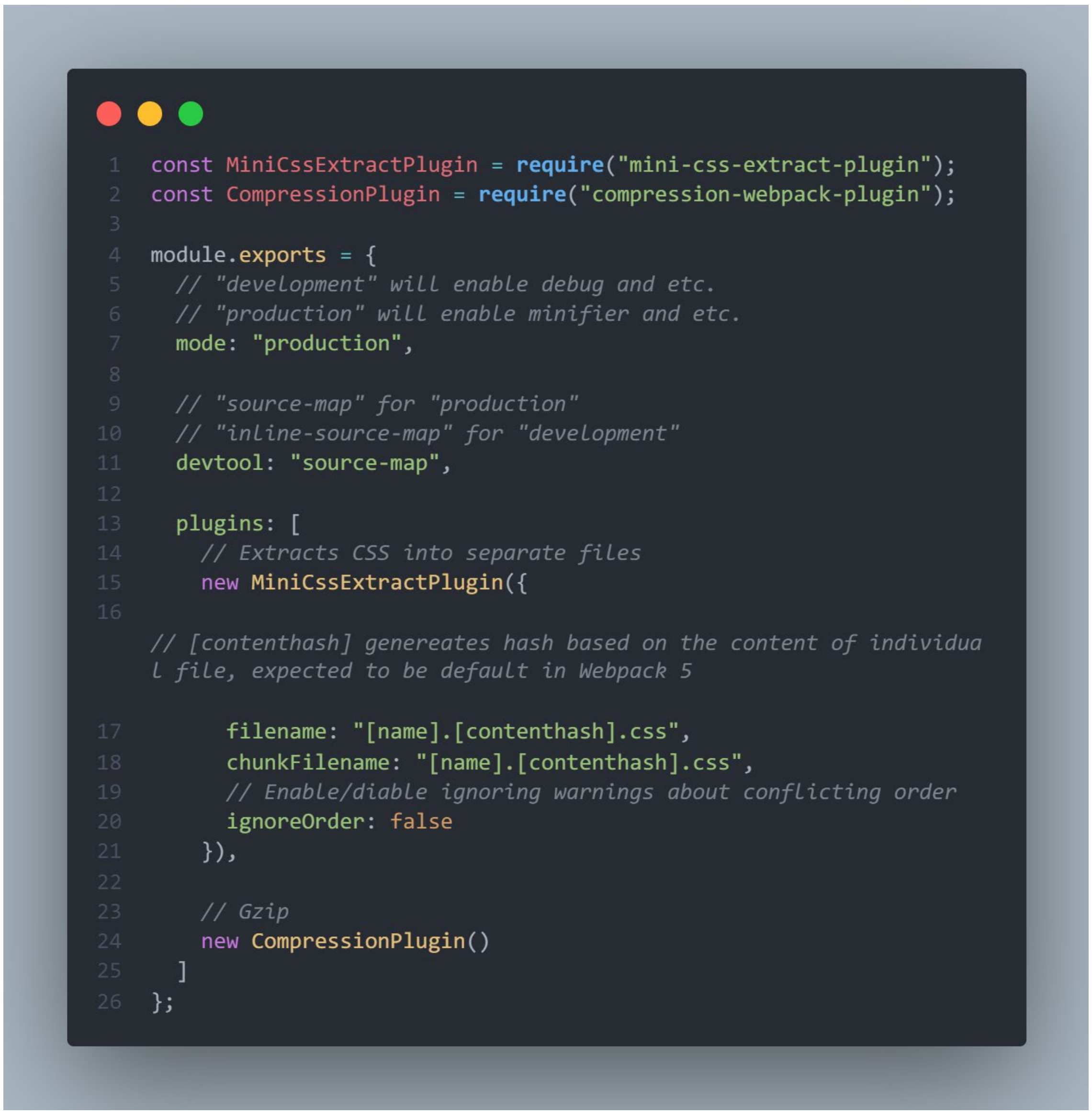
```
const MiniCssExtractPlugin = require("mini-css-extract-plugin");

module.exports = {
  // "development" will enable debug and etc.
  // "production" will enable minifier and etc.
  mode: "production",

  plugins: [
    // Extracts CSS into separate files
    new MiniCssExtractPlugin({

// [contenthash] genereates hash based on the content of individua
l file, expected to be default in Webpack 5

      filename: "[name].[contenthash].css",
      chunkFilename: "[name].[contenthash].css",
      // Enable/diable ignoring warnings about conflicting order
      ignoreOrder: false
    })
  ]
};
```

Figure 41. How to add CSS minification for Webpack.

In Figure 41, to allow Webpack to minify CSS codes, plugins like `mini-css-extract-plugin` can be added to the Webpack building process.

Solution and case study: Compression

To further reduce file size, compression algorithms like Gzip can be integrated into a project to generate a compress version of the same file while keeping the original file so that if a web server does not support compression, it can still serve the original files. Gzip is a software application written by Jean-loup Gailly and Mark Adler for data compression and depression (Free Software Foundation, Inc., 2018).



```
const MiniCssExtractPlugin = require("mini-css-extract-plugin");
const CompressionPlugin = require("compression-webpack-plugin");

module.exports = {
  // "development" will enable debug and etc.
  // "production" will enable minifier and etc.
  mode: "production",

  // "source-map" for "production"
  // "inline-source-map" for "development"
  devtool: "source-map",

  plugins: [
    // Extracts CSS into separate files
    new MiniCssExtractPlugin({

// [contenthash] genereates hash based on the content of individua
l file, expected to be default in Webpack 5

      filename: "[name].[contenthash].css",
      chunkFilename: "[name].[contenthash].css",
      // Enable/diable ignoring warnings about conflicting order
      ignoreOrder: false
    }),

    // Gzip
    new CompressionPlugin()
  ]
};
```

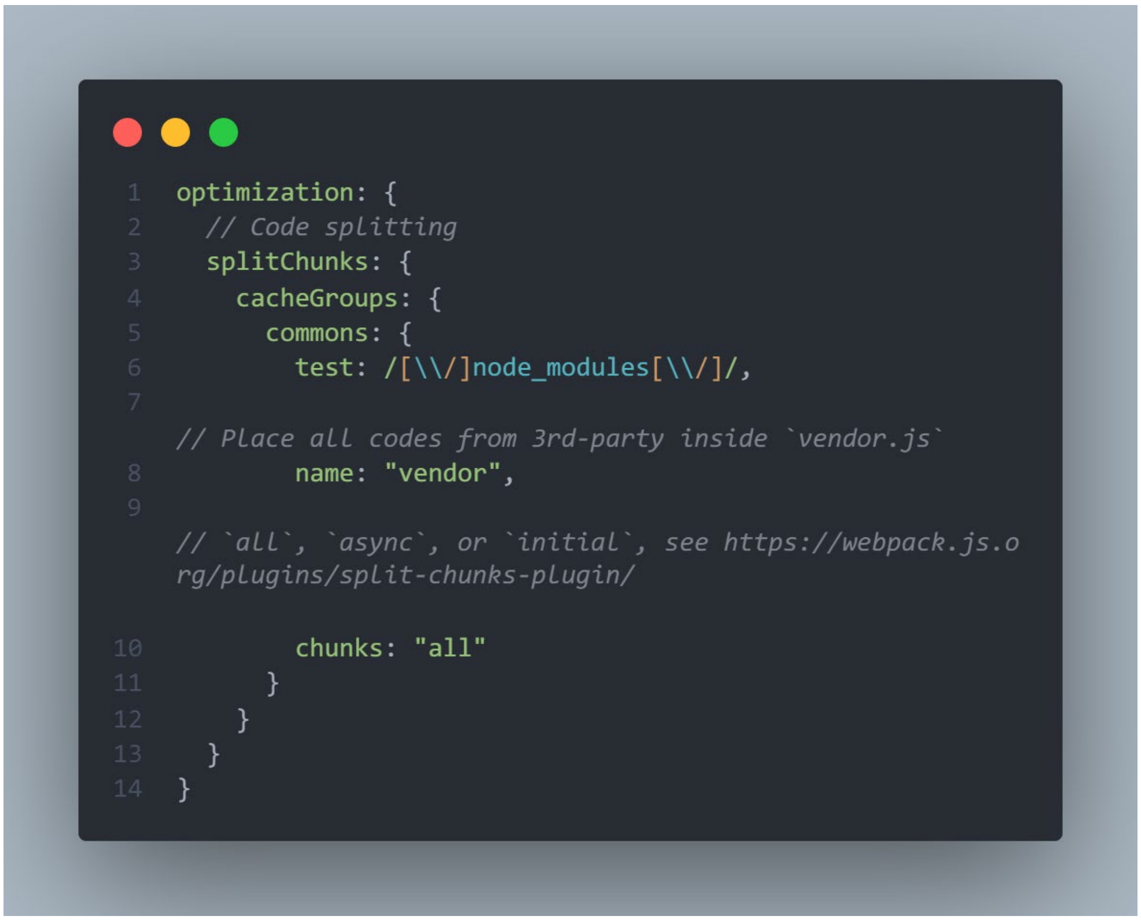
Figure 42. How to add file compression for Webpack.

As can be seen in Figure 42, to allow Webpack to enable compression for all non-asset files, `compression-webpack-plugin` needs to be installed.

Solution and case study: Code splitting

JavaScript is often considered as an interpreted language and its codes can be run without compilation. Traditionally, when a browser receives JavaScript source codes, they are usually passed to the browser's interpreter where each line of code will be interpreted. Nowadays, more modern browsers such as Google Chrome has implemented JIT (Just-in-time) compiler in its engine to increase performance when executing JavaScript codes (Hinkelmann, 2017). However, no matter if a browser interprets or compiles JavaScript source codes, it still needs to go through all the codes first before any meaningful content is rendered and this is especially true for the client-side rendered SPAs. Thus, if a website has all of its JavaScript source codes in one file and a user is visiting the site for the first time, it will take a long time before users can see anything meaningful or interact with any elements on the page. This is why code splitting has now become a major part of front-end web development.

Code splitting as its name suggests is a process of splitting JavaScript source codes into smaller chunks based on some sort of principles. One of the common ways to handle code splitting is to place all the codes from third-part libraries and frameworks in one file while keeping the codes in another (Wagner & Osmani, 2018).



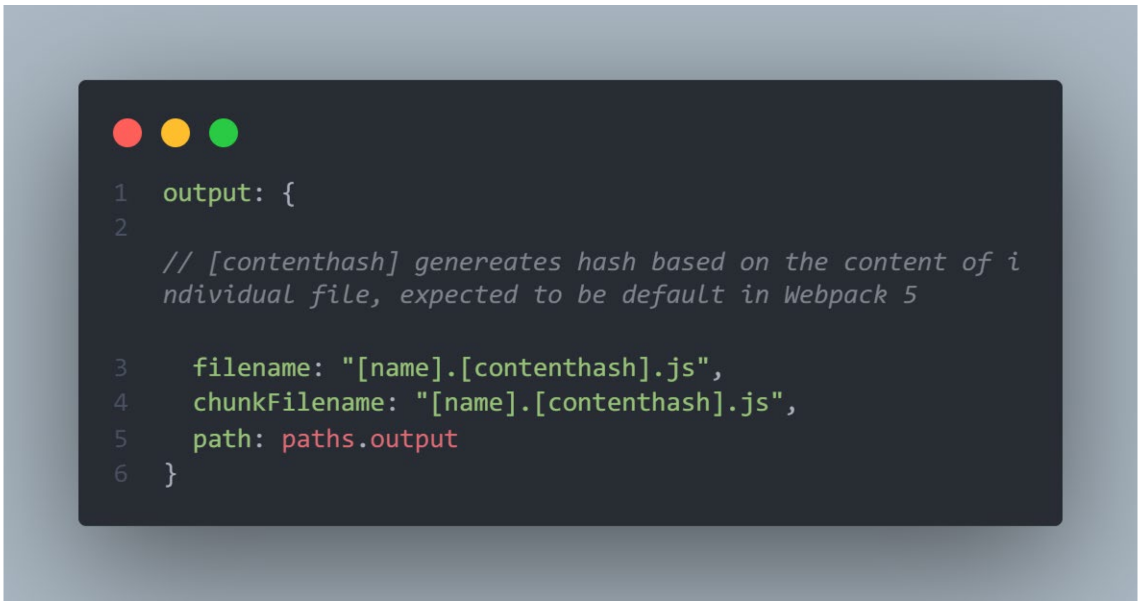
```
optimization: {
  // Code splitting
  splitChunks: {
    cacheGroups: {
      commons: {
        test: /[\\/]node_modules[\\/]/,

// Place all codes from 3rd-party inside `vendor.js`
        name: "vendor",

// `all`, `async`, or `initial`, see https://webpack.js.o
rg/plugins/split-chunks-plugin/

        chunks: "all"
      }
    }
  }
}
```

Figure 43. Webpack supports code splitting by default.

As shown in Figure 43, Webpack supports code splitting out-of-the-box, to enable it, simply add a `splitChunks` object inside the `optimization` object.

Solution and case study: Caching

When using a browser such as Google Chrome, it will store some of the data loaded from a website in its local cache (Google LLC, 2020). This causes a problem sometimes when a website has updated its files, but because the names of those files remained the same, browsers will just load the old files from its local cache. To toggle this issue, some kinds of automatic naming mechanism to name files based on their content will be the solution, and Webpack supports exactly that.



```
output: {
// [contenthash] genereates hash based on the content of individual file, expected to be default in Webpack 5

  filename: "[name].[contenthash].js",
  chunkFilename: "[name].[contenthash].js",
  path: paths.output
}
```

Figure 44. Webpack supports generating hash based on the content of a file.

As demonstrated in Figure 44, by adding `[contenthash]` to the `filename` and `chunkFielname` inside the `output` object, a hash code will be generated automatically by Webpack based on the content of each individual file.

6 RESULTS

The following chapter describes what are the results of the analyses conducted in the chapter 4. In general, the analyses showed that every single library, framework, and utility created for the front-end web development is to address one or several issues caused by the design of HTML, CSS, and JavaScript. Although creating a whole library sorely to address a specific issue seems like overkill at times, it is, however, a result of missing authority figure to provide a carefully designed IDE for the front-end web development.

Starting at addressing the repetition issue inherited from the design of HTML and CSS, then expanding to improving the way JavaScript interacts with UI elements, tools like React, Sass and more are created. As the front-end part of the web development becomes increasingly complex, more and more developers from traditional software engineering have joined in and brought many tested concepts with them to the front-end web development. Since the design of HTML, CSS, and JavaScript is often at odds with these design patterns from the more mature programming languages, the number of front-end web tools created to address all these increases even more.

With the ever more demanding requirements for the front-end web development, the three standards HTML, CSS, and JavaScript are also evolving constantly in hope to toggle all the issues without the need for any third-party web libraries, frameworks, and utilities. However, the speed at which the standards are evolving is not fast enough to catch up with the growth of the web, and the reason for that is all the new specifications though have improved the front-end web in general, they have not solved the most fundamental problem with the front-end web development that is the lack of a proper IDE. What those new features from the later iterations of the web standards have provided is actually to allow more powerful libraries, frameworks, and utilities to be created.

To summarize, this thesis found that the fundamental problems with the way HTML, CSS, and JavaScript are designed, combined with the lack of an authority figure in the front-end web development like that of mobile or desktop platforms is what ultimately results in the creation of a large number of third-party front-end web libraries, frameworks, language extensions, etc.

7 CONCLUSION

The thesis first addresses the problems that are currently present in the web development field, from the fundamentals to the advances. These problems have stayed within the web industry for a very long time.

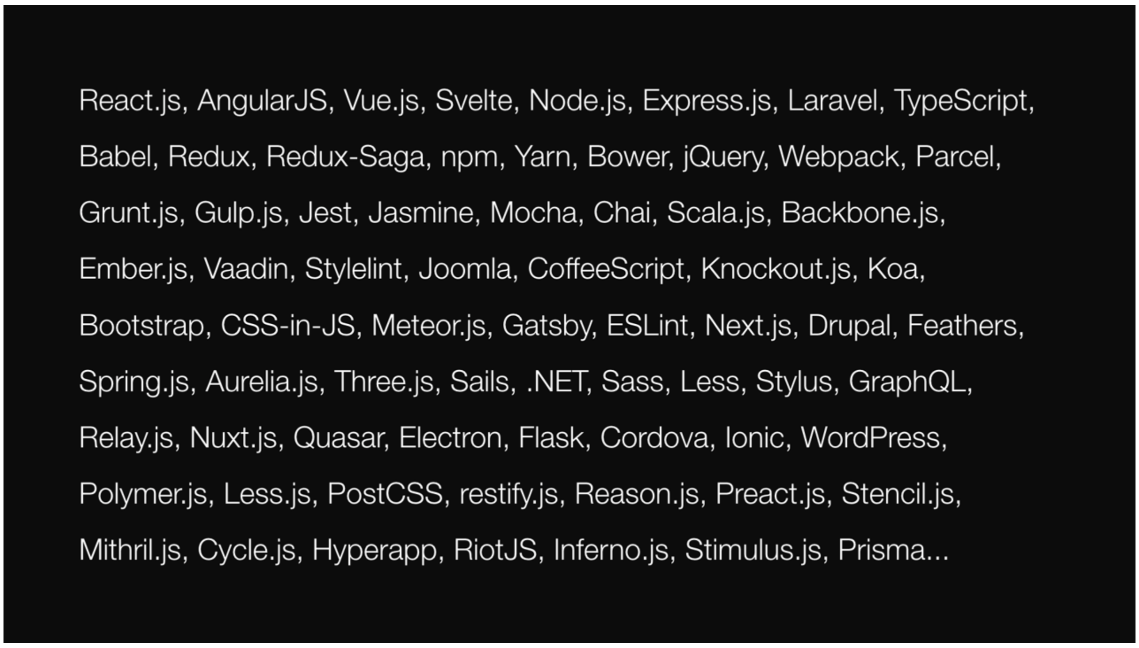

React.js, AngularJS, Vue.js, Svelte, Node.js, Express.js, Laravel, TypeScript, Babel, Redux, Redux-Saga, npm, Yarn, Bower, jQuery, Webpack, Parcel, Grunt.js, Gulp.js, Jest, Jasmine, Mocha, Chai, Scala.js, Backbone.js, Ember.js, Vaadin, Stylelint, Joomla, CoffeeScript, Knockout.js, Koa, Bootstrap, CSS-in-JS, Meteor.js, Gatsby, ESLint, Next.js, Drupal, Feathers, Spring.js, Aurelia.js, Three.js, Sails, .NET, Sass, Less, Stylus, GraphQL, Relay.js, Nuxt.js, Quasar, Electron, Flask, Cordova, Ionic, WordPress, Polymer.js, Less.js, PostCSS, restify.js, Reason.js, Preact.js, Stencil.js, Mithril.js, Cycle.js, Hyperapp, RiotJS, Inferno.js, Stimulus.js, Prisma...

Figure 45. Some of the web libraries, frameworks.

To solve all these problems, a large number of web libraries, frameworks, etc. has been created by developer communities as depicted in Figure 45.

Besides, the thesis also answers the questions of why there are so many web frameworks existing in the web field. It can be concluded that although the number of front-end web libraries and frameworks is overwhelmingly large, it should not be an obstacle to developers. If the organization responsible for giving out standards was not able to follow up with the prosperity of this fast-changing field, web development would not evolve significantly if there is not any creation of these frameworks. The authors believe that it is beneficial because it provides more alternative solutions for developers and helps creating a more open and decentralized web.

The research in this thesis revealed that the number of frameworks, libraries and their alternatives is immense, but it is a necessity for the growth of this field. It is a limitation, but it is undeniably an advantage. Many of the frameworks that exist in the market share the same philosophy and aim for the same outcome. Although the paths each framework chooses are different, they always reach the same direction and goal: to create feature-rich and powerful web applications. To choose which tools to apply is a challenging decision for junior as well as experienced developers. Accordingly, web developers nowadays are required to deeply comprehend or at least, have experiences in more than one or two frameworks and libraries rather than just studying in a specific one. Although this statement is rather not fully accurate, the diversity in knowledge is always important.

The market has responded with a few limited actions so far. There is a rise in web builders and CMS toward low-skilled programmers or implementers. Wix, WordPress, Microsoft Power Apps, Google App Maker are a few examples. These kinds of solutions are far from being a completed solution to what lies behind the scene. They are fast solutions for small start-ups or even personal websites, but not for enterprises. Therefore, the need for experienced web developers is still required.

For a broader view, in the last few years, an outburst in the diversity of consumer devices has been witnessed. This means a lot to the web development field. A larger diversity of devices comes with a larger diversity of platforms and types and shapes of inputs and outputs, for the technological field in general and web development field in specific. It is challenging to obtain the maximum usage for web apps, developers nowadays must plan and make them available to a broad and unpredictable range of devices, that is, building them in such a way that would enable web developers to customize them efficiently to any current devices but also, to be prepared for the future upcoming.

REFERENCES

.NET Foundation, 2020. *xUnit.net.* [Online]
Available at: https://xunit.net/
[Accessed 22 February 2020].

Aho, A. V., Lam, M. S., Sethi, R. & Ullman, J., 2007. *Compilers: Principles, Techniques, and Tools.* 2nd ed. s.l.:Pearson Education, Inc.

Airbnb, Inc., 2020. *Aribnb/Enzyme: JavaScript Testing utilities for React.* [Online]
Available at: https://github.com/airbnb/enzyme
[Accessed 24 February 2020].

Bos, B., Çelik, T., Hickson, I. & Lie, H. W., 2016. *Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification.* [Online]
Available at: https://www.w3.org/TR/2011/REC-CSS2-20110607/
[Accessed 24 February 2020].

Bos, B., n.d. *Maintainability.* [Online]
Available at: https://www.w3.org/People/Bos/DesignGuide/maintainability.html
[Accessed 15 February 2020].

CSS WG, 2020. *Cascading Style Sheets.* [Online]
Available at: https://www.w3.org/Style/CSS
[Accessed 18 February 2020].

Deveria, A. & Schoors, L., 2020. *Can I use... Support tables for HTML5, CSS3, etc.* [Online]
Available at: https://caniuse.com/#feat=const
[Accessed 22 February 2020].

Ecma International TC39, 2014. *Proposal Object Rest Spread.* [Online]
Available at: https://github.com/tc39/proposal-object-rest-spread
[Accessed 21 February 2020].

Ecma International, n.d. *History of ECMA.* [Online]
Available at: https://www.ecma-international.org/memento/history.htm
[Accessed 24 February 2020].

Ecma International, n.d. *What is ECMA.* [Online]
Available at: https://www.ecma-international.org/
[Accessed 24 February 2020].

Elliott, E., 2014. *Chapter 5. Separation of Concerns.* s.l.:O'Reilly Media, Inc..

Facebook Inc., 2020. *Facebook/Jest: Delightful JavaScript Testing..* [Online]
Available at: https://github.com/facebook/jest
[Accessed 23 February 2020].

Facebook Inc., 2020. *React – A JavaScript library for building user interface.* [Online]
Available at: https://reactjs.org
[Accessed 17 February 2020].

Facebook Inc., 2020. *Reconciliation.* [Online]
Available at: https://reactjs.org/docs/reconciliation.html
[Accessed 29th February 2020].

Flanagan, D., 2006. *JavaScript - The Definitive Guide.* 5th ed. s.l.:O'Reilly Media, Inc..

Free Software Foundation, Inc., 2018. *Gzip - GNU Project - Free Software Foundation.* [Online]
Available at: https://www.gnu.org/software/gzip/
[Accessed 23 February 2020].

Google LLC, 2018. *Minify Resources (HTML, CSS, and JavaScript).* [Online]
Available at: https://developers.google.com/speed/docs/insights/MinifyResources
[Accessed 23 February 2020].

Google LLC, 2020. *Clear cache & cookies.* [Online]
Available at: https://support.google.com/accounts/answer/32050
[Accessed 23 February 2020].

Greif, S. & Benitte, R., 2019. *The State of CSS 2019: Pre & Post Processors.* [Online]
Available at: https://2019.stateofcss.com/technologies/pre-post-processors/
[Accessed 20 February 2020].

Greif, S. & Benitte, R., 2019. *The State of CSS 2019: Technologies.* [Online]
Available at: https://2019.stateofcss.com/technologies/
[Accessed 24 May 2020].

Greif, S. & Benitte, R., 2019. *The State of JS 2019: Testing.* [Online]
Available at: https://2019.stateofjs.com/testing/
[Accessed 22 February 2020].

Greif, S., Benitte, R. & Wattenberger, A., 2019. *The State of JavaScript 2019: Overview.* [Online]
Available at: https://2019.stateofjs.com/overview/
[Accessed 24 May 2020].

Hinkelmann, F., 2017. *Understanding V8's Bytecode.* [Online]
Available at: https://medium.com/dailyjs/understanding-v8s-bytecode-317d46c94775
[Accessed 23 February 2020].

Johnson, S. C., 1978. *Lint, a C Program Checker.* s.l.:s.n.

Microsoft Corporation, 2020. *TypeScript Documentation.* [Online]
Available at: https://www.typescriptlang.org/docs/home.html
[Accessed 24 2 2020].

Mitton, R., 2015. *A Note On Layout Language.* [Online]
Available at: http://www.codersnotes.com/notes/a-note-on-layout-language/
[Accessed 21 February 2020].

Mozilla and individual contributors, 2020. *Array.prototype.concat() - JavaScript | MDN.* [Online]
Available at: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/concat
[Accessed 22 February 2020].

Nguyen, H., Johnson, B. & Hackett, M., 2003. *Testing Applications on the Web (2nd Edition).* 2nd ed. s.l.:Wiley.

Occhino, T. & Walke, J., 2013. *Tom Occhino and Jordan Walke: JS Apps at Facebook..* [Online]
Available at: https://youtu.be/GW0rj4sNH2w
[Accessed 17 February 2020].

Poole, C. & Prouse, R., 2019. *NUnit.org.* [Online]
Available at: https://nunit.org/
[Accessed 22 February 2020].

Prettier.io, 2020. *What is Prettier?.* [Online]
Available at: https://prettier.io/docs/en/index.html
[Accessed 24 February 2020].

Robie, J., 2020. *What is the Document Object Model?.* [Online]
Available at: https://www.w3.org/TR/WD-DOM-Level-2/introduction.html
[Accessed 14 February 2020].

Wagner, J. & Osmani, A., 2018. *Reduce JavaScript Payloads with Code Splitting.* [Online]
Available at: https://developers.google.com/web/fundamentals/performance/optimizing-javascript/code-splitting
[Accessed 23 February 2020].

Walke, J., 2011. *FaxJS: Fax JavaScript UI Framework https://github.com/jordwalke/FaxJs.* [Online]
Available at: https://github.com/jordwalke/FaxJs
[Accessed 17 February 2020].

Webpack, 2020. *Optimization* | *Webpack.* [Online]
Available at: https://webpack.js.org/configuration/optimization/#optimizationminimizer
[Accessed 23 February 2020].

WHATWG, 2020. *HTML Living Stardard.* [Online]
Available at: https://html.spec.whatwg.org/
[Accessed 24 05 2020].

WHATWG, 2020. *HTML Standard.* [Online]
Available at: https://html.spec.whatwg.org
[Accessed 17 February 2020].

World Wide Web Consortium, 2016. *https://www.w3.org/standards/webdesign/.* [Online]
Available at: https://www.w3.org/standards/webdesign/
[Accessed 24 05 2020].

You, E., 2019. *dotJS 2019 - Evan You - State of Components.* [Online]
Available at: https://www.youtube.com/watch?v=bOdfo5SmQc8
[Accessed 25 February 2020].

Zaytsev, J., 2020. *ECMAScript 6 compatibility table.* [Online]
Available at: https://kangax.github.io/compat-table/es6/
[Accessed 22 February 2020].