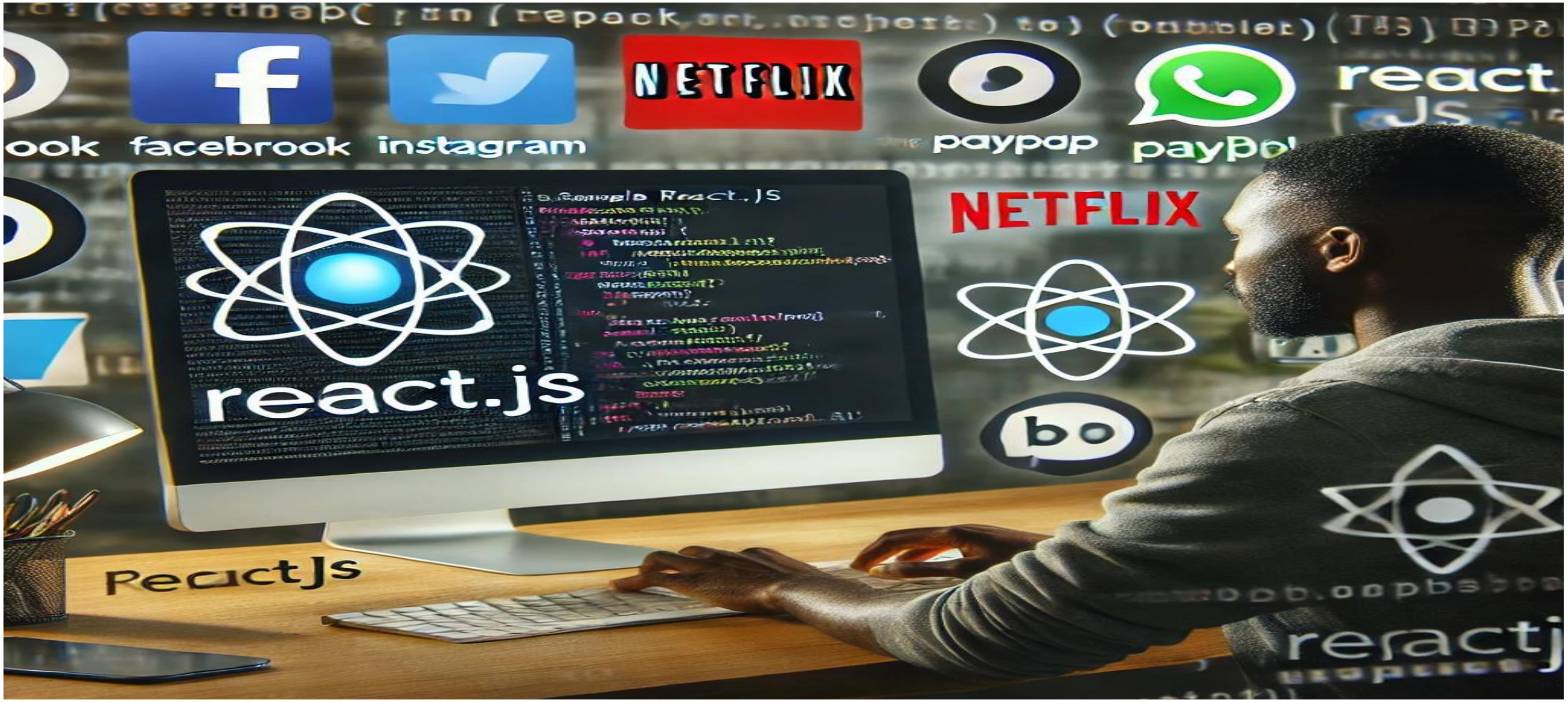
MODULE CODE AND TITLE: SWDFA501 FRONTEND APPLICATION DEVELOPMENT WITH REACT.JS

Learning Outcome 1: Develop React.Js Application



Indicative contents

1.1 Preparation of React.Js Environment
1.2 Applying React Basics
1.3 Applying UI Navigation
1.4 Applying React Hooks
1.5 Implementation of Events Handling
1.6 Implementation of API Integration

1.1: Preparation of React.Js Environment

Description of key concepts

❖ ReactJS

ReactJS is a popular JavaScript library for building user interfaces or reusable UI Components, particularly single-page applications where you need a fast, interactive experience. To create a React JS application we need NodeJS runtime environment which will provide npm services to download and install ReactJs packages like Router, redux etc.

▪ Component

A component in React is an independent, reusable piece of UI. Components can be class-based or function-based.

Functional components are simpler, defined as JavaScript functions, and use hooks like useState and useEffect for managing state and lifecycle.

Class components are ES6 classes, requiring render() for JSX output and have lifecycle methods like componentDidMount.

Functional components are preferred in modern React for being more concise and easier to test, with hooks offering greater flexibility. Class components were traditionally used for state management before hooks, but are now less common due to the advantages of functional components.

- JSX (JavaScript XML)

JSX is a syntax extension for JavaScript that looks similar to XML or HTML. It allows you to write HTML structures in the same file as JavaScript code.

- **Props**

Props (short for properties) are read-only attributes that are passed from a parent component to a child component. **They allow data to be passed around your application.** Props are immutable, meaning that **once they are passed to a component, the component cannot modify them.** Instead, they are used to render dynamic content based on the values passed.

State

State is a built-in object that holds data or information about the component. Unlike props, state is managed within the component and can be changed over time, usually as a result of user actions.

Lifecycle Methods

Lifecycle methods are special methods in React components that allow you to run code at particular times in the component's lifecycle, such as when the component mounts, updates, or unmounts.

Hooks

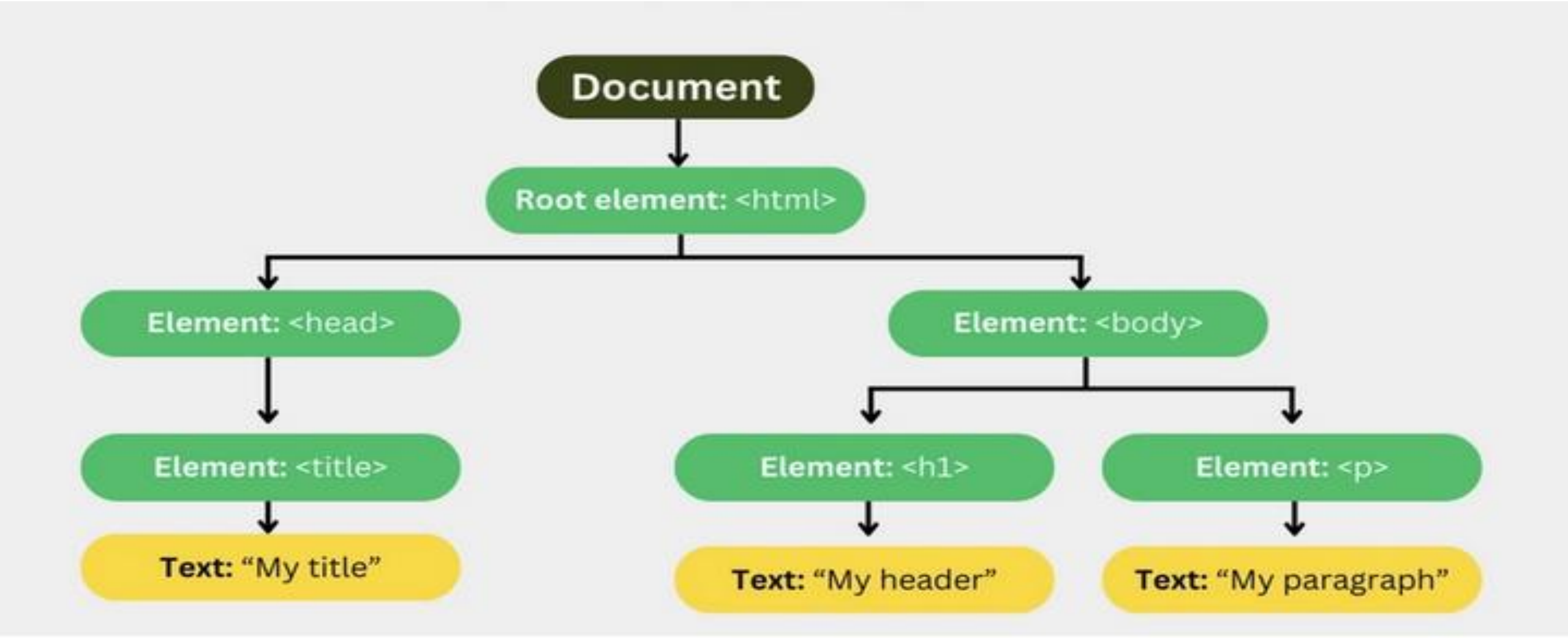
Hooks are functions that let you use state and other React features in functional components. Examples include useState, useEffect, and useContext.

Virtual DOM

The Virtual DOM is a lightweight copy of the actual DOM that React uses to optimize rendering. React updates the Virtual DOM, then efficiently updates the real DOM to match.

DOM

The DOM (Document Object Model) is a programming interface for HTML and XML documents that represents the document's structure as a tree of nodes, making it accessible and modifiable by scripting languages like JavaScript.



- React Router

React Router is a library for routing in React applications. It enables navigation among views of various components in a React application, allowing you to change the browser URL and keep the UI in sync with the URL.

- Redux

Redux is a predictable state container for JavaScript apps, often used with React to manage the application's state in a centralized store.

- Uses of reactJS

- Building dynamic web applications
- Developing single-page applications (SPAs)
- Creating reusable UI components

Features of ReactJS

- Declarative syntax: React uses declarative programming, meaning that instead of manually telling the system how to change the state of the UI, you declare the desired state, and React will manage the transitions and updates automatically.
- Component-based architecture: React allows developers to build the UI by breaking it into reusable, self-contained components. Each component manages its own state and logic, making development modular and maintainable.
- Efficient rendering using the Virtual DOM: React utilizes a Virtual DOM to improve performance. Instead of updating the real DOM (which can be slow), React keeps a lightweight copy (the Virtual DOM) and only updates the real DOM when necessary.
- Rich ecosystem with tools like React Router and Redux: A routing library that allows you to manage navigation and views in a React app. It helps with dynamic routing and renders components based on the current URL. A predictable state management library. It helps in managing the entire application's state in a single store, making it easier to debug and maintain applications with complex states.

Application of ReactJS

As a **JavaScript framework**, React.js has a wide range of applications due to its flexibility, component-based architecture, and efficient rendering system. Here are key applications of React.js as a JS framework:

Single Page Applications (SPAs)

React is widely used to build **SPAs**, which load a single HTML page and dynamically update content based on user interaction without reloading the page. React's virtual DOM and component structure allow for efficient updating of the UI, making it perfect for SPAs.

Examples: Gmail, Facebook, Instagram.

- Dynamic and Interactive Web Interfaces

React is ideal for building **highly interactive and dynamic user interfaces** that require real-time updates or frequent UI changes. The declarative nature of React allows developers to efficiently manage the state of an application and reflect changes in the UI without complex DOM manipulation.

Examples: Online marketplaces, social networking sites, news feeds.

Progressive Web Applications (PWAs)

React is used to build **PWAs** that offer a mobile-app-like experience within a browser. PWAs are responsive, work offline, and provide fast loading times. React's component-based architecture and efficient rendering system make it a suitable choice for creating PWAs.

Examples: Twitter Lite, Pinterest.

▪ Mobile Applications (React Native)

React.js powers **React Native,** a framework for building native mobile applications using the same codebase as a web app. React Native allows developers to create **cross-platform mobile apps** for iOS and Android by reusing components across both platforms.

Examples: Facebook, Instagram, Airbnb.

Content Management Systems (CMS)

React can be used to create **content management systems** or integrate with existing CMS platforms. Its component structure allows for dynamic previews, content updates, and real-time interaction, making it an ideal solution for CMS-based applications.

Examples: Netlify CMS, WordPress (with headless CMS).

E-Commerce Platforms

React is commonly used to build **e-commerce websites** that need fast page loads, dynamic product filtering, seamless navigation, and a smooth checkout experience. React's modularity allows developers to create reusable components like product cards, filters, and carts.

Examples: Shopify, Walmart, Amazon.

Data-Driven Dashboards and Analytics

React is well-suited for building **data visualization and analytics dashboards.** React can efficiently manage frequent data updates and render complex visualizations using libraries like D3.js or Chart.js. Dashboards built with React offer interactive elements like filtering, sorting, and real-time updates.

Examples: Financial dashboards, Google Analytics, health tracking platforms.

Enterprise Web Applications

React is widely used for **enterprise-level applications** that require scalable, maintainable, and high-performance web interfaces. Applications such as ERP (Enterprise Resource Planning), CRM (Customer Relationship Management), and HR systems benefit from React's modular and reusable components.

Examples: Salesforce, SAP Fiori, Jira.

Real-Time Applications

React.js is excellent for building **real-time applications** like chat applications, collaboration tools, and streaming services. By leveraging technologies like WebSockets or Firebase, React can handle real-time data efficiently, making it ideal for such use cases.

Examples: WhatsApp Web, Slack, Zoom.

Form-Based Applications

React is also suitable for building complex, interactive **form-based applications** that require real-time validation, conditional rendering, and smooth user interaction. These can range from survey forms to financial applications that require dynamic inputs.

Examples: Tax filing apps, survey tools, insurance calculators.

Search Engines and Filtering Tools

React is used in creating powerful **search engines and filtering tools** with instant feedback. Its ability to handle state management and dynamic updates makes it perfect for providing real-time filtering and search results.

☐ **Examples:** E-commerce product searches, library search systems.

Interactive Educational Platforms

React can be used to create **educational platforms** with interactive features like quizzes, videos, progress tracking, and discussion boards. Its modular approach allows for the creation of reusable components for courses and lessons.

Examples: Coursera, Udemy, Khan Academy.

Social Media Platforms

React.js is a go-to framework for building **social media platforms**, given its ability to handle large-scale, dynamic content, real-time updates (like posts, comments, and likes), and seamless user experience.

Examples: Facebook, Instagram, Twitter.

Advantages of using react

- Faster debugging and rendering.

ReactJS offers interaction for any UI layout and is simple to use. Additionally, it enables the rapid and quality assured development and rendering of applications, saving time for clients and React developers.

- **Reusability of components.**

React developers may leverage the reusable components that ReactJS offers to build new applications. Each of its web application's several components has its logic and control. These parts produce a short, reusable chunk of html code that you use anywhere you need it.

- **Readily available JavaScript libraries**

A robust JavaScript and HTML syntax mix streamlines the entire code development process for the intended project. The reason the majority of web React developers prefer ReactJS is that it provides a robust JavaScript library.

experiences for all browsers and search engines, making it easier for React developers to be found on different search engines.

- **Use of virtual Document Object Model or V-DOM.**

ReactJS leverages Virtual DOM, an API programming cross-platform interface that deals with web development to enhance performance. When creating an app with lots of user interaction and data exchanges, it's essential to test how the structure of your app will affect speed. Despite of fast client platforms and JavaScript engines, extensive DOM manipulation can affect the platforms operating speed and produce an uncomfortable user experience. React addresses this by employing a virtual DOM, representing the DOM of a web browser that lives in memory. As a result, we avoid writing to the DOM when creating React Components; instead make virtual components that respond and transform into DOM, resulting in more seamless and faster performing platforms.

Great for SEOs

Search engine optimization(SEO) makes it more straightforward for search engines to identify relevant material for users. When a user searches, the search engine determines which page is the most pertinent to that particular search.
Conventional JavaScript frameworks need help to handle SEO. JavaScript-heavy apps have problems being read by search engines. ReactJS solves this issue by enabling search engine-optimized user experiences for all browsers and search engines, making it ct Developers to be found on different search engines.

Easy to learn and use.

ReactJS trains the programmer how to traverse the framework and offer you new thought patterns that might help you in other fields of programming. it contains a wealth of information, video, and teaching materials and is simple to use and pick up.

Active React JS Community.

ReactJS boast one of the most significant communities of react Developers and designers. React code is also open source, allowing anybody to download, edit, and distribute it with others to develop the framework.

Readily available tools for experimenting

React enables you to view the virtual DOM's React component hierarchies. It consists of a browser plugin called React Developer Tools that are available for both Chrome and Firefox, using which any component's hierarchy may be tracked, allowing you to find both parent-child components.

Worldwide usage of ReactJS by prominent brands.

For their applications, websites and internal projects, ReactJS has been selected by thousands of businesses worldwide. Companies like Walmart Labs, Tencent, Tesla, Bloomberg, Airbnb, Baidu Mobile, GoDaddy, Gyroscope, and many more developed mobile apps using Reactnative. Handreds of websites use React, including Coursera, Paypal, Netflix, Dailymotion, IMDb, Chrysler, BBC, American Express, Khan Academy,Lyft, New York Times, DropBox, Reddit, and others.

1.1.2: Installing NodeJS and Node Package manager

There are steps to install

Step 1: Download Node.js

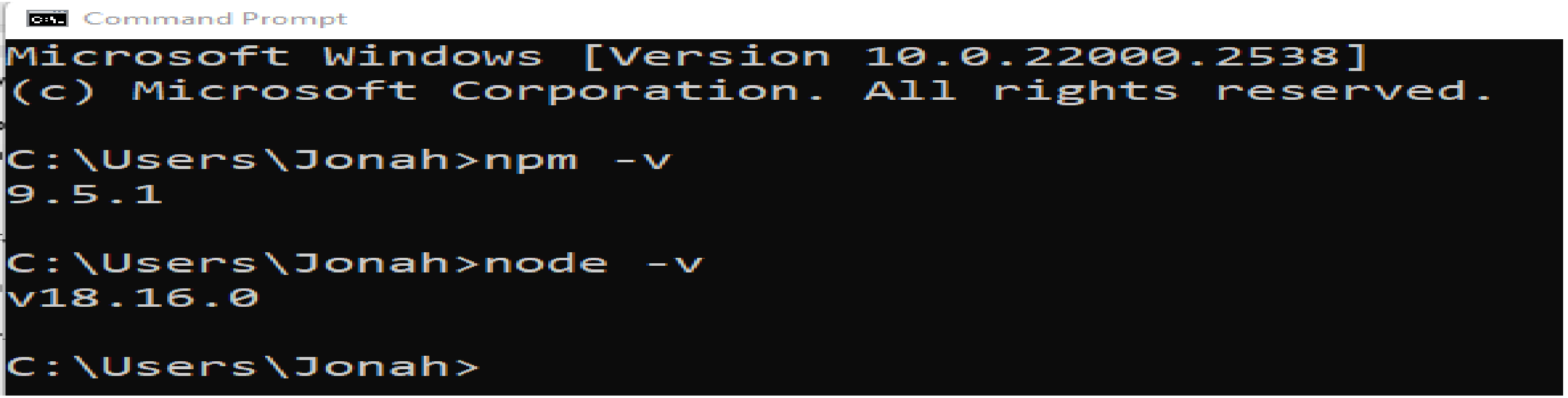
- Go to the official Node.js website: https://nodejs.org/
- Download the Windows Installer (.msi) file for the LTS (Long-Term Support) version.

Step 2: Install Node.js

- Run the downloaded installer.
- Follow the prompts in the Node.js Setup Wizard, making sure to check the box that says "Automatically install the necessary tools" (this installs NPM along with Node.js).
- Restart your system if required.

Step 3: Verify the installation

- Open Command Prompt.
- Check Node.js version by typing the command below in Command Prompt: node -v
- Check NPM version by typing the command below in Command Prompt: npm –v



1.1.3: Creating ReactJS application

Steps to create of ReactJS application

Step 1. Install and create a new React project:

Run npx create-react-app my-app Command

This will create a new directory named my-app with the basic structure of a React app.

```
npm install react react-dom react-scripts cra-template

Microsoft Windows [Version 10.0.22000.2538]
(c) Microsoft Corporation. All rights reserved.

E:\ReactApps>npx create-react-app my-app

Creating a new React app in E:\ReactApps\my-app.

Installing packages. This might take a couple of minutes.
Installing react, react-dom, and react-scripts with cra-template...

[                              ] / idealTree:my-app: sill idealTree buildDeps
```

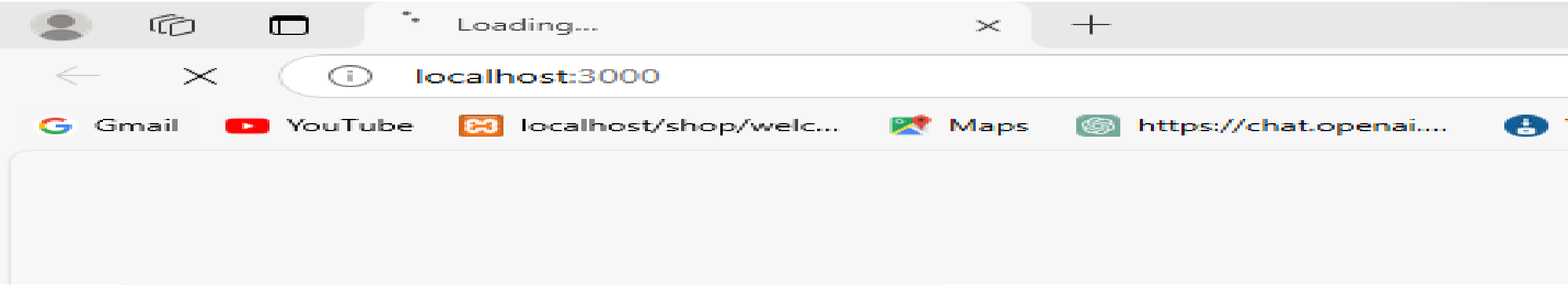
Step 2. Navigate into the project directory:

Run Command: cd my-app

Step 3. Start the development server:

Run Command: npm start

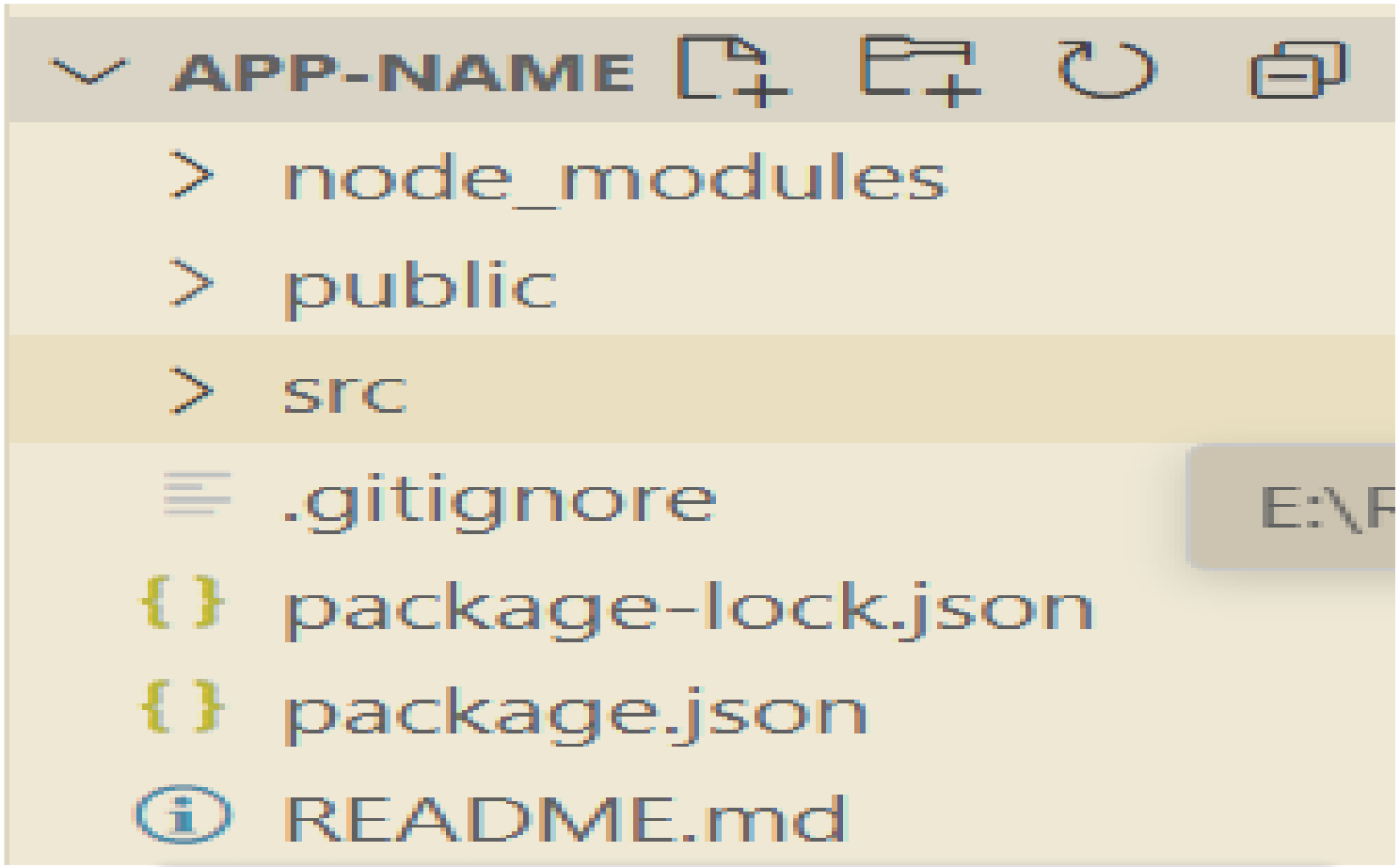
This will start the local development server, and you should be able to see your React app running at http://localhost:3000 in your browser.



1.1.4: Exploring of React project structure

The root directory of your React project contains several important files and folders as shown in the screen shot below.

The following are steps to explore the project structure



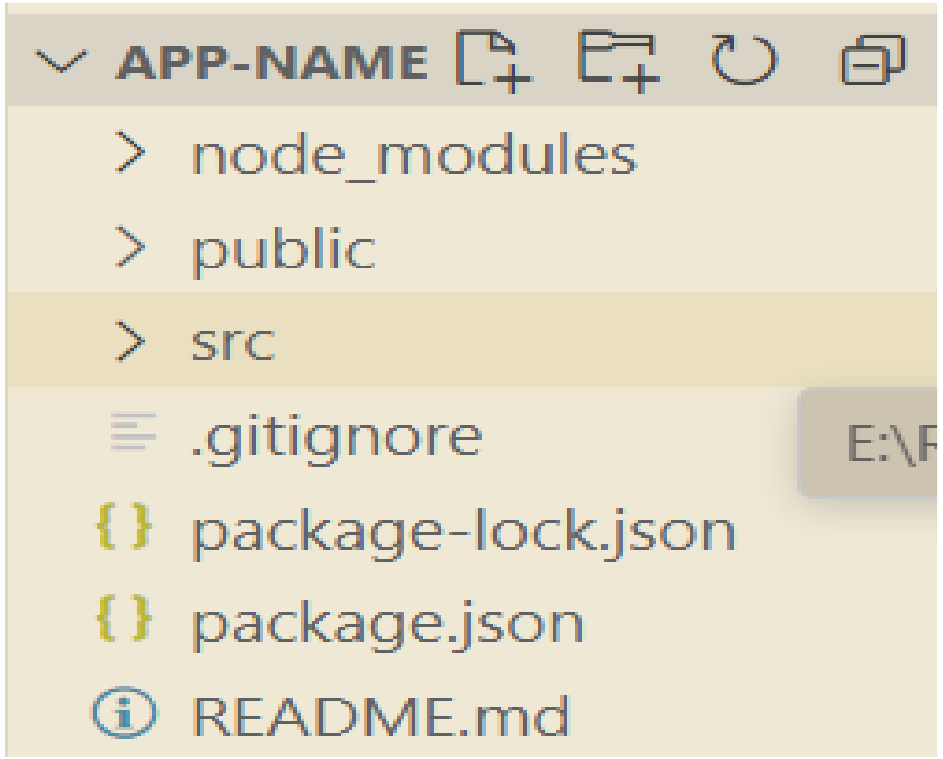
Explanation of common folders and files

Step 1. Browse to the folder where your project is created

Step 2. Open Command prompt

Step 3. Type the command to open the project in your text editor

Step 4. View and analyse every file and folder from the project explorer in the text editor as shown below



Explanation of common folders and files

- node_modules/: Contains all the dependencies and packages installed via npm or yarn. You usually don't interact with this directly.
- public/: This folder holds static assets that are served directly by the server without any modification. Key files include:

  - index.html: The single HTML file that React injects into. This file is responsible for displaying the React app in the browser.
  - favicon.ico: The icon displayed on the browser tab.
  - manifest.json: Used for Progressive Web Apps (PWA). It provides metadata about the app.
  - robots.txt: Instructions for search engine bots.

- src/: The main source folder for all the React code. Most of your development happens here.
- package.json: Lists all dependencies, scripts, and metadata about the project.
- README.md: A markdown file that usually contains information about the project and instructions for setup.

❖ .gitignore: Specifies which files/folders to ignore in version control (like Git).

1.1.5: Installing of additional React tools and libraries

Method 1: Installing a single package

✓ Install Redux

Run command: npm install redux react-redux

✓Install React Router

Run command: npm install react-router-dom

✓Install Axios

Run command: npm install axios

Method 2: Install all packages using single line of command

npm install redux react-redux react-router-dom axios

1.2: Applying React Basics

1.2.1: Creating React Class Components

Steps to Create a Class component in an existing ReactJS Project

Step 1. Create the Component File

Step 2. Inside your src folder, create a new file for your component. For example, you can call it App.js.

Step 3. Write the Class Component.

Step 4. In the App.js file, define the class component as follows.

Welcome | JS index.js | JS App.js

src > JS App.js > default

```
1   import { Component } from "react";
2
3   class App extends Component{
4       render(){
5           return (<h1>My Class Component</h1>)
6       }
7   }
8   export default App;
```

Step 5. Import and Use the Component.

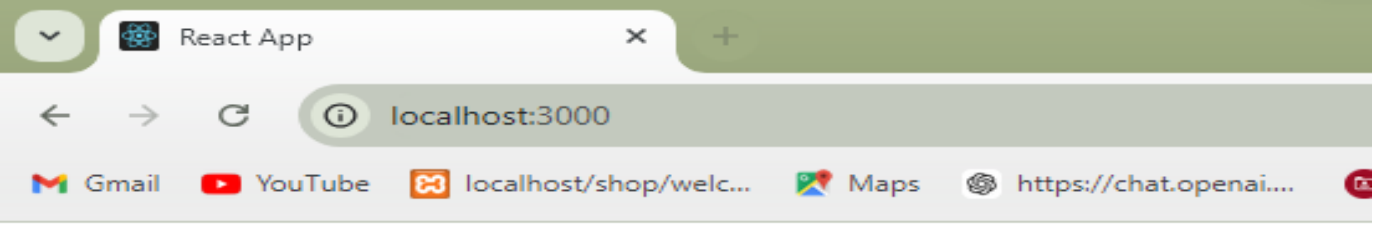
Step 6. In your src/index.js file (the entry point for your React app), ensure that your App component is imported and used correctly.

me JS index.js × JS App.js

index.js > ...

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <App />
);
```

Step 7. Run the Application



My Class Component

1.2.1: Creating React Functional Components

Steps to Create a functional component in an existing ReactJS Project

Step 1. Create the Component File

Step 2. Inside your src folder, create a new file for your component. For example, you can call it App.js.

Step 3. Write the functional Component

Step 4. In the App.js file, define a functional component as follows.

Breakdown of Functional Component

Declaration: The function App is declared using the arrow function syntax (const App = () => { ... }).

Return Statement: The component returns JSX, which looks like HTML but is written in JavaScript.

Exporting the Component: The export default App; statement **is used to make this component available for import in other parts of the app.**

Welcome | JS index.js | JS App.js

src > JS App.js > [e] default

```
1
2  const App = () =>{
3      return(<h2>Functional based Components</h2>)
4  }
5  export default App
```

Step 5. Import and Use the Component

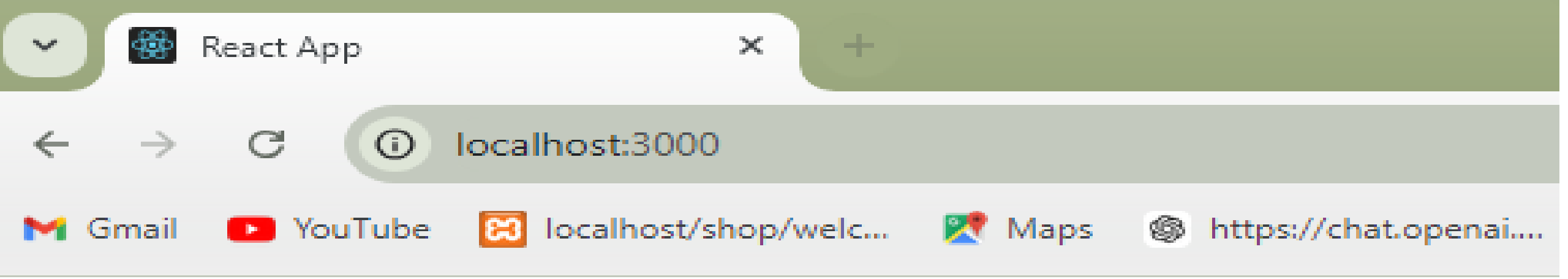
Step 6. In your src/index.js file (the entry point for your React app), ensure that your App component is imported and used correctly

me JS index.js × JS App.js

index.js > ...

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <App />
);
```

Step 7. Run the Application



Functional based Components

1. Using Visual Studio Code, create a new directory and open the integrated terminal.
2. Type npx create-react-app react-js-foundations into the terminal and press Enter.
3. Once Create React App finishes its work, type cd react-js-foundations followed by npm start. Create React App will start up your application and open it in a browser.
4. Open src/App.js in Visual Studio Code.
5. Update App.js to match Listing 1-3 and then save the file.

LISTING 1-3: An interactive Hello, World component

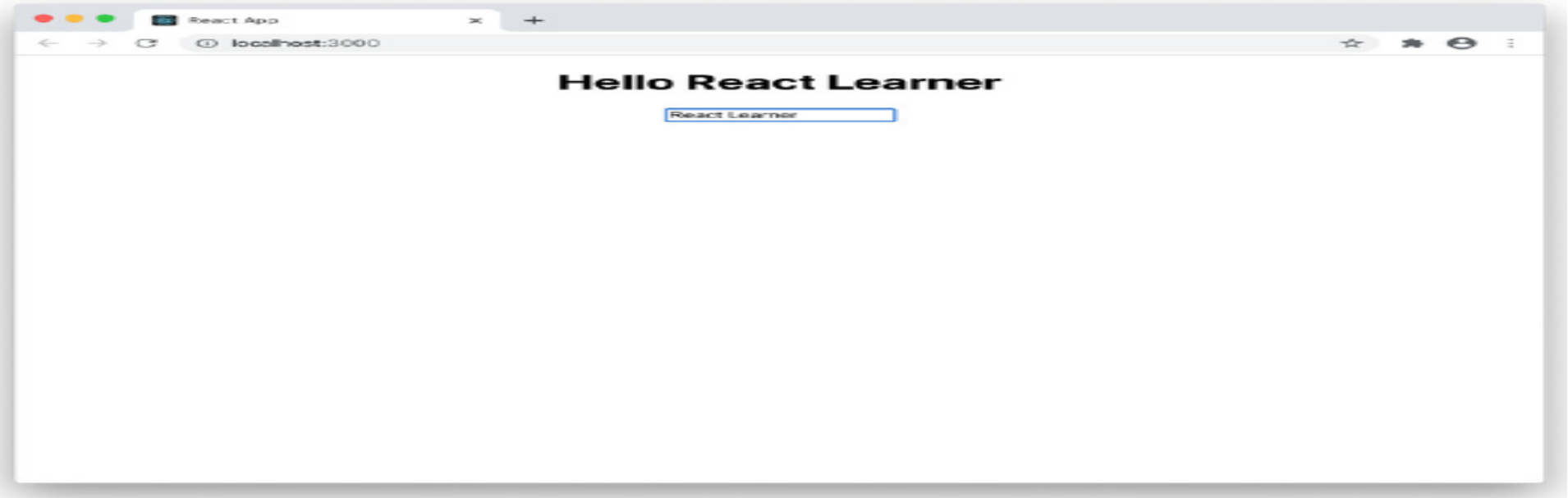
```
import React from 'react';
import './App.css';
function App() {
const [personName,setPersonName] = React.useState('');
return (
<div className="App">
<h1>Hello {personName}</h1>
<input type="text" onChange={(e) =>
setPersonName(e.target.value)}/>
</div>
);
}
export default App;
```

6.Return to your browser, and notice that the default Create React App screen has been replaced with an input field and an h1 element above it.

NOTE This ability of an app running in Create React App to detect when files have changed and update what's showing in the browser without you having to manually reload the page is called "hot reloading."

7. Type into the input field. Everything you type should appear inside the h1 element, as shown in Figure 1-3.

FIGURE 1-3: The finished interactive Hello, World component!



8.When you're done playing around with this component, return to the built-in terminal in VS Code and press **Ctrl+c** to stop the recompiling and hot reloading script.

The Foundation of React

React is a JavaScript library for creating interactive user interfaces using components. It was created by Facebook in 2011 for use on Facebook's newsfeed and on Instagram. In 2013, the first version of React was released to the public as open source software. Today, it's used by many of the largest websites and mobile apps, including Facebook, Instagram, Netflix, Reddit,Dropbox, Airbnb, and thousands of others.

Writing user interfaces with React requires a bit of a shift in how you think about web applications.

You need to understand what React is, how it works at a higher level, and the computer science ideas and patterns that it's based on. In this chapter, you'll learn:

- Why it's called React.
- What a Virtual DOM does.
- The difference between composition and inheritance.
- The difference between declarative and imperative programming.
- The meaning of "idiomatic" with regard to React.

WHAT'S IN A NAME?

Let's start with the name "React." Facebook designed React in response to its need to be able to efficiently update websites in response to events. **Events that can trigger updates in websites include user input, new data coming into the application from other websites and data sources,and data coming into the application from sensors (such as location data from GPS chips).**

Facebook wanted to create a way to more easily build applications that respond, or react to new data, rather than simply refreshing pages whether the underlying data has changed or not. You can think of the difference in approaches as *pull* (which is the traditional way of updating websites) vs. *push* (which is the reactive way to build websites).

This method of updating a user interface in response to data changes is called *reactive programming*.

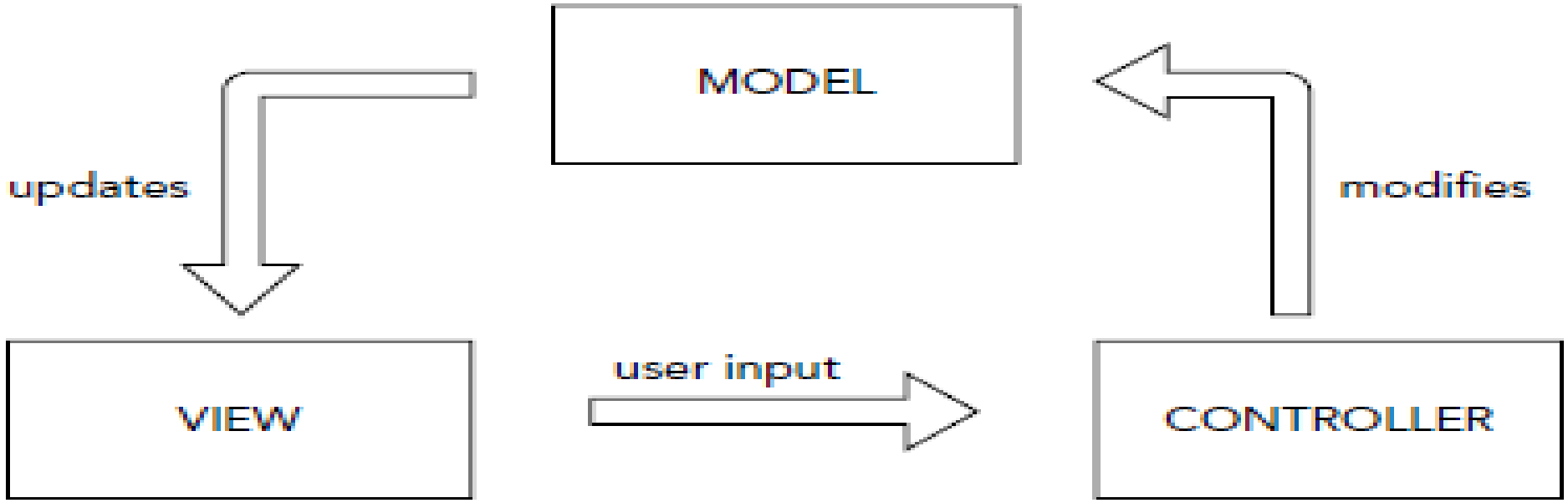
UI LAYER

Web applications typically are built and described using the Model-View-Controller (MVC) pattern.

The Model in MVC is the data layer, the Controller facilitates communication with the data layer, and the View is what the user sees and interacts with. In an MVC application, the View sends input to the Controller, which passes data between the data layer and the View. ***React is only concerned with the V in MVC. It takes data as input and presents it to the user in some form.***

Figure 2-1 shows a diagram of the MVC pattern.

FIGURE 2-1: The MVC pattern



React itself doesn't care whether the user is using ***a mobile phone, a tablet, a desktop web browser, a screen reader, a command-line interface, or any other kind of device or interface that may be invented in the future.*** React just ***renders components***. How those components get presented to the ***user is up to a separate library***.

The library that handles rendering of React components in web browsers is called ReactDOM. If you want to render React elements to **native mobile apps**, you use **React Native**. If you want to render React components to static HTML, you can use ReactDOMServer.

ReactDOM has a number of functions for interfacing between React and web browsers, but the one that every React application makes use of is called **ReactDOM.render**. Figure 2-2 illustrates the relationship between React, ReactDOM, and a web browser.

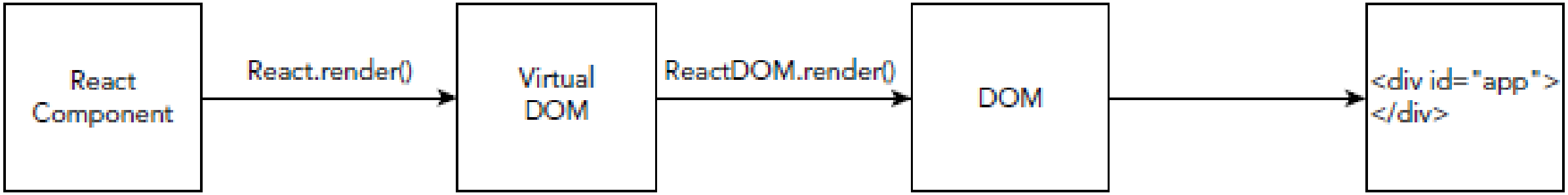


FIGURE 2-2: React and ReactDOM

ReactDOM ***is what makes it possible for user interfaces built in React to handle the quantity of screen changes required by modern web applications so efficiently***. It does this through the **use of a Virtual DOM**.

VIRTUAL DOM

The Document Object Model, or DOM, is a web browser's internal representation of a web page. It converts HTML, styles, and content into nodes that can be operated on using JavaScript.

If you've ever used the getElementById function or set the innerHTML of an element, you've interacted with the DOM using JavaScript. ***Changes to the DOM cause changes to what you see in your web browser, and updates made in the web browser (such as when you enter data into a form) cause changes to the DOM.***

Compared to other kinds of JavaScript code, DOM manipulation is slow and inefficient. This is because whenever the DOM changes, the browser has to check whether the change will require the page to be redrawn and then the redrawing has to happen.

Adding to the difficulty of DOM manipulation is that the DOM's functions aren't always easy to use and some of them have excessively long names like Document.getElementsByClassName. For both of these reasons, many different JavaScript DOM manipulation libraries have been created. The single most popular and widely used DOM manipulation library of all time was *jQuery*. **It gave web developers an easy way to make updates to the DOM, and that changed the way we build user interfaces on the web.**

Although jQuery made DOM manipulation easier, it left it up to programmers to program specifically when and how changes to the DOM would happen. The result was often inefficient user interfaces that were slower both to download and to respond to user interactions because of their use of jQuery.

As a result, jQuery got a reputation for being slow.

When the engineers at Facebook designed React, they decided to take the details of how and when the DOM is modified out of the hands of programmers. ***To do this, they created a layer between the code that the programmer writes and the DOM. They called this intermediary layer the Virtual DOM.***

Here's how it works:

1. A programmer writes React code to render a user interface, which results in a single React element being returned.
2. ReactDOM's render method creates a lightweight and simplified representation of the React element in memory (this is the Virtual DOM).
3. ReactDOM listens for events that require changes to the web page.
4. The ReactDOM.render method creates a new in-memory representation of the web page.
5. The ReactDOM library compares the new Virtual DOM representation of the web page to the previous Virtual DOM representation and calculates the difference between the two. This process is called ***reconciliation.***

6.ReactDOM applies just the minimal set of changes to the browser DOM in the most efficient way that it can and using the most efficient batching and timing of changes. By taking the programmer out of the process of actually making updates to the browser DOM, ReactDOM can decide on optimal timing and the optimal method for making required updates. This greatly improves the efficiency of making updates to a browser view.

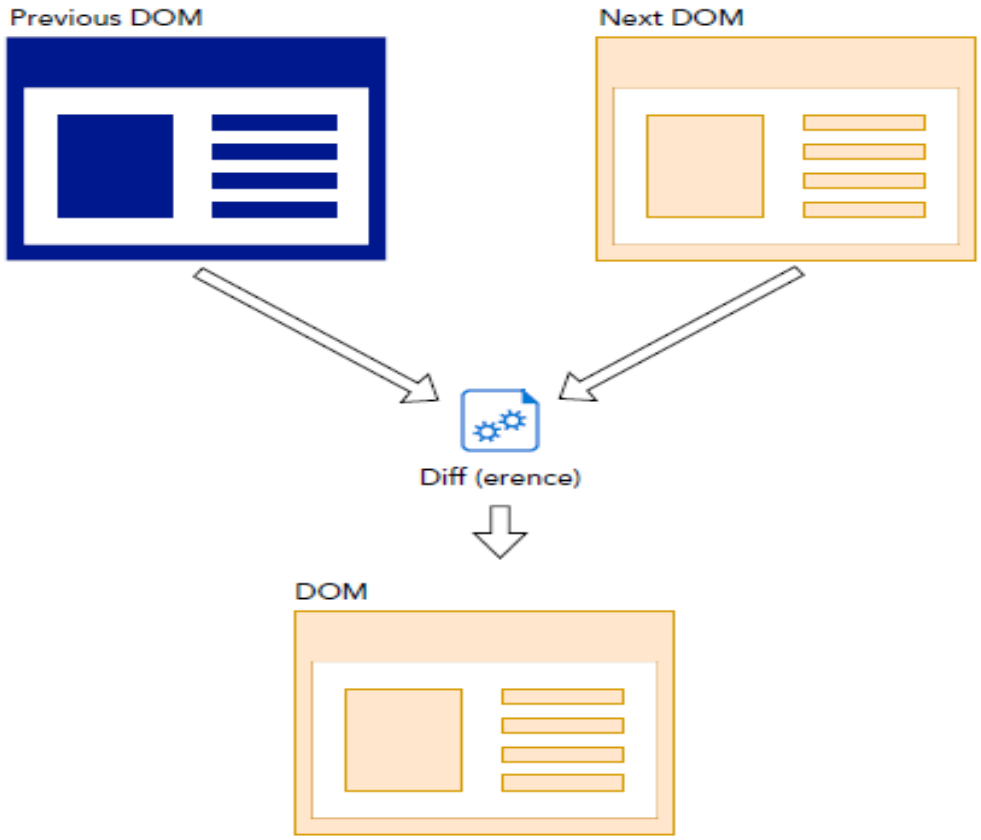
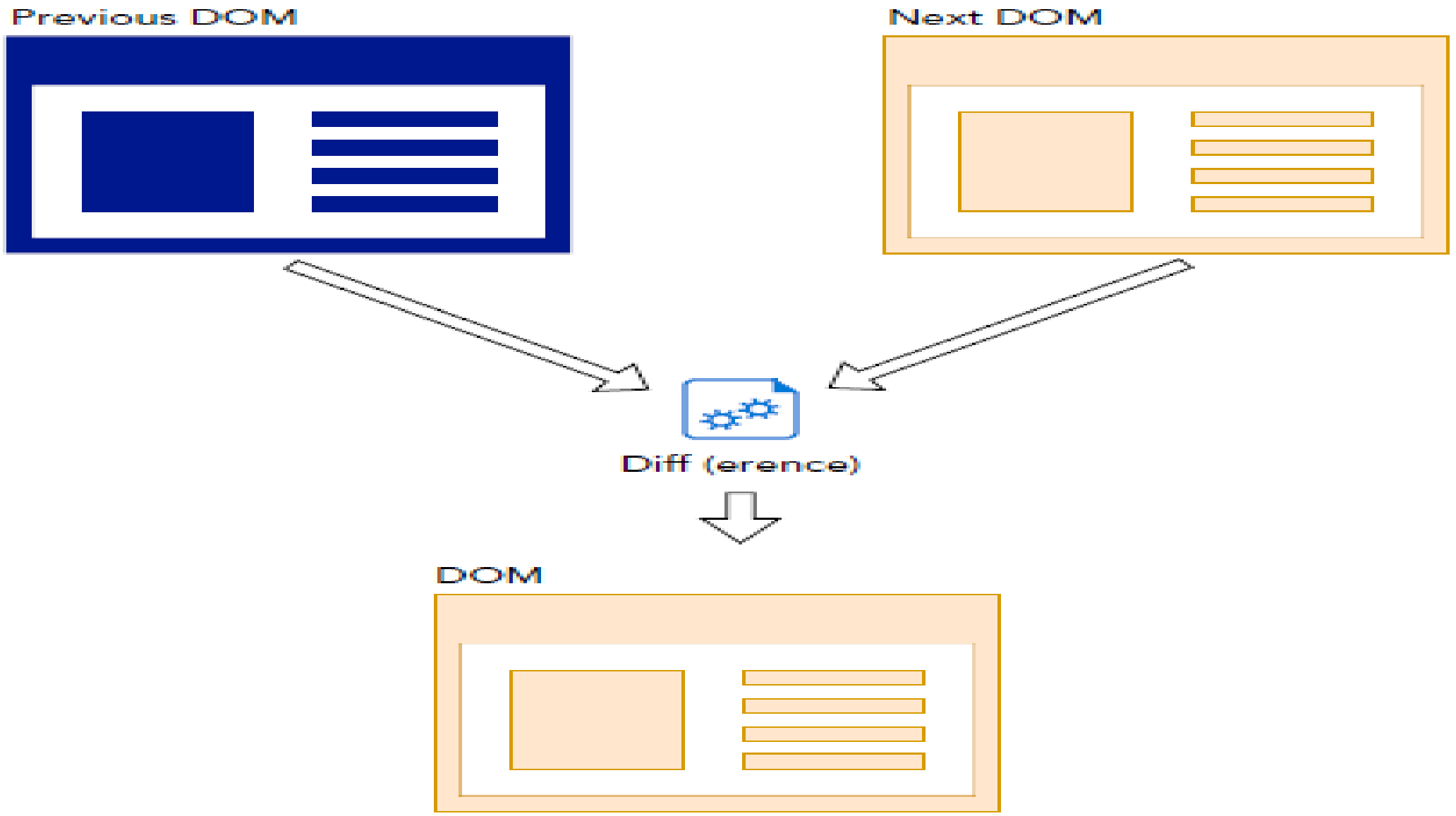


Figure 2-3 is a diagram showing how the Virtual DOM works.

FIGURE 2-3: How the Virtual DOM works



THE PHILOSOPHY OF REACT

If you've used other JavaScript libraries, you may find React to be quite a bit different from your past experience with programming dynamic user interfaces. By understanding the thinking behind why React is like it is, you'll have a better understanding and appreciation of it.

Thinking in Components

React is a library for creating and putting together (or *composing*) components to build user interfaces. React components are independent pieces that can be reused and that can pass data to each other.

A component can be something as simple as a button or it can be more complex, such as a navigation bar that's made up of a collection of buttons and dropdowns.

As the programmer, it's your job to decide how big or how small each component in your application should be, but a good rule of thumb to think about is the idea of ***single responsibility.***

The single responsibility principle, in programming, *is the idea that a component should have responsibility for a single part of a program's functionality*. Robert C. Martin, also known as "Uncle Bob," is one of the most important thinkers and writers on software design. He described the single responsibility principle this way:

Single Responsibility means that a class [or what we call a "component" in React] should have only one reason to change.

Composition vs. Inheritance

In object-oriented programming (OOP), it's common to create variations of classes that inherit properties from a parent class. For example, a program might have a class called Button, ***which might have a child class called SubmitButton. SubmitButton would inherit all of the properties of Button, and then override or extend them to create its unique functionality and look.***

Rather than using inheritance to create more specific components to deal with specific cases (such as a submit button), ***React encourages the creation of a single component that is more broadly reusable but that can be configured by passing data into it and then combining it with other components to handle more specific cases.***

For example, in the case of a submit button, you might simply pass a parameter to a Button component called **label** and another parameter called **handleClick** ***that contains the action to be performed by the button. This generalized button can then serve multiple purposes, depending on the values of label and handleClick that are passed to it.*** Listing 2-1 shows what this component might look like.

LISTING 2-1: Creating configurable components

```
function Button(props){
return(
<button onClick={props.handleClick}>{props.label}</button>
);
}
```

Once you've created a configurable component, you can create more specific components by combining more generalized ones. This technique is called ***composition***. Listing 2-2 shows how you can create a specific WelcomeDialog component from a general Dialog one using composition.

LISTING 2-2: Using composition

```
function Dialog(props){
return(
<div className="dialogStyle">{props.message}</div>
)
}
function WelcomeDialog(props){
return(
<Dialog message="Welcome to our app!" />
)
}
```

React Is Declarative

One way to describe the difference in approach between programming with React and programming with many other JavaScript libraries is to say that React is ***declarative*** while many other libraries **are *imperative*.**

By breaking down a complex process into small steps, a task eventually becomes simple enough for a computer to perform. **We call this step-by-Step style of programming *imperative programming*.** The imperative approach is the way that most DOM manipulation libraries worked prior to React.

React takes a different approach, which we call ***declarative programming***. In declarative programming, *the computer (or the computer language interpreter, rather) has some intelligence about the types of tasks that it can perform, and the programmer only needs to tell it what to do, rather than how to do it.*

In declarative programming, our sandwich-making robot would know the steps for making sandwiches, and programming it would involve the programmer saying something like "make me a sandwich that looks like this."

Applied to DOM manipulation, ***the declarative approach that React takes is to allow the programmer to say, "Make the page look like this." React then compares the new way that the page should look with the way that it currently looks and figures out what's different and what needs to change and how to do it.***

Building and updating a React user interface, from the programmer's perspective, is just a matter of specifying what the user interface should look like and telling React to render it.

React Is Idiomatic

React itself is a small library with limited functionality when compared to other JavaScript libraries. Except for a handful of concepts and methods that are unique to React, React components are just JavaScript.

the good news is that by getting better at JavaScript, you'll get better at programming with React.

You may hear the term "**idiomatic JavaScript"** used to describe React. What this means is that React code is easily understandable to people who program JavaScript. The reverse is also true: if you know JavaScript, understanding how to write React is not too much of a stretch.

Why Learn React?

React's popularity has been growing from day one of its release into the wild. React is going to be around for a long time to come, and there's never been a better time to learn it.

React vs....

One good way to learn a new language is by comparing it to something that you already know, and the question "how does React compare to (x)" is one of the most common questions that my students ask me.

React vs. Angular

Angular (angular.io) was created by Google, and it's been around longer than React in one form or another. Let's start with the similarities:

1. Purpose. Both Angular and React can create scalable and dynamic user interfaces.

2. Stability. Both Angular and React were created by one of the largest companies on the internet and they both have huge numbers of developers and enthusiasts.

3. Robustness. A major concern with any JavaScript library or framework is how safe, secure, and generally acceptable it is for enterprise development. Both Angular and React are popular and widely used in corporate software development.

4. License. Both frameworks use the MIT license, which allows for unlimited use and reuse for free as long as the original copyright and license notices are included in any copy of the source code.

- Angular is considered to be a "framework," while React calls itself a "library." The difference between a library and a framework is that a framework is usually an all-encompassing way of doing something, while a library is generally seen as a tool for a more specific purpose.

- The React library itself is a tool for making user interfaces out of components. Angular, **on the other hand, is a complete system for building front-end web applications.** By assembling components and libraries, React can do everything that Angular can do. But, if you need something smaller, such as to generate some HTML, React can do that as well.
- Angular has a steeper learning curve than React. In addition to the learning curve required to use the framework itself, Angular requires the use of Microsoft's TypeScript, which is a superset of JavaScript that's designed for the development of large applications. It's possible to use TypeScript to write React as well, **but with Angular it's a requirement.**
- Unlike React, Angular operates on the real DOM, rather than on a Virtual DOM, and it optimizes changes to the DOM by using an approach it calls Change Detection. When an event causes data changes in Angular, the framework checks each component and updates it as needed. You may recognize this approach as a more imperative approach (as compared to React's declarative approach) to DOM manipulation.

React and Angular also differ in how data flows within an application. React, as you'll see, uses **one-way data flow.** What this means is ***that every change that happens in the browser starts out as a change in the data model***. This differs from both Angular and Vue, which both feature two-way data binding, in which ***changes in the browser can affect the data model directly and changes to the data model can affect the view.***

React vs. Vue

Vue.js (vuejs.org) is a relative newcomer to the universe of JavaScript frameworks, but it has been growing in popularity and is now considered one of the top three, along with React and Angular.

- Like React and Angular, Vue is open source. Unlike React and Angular, however, Vue isn't backed or controlled by a large corporation. Instead, it's the work of many programmers and companies donating their skills to maintain and support it. This can be seen as either a plus or a minus, depending on your view of giant internet companies.

- Vue takes a middle ground between the bare-bones approach of React and the approach of Angular. Like Angular, it has built-in functionality for state management, routing, and managing CSS. But, like React, it's a small library (even smaller than React in terms of total kilobytes that must be downloaded to the browser) and how you use it is highly customizable.

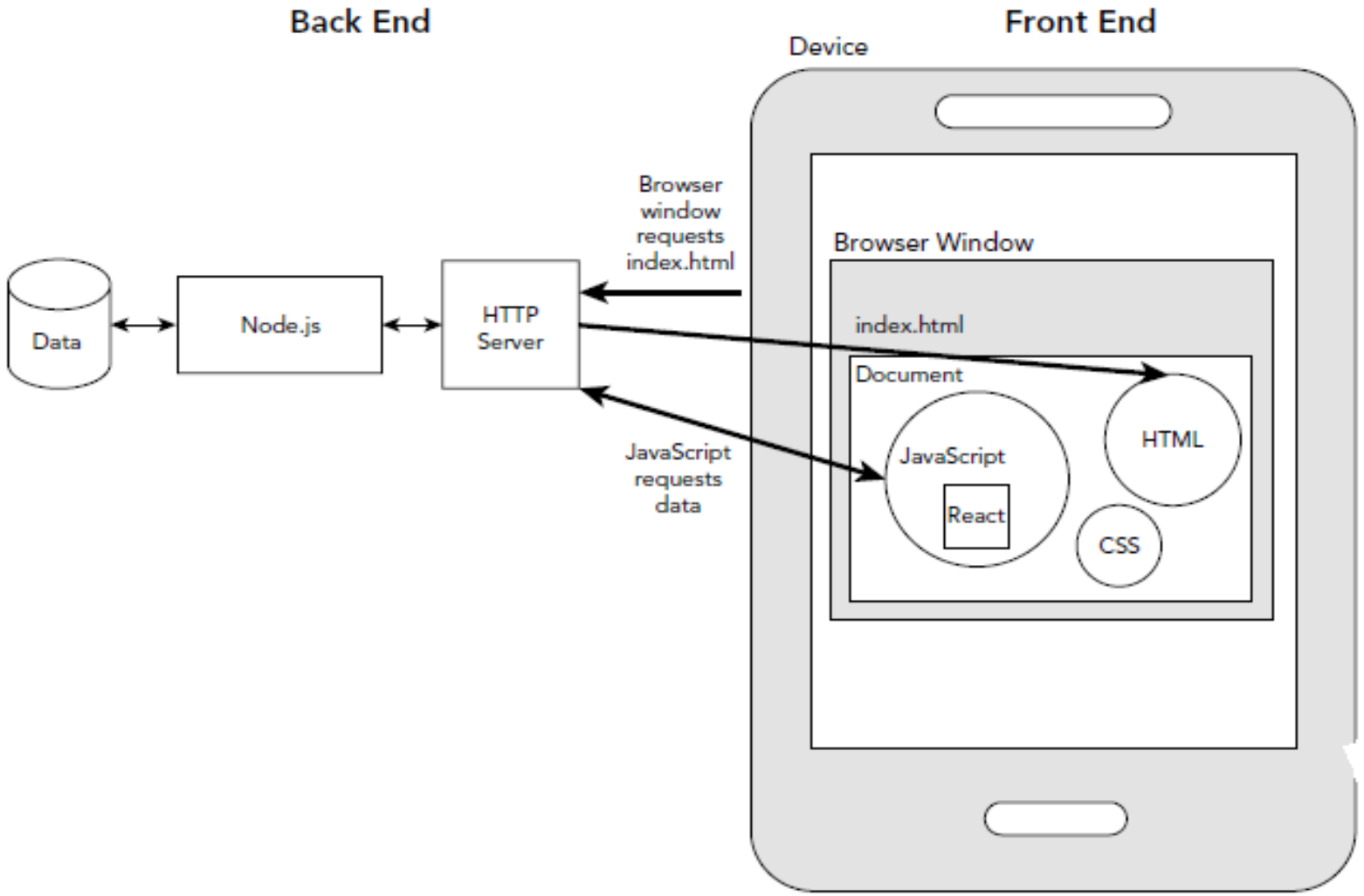
Of the three libraries, Vue is probably the easiest to learn.

What React Is *Not*

React, as it's used most often, is a front-end library. This means that everything React does happens in the web browser. It can't directly control or access remote databases (except through the web), and it doesn't serve web pages.

When I'm asked about the differences between front-end,back-end, and development environments, I like to draw something like the diagram in Figure 2-4.

FIGURE 2-4: How the web works



You know already that React is a front-end JavaScript library. As you can see from the preceding diagram, there are many other pieces of the web ecosystem. Here are a few of the things that React is *not*.

React Is Not a Web Server

A web server, also known as an HTTP server, has as its primary job to accept requests for web pages from web browsers and then return those web pages to the browser along with all of their linked images, CSS files, and JavaScript files.

- The React library and the user interfaces you create using React are among the linked JavaScript files that a web server sends to a web browser, and React itself has no ability to handle requests from web browsers, although **it can interact with the browser, just as any JavaScript code can, through the browser's application programming interfaces (APIs).**

React Is Not a Programming Language

React is a JavaScript library, which means that it is just a collection of JavaScript functions that programmers can make use of. The idea of a library is to simplify common tasks that programmers need to do frequently so that they can just focus on writing the code that makes a program unique.

If you had enough time and knowledge, you could rewrite every bit of the React library yourself using JavaScript—but, of course, there's no reason to do that, because the React developers have done it for you.

React Is Not a Database Server

React doesn't have any abilities to store data in a secure way or in a permanent way. Instead, *React user interfaces (like every web and mobile web user interface) communicate with server-side databases* ***over the internet to store and receive data such as login information, ecommerce transaction data, news feeds, and so forth.***

- The data that React uses to make user interfaces dynamic, and the data that you're viewing at any one time in a React user interface (what we call "session" data), is not persistent. Unless your React user interface saves session data in the web browser (using cookies or the browser's local storage), it all gets erased when you navigate to a different URL or refresh your browser window.

React Is Not a Development Environment

As you saw in the book's Introduction, you'll use plenty of different tools to program with React. Collectively, these are known as your development environment. I present the most commonly used tools (and, in some cases, just my own personal preferences), but there's nothing about React that requires you to use these specific tools. In many cases, there are alternatives available, and you may discover that you prefer different ones than I do (or, you may discover a better tool that I'm not currently aware of). It's possible to write perfect React code using any tools you want, or using no tools at all (other than a text editor).

React Is Not the Perfect Solution to Every Problem

React works well for many types of applications, but it's not always the best solution. This is true of any JavaScript library, and it's probably true of every single tool ever invented. It's important to know about a wide variety of different languages and libraries so that you can make good choices and know when the tools you prefer to use or that you know the best are the best and when they might not be the best.

SUMMARY

Because React is a different way of writing user interfaces for the web, it does have some concepts and foundational ideas behind it that are important to understand before you can work with it effectively. In the end, however, writing React user interfaces is straightforward:

1. You write components to describe how the user interface should look and act.
2. React renders your components to create a tree of nodes.
3. A React renderer figures out the differences between the latest rendered component tree and the previous one and updates the user interface accordingly.

In this chapter, you learned:

- Why React is called React, and what is meant by the term “reactive programming.”
- The purposes of the React library and of the ReactDOM library.

What composition is.

- About declarative programming and how it’s different from imperative programming.
- Why you should learn React.
- How React compares to Angular and Vue.js.
- The role of React in a web application and what roles React does not fill within the web application ecosystem.

JSX

Newcomers to React often remark on how it appears that React breaks one of the cardinal rules of web development, which is to not mix your programming logic with your HTML.

This chapter explains where this misperception about React comes from and introduces JSX, which gives us an easy, HTML-like syntax for composing React components. In this chapter, you'll learn:

- How to write JSX.
- How modules work in JavaScript.
- What a transpiler does.
- How to include literal JavaScript in JSX code.
- How to do conditional rendering in React.
- How to render children in JSX.

JSX IS NOT HTML

Take a look first at Listing 3-1. If you know some HTML, you can probably guess what the result of this function will be—a form containing two input fields and a button will be returned.

LISTING 3-1:
A React component

```
import React from "react";
function Login(){
const handleSubmit = (e)=>{
e.preventDefault();
console.log(`logging in
${e.target[0].value}`);
}
return (
<form id="login-form"
onSubmit={handleSubmit}>
<input type="email"
id="email"
placeholder="E-Mail
Address"/>
<input type="password"
id="password"/>
<button>Login</button>
</form>
);
}
export default Login;
```

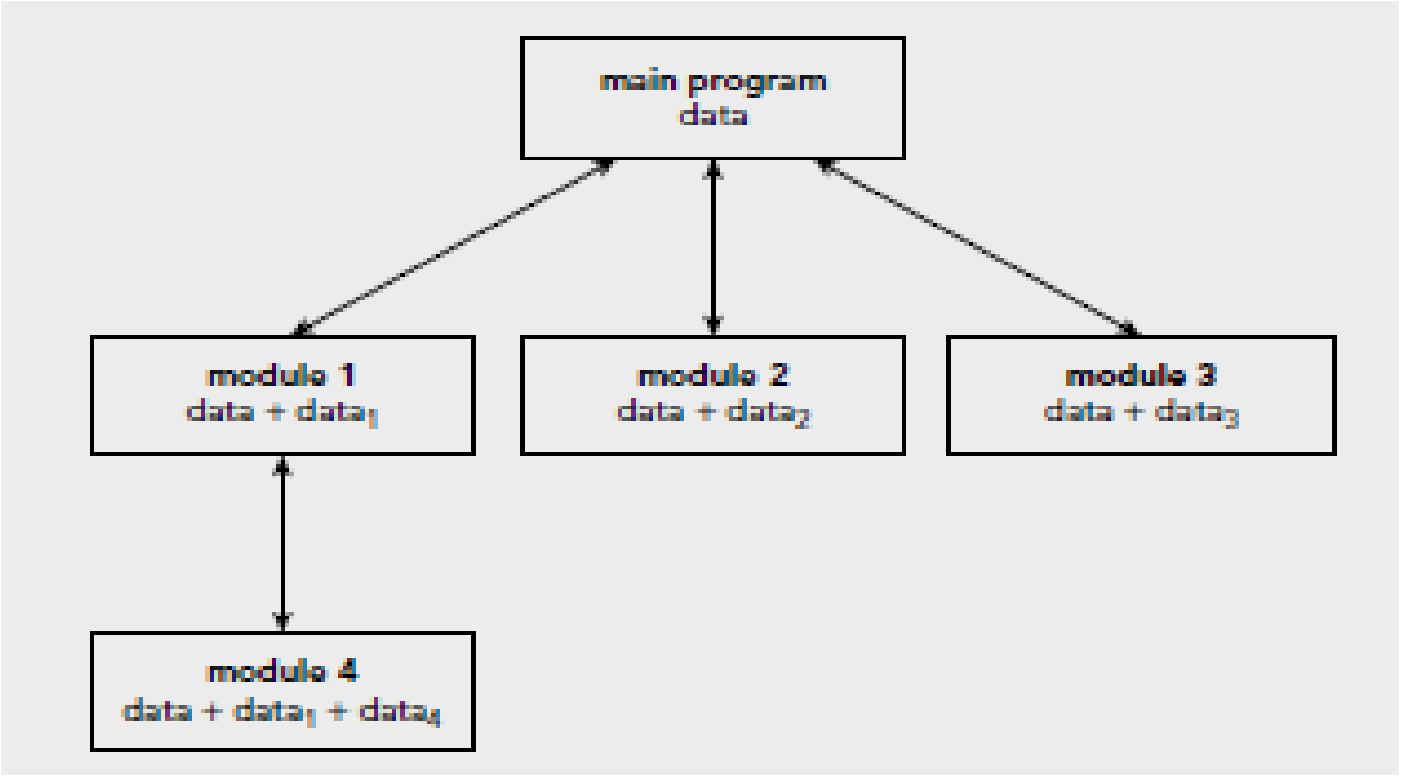
But, if you know some JavaScript, you might think the result of running this JavaScript function should be an error—HTML is not valid JavaScript, and so the value of the return statement will cause this function to fail.

However, this is a perfectly well-formed React component and that markup inside of the return statement actually isn't HTML. Instead, **it's written in JSX, which is an extension of JavaScript that's used as a visual aid to help you describe what a component should look like.**

JAVASCRIPT LESSON: MAKING SENSE OF MODULES WITH AMD, CJS, AND ESM

- Modularization is a fundamental concept in software development in which a program is organized into reusable units of code.
- Modularization makes software easier to build, debug, test, and manage and it also enables team development of software. Just as functions create units of functionality that can be reused in a JavaScript file, modules create JavaScript files that can be reused within a program.

computer program made up of modules might look something like this:



THE HISTORY OF JAVASCRIPT MODULES

JavaScript started its life as a scripting language for web browsers. In its early days, scripts written in JavaScript were small. Because it was seen as less than a "real" programming language, no thought was given to including a way to create modules in JavaScript.

Instead, programmers would just write their JavaScript in a single file and import it into HTML files using the script element, or write all of their JavaScript directly into the script element.

- As JavaScript became a more powerful language, and as the number of things that people were doing with JavaScript began to grow, so too did the complexity and size of JavaScript files.

The Rise of the JavaScript Module

Because JavaScript couldn't do modules natively, when the need for and benefits of breaking up large JavaScript programs into smaller pieces became apparent, JavaScript developers did what they **always do and created new libraries that could be used to add modularization into JavaScript**.

ES Modules

- Having more than one way to create and use modules made modules less reusable, however, and the ultimate dream of JavaScript programmers was always that JavaScript would someday have a built-in way to modularize code. This dream became a reality with the standardization of ECMAScript Modules (ESM).

- ESM features asynchronous module loading, like RequireJS, but has a simple syntax, like CommonJS. ***The statements that you use to create and use ES Modules are import and export.***

USING IMPORT AND EXPORT

React components are JavaScript modules, and so you'll see import and export statements everywhere in React. ***The most basic thing to know about import and export is that the export statement creates modules, and the import statement imports modules into other JavaScript code.***

Quiz 3

Q1. what is modularization

Q2. what are 2 Javascript libraries to make modularization.

Q3. what is main difference between export and import.

Q4. discuss ES modules

Q5. write down an html form to record first name, age, tel,comment.

Since import and export are built into JavaScript now, there's no need to include a separate library to make use of them.

export Creates Modules

Let's say that you have a function that calculates shipping charges for your ecommerce store. The basic skeleton of this function might look something like this:

```
Function calculateShippingCharge(weight,shippingMethod){
// do something here
return shippingCharge;
}
```

Turning this function into a module ***would make it more reusable, since you'd then be able to simply include it into any file that needs to calculate shipping charges***, and you could even make use of it in different programs as well.

The basic syntax for using export is to put the export keyword before the definition of the function, like this:

```
export function calculateShippingCharge(weight,shippingMethod){
// do something here
return shippingcharge;
}
```

Now, you can put this module into a file with other modules (maybe the file would be named ecommerce-utilities.js) and you can import individual functions, or every function, from this file into any other file in your program.

import Imports Modules

To import a function, variable, or object from a JavaScript module, you use the import statement. To use import, ***name at least one module, followed by the from keyword, followed by the path to the file that contains the module or modules you want to import.***

```
import { shippingMethods, calculateShippingCharges } from‘./modules/ecommerce-utilities.js';
```

Using Default Exports

Another way to use export is to create a *default export*. A default export can be used **to specify a default function provided by a module**:

```
function calculateShippingCharge(weight,shippingMethod){
// do something here
}
export default calculateShippingCharge;
```

You can only have one default export per file. When you have a default export, you can import the module specified with the default export by using the import statement without the curly braces, like this:

```
import calculateShippingCharge from
'./modules/calculateShippingCharge.js';
```

You can import individual items from a file by surrounding them with curly braces, like this: **import calculateShippingCharge from'./modules/calculateShippingCharge.js';**

React components are usually created using default exports, unless you're creating a library of components.

Note: you'll often see the path to a module specified without the .js at the end.
For example:

```
import calculateShippingCharge from'./modules/calculateShippingCharge';
```

- When you omit .js at the end of a filename in an import, the import will work exactly the same as if you had specifically written it.
- Also notice that the path to the module file starts with **'./'**. This is the UNIX way of saying to start with the current directory and to create a relative path from it.
- ES Modules require that the path to the module is a relative path, so it will always start with ./ (the current directory) or ../ (indicating the parent directory). Oftentimes, you may need to have more than one ../, if the module you want to load is higher up in the file hierarchy.

So, in the previous case, the modules folder is a subdirectory of the directory containing the file that's importing the module.

If you've installed Node.js packages using npm, such as the React library itself, you don't need to use ./ or to specify the path to the Node.js package when you import it. For example, *components that use the React library's functions have an import statement that imports React. This usually looks like this:*

```
import React from 'react';
```

Although you may also see individual objects from the React library imported separately, like this:

```
import React, {Component} from 'react';
```

Some Important ES2015 Module Rules

There are just a few more important rules for how to use import and export:

- Both import and export statements need to be at the top level of your JavaScript file—that is, not inside of a function or any other statement.
- Imports must be done before any other statements in a module.
- import and export can only be used inside modules (not inside of ordinary JavaScript files).

WHAT IS JSX?

JSX is an XML-based syntax extension to JavaScript. In plain English, it's a way to write JavaScript code using XML. Although it's not specific to React, and it's not even required in order to write React components, ***JSX is an integral part of how every React developer writes components because it makes writing components so much easier and has no negative impact in terms of performance or functionality***.

How JSX Works

React uses JSX elements to represent custom components (which are also known as ***user-defined components***). If you create a component named SearchInput, you can make use of that component in other components by using a JSX element named SearchInput, as shown in Listing 3-2.

LISTING 3-2: Using a user-defined React component in JSX

```
import {useState} from 'react';
import SearchInput from './SearchInput';
import SearchResults from './SearchResults';
function SearchBox() {
const [searchTerm,setSearchTerm] = useState('');
return (
<div id="search-box">
<SearchInput term={searchTerm}
onChange={setSearchTerm}/>
<SearchResults term={searchTerm}/>
</div>
)
}
export default SearchBox;
```

quiz4

Q1. list at least one ES modules rule
Q2. Discuss when we use export and import
Q3. write a syntax to import modules
Q4. create a link of **w.html** on a html file **m.html** in its subfolder x

In the same way, ***React has components built into it for each of the elements in HTML5***, and ***you can use any HTML5 element name when you write your React components and the result will be that React will output that HTML5 element.*** For example, say you want your React component to result in the rendering of the following piece of HTML markup:

```
<label class="inputLabel">Search: <input type="text“
id="searchInput"></label>
```

The JSX code for telling your React component to output that HTML would look like this:

```
<label className="inputLabel">Search: <input type="text" id="searchInput"/>
</label>
```

If you study both of the preceding snippets closely, you'll find a couple of differences. The difference between them, and the fact that JSX is not HTML, are of vital importance to understanding what JSX is really doing.

It's fully possible to create React components without using JSX by using the React.createElement method. Here's what the code to output the previous HTML markup looks like when you write it using React.createElement:

```
React.createElement("label", {className: "inputLabel"}, "Search:",
React.createElement("input", {type: "text", id: "searchInput"}));
```

If you examine this JavaScript code closely, you should be able to figure out basically how it works. The React.createElement method accepts **an element name**, **any attributes of the HTML element**, **the element's content** (**"Search:"** in this example) and **its child element or elements**. In this case, the label element has one child, input.

That's pretty much all there is to React.createElement. If you're interested in learning the exact syntax of React.createElement, you can read more about it here:

https://reactjs.org/docs/react-without-jsx.html

In reality, however, very few React developers ever have to think about React.createElement, because we use a tool called a ***transpiler*** as part of our development environment.

Transpiler . . . Huh?

Before you can run a React application that uses JSX and modules, it must first be compiled. During the compile (also known as "build") process, all of the modules are joined together and the JSX code is converted into pure JavaScript.

Compilation vs. Transpilation

Compilation of React applications is somewhat different from how programmers of truly "compiled" languages (like C++ or Java) understand compilation. In compiled languages, **the code that you write is converted into low-level code that can be understood by the computer's software interpreter**. This low-level code is called *bytecode*.

When React applications are compiled, on the other hand, they're converted from one version of JavaScript to another version of JavaScript. Because the React compilation process doesn't actually create bytecode, a more technically correct word for what happens is **transpilation.**

JSX Transform

One of the steps in the transpilation of React code is the JSX Transform. The JSX Transform is a process in which the transpiler takes JSX code (which isn't natively understood by web browsers) and converts it into plain JavaScript (which is natively understood by web browsers).

Introducing Babel

The tool we use for transpilation in JavaScript is called *Babel*. Babel is integrated into ***Create React App*** and is an automatic part of compiling a React app built with ***Create React App***.

It can be interesting sometimes to see how Babel converts JSX into JavaScript, and you can do this by either viewing the source code for a running React application or by pasting your JSX code into the web-based version of Babel at https://babeljs.io/repl, as shown in Figure 3-1.

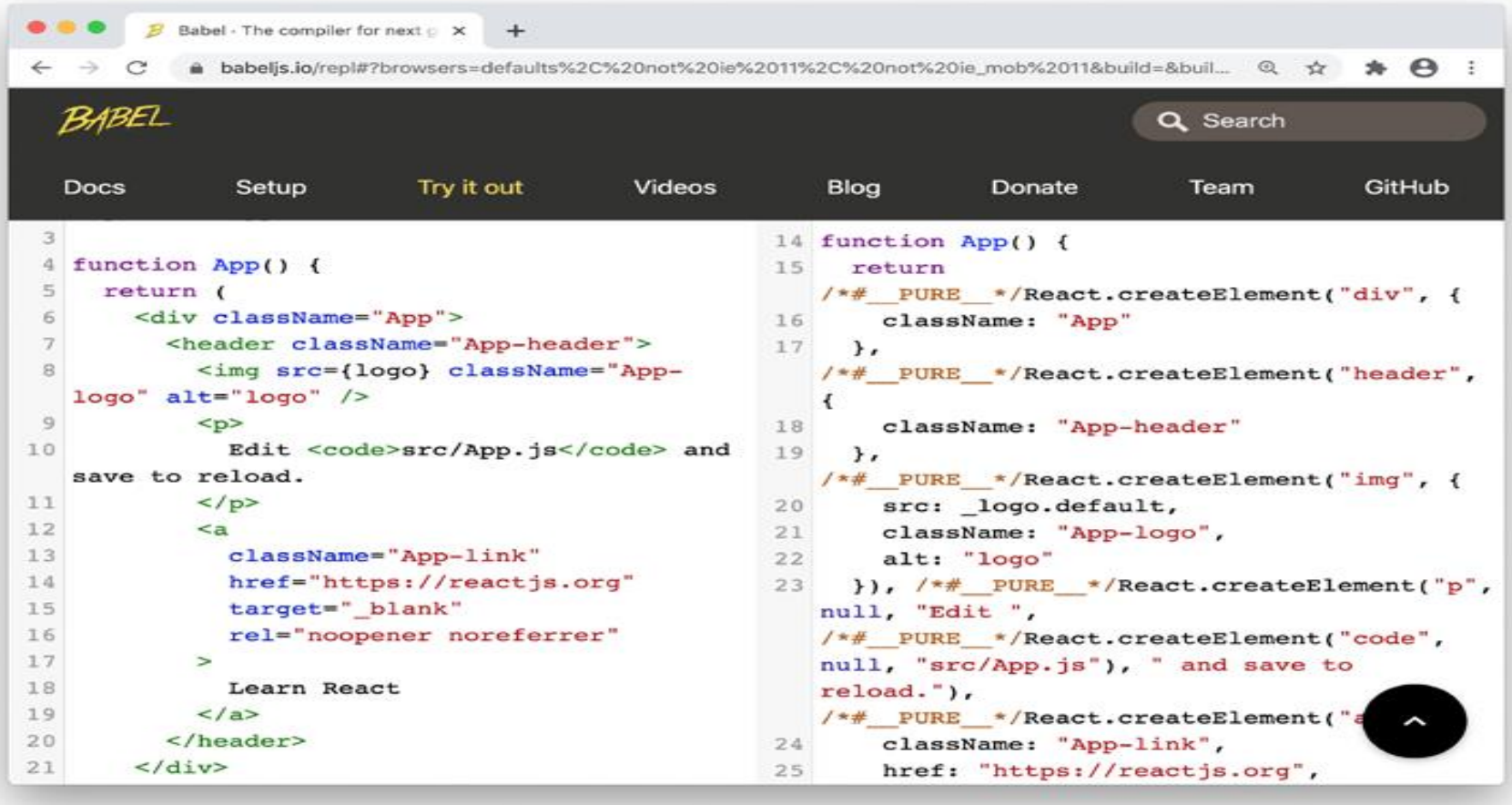


FIGURE 3-1: Trying out Babel on the web

Babel does much more than just convert JSX into JavaScript. ***It also takes JavaScript in your components that's written using new and experimental syntax that might not be supported by all of your target web browsers and converts it into JavaScript that can be understood and run in any web browser that you expect to access your React user interface.***

Eliminating Browser Incompatibilities

Using transpilation does away with the age-old problem of browser incompatibilities and ***having to wait until every browser supports a new JavaScript language feature before using it.*** Rather than developers having to write special code and multiple if/then branches to accommodate older browsers, ***Babel makes it possible for developers to just write JavaScript using the latest syntax and then transpile that new JavaScript into a common denominator that will run in any web browser that's likely to access the app.***

SYNTAX BASICS OF JSX

As I may have mentioned (and I'll mention again, because it's a really important point), JSX is not HTML. Because it's not HTML, you can't write JSX in the same loosey-goosey way that you may be used to writing HTML.

JSX Is JavaScript XML

The first thing to know about JSX is that it's XML. So, if you know a little bit about XML (or if you've used XHTML), the rules of writing JSX should sound familiar. Namely:

- All elements must be closed.
- Elements that cannot have child nodes (so-called“empty” elements) must be closed with a slash. The most commonly used empty elements in HTML are br, img, input, and link.
- Attributes that are strings must have quotes around them.
- HTML elements in JSX must be written in all lowercase letters.

Beware of Reserved Words

Because JSX compiles to JavaScript, there is the potential that an element name or attribute name that you use in your JSX code can cause errors in your compiled program. To guard against this, certain HTML attribute names that are also reserved words used in JavaScript have to be renamed,as follows:

- The **class** attribute becomes **className**.
- The **for** attribute becomes **htmlFor**.

JSX Uses camelCase

Attribute names in HTML that contain more than one word are camel-cased in JSX. For example:

- The onclick attribute becomes onClick.
- The tabindex attribute becomes tabIndex.

Preface Custom Attributes in DOM Elements with data-

Prior to version 16 of React, if you needed to add an attribute to a DOM element that doesn't exist in the HTML or SVG specification for the element, you had to preface it with data-,or else React would ignore it. Listing 3-3 shows a JSX HTML equivalent element with a custom attribute.

```
<div data-size="XL"
data-color=“ black"
data-description="awesome">
My Favorite T-Shirt</div>
```

Starting with React 16, however, you can use any custom attribute name with built-in DOM elements.

Custom attributes in DOM elements ***can be useful for including arbitrary data with your markup that doesn't have any special meaning or affect the presentation of the HTML in the browser.***

Although it is possible to use custom attributes for DOM elements, this is not generally considered **a good practice**.

User-defined elements, on the other hand, can have custom attributes with any name, as shown in Listing 3-4.

LISTING 3-4: User-defined elements can have any attributes

```
import MyFancyWidget from './MyFancyWidget';
function MyFancyComponent(props){
return(
<MyFancyWidget
widgetSize="huge"
numberOfColumns="3"
title="Welcome to My Widget" />
)
}
export default MyFancyComponent;
```

Using custom attributes with user-defined elements is the primary way that React passes data between, as you'll see in Chapter 4.

JSX Boolean Attributes

In HTML and in JSX, certain ***attributes don't require values***, *because their presence is interpreted as setting their value to a Boolean true*. For example, in HTML, the **disabled** attribute of input elements causes an input to not be changeable by the user:

```
<input type="text" name="username" disabled>
```

In JSX, ***the value of an attribute can be omitted when it is explicitly true***. So, to set the disabled attribute of a JSX input element to true, you can do either of the following:

```
<input type="text" name="username" disabled = {true}/>
<input type="text" name="username" disabled/>
```

Use Curly Braces to Include Literal JavaScript

When you need to include a variable or a piece of JavaScript in your JSX that shouldn't be interpreted by the transpiler, use curly braces around it. Listing 3-5 shows a component whose return statement includes literal JavaScript in JSX attributes.

LISTING 3-5: Using literal JavaScript inside of JSX

```
function SearchInput(props) {
return (
<div id="search-box">
<input type="text"
name="search"
value={props.term}
onChange={(e)=>{props.onChange(e.target.value)}}/>
</div>
)
}
export default SearchInput;
```

e is the short var reference for event object which will be passed to event handlers. The event object essentially has lot of interesting methods and properties that can be used in the event handlers.

Remember to Use Double Curly Braces with Objects

One common mistake is to forget that if you're including a JavaScript object literal inside of JSX, the JSX code will have double curly braces, as shown in Listing 3-6.

LISTING 3-6:Object literals in JSX result in double curly braces

```
function Header(props){
return (
<h1 style={{fontSize:"24px",color:"blue"}}>
Welcome to My Website
</h1>
)
}
export default Header;
```

Put Comments in Curly Braces

Because JSX is actually a way of writing JavaScript, HTML comments don't work in JSX. Instead, you can use JavaScript block comment syntax (/* and */). However, because you don't want to transpile your comments, they must be enclosed in curly braces, as shown in Listing 3-7.

LISTING 3-7:Enclose comments in curly braces

```
function Header(props){
return (
<h1 style={{fontSize:"24px",color:"blue"}}>
{/* Todo: Make this header dynamic */}
Welcome to My Website
</h1>
)
}
export default Header;
```

When to Use JavaScript in JSX

The concept of separation of concerns in programming says that layout code should be separated from logic. What this means in practice is that code that does calculations, retrieves data, combines data, and controls the flow of an application should be written as functions outside of the return statement in a component, rather than inside of curly braces in JSX.

Limited amounts of logic are necessary and perfectly normal inside of the return statement, however.

There's no hard-and-fast rule for how much is too much, but, generally, any **JavaScript that you write in your JSX should only have to do with presentation, and it should be single JavaScript expressions, rather than functions or complex logic.**

An example of purely presentational JavaScript would be the case of conditional rendering.

Conditionals in JSX

Oftentimes, a component needs to output different subcomponents, or hide certain components, **based on the results of expressions** or ***the values of variables***. We call this ***conditional rendering***.

There are three ways to write conditional statements in JavaScript, and you may use any of these to do conditional rendering.

Conditional Rendering with if/else and Element Variables

JSX elements can be assigned to variables, and these variables can be substituted for the elements inside a component's return statement, as shown in Listing 3-8.

LISTING 3-8: Using element variables

```
import Header from './Header';
function Welcome(){
let header = <Header/>;
return(
<div>
{header}
</div>
);
}
export default Welcome;
```

By using a conditional statement, you can assign a different element to a variable and thus change what gets rendered, as shown in Listing 3-9.

LISTING 3-9: Conditional rendering with element variables

```
import Header from './Header';
import Login from './Login';
function Welcome({loggedIn}) {
let header;
if (loggedIn) {
header = <Header/>;
} else {
header = <Login/>;
}
return (
<div>
{header}
</div>
);
}
export default Welcome;
```

Conditional Rendering with the && Operator

Rather than having your conditional logic outside of the return statement, you can write it inline by using the logical AND operator, &&. The && operator evaluates the expressions on its left and right. ***If both expressions evaluate to a Boolean true, the && will return the one on the right. If either side of the && operator is false, then a value of false will be returned.***

By applying this fact, you can conditionally return an expression from the right side of && if the left side of && is true.

This can be a little confusing at first. Take a look at Listing 3-10. This code will render the Header component if loggedIn evaluates to true.

LISTING 3-10: Conditional rendering with &&

```
import Header from './Header';
function Welcome({loggedIn}){
return (
<div>
{loggedIn&&<Header />}
Note: if you don't see the header
messsage,
you're not logged in.
</div>
)
}
export default Welcome;
```

Conditional Rendering with the Conditional Operator

The conditional operator is a way to combine the simplicity and conciseness of inline conditional rendering ***with the ability to have an else case that element variables combined with if and else gives us.***

Listing 3-11 shows an example of using the conditional operator.

```
import Header from './Header';
import Login from './Login';
function Welcome({loggedIn}){
return(
<div>
{loggedIn ? <Header /> : <Login />}
</div>
)
}
export default Welcome;
```

In this example, **the expression to the left of the question mark is evaluated. If it's true, the WelcomeMessage component** is returned. **If it's false, the Login component is returned.**

Expressions in JSX

You can use any JavaScript expression inside of your JSX or inside of React element attribute values by surrounding it with curly braces. JSX elements themselves are JavaScript expressions as well, because they get converted into function calls during compilation.

To understand what JavaScript you can and can't include in JSX, let's take a brief look at what a JavaScript expression is.

- An expression is any valid unit of code that resolves to a value. Here are some examples of valid JavaScript expressions:

- Arithmetic: 1+1
- String: "Hello, " + "World!"
- Logical: this !== that

- Basic keywords and general expressions: This includes certain keywords (such as this, null,true, and false) as well as variable references and function calls.

Examples of structures in JavaScript that do not return a value (and are thus not expressions) include for loops and if statements, as well as function declarations (using the function keyword). You can still use these in your React components, of course, but you'll need to use them outside of the return statement, as we did in Listing 3-9.

Functions can be included in JSX, provided that they're invoked immediately and that they return a value that can be parsed by JSX, or that they're passed as values for an attribute. The component in Listing 3-12 has a return statement that includes a function as an event handler.

LISTING 3-12:Using an arrow function as an event handler

```
import {useState} from 'react';
function CountUp(){
const [count,setCount] = useState(0);
return (
<div>
<button onClick={()=>setCount(count+1)}>Add One</button>
{count}
</div>
);
}
export default CountUp;
```

Listing 3-13 shows an example of using a function that's immediately invoked and that's valid in JSX.

LISTING 3-13: Immediately invoking a function in JSX

```
function ImmediateInvoke(){
return(
<div>
{(()=><h1>The Header</h1>)()}
</div>
);
}
export default ImmediateInvoke;
```

Using Children in JSX

The return statement in a React component can only return one thing. This one thing can **be a string**, **a number**, ***an array***, ***a Boolean***, or ***a single JSX element***. Keep in mind, however, ***that a single JSX element can have as many children as you like. As long as you start and end your return statement with a matching opening tag and closing tag, everything in between (provided that it's valid JSX or a JavaScript expression) is fine.***

Here's an example of an invalid JSX return value:

```
return(
<MyComponent />
<MyOtherComponent />
);
```

One way to make this a valid JSX return value is to wrap two elements with another element,like this:

```
return(
<div>
<MyComponent />
<MyOtherComponent />
</div>
);
```

With the div element wrapping the two user-defined elements, we now have a single element being returned.

React Fragments

Although it's quite common to see multiple elements wrapped with a div element or another element for the purpose of returning a single JSX element, **adding div elements just for the sake of eliminating errors in your code, rather than to add necessary meaning or structure to your code, creates code bloat and decreases the accessibility of your code.**

To prevent the introduction of unnecessary elements, ***you can use the built-in React. Fragment component. React.Fragment wraps your JSX into a single JSX element, but doesn't return any HTML.***

You can use the React.Fragment component in one of three ways:

1. By using dot notation: ***<React.Fragment></React.Fragment>***
2. By importing Fragment from the react library using curly braces
3. By using its short syntax, which is just a nameless element: < > < / >

Listing 3-14 shows how to use React.Fragment in a component.

LISTING 3-14: Using React.Fragment

```
import {Fragment} from 'react';
function MyComponent(){
return(
<Fragment>
<h1>The heading</h1>
<h2>The subheading</h2>
</Fragment>
);} export default MyComponent;
```

Listing 3-15 shows how to use the short syntax for React.Fragment.

```
function MyComponent(){
return(
<>
<h1>The heading</h1>
<h2>The subheading</h2>
</>
); }
```

LISTING 3-15: Using React.Fragment's short syntax

```
function MyComponent(){
return(
<>
<h1>The heading</h1>
<h2>The subheading</h2>
</>
);
}
export default MyComponent;
```

NOTE Notice that when you use React.Fragment's *short syntax, you don't need to import* Fragment *from React.*

The result of running either Listing 3-14 or Listing 3-15 is that just the h1 and h2 HTML elements will be returned.

SUMMARY

JSX is an important tool that is used in the development of nearly every React component. In this chapter, you learned:

- Why we use JSX, the XML language that React uses to make it easier to visualize and write the output of components.
- That JSX is not HTML, but that React uses JSX to generate HTML.
- The history of JavaScript modules, which make distributed development and reusable components possible, and how to use import and export to create and use modules.
- What transpiling is.
- How to write JSX code.

- What conditional rendering is and how to do it in JSX.
- How to use JavaScript expressions inside JSX.
- How to use comments in JSX.
- How to use React.Fragment to group elements together without returning extra HTML elements.

All About Components

Up until now, we've mostly been talking about the tools that make React development possible, including your development environment, Node.js, ReactDOM, JavaScript modules, and JSX.

Now it's time to dig deeply into the heart of what makes React tick: **the component**.

In this chapter, you'll learn:

- The relationship between components and elements.
- How to use React's HTML elements.
- How to pass data between components with props.
- How to write class components.
- How to write function components.
- How to bind functions in JavaScript.
- How to manage React state.

WHAT IS A COMPONENT?

Components are the building blocks of React applications. **A React component** is a function or a JavaScript class that optionally accepts data and returns a React element that describes some piece of the user interface. ***A React user interface is made up of a hierarchy of components that build up to a single component (called the root component) that is rendered in the web browser.***

Figure 4-1 shows an example of a React component tree.
It's possible to create a React application with only a single component, but for all but the smallest apps, breaking your app up into multiple components makes development and management of the code easier.
In this very simple example, WelcomeMessage is a React component that was created using a function and exported as a JavaScript module. Once it's exported, WelcomeMessage can be imported into any other React component where you need to make use of its functionality, as shown in Listing 4-2.

LISTING 4-2: Components can be imported into other components

```
import WelcomeMessage from './WelcomeMessage';
function WelcomeTitle(){
return <h1><WelcomeMessage /></h1>;
}
export default WelcomeTitle;
```

It's not a requirement that each component have its own module, ***but that's the most common way components are defined***. In components created using a default export, the file containing the module usually takes the name of the component defined in the file.

Once you import a component into another component, **this is where React elements come in.**

Elements Invoke Components

Once you've imported a component into another component, *the imported component's functionality can be included in your new component's JSX using an element.* ***You can include as many components inside another component as you need to, and there's no limit to how many levels of components a tree of components can have.***

Once you import a component, ***you can use the element it defines as many times as you need to and each usage will create a new instance of the component with its own data and memory.***

In general, the point of using components ***is to provide a higher level of abstraction that reduces the complexity of an application and enables reuse***.

Listing 4-3 shows an example of a top-level React component that uses the functionality of other components to display a shopping cart user interface.

LISTING 4-3: Using components to reduce complexity

```
import React from 'react';
import CartItems from './CartItems';
import DisplayTotal from './DisplayTotal';
import CheckoutButton from './CheckoutButton';
import styles from './Cart.css.js';
function Cart(props){
return(
<div style={styles.cart}>
<h2>Cart</h2>
<CartItems items = {props.inCart} />
<DisplayTotal items = {props.inCart} />
<CheckoutButton />
</div>
);
}
export default Cart;
```

Notice that the component in Listing 4-3 uses a combination of ordinary JavaScript and imported modules to return a combination of custom elements and HTML elements. It's fairly trivial to figure out the gist of what will be rendered by this component just by looking at the return statement.

The entire component could have been written with everything in a single file, as shown (partially) in Listing 4-4, ***but the result would be a file that would be much larger, more difficult to work with, and more difficult to maintain***.

Don't worry if much of the code in Listing 4-4 looks strange or unfamiliar to you. Remember that React is just JavaScript, and this example uses several relatively new JavaScript tools and functions that I'll explain later in this chapter.

LISTING 4-4: Putting everything in one component

```
import React,{useState} from 'react';
import styles from './Cart.css.js';
function Cart(props){
const [inCart,setInCart] = useState(props.inCart);
const removeFromCart = (item)=>{
const index = inCart.indexOf(item);
const newCart = [...inCart.slice(0, index), ...inCart.slice(index + 1)];
setInCart(newCart);
};
const calculatedTotal = inCart.reduce((accumulator, item) => accumulator + (item.price || 0), 0);
let ItemList = inCart.map((item)=>{
return (<div key={item.id}>{item.title} –{item.price}
<buttononClick={()=>{removeFromCart(item)}}>remove</button></div>)
});
return(
<div style={styles.cart}>
<h2>Cart</h2>
{ItemList}
<p>total: ${calculatedTotal}</p>
<button>Checkout</button>
</div>
);
} export default Cart;
```

BUILT-IN COMPONENTS

React has built-in components for the most commonly used HTML elements and their attributes. There are also built-in components for Scalable Vector Graphics (SVG) elements and attributes. ***These built-in components produce output in the DOM and are the base for your custom components***.

HTML Element Components

React's built-in HTML element components have the same names as elements from HTML5. Using them in your React app causes the equivalent HTML element to be rendered.

Many React developers (and web application developers in general) tend to use the div element for every type of container in their user interfaces. While this is convenient, it's not always recommended.

HTML is a rich and descriptive language when used correctly, and using meaningful (aka semantic) HTML elements to mark up your content makes it more accessible for search engines and people as well.

Table 4-1 shows all the HTML elements that React supports, along with a brief explanation of each element. If an element that you want to use in your user interface isn't on this list, try using it to see if it's been added since this list was compiled. If it isn't, you can submit a request to Facebook that the element be added to React by filing an issue in the React github.com repository at https://github.com/facebook/react/issues/new.

TABLE 4-1: HTML Elements Supported by React

HTML ELEMENT	DESCRIPTION
a	Creates a hyperlink.
abbr	Represents an abbreviation or acronym.
address	Indicates that the containing HTML includes contact information.
area	Defines a clickable area in an imagemap.
article	Represents a self-contained composition (such as a story or an article) in a page.
aside	Represents content that is indirectly related to the main content.

HTML ELEMENT	DESCRIPTION
audio	Embeds sound content.
b	Used to draw the reader's attention to the contents. Previously, this was the "bold" element, but it's now called the "Bring to Attention" element to separate its purpose from how it's styled.
base	Specifies the base URL for all relative URLs in the document.
bdi	Bidirectional Isolate. Isolates text that may flow in a different direction from text around it.
bdo	Bidirectional Text Override. Changes the direction of text.
big	Renders text at a font size one level larger (obsolete).
blockquote	Indicates an extended quotation.
body	Represents the content of an HTML document.
br	Produces a line break.
button	Represents a clickable button.
canvas	Creates an area for drawing with the canvas API or WebGL.

caption	Specifies a caption for a table.
cite	Describes a reference to a cited work.
code	Indicates that its content should be styled as computer code.
col	Defines a column within a table.
colgroup	Defines a group of columns in a table.
data	Links content to a machine-readable translation.
datalist	Contains `option` elements indicating the permissible options available for a form control.
dd	Provides the definition for a preceding term (specified using `dt`).
del	Represents text that has been deleted from a document.
details	Creates a widget in which information is visible when the widget is toggled to its "open" state.
dfn	Indicates the term being defined within a sentence.
dialog	Represents a dialog box, subwindow, alert box, or other such interactive element.
div	A generic container with no effect on content or layout.

HTML ELEMENT	DESCRIPTION
dl	Represents a description list.
dt	Specifies a term in a definition list. Used inside `dl`.
em	Marks text that has emphasis.
embed	Embeds external content in the document.
fieldset	Groups controls and labels within a form.
figcaption	Describes the contents of a parent `figure` element.
figure	Represents self-contained content, optionally with a caption.
footer	Represents a footer for its nearest sectioning content.
form	Represents a document section containing interactive controls.
h1	First-level section heading.
h2	Second-level section heading.
h3	Third-level section heading.
h4	Fourth-level section heading.
h5	Fifth-level section heading.
h6	Sixth-level section heading.
head	Contains machine-readable information about the document.

header	Represents introductory content.
hr	Represents a thematic break between sections.
html	Represents the root of an HTML document.
i	Represents idiomatic text that is set off from the normal text.
iframe	Represents a nested browser context.
img	Embeds an image into the document.
input	Creates interactive controls for web-based forms.
ins	Represents a range of text that has been added to the document.
kbd	Represents a span of text denoting textual user input.
keygen	Facilitates generation of key material and submission of the public key in an HTML form.
label	Represents a caption for an item in a user interface.
legend	Represents a caption for an element in a `fieldset`.

HTML ELEMENT	DESCRIPTION
li	Represents an item in a list.
link	Specifies a relationship between the document and an external resource. Commonly used to link stylesheets.
main	Represents the dominant content of the `body` of a document.
map	Used with `area` elements to define an imagemap.
mark	Represents marked, or highlighted, text.
menu	Represents a group of commands.
menuitem	Represents a command in a `menu`.
meta	Represents metadata that can't be represented with other metadata elements (such as `title`, `link`, `script`, or `style`).
meter	Represents a fractional value or a scalar value within a known range.
nav	Represents a section containing navigation links.
noscript	Represents a section to be inserted if a script type is unsupported or if scripting is disabled in the browser.
object	Represents an external resource.
ol	Represents an ordered list.
optgroup	Creates a grouping of options within a `select` element.
option	Defines an item in a `select` or `optgroup`.

output	Creates a container for the results of a calculation or for user input.
p	Represents a paragraph.
param	Defines parameters for an `object`.
picture	Contains `source` elements and an `img` element to provide alternative versions of an image.
pre	Represents preformatted text which should be presented exactly as written.
progress	Displays an indicator showing progress towards the completion of a task, such as a progress bar.
q	Indicates that its content is a quotation.
rp	Used to provide fallback content for browsers that don't support ruby annotations using the `ruby` element.
rt	Specifies the ruby text component of a ruby annotation.
ruby	Represents annotations for showing the pronunciation of East Asian characters.

HTML ELEMENT	DESCRIPTION
s	Represents a strikethrough.
samp	Encloses text that represents sample output from a computer program.
script	Embeds executable code or data.
section	Represents a standalone section in a document.
select	Represents a control that shows a menu of options.
small	Represents small print, such as copyright or legal text.
source	Specifies multiple media resources for `picture` and `audio` elements.
span	A generic inline container.
strong	Indicates that its contents have strong importance.
style	Contains style information for a document.
sub	Specifies inline text that should be displayed as subscript.
summary	Specifies a summary, legend, or caption for `details` content.
sup	Specifies inline text that should be displayed as superscript.
table	Represents tabular data.
tbody	Encapsulates table rows in a `table`.
td	Defines a cell in a `table`.
textarea	Represents a multi-line text editing control.

tfoot	Defines a set of rows summarizing the columns in a table.
th	Defines a cell as a header of a group of table cells.
thead	Defines a set of rows defining the head of the columns in a table.
time	Represents a period of time.
title	Defines the title that is shown in the browser's title bar and browser tab.
tr	Defines a row of cells in a table.
track	Contains timed text tracks (such as subtitles) for audio and video content.
u	Originally the underline element, specifies that text should be rendered in a way that indicates that it has non-textual annotation (whatever that means).
ul	Represents an unordered list (usually rendered as a bulleted list).
var	Represents the name of a variable in mathematic or programming context.
video	Embeds a media player that supports video playback.
wbr	Represents a word break opportunity, where the browser may optionally break a line.

Attributes vs. Props

In markup languages (such as XML and HTML), attributes define properties or characteristics of the element, and are specified using the name=value format. Because JSX is an XML markup language, ***JSX elements can have attributes, and there's no limit to the number of attributes that a single JSX element can have.***

Passing Props

Attributes that you write in JSX elements ***are passed to the component represented by the element as properties, or props for short***. **You can access props inside the component using the component's props object.**

To illustrate how props are used for passing data between components, I'll use the example of a component called Farms, which includes multiple instances of the Farm component, as shown in Listing 4-5. **Props that you pass into the Farm component are what make it possible for the generic Farm component to represent any farm.**

LISTING 4-5: Passing props

```
import Farm from './Farm';
export default function Farms(){
return(
<>
<Farm
farmer="Old McDonald"
animals={['pigs','cows','chickens']} />
<Farm
farmer="Mr. Jones"
animals={['pigs','horses','donkey','goat']} />
</>
)
}
```

Accessing Props

Once values have been passed as props, you can access that data inside the component. Listing 4-6 shows the Farm component and how it makes use of the data passed into it.

LISTING 4-6: Using props inside a component

```
export default function Farm(props){
return (
<div>
<p>{props.farmer} had a farm.</p>
<p>On his farm, he had some {props.animals[0]}.</p>
<p>On his farm, he had some {props.animals[1]}.</p>
<p>On his farm, he had some {props.animals[2]}.</p>
</div>
)
}
```

As in all JavaScript functions, if data is passed into a function component, ***you can give that data a name inside the function arguments.*** This name, technically, could be anything. However, since React's class-based components accept passed data using this.props, it's standard practice and smart to use the name props in function components as well.

Notice that ***when you use props inside the return statement, you have to enclose them in curly braces.*** You can use props elsewhere inside a component as well, as shown in the slightly improved version of the Farm component shown in Listing 4-7.

LISTING 4-7: An improved version of the Farm component

```
export default function Farm(props){
let onHisFarm = [];
if(props.animals){
onHisFarm = props.animals.map((animal,index)=>
<p key={index}>On his farm he had some {animal}.</p>);
}
return (
<>
<p>{props.farmer} had a farm.</p>
{onHisFarm}
</>
)
}
```

JAVASCRIPT LESSON: USING ARRAY.MAP()

JavaScript's Array.map function creates a new array using the result of applying a function to every element in an existing array. The map function is commonly used in React to build lists of React elements or strings from arrays.

The syntax of Array.map is as follows:

```
array.map(function(currentValue, index, arr),thisValue)
```

Take a closer look at the details:

- The array is any JavaScript array. The function passed into the map function will run once for every element in the array.
- The currentValue is the value passed into the function and will change with every iteration through the array.
- The index parameter is a number representing the current value's position in the array.
- The **arr** parameter is the array object that the currentValue belongs to.

- The **thisValue** parameter is a value to be used as the "**this**" value inside the function.

The only required parameter is currentValue. It is also what you will most commonly see in real-world React applications. Here's how you can use Array.map() to make a series of list items from an array:

```
const bulletedList = listItems.map(function(currentItem){
return <li>{currentItem}</li>
}
```

For performance reasons, React requires each item in a list of JSX elements (such as one built from an array) to have a unique key attribute. One way to give each element a unique key is to use the index parameter, like this:

```
const bulletedList = listItems.map(function(currentItem,index){return <li
key={index}>{currentItem}</li>
}
```

Standard HTML Attributes

As you saw in Chapter 3, React's HTML components support most of the standard HTML attributes, but with a couple of important differences, which I'll reiterate and expand upon here.

Attributes Use camelCase

Whereas HTML5 attributes use all lowercase letters, and a few of them use dashes between multiple words (such as the accept-charset attribute), **all attributes in React's HTML components use capital letters for words in the attribute after the first one.** This type of capitalization is commonly called *camelCase*.

For example, the HTML tabindex attribute is represented by tabIndex in React and onclick is represented by onClick.

Two Attributes Are Renamed

In a couple of cases, React attributes for built-in elements have different names than HTML attributes. The reason for this is to avoid potential clashes with reserved words in JavaScript. The attributes that are different in React are:

- class in HTML is className in React.
- for in HTML is htmlFor in React.

React Adds Several Attributes

Several attributes that are available for React's built-in HTML components don't exist in HTML.

Chances are good that you'll never need to use any of these special attributes, but I'm including them here for completeness. These are:

- dangerouslySetInnerHTML, which allows you to set the innerHTML property of an element directly from React. As you can tell by the name of the attribute, this is not a recommended practice.
- suppressContentEditableWarning, which suppresses a warning that React will give you if you use the contentEditable attribute on an element that has children.
- suppressHydrationWarning. No, it's not a way to tell React to stop nagging you to drink more water. This attribute will suppress a warning that React gives you when content generated by server-side React and client-side React produce different content.

Some React Attributes Behave Differently

Several attributes behave differently in React than they do in standard HTML:

- **checked** and **defaultChecked**. The checked attribute is *used to dynamically set and unset the checked status of a radio button or checkbox*. The defaultChecked attribute sets whether a radio button or checkbox is checked when the component is first mounted in the browser.

- **selected**. In HTML, when you want to make an option in a dropdown be the currently selected option, you use the selected attribute. In React, **you set the value attribute of the containing select element instead.**
- **style**. React's style attribute accepts a JavaScript object containing style properties and values, rather than CSS, which is how the style attribute in HTML works.

React Supports Many HTML Attributes

The following list contains the standard HTML attributes supported by React's built-in HTML components:

accept acceptCharset accessKey action allowFullScreen allowTransparency alt async autoComplete autoFocus autoPlay capture cellPadding cellSpacing charset challenge checked classID className cols colSpan content contentEditable contextMenu controls coords crossOrigin data dateTime defer dir disabled download draggable encType form formAction formEncType formMethod formNoValidate formTarget frameborder

headers height hidden high href hrefLang htmlFor httpEquiv icon id inputMode keyParams keyType label lang list loop low manifest marginHeight marginWidth max maxLength media mediaGroup method min minLength multiple muted name noValidate open optimum pattern placeholder poster preload radioGroup readOnly rel required role rows rowSpan sandbox scope scoped scrolling seamless selected shape size sizes span spellCheck src srcDoc srcSet start step style summary tabIndex target title type useMap value width wmode wrap

Non-Standard Attributes

In addition to the standard HTML attributes, React also supports **several non-standard attributes** ***that have specific purposes in some browsers and meta-data languages, including:***

autoCapitalize and autoCorrect, which are supported by Mobile Safari.

- property is used for Open Graph meta tags.
- itemProp, itemScope, itemType, itemRef, and itemID for HTML5 microdata.
- unselectable for Internet Explorer.

- **results and autoSave** are attributes supported by browsers built using the WebKit or Blink browser engines (including Chrome, Safari, Opera, and Edge).

Custom Attributes

As of version 16, React will pass any custom attributes that you use with HTML components through to the generated HTML, provided that the custom attributes are written using only lowercase letters.

USER-DEFINED COMPONENTS

Have you ever thought that it would be awesome if you weren't just limited to the standard set of HTML elements? What if you could, for example, make an element called PrintPageButton that you could use anywhere that you need to display a functional print button in your app? Or what if you had an element called Tax that would calculate and display the taxes in your online store's shopping cart?

Essentially, this is what React components enable through **custom components. CUSTOM COMPONENTS**, also known as **USER-DEFINED COMPONENTS**, *are the components that you make by putting together built-in components and other custom components.*

The possibilities for custom components are infinite. Even better, if you design your components to be reusable, you can reuse components not only inside of a single React application, but across any number of React applications. *There are even hundreds of open source libraries of custom components created by other developers that you can repurpose inside your own apps.*

Writing useful and reusable React components can sometimes require considerable work up front, but the benefits of writing them the right way are that you can reduce work for yourself overall and make apps that are sturdier and more dependable.

In the rest of this chapter, you'll learn about writing custom components and putting them together to build **robust user interfaces.**

TYPES OF COMPONENTS