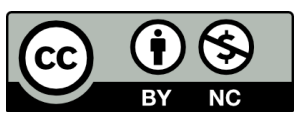
Building User Interfaces for Modern Web Applications

React Programming

Cheer-Sun Yang, Ph.D.

A Member of The Pennsylvania Alliance for Design of Open Textbooks



The contents of this eTextbook were developed under a grant from the Fund for the Improvement of Postsecondary Education, (FIPSE), U.S. Department of Education. However, those contents do not necessarily represent the policy of the Department of Education, and you should not assume endorsement by the Federal Government.

Table of Contents

Preface

Ever since the release of *React* by Facebook and Instagram in 2013, building user interfaces (UIs) with front-end technologies for modern web applications has attracted web developers' attention due to three major benefits.

First, React User Interface (UI) components are reusable. Like building a LEGO® toy product, such as a spaceship or a playground using LEGO® bricks, building a UI with React becomes modularized with reusable React components. Hence, just as LEGO® bricks are the fundamental building blocks of an iconic product, React reusable components are the basic building blocks for the UIs of modern web applications.

Second, React components are not only reusable but also "relocatable". This concept is known as the "cross-platform" development. With the assistance of React Native, a web application UI can be built once for a desktop machine and reused on a mobile device. When the source code becomes a "single source of truth," maintainability is enhanced.

Lastly, React can be used to realize the concept of isomorphism that allows React to change a web application from a "single-page application" (SPA) execution mode to a "server-side rendering" (SSR) execution mode. This flexibility makes React a preferred choice for advanced web applications.

Since the release of React, many print books have appeared in the market. The reasons to "re-invent another wheel" is threefold:

1. The need for timely updates - React is a "moving target"; it continues to evolve. Its continuous evolution makes writing and publishing a paper-based book about React a poor choice; many printed works become outdated after a year or two. An e-book solves the dilemma of timely updating. An e-book may need to be updated once a year or so depending on how drastically the new releases change, but such updates can be done far more easily than publishing a new print edition.

2. The soundness of examples - Although there have been many books for beginners, many do not emphasize software engineering principles. In this book, we present examples and exercises with software engineering principles, such as design methodologies (modularity, object-oriented design and development), design patterns (singleton and prototype, chaining of responsibilities), reusability, maintainability, and scalability while introducing the fundamental concepts about React.

3. The coverage of prerequisite contents and exercises - Many textbooks are written for an audience with a strong knowledge of HTML/CSS. Many others are written for audiences with a basic knowledge of React or web client and web server interactions. Some others do not provide enough examples or exercises to reiterate how the basic concepts can be used. Based on the

author’s experiences, these flaws make these books insufficient for teaching the User Interface course.

In this e-book, Part I covers prerequisite information about HTML, CSS, JavaScript, and Bootstrap for styling. Part II covers the main concepts of React including Components, Properties, State, Events, and Component Lifecycles. Part III discusses the rendering of UI components regarding conditional rendering, lists, and forms as examples. Part IV discusses design issues such as URL routing, lifting (up) state, and state management methods such as Redux Data Store and Redux Tool Kit.

In each chapter, *Definitions* and *Concepts* are introduced first, followed by *Programming Examples*. At the end of each chapter, *Review Questions*, and *Programming Exercises* are provided, if needed. As part of the expected learning outcomes, students can learn the basics about React seamlessly with minimum knowledge about modern web programming.

Cheer-Sun D. Yang, Ph.D.

Professor

Computer Science Department

West Chester University

Part I Background Skills

In the first part of this book, we cover some requisite skills prior to discussing React.

- Chapter 1 Introduction and setup
- Chapter 2 HTML
- Chapter 3 CSS and Bootstrap
- Chapter 4 JavaScript and JSX

Chapter 1 Introduction and Setup

As you begin to read this book, you probably have decided to learn the development of user interfaces (UIs) associated with modern web applications. After a short description about the term "modern web applications" and the idea of what React is for, we will then commence our study of React programming, starting from the setup of the development environment. We will begin with the "manual" setup of a web server so that the concept of the interactions between a web client and a web server becomes crystal clear. In Part II of this book, we will use a "scaffolding" technique to generate a framework for a React application. Thus, we will not have to worry about the setup issues regarding the framework's organization and can instead focus on the logic behind the web.

1.1 What Is a Modern Web Application?

In general, there is no definitive definition of the term "modern web applications" or "advanced web applications." We can only describe the characteristics a modern web application may have based on the article posted on Microsoft's website [1]. According to this article, a modern web application (MWA) may have one or more of the characteristics described below:

1. Cloud-hosted and scalable – With the low-memory and high-throughput, the same server can serve more clients. As indicated in the article, "high throughput means you can serve more customers from an application given the same hardware, further reducing the need to invest in servers and hosting infrastructure." Scalability, therefore, can be manifested in two areas: the throughput in the number of clients, and the latency a server performs upon receiving a web request.

2. Cross-platform – It must be supported by Windows, Mac OS X, or Unix/Linux machines. Also, the execution platform can be on a desktop, a web, or a mobile device.

3. Modular and loosely coupled – As a LEGO brick can be reused to build different iconic products, a component can be reused in the same application or different one.

4. Traditional and SPA supported – Both the Server-Side Rendering (SSR) and Single Page Application (SPA) execution modes are supported. The characteristic is known as Isomorphic.

5. Simple development and deployment – Continuous Integration (CI)/ Continuous Delivery (CD) should be employed to support progressive web application development.

6. Easily tested with automated tests.

1.2 What Is React?

React is also known as React.js or ReactJS. To define it, we will turn to the official website for React for React [2] which states that "React is a JavaScript library for building user interfaces." However, this definition requires further elaboration because it doesn't define the term "user interfaces (UIs)." While the term can be intuitively envisioned, this technical term requires a more concrete explanation for several reasons.

First, a *user interface* is how a human user interacts with a computer. A UI is also known as the human-computer interface (HCI) or human-machine interface (HMI). Therefore, this academic area may be extended to cover the artifact aspect of the design or the technical aspect of how the interface is designed and developed. Moreover, this academic area can also be extended to cover the artistic or aesthetic aspect of how a user "touches" and "feels" about a product also known as *user experience*. For our purposes, we will focus on the *technical aspects of the definition for the HCI.*

Furthermore, when a user interface is defined like this, we can use it to refer to how a user interacts with a computer via a desktop application, a mobile application on Android phones or iPhones, and a web application. For this book, we are limiting UI to be *the "boundary" between a user and a web application*. To be more precise, the UI we are interested in is known as a "*web application UI*" or simply a "web UI". As an example, when a user accesses a browser to surf Facebook or Instagram, the browser provides a user interface for the web application. Hence, a web application can be a social media application, such as Facebook, or an e-commercial website, like amazon.com. We can now modify the definition of React as *a JavaScript library for the design and development of web UIs.*

Lastly, when we talk about the term "web applications" today, we need to consider the devices on which we surf the web. It could be a desktop PC, a laptop computer, or a mobile phone. Indeed, the development of a web UI for a mobile device imposes different areas of difficulty. Web UI design and development must consider the actual execution environment or the type of devices.

In conclusion, *React is the set of JavaScript libraries for the design and development of web UIs running on a desktop machine.* Moreover, the ecosystem of React works with React to provide extended features. For example, *React Native* is used for the design and development of web UIs running on a mobile device. While React works with Node.js to support the development of single-page applications (SPAs), *Next.js* works with the front-end React to support server-side rendering (SSR) web applications. Therefore, Next.js is considered part of the ecosystem of React. We will talk about the difference between an SPA and SSR briefly later in this chapter. Additionally, while *Bootstrap* provides an additional abstract layer for styling of a web page, there are other visual development frameworks including *Material UI* (MUI) and *Semantic UI* (SUI) to name just a few, that could be used with React to develop interactive web UIs.

However, the coverage of the ecosystem is beyond the scope of this book. Figure 1-1 only includes some popular ones and is definitely not an exclusive list. It is purely meant to introduce the concept of the React ecosystem.

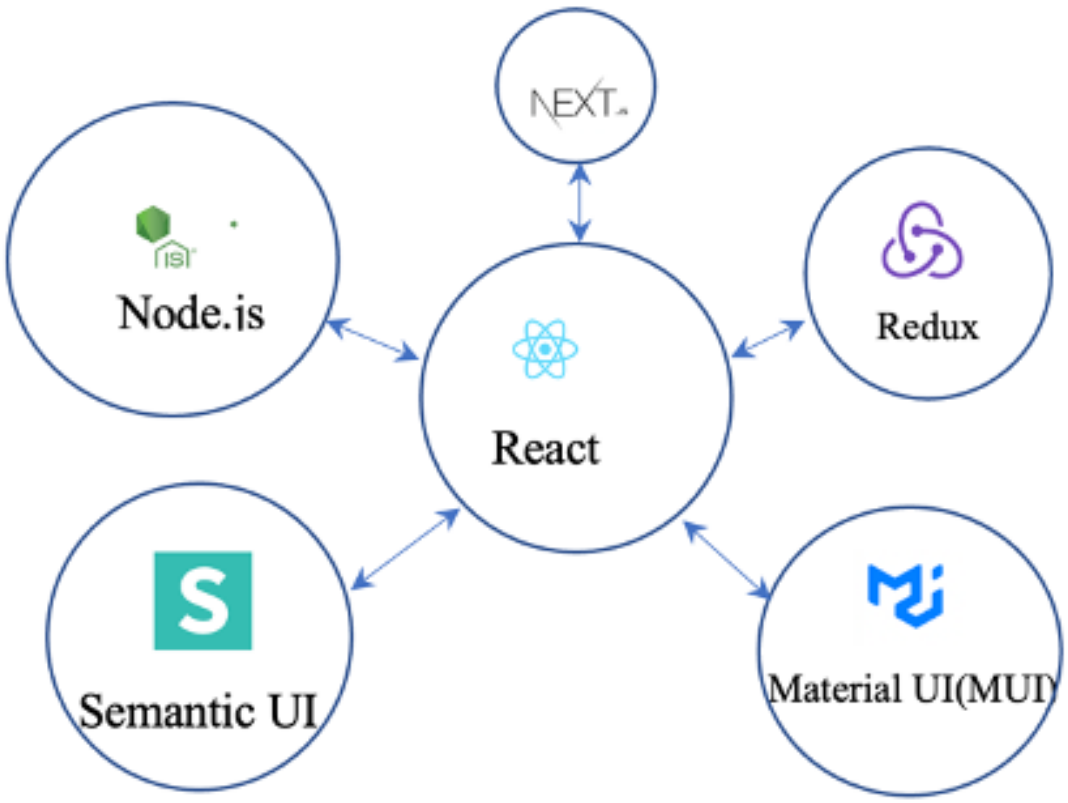


Figure. 1-1.The Ecosystem for React.

1.3 What Do We Use React for?

In the area of web application development, the acronym MVC stands for Model-View-Controller, which is a software architectural model for the development of modern web applications (Figure 1-2). Modern web applications are often

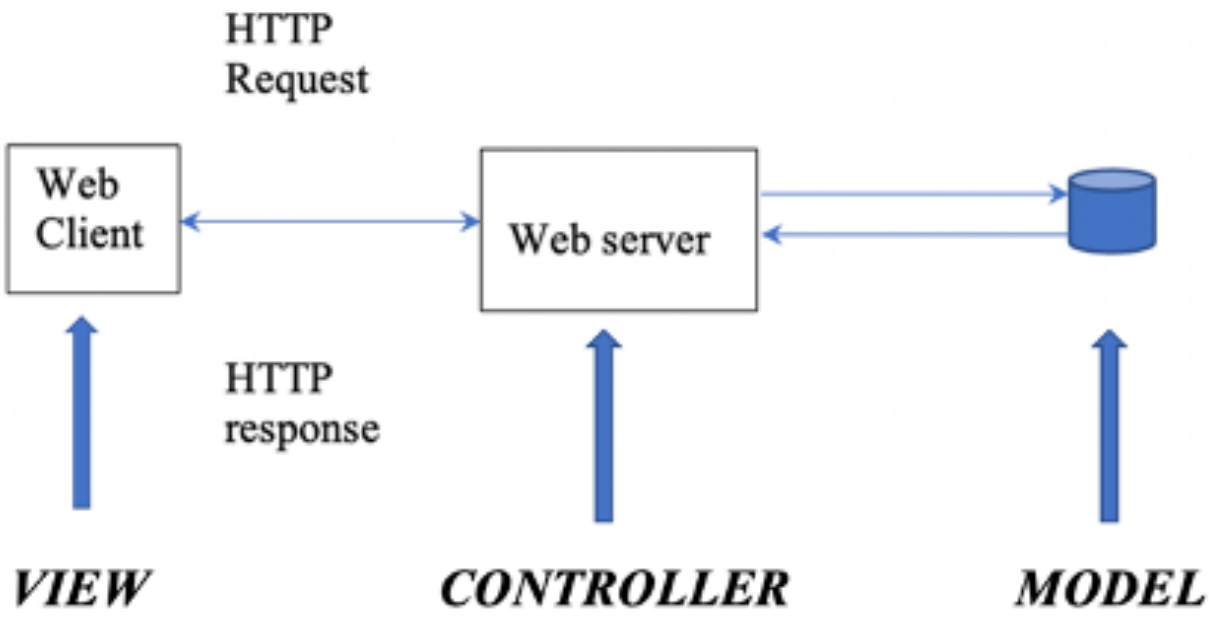


Figure 1-2. Model-View-Controller Web Application Model.

developed with the MVC architecture as a reference model. The letter 'V' in MVC is related to the UI of a web application. Hence, it is referred to as the frontend of a web app. We also call the user interface of a web application a front-end UI. The letter 'M' refers to the model used for storing data with databases. The letter "C" refers to the server function of controlling the server operations. Hence, the Web Server with the database handling is the "backend" of a web application.

1.4 What Is HTTP Used for?

The HTTP communication protocol describes the syntax and the semantics of a message to be exchanged between a web client and a web server. In Figure 1-2, a web client sends an HTTP request to the web server to request for a web page to be returned. The server parses the request and extracts the name and the location of the web page. Normally, the server searches for the web page on its hard drive. If found, it formats the web page into an HTTP Response message. It then sends the web page back to the browser. With some other requests, the server may need to "process" the HTTP request and perform specific operations. The processing of formatting a web page document into an HTTP Response is known as a *rendering* process.

As an example, if a user enters a text string known as an URL (Universal Resource Locator) via a web browser, such as "http://www.facebook.com", and hits the *enter* key, several activities will occur behind the scene. First, an HTTP request message is formatted by the web browser to include the name and the web page address, to be sent to the web server indicated by the domain name "Facebook.com" with the hidden port ID 80. Then, the web server finds the web page and formats it as an HTTP response message. Finally, the HTTP response is sent back to the client. A web browser will display the requested web page on the client's browser. Now, we can define HTTP as a communication protocol with which both the web server and the web client comply to achieve successful communications.

As an example, we will use a web project in the next chapter to demonstrate the interactions between a web browser and a local server represented by the domain name *localhost*. Instead of using two machines running two programs, we can use one machine alone to simulate the client machine and the server machine.

1.5 What Background Skills Do We Need To Learn React?

As a general consensus, we need to learn some fundamental skills prior to commencing the journey of learning about React. The prerequisites include the following:

1. HTML,

2. CSS,

3. Bootstrap and React Bootstrap (as an abstract layer over CSS),

4. JavaScript, and

5. JSX.

Once we introduce the fundamental prerequisite skills in the next several chapters, we will then shift our attention to React.

1.6 What IDE Tool Do We Need?

When we develop a web application with React for the frontend, we need to use an Integrated Development Environment (IDE) tool. We chose Visual Studio Code to be the IDE in this book for the simplicity of use and the support of the Node.js as the backend environment.

1.6.1 Installation of Visual Studio Code

You need to download the Visual Studio Code installer and then click the downloaded file to initiate the installation of the Visual Studio Code.

For the development of React running as the frontend and JavaScript code running at the backend, Node.js will be used as a single threaded web server platform. In Part I of this book, we will provide the backend with a simple express.js server; this way we can fully understand the concepts of frontend and backend communications.

1.6.2 Installation of Node.Js

Similarly, you need to download the node.js installer and install Node.js by following the direction to install Node.js.

1.6.3 Download the Sample Code

You may download the sample code in this book from GitHub repository. Once you open the browser and navigate to the GitHub page, there are several steps you need to take. First, you may copy the URL displayed under the tab “HTTPS” after clicking the green button “<>Code.” Then you can find and copy the option “URL” for the sample code under the tab “HTTPS”. Finally, you can open a Command Prompt window or a Terminal window on your own machine, and enter the following command:

```
>git clone <URL>
```

(As an example, if the<URL> is https://github.com/usapokemon/Booksamples.git, a folder with the name Booksamples will be generated on your local drive as a local repository.) An alternative is to click the “Download ZIP” to download the sample code packaged as a ZIP file. You need to “unzip” the file afterword before you can use the sample code.

Chapter 2 HyperText Markup Language

When we develop the source code of a web page, we need to use both HTML and CSS. While HTML describes the structure of a web page, CSS provides the presentation styles of the detailed elements in the web page. React is used to provide extensive capabilities of supplementing the static nature of HyperText Market Language (HTML) with the dynamic nature of a programming language such as JavaScript. In this chapter, we start with the basics of HTML. Our goal is to provide the required knowledge about HTML for learning React in lieu of covering HTML in depth.

2.1 What Is HTML Used for?

In general, we know that two programs running on two different machines respectively can communicate with each other similar to two people talking to each other. Hence, we sometimes refer to the case as two programs talking to each other. In the case of the communication between a web client and a web server, the server needs to "understand" what the client requests for, as an example, the names of the web pages and the locations. There are two requirements to achieve this goal. First, a special set of rules are needed to regulate the formats of messages. The set of rules is known as *Hypertext Transfer Protocol (HTTP)*. Second, a "web page development language" is required to develop a web page and a set of rules is needed for the client and the server to be able to understand each other. Hence, the language known as the *Hypertext Text Markup Language (HTML*) specifies the rules about how a web document is formatted. It is used for the development of *web page documents* as the *sources* of web pages. In the next section, we will explain the detailed rules about HTML.

2.2 What Does an HTML Document Contain?

As we have seen, the acronym HTML stands for *HyperText Markup Language*. A markup tag is a pre-defined web page construct that appears in angle brackets, for example, <p> for a paragraph and <title> for the title of the web page.

A web page document may be developed to include many pairs of *markup tags*. Each pair of markup tag normally starts with a *Start Tag* and ends with a matching *End Tag*. The start tag, the end tag, and the text in between, collectively are called an *HTML Element*. As an example, the element "<title>Essentials of Computing</title>" appears at the top of the tab or the browser window with "Essentials of Computing" as the title of the web page. Similarly, we can use <p> to indicate the beginning of a paragraph in the web document.

Many tags can be annotated with *attributes* to provide additional information about a tag. For example, the attribute *href* can be used to describe the web page address with the tag <a> which stands for an "anchor." Usually, an attribute

is specified with the name of the attribute followed by the value of the attribute. For example, the attribute *href* in the following element *West Chester University* makes the text "West Chester University" clickable when this web page is displayed on a browser window. Clicking the link will bring up the main page of the West Chester University.

1. HTML Tag – Examples are a start tag and the matching end tag, e.g., <h1> and </h1>, <head> and </head>.

2. Attribute – The word "style" in <body style="color: green"> is known as an attribute.

3. An element – The set of start tag, text, and the end tag—for example, <h1>Learning HTML 5</h1> altogether—together is called an element.

4. An entity – If you need to add some special symbols inside the text portion of an element, you need to use special symbols. For example, if you need to add a less than sign inside the text portion, you must replace it with '<' including the '&' and the ';'. If you want to add "spaces," you cannot simply add spaces between text in your HTML file. A browser will remove extra spaces during the process of rendering a web page. We need to add the entity " " at the place you need a single space.

5. Root Element – There is only one root element in HTML, that is, <HTML>text and tags</HTML>.

Since learning HTML is beyond the scope of this course, we will introduce just enough HTML tags needed to understand a sample HTML web page:

- HTML: <html>
- Body: <body>
- Title: <title>
- Paragraph: <p>
- Break: < /br>
- Horizontal Break: <hr /> draws a horizontal line.
- Heading Text: <h1>, <h2>, <h3>, <h4>, <h5>, and <h6>.
- Italicized Text: <i>
- Bold-faced Text: <b>
- Ordered List: <ol> for the list and <li> for each list item
- Unordered List: <ul> for the list and <li> for each list item
- Hyperlinks: West Chester University
- Image:

- src: the location of the image file tulip.jpg in the folder “images”
- alt: a brief text description of the image
- title: the title for the image
- width: the width of the image
- height: the height of the image
- align=“left” or “right”: align the image to the left or to the right
- align=“top”, “middle”, or “bottom”

For more information about HTML, please refer to the Additional Resources at the end of this chapter.

2.3 an Experiment of a Web Application Development Environment

When one is learning HTML, a popular approach is to review all HTML tags and demonstrate the syntax and semantics of them, possibly with the result of including these tags. Since learning HTML is not the major goal for this book, we will not take this approach. Instead, we will use an experiment to demonstrate the interaction of a web server and a web client. In this situation, the web browser is a web client. We will use this experiment for some of the examples in Part I of this book. Starting with Part II, we will use a more compact environment for the design and development of UIs; the server will be hidden inside a folder node_modules with other required library modules.

2.3.1 Prepare a Web Page

Once the installations of node.js and Visual Studio Code, respectively, are done, you can prepare an HTML file in the same folder where the server program is stored. The sample HTML document contains the following in a file with the file name as test.html:

```
<html>
<head>
	<title>Test</title>
	<style>
		p {
		color: rgb(40, 39, 43);
		background-color: yellow;
		}
	</style>
</head>
<body>
	<p style="color: Green">This is a Test</p>
	<p>This is another line</p>
```

```
</body>
</html>
```

2.3.2 Prepare a Web Server

We will use a simple web server running on the same machine where your browser is running. Due to its simplicity, we can use it to display some web pages with text or even images. This is called an Express.js web server written in JavaScript. For now, we will briefly explain what each statement in the server file is conducting with some caveats about the syntactical rules, when possible.

We will use the following Node.js code to implement an express server (with the file name as server.js). Since the backend operations are beyond the scope of this book, we will annotate some statements in the server code to illustrate what the server does.

```
const express = require('express');//NOTE 1 //NOTE 2
const app = express(); //NOTE 3
console.log("*** before calling static: public");//NOTE 4
app.use(express.static('public')); //NOTE 5
console.log("*** before listen ***"); //NOTE 6
app.listen(3000, function () {//NOTE 7
console.log('App started on port 3000');
});
```

Note: The statements in the server are illustrated below:

1. The 'require' function tells the backend server to import a library known as a module. In this example, the module is express.js. We do not need to specify the extension, e.g., ".js", for the library when specifying the name of a JavaScript library using require.

2. Whenever a module is required, we need to enter the command "npm install <name>" before starting up the app. If your app requires more than one module, you may enter "npm install <name1> <name2> <name3>" and the module names must be separated by at least one space character.

3. A named constant "app" is generated by calling express () to represent a server application.

4. A console.log statement displays any data/text for debugging. You can check the data/text at the console of the browser's developer's tools to find out about the execution scenario.

5. This statement declares to Node.js that all static files are stored under the "public" folder.

6. Another colsole.log statement.

7. The server listens to port 3000 after the console.log statement displays the string indicating that the server has been started successfully.Organize a File Structure and Complete the Experiment.

2.3.3 Organize a File Structure and Complete the Experiment

In this section, we will create the file structure and files initially as in Figure 2-1.

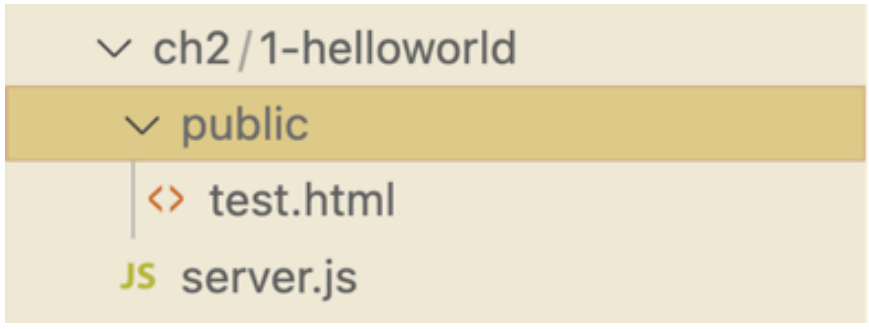


Figure 2-1. A File Structure for Testing HTML.

Notice that we use only lowercase for the name such as "1-helloworld" as recommended by React. In the project folder for the web application generated by the npx command, we will create a folder with the name public in which the HTML file test.html is stored. Moreover, if there is only one folder in the project, Visual Studio Code uses a short-hand notation to display the project title with the name of the only folder, together on the same line. In this example, Visual Studio Code displays the title of the project with the folder as "ch2/1-helloworld."

As a rule of thumb, we can store all future HTML files in the public folder prior to bringing up a browser. When we demonstrate how the server displays the source HTML document, we need to start the server first. Then, we need to bring up a browser and enter the text string "http://localhost:3000/<name>" from a web browser screen. For example, to display the name of the HTML document as test.html, we can enter the 'node server.js' command via a Command Prompt window or a Terminal Window first. Then, we can enter the URL as http://localhost:3000/test.html.

Inside the "1-helloworld" folder, there is a server.js file. In the server.js file, we use JavaScript statements to declare to Node.js that the HTML files are stored in the 'public' folder. As a result, the URL does not need to include the name "public"; Node.js knows the name of the folder where the HTML file is stored. For a beginner, this concept may be perplexing. This project example provides the details about how a web server and a web client interact internally.

Here is the detailed procedure for creating the file structure and completing the experiment:

1. On your hard drive, create a folder with a name you like, e.g., 1-helloworld, using the Terminal screen on the bottom-right side of the Visual Studio Code.

2. Enter the command: "mkdir 1-helloworld" via the Terminal.

3. Enter the command: "cd 1-helloworld" via the Terminal.

4. Next, we need to create a sub-folder ‘public’ inside the current project. Conventionally, a browser-accessible static data, such as web documents, images, and so on, are all kept in the “public” folder on the hard drive of a server machine.
5. Use Visual Studio Code to create an HTML file test.html as directed previously.
6. Check your structure using “ls” command (Mac OS X) or “dir” (Windows Powershell) when you are inside the “public” folder.
7. Enter the command “cd ..” to go back to the “root” folder (“cd” followed by a blank and two dots).
    a. Now, you may enter the following commands to prepare your environment for executing the server program. Remember that you need to be in the “root” folder of the application to enter these commands, that is, 1-helloworld.
    b. npm init --yes
    c. npm install
    d. npm install express
    e. node server.js
    f. If you run into errors, you need to check the error messages. For example, if the error message is “Module not found: express,” it tells you that your node_modules folder does not include the module express. You must have missed the step c previously mentioned or didn’t complete step c successfully. You need to go back to step c again.
    g. Now you can bring up a browser and enter the URL localhost:3000/test.html. Figure 2-3 shows the result when the web server is in execution.

2.3.4 Execution Result

In this example, the local machine is used as the server machine on which the server program is running. It is also used as a client machine on which the browser (program) is running. We have just developed a web app development environment to simulate both a client machine and a server machine.The execution result displays the web page with three lines of text:

(Line1) It shows a text “This is a Test”. It is shown in green.

(Line2) Space Line

(Line3) This is another line. It is shown in red.

The background color for visible lines is yellow.

2.4 Review Questions

1. What is a web browser, a client, and a server?
2. Who sends an HTTP request? Who sends back an HTTP response?
3. What is the term “rendering a web page” referring to?
4. What is an attribute associated with an HTML tag?
5. What is an HTML element?
6. What is an HTML entity? What is it used for?Exercises

2.5 Exercises

1. Create a web page using HTML to display a title “<your name>’s Report” with a table to list courses you took so far including the data: Course Number, Description, Year, Semester, and Grade for each course.
2. Add an <input> tag to allow a user to change the user’s name to be displayed as part of the title bar.
3. Use the web server to bring up your HTML files via a web browser.

2.6 Additional Resources

1. Learning Web Design with HTML, W3 Schools.
2. HTML Tutorials on HTML Dog – choose HTML to learn more about HTML.

Chapter 3 Cascading Style Sheets & Bootstrap

In the previous chapter, we claim that HTML is only used to provide the "structure" of a web page. The "styles" that a browser should use are still needed for a browser to display a web page with specific styles. A Cascading Style Sheet (CSS) is the mechanism to specify the styles a browser uses to display each HTML element. In this chapter, the concept of CSS is briefly introduced followed by the introduction of *Bootstrap* and *React-Bootstrap* that are used to build an abstract layer on top of CSS.

First, we will discuss the fundamentals regarding CSS to the extent needed for learning Bootstrap and using it with React. Once you learn the fundamentals of CSS, you can progressively study the remaining CSS properties. An excellent source of learning about CSS can be found at "CSS Tutorials."

Then, we will learn how to use Bootstrap to specify the styles of a React element. Conceptually, Bootstrap and React-Bootstrap are both used to form an abstract layer on top of CSS. With the knowledge of Bootstrap, the specification of styles about a React element can be simplified.

Finally, when Bootstrap can be used with HTML and React, React-Bootstrap is a total rewrite of Bootstrap for the purpose of using with React. A React-Bootstrap component can be used to define a React component seamlessly. We will introduce several commonly-used components such a Table, Lists, and Forms.

A person with the goal of becoming a UI/UX designer would want to learn more about various kinds of abstract layer software in the future.

3.1 What Is a Style Sheet?

As HTML provides information regarding the structural constructs of a web page, CSS provides detailed styles, known as properties, for each HTML tag. With the properties as the additional formatting information, a browser displays each HTML element using the specified styles, hence the name "style sheet." A style may consist of many (property: value) pairs to specify the specific format for each HTML tag or related HTML tags. The format will be applied to the specific HTML tag (known as "selectors") to specify which tags are involved. Therefore, for each HTML element, a style sheet specifies (1) a selector and (2) a set of properties for the additional format information.

As an example, if we would like to display all text in a specific "paragraph" represented by the HTML tag <p> with the color "red", we would like to specify the following style sheet, where p is the selector, color is the property and "red" is the value for the property.

```
p {
      color: red;
}
```

CSS uses the property “color” to refer to “the font color.” As you may imagine, there are so many properties for various selectors. It takes time to become familiar with them. Like learning about HTML, we will briefly introduce the way CSS styles are specified, using some frequently used styles. We also will explain the concept about the CSS “box model” and why the term “cascading” in CSS.

3.2 What Is the Box Model?

Each HTML element is treated as a rectangle with the element in the center and surrounded by several format layers, similar to the layers in an onion. The model is known as the CSS “box model” as shown in Figure 3-1.

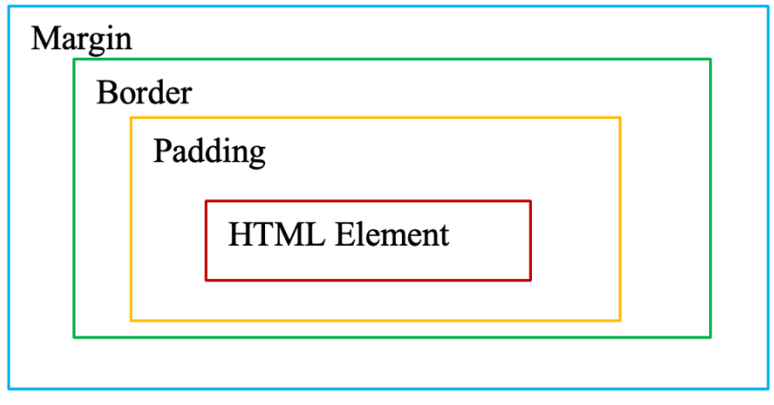


Figure 3-1. The CSS Box Model.

It is important to become aware of the sequence of the layers in the box model from the center outward: padding, border, and margin. At the center of the box is an HTML element. A padding surrounds the element; a border surrounds the padding; and the margin surrounds the border. We can specify the specific properties regarding padding, border, and margin for the purpose of presenting an element.

The way to remember this layering relationship is that “margin” is the outer most layer and is associated with the “outside” of the HTML element. The “padding” is the inner most layer surrounding the HTML element. The “border” is in the middle of the two layers.

To specify the styles, we may need to specify the sizes of the margin, border, and the padding, respectively. We could use the number of pixels, with the short-hand notation px or the percentage of the width of the parent element, for example, 10%. If the parent element has the width of 100 pixels, the padding is 10% of the parent and the padding exists only at the top and the bottom of the element; we may use the following style sheet with an HTML tag and the id attribute as test, that is .

```
#test {
      padding-top: 0px;
      padding-right: 100px;
      padding-bottom: 0px;
      padding-left: 100px;
}
```

If we need to specify the margin for this tag, we need to use the property *margin*.

```
#test {
      margin-top: 0px;
      margin-right: 100px;
      margin-bottom: 0px;
      margin-left: 100px;
}
```

We may also use a short-hand notation to specify the sizes for the four sides in the sequence from top, right, bottom, and left. For example, the short-hand notation for the above margin will be the following:

```
margin: 0px 100px 0px 100px;
      or
padding: 0px 100px 0px 100px;
```

There is yet a further simplified way to specify the values for the sizes with only two values. For example, 'margin: 0px 100px;' simply indicates that the sizes for the top and the bottom are 0 pixels, and the left and the right sizes are 100 pixels.

If all four sides surrounding an element are the same, we only need to use one number to indicate the size. For example,

```
margin: 100px;
```

indicates that the four sides of the element will have the margin size as 100 pixels.

3.2.1 How Can We Specify CSS Styles?

To specify CSS styles, we need to specify what kind of HTML tags to which the styles will apply and the styles for a property. The way to specify the type of HTML tags is known as a "selector." In CSS 3 (version 3 of the CSS specification), there are many ways to specify the selector. We summarize some commonly used ones for your information. For more details, please refer to the Additional Resources Section at the end of this chapter. In the following, we provide a summary of (1) the selector, (2) an example style sheet, and (3) a brief explanation for the example selector.

1. The Type Selector: types of HTML tags such as *p* or *h1*, etc. Here "p" is used to refer to a paragraph; "*h1*" represents the header level 1, that is, the level of heading with the largest font size.
2. The Descendent Selector: *p b { color: red; } or ul li b { color: red;}*In the box model, if the element "b" is inside the "paragraph", we may use "p b" to indicate the relationship of "b" being the descendent of p. A descendent could be a direct child, a grandchild, and so on.
3. The Child Selector: *p > b {color: red; }* The tag "b" is the direct child of "p."
4. '#' for an ID; "." for a class. If an HTML tag has an attribute '*id="name";'*, the selector and the font color may be specified as "*#name { color: red; }*". If an HTML tag has an attribute "*class= 'name'"*, the selector and the style may be specified as "**.**name { color: red; }". Note that there is a "dot" in front of the selector "name": **.**name.
5. The Universal Selector: *

   If the selector is a symbol "*", the style sheet applies to "all" tags.

Since there are myriad ways of specifying the selectors in CSS 3, you may refer to the Additional Resources at the end of this chapter for more details.

3.3 Three Methods To Specify Styles

3.3.1 In-Line Styles

First, this sample demonstrates the "in-line" method by adding a "style" attribute to the HTML tag <body> to change the font color. As you can see that this method only affects the specific HTML tag where the *style* is specified.

```
<html>
<head>
      <title>HTML and CSS </title>
</head>
<body style="color: red">
      This is a test.
</body>
</html>
```

3.3.2 Internal Style Sheet

If you have many tags that share the same styles, you wouldn't want to use the in-line stylesheet method to specify styles because you would need to add the attribute *style* in all these HTML tags in a web page. Instead, we can use an internal stylesheet.

In the previous example, we specify the "internal stylesheet" inside the <head> element. The symbol 'p' in front of a (property, value) pair—for example, (color :

blue)—is called a selector as we mentioned previously. When the tag <p> is referenced in this HTML document, the font color is changed to *blue* as specified unless an inline style is used to override the internal CSS style. There is an overwhelming list of ways to specify selectors. We only need to learn some selected methods for our purposes of learning React.

We will use the following HTML/CSS file to illustrate the internal CSS and the in-line CSS features. For simplicity, we just use selected styles in the sample code.

```
<!DOCTYPE html>
<html>
<head>
      <meta charset="UTF-8" />
      <title>Add React in One Minute</title>
      <style>
            h2 {
            color:rgb(0, 255, 123);
            }
            p {
            color:rgb(229, 255, 0);
            }
      </style>
</head>
<body>
      <h2>Add React in One Minute</h2>
      <p style="color:rgb(38, 17, 132)">
      This page demonstrates using React with no build tooling.
      </p>
      <p style="color:rgb(168, 4, 4)">
      React is loaded as a script tag.
      </p>
</body>
</html>
```

3.3.3 External Style Sheet

If you have many web pages using the same stylesheets, you don’t want to use any of the above methods. Instead, you may use “external stylesheets” which is contained in an extra file to include all styles. For example, we can use style.css in our example below. Style.css is the name of the extra file in which the shared styles are specified. In this example, this style.css must be stored in the same folder where all the web pages sharing the style.css are stored. This method is known as an absolute file position. There are ways that we can use to specify the folder location for sharing the CSS files. The details are beyond the scope of this book. If you are interested in this topic, you may refer to the additional CSS resources at the end of the chapter.

```
<html>
<head>
      <title>External Stylesheets</title>
      <style> <link rel=”stylesheet” href=”style.css”> </style>
</head >
<body>
      <h1>This is the first line.</h1><h2>This is the second line.</
      h2>
</body>
<html/>
```

With this knowledge of CSS, we will shift our attention to Bootstrap and React-Bootstrap for specifying the styles of various UI components instead of applying CSS from scratch.

3.3.4 Example With the Internal CSS Style

We will use the example in the sample file “ch03/0-css/index.html” to illustrate how to specify CSS with an internal style in the HTML <head> element including one React component defined in like_button.js. We can also use this approach later to include React components in the HTML <body>, if needed.

When we specify a JavaScript file in the body of an HTML file, we need to use the <script> element to specify the name of the JavaScript file with the extension “.js”.

But, when including a React component in the body of an HTML file, we can use a element to specify the React component. Thus, when this HTML file is rendered at the client side by a web browser, the React component becomes part of the web page.

3.3.5 Execution Result

You need to enter the following commands before you can use a browser to request for the main webpage:

> cd ch03/0-css

>npm install

>npm start

If a browser is started to show the HTML code, the result will be displayed. In the screen shot below, the ‘Like’ button is generated by using a React component which we haven’t introduced. The execution result of the project to demonstrate in-line CSS and internal CSS.

The first line is displayed in red.

The second line is displayed with color: rgb(168, 4, 4).

The third line is displayed in Brown.

The fourth line is displayed in Blue.

A button is displayed at the bottom that is generated with a React component which we haven't introduced.

3.4 Bootstrap

Instead of learning properties in CSS, we will use Bootstrap and React-Bootstrap as an abstract layer on top of CSS to complete our job of specifying the styles of React elements. We will introduce basics about Bootstrap and React-Bootstrap. For more detailed information about Bootstrap and React-Bootstrap, you should refer to the official website about Bootstrap and the official website about React-Bootstrap, respectively.

Starting now, we will use the command “npx” for establishing the scaffolding of an application structure. We will no longer use the individual server.js program to bring up an application.

3.4.1 Basics About Bootstrap

Bootstrap can be used not only with React components, but also with HTML tags without using CSS properties directly. React-Bootstrap is defined to provide React with some “built-in” components to simplify the development of UI building blocks. Hence, the official website also strongly recommends using React-Bootstrap for the development of React components, yet knowing basics about Bootstrap will help in learning React-Bootstrap.

To learn about Bootstrap, we need to understand Bootstrap styles and the setup for using Bootstrap. From the official website for Bootstrap, we learn that Bootstrap can be used to specify layouts (such as grid, media), develop components (such as buttons, forms, and tables), and use other customized styles with utilities (such as custom styles). To provide basic introduction about Bootstrap, we will introduce the major concepts about utilities (color, display, position, text) and selected components (tables, form control, input, and buttons).

3.4.2 How Does the Bootstrap Online Web Document Describe Bootstrap Styles?

Although Bootstrap web document describes all features, the descriptions are not quite “user-friendly” and require some annotation for beginners. When the document on the official website describes a specific font color, the documentation describes about possible color options with a handful of color utility classes. For instance, the class ‘text-primary’ implies that the font color is ‘light blue’ as in the HTML element <p>. The Bootstrap website uses the following representation to indicate how to add the *light blue font color* for the text “.text-primary” below:

```
<p class="text-primary">.text-primary</p>
```

Notice that the value for the attribute “class” is “text-primary” without the leading dot, and the text within the <p> element shows as “.text-primary” as an example of the text to be displayed in light blue. While this description is understandable, a more straightforward claim is to say that {class}-primary will be displayed with a light blue color, where {class} is a metaphoric term representing a contextual class, e.g., text, bg (for text and background), etc.

To use Bootstrap with React, a user must be aware of several critical rules:

1. The attribute *class* needs to be changed to *className*. Otherwise, a warning message will be displayed at the “console” under the “view > developer > developer tool” tabs. Nothing may be displayed on the screen if you do not follow this rule.

2. We may enclose all possible styles in single or double quotes as the value for the className attribute. For example, if we need to specify the background color in a React app for the tag <p>, we may to specify the values for the className attribute as follows. When the online document refers to the styles for the background color, font color, and the position as “.bg-primary,” “.text-white,” and “.text-center,” we may enclose all of them in double quotes and use these style values to be the value for the attribute *className*. The following is an example:

```
<p className=”bg-primary text-white text-center p-4 m-2”>
The paragraph will be displayed with a blue background,
and the font color for text will be white, centered. The padding
is at level 4 and the margin is at level 2.
</p>
```

3.4.2.1 Colors

As posted on the website, the following is a list of text classes for the HTML element <p> to specify the typography and the color of the text in the element:

```
<p class="text-primary">.text-primary</p>
<p class="text-secondary">.text-secondary</p>
<p class="text-success">.text-success</p>
<p class="text-danger">.text-danger</p>
<p class="text-warning bg-dark">.text-warning</p>
<p class="text-info bg-dark">.text-info</p>
<p class="text-light bg-dark">.text-light</p>
<p class="text-dark">.text-dark</p>
<p class="text-body">.text-body</p>
<p class="text-muted">.text-muted</p>
<p class="text-white bg-dark">.text-white</p>
<p class="text-black-50">.text-black-50</p>
<p class="text-white-50 bg-dark">.text-white-50</p>
```

The text under the class “text-primary” shows in blue, the text under the class “text-danger” shows in red, etc.

3.4.2.2 Utility: Sizes – Padding and Margin

In the specification for left-to-right HTML elements, there are four sides: top, bottom, left, and right. For right-to-left HTML elements, there are four sides: top, bottom, start, and end. The sizes are referring to the sides in a “box model”.

Bootstrap supports responsive web programming. The metaphorical term {breakpoint} refers to six different sizes: extra small (xs), small (sm), medium (md), large (lg), extra-large (xl), and extra-extra-large (xxl). The padding and margin are defined under the online docs category spacing under *utility* (click “utility” > “spacing”). The following is an excerpt from the online document.

The classes are named using the format {property}{sides}-{size} for xs and {property}{sides}-{breakpoint}-{size} for sm, md, lg, xl, and xxl.

1. The metaphorical term {property} is one of m(margin) or p(padding)
2. The metaphorical term {sides} is one of t (for top), b (for bottom), s (for start in left-to-right), e (for end in left-to-right), x (for both left and right), and y for top and bottom.
3. The metaphorical term {sizes} is one of 0 (no padding or margin), 1 (1/4 of the total spacer), 2 (half of the spacer), 3 (the same as the spacer), 4 (1.5 times of the spacer), 5 (three times as the spacer), and auto (set the margin and padding as automatic).

3.4.2.3 HTML Table Tag

The base class “.table” and the attribute value “table” are required for defining an HTML <table>. There are other opt-in extension style values: variants for various colors, accent including *table-striped-rows* or *table-striped-columns, table-hover, and table-active* for highlighting a table row cell or column cell. The related styles for the HTML <table> tag will be further simplified in React-Bootstrap.

3.4.2.4 HTML Input And Text Area

Bootstrap provides a rich set of form controls for <input> or <textarea> for the purpose of controlling the color, size (form-control-sm or form-control-lg), disabled or not, readonly (such as <input readonly>), and many more.

Examples are given below for <input> and <textarea> tags, respectively:

```
<input type="email" class="form-control"
id=“exampleFormControlInput1"
placeholder="name@example.com">
```

```
<textarea class="form-control" id="exampleFormControlTextarea1"
rows="3"></textarea>
```

3.4.2.5 More HTML Tags

Bootstrap supports a rich set of styles for many components including buttons and button groups. We may use them to specify the color, size, outline, disabled state, animation, etc. The list of components includes Accordion, Alerts, Badge, Breadcrumb, Buttons, Button group, Card, Carousel, close button, Collapse, Dropdown, List group, Modal, Navbar, Navs & tabs, Offcanvas, Pagination, Placeholders, Popovers, Progress, Scrollspy, Spinners, Toasts, and Tooltips. 3.4.3 Setup for testing Bootstrap CSS Styles

Before we can use Bootstrap in an application, we must describe how to set up a project structure. We describe the detailed steps for using Bootstrap in our project.

1. Setup the application file structure- Run "npx create-react-app <name>." This command npx creates a project with the name specified here as <name>. For example, if the name of the application is 1-bootstrap-npx, we can use the command "npx create-react-app 1-bootstrap-npx" to create the application structure.

2. After creating the structure, we need to add the import statement below in the index.js file that is stored in the folder src:

```
import 'bootstrap/dist/css/bootstrap.css';
```

3. Since we added this import statement, we need to run the "npm" command prior to bringing up the application:

```
>npm install bootstrap react-bootstrap
```

4. We will change the App.js to remove the redundant parts and replace it with our example below.

```
import './App.css';
function App() {
 return (
      <div className="primary text-white text-center p-3 m-3">
      <h1 className="primary text-white text-center p-3 m-3">
            Bootstrap Tutorial
      </h1>
      <p>Bootstrap is a popular abstract layer of HTML,CSS</p>
      <h1 className="jumbotron text-center">
            Test
      </h1>
      <h4 className="bg-primary text-white text-center p-2 m-1">
            Static HTML Element
      </h4>
```

```
      </div>
 ); //end of return()
} // end of App()
export default App;
```

We need to make sure that the attribute of a tag in a React component is ‘className’ instead of ‘class’ or ‘classname’ as used in the pure HTML. Here the ‘p-2 m-1’ represents ‘padding at level 2’ and ‘margin at level 1.’

5. The index.js file should look like the code below.

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';

import 'bootstrap/dist/css/bootstrap.css';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
      <React.StrictMode>
            <App />
      </React.StrictMode>
);


reportWebVitals();
```

6. We can then run ‘*npm install bootstrap react-bootstrap’* followed by the command *“node server.js.”*

3.4.4 Execution Result

Running the app with bootstrap with a browser, we will see the display.

This browser shows the heading with the text embedded into as follows:

(Large font) Bootstrap Tutorial

(Smallest font) Bootstrap is the most popular abstract layer of HTML and CSS.

(Large font)Test

(Medium font)Static HTML Element

The left edge of the title bar does not touch the window frame; that is the effect of padding and margin as “p-2 m-1” in the source listing.

3.5 React-Bootstrap

As mentioned previously, *React-Bootstrap* is also an abstract layer over CSS like Bootstrap. When Bootstrap supports HTML code as well as React code, React-Bootstrap is design to support React code seamlessly.

3.5.1 When Do We Use React-Bootstrap?

React-Bootstrap supports Layout (Breakpoints, Grids, Stacks), Forms (Form Control, Form Text, Select, Check Radios, Range, Input Group, Floating Labels, Layout, Validations), Components (Alerts, Accordion, Badge, Breadcrumb, Buttons, Button Group, Cards, Carousel, Closed Button, Dropdowns, Figures, Images, List Group, Modal, Navs, Navbar, Offcancas, Overlays, Pagination, Placeholder, Popovers, Progress, Spinners, Table, Tabs, Tooltips, Toasts), and Utilities (Transitions, Ratio, @restart/ui).

The official website for Bootstrap recommends using React-Bootstrap with React when using Bootstrap with JavaScript. Here is the excerpt:

Note: Usage with JavaScript frameworks

> *Although the Bootstrap CSS can be used with any framework including React, the Bootstrap JavaScript is not fully compatible with JavaScript frameworks like React, Vue, and Angular which assume full knowledge of the DOM. (omitted) A better alternative for those using this type of frameworks is to use a framework-specific package instead of the Bootstrap JavaScript. Here are some of the most popular options:*
>
> - React: React Bootstrap
> - Vue: BootstrapVue
> - Angular: ng-Bootstrap

In summary, if one need to customize the CSS styles, Bootstrap is a better choice. Otherwise, React-Bootstrap components should be used in our React code. For our purpose of learning React in this book, we will only need to use a subset of color schemes in Bootstrap and a subset of components in React-Bootstrap such as Buttons, List Groups, and Tables.

3.5.2 Setup for Testing React-Bootstrap

The procedure to setup for using React Bootstrap is described below.

1. Use the "npx create-react-app <name>" to create an app with the name. For example, if the name of the app is '2-React-Bootstrap,' we will enter the command below:

```
>npx create-react-app 2-react-bootstrap
```

2. We will then change the directory using 'cd 2-react-act-Bootstrap.'

3. Add the import statement in the index.js under the 'src' directory.

```
import 'bootstrap/dist/css/bootstrap.min.css'
```

4. Change App.js to include a React components.

5. The source code of App.js is shown below.

```
      import './App.css';
      import MyTable from './MyTable';

      export default function App() {
            return (
                  <div>
                        <MyTable />
                  </div>
            );
      }
```

6. The structure sample code for MyTable.js is shown below. We show how to import the React-Bootstrap <Table> and 'hard-code' three sample rows. The original code was taken from the official website for React-Bootstrap Table.

```
      import Table from 'react-bootstrap/Table';
      export default function MyTable() {
            return (
                  <Table striped bordered hover>
                        <thead>
                              <tr>
                              ...
                              </tr>
                        </thead>
                        <tbody>
                  <tr>
                  ...
                  </tr>
            </tbody>
            </Table>
```

```
            );
      }
```

The complete sample code is shown below:

```
import Table from 'react-bootstrap/Table';
export default function MyTable() {
      return (
            <div>
            <h4 className="bg-primary text-white text-center p-4
            m-4">
            React-Bootstrap Demonstration
            </h4>
            <Table striped bordered hover>
            <thead>
            <tr>
            <th>#</th>
            <th>First Name</th>
            <th>Last Name</th>
            <th>Username</th>
            </tr>
            </thead>
            <tbody>
            <tr>
            <td>1</td>
            <td>Mark</td>
            <td>Otto</td>
            <td>@mdo</td>
            </tr>
            <tr>
            <td>2</td>
            <td>Jacob</td>
            <td>Thornton</td>
            <td>@fat</td>
            </tr>
            <tr>
            <td>3</td>
            <td colSpan={2}>Larry the Bird</td>
            <td>@twitter</td>
            </tr>
            </tbody>
            </Table>
            </div>
      );
}
```

We use this sample solely to illustrate the feature of a Table component provided by React-Bootstrap. This approach does not consider the design principle of ‘reusability’ for including the table rows. We will learn how to apply the principles of modularity and reusability for the implementation of a Table after introducing React components.

3.5.3 Execution Result

The execution result of the app with a React-Bootstrap Table component is shown below. It is a table with the heading “React-Bootstrap Demonstration”, heading with four tabs “#”, “First Name”, “Last Name”, and “Username”. Three lines of records are displayed below the heading. The third table row combines the First Name and the Last Name as an example of merging two fields. This example was taken from the Table component in the package of React-Bootstrap.

React-Bootstrap Demonstration

#	First Name	Last Name	Username
1	Mark	Otto	@mdo
2	Jacob	Thomas	@fat
3	Lary the Bird		@twitter

3.6 Review Questions

1. What does CSS stand for?
2. How many methods are there for specifying the CSS styles?
3. What is the attribute to be used for specifying a font color if we use the in-line method?
4. What is the property for specifying the background color of a web page?
5. What do we need to do before the execution time for using Bootstrap in our application with React?
6. What do the space values for className mean: (1) mt-1, (2)px-1, (3) my-1, (4) mb-1, and (5) ms-1?

3.7 Exercises

1. Change your previous HTML file to display your name in red and your courses in green.
2. Change the background to yellow.
3. Now, you can change the ‘p-2 m-1’ to ‘p-4 m-2’ and see how the style changes. You should see a wider gap between the <h4> element and the edge of the window.
4. Change your HTML exercises in the previous chapter with what you’ve learned about Bootstrap.

3.8 Additional Resources

1. Webpages at “HTML Tutorials,” accessed in November 2022.
2. Webpages at “CSS Tutorials,” accessed in November 2022.
3. Webpages at “W3 Schools,” accessed in November 2022.
4. Wikipedia CSS Webpages at “Wikipedia for CSS,” accessed in November 2022.
5. Bootstrap Webpages at https://getBootstrap.com/, accessed in December 2022.
6. React-Bootstrap Webpages at https://React-Bootstrap.github.io/, accessed in December 2022.

Chapter 4 JavaScript & JSX

When JavaScript was first unveiled, it was often considered a programming language for developing client-side dynamic web pages. When Node.js was introduced in 2009, JavaScript became available for the development of backend Node.js web servers. Today, JavaScript can be used for the development of web applications from the frontend to the backend, a use known as a full-stack web application development. Furthermore, combining JavaScript with HTML derives another "hybrid language," known as JSX (JavaScript eXtensible). In this chapter, we focus on introducing JavaScript as a prerequisite skill for the purpose of learning React.

4.1 JavaScript

To execute a JavaScript program using Node.js, we need to prepare JavaScript script files and execute these script files. For example, if *test.js* contains the JavaScript source code, we could execute it using the following command:

```
>node test.js
```

For testing barebones JavaScript, you won't have to bring up any server; you can create a new file with any file editor such as Visual Studio Code and execute the file entering the command "node <name>" from a Command Prompt or Terminal window.

If the JavaScript file *test1.js* is the source code stored in the src folder, you can simply "change directory" to the folder and then enter the command "node test1.js" via a Command Prompt or Terminal window. The file *test1.js* is shown below.

Example: test1.js

```
a = 10;
b = 20;
console.log( a + b );
```

After the command "*node test1.js*" is entered, the result of "30" will be displayed on the screen.

```
> node test1.js
30
```

Therefore, we can use the Visual Studio Code environment to learn barebones JavaScript first. In our book examples, we will use "npx create-react-app <app name>" to create a skeleton app structure and store all of the barebones JavaScript files in the folder src. When we use this structure and test exporting and importing modules, we save ourselves the effort of requiring additional file structures.

4.1.1 Data Types –Variable Scope

JavaScript supports literals, i.e., data enclosed in quotes, constants (named or unnamed), and variables (with *let* or *var*). Literals are usually defined to provide the initial values for a named constant or a variable. If the named data type is a constant, the value stored in the name cannot be changed. The value a variable holds in memory can be changed. When we define a variable, we need to consider the concept of scope about a variable.

A *scope,* also known as *visibility,* is *the collection of statements where the variable or constant is considered defined.* As we had mentioned previously, we cannot use a variable prior to defining it if it is declared by using 'var.' Also, when we create a variable using '*let*' or '*const*' inside the body of an if-statement, we create a *block-scoped variable* inside the body of the if-statement. Here a *block* is a group of statements enclosed inside a pair of curly braces such as the body of an if-statement. The following example illustrates this concept of '*block scope*.' If we use '*var*' to declare a variable, we create a *global-scoped variable*. Here *global* is used to refer to the whole program from the top of the file to the end. In JavaScript, a variable defined in a different block with 'var' is referring to the same variable.

Example: test1.js (modified)

```
var num = 10; // the scope is the whole program
{
      var num = 20; // this is referring to the same num outside
}
console.log(num); // the num is the global-scoped num
```

When a variable is redefined using *var*, the scope of the variable covers the complete program. Therefore, the following result will be displayed on the screen:

```
>node test1.js
20
```

Example: test1.js (modified)

```
let num = 10;
{
      let num = 20; // the scope is inside the curly braces.
}
console.log(num);
```

When we change the keyword *var* to *let*, his code behaves differently. The *num* defined inside the curly braces is referring to another memory location. The variable *num* outside the curly braces has a different scope. The variable *num* defined inside the curly braces has a block scope inside the curly braces. When the execution moves outside of the curly braces, the scope of the *num* is the scope of the first *num*. Hence, this code generates the following result:

```
>node test1.js
10
```

If you add another console.log(num) statement inside the curly braces after the num was redefined, the program will display 20 followed by 10.

4.1.2 Data Types –Variable Name Hoisting

Normally, a variable *x* is defined by using a statement such as "let x = 10"; the variable *x* is used in a statement "y = x + 20;" because x appears to the right of an equal sign. It is a general rule in most programming languages that *a variable must be defined prior to being used*. However, in JavaScript, if a variable is defined with the keyword 'var', JavaScript applies the concept of *name hoisting*. If name hoisting is in effect, regardless of where the variable is defined, the variable becomes available throughout the block scope. Therefore, you may use a variable before it is defined. In JavaScript, how a variable is defined also determines whether or not name hoisting is applied.

1. Variables declared by '*let'* are only available inside the block where they're defined. Name hoisting will not apply.
2. Variables declared by '*var*' are available throughout the function in which they're declared. Variable name hoisting applies.

Example: (test1.js modified)

The following code will generate an error.

```
a = 10;
b = 20;
console.log( a + b );
let a = 10; // a will not be hoisted to the top
var b = 10;
```

When executing the code, this error message will be displayed:

```
a = 10;
^
ReferenceError: Cannot access 'a' before initialization.
```

Example: test1.js(modified)

But if we change 'let a = 10;' to 'var a = 10', it works.

```
a = 10;
b = 20;
console.log( a + b );
var a = 10;
var b = 10;
```

When executing the code, it works.

```
> node test1.js
30
```

The concept of name hoisting is demonstrated in the code above when we use ‘var’ to declare variables after they are used; the names are actually ‘hoisted’ to the top of the code block. If this is not what you expected, you should use ‘let’ to define the variable. The keyword ‘var’ defines a global-scoped variable and is hoisted to the top of the code block. The keyword ‘let’ does not hoist a variable the same way and must be declared before used.

Example: test1.js (modified)

```
u = 'a';
console.log(u);
var u;// u will be hoisted to the top
      if (u === 'a') {
      u = 'b';
      console.log(u);
      var u; // u will be hoisted to the top of the block.
   }
console.log(u);
```

This code will generate the result:

```
> node test1.js
a
b
b
```

Example: test1.js (modified)

```
let u = 'a';
console.log(u);

if (u === 'a') {
      let u = 'b'; // this u has a different scope than the global
      one
      console.log(u);
      }
console.log(u);
```

This code generates the following result:

```
> node test1.js
a
b
a
```

4.1.3 Operators – Triple Equal Sign

JavaScript supports many operators as in Java or other high-level programming languages with some exceptions. The most notable one is the triple equal sign for checking both the type and the value. The double equal sign only checks the value of a variable, but not the type.

Example: test2.js.

```
var u = '123';
var v = 123;

if (u == v)
     console.log("They are equal with ==");
else
     console.log("The are not equal with ==");

if (u === v)
     console.log("They are equal with ===");
else
     console.log("They are not equal with ===");
```

Executing the code generates the following result:

```
> node test2.js
They are equal with ==
They are not equal with ===
```

It is highly recommended that we use the triple equal sign if we need to check both the data type and the value of two variables.

4.1.4 Operator – Tick Vs. Quotes

In the latest JavaScript, there is a new way to provide the method of displaying the value of a variable while using the variable inside a literal string. We need to enclose the literal with ticks (`) instead of single or double quotes (‘) or (“).

Example: test3.js

```
var s = 'This is a test';
console.log (`We like to display the value of ${s}`);
console.log ('But we used single quotes instead of ticks: ${s}');
```

This example displays the following result:

```
> node test3.js
We like to display the value of This is a test
But we used single quotes instead of ticks: ${s}
```

4.1.5 Control Flows – if Then Else, Switch,

The if-statement is no different from that in Java or other high-level programming languages. The only point that we need to mention is that JavaScript provides some forms of the "false" value including false, undefined, null, NaN, 0, and the empty string ("").

The switch statement in JavaScript is the same as that in Java.

4.1.6 Iterative Controls – While Loops and for Loops

The while loop is identical to that in Java. JavaScript supports the same for loop syntax as that in Java. In addition, JavaScript supports several additional for-loop formats.

4.1.7 Iterative Controls – for Loops With "in"

In JavaScript, the "for...in object" statement, i.e., the for-loop with the keyword in allows one element in the object to be accessed at a time from the first element to the last element. Here, the object is an array variable following the keyword in. If the object is a JSON object containing several (property: value) pairs enclosed in curly braces {}, the property of each pair will be used one at a time inside the body of the if-statement.

Example: test4.js

```
var a = [1,2,3,4,5];
console.log('The elements are:');
var str = ' ';
for (const i in a)
      str += a[i]+ " ";
console.log(str);
```

Running this code will generate the following result:

```
> node test4.js
The elements are:
 1 2 3 4 5
```

In this example, the index, not the value, of the element in the array is stored in the variable i with the statement *for…in*.

Example: test4.js (modified)

```
var a = [1,2,3,4,5];
console.log('The elements are:');
var str = ' ';
for (const i in a)
     str += a[i]+ " ";
console.log(str);

var a =[{ name: 'Mary', major: 'CS'},
{ name: 'John', major: 'Math'},
{ name: 'Steve', major: 'Biology'}] ;
for ( n in a )
     console.log( a[n].name + " majors in " + a[n].major + '.' );
```

Executing the code generates the following result:

```
> node test4.js
The elements are:
 1 2 3 4 5
Mary majors in CS.
John majors in Math.
Steve majors in Biology.
```

4.1.8 Iterative Controls – for Loops With "of"

The "*for…of*" statement allows the element in the object to be accessed one at a time.

Example: test5.js

```
var a = [1,2,3,4,5];
console.log('The elements are:');
var str = ' ';
for (const i of a)
     str += i + " ";
console.log(str);

var a =[{ name: 'Mary', major: 'CS'},
{ name: 'John', major: 'Math'},
{ name: 'Steve', major: 'Biology'}] ;
for ( const n of a )
     console.log( n.name + " majors in " + n.major + '.' );
```

Executing the code generates the result below.

```
> node test5.js
The elements are:
```

```
 1 2 3 4 5
Mary majors in CS.
John majors in Math.
Steve majors in Biology.
```

Although the result looks the same, the semantics of the "*for…of*" is different.

Example: test5.js (modified)

The script file contains the code below.

```
var a = [1,2,3,4,5];
console.log('The elements are:');
var str = ' ';
for (const i of a)
     str += i + " ";
console.log(str);

var a =[{ name: 'Mary', major: 'CS'},
{ name: 'John', major: 'Math'},
{ name: 'Steve', major: 'Biology'}] ;
for ( const n of a )
     console.log( n.name + " majors in " + n.major + '.' );

var a = { name: 'Mary', major: 'CS' }
for ( const [key, val] of Object.entries(a) )
     console.log(key + ":" + val);
```

Executing this code generates the result below.

```
> node test5.js
The elements are:
 1 2 3 4 5
Mary majors in CS.
John majors in Math.
Steve majors in Biology.
name:Mary
major:CS
```

4.1.9 Array Functions – Push, Pop, Map, Filter

In JavaScript, there are several helping functions that are associated with arrays including push(), pop(), and unshift(). We will explain how to use them in the following example.

Example: test6.js

```
const fruits = ['banana', 'orange', 'kiwi'];
var str = [];
for (const i in fruits)
     str.push(fruits[i] + " ");
console.log("Original:", str);
fruits.pop();
console.log ("After pop():", fruits);
fruits.push('apple');
console.log("After push() 'apple':", fruits);
fruits.unshift( 'watermelon');
console.log('Unshift "Watermelon" adds to the beginning:', fruits);
```

Executing the code generates the result below.

```
> node test6.js
Original: [ 'banana ', 'orange ', 'kiwi ' ]
After pop(): [ 'banana', 'orange' ]
After push() 'apple': [ 'banana', 'orange', 'apple' ]
Unshift "Watermelon" adds to the beginning: [ 'watermelon',
'banana', 'orange', 'apple' ]
```

4.1.10 Functions – Named Functions, Anonymous Functions, and Arrow Functions

A function is a repeatable block of code like in C or Java. You may define a function with a default parameter in JavaScript. Default parameters appear at the end of the list of parameters.

Example: test7.js.

```
const even = function ( num ) {
     return (Math.floor(num/2)*2 === num? num: null);
}
var numbers = [1, 2, 3, 4, 5];
const even_numbers= numbers.filter(even);
console.log( even_numbers);
```

Executing this code generates the following:

```
> node test7.js
[ 2, 4 ]
```

Example: test8.js

An arrow function can be used in the previous code.

```
var even = function ( num ) {
     return (Math.floor(num/2)*2 === num? num: null);
}

var even = ( num ) => {
     return (Math.floor(num/2)*2 === num? num: null);
}
var numbers = [1, 2, 3, 4, 5];
const even_numbers= numbers.filter(even);
console.log( even_numbers);
```

The same result will be generated. The situation in which we may use an arrow function is when we pass a function as a parameter to another function. In the previous case, we could change the code to the following:

```
const even_numbers= numbers.filter( num =>
     (Math.floor(num/2)*2 === num? num: null); );
```

4.1.11 Functions – With Spread Syntax

We can use the spread syntax (with three dots) to represent the "remaining" arguments.

Example: test9.js

In this example, we provide the first argument every time we call the function sum().

```
var sum = function (a, b, c, d )
           { return a + b + c + d; }
const arg1 = 1;
const remainingNumbers = [10, 20, 30];
console.log( sum( arg1, ...remainingNumbers));
```

Executing this code generates the result below.

```
> node test9.js
61
```

4.1.12 Functions – Immediately Executed Functions

We can define an anonymous function and execute it immediately.

Example: test10.js

```
var x = (a, b) => a + b;
console.log ( x(4, 5) );
(function () {
      var foo = (a, b) => { return a + b }
      console.log("Adding 10 and 20 gives you " , foo( 10, 20 ));
})();
```

Executing this code gives you the result.

```
> node test10.js
9
Adding 10 and 20 gives you 30
```

4.1.13 Passing Data Back – Multiple Values in an Array

JavaScript does not allow a return statement to return multiple values. However, we may return an array or an object with a single return statement.

Example: test11.js

```
function returnInfo () {
var info = { name: "Mary", major: "CS" };
      return [info.name, info.major];
}

var [ personName, personMajor ] = returnInfo();
console.log( personName );
console.log( personMajor );
```

Executing this code generates the following:

```
> node test11.js
Mary
CS
```

4.1.14 Generating Callable Modules

Making a JavaScript script file a callable module in Node.js could be confusing due to the fact that there are two popular ways to achieve this goal: CommonJS and ES Modules. If you are learning node.js for the backend web server development, you will stick to the CommonJS module format, using "exports.module" and "require." For React developers, we will adopt the ES Module method. We will use ECMA2025 Specification for the purpose of creating a callable module. We will use "export" or "export default" in the callable module file, and "import <name>" or "import {<name> }" in the calling React file. We will not use a "require" statement to import a callable module in a React file as defined for CommonJS modules.

In the following, we first describe the steps to convert a JavaScript script file to an ES Module and then show the example files.

1. Generate an App using the command “npx create-react-app <name>”.
2. Use “cd” to move into the created app folder <name>.
3. Create a callable module by adding “export default” at the end of the file where the callable function is stored, say callable.js. For example, if we want to make the function additionFunction in callable.js callable, we will add “export default additionFunction” at the end of the file callable.js. Notice that the file extension “.js” is needed here. In the future, when we learn React Programming, the file extension is not needed in the import statement.
4. Add the entry to the package.json file: “type”: “module” including the double quotes.
5. Add the import statement at the beginning of the JavaScript file that imports the callable module. For example, in the main.js file, we will add the following statement: import additionFunction from “./callable.js” ;
6. Then, we can enter “node main.js” to execute the main.js script file.

Example:

Using “export default” and “import from ./callable.js”. The extension “.js” for the imported file callable.js is required here since we are not using the create-react-app command to create the app. Notice that in ECMA2025 Script Language Specification, the file extension is attached to the imported file name. When we create a React app later in this book using the create-react-app command, we do not need to add the file extension in an import statement.

In the callable module file callable.js, we define the function additionFunction as an arrow function.

```
let additionFunction = (arrays) => {
      return arrays.reduce((total, val) => total + val, 0);
}
export default additionFunction;
```

In the main.js file, we will use an import statement to import this function.

```
import additionFunction from "./callable.js" ;
let arrays = [10, 20, 30, 40, 50];
let sum = additionFunction(arrays);
console.log(`Total: ${sum}`);
```

Also, in case we use “create-react-app” to generate a boilerplate app structure, we need to add an entry “type”: “module” to the package.json file. The package.json file looks like the following one:

```
{
      "name": "1-javascript",
      "version": "1.0.0",
      "description": "JavaScript Testing",
      "main": "index.js",
      "type": "module",
      "scripts": {
            "start": "nodemon -w server server/server.js",
            "compile": "babel src --out-dir public",
      }
}
```

Notice that the “type” of the files is set to be “module” including double quotes. With this, the JavaScript file extensions can be ‘.js’ instead of ‘.mjs’.

Executing the example.js with the command “node example.js” generates the result below.

```
src> node example.js
Total: 150
```

4.1.15 Exercises

1. Define an array of numbers and list them. (Hint: Use ‘map()’ for rendering an array element.)
2. In the previous question, please add an additional number, remove the last number, or the first.
3. List all even numbers in the array. Hint: Use ‘filter()’.

4.1.16 Additional References About JavaScript

1. MDN Web Docs, “A Re-introduction to JavaScript (JS Tutorial),” accessed 1/24/2022.
2. David Flanagan, “JavaScript: The Definitive Guide (Sixth Edition),” O’Reilly, 2011.
3. Marjin Haverbeke, “Eloquent JavaScript (3re Edition),” No Starch Press, 2018
4. Basarat Syed, “Beginning Node.js,” Apress, 2014.

4.2 JSX (JavaScript XML)

4.2.1 What Is JSX?

Now that we have discussed HTML and JavaScript respectively, can you tell what kind of statement it is?

```
const element = <h1>Welcome!</h1>;
```

The syntax to the right of the equal sign looks like HTML, but the syntax to the left of the equal sign looks like JavaScript. This is a "hybrid" language syntax called JSX (JavaScript eXtensible). When we develop a React project, we will use JSX to define a React element.

When we explain how JSX works, we will use the manual web application file hierarchy as when we test HTML or CSS.

4.2.2 Fundamental JSX Syntax Rules for Developers

When JSX is used to define a React element, the React UI rendering statement becomes more "powerful" despite the fact that a developer must comply with extra syntactic rules. In this section, we cover only fundamental rules here. Please refer to the articles entitled "Writing Markup with JSX" and "JavaScript in JSX with Curly Braces" posted online under the title "Describing the UI" at the React Official Website for more in-depth discussions.

There are several rules when using JSX to define and render a React element:

1. A ternary operator can be enclosed in an HTML element, but you cannot enclose an if-statement inside an HTML element.

Correct:

```
const element = <h1>{logon? 10 : 20}</h1>;
```

Incorrect:

```
const element =
<h1>
{if (logon)
      return 10
else
      return 20}
</h1>;
```

2. A JavaScript expression must be enclosed in curly braces when enclosed inside an HTML element.

```
//will display 30 instead of "10+ 20".
const element= <h1>{10 + 20}</h1>;
```

3. The value of an HTML attribute may also be specified as a JavaScript expression enclosed inside curly braces. You may use double quotes or single quotes to specify string literals as the value for an attribute of an HTML tag, e.g., <a>.

```
const element = <a href={<a valid web page address>}></a>; or
const element = <a href ={https://www.wcupa.edu/}> </a>;
```

4. An empty HTML element, e.g., <p/>, may be used to define the content of a React element.

5. A JSX-based React element may enclose other children HTML elements.

```
const element1 = <h3>Hallo</h3>;
const element2 = <h3>World</h3>;
const element = <div>{element1}     {element2}</div>
```

4.2.3 Two Execution Issues Regarding JSX

When JSX introduces new syntax rules, there are also issues regarding the execution of a web app with JSX: (1) how the various JavaScript modules are bundled together and (2) how JSX statements are translated into a backward-compatible JavaScript syntax. This section briefly describes Webpack, a module bundler. The next section describes Babel, a compiler that translates a JSX file into a backward-compatible JavaScript file. We will only introduce the basic concepts with examples.

4.2.4 What Is Webpack?

In a React web app, a module can be defined as a JavaScript file with data or functions accessible from another file. When we create a modern web application, there may be multiple modules involved for creating the Virtual DOM tree. If the final React application is executed on the front-end machine as a Single Page Application (SPA), all modules must be packaged into a single file during the rendering process of a web page. The tool for bundling modules together is called a module bundler. Webpack is one of the most popular module bundlers currently available. As matter of fact, Webpack is not used in the web server example in this chapter; we need to start a web server manually. We will discuss more about the role Webpack plays later.

4.2.5 How Does Babel Work?

In short, Babel is a JavaScript compiler that accepts ECMAScript 2015+ code to backward compatible JavaScript code. With the help of Babel, we may use the latest version of JavaScript in the format of JSX.

Example: Babel Official Website

The App.jsx below will be transformed to the App.js file with the software *Babel*.

```
// Babel Input: ES2015 arrow function
[1, 2, 3].map(n => n + 1);
```

Babel transforms the above code to the bottom one:

```
// Babel Output: ES5 equivalent
[1, 2, 3].map(function(n) { return n + 1; });
```

Example:

As an example, we develop a web application with the name *3-jsx*. The file App.jsx file is stored in the folder *src*. The application file structure is also shown as the Figure 4-1 in the Appendix.

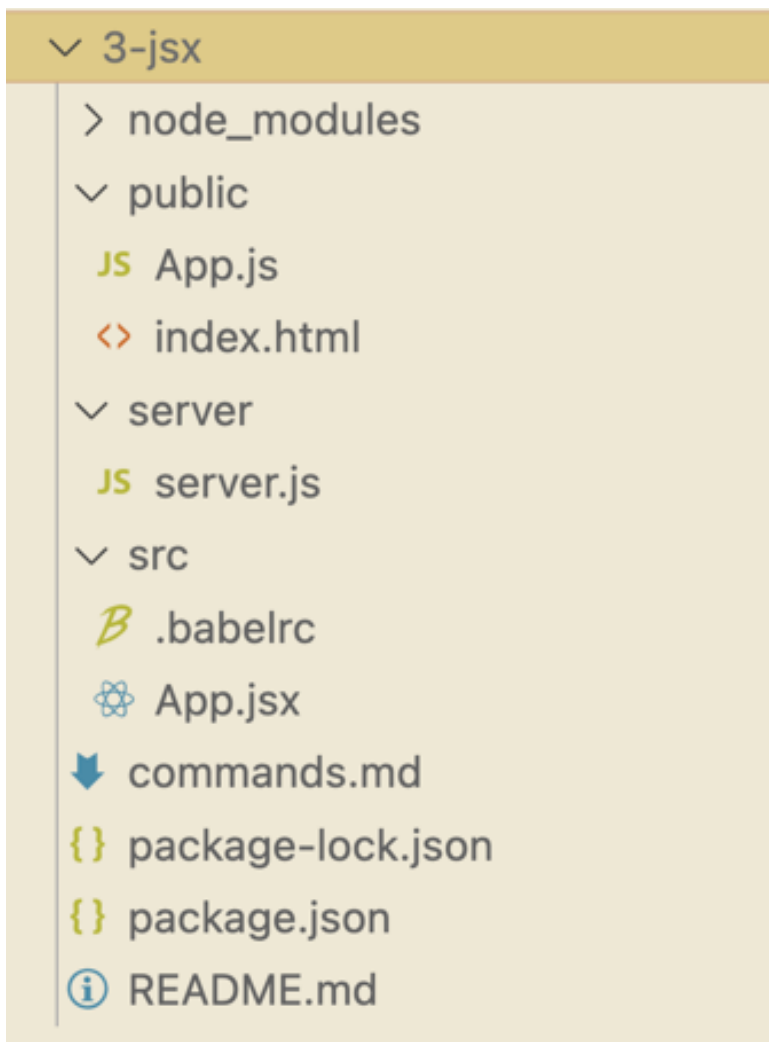


Figure 4-1. JSX App Structure (Figure 4-1 Accessible Version)

The App.jsx file in the src folder will be translated to App.js and stored in the *public folder*. If we create a web application file structure without using the command create-react-app, we need to manually compile the JSX files in an extra step every time App.jsx is modified. We need to use the shell command *"cd"* to move into the *"3-jsx"* folder first. Then, we need to modify the *package.json* file and add the following entry under the *"scripts"* property:

```
"compile": "babel src --out-dir public",
```

Your package.json file should look like the following sample:

```
{
      "name": "3-jsx",
      "version": "1.0.0",
      "description": "Reactjs.org jsx",
      "main": "index.js",
      "scripts": {
            "start": "nodemon -w server server/server.js",
            "compile": "babel src --out-dir public",
            "watch": "babel src --out-dir public --watch --verbose",
            "test": "echo \"Error: no test specified\" && exit 1"
            },
      "dependencies": {
            "express": "^4.18.2",
            "nodemon": "^1.19.4"
      },
      "devDependencies": {
            "@babel/cli": "^7.2.3",
            "@babel/core": "^7.2.2",
            "@babel/preset-env": "^7.2.3",
            "@babel/preset-react": "^7.0.0"
      },
}
```

Then, we can enter the following command:

```
> npm run compile
```

Once the command is entered, the following trace messages are displayed:

```
> 2-jsx@1.0.0 compile
> babel src --out-dir public
Successfully compiled 1 file with Babel.
```

When the command *"npm run compile"* is entered, the actual command to be executed is "babel src –out-dir public." We need to execute the compilation step every time the App.jsx file is changed. Once the compilation completes, an App.js file will be stored in the *"public"* folder. We can use the *"npm start"* command to start the application.

The original App.jsx is shown below.

```
function formatName(user) {
      return user.firstName + ' ' + user.lastName;
}
const user = {
      firstName: 'John',
      lastName: 'Wayne'
};
const element = (
<h1>
      Hello, {formatName(user)}!
</h1>
);
ReactDOM.render(
      element,
      document.getElementById('root')
);
```

The resulting App.js file is shown below.

```
"use strict";

function formatName(user) {
      return user.firstName + ' ' + user.lastName;
}

var user = {
      firstName: 'John',
      lastName: 'Wayne'
};
var element = React.createElement("h1", null, "Hello, ",
      formatName(user), "!");
ReactDOM.render(element, document.getElementById(‘root'));
```

4.2.6 A Modern Web Application Backend Using Node.Js/ Express.Js

When we manually create a web app, we may use the following procedure to start the server:

1. Bring up a Terminal or a Command Prompt Window.

2. Prepare the server file and store it as server.js.

3. Enter “npm install express.”

4. Enter “node server.js.”

5. Bring up a browser and enter the URL “localhost:3000.”

Example: server.js

```
const express = require('express');// includes the express
const app = express();// create a server object
console.log("*** before calling static: public");
app.use(express.static('public')); // static resources
console.log("*** before listen ***");
// server listens to port 3000
app.listen(3000, function () {
console.log('App started on port 3000');
});
```

When the server is started with the command “npm run start,” the screen displays:

```
>npm start
> 2-jsx@1.0.0 start
> nodemon -w server server/server.js
[nodemon] 1.19.4
[nodemon] to restart at any time, enter `rs`
[nodemon] watching dir(s): server/**/*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node server/server.js`
*** before calling static: public
*** before listen ***
App started on port 3000
```

4.2.7 Browser Screen

At this time, we can bring up a browser and enter the URL “localhost:3000.” The browser should display three lines of text with “Hello First” in line one, “3-RenderingElements” in line 2, and “Hello, John Wayne” in line 3.

In summary, two software tools are involved in the execution of a manually created web app: Webpack and Babel. If we manually create the application files and the App.jsx file is changed, we need to enter the Babel compiling command for refreshing the content in App.js. Otherwise, no matter what you do with the App.jsx, the App.js stored in the *src* folder will not be changed.

When we use the command “create-react-app” to create a web app file structure, these two steps of running Webpack and Babel as well as the execution of the server are all executed in the react-scripts package. We need only to install required modules and bring up the server with “npm install” and “npm run start” commands.

4.3 Additional References

1. Reactjs.org, “Concepts about React,” published at http://reactjs.org. Accessed in April 2024.

2. Babeljs.io, “Babel is a JavaScript Compiler,” published at https://babeljs.io/. Accessed in April 2024.

Part II Fundamental Concepts

In this part of the book, we will focus on fundamental concepts about React fundamentals.

- Chapter 5 React Components,
- Chapter 6 Properties,
- Chapter 7 State,
- Chapter 8 Events, and
- Chapter 9 Component Lifecycle & Reconciliation.

Chapter 5 React Components

A React component is a fundamental building block of a web UI. When a web page is represented internally in React as a virtual Document Object Model (DOM) tree, a node in a DOM tree is associated with a React component. Upon executing the React render() function in the definition of a React App component, a React element "root" will be created as the root of the DOM tree. A component may be associated with a child node under the root node. The definition of a React component only declares what a building block of a web UI looks like. It does not generate the node when the component is initially packaged by Webpack and sent from the server to the client until the React root element and the children components are mounted and rendered. As such, React is considered to be a declarative Web UI framework or library.

Since a web page may be composed of similar objects, it is a design goal that once a component is defined, the component can be reused. Therefore, the design and development of a React application begin with the definition of required components. When we learn how to define a React component, we will also learn how to create a React element with a *render()* function among other syntactical tasks, such as defining props, states, and events.

Although there are two approaches for defining React components (prior to React version 18), i.e., *Function Components* and *Class Components*, they serve the same purpose—declaring how a React element should be displayed on a browser window. For example, if we design a web application UI containing a table with a variable number of rows, we will develop a *Table* component containing another *TableRow* component. Then, we can generate a variable number of rows using the same component *TableRow*, iteratively. If the UI contains several tables with a different number of rows, we may reuse the Table component in which a *TableRow* component is enclosed in the render() function for the definition of the *Table* component. Thus, React can be used to provide dynamic and interactive web UI's.

In this chapter, we will learn the basics about how to define a component including the naming convention, the syntax rules for the two types of components, and the syntax for the render() function.

Before we talk about how to define React components, we need to digress a bit and talk about how to use the command "*npx create-react-app*" to create a React project scaffold and how a project server is started.

5.1 Scaffolding a React Application Project

5.1.1 Create a React App

Starting this chapter, we no longer need to worry about creating a server. Also, we no longer need to worry about using *Babel* to compile JSX files to JavaScript files prior to executing an application. What we need to do are:

1. Run the command “npx create-react-app <project name>” to create a project.
2. Enter “npm install” to generate the node_modules folder. All necessary modules including Webpack, Express, Babel, and react-scripts will be packaged in the node_modules file.
3. We only need to enter the command “npm run start” or “npm start” to start the server.

5.1.2 How Does Webpack Work?

In the node_modules folder, all required modules are stored. If we use the “npx create-react-app” command to create the project folder and run the “npm install” command, the node_modules will contain some critical modules: babel-related modules (e.g., babel-loader, babel-preset-react-app), express, htmlparser2, react, react-scripts, and webpack. Among these, webpack is the module bundler for packaging all React application resources such as the source code, the CSS files, and the image files in one package.

5.1.3 Where Is the Server?

When we enter the “npm install” command in an app created by “npx create-react-app”, a *webpack-dev-server* is also installed in the node_modules folder. As a result of executing the command *npm start*, the *react-scripts start* command starts the server. As a React UI developer, we can focus on the development of a web UI using React without worrying about how to package required modules or compile our code using Babel. If you are interested in knowing more about the development server, you can find the source code in the file path: *node_modules/webpack-dev-server/bin/webpackdevserver.js*.

5.2 Define Components

As we mentioned previously, there are two approaches for defining a component. Hence, there are two types of components. While function components are chronologically devised first, class components provide more capabilities for handling state and events until special functions known as *hooks* are devised for function components. Currently, it is a personal choice to define a component as a function or a class. In the official website of React version 18, only class components are used. In the future, if you start with a new React project, you should use function components. However, you may still run into the React code that uses class components for legacy purposes.

5.2.1 Define a Function Component

The best way to describe how to define a function component is to use an example. We will start from the definition of a function component *Message* in which one line of <h1> text is rendered. Then, we will modify the requirement to enclose several lines of the same text. Using this approach, students can learn rules about how to define a function component with the consideration of

reusability. As the first rule, we always capitalize the initial of a component name, e.g., Message, and use the camel case style for the full name of a component, i.e., StudentInfo. This way we can distinguish a React function component from a JavaScript function by name.

5.2.2 Example of Defining a Function Component

Assuming that we will display one line of the <h1> text "Test Component," we can define a function component returning only one line of the text. We start with defining a function component *Message* in the *Message.js* file displaying only one line of text in the *return()* statement below:

```
import React from 'react';

export function Message(props) {
      console.log(' * Message - before return');

      return (
            <h1 className="bg-primary text-white ">
            Test component
            </h1>
      )
}
```

In this function component, the parameters passed to the function component are called "props" which stands for properties. We will talk more about how to pass data values via props in the next chapter.

In this function component Message, the statement *console.log* displays the text message on a web browser's console. It is added only for the purpose of debugging. Using Google Chrome, you can find the console message by clicking the "console" tab on a "developer tool" screen after clicking *View* > *Developer* > *Developer Tools.*

In this example, there is no render() function. Instead, a *return()* statement serves the purpose of "declaring" the rendering operation. In this case, a level one header is created as defined in the function component Message.

We will need to add this component to the *App* component to complete our example. In the Virtual DOM, the component App is a node with the node Message as the child node of App. The *parent-child* relationship is important in a React DOM tree hierarchy because React allows data flow only *one-way* from a parent node down. The reverse data flow must be achieved via an "event handling" during the lifecycle reconciliation process to be discussed later.

The *App.js* file contains the definition of the function component *App* as a function component.

```
import './App.css';
import { Message } from './Message';
```

```
function App() {
console.log("* App.js - before return")
return (
      <div className="App">
            <Message />
      </div>
      );
}
export default App;
```

Note: Some syntax rules are illustrated below:

1. The file extension for the source file "./Message" is not needed in the import statement; you don't want to add the extension ".js" as in the following import statement:

```
      import from "./Message.js";
```

2. If the *Message component* inside the file Message.js is exported with the keyword "*export"* without using the keyword "default," e.g.,

```
      export function Message (props);
```

, the import statement needs to specify the component with curly braces.

```
      imports { Message } from "./Message";
```

3. If the Message *component* inside the file Message.js is exported with the keyword "*export default*," the import statement must not enclose the component with curly braces, i.e., "{}".

For example, if the export statement is written as follows:

```
      export default function Message (props);
```

, the import statement can be written as

```
      import Message from "./Message";
```

4. In the function component *App*, there is a required *return()* statement enclosing the <Message /> as the content inside a tag.

5. When a Message component is enclosed in a *return()* statement, the *App* component becomes the parent component of the Message component. When the Message function is rendered, the function component App creates the React element with the text message in it.

5.2.3 Add Several Instances of the Same Component

Let's now change the requirement a little bit. Let's say we want to display the "Message component" four times. As you can imagine with what we have learned so far, we only need to change the return() statement in the App.js to the following:

```
return (
<div className="App">
      <Message />
      <Message />
      <Message />
      <Message />
</div>
);
```

For now, this is an acceptable style of coding before we introduce the concept of "props" (properties). Eventually, this style will be replaced with a more scalable iterative style of code. This style as it stands now is known negatively as a "hard-coded coding style" that should be avoided later. In general, a "hard-coded coding style" reduces reusability and increases the cost of maintainability. For example, when the requirement is changed, we need to rake through the original code finding required changes. The activity is known as *code maintenance*. As a component becomes more complex, the issues of reusability and maintainability can become prominent.

Note: The following syntax rules of the *return()* statement are required:

1. Within a return statement, we can return exactly one outermost HTML element. We can enclose as many elements as you like inside the single HTML element, but they all need to be enclosed in one outermost start tag and end tag. In this example, we use … as the single element in which several instances of <Message /> components are enclosed.

2. The 'return' keyword must be followed by some JavaScript code immediately. The "return" keyword cannot appear in one line by itself.

We present some incorrect and correct examples below to illustrate the syntax rules about return().

Incorrect Example:

If we do this, JavaScript considers that the return statement is ended and the becomes an error.

```
render() {
      return
      <div>
            <h1>Hello, world!</h1>
      </div>
}
```

Correct Example:

```
render() { // return must not be followed by a blank line
return <div>
      <h1>Hello, world!</h1>
      <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
</div>
;
}
```

Recap:

At the same line where the keyword *return* is written, there must be at least another token after *return*. Otherwise, React considers that this line is completed. The first character of the next line becomes an unrecognized character, therefore, a syntax error. This rule may be changed in later versions of React.

5.3 Define a Class Component

Historically, function components are the only way to declare a component. Later, class components are added to the language features due to the addition of the state of a component. In the most recent version of React website for React version 18, class components are totally removed. New users of React need to learn function components only. However, we still cover class components because the audiences may run into older React projects developed prior to version 18.

Previously, we used an example to illustrate how to define function components. In the following, we convert the Message component to a class component.

```
import React, { Component } from 'react';
export class Message extends Component {

      render() {
      console.log(' * Message - before return');
      return (
            <h1 className="bg-primary text-white ">
            Test component
            </h1>
            )
      }
}
```

5.3.1 More Exercises of Defining Class Components

In the following example, we define another class component Welcome to introduce the syntax rules. First, we will need to add the import statement. Then, we will need to define the component with the keyword “class” and “extends.”

Finally, we need to add a render() function call to enclose the return statement at the end.

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

export default class Welcome extends Component {

      render() {
            console.log("here in Welcome");
            return (<h1>Hi, John </h1>);
      }
}
```

In the following, we define yet another class component Comment in the file Comment.js. In this example, we use curly braces to enclose this.props.note which will be explained in the next chapter. Here you only need to know that the property note will be passed as an attribute when the component Comment is used later in the code for the component App.

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

export default class Comment extends Component {
      render() {
            console.log("Comment: inside render()")
            return (<h1>Hi, {this.props.note}</h1>);
      }
}
```

Note:

Since this is defined as a class component, we need to enclose the property in curly braces, e.g., *{this.props.note}*. We will introduce the semantics of *this.props* in the next chapter.

5.3.2 Putting Things All Together

Now, we assume that the requirement is changed again to include one instance of the *Welcome* component and one instance of the *Comment* component. Since we have defined these two components previously, we can use a class component *App* to complete the render() statement.

The source code for the file App.js is shown below:

```
import React, { Component } from 'react';
import './App.css';
import Welcome from './Welcome';
import Comment from './Comment';

export default class App extends Component {
render() {
```

```
      return (
      <div className="App">
            <div>
            <Welcome user="Sara" />
            </div>

            <div>
            <Comment note="Sara is beautiful." />
            </div>
      </div>
      )
      }
}
```

5.4 Rendering a Component

5.4.1 Child Components and Parent Components

When a component A includes another component B in the body of the render() or return() statement, we consider the component B a *child component* of the *parent component* A. The child-parent relationship is also shown in a DOM tree at different hierarchical levels. We will need to describe how to define properties in the child component and how to use the properties in the parent component when the component renders a node.

5.4.2 the Syntax for Rendering a Component

When creating a web UI, we first create a React element with the name *root* and insert the children components of the root node to form the React virtual DOM tree. The process is known as the *rendering process* of the React root element. In a function component, the return() statement declares the content of the basic building block. In a class component, the function render() encloses statements including a return() statement in the body of the render() function to declare the content of the basic building block. We now focus on the syntax rules about the render() and the return() statements.

The syntax of a return statement declares what should be rendered in a function component. The return statement is simply a return() function call enclosing exactly one HTML element.

```
import React from 'react';
export default function Message() {
console.log(' * Message - before return');
return (
      <h1 className="bg-primary text-white ">
            Test component
      </h1>
      )
}
```

In a class component, we need to define what to render by enclosing a return() statement within the body of the render() function. We use the same component Message written as a class component below.

```
import React, { Component } from 'react';
export default class Message extends Component {
render() {
      return (
      <h1 className="bg-primary text-white ">
            Test component
      </h1>
      )
      }//end of render()
} //end of component
```

In each component, there is only one element to be returned or rendered. That is, you must enclose all HTML elements, JSX statements, and React components (to be considered as an HTML element) with one outer start tag and an end tag although you may enclose as many inner elements as possible inside. If you need to render more than one HTML element, you may use an additional opening tag and closing tag to enclose these elements. As an example, if we need to return two instances of the <Message /> component in App.js, we need to enclose both instances with and as follows:

```
<div>
      <Message />
      <Message />
</div>
```

You may also use empty opening tag and closing tag as “<>” and “</>” instead of “ ” and “ ".

The rendering statement below is incorrect because two instances of React component are returned as two elements.

```
import React from 'react';
import Message from “./Message”;

export default function App() {
```

```
      console.log(' * Message - before return');
      return (
            <Message />
            <Message />
      )
}
```

To correct the previous error, the return() statement must be changed to the following code.

```
import React from 'react';
import Message from "./Message";
export default function App() {
console.log(' * Message - before return');
      return (
      <div>
            <Message />
            <Message />
      </div>
      )
}
```

5.5 Summary of the Rules About Defining React Components

1. We use the following naming conventions:
    a. We use all lowercase letters to define the name of a React project.
    b. We use a CamelCase convention for the name of a component, e.g., StudentData or BookTable.
    c. In the illustration of the syntax, we use angle brackets to refer to a meta language object. For example, in the description of a student name, we may use the meta symbol <student name> for 'John' or 'Mary' when we need to use a student name as the value of a property.
    d. We use only lowercase letters to represent the name of a property such as this.props.student_name.
2. There are two types of React Components: Function Components and Class Components.
3. The syntax for rendering an element in a function component is different from that in a class component. In a function component, you simply add a "return" statement with the JSX to describe what needs to be rendered, whereas in a class component, you will enclose a return statement inside a "render()" function call.

5.6 How Can a React Element Be Rendered Into a DOM Tree?

React creates a “root” element first in a virtual DOM tree in memory and then renders each child component into the virtual DOM tree as a node. Whenever there are changes in the state of any node, any properties, or there may be a forced update to any of the nodes in the DOM, the rendering process is repeated again. We will start learning how to create the “root” element for a document using React version 18.

5.6.1 Using React Version 18

```
import React from 'react';
import ReactDOM from 'react-dom/client'; // React 18
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';
import 'bootstrap/dist/css/bootstrap.css';
const root = ReactDOM.createRoot(document.getElementById('root'));
const element = <React.StrictMode><App /></React.StrictMode>
```

Note: In React version 18, there are several required changes from previous version:

1. The import statement for ReactDOM is changed to use “react-dom/client.”
2. Use ReactDOM.createRoot() instead of ReactDOM.render().
3. Call root.render(element). Please refer to the posted webpage at the official website of react [6].

In summary, for each React project, we start with creating a root element using a version dependent function known as *render()*. For example, using React version 18, we first create a root and provide a name to the root node.

```
const root = ReactDOM.createRoot( document.getDocumentByID(“root”));
```

Then we can define an element as follows:

```
const element = <h1>Hello World </h1>;
```

Finally, we can generate the React DOM by calling the *render()* function:

```
root.render ( element );
```

We need to remember that after the root node with a specific element is generated, we cannot change the structure of the DOM tree. When the content or data associated with an element is changed, the only way to change how the web page looks is to reconstruct a new element and re-render the element.

5.6.2 HTML Element Versus React Element

We have explained the term HTML element previously in Part I of this textbook. We also define *React component*s. We should not be confused by the term "*React element*."

In any HTML document, an *element* is the collective term for a start tag, texts, and an end tag. For example, if we enclose a line of text "Learn React" between the start tag and the end tag, the result will be "<h1>Learn react</h1>" that is called an *HTML element*. In each HTML document, there is a single "root" element with the start tag <html> and the end tag </html>.

A React element is like an HTML root element but implemented with JavaScript and HTML together, known as JSX. An example of a React element is the *root* node in a virtual DOM tree associated with a web document. Inside a virtual DOM tree, each child becomes another React element. In fact, the major difference between an HTML element and a React element is that a React element (a node in the DOM tree) contains attributes such as *type, key, props, and children* for each node associated with the React element.

A *React component* is a "declarative" representation of a node in the virtual DOM tree representing a React web UI. If a React element encloses several React components in the rendering process, the children nodes will be associated with the children components.

5.6.3 What Is a Virtual DOM?

We called the DOM tree maintained internally inside React at the frontend a *virtual DOM tree*. The original virtual DOM tree was created initially after a request was sent to the server and the server sent it back with all of the CSS files, node_modules, image files, favorite icons, and JavaScript modules in one package. This packaging work was done by the software *Webpack*. Afterward, whenever there are changes in the state or properties, or there is a forced update, the memory version at the frontend will be checked and a new version will be generated to reflect the changes. As a performance improvement, only the node that was changed and the children nodes that are involved in the state or properties will be changed. The procedure of updating the necessary nodes in the virtual DOM tree is called *reconciliation*. During the reconciliation process, only the nodes involved will be updated for the purpose of improving performance. Using this virtual DOM concept, React does not need to regenerate a whole new copy of the original web page as a DOM tree every time only a small number of nodes is changed. That is why the DOM tree maintained by React a virtual DOM tree because it was not always generated from a web page document package to be sent from a backend server.

5.7 Summary

Here is the summary of some major concepts and rules regarding components:

1. Components are the basic building blocks of a web page.
2. There are two approaches for specifying a component: function component and class component.
3. We always use upper case for the first letter of a component name and a camel case style for naming a component.
4. After we define a component, we can use it as a tag in the render() function. For example, if a web page has a search box and a table, we will at least define a *Searchbox* component and a *Table* component. Then in the render() function, we can return the following:

```
<div>
      <Searchbox />
      <Table />
</div>
```

5.8 Review Questions

1. How many types of React components are there?
2. What is the naming convention for a React component?
3. What is the correct import statement for the following component?

```
export default function Message (props);
```

4. What is the correct export statement if the component is imported as below?

```
import {Student} from "./Student";
```

5. Is the syntax of the following "import statement" correct?

```
import React, Component from 'react.js';
```

6. What is a frontend of a web application, a client, or a server?
7. What programming language(s) can used for the development of the backend server when we use React to develop the front-end UI?
8. What is a virtual DOM? What is the difference between an HTML DOM and a virtual DOM? Which DOM does React use?
9. What is the difference between an element and a component in React?
10. What is the data flow direction from a node in a DOM tree to the children node(s) of a parent node?

5.9 Exercises

1. Convert the function component Message to a class component.
2. Define an array of names as [John, Mary, Steve] using a component *NameList* first. Define a class *App* component to render this component.
3. Convert the previous class component to a function component.

5.10 Additional Resources

1. Reactjs.org, “Concepts about React.” Accessed in August 2022.
2. Babeljs.io, “Babel is a JavaScript Compiler. “ Accessed in August 2022.

Chapter 6 Properties (Props)

Defining a function in a high-level programming language without specifying parameters minimizes the usage of the function. When React features passing data to a function via parameters, the React function becomes a general-purpose building block. Reusability is enhanced. Conceptually, defining a React component is analogous to defining a function in any high-level programming language, and often, we need to consider passing props to a component. The term 'props' is derived from the word properties and is used as a keyword inside a class component to refer to the props. For example, we can use this.props.studentName to refer to the property studentName.

In this chapter, we will talk about the following topics regarding props:

1. What are React properties or props?

2. How can we render a component with props?

3. How can we pass data via props defined in child components?

4. What is the generic nature of props?

6.1 What Are Properties in a React Component?

In the previous chapter, we discussed how to define class components and function components. We used a simple toy-like component *Message* again to display one line of text. Then, we added three lines of code using a hard-coded style. The problem of this coding style is that the revised component cannot be reused; we cannot enclose <Message /> to display a variable number of lines. The solution to this issue is the use of props as an attribute with the React component tag, e.g., <Message *line*="number"/>, where *line* must be defined in the component Message as a prop. We will discuss further details next.

6.1.1 Why Do We Need Properties?

As mentioned previously, we need to share data between a parent component and a child component in React. We need to learn how to pass data to the child component when we define the parent component and how to define props and use the props when we define a child component. This way we use properties to pass data from a parent component to a child component. In an HTML document, we can specify attributes to provide more information about an HTML tag. Similarly, we can provide data via props to a child component. Conceptually, "props" to components is semantically the same as "attributes" to HTML tags. The way of passing "extra information" from one component to the child components is via *props* (meaning properties). Syntactically, the keyword "props" is not only used to define a property, but also required in a class component to refer to a property. As an example, we defined a *Message* component in the previous chapter to display a hard-coded number of lines of text.

Now, let's change the requirement for the component *Message* again to illustrate what properties we need to pass to this component. We need to define these properties inside the Message component and then pass the values of the properties to the Message component. Let's say we need to display the following:

```
Test component #1 Hello John
Test component #2 Hello Mary
Test component #3 Hello Steve
```

With the introduction of props in defining the line number, e.g., #1, #2, and #3, and the names "John," "Mary," and "Steve", we can define two attributes to the <Message /> tag as *this.props.number* and *this.props.name* and enclose them with curly braces.

We need to modify the definition of <Message /> to the following:

```
import React, { Component } from '../../../ch6/1-textlist/
     node_modules/@types/react';
export class Message extends Component {

 render() {
     console.log(' * Message - before return');
     return (
          <h1 className="bg-primary text-white ">
          Test component for {this.props.number}: {this.props.name}
          </h1>
     ) // end of return
 } // end of render
} // end of component
```

With this change to the Message component, we need to modify the *App* component as the parent component of the *Message* component. We can pass the values for the attributes: *number* and *name*. Then, when we enclose the three instances of the <Message /> tag with the values for the attributes in the App component. With this approach, we need to change the App component to the following:

```
import React, { Component } from 'react';
import './App.css';
import { Message } from './Message';
export default class App extends Component {
 render() {
     return (
          <div className="App">
          <Message number='1' name="John" />
          <Message number='2' name="Mary" />
          <Message number='3' name="Steve" />
          </div>
     ) // end of return()
```

```
 } // end of render()
} // end of component
```

In summary, we can recap what we have learned.

1. We need to use “import { Message }” instead of “import Message” in App because the definition of the component Message is defined with “export” instead of “export default.”

2. The attribute *number* for the tag <Message> is actually a prop defined inside the Message component as *this.props.number*. In HTML, we call these attributes. In React, we call them *props* because we defined them with *this.props.number* and *this.props.name*.

As a general consensus, this approach is definitely an improvement over a “hard-coded” style. The reusability of the tag <Message /> is retained and the maintenance of the source code is improved. As such, adding props to a component enhances the reusability of the component. A revised syntax for defining props in a function component requires the props to be enclosed in the heading of the function with curly braces. The revised syntax can further avoid the integrity issue effectively. We will discuss how to define props using this syntax when we define class components whenever possible.

This code as it stands now can further be improved. We will introduce another approach to display a list of similar React elements , e.g., <Message />, using a list in the next section.

6.1.2 Defining Properties in the Example Component

In this example, we will change the Message component to define an array for storing three names. Then, we will use a for-loop to push an array operation in JavaScript one line at a time to this array of text lines. Then, we can call the render() function to render the array of text lines. We will illustrate this idea with actual code.

We define the Message component as a function component.

```
import React from 'react';

export function Message({name}) {
      return (
            <div>Hello! {name}! </div>
      )
}
```

In this revised code, we don’t need to use props.name to refer to the props to be passed to the component from its parent component. This revised syntax allows type checking and improves maintainability. Don’t forget the curly braces enclosing the props!

6.1.3 Defining NameList Component

The Message component can be considered as a building block in the final text list. Then, we can define a parent component NameList to enclose the Message component below. In this example, we change the style of prop passing to the latest style. In the latest version of React, we can enclose props with curly braces and pass the props as a parameter to a function component.

```
import React from 'react';
import { Message} from './Message';

export default function NameList({name}){

//let name = props.name;
 return (
      <div className="bg-info text-white text-center p-2 m-2">
            <Message name={name}/>
      </div>
      )
}
```

6.1.4 Defining App

Later in this book, we will learn how to define a text box which allows a user to enter a name to be added to an array of names inside the parent component. With that change, we could achieve true reusability eventually. Yet for now, we simply define a variable as nameArray in App next to specify three 'hard-coded' names.

In the App component, we define a nameArray with the elements of three names in the array: [John, Mary, Steve]. We can use a list.map() function to enclose the <NameList> component three times within the body of the map() function dynamically.

```
import './App.css';
import NameList from './NameList';

function App() {
const nameArray= ['John', 'Mary', 'Steve']
      return (
            <div>
            {
                  nameArray.map((name) => <NameList name= {name}/>)
            }
            </div>
      );
}
export default App;
```

6.1.5 Execution Result

When a browser displays the web page, three lines are displayed with text in white and the background of each line in light blue:

Line 1: Hello John!

Line 2: Hello Mary!

Line 3: Hello Steve!

6.2 Define Properties for Function Components and Class Components

When we need to pass data from a parent component to a child, we need to define these data as properties in the child component. We will discuss how to compose props in a child component and how to pass data from the parent component to the child component in the next several sections.

6.2.1 Composing Properties in Child Components

Properties are defined with the keyword “props” in front of the name of a property. The syntax for defining properties in a function component is slightly different from that in a class component. In a class component, we need to specify “this” as follows for a property with the name of the props defined in <name> as follows:

```
this.props.<name>
```

In a function component, we do not need to specify “this” as follows for a property with the name defined in <name> as follows:

```
props.<name>
```

For example, if we need to define the property “age” in a class component, we need to use “this.props.age” inside the class component. If we define the same property in a function component, we simply use the prop as “props.age” inside the function component. As we mentioned previously, a could use a revised syntax to pass props to a function component by enclosing the list of props in curly braces. Using this revised syntax, we could refer to a prop using the name of the prop without the prefix props.

In year 2023, a revised syntax for passing props to a function component was devised. We have used the new style without describing the details in this chapter. We can pass the list of props in parenthesis as arguments. The props must be enclosed with curly braces. In the rest of the book, we will show both the conventional syntax and the revised syntax for function components.

In summary:

1. Function Components – props are arguments and referenced by *props.<name>* or simply by the name of the prop.

2. Class Components – props are referenced as *this.props.<name>*.

Function Component:

```
export default function Welcome(props) {
      return <h1>Hi, {props.name}</h1>;
}
```

Revised Function Component:

```
export default function Welcome({name}) {
      return <h1>Hi, {name}</h1>;
}
```

Class Component:

```
import React, { Component } from 'react';
export default class Welcome extends Component {
      render () {
            return <h1>Hi, {this.props.name}</h1>;
      }
}
```

6.3 Passing the Values of Properties to Child Components

In an HTML tag, we may need to pass extra data via attributes. Similarly, when a child component is included in the body of a render() function, we may need to pass the values of the properties. We must use the correct names as the attributes for the props defined in the child component when we include an instance of a child component in the render() or the return() function of a parent component.

For example, if the child component <Message> uses *this.props.text* or *props.text* inside Message.js, the parent component must use the following to refer to the child component Message with the attribute as *text*:

```
<Message text='some text'/>
```

Note: the attribute '*text*' matches the definition of this.props.*text* after removing the prefix *this.props* in Message.js.

6.4 Examples

When we use examples in the next section to illustrate how to use props to pass data, we may also pass a function name as the value of an attribute. For example, an event handler is a function to be passed from a parent to a child. While data flow is only one way for a given DOM tree from a node down, the reverse data flow requires specific considerations. We will discuss the idea of passing an event handler later.

6.4.1 Example#1 - Passing Number via a Property

In previous chapter, we defined a Message component in the file Message.js without using props. In this example, we change the requirement and present the solution below.

Requirement Change: What should we do if we need to display, say, five lines of text using Message.js without “hard coding” five instances of <Message />?

Solution:

Change App.js to include the following code:

```
import './App.css';
import TextList from './TextList';

function App({number}) {
// define a property with the name “number”
// const number = props.number;
      return (
      <div>
            <header>
            <title>Learn Props</title>
            </header>
            <TextList number={number} />
      </div>
      );
}
export default App;
```

Then, we can define the TextList as a function component receiving the property number.

```
import React from 'react';
import { Message } from './Message';

export default function TextList({number}) {

      let textArray = [];
```

```
      for (let i = number; i > 0; i--) {
            textArray.push(
            <div key = {i}>
            <Message textstring={`This is a test - ${i}`} />
            </div>
            );
            }
      return (
            <div className="bg-info text-white text-center p-2 m-2">
            {textArray}
            </div>
      )
}
```

In this example, we demonstrate how to pass the value of props to a function component. In the definition of App, the value of the number of lines is passed to the component TextList, and the value is stored in a variable *number*. It is used as a control variable, regulating how many instances of <Message> component will be added to the final array variable textArray. The browser displays five lines of “This is a test” all in white font over the light blue background.

6.4.2 Example#2 - Passing an Array

In this example, we changed the requirement slightly to display an array of names. First, we will define a NameList component. The requirement can be satisfied by passing the complete array to the *NameList* and letting the component take one name at a time and use JavaScript statement to add the name to an array. We need to define an array of names *nameArray* in App.js.

```
import './App.css';
import NameList from './NameList';
export default function App() {
const nameArray = ['John', 'Mary', 'Steve'];
      return (
      <div>
            {nameArray.map((name)=><NameList name={name} key={name}/
      >)}
      </div>
      );
}
```

Then, we define a NameList component receiving one name at a time and providing the value as a property to the <Message> component using the property “name.”

```
import React from 'react';
import { Message} from './Message';
```

```
export default function NameList({name}){

      return (
            <div className="bg-info text-white text-center p-2 m-2">
            <Message name={name}/>
            </div>
      )
}
```

Finally, the Message component is changed to the following code:

```
import React from 'react';

export function Message({name}) {
      return (
            <div>Hello! {name}! </div>
      )
}
```

Once a browser is brought up, a screen of three lines will be displayed with the font in white over the light blue background:

Hello John!

Hello Mary!

Hello Steve!

6.4.3 Example#3 - Display a Table

Displaying a table seems to be similar to previous examples. But the difference is instructive. We will define a BookTable component in which each row of the table is defined as a BookTableRow component. Using this approach, we modularize the BookTable component and the BookTableRow component. Each component is defined in a separate file. Our goal is to make the BookTableRow component reusable.

In App.js, we will include an instance of <BookTable /> in the project.

```
import './App.css';
import BookTable from './BookTable';
import React from 'react'

function App() {
      return (
            <BookTable />
      );
}

export default App;
```

First, the BookTable component is defined as follows as a class component. It uses BookTableRow as a child component.

```
import React, { Component } from "react";
import BookTableRow from "./BookTableRow";
export default class BookTable extends Component {
render() {
const books = [
 {id : 1, title: 'Gone with the Wind', price: 13.95 },
 {id : 2, title: 'How can we deal with office politics?', price:
     20.0 },
 {id : 3, title: 'Fairy Tales in Egypt', price: 12.95 },
 {id : 4, title: 'No Nonsense React Programming', price: 65.95 },
]
const rows = [];
books.forEach((book) => {
           rows.push(
           <BookTableRow book={book} key={book.id} />
           );
     });
     return (
           <table className="table table-sm table-striped
           table-bordered">
           <thead className="bg-info text-white text-center m-2
                p-2">
           <tr colSpan="3">
           <th>ID</th><th>Title</th><th>Price</th>
           </tr>
           </thead>
           <tbody colSpan="3">
           {rows}
           </tbody>
           </table>
     );
}
```

Then, we need to define a BookTableRow component that accepts a book object via props. The project displays the table of books below.

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

export default class BookTableRow extends Component {
 render() {
     const book = this.props.book;
     return (
           <tr>
           <td className="text-center">{book.id}</td>
```

```
            <td className="text-center">{book.title}</td>
            <td className="text-center">
                  ${Number(book.price).toFixed(1)}
            </td>
            </tr>
      );
 } //end of render
} // end of component
```

The execution result displays the table with four rows below:

ID	Title	Price
1	Gone with the wind	$13.9
2	How can we deal with office politicss?	$20.0
3	Fairy Tales in Egypt	$12.9
4	No Nonsense React Programming	$66.0

6.4.4 Example#5 - a Bug Regarding Props

We will demonstrate one bug that could be easily ignored for a beginner. The way to pass values for properties defined in a child component is to use an attribute with the name exactly the same as the <name> in the definition of a child component. For example, in a Student component we need to define the properties with the *"studentName"* and *"totalCredits"* to represent the student's name and the total credits taken, respectively.

We define the Student component as below.

```
import React from "react";

export default function Student (props) {

const id = props.id;
const studentName = props.studentName;
const totalCredits = props.totalCredits;
      return (
            <tr>
            <td className="text-center">{id}</td>
            <td className="text-center">{studentName}</td>
            <td className="text-center">{totalCredits}</td>
            </tr>
      );
}
```

In a parent component where the <Student> component is used, we need to specify the props with correct attribute names, i.e., id, studentName, and

totalCredits. If we mistakenly specify the attributes using '*name*,' instead of '*studentName*,' the table will be displayed without showing the name of the student in each row.

```
students.forEach((student) => {
      rows.push(
            <Student
            id={student.id}
            name={student.studentName}
            totalCredits={student.totalCredits}
            />
      );
});
```

The webpage screen shows that the student names are not displayed. The incorrect table is displayed without showing the Name field for each row as follows:

ID	Name	Total Credits
1		50
2		33
3		90
4		100

To fix this bug, we need to change the attribute from "name" to "studentName" for <Student>.

```
students.forEach((student) => {
      rows.push(
            <Student
            id={student.id}
            studentName={student.studentName}
            totalCredits={student.totalCredits}
            />
      );
});
```

Then, the correct output is displayed.

ID	Name	Total Credits
1	Mary	50
2	John	33
3	Steve	90
4	Chuck	100

6.5 Properties Are Immutable

Props are immutable. We cannot change the value of a property in a child component. In other words, a property cannot appear to the left of an equal sign.

6.5.1 Define App

We will define a component App to enclose a component *NumberList* with the attribute numberArray. We pass an array as the value for numberArray to *NumberList*. We will then show the code for the component *NumberList*. In the NumberList, we will try to change the value of the prop *numberArray* to see how React behaves. We first show the code for App.

```
import './App.css';
import NumberList from './NumberList';

function App() {
      const number = [1,2,3,4,5];
      return (
            <div>
            <header>
            <title>Learn list</title>
            </header>
            <NumberList numberArray={number} />
            </div>
      );
}
export default App;
```

6.5.2 Define the Component NumberList

We will then show the code for the component NumberList. In the render(), we try to mutate the value of numberArray with the statement:

```
this.props.numberArray = this.props.numberArray.push(100);
```

We will bring up a browser and run the application to inspect the behavior. We show the result in the next section. An error occurs without a surprise.

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

export default class NumberList extends Component{
      render() {
      this.props.numberArray = this.props.numberArray.push(100);
            return (
            <div>
            The number list is {this.props.numberArray.map(n =>
            <div key={n.toString()}>{n}</div>
```

```
            )}
            The reverse is {this.props.numberArray.reverse().map(n=>
            <div key={n.toString()}>{n}</div>)}
            </div>
            )
      }
}
```

6.5.3 Execution Result

An error message will be displayed at the console of a developer's tool if we use Chrome as the web browser with nothing to be displayed on the left side of the browser's screen. This immutable nature of properties makes it imperative to introduce another feature or method for passing data from a parent component to a child component or even sharing data among siblings. This next feature to be discussed is called *state* and will be discussed in the next chapter.

6.6 Summary

1. We use properties or props to specify more sharable information when enclosing a child component in a parent component. This way we change the hard-coded style when defining a child component.
2. Although props may be passed from a parent to the children, props are immutable; you may not change the values of properties inside the child component.
3. To define mutable data, we need to use *state* that will be the topic for the next chapter.
4. If you use the latest versions of React, you may use either function components or class components. But you just need to be aware of the different syntax. Usually, we use a class component when we need more complicated features such as state or lifecycle methods. With Function Components, you must use *hooks* for accessing states.

6.7 Exercises

1. (1) Define a Book component with the state of the variable *books*. Each book contains three data items: id, title, and price. (2) Define a TableRow component with the name *book* as the prop. Use the App.js to show the heading and the body of the table.

2. (1) Define a Student component with the following data: id, name, and total credits taken; (2) Define a StudentTable with props for the minimum and maximum number of credits taken; and (3) Define a StudentTableRow component to display one row in the student table.

3. Change the example for displaying a list of <Message /> elements to add the props id and the *names* without using the hard-coded style. You may need to add an HTML <input> tag to allow a user to enter the total number of the <Message /> component you need to display.

6.8 Additional Resources

Web article, “Components and Props,” reactjs.org, accessed in September 2022.

Chapter 7 State

In the previous chapter, we learned that defining props is an important step in defining components. However, props are immutable. The need to define "mutable" data. This requirement leads to the React feature of state.

We will cover details for defining the state of a component in this chapter. We will provide the purpose of state and the rules for defining state variables in function components with some examples.

7.1 the Purpose of Defining State in a Component

As we have mentioned, a property defined with props inside a component is immutable. It cannot be changed even within the component. If you use a child component inside a parent component and pass some properties from the parent to the children, the properties cannot be changed in the parent or the children. If you define sibling components, you cannot even pass the values of properties from one to another sibling components. The solution is to define "state" instead.

You may recall that in React, data flow is only one way – from top of the DOM tree downward. You cannot pass the data up the DOM tree nodes or change the value of a property. As a result, you need to define state variables in a parent component and pass it downward to a child node. If there is a need to modify the values of the state variables defined in the parent component, a callback function must also be passed down as a prop from the parent to the children. When the state must be changed, the callback functions will be called inside the event handling function(s) of the children component. This may sound too abstract without showing some examples. But we need to elaborate further about the characteristics of the state in a component.

7.2 What Is the State of a Component?

A state is a collection of (name: value) pairs defined within a component with the following characteristics:

1. State is local,
2. State is mutable with the *setState()* hook,
3. State can be passed down from one node to its children,
4. Reverse update from a child component to change the value of the state value(s) defined in a parent component is achieved by passing a callback function as a property from the parent to the child. This process is discussed in the next chapter when event handling is discussed.

In a class component, a *state* of a component is a set of name-value pairs. Each state is defined as a collection of one or more of the (name : value) pairs, and these name-value pairs will be enclosed by curly braces. If there are two or more

name-value pairs to be surrounded by the curly braces, each data-value pair will be separated by a comma. This format is known as a JSON format. These JSON names can be referenced in a JSON formatted object with the prefix “this.state”. The term JSON format represents a set of (name : value) pairs.

Usually, in a class component, the set of the JSON pairs collectively is known as the state of a component. For the purpose of consistency, we may informally refer to these name field in the (name : value) pairs as *state variables* despite the fact that they are actually not directly updatable as local variables. However, we call the state of a function component a *state variable*. If we define a function component and use the hook (a function) *useState*, we can define a state variable, and the mutating function for changing the value of the state variable. We will discuss defining the state in a class component first.

7.3 Defining State in a Class Component

As there are two types of components—that is, function and class components--there are two ways to define the state in a component. Either way, we need to define the set of (name : value) pairs. We need to use the syntax in a class component shown below to define state. We will explain how to use hooks in a function component to define state variables later.

We always define state in the constructor of a class component. One of the main purposes of the constructor is to store the (name : value) pairs in the constructor. We may need to link an event handling function with the keyword ‘this’ using *bind()* in a constructor.

We don’t want to change the state value directly by assigning a value to a name. Instead, we will change the value using *this.setState({})* function, where the curly braces are required to enclose the new value for the name. We don’t want to simply change the state variable with an assignment statement. If we do, we actually bypass the lifecycle handling of trickling the new value down the DOM tree.

We will use an example to illustrate this idea below.

```
import React, {Component} from 'react';
export default class Toggle extends Component {
      constructor(props) {
            super(props);
            this.state = {isToggleOn: true};
            // This binding is necessary
            this.handleClick = this.handleClick.bind(this);
      }
      handleClick() {
            this.setState(prevState => ({
            isToggleOn: !prevState.isToggleOn
            }));
      }
```

```
      render() {
      return (
            <button onClick={this.handleClick}>
            {this.state.isToggleOn ? 'ON' : 'OFF'}
            </button>
            );
      }
}
```

In this code, we define a class component Toggle with a state variable isToggleOn. Then, we enclose an instance of <Toggle value={true} /> in App.js. Finally, we can then use “npm install” and “npm run start” to start the project. You will see the button being displayed:

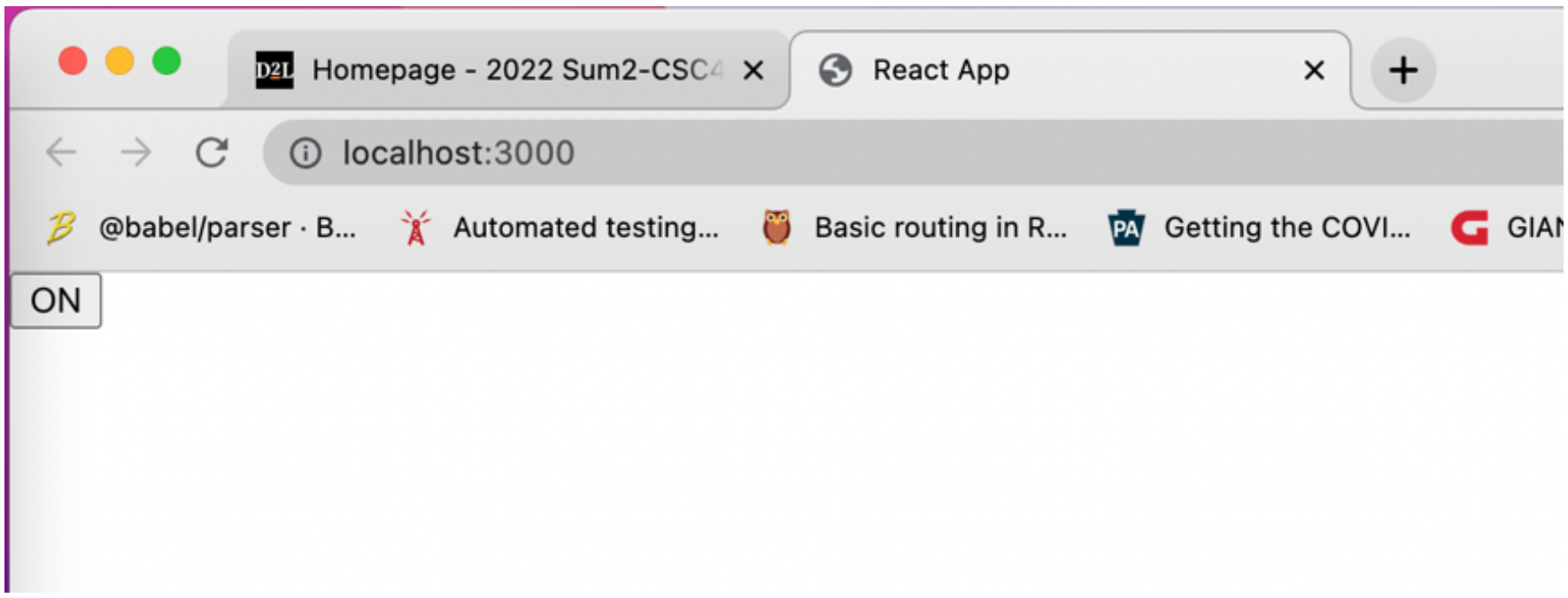


Figure 7-1. Browser Output of the Toggle Button.

When we click the button “ON”, the state is changed, and the button displays “OFF”.

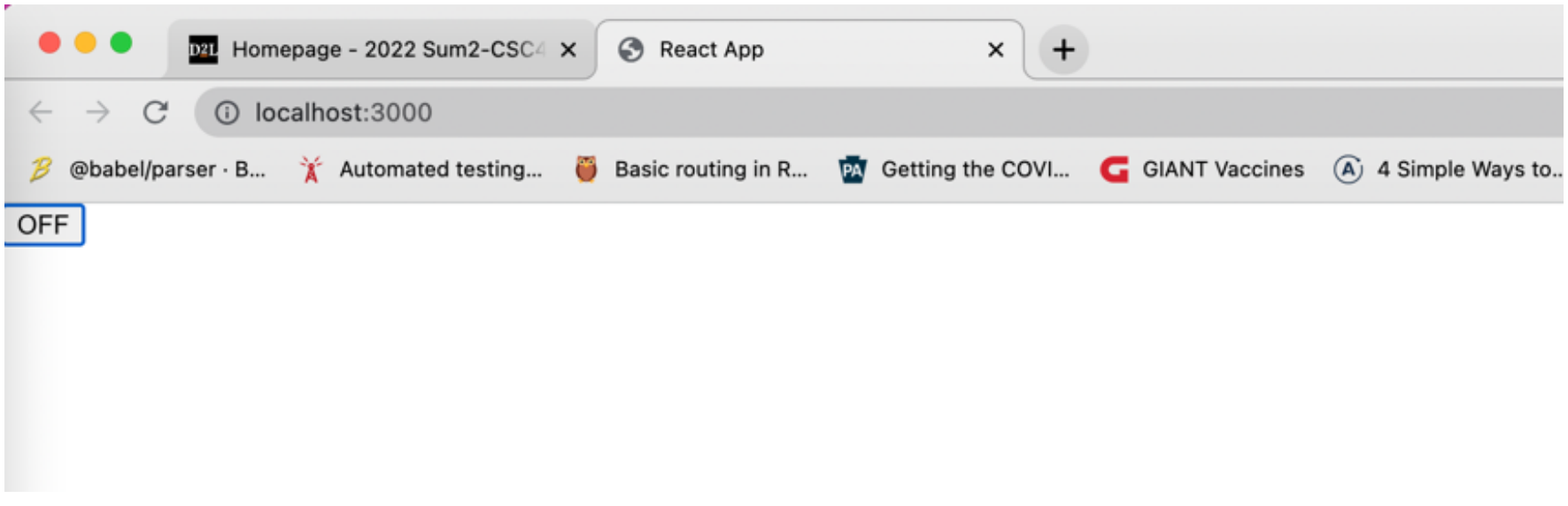


Figure 7-2. Clicking the Button ON.

7.4 Defining State Variables in a Function Component

In a function component, a state consists of one of more state variables defined individually by a hook function useState() with the initial value for the state variable passed as an argument to this hook function. In React, certain actions are required to be achieved via calling some special functions. These “utility” functions are called hooks. The ultimate goal for providing hooks is to allow state to be defined in function components. The name in the (name : value) JSON pair in a class component is called as a state variable in a function component. In a class component, the value in the (name : value) JSON pair defines the initial value for the name. In a function component, the initial value for the state variable is passed to the useState() function as an argument.

For example, in the latest official document of Reactjs.org, the hook useState(initial value) is used to define a state variable with the function to change the value of the state variable. The feature to define class components has been removed with the simple reason that a user needs to learn only one way to define a component.

We will use an example to explain how to use hooks to define state variables. As you might noticed, when we refer to state in class components, we use the singular term state. When we refer to state in function components, we use the term state variables.

In the file Toggle.js, we define a function component ToggleWithHooks with the state isToggleOn and the mutation function setIsToggleOn. Formally, when the data of state needs to be changed, we need to pass the new value to the mutation function setIsToggleOn(). The state variable and the mutation function are returned by the hook setState

In the following example, we begin to use the event handling function that has not yet been introduced. The concept will be discussed in the next chapter.

```
import React, {useState} from 'react'; // Note#1

export default function ToggleWithHooks ({value}) { // Note#2
            const [isToggleOn, setIsToggleOn]: useState(value);
            const handleClick() { //NOTE#3
            setIsToggleOn ( !isToggleOn );
      }
      return (
            <button onClick={this.handleClick}>
            {isToggleOn ? 'ON' : 'OFF'}
            </button>
      );
}
```

Note:

1. The import statement must be changed.
2. In a function component, there is no constructor. The way to define a state and the mutation function for changing the state variable is to call the hook function *useState()* with the initial value for the state variable *isToggleOn*.
3. The setState() function is used to change the value of the state. We will define the event handler handleClick() as follows:

```
const handleClick() {
      this.setState(prevState => (
      {isToggleOn: !prevState.isToggleOn}));
}
```

The argument for the setState() is a call back function receiving one argument prevState.The App.js looks like below:

```
import './App.css';
import Toggle from './Toggle';
import React, {Component} from 'react';

export default class App extends Component {
      render() {
      return (
            <div>
            <ToggleWithHooks value={true} />
            </div>
            );
      }
}
```

Note: in this App.js, there is no need to add a constructor because we do not need to define any state variables inside App.js.

7.5 State With Multiple State Variables

We will demonstrate how to allow a user to enter a name and the total number of credits the user has taken. Then the app will display the remaining credits needed to graduate.

In the component ChangeStates, two state values are defined: *name* and *credits*.

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

export default class ChangeStates extends Component {
      constructor (props) {
            super(props);
            this.state = {
```

```
            name: "",
            credits: 0,
            } // end of this.state
      this.enterName = this.enterName.bind(this);
      this.enterCredits = this.enterCredits.bind(this);
} //end of constructor

      enterName ( n ) {
      console.log("name: " + n.target.value);
      this.setState ( {
                  name: n.target.value
      }
      )
}

      enterCredits ( c ) {
            console.log("credits" + c.target.value)
            this.setState ( {credits: c.target.value})
      }
      render () {
      return(<div>
      <h4 className="bg-primary text-white text-center p-2 m-2">
      Please enter your name and your credits below
      </h4>
      <label>Name</label>
      <input className="form-control"
                  name="name"
                  type="text"
                  value={this.state.name}
                  onChange={this.enterName} />

      <label>Credits</label>
      <input className="form-control" name="credits"
            type="number"
            value={this.state.credits}
            onChange={this.enterCredits} />

      <h2 className="bg-primary text-white text-center m-2 p-2">
            Hello! {this.state.name}! <br/>
            Credits: You need additional {120-this.state.credits}
            credit(s) to graduate.
      </h2>
      </div>) //end of return()
      }
}
```

In the App.js, the ChangeStates component is rendered.

```
import './App.css';
import React, { Component } from 'react';
import ChangeStates from './ChangeStates';

export default class App extends Component {
      render = () => {
      console.log("* In App.js");
      return(
            <div>
                  <ChangeStates />
            </div>
            )
      }
}
```

7.6 the Scope and an Incorrect Way To Define a State

For beginners, it may be confusing to decide where the State of a component should reside. Also, the right way to change the value of a state (variable) may also seem to be confusing. We will discuss the following topics in this section.

1. State is Local

2. Passing the name of an Event Handling Function

7.6.1 State Is Local

State is defined in a component and can only be changed in which the component the state is defined. If the value of the state is used in a child component, the value may be passed down from a parent to a child component as the value of a property.

We are demonstrating an incorrect way to define two buttons in App.js. Since we need to add two buttons, an immediate idea is to define a state in the App component and also include two instances of the <Toggle> component.

```
import React, {Component} from 'react';
export default class Toggle extends Component {
      constructor(props) {
            super(props);
            this.state = {isToggleOn: true};

            // This binding is necessary to make `this` work in the
      callback
            this.handleClick = this.handleClick.bind(this);
            }
      handleClick() {
            this.setState(prevState => ({
            isToggleOn: !prevState.isToggleOn
            }));
      }
      render() {
            return (
            <button onClick={this.handleClick}>
                  {this.state.isToggleOn ? 'ON' : 'OFF'}
            </button>
            );
      }
}
```

In the Toggle component, the state encloses the state “variable” isToggleOn. The state is local to the Toggle component.

In the App.js, the App component defines the state with the name flag and incorrectly intends to change the state values for the child components Toggle and Toggle1 that are reusing the same definition of the Toggle component. However, the state defined in App component can only be accessible from within the component App. The state in App is only used as the values for the props with the name of value. It is no longer used inside the component Toggle. Hence, the state defined in the App is redundant and useless despite the fact that the project seems to work as expected.

```
import './App.css';
import Toggle from './Toggle';
import Toggle1 from './Toggle';
import React, {Component} from 'react';

export default class App extends Component {
      constructor(props) {
      super(props);
      this.state = {
      flag : true
      }
```

```
      this.handleClick = this.handleClick.bind(this);
      }

      handleClick() {
            this.setState(prev => ({
            flag: !prev.flag
            }));
      }
      render() {
            return (
            <div>
                  <Toggle value= {this.state.flag} />
                  <Toggle1 value={!this.state.flag} />
      </div>
      );
      }

}
```

7.6.2 Passing a Function Name as the Value of a Property

State can be accessible only from the component in which it is defined. If changing the state is necessary from a child component, the name of an event handling function must be passed as a property to the child component. Note that we can only pass a function definition as an arrow function or just the name of an event handling function instead of calling the function directly when we actually intend to pass the name.

We will defer the further discussion of this issue about passing an event handling function to the next chapter.

7.7 Review Questions

1. What is the state of a component used for?
2. What is the difference between properties and state (variables)?
3. Can the value of a state variable be passed to its child components?
4. Can the value of a state variable x with the value of 5 be changed with x = 10 inside the component x was defined? Is there any issue related to this assignment statement?
5. What is the correct way to change the value of x in the previous question?

7.8 Exercises

1. Change the syntax of the class component ChangeStates to ChangeStatesWithHooks component as a function component.

2. Modify the App.js and add one more button. The two buttons will display “on” and “off,” independently. When you click on one button, the button changes the state.

3. Can you modify the previous problem to click one button so that the other one also changes? If you cannot achieve this, can you define a screen with two values: true and false? Also define one button. When this button is clicked, the left value changes to false, and the right value changes to true, and vice versa.

7.9 Additional Resources

1. Reactjs.org, “State and Lifecycle,” https://reactjs.org/docs/state-and-lifecycle.html, accessed in August 2022.

Chapter 8 Events

In this chapter, we will talk about the following topics:

1. React Synthetic Events

2. State with Events

8.1 React Synthetic Events

Actually, we began to use event handling in the previous code when defining an attribute onChange to associate with a textbox input HTML tag, or onClick to associate with a HTML button tag to toggle the state from ON to OFF or vice versa. Here we explain more about the concept.

An event in React is very different from an event in HTML. In the execution environment of a program running on a computer, events are handled by a browser. For example, a user clicks a submit button, that triggers an event. We can find the Action and the Method that are associated with an HTML <Form>. As a result, an event handling function or method is executed at the server side. Although the concept of event handling function, or event handler, is similar in React, the execution is totally different. In React, events are handled entirely by React on the client-side. Therefore, these events are called Synthetic Events [8].

Let's use the previous example of a Toggle button in which an HTML <button> is enclosed to explain more about synthetic events.

Inside the constructor, we will add "super(props)" to call all constructors defined in the parent class of the class and passed all props from the parent to the child. Remember that the internal data flow that React supports operates only one way – from a parent to the children. There is no way for a child to pass data back to its upper parent using a prop. This is what we meant by "one-way data flow." We must use the event handler passed down from the parent component. Here is the source code of the component Toggle.

```
import React, {Component} from 'react';
class Toggle extends Component {
      constructor(props) {
      super(props);
            this.state = {isToggleOn: true};
            this.handleClick = this.handleClick.bind(this);//NOTE #1
      }
      handleClick() {//NOTE #2
            this.setState(prevState => ({
            isToggleOn: !prevState.isToggleOn
      }));
      }
      render() {
```

```
            return ( // NOTE#3
            <button onClick={this.handleClick}>
            {this.state.isToggleOn ? 'ON' : 'OFF'}
            </button>
            );
      }
}
```

Note:

1. We need to connect the event handling routing with “this” if the event handling routing is not an arrow function. (This is yet another rule.)
2. You can see that the handleClick() is not an arrow function.
3. The <button> tag is the HTML tag “button” because it does not start with an uppercase letter. If we define our own button, like our Toggle button, we will use an uppercase character for this reason. An HTML tag requires an attribute to define the event handler using the attribute “*onClick*.”

In summary, there are several differences between HTML events and React events:

1. Syntax for defining an event handling function.
2. Syntax for Passing the event handling function.
3. Binding with the keyword this if the Function is not an arrow function.
4. What attributes can be used with a React event tag?
5. Passing arguments to an event handling function

We define the events and event handlers in this chapter. We will cover more about the relations of events and UI handling—for example, conditional rendering, lists, and forms—in the next couple of chapters. The organization and some contents will closely follow the basic concepts posted at the Reactjs.org website [8].

8.2 Syntax for Defining Event Handling Functions

When we need to associate a synthetic event with a component, e.g., a button, we need to define an event handling function as the value of the attribute onClick. At the official website reactjs.org [8], there is an excellent example to compare an HTML tab <button> with that in React:

For example, the HTML syntax:

```
<button onclick="toggleState()">
      Toggle from ON to OFF or OFF to ON
</button>
```

is slightly different from that in React:

```
<button onClick={toggleState}>
      Toggle from ON to OFF or OFF to ON
</button>
```

Note: The attribute of the React tag <button> includes “onClick”; you will not find “action” or “method” associated with the <button> tag. The event handling function *toggleState* is executed on the client machine by React. When we specify the event handler in React, we only specify the NAME only. Do not pass the parenthesis after the name as toggleState() as in the HTML counterpart. That will cause a never-ending execution of toggleState as an infinite loop. We will examine this concept when we discuss the Component Life Cycle and Reconciliation later.

8.3 State With Events

In the next several paragraphs, we will illustrate how to assign event handlers using two examples. The first example requires us to define one button in the component Toggle. When the button is clicked, it changes the display from ‘ON’ to ‘OFF’ and vice versa.

The second example requires us to define one button with two labels: one displays ‘ON’ and another displays “OFF.’ When the single button is clicked, the first label changes from ‘ON’ to ‘OFF’ and the second label changes from ‘OFF’ to ‘ON.’ When the button is clicked again, the two labels flip from ‘OFF’ to ‘ON’ and from ‘ON’ to ‘OFF,’ respectively.

8.3.1 Example of Using Two Buttons

As mentioned previously, the first example requires us to define one button showing ‘ON’ initially. When the button is clicked, it changes the display from ‘ON’ to ‘OFF’ and vice versa. The code for the Toggle component is shown below.

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

export default class Toggle extends Component {
      constructor(props) {
            super(props);
            this.state = {isToggleOn: true};
            this.handleClick = this.handleClick.bind(this);
      }

      handleClick() {
            this.setState(prevState => ({
            isToggleOn: !prevState.isToggleOn
            }));
      }
```

```
      render() {
            return (
            <button onClick={this.handleClick}>
            {this.state.isToggleOn ? 'ON' : 'OFF'}
            </button>
            );
      }
}
```

The App component must render two instances of the <Toggle />.

```
import './App.css';
import Toggle from './Toggle';
import React, {Component} from 'react';

export default class App extends Component {
      constructor(props) {
      super(props);
      this.state = {
            flag : true
            }
      this.handleClick = this.handleClick.bind(this);
      }

      handleClick() {
            this.setState(prev => ({
            flag: !prev.flag
            }));
      }
      render() {
            return (
            <div>
            <Toggle value= {this.state.flag} />
            <Toggle value={!this.state.flag} />
            </div>
            );
      }
}
```

The execution result is shown to display two independent buttons. Clicking any button does not affect the other one.

Figure 8-1. Two Independent Buttons

8.3.2 Example of Using One Button

The execution of the second example is shown below.

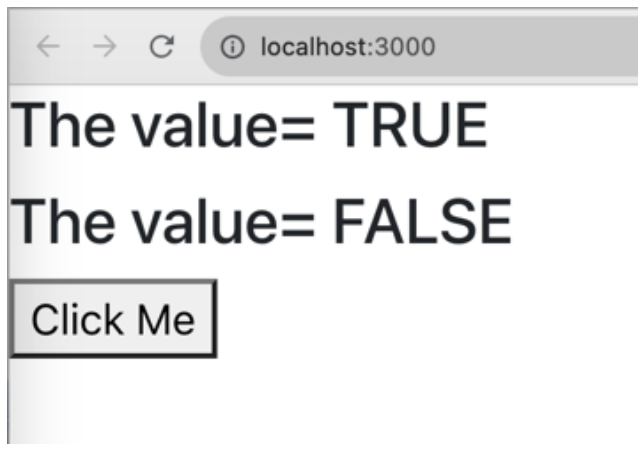


Figure 8-2. Single Button.

When the button “Click Me” is clicked, the first label changes from TRUE to FALSE and the second label changes from FALSE to TRUE. To achieve this, we need to change the App component and also the Toggle component.

```
import './App.css';
import Toggle from './Toggle';
import React, {Component} from 'react';

export default class App extends Component {
      constructor(props) {
            super(props);
            this.state = {
            flag : true
      }
      this.handleClick = this.handleClick.bind(this);
      }
      handleClick() {
            this.setState(prev => ({
            flag: !prev.flag
            }));
      }
      render() {
            return (
            <div>
                  <Toggle value={this.state.flag} />
                  <Toggle value={!this.state.flag} />
                  <button
                        onClick={this.handleClick}>Click Me
                  </button>
            </div>
            );
      }
}
```

In this example, the <Toggle/> component is changed to show a label instead of a button. The source code for Toggle is shown below.

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

export default class Toggle extends Component {

      render() {
            return (
            <h4>
            The value= {this.props.value ? 'TRUE': 'FALSE'}
            </h4>
            )
      }
}
```

8.4 Types of Synthetic Events and Attributes

What are the attributes available for synthetic events? As we learned before, we know that for an instance of <button>, we can use onClick as an attribute for specifying an event handler. We will refer to the official documentation for reactjs.org about synthetic events [9] and discuss the usage of only a few of them.

8.4.1 <Input>

The type of <input> is determined by the attribute type. We can refer to the website [10]. The possible types are listed at the associated webpage [11].

As an example, you can define the attributes for the <input> tag as follows:

```
render() {
      const id = this.state.formData.id;
      const flag = (id===1 || id===2 || id===3)? true: false;
      return <div className="m-2">
      <div className="form-group">
            <label>ID</label>
            <input className="form-control" name="id"
            disabled
            value={ this.state.formData.id }
            onChange={ this.handleChange } />
      </div>
      <div className="form-group">
            <label>Description</label>
            <input className="form-control" name="description"
            value={ this.state.formData.description }
            onChange={ this.handleChange } />
      </div>
```

```
      <div className="form-group">
            <label>Semester</label>
            <input className="form-control" name="semester"
            value={ this.state.formData.semester }
            onChange={ this.handleChange } />
      </div>
      <div className="form-group">
            <label>Prefix</label>
            <input className="form-control" name="prefix"
            disabled={flag}
            value={ this.state.formData.prefix }
            onChange={ this.handleChange } />
      </div>
      <div className="form-group">
            <label>Number</label>
            <input className="form-control" name="number"
            value={ this.state.formData.number }
            onChange={ this.handleChange } />
      </div>
      <div className="form-group">
            <label>Grade</label>
            <input className="form-control" name="grade"
            value={ this.state.formData.grade }
            onChange={ this.handleChange } />
      </div>
      <div className="text-center">
            <button className="btn btn-primary m-1"
            onClick={ this.handleClick }>
            Save
            </button>
            <button className="btn btn-secondary"
            onClick={ this.props.cancelCallback }>
            Cancel
            </button>
      </div>
</div>
}
```

Note:

1. If the value in the <input> field is changed, the <input> field with the *onChange* attribute will trigger a re-rendering of this component.
2. If we specify disable= “true” for a name, the input field becomes disabled and cannot be changed.

3. If we specify the *onChange* attribute to trigger the event handler, we do not need to add a "submit" button. We don't need to allow a use to enter data and click the "submit" button.

4. If we assign a local function to associate with the 'attribute' *onChange* with the HTML textbox input tag <input type='text' … />, React changes the value of the state variable each time the value in the input textbox is changed. For example, if the value *"100"* is entered one digit at a time, the *function* associated with the *onChange* will be invoked *three* times: once after the digit *'1'* is entered, the second time after the digit *'0'* is entered, and the third time after the last digit *'0'* is entered.

8.4.2 <Button>

The <button> tag also supports the event attributes in HTML with the change to the camelCase for the attribute [12]. A <button> tag must be associated with the attribute "onClick" to specify an event handler.

8.4.3 <Form>

The HTML <form> tag may require attributes including onSubmit. A list of event attributes can be found on the web page under Form Events [13].

8.5 Example - Display Courses With an Option

This example demonstrates how to define a state variable in the App component and change the value via an option box in the Search component. The course table will be displayed based on the value of the state variable a user chooses.

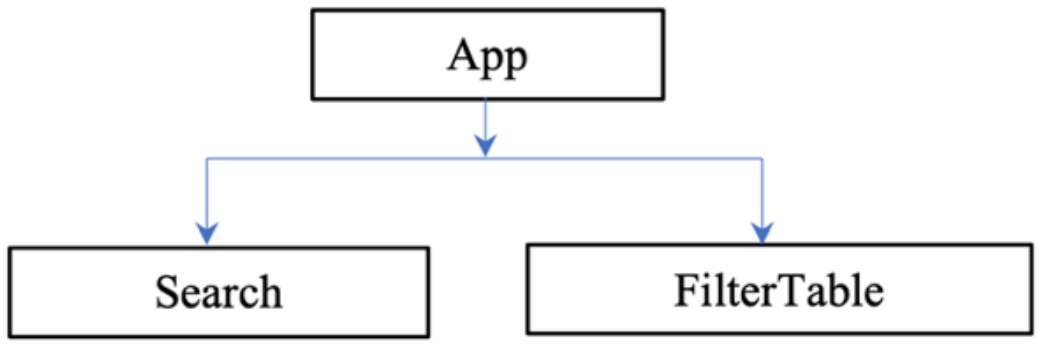


Figure 8-3. Example of Defining State.

The component hierarchy diagram (Figure 8-3) shows that the component App is defined as the parent for two children components: Search and FilterTable. The state variable "tookOnly" is defined in App to represent an option of only displaying courses that have been taken. We will need to change the data in the Search component. Once changed, the propagation down the component hierarchy to the FilterTable is not a problem because data flow in React is from top down. But how can we pass the option flag from Search back to App? We will need to pass an event handler from App down to Search via "props."

We need to define an event handler in FilterTable as below:

```
handleTookChange(tookOnly) {
      this.setState({
      tookOnly: tookOnly
      })
}
```

Then, we need to pass the name of the event handler to the Search component as below:

```
<Search tookOnly={this.state.tookOnly}
      onTookChange={this.handleTookChange} />
```

The complete code for App is shown next.

8.5.1 App.Js

```
import './App.css';
import React, {Component} from 'react';
import FilterTable from './FilterTable';
import Search from './Search';
import { COURSES } from './FilterTable';

export class App extends Component {
      constructor(props) {
            super(props);
            this.state = {
            tookOnly: false
      };
      this.handleTookChange = this.handleTookChange.bind(this);
      }

      handleTookChange(tookOnly) {
            this.setState({
            tookOnly: tookOnly
            })
      }

      render () {
            return (
            <div className="App">
            <div>
                  <Search tookOnly={this.state.tookOnly}
                  onTookChange={this.handleTookChange} />
            </div>
            <div>
                  <FilterTable courses={COURSES}
```

```
                  tookOnly={this.state.tookOnly}
            />
            </div>
            </div>
            );
      }
}

export default App;
```

8.5.2 Search.Js

The complete code for component Search is shown as follows:

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

//import Form from 'react-bootstrap/Form';

export default class Search extends Component {
      constructor(props) {
            super(props);
            this.handleTookChange = this.handleTookChange.bind(this);
      }
      handleTookChange(e) {
            this.props.onTookChange(e.target.checked);
      }

      render() {

            return (
            <form className='bg-primary p-2 m-4'>
            <input
            type="checkbox"
            checked={this.props.tookOnly}
            onChange={this.handleTookChange}
            />
            {' '}
            Only show courses took already
            </form>
            );
      }
}
```

8.5.3 FilterTable.js

```
import React, {Component} from 'react';
import CourseRow from './CourseRow';

export default class FilterTable extends Component {
 render () {
      const tookOnly = this.props.tookOnly;
      const rows = [];
      this.props.courses.forEach((course) => {
      if (tookOnly && !course.took) {
            return;
      }
      rows.push(
            <CourseRow
            course={course}
            key={course.name}
            />
       );
      });
      return (
      <table className="table table-striped table-bordered
      table-sm p-2 m-2">
      <thead className="bg-info text-white">
            <tr>
            <th>Name</th>
            <th>Desc</th>
            <th>Took</th>
            </tr>
      </thead>
      <tbody>{rows}</tbody>
      </table>
      );
 } // end of render()
} // end of component
export const COURSES = [
      {name:'CSC115', description:"Python", took:true},
      {name:'CSC141', description:"Java I", took:true},
      {name:'CSC142', description:"Java II", took:true},
      {name:'CSC240', description:"Java III", took:true},
      {name:'ART110', description:"Art Appreciation", took:false},
      {name:'ESS335', description:"Ecology", took:false},
      {name:'LIT300', description:"Literature", took:true},
      {name:'ENG368', description:"English", took:false}

];//end of COURSES
```

8.5.4 CourseTable

We still need to show the code for CourseTable.

```
import React, { Component } from "react";
import CourseTookRow from "./CourseTookRow";
import CourseRow from "./CourseRow";
export default class CourseTable extends Component {
 render() {
      const filterText = this.props.filterText;
      const tookOnly = this.props.tookOnly;
      const rows = [];
      let lastTook = null;
      let index = 0;
      this.props.courses.forEach((course) => {
      if (course.name.indexOf(filterText) === -1) {return;}
      if (tookOnly && !course.took) {return;}
      if (course.took !== lastTook) {
            index = index+1; // maintain the uniqueness of a key
            rows.push(
            <CourseTookRow
            took={course.took}
            key={index}
            />
      );
      }
      index=index+1; // main the uniqueness of the key
      rows.push(
            <CourseRow
            course={course}
            key={index}
            />
      );
      lastTook = course.took;
      });
      return (
      <div className="position-absolute bottom-50 end-50">
      <table className="bg-light m-4 striped bordered hover">
      <thead>
            <tr>
            <th>Name</th>
            <th>Description</th>
            <th>Took</th>
      </thead>
      <tbody>{rows}</tbody>
      </table>
```

```
      </div>
      );
 } // end of render()
} // end of component
```

8.5.5 CourseRow

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

export default class CourseRow extends Component {
 render() {
      const course = this.props.course;
      const name = course.took ? course.name :
            <span style={{color: 'red'}}>
            {course.name}
            </span>;
      const flag = course.took? "Took" : "Not Yet"

      return (
            <tr>
            <td>{name}</td>
            <td>{course.description}</td>
            <td>{flag}</td>
            </tr>
            );
 }//end of render()
}//end of component
```

8.5.6 CourseTookRow

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

export default class CourseTookRow extends Component {
 render() {
      const took = this.props.took;
      return (
            <tr>
            <th colSpan="2">
            {took}
            </th>
            </tr>
      );
 } // end of render
} // end of component
```

8.5.7 Execution Result

In the browser display screen, an option button with the label “Only show courses took already” is displayed at the first line. Below it, a table with three fields is displayed including the Name of a Course, the Description and the Took flag for each row. Some sample courses are displayed in the table as table rows. These rows are meant to be samples.

8.6 Review Questions

1. What is the difference between the name of an attribute in HTML and that in React?
2. What is the difference between the way to specify the event handler in HTML and in React?
3. Where can we find the list of attributes for the tag <input>, <button>, and <form>?

8.7 Additional Resources

1. Reactjs.org, “The Reference Guide for Synthetic Event,” Handling Event Guide, (Accessed in April 2024).

Chapter 9 Component Lifecycle and Reconciliation

In this chapter, we will talk about the following topics:

1. The lifecycle of a component,

2. The reconciliation, and

3. The lifecycle API.

9.1 What Is the Lifecycle of a Component?

The lifecycle of a component can be illustrated a diagram posted on GitHub by Wojciech Maj [14] and is related to the Interactive Version tweeted by Dan Abramov [15].

There are three stages in React for each component: Mounting, Updating, and Unmounting. The lifecycle indicates that React starts the cycle with the Mounting stage by executing the constructor, if any, and moves on to complete the render() activities. Then, React updates the DOM and refs which we didn't introduce yet, if needed. The first stage is concluded with executing the function *componentDidMount()*. This process continues for all components required to "paint" the main web page related to the root component and all child components. If there are other child components for each component, the Mounting stage for the root component is not completed.

As soon as any properties or state values are changed or a forced update occurs, React initiates an Updating stage. Similar activities occur internally to execute the render() function for the involved components and concludes with the execution of the function *componentDidUpdate()*.

There is an additional function before any node is removed from the DOM; the involved component will trigger the execution of a function *componentWillUnmount()*.

9.2 What Is Reconciliation?

When there are new changes that are related to a node in the DOM tree, the lifecycle of react triggers a re-rendering of all related child components in the virtual DOM tree. These changes include (1) new props, (2) any changes in the state, or (3) there are forced updates. The process of re-rendering all components is called "reconciliation." All changes will be propagated in the virtual DOM tree from the root node down to the bottom of the tree nodes. This process will take place asynchronously.

9.3 What Do We Need To Know About Lifecycle API Functions

These API methods are explained in detail on the official website of React [16]. We need to keep the concepts of component lifecycle and reconciliations in mind for at least two reasons:

1. Knowing the option of making changes at the last minute prior to the creating or updating of a component – We need to remember the option of modifying the content of a node after mounting or updating is about to conclude. In some cases, the extra manipulation of the content for a node in the DOM is needed. The only rule of thumb to remember is that when additional activities are needed, an internal documentation may be required for anyone to understand why these modifications must be done at the last minute of the mounting or updating of a component.

2. Understanding the execution trace of the web application – When we learn React, the most difficult concept to grasp is how the execution takes place and how React creates the virtual DOM. By adding a trace statement such as "console.log" in each lifecycle API method, we can learn the execution sequence of rendering components.

9.4 Example

We will use the example of displaying what a user enters via an HTML <input> tag to illustrate the Lifecycle API methods. We will need to add these methods to each component in this application.

9.4.1 App.Js

```
import './App.css';
import React, {Component} from 'react';
import TextMain from './TextMain';

export default class App extends Component {

     componentDidMount (prevProps, prevState, snapshot) {
          console.log("+ App - Inside ComponentDidMount");
     }

     componentDidUpdate(prevProps, prevState, snapshot) {
          console.log("+ App- Inside ComponentDidUpdate");
     }

     componentWillUnmount(prevProps, prevState, snapshot) {
          console.log("+ App - Inside ComponentWillUnmount");
     }
```

```
      render() {
            return (
            <div>
            <TextMain />
            </div>
            );
      }//End of render()

}//End of Component
```

9.4.2 TextInput

```
import React, {Component} from 'react';
export default class TextInput extends Component {
      constructor(props){
      super(props);
      this.handleFilterTextChange =
      this.handleFilterTextChange.bind(this);
      }

      handleFilterTextChange(e) {
      this.props.onFilterTextChange(e.target.value)
      }
      // Add the Lifecycle API function here

      render() {
            console.log("Inside TextInput");
            return (
            <form>
                  <input
                  type="text"
                  placeholder="Enter text..."
                  value={this.props.text}
                  onChange={this.handleFilterTextChange}
                  />
            </form>
            );//end of return()
      }//end of render()
}//end of component
```

9.4.3 TextDisplay

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

export default class TextDisplay extends Component {
      // Add the lifecycle API methods here
      render() {
            console.log("inside TextDisplay");
            console.log(this.props.text);
            return (
            <h4 className="sm-primary text-white text-center p-2
                  m-2">
            You just entered: {this.props.text}
            </h4>
            )
      }//end of render()
}//end of component
```

9.4.4 TextMain

```
import React from "react";
import TextInput from "./TextInput";
import TextDisplay from "./TextDisplay";
export default class TextMain extends React.Component {
constructor(props) {
      super(props);
      this.state = {
      filterText: ''
      };
      this.handleTextChange = this.handleTextChange.bind(this);
}

handleTextChange(text) {
      this.setState({
      filterText: text
      });
}
// Add the lifecycle API methods here

 render() {
      console.log("Inside main");
      return (
            <div>
            <TextInput
            text={this.state.filterText}
            onFilterTextChange={this.handleTextChange}
```

```
            />
            <TextDisplay
            text={this.state.filterText}
            />
            </div>
            );//end of return
 }//end of render()
}//end of component
```

9.4.5 Execution Result

The execution sequence of this App can be illustrated with the trace captured via the Google Chrome web browser.

The execution starts with the root node in which the App component is executed during the mounting phase. The mounting stage involves the mounting of the component TextMain. The mounting process in turn triggers the mounting stages of the component TextInput and TextDisplay. The process starts with the mounting of App but will not finish until all child components are mounted. Finally, the mounting process of App is completed.

9.5 Exercise

Find one application with controlled components and add the lifecycle API functions in all components in the App. Describe the execution sequence when the web app is executed.

9.6 Additional Resources

1. W3Schools, “React Lifecycle,” URL: https://www.w3schools.com/react/react_lifecycle.asp, accessed in July 2023.
2. Legacy.Reactjs.org, “State and Lifecycle,” URL: https://legacy.reactjs.org/docs/state-and-lifecycle.html, accessed in July 2023.

Part III Rendering of UI Components

In this part of this book, we begin to focus on rendering activities in the following chapters:

- Chapter 10 Conditional Rendering,
- Chapter 11 Lists, and
- Chapter 12 Forms.

Chapter 10 Conditional Rendering

In the following, we will go over several examples to illustrate the idea of conditional rendering. Some of these examples are inspired by the article "Conditional Rendering" posted on the website of React.js [8]. The major goal is to provide a feature for rendering web UI conditionally.

10.1 Example - if-Statement

We will use the example of displaying/editing a course table to illustrate two conditional rendering approaches: deciding if editing of displaying, and conditional pushing.

10.1.1 Decide if Editing or Displaying

In the example of displaying a course table, we may want to provide the capability of editing or displaying the course information using two different components. There is a state property showEditor defined in a class component and the render() is shown as follows:

```
render() {
if (this.state.showEditor) {
      return <CourseEditor
            key={ this.state.selectedCourse.id || -1 }
            course={ this.state.selectedCourse }
            saveCallback={ this.saveCourse }
            cancelCallback={ this.cancelEditing } />
      } else {
      console.log("-- in CourseDisplay --");
      return <div className="m-2">
            <CourseTable
            courses={ this.props.courses }
            credit = { this.props.credit }
            editCallback={ this.startEditing }
            deleteCallback={ this.props.deleteCallback } />
            </div>
      }//end of if-statement
}//end of render()
```

10.1.2 Example: Conditional Push

For the application of displaying course tables, we use the technique of *conditional push* frequently.

```
import React, { Component } from "react";
import CourseTookRow from "./CourseTookRow";
import CourseRow from "./CourseRow";
```

```
export default class CourseTable extends Component {
render() {
const filterText = this.props.filterText;
const tookOnly = this.props.tookOnly;
const rows = [];
let lastTook = null;
let index = 0;
this.props.courses.forEach((course) => {
      if (course.name.indexOf(filterText) === -1) {return;}
      if (tookOnly && !course.took) {return;}
      if (course.took !== lastTook) {
            index = index+1; // maintain the uniqueness of a key
            rows.push(
            <CourseTookRow
            took={course.took}
            key={index}
            />
            );
      }
      index=index+1;
      rows.push(
            <CourseRow
            course={course}
            key={index}
            />
      );
      lastTook = course.took;
});

return (
      <table className="m-4 striped bordered hover">
      <thead>
            <tr>
            <th>Name</th>
            <th>Description</th>
            <th>Took</th>
            </tr>
      </thead>
      <tbody>{rows}</tbody>
      </table>
      ); // end of return()
      }//end of render()
}//end of component
```

10.1.3 Execution Result

The execution result of the conditional push can be illustrated by entering 41 as the total transfer credits. The Writing Emphasis table should display two lines.

Line 1: The label “Student Name” is followed by a text field.

Line2: The label “Total Transfer Credits” is followed by a text field.

A table “Writing Emphasis for John” is followed by what the user enters in the *Student Name* textbox.

The “Table Heading” includes the following fields: ID, Description, Semester, Prefix, Number, Grade, Editing

Each row displays the data below the headings. Each row includes one *Edit* button at the end of each row. A user can push the edit button to edit the data in each row using a new screen.

10.2 Example Posted on Reactjs.Org - Logical AND With an HTML Tag

10.2.1 Mailbox

```
export default function Mailbox(props) {
      const unreadMessages = props.unreadMessages;
      return (
            <div>
            <h1>Hello!</h1>
            {
                  unreadMessages.length > 0 &&
                  <h2>
                  You have {unreadMessages.length} unread messages.
                  </h2>
            }
            </div>
      );
}
```

This example demonstrates the potential power of JSX. We can define a Boolean variable based on the value of a property and use the value inside a render() statement or a return() statement.

10.2.2 Execution Result

The browser displays a simple web page with two lines of centered text. The first line displays “Hello!” And the second line displays “You have 3 messages.” In a smaller font. This example is used to illustrate the concept of using a “logical AND with an HTML tab.”

10.3 Conditional Styling

Conditional styling of a column in a table is an interesting technique. In this example, we use this approach to determine the font color for the course name in a course row.

10.3.1 CourseRow.js

```
import React, {Component} from 'react';
export default class CourseRow extends Component {
      render() {
      const course = this.props.course;
      const name = course.took ? course.name :
      <span style={{color: 'red'}}>
      {course.name}
      </span>;

      return (
            <React.Fragment>
            <tr >
            <td>{course.category}</td>
            <td>{name}</td>
            <td>{course.desc}</td>
            <td>
            <input type='checkbox' checked={ course.took }
            onChange= { () => this.props.toggleTookChange(course) }
            />
            </td>
            </tr>

            </React.Fragment>

            );
      }//end of render()
}//end of component
```

10.3.2 Execution Result

The execution result shows the outcome of Conditional Styling. A browser displays the following:

1. A textfield for a user to enter a search phrase, e.g., a course name such as CSC or CST.

2. A course table displaying all courses that have not been taken. Each row contains the Category (Major or Minor), Course Name (Course ID), Description (title of the course), and the Took flag. For each row there is a Took Flag. If it is not checked, this course shows up in the top table. If it is checked, this course will be moved down to the bottom table.

3. A second course table displaying all courses that have been taken. For each row there is a Took Flag that have been checked. If it is unchecked, this course will be moved up to the top table.

4. A button for adding a new course if the button is pressed.

Chapter 11 Lists

In this chapter, we will talk about the following topics:

1. JavaScript for accessing an array of numbers,

2. Example of displaying a list of numbers, and

3. Example of displaying a list of courses in a table.

11.1 Displaying a List of Numbers

First, we explain how to use JavaScript to display a list of numbers using the following example.

```
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) =>
            console.log(number)
      );
```

The *numbers.map()* function receives an arrow function as its argument. The elements in the array numbers will be given one element at a time to the arrow function and returns an element as a list item. The final listItems can be used in a rendering process as part of an element.

11.2 NumberList

Then, we present a complete React project to generate a list of numbers. We define a NumberList component as a class component and use a root component App to display the list of numbers. The NumberList component is enclosed the *return statement* within App.js.

```
import './App.css';
import NumberList from './NumberList';

function App() {
      const number = [1,2,3,4,5];
      return (
            <div>
            <header>
            <title>Learn list</title>
            </header>
            <NumberList numberArray={number} />
            </div>
      );
}
export default App;
```

The NumberList component is defined as a class component.

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

export default class NumberList extends Component{
            render() {
      return (
      <div>
            The number list is
            {this.props.numberArray.map(n =>
                  <div key={n.toString()}>{n}</div>)
            }

            The reverse is
            {this.props.numberArray.reverse().map(n=>
                  <div key={n.toString()}>{n}</div>)
            }
      </div>
      ) //end of return()
            } //end of render
} // end of component
```

Notice that the arrow function in NumberList requires an attribute key as follows:

```
<div key={n.toString()}>{n}</div>
```

The keyword *key* is used internally by React to identify each item in a list uniquely such as this.props.numberArray. As a rule, using a list in React, we need to provide the attribute with a unique data to distinguish the list item uniquely. Otherwise, we will receive a warning message.

11.3 Define a Table Component

In this next example, we define a *BookTable* component containing rows of individual book information in each row defined as a *BookTableRow* component. We show the App.js component first followed by the BookTable component and the BookTableRow component.

```
import './App.css';
import BookTable from './BookTable';
import React from 'react'

function App() {
      return (
            <BookTable />
      );
}
export default App;
```

The component BookTable component is defined next.

```
import React, { Component } from "react";
import BookTableRow from "./BookTableRow";

export default class BookTable extends Component {

 render() {
      const books = [
            {id : 1, title: 'True Crime', price: 15.95 },
            {id : 2, title: 'Fake Crime', price: 15.95 },
            {id : 3, title: 'Make Believe', price: 15.95 },
            {id : 4, title: 'No Nonsense', price: 15.95 },
      ]

      const rows = [];

      books.forEach((book) => {
            rows.push(<BookTableRow book={book} key={book.id} />);
      });

      return (
      <table className="table table-sm table-striped table-bordered">
      <thead className="bg-info text-white text-center m-2 p-2">
            <tr colSpan="3">
            <th>ID</th><th>Title</th><th>Price</th>
            </tr>
      </thead>
      <tbody colSpan="3">
            {rows}
      </tbody>
      </table>
      );// end of return()
 } // end of render()
} // end of component
```

In the component BookTable, a child component BookTableRow is used to define one row of the BookTable.

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

export default class BookTableRow extends Component {
 render() {
            const book = this.props.book;
            return (
      <tr>
      <td className="text-center">{book.id}</td>
      <td className="text-center">{book.title}</td>
      <td className="text-center">${Number(book.price).toFixed(1)}
      </td>
      </tr>
            );
 } //end of render()
} // end of component
```

11.4 Exercises

Add two lists to the App.js: one displays characters ‘a’, ‘b’, ‘c’, and ‘d’, and the other displays the same array in reverse order.

Chapter 12 Forms

In this chapter, we will talk about the following topics:

1. What is a React Form?

2. Examples of React Forms

12.1 What Is a React Form?

In general, a form is a web page with user input fields. In React, we need to learn how to define event handling functions to process the values a user enters; hence, the topics of form handling and state are revisited.

React Form Handling and HTML Form Handling are different. An HTML form must be defined by the value for the method and the action attributes to specify what HTTP method (such as post or get) and action (the route) are used. A form input will be sent from the front-end browser to the backend server. The server will check the action and the method to trigger the correct controller function. The processing of the data a user enters is done at the backend server.

React forms are handled by React at the frontend for an SPA. In the component the form input is enclosed, and some event handling functions must be defined. When a user enters data, or clicks a button, these event handling functions will be triggered at the front-end machine without the interference of a backend server. Learning what event handling functions are needed for what type of tags is important. For example, if we define an <input> tag without providing an attribute onChange, this input field becomes a read-only input, and we will receive a warning at run-time. We need to specify which event processing functions will be associated with a button, which event function will be triggered when the content of a text box is changed, and so forth.

12.2 Examples of React Forms

We will use the examples posted at the Official Reactjs Website to illustrate the concept of Controlled Components. (Note: The webpage is a legacy version and will not be maintained.)

In HTML, the user input entered via a form is handled by the browser. When the form data are submitted, the browser formats the data and sends an HTTP Post request to the server. The server will trigger the Controller to process the data.

In React, the input data entering via a form is handled by a component in which the form is enclosed as a child component of the form component. In React, components such as <input>, <textarea>, or <select> may need to maintain the state data in the components in which these tags are rendered. React must handle event such as button pressed, or text field data changed. Thus, once the <form> is submitted, the React lifecycle process will take place. A React component rendering React elements with which event handling functions are

associated is called a *Controlled Component*. The input data may be “handled” with an event handling for which the event handling function will be triggered by the props values as onChange or onSubmit.

12.2.1 Example of Controlled Components With a Text Field and a Button

As an example, we will use a *NamedForm* component to define a form with a text field for a user to enter the user’s name. We may use <input type=’text’> to achieve the effect. If we need to define a button with which the value a user enters may be submitted, we may use <input type=’submit’> to achieve the effect. After the form is submitted, React triggers the lifecycle methods to display a pop-up alert window showing a simple text such as “A name was submitted: <name>.” Here is the NamedForm Component:

```
import React, {Component} from 'react';
export default class NamedForm extends Component {
      constructor(props) {
            super(props);
            this.state = {value: ''};
            this.handleChange = this.handleChange.bind(this);
            this.handleSubmit = this.handleSubmit.bind(this);
      }

      handleChange(event) {
            this.setState({value: event.target.value});
      }

      handleSubmit(event) {
            alert('A name was submitted: ' + this.state.value);
            event.preventDefault();
      }

      render() {
      return (
            <form onSubmit={this.handleSubmit}>
                  <label>
                  Name:
                  <input type="text" value={this.state.value
                        onChange={this.handleChange} />
                  </label>
                  <input type="submit" value="Submit" />
            </form>
            );
      } // end of render()
}// end of component
```

As the user begins typing in a name, the value handleChange will trigger the lifecycle to re-render the NamedForm component each time the user enters an additional letter.

12.2.2 Execution Result

The browser displays a form with two lines. On the first line the following text box is displayed in the center of the row:

Name: a text field with the names “Jack” displaying gin the textbox.

In the second line, a Submit button is shown in the center of the row.

Upon hitting the submit button, an Alert Window pops out with the text “localhost:3000 says” on the first line and “A name was submitted: Jack” on the second line. An button is displayed at the bottom right of the warning box. Clicking the OK button makes the warning box to disappear.

12.2.3 An A Dropdown Box Example of Using <Select>

In this next example, we demonstrate the use of a dropdown box and the use of a <select> HTML tag for holding the option a user chooses.

12.2.4 App.Js

For the simplicity, we keep the main component in App.js and define the state also in App.js. In App.js, the return statement encloses a <form> in which a <select> tag is used.

```
import MyDisplay from './MyDisplay'
import { useState } from 'react';

export default function App() {
 var [value, setValue] = useState('1');

return (
<>
<div >
      <form className="w-25 p-1" >
      <div className="form-group p-1">
      <label forhtml="selectOption ">Select Option Number</label>
            <select
                  className="form-control"
                  id="exampleFormControlSelect1"
                  onChange={(e) => setValue(e.target.value)} >
                        <option>1</option>
                        <option>2</option>
                        <option>3</option>
                        <option>4</option>
```

```
                        <option>5</option>
            </select>
      </div>
      </form>
</div>
<MyDisplay number={value}/>
</>
) // end of return();
} // end of component
```

12.2.5 MyDisplay

```
export default function MyDisplay ({number}) {
console.log("inside MyDisplay");
      //console.log(this.props.text);
      return (
      <h4 className="bg-primary text-white text-center p-2 m-2">
            You just select the number: {number}
      </h4>
      )

}
```

12.2.6 Execution Result

The execution result (Figure 12-1) displays the following:

1. “Select Option Number” - displayed as a label
2. A textbox showing a default option 1
3. A text line showing “You just select the number: 1”

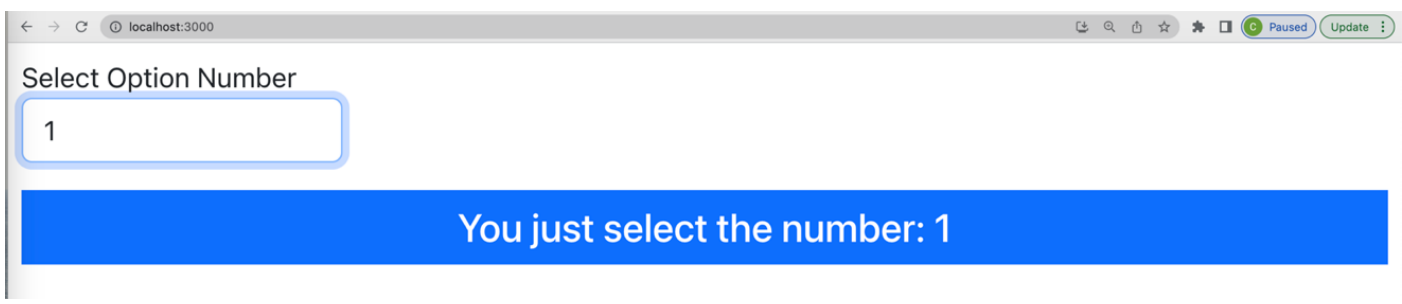


Figure 12-1. The <Select> Example.

If a user changes the option, the chosen option will be displayed in the text line. When the cursor is hovered over the selected field, a dropdown menu shows up. It is not shown in this figure. Notice that the number a user selects will be immediately displayed below dynamically.

12.3 Examples of an Uncontrolled Form Component

When we discussed the concept of Controlled Components, one may wonder if there are uncontrolled forms. A React uncontrolled component may be defined with a reference to associate with a DOM tree node using the useRef() hook directly. The use of a reference to a DOM node is not recommended for beginners in most cases if not needed. Hence, the concept will be illustrated next with examples only for your reference. But you may skip this section without disrupting your learning experience.

12.3.1 Input With useRef()

The following example illustrates the concept of using a reference to store data in lieu of state. The difference is that using ref to store data will not trigger the Update Phase in the life cycle of a component unlike when the state value is changed. We will use an example to illustrate the concept next.

We will create a ref using the hook useRef() first. Then, we can define an <input> button with an onClick event handler and the value stored in the ref will be used in an HTML alert popup window.

```
import { useRef } from "react";
const UncontrolledForm = () => {
      let ref = useRef(0);
      var number = 0;

      var handleClick = (e) => {
            ref.current = ref.current + 1;
            number = number + 1;
            alert('You click the submit button ' + ref.current + '
            times');
      };

      return (
            <>
            <h1>Uncontrolled Form</h1>
            <input type="button" value="Click Me"
            onClick={handleClick} />
            </>
      );
};

export default UncontrolledForm;
```

As we mentioned previously, a controlled component keeps track of data using props and state, an uncontrolled component uses references. In this example, the number of times the button is clicked will be stored in the reference. The current value of the reference can be referred to by using the predefined

property '*ref.current*' in our example. Although we may use the uncontrolled component to access a node in a DOM tree directly, we cannot access the node with {ref} inside the render() or return() function because a reference node will not be involved in the rendering process. In other words, we cannot add the following in our render() or return():

```
<input type='submit' value={ref}/>
```

or

```
<MyDisplay text={ref.current} />
```

12.3.2 Example of Forward Referencing

In some cases, if we really need to use references to store data and pass the value down to the children nodes, we may use a state variable and a reference as a hybrid paradigm known as a forwarding reference to achieve the result.

The App.js code is used to define a reference and a state. The state value will be used to share the data with other components and the reference will be used to store the value.

```
import "./styles.css";
import MyInput from './MyInput';
import MyDisplay from './MyDisplay';
import { useRef, useState } from 'react';

export default function App() {
      const inputRef = useRef(null);
      const [value, setValue] = useState('');
      var handleChange = (e) => {
      setValue(e.target.value)
      }

      return (
            <>Please enter a course title in Computer Science
            <br />
            <MyInput ref={inputRef}
                  onChange={handleChange}
                  value={value} />
            <MyDisplay text={value} />
            </>
      )
}
```

12.4 Exercises

1. Create an application to include a text field for a user to enter any text. Your application will display what the user enters.

2. Change the application to include a button that enables what the text user enters to be displayed after the button is clicked.

Part IV Design Issues With ReactJS

In this section, we focus on design issues for developing React projects with the software engineering principles to achieve modularity, reusability, and maintainability.

- Chapter 13 covers the topic of lifting state up,
- Chapter 14 covers the idea of “Think in React” at the design phase and development phase with software engineering principles,
- Chapter 15 talks about React Routing, and
- Chapter 16 covers React Redux State Management with the Redux Toolkit and hooks.

Chapter 13 Lifting Up State

The introduction of the component state drastically complicates the design of React applications drastically for several reasons. First, the data flow from one component to another can only be one way via props from the upper hierarchical level of the DOM tree node downward to other nodes below unless extra event handler is passed down from the upper-level node down to the child node. The location where the state resides is a critical issue for the design of React components. Without using additional state management tools, the fundamental way to share state among parent and child components or sibling components is to lift up the state and define the state in the common ancestor of the components that need to share the state.

In this chapter, we will talk about the following topics: lifting up state, avoiding recursive re-rendering, and debugging with examples.

13.1 An Example of Lifting Up State Data

We used an example previously to illustrate the idea of how to develop controlled components. Now, we will use this example to illustrate two additional concepts: (1) Lifting up state, and (2) Avoiding recursive rendering.

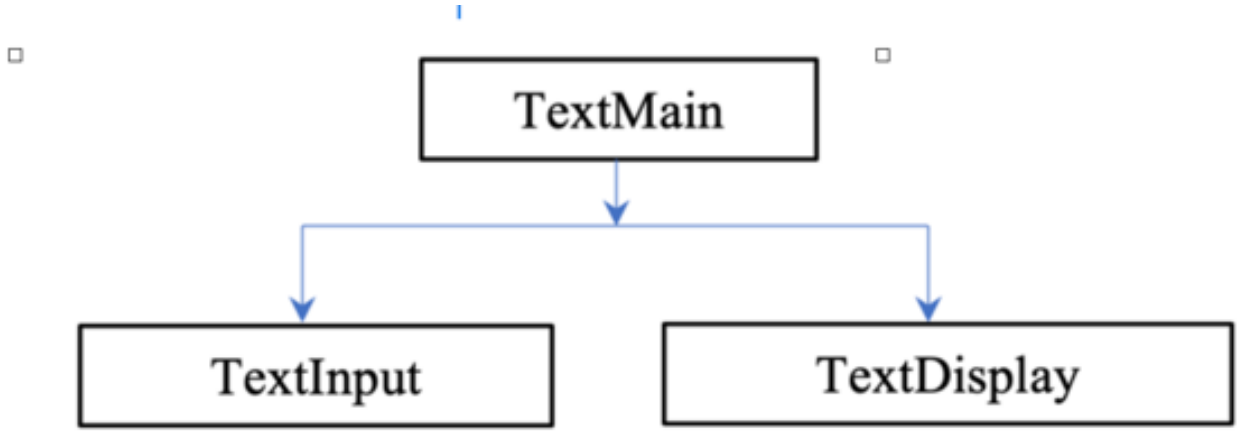


Figure 13-1. Lift (Up) State

13.1.1 Define State in TextMain

In this example (Figure 13-1) , we define a text field in TextInput. A user can enter text data via the text field. Then, the TextDisplay component will display the text data on the screen. When we define the input component and the display component in two separate components, we may reuse each of them independently. We call this “component splitting” or “component decoupling.” However, if we use only what we have learned so far, there is an issue. We cannot pass data from one component to another sibling. The solution is to define a TextMain component as the “common” ancestor of the two sibling components. We explain how we should define the TextInput, TextDisplay, and the TextMain components.

Starting from the TextInput, we need to allow a user to enter the text data that the component TextDisplay uses to display on the screen. But these two components are siblings. They cannot share any state data. The sharable data must be stored in the “common ancestor” that is the TextMain component. In the TextInput component, we must specify (1) the state data as a placeholder for storing the data user enters later, and (2) the name of the event hander both received from TextMain as props. When a user enters the text data, the data will be passed back to the TextMain. The state of TextMain will be changed. As a result, the children nodes are re-rendered whenever the state of a parent node is mutated. This state change triggers the re-rendering of both children. Whenever a character is entered in the text field, the reconciliation occurs from top down.

With this approach, we lift the state data from the TextInput up to the TextMain and is, hence, referred to as “lifting up state.” In the following, we show the code for TextInput first, the TextDisplay, next, and then the code for TextMain.

```
import React from 'react';
export default function TextInput ({text, onFilterTextChange}) {

      const handleFilterTextChange = (e) => {
            onFilterTextChange(e.target.value)
      }

      console.log("Inside TextInput");
      return (
            <form>
                  <input
                  type="text"
                  placeholder="Enter text..."
                  value={text}
                  onChange={handleFilterTextChange}
                  />
            </form>
            );
}
```

The second component is TextDisplay, that is a sibling of the component TextInput because both of them are used in the TextMain component. They both need to use the value the user enters with the name of the prop text. We will define the state inside TextMain. The component TextDisplay is illustrated below.

```
import React from 'react';

export default function TextDisplay({ text }) {
      console.log(`inside TextDisplay -- pass--${text} `);
      return (
      <h2 className="bg-primary text-white text-center p-2 m-2">
            You just entered: {text}
```

```
            </h2>
      ) // end of return
}
```

Finally, we define the component TextMain.

```
import React, {useState} from "react";
import TextInput from "./TextInput";
import TextDisplay from "./TextDisplay";

export default function TextMain () {
      const [myText, setText] = useState('');
      const handleTextChange = (text) => {setText(text.slice());}
      console.log("Inside main -- pass 1 --" + myText);
      return (
            <div>
            <TextInput
            text={myText}
            onFilterTextChange={handleTextChange}
            />
            <TextDisplay
            text={myText}

            />
            </div>
      );
}
```

13.1.2 Avoiding Recursive Rendering

The “onChange” attribute associated with the <input> tag may become a potential pitfall. Many beginners define the TextInput component incorrectly as follows.

```
<TextInput
      text={myText}
      onFilterTextChange={handleTextChange()}
/>
```

Let’s use this example to illustrate why we cannot specify the value for the onFilterTextChange as handleTextChange() instead of specifying handleTextChange without the parentheses. Adding the two parentheses is telling React to “call” the event handler. Without the parentheses, we are telling React the name of the event handler. Semantically, these two values trigger dramatically different operations. The value for the attribute onChange must be a name of the event handler instead of a “call” to the event handler. If we do, this rendering of the component triggers the handleFilterTextChange() function to be executed immediately. When that happens, the TextMain would be re-rendered,

that in turn triggers the re-rendering of the two children components, known as the *recursive rendering*. When the TextInput is re-rendered, the onChange attribute immediately triggers the execution of the event handler again. Hence, a recursive re-rendering occurs. If you remove the '()', the project should work.

13.2 Debugging

If your code causes the recursive rendering, a browser will eventually terminate the execution and displays the error message. For Google Chrome, the error message can be found at the Console of the Google Chrome Developer's Tool.

The error message indicates that a repetitive error "Cannot read properties of undefined (reading 'target') occurred at TextInput.handleFilterTextChange. The screenshot only shows several occurrences of the error messages. Then, the reconciliation process stops with the recommendation of considering adding an error boundary.

The core of the error indicates that *Inside <TextInput>, a function handleFilterTextChange is called repeatedly until the browser quits*. It was caused by the '()' inside the value passed to the attribute *onChange*. If you remove the '()' and restart the project, the project works.

13.2.1 Another Example

The legacy official website for Reactjs.org provides an instructive example to illustrate the concept of Lifting State Up. This example presents a Calculator component to calculate the temperature transformation from the scale of Celsius to Fahrenheit or vice vera. The Calculator component will need to use the TemperatureInput component and the BoilingVerdict component to do the job. The state data the temperature and the scale (Celsius or Fahrenheit)are required in both the TemperatureInput and the BoilingVerdict components. Where should we define the state variables? We will lift the state from the TemperatureInput component to the Calculator since the temperature will be entered from the TemperatureInput component and then used in the Calculator and the Calculator is the parent of the TemperatureInput. We always find the nearest common ancestor as the component to define the state variables, in this case, inside the Calculator component. The structure of the DOM looks like the following:

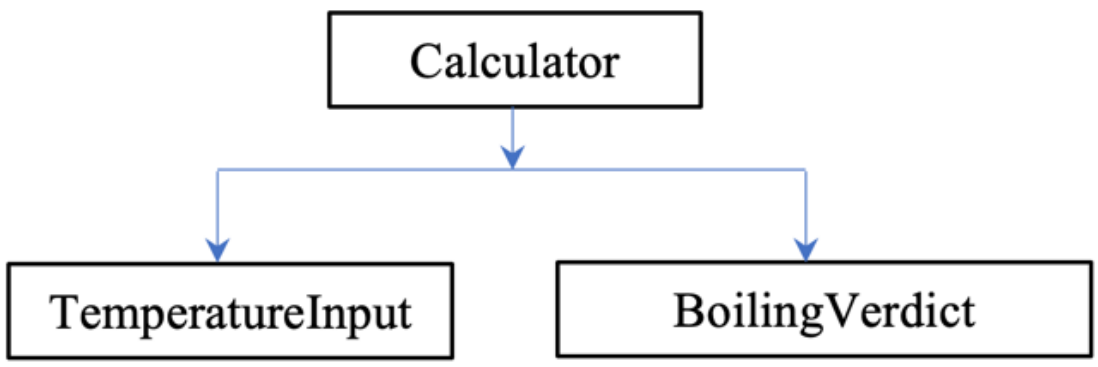


Figure 13-2 Lift (Up) State.

Then, there is another issue. When data flow always goes from top down, there is no problem passing the state variable values from the Calculator component to the BoilingVerdict component. But how can we pass the user input values from TemperatureInput component back to the Calculator component?

The answer to this question is that we need to define a callback function inside the Calculator and pass the callback function to the TemperatureInput component. We begin our design for each component individually and then determine where the state variables will be stored eventually.

First, the TemperatureInput component looks like the following:

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

export default class TemperatureInput extends Component {
      constructor(props) {
            super(props);
            this.handleChange = this.handleChange.bind(this);
      }

      handleChange(e) {
            this.props.onTemperatureChange(e.target.value);
      }

      render() {

            const scaleNames = {
            c: 'Celsius',
            f: 'Fahrenheit'
            };

            const temperature = this.props.temperature;
            const scale = this.props.scale;
            return (
            <fieldset>
            <legend>Enter temperature in {scaleNames[scale]}:</
      legend>
            <input value={temperature}
            onChange={this.handleChange} />
            </fieldset>
            );
      } //end of render()
} end of component
```

Then, the BoilingVerdict component is defined as the following:

```
export default function BoilingVerdict({celsius}) {
      if (celsius >= 100) {
            return <p>The water would boil.</p>;
      }
      return <p>The water would not boil.</p>;
}
```

The component Calculator is then defined below.

```
import React, {Component} from 'react'
import TemperatureInput from './TemperatureInput';
import BoilingVerdict from './BoilingVerdict';
import tryConvert from './tryConvert';
import toCelsius from './toCelsius';
import toFahrenheit from './toFahrenheit';

export default class Calculator extends Component {
constructor(props) {
      super(props);
      this.handleCelsiusChange = this.handleCelsiusChange.bind(this);
      this.handleFahrenheitChange =
      this.handleFahrenheitChange.bind(this);
      this.state = {temperature: '', scale: 'c'}; // Defined states
      }

handleCelsiusChange(temperature) {
      this.setState({scale: 'c', temperature});// USE setState
}

handleFahrenheitChange(temperature) {
      this.setState({scale: 'f', temperature}); // USE setState
}

render() {
      const scale = this.state.scale;
      const temperature = this.state.temperature;
      const celsius = scale === 'f' ?
      tryConvert(temperature, toCelsius) : temperature;
      const fahrenheit = scale === 'c' ? tryConvert(temperature,
      toFahrenheit) : temperature;
      return (
      <div>
            <TemperatureInput
            scale="c"
            temperature={celsius}
```

```
            onTemperatureChange={this.handleCelsiusChange} />
            <TemperatureInput
            scale="f"
            temperature={fahrenheit}
            onTemperatureChange={this.handleFahrenheitChange} />
            <BoilingVerdict
            celsius={parseFloat(celsius)} />
      </div>
            ); // end of return
      } // end of render
} // end of component
```

13.2.2 Do Not Change the State Directly

In the previous example, the Calculator component defines the state inside the constructor of a class component. The state should be changed by calling the function setState() instead of using an assignment statement to change the state directly. Otherwise, the reconciliation, i.e., the lifecycle process of re-rendering all affected components, will not take place.

Incorrect: this.state.temperature = 100;

Incorrect: this.state.scale='f';

Correct: this.setState({scale: 'c', temperature}; // temperature is a parameter.

We should not change the state variables directly using "this.state.temperature=100." The reason is because if we do, we bypassed the lifecycle methods totally. React will not be "notified" that the state values are changed. The rippling-effect of making asynchronous changes to all related components will not take place. If you do so, the change of the temperature will not be propagated down the DOM tree to the children components.

13.2.3 Execution Result

When the app is executed and a browser is brought up to display the result, we will see the screen with three parts.

The first part includes the centered text "Enter temperature in Celsius" and a textfield that allows a user to enter the temperature in celsius.

The second part includes the centered text "Enter temperature in Fahrenheit" and a textbox that allows a user to enter the temperature in Fahrenheit.

Finally, at the bottom of the screen, a text message is displayed to show if the water would boil or not based on the temperature the user entered.

13.3 Define State Variables With Hooks in Function Components

Starting with React Version 16, we can define state variables using hooks. We will use a simple example for which a user is prompted for one line of text using a text field. Then, the text will be displayed below the text field.

13.3.1 TextMain

```
import React, {useState} from "react";
import TextInput from "./TextInput";
import TextDisplay from "./TextDisplay";
export default function TextMain () {
      const [myText, setText] = useState('');
      const handleTextChange = (text) => {setText(text.slice());}
      console.log("Inside main" + myText);
      return (
      <div>
            <TextInput
            text={myText}
            onFilterTextChange={handleTextChange}
            />
            <TextDisplay
            text={myText}

            />
      </div>
      ); end of return()
} //end of component
```

13.3.2 Execution Result

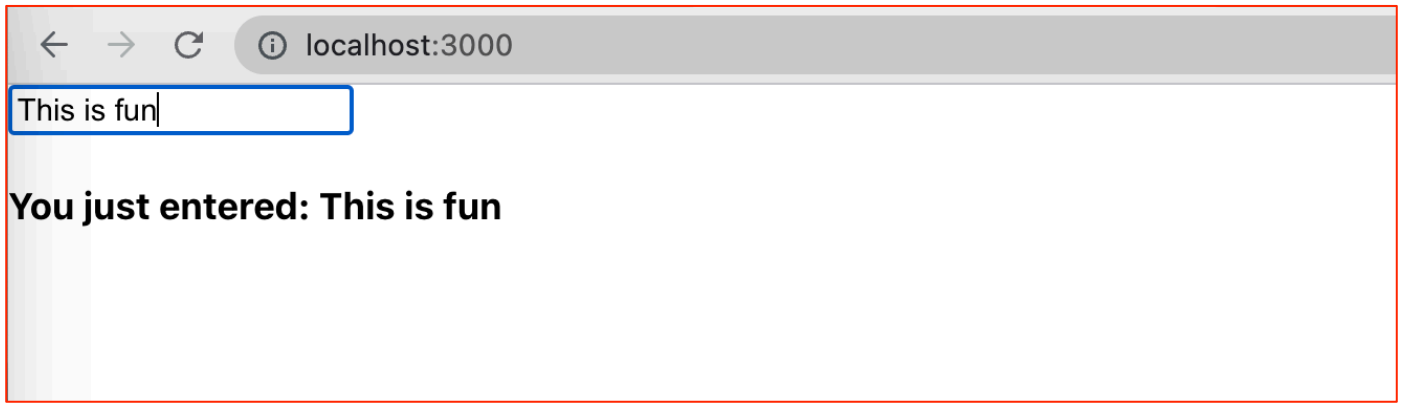


Figure 13-3. Define State Variables with Hooks.

There are two lines of objects when the web browser displays the webpage. Line 1 displays a textfield that is used to prompt a user for one line of text, e.g., “This is fun.”

Line 2 displays the text “You just entered: This is fun” were “This is fun” is what the user enters.

13.4 Exercise

You may choose to work on this exercise now or wait until you read the next topic – Think in React.

Define a component *TextInput* in which it contains an input textbox; and define another component *TextDisplay* as a heading with the value showing what the user enters. Notice that there is no “submit’ button. You need to use a tag as follows:

<input type= “text” onChange= “….”>. You may need to define a file *main.js* in which the state of the text will be stored and shared between the two components.

Chapter 14 Think in React

Although we introduced React fundamentals comprehensively, it is still essential to design and develop web applications in compliance with software engineering principles. Right now, we should be able to apply the skills to the design and development of a medium-size web application. But, as the size and the complexity of an application may scale up in the future, we should strive to apply principles of maintainability, reusability, modularity, and readability to the development of quality applications. We briefly explain what these principles are and what they meant below:

1. Maintainability - Maintain a "single source of truth" to reduce the cost of software maintenance,
2. Reusability - Do not "copy and paste" any part of the component and hope you may simplify the development,
3. Modularity – Define one component in one file as a basic build block for your web UI, and
4. Readability - Document and explain the semantics of the properties and state for each component. Check the properties carefully in the parent component to ensure that the names defined in the child components are consistent with the attributes used.
5. Scalability – Consider the design with a possible future growth of the data size. For example, the number of rows in a table, the number of data fields for a course, etc.

In this chapter, we will illustrate how to think in React by implementing *Conditional Rendering*. Also, we will demonstrate how to apply software engineering principles to the development of the project for achieving reusability, maintainability, and scalability. In this chapter, we will demonstrate how to add a Writing Emphasis Table to an advising sheet. Moreover, we will leave an exercise to add a Speaking Emphasis Table to the readers.

14.1 How Do We Start?

As the degree requirement in a university, there may be several categories of courses needed to graduate. As an example, the requirements may include the following areas of courses: Academic Foundations, Distributive, General Education Requirements, and Major Requirements. We will focus on the implementation of the General Education Requirements. Under the category of General Education Requirement courses, there are two similar groups of courses: Writing Emphasis courses and Speaking Emphasis courses. We will demonstrate how to implement the Writing Emphasis Table under the General Education Requirement in this chapter. We begin with a Requirement Analysis phase, followed by the Design phase , and complete with the Development phase. We may refer to the requirement for Writing Emphasis and Speaking Emphasis

courses as Additional Requirement or Gen Ed Requirement interchangeably. The implementation of Speaking Emphasis table will be left as an exercise to the readers.

14.1.1 Application Requirements

We must understand the rules of Writing Emphasis first. As an example, we will impose the following rules. For students with transfer credits, the following rules are applied:

1. If transfer credits < 40, three courses are required,
2. If transfer credits < 71, two courses are required, and
3. If transfer credits >= 71, only one of the two courses listed in the advising sheet is required.

We must define a system for a user to do the following:

1. Allow a user to enter the student’s name,
2. Allow a user to enter the total number of transfer credits, and
3. Allow a user to enter the courses under the Writing Emphasis & the Speaking Emphasis requirements.

Obviously, we may use “Conditional Rendering” to complete the project.

14.1.2 Conditional Rendering

Some may define an array with three elements and hide part of the array depending on the total number of transfer credits. There is a flaw in this approach. By doing so, you assume that the maximum number of courses you need is three courses. While this is true for this special case, it will not be sound if the maximum number of courses is changed later.

What we suggest is to define an empty array of a table row for each course and then increase the number of courses in the array by checking the total number of transfer credit. This approach makes no assumption that the maximum number of courses is three.

14.2 Design Phase

We will take the top-down approach for the design of the project. First, we check the requirement and decide what we need to implement in the first Writing Emphasis table. Based on the requirement, we can determine how many components are needed, what each component encapsulates, and where state data should be eventually stored.

The next step is to decide how many components are needed by checking the final table. While we could define one App component to complete the whole activities of adding the two text boxes, insert one Writing Emphasis table to

complete the first stage, this is not a scalable design. We notice that we need to define two similar tables for the same student and transfer credits. All table rows are similar with the same types of data fields in each row. We would split the App component to two parts: the top part includes the two input text boxes; and the bottom part includes the Writing Emphasis table. Then, we can further split the component for displaying the table into two components: one defines the CourseTable component, and another defines the TableRow component. This way the design considers scalability and modularity. We may reuse the CourseTable and the TableRow components as two separate building blocks.

We can describe the design process in several stages.

14.2.1 Stage I – Implement the Table for Writing Emphasis Courses

1. If the number of transfer credits is less than 40 credits, the student needs to take three Writing Emphasis courses.
2. If the number of transfer credits is between 40 and 70 credits, the student needs to take two writing emphasis courses. The screen is shown in Figure 14-1.

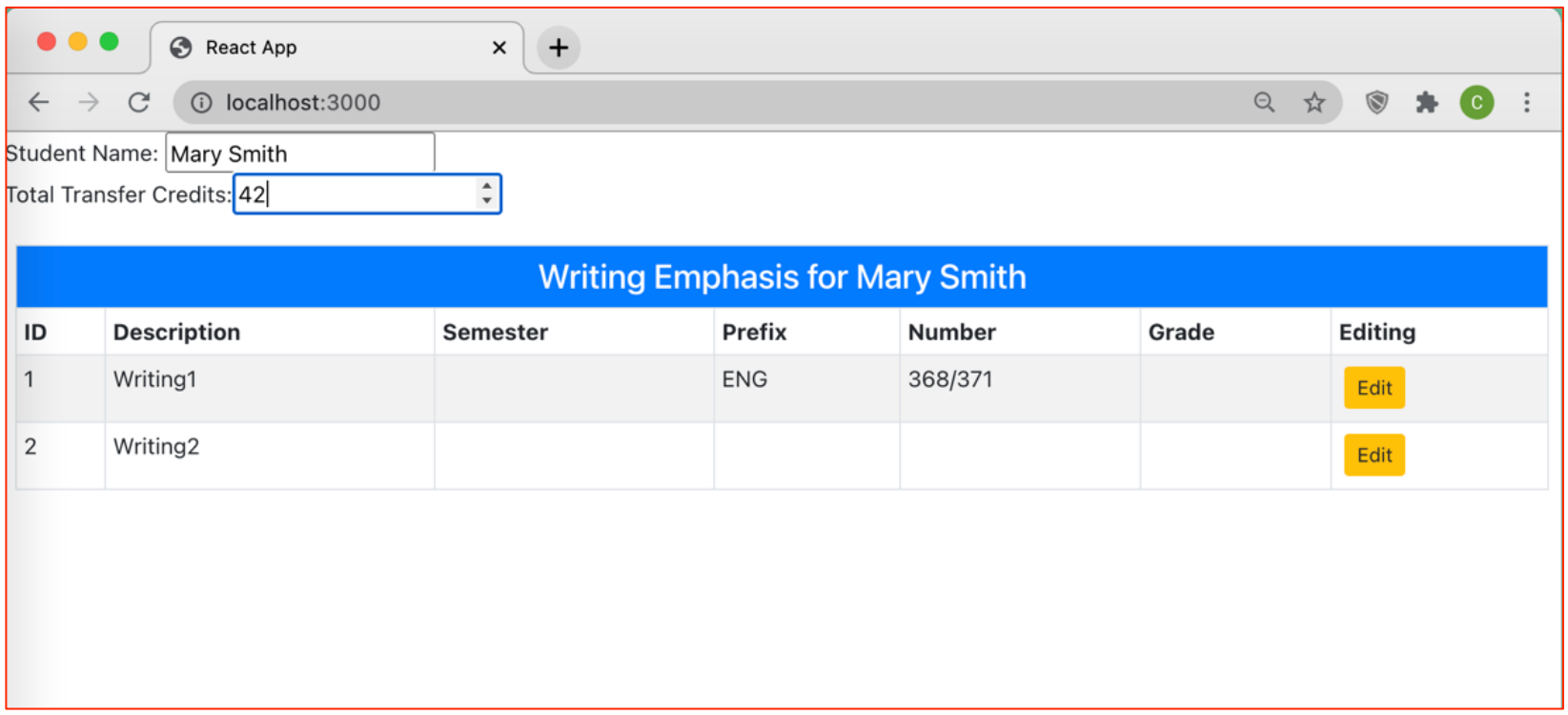


Figure 14-1. Transfer between 40 and 70 and credits.(Figure 14-1 Accessible Version)

3. If the number of transfer credits is greater than 70 credits, the student only needs to take one writing emphasis course. The screen shot is similar to the previous cases with only one course shown in the table.

14.2.2 Stage II – Implement the Table for Speaking Emphasis Courses

After the first stage is completed, we can begin to consider adding a Speaking Emphasis table without prompting for the student name and the total number of transfer credits. Similarly, we need to consider three cases.

1. Transfer Credits < 40 - Three courses are needed.
2. Transfer Credits < 71 - Two courses are needed.
3. Transfer Credits >= 71 - Only one course will be needed in this case.14.2.3 Summary of the Design Phase

After checking the requirements, we decided that we will need to break the UI into the following components:

1. Info Bar – containing two fields for a user to enter the name and the total number of transfer credits. Note that there is no additional button for a user to click on after filling out the two data fields: student name and the transfer credits. We will need to use the "*onChange*" attribute for the <input> fields instead of adding an additional <button> with the "*onClick*" attribute.
2. One TableRow component – we need to define only one table row component with the same data fields in each row.
3. One Table component with a property to distinguish if it is a writing emphasis or a speaking emphasis table. Again, when defining the table, we cannot assume that the maximum number of rows or courses is three.
4. Editing component – this component is used to display the editing screen for each course when the "Edit" button is clicked.

We will begin with the first four items focusing on adding only one table initially, say, the Writing Emphasis table. We may start with displaying the static part of the project without adding the event handling functions.

14.3 Development Phase for the Writing Emphasis Table

We will take a bottom-up approach for the implementation of each required component and describe the detailed steps from scratch.

1. Begin the development by creating a scaffold of a React app with the name *0-gen-ed-writing-emphasis-table* using the following command:

    >npx create-react-app 0-gen-ed-writing-emphasis-table

2. Install Bootstrap and React-Bootstrap libraries.

    >cd 0-gen-ed-writing-emphasis-table

>npm install react-bootstrap bootstrap

3. Modify the content in the file *index.js* to add the import statement for importing the Bootstrap library.

4. Add Components:

   a. Add a *FilterCourseTable* component in which an *InfoBar* component and a *CourseDisplay* component will be rendered.

   b. We need to add an InfoBar to allow a user to enter a name field and a transfer credit field.

   c. We need to add a *CourseDisplay* component to implement a table with rows of courses in the *CourseDisplay* component. We need to define a table component and a table row component.

5. The final step is to add an editing capability with an editing screen.

We describe the details about adding components next starting with some experiments to add components.

14.4 Adding Experimental Components

In our application, we need to add (1) user input fields, and (2) a table for displaying some rows of data. When we had no idea about how to begin our development, we will begin with some experiments initially using the react-bootstrap Form and Table components.

14.4.1 Add User Input Fields

First, we can find an example for adding a Form-like component so that a user can enter two fields. We can browse the official website of the React Bootstrap. We can find the tab Forms>Overview for implementing a BasicExample component which allows a user to enter a username and a password with a submit button. Now we should be able to start our first experiment by developing a UserInput component and add this BasicExample component to the App component.

```
import 'react-bootstrap';
import React, { Component } from 'react';
import Form from 'react-bootstrap/Form';
import Button from 'react-bootstrap/Button';

export default class UserInput extends Component {
      render() {
            return (
            <Form className="container-fluid">
            <Form.Group controlId="formBasicEmail">
            <Form.Label>Email address</Form.Label>
```

```
            <Form.Control type="email" placeholder="Enter email" />
            <Form.Text classNAme="text-muted">
            We'll never share your email with anyone else.
            </Form.Text>
            </Form.Group>

            <Form.Group controlId="formBasicPassword">
            <Form.Label>Password</Form.Label>
            <Form.Control type="password" placeholder="Password" />
            </Form.Group>
            <Form.Group controlId="formBasicCheckbox">
            <Form.Check type="checkbox" label="Check me out" />
            </Form.Group>

            <Button variant="primary" type="submit">
            Submit
            </Button>
            </Form>
            )
      }
}
```

14.4.2 Add a Table

Next, we can also find a component with the name Tables under the tab Components on the left-side of the Bootstrap Website again We will modify the BasicExample sample code for adding a Table component to our MyCourseTable component.

```
import React, {Component} from 'react';
import Table from 'react-bootstrap/Table';

export default class MyCourseTable extends Component {
      render() {
      return (
            <div classNAme="col-4">
            <Table bordered striped hover responsive>

            <thead>
            <tr>
            <th>#</th>
            <th>First Name</th>
            <th>Last Name</th>
            <th>Username</th>
            </tr>
            </thead>
```

```
            <tbody>
            <tr>
            <td>1</td>
            <td>Mark</td>
            <td>Otto</td>
            <td>@mdo</td>
            </tr>
            <tr>
            <td>2</td>
            <td>Jacob</td>
            <td>Thornton</td>
            <td>@fat</td>
            </tr>
            <tr>
            <td>3</td>
            <td colSpan="2">Larry the Bird</td>
            <td>@twitter</td>
            </tr>
            </tbody>

            </Table></div>
            )
      }
```

14.4.3 Putting Things Together

After these two experimental components are completed, we can define an App component to render the UserInput and the MyCourseTable components as follows:

```
import React from "react"
//import logo from './logo.svg';
import './App.css';
import MyCourseTable from "./MyCourseTable";
//import FilterableCourseTable from './FilterableCourseTable';
//import writingCourses from './FilterableCourseTable';
import UserInput from './UserInput';

class App extends React.Component {
      render () {
            return (
                  <div>
                  <UserInput />
                  <MyCourseTable />
                  </div>
            );
```

```
      }

}

export default App;
```

14.4.4 Execute the Final Experiment

We can test our experimental application using the following steps:

1. Entering the command : npm init —yes
2. Entering the command: npm init —yes
3. Enter the command: npm install react-bootstrap react-bootstrap react react-dom react-scripts
4. Add the entry under “Scripts” in the package.json file as follows:

```
      "scripts": {
      "start": "react-scripts start"
      }
```

5. Enter the command: *npm start*

This is a good start. Although this is not exactly what we want, we can envision the skeleton of our application with the two input fields on top and the table at the bottom. We can begin to modify the example code in the *UserInput* and *MyCourseTable* components based on the requirements of our application. More specifically, the MyCourseTable needs to be split into two components as we discussed previously. Moreover, we will need to use conditional rendering eventually in the final app.

14.5 Moving Towards the Goal

14.5.1 Application Structure

Based on the principle of “thinking in React,” we can envision a prospective project structure consists of the *public* folder, the *src* folder. The project structure has been discussed previously. We will define and include the following components in the *src* folder: I*nfoBar, CourseTable, CourseRow, CourseDisplay,* and *CourseEditor*. We will show the execution result and explain how each component is implemented.

14.5.2 Execution Result

The execution result shows what we need. This application contains an InfoBar component and a CourseTable component. Within the CourseTable, there are some instances of the CourseRow component. The CourseDisplay component encloses the CourseEditor and the CourseTable conditionally. When we execute

this application with a browser, the upper part is implemented by the InfoBar component, and the bottom part is implemented by a CourseDisplay component both inside the FilterableCourseTable.js.14.6 Implementation of the Required Components

14.6 Implementation

14.6.1 Implementation of the Input Fields

We will explain in detail how each required component is implemented next. In the InfoBar component, there are two input fields to be rendered: name and transfer credits. First, we will use the source code for implementing to illustrate how to add the Writing Emphasis Table. Then, we will consider extending the requirement to include the Speaking Emphasis Table afterward.

14.6.2 Implementation of the InfoBar Component

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

export default class InfoBar extends Component {
      constructor(props) {
            super(props);
            this.props = {
            studentName: "",
      transferCredits : 0
}
this.handleNameChange = this.handleNameChange.bind(this);
this.handleCreditChange = this.handleCreditChange.bind(this);
}

handleNameChange(e) {this.props.onNameChange(e.target.value);}

handleCreditChange(e)
{this.props.onCreditChange(e.target.value);}
render() {
const studentName = this.props.name;
const transferCredits = this.props.credits;
console.log("--- InfoBar ---");
return (
      <form>
      <form-label>Student Name: </form-label>
      <input type=“text" placeholder="Enter Name"
      value={studentName} onChange={this.handleNameChange}
      />
      <p>
      <label>Total Transfer Credits:</label>
      <input type=“number" placeholder="Enter Number of
```

```
            Transfer Credits” value={transferCredits}
            onChange={this.handleCreditChange}
            />
            </p>
            </form>
            );// end of return
      }// end of render
}// end of component
```

14.6.3 Implementation of the Writing Emphasis With Editing

The CourseDisplay component is used to choose to display or to edit an instance of a CourseRow inside the render() function.

```
import React, { Component } from "react";
//import { GenEdTables } from "./GenEdTables"
import { CourseEditor } from "./CourseEditor";
import { CourseTable } from "./CourseTable";

export default class CourseDisplay extends Component {
      constructor(props) {
      super(props);
      this.state = {showEditor: false, selectedCourse: null}
      }
      startEditing = (course) => {
      this.setState({ showEditor: true, selectedCourse: course })
}

createCourse = () => {
      this.setState({ showEditor: true, selectedCourse: {} })
}
cancelEditing = () => {
      this.setState({ showEditor: false, selectedCourse: null })
}

saveCourse = (course) => {
      this.props.saveCallback(course);
      this.setState({ showEditor: false, selectedCourse: null })
}
 render() {
      if (this.state.showEditor) {
            return <CourseEditor
            key={ this.state.selectedCourse.id || -1 }
            course={ this.state.selectedCourse }
            saveCallback={ this.saveCourse }
            cancelCallback={ this.cancelEditing } />
      } else {
```

```
            console.log("-- in CourseDisplay --");
            return (
            <div className="m-2">
            <CourseTable
            name={this.props.name}
            courses={ this.props.courses }
            credit = { this.props.credit }
            editCallback={ this.startEditing }
            deleteCallback={ this.props.deleteCallback } />
            </div>)
      }
 } // end of render
} // end of component
```

14.6.4 Implementation of the CourseTable Component

```
import React, { Component } from "react";
import CourseRow from "./CourseRow";

export class CourseTable extends Component {

      render() {
      const name = this.props.name;
      const credit = this.props.credit;
      const courses = this.props.courses;

      const rows = [];
      console.log("--- CourseTable ---");
      console.log ("Name:" + name);
      console.log ("Credit:" + credit);
      courses.forEach((c) => {
      console.log("Received a course: " + c.id + " "+ c.prefix +
      " "+ c.number);
      if (c.id === 1) {
            console.log("--course 1 " + c.course);
            rows.push(
            <CourseRow
            course={c}
            key={c.id}
            editCallback= { this.props.editCallback }
            />
            );
      }
      if (c.id === 2 && credit < 71) {
            console.log("--course 2 " + c.course);
            rows.push(
```

```
            <CourseRow
            course={c}
            key={c.id}
            editCallback= { this.props.editCallback }
            />
            );
      }
      if (c.id === 3 && credit < 41) {
            console.log("--course 3 " + c.course);
            rows.push(
            <CourseRow
            course={c}
            key={c.id}
            editCallback= { this.props.editCallback }
            />
            );
      }
      });
      return (
            <table className="table table-sm table-striped table-
      bordered">
            <thead>
            <tr>
                  <th colSpan="7" className="bg-primary
                  text-white text-center h4 p-2">
                  Writing Emphasis for {name}
                  </th>
            </tr>
            <tr>
                  <th>ID</th>
                  <th>Description</th>
                  <th>Semester</th>
                  <th>Prefix</th>
                  <th>Number</th>
                  <th>Grade</th>
                  <th>Editing</th>
            </tr>
            </thead>
            <tbody>{rows}</tbody>
            </table>
      ); // end of return
      } // end of render()
} // end of component
```

14.6.5 Implementation of the CourseRow Component

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

export default class CourseRow extends Component {

render() {
      let p = this.props.course;
      console.log("--- CourseRow ---" );
      return (
            <tr>
            <td>{p.id}</td>
            <td>{p.description}</td>
            <td>{p.semester}</td>
            <td>{p.prefix}</td>
            <td>{p.number}</td>
            <td>{p.grade}</td>

            <td>
            <button className="btn btn-sm btn-warning m-1"
            onClick={ () =>this.props.editCallback(p)} >Edit
            </button>

            </td>
            </tr>
      ); // end of return
      } end of render
} // end of component
```

14.6.6 Implementation of the CourseEditor Component

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

export class CourseEditor extends Component {

constructor(props) {
      super(props);
      this.state = {
            formData: {
            id: props.course.id || "",
            description: props.course.description || "",
            semester: props.course.semester || "",
            prefix: props.course.prefix || "",
            number: props.course.number || "",
            grade: props.course.grade || ""
            }
```

```
            }// end of formData
      } // end of this.state
handleChange = (ev) => {
      ev.persist();
      this.setState(state =>
      state.formData[ev.target.name] = ev.target.value);
}

handleClick = () => {
      this.props.saveCallback(this.state.formData);
}

render() {
            return <div className="m-2">
      <div className="form-group">
            <label>ID</label>
            <input className="form-control" name="id"
            disabled
            value={ this.state.formData.id }
            onChange={ this.handleChange } />
      </div>
      <div className="form-group">
            <label>Description</label>
            <input className="form-control" name="description"

            value={ this.state.formData.description }
            onChange={ this.handleChange } />
    </div>
    <div className="form-group">
            <label>Semester</label>
            <input className="form-control" name="semester"

            value={ this.state.formData.semester }
            onChange={ this.handleChange } />
    </div>
    <div className="form-group">
            <label>Prefix</label>
            <input className="form-control" name="prefix"
            value={ this.state.formData.prefix }
            onChange={ this.handleChange } />
    </div>
    <div className="form-group">
            <label>Number</label>
            <input className="form-control" name="number"
            value={ this.state.formData.number }
            onChange={ this.handleChange } />
```

```
            </div>
            <div className="form-group">
            <label>Grade</label>
            <input className="form-control" name="grade"
            value={ this.state.formData.grade }
            onChange={ this.handleChange } />
      </div>
      <div className="text-center">
            <button className="btn btn-primary m-1"
            onClick={ this.handleClick }>
            Save
            </button>
            <button className="btn btn-secondary"
            onClick={ this.props.cancelCallback }>
            Cancel
            </button>
      </div>
      </div>
      }//end of render()
}//end of component
```

14.6.7 Implementation of the GenEdCourseTable Component

```
import React, { Component } from "react";
import InfoBar from "./InfoBar";
import CourseDisplay from "./CourseDisplay";

export default class GenEdCourseTable extends Component {
constructor(props) {
super(props);
this.state = {
      name: "",
      credits: 0,
      writingCourses : [
       {id: 1, description: "Writing1",semester:'' ,
       prefix:'ENG', number: '368/371', grade: ' '},
       {id: 2, description: "Writing2",semester:'' ,
       prefix:'', number: ' ', grade: ' '},
       {id: 3, description: "Writing3",semester:'' ,
       prefix:'', number: ' ', grade: ' '}
      ]
};//end of this.state

      this.handleNameChange = this.handleNameChange.bind(this);
      this.handleCreditChange = this.handleCreditChange.bind(this);
} // end of constructor
```

```
handleNameChange(studentName) {
      this.setState({
      name: studentName
      });
}

handleCreditChange(trCredit) {
      this.setState({
      credits: trCredit
      })
}

saveData = (collection, item) => {
      console.log("*** Collection: " + collection);
      console.log("*** Item:" + item);
      if (item.id === "") {
            item.id = this.idCounter++;
            this.setState(state =>
            state[collection] = state[collection].concat(item));
      } else {
            this.setState(state =>
            state[collection] = state[collection].map(stored =>
            stored.id === item.id ? item: stored))
      }
}

deleteData = (collection, item) => {
      this.setState(state => state[collection]
      = state[collection].filter(stored => stored.id !== item.id));
}
render() {
      console.log("--- FilterableCourseTable ---");
      return (
            <div>
            <InfoBar
            name={this.state.name}
            credit={this.state.credits}
            onNameChange={this.handleNameChange}
            onCreditChange={this.handleCreditChange}
            />
            <CourseDisplay
            name={this.state.name}
            credit={this.state.credits}
            courses={ this.state.writingCourses }
            saveCallback={ c =>
            this.saveData("writingCourses", c) }
```

```
            deleteCallback={ c =>
            this.deleteData("writingCourses", c) }
            />
            </div>
      ); // end of return()
      } // end of render()
} // end of component
```

14.7 Adding the Editing Function

The final project requires the Writing Emphasis rows to be editable. If we click the *Edit* button, we will be able to enter some updated data. This is a form for editing each and every field in a course row including the description, the semester, the prefix, the course number, and the grade. Two buttons “save” and “Cancel” are displayed at the bottom.

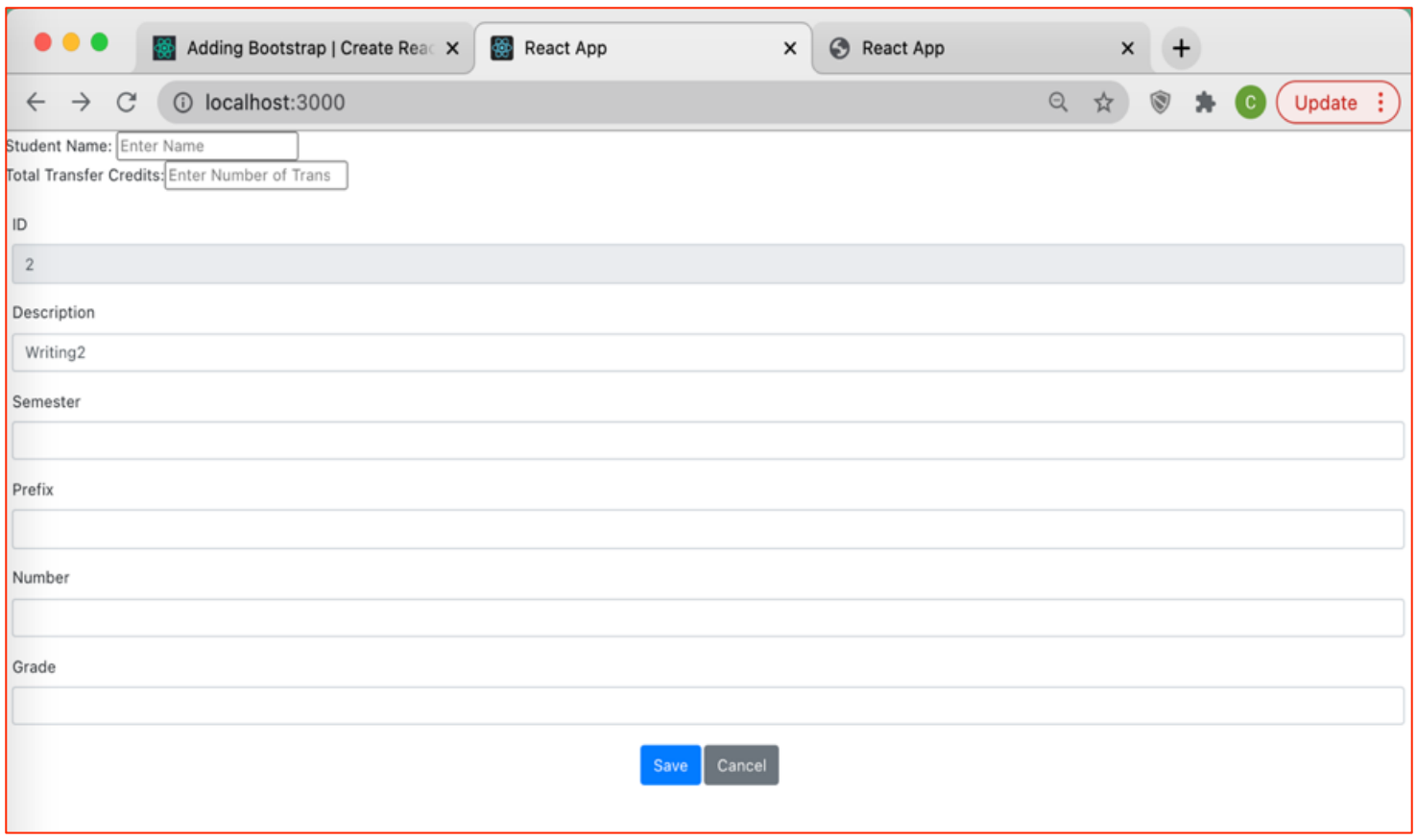


Figure 14-2. Editing Screen.

14.8 Development Phase for Speaking Emphasis Course Table

After the Writing Emphasis Table was completed, we can begin to consider adding the second table: Speaking Emphasis Table. Although it may seem to be trivial, it is actually not as easy as it looks. We will present several examples which use incorrect approaches to implement the second table. Then we present a direction on what a correct approach should be. Obviously, we cannot list all wrong approaches exhaustively. Here are two typical examples. Although the execution results may seem to be correct, these implementations violate the software engineering principles: modularity, reusability, maintainability, and readability. The soundness of these implementations is compromised.

14.8.1 Wrong Approach #1

An immediate approach is to define courses as part of the state for Writing Emphasis courses. To extend the courses for Speaking Emphasis, a quick-and-dirty design attempt is to duplicate state or functions in the ClassList component to include Speaking Emphasis. There are several severe flaws in this approach.

First, the component is extremely lengthy. Having inspected the content, we can see that the input fields for the name and the transfer credit are included in the definition of the two tables. This approach violates the principle of modularity.

Second, the conditional rendering was repeated. This implementation method duplicates the testing of transfer credits for the conditional rendering. Although this approach does not copy the set of components, it duplicates the testing part. This approach compromises the maintainability and modularity as well.

Third, the table row for each requirement includes a field “tableName” that is not needed. It is redundant because the name of the array already defines the type of rows as “writing.”

Fourth, the table rows are hard coded in the state data using the array containing the three somewhat similar but redundant objects: writing, writing1, and writing3 with the duplication of three objects: speaking, speaking1, and speaking2. This approach not only contributes to the size of the code, but also reduces the reusability of the ClassList component tremendously.

14.8.2 Wrong Approach #2

Another immediate intuition for developing the second table is to make a copy of all the components for implementing the first table and change the name from “Writing” to “Speaking.” This approach is also considered a quick-and-dirty approach or a copy-and-paste approach. Applying this design idea, we can find the following file structures inside the folder “components”:

1. Banner.js
2. Banner1.js
3. Form.js
4. NameDisplay.js
5. Table1Display.js
6. Table1Row.js
7. TableDisplay.js
8. TableRow.js

Upon checking the content in TableDisplay.js and Table2Display.js, we can find that they contain similar code for displaying tables.

This approach may seem to be easy and satisfy the requirement of displaying two tables. However, this approach sacrifices the reusability of the React components and makes the maintenance of the project a nightmare. When you need to make changes, you must rake through all relevant components and propagate the changes. Considering this following Java programming task: If you are given a Java function which displays one line of text with a serial number and you need to change the code to display one hundred lines of the same text each with a different serial number, you wouldn't want to make one hundred copies of the original function one hundred times and change the serial number accordingly. The correct way is to pass a parameter to the function and use this parameter to be the value for the serial number. Similarly, for the development of a Speaking Emphasis table, we need to consider a general solution to this requirement of displaying two tables with the possibility of scaling the project up to display many tables using a single set of components.

We will compare some of the similar components using the incorrect approach of duplicating components. The first one is Banner.js and the second one is Banner1.js.

What the developer does is making a copy of the file Banner.js for defining Banner, changing the term "Writing Emphasis" to "Speaking Emphasis," and calling the file Banner1.js. Although this is a quick-and-dirty approach, it generates a headache for maintenance in the future. A conscientious developer should strive to avoid this coding style entirely. However, some may disagree. "This approach has nothing to do with the behavior of the application," one may argue. "Why is it so bad?" The reason is that, in general, whenever a user's requirements change, we don't know how many files must be changed. Finding where the changes are within an unknown application code may become a major endeavor of maintenance similarly to finding a needle in a haystack. It is costly. We describe a correct approach using only a single "source of truth" in the next section.

14.8.3 Toward the Right Track

In general, there is no single solution that is robust. We are suggesting that we use the basic concepts we have covered to implement this application. The crux requires these components to define and use correct number of state and props. We present two major design ideas next.

First, For example, we should use a single MyCourseTable with a property to distinguish if this is a Writing Emphasis table or a Speaking Emphasis table. Our goal is to provide a robust approach considering the software engineering principles of reusability, modularity, and scalability. Although we are implementing three writing emphasis course rows in each table as well as three speaking emphasis course rows, we should be able to define a property to distinguish the table type. In the future, this project could be changed easily to scale up for a dynamic number of tables in the future.

Moreover, we should only need to use a single set of conditions to figure out the number of courses to be added to an array in the CourseTable component. This component can be used for generating a Writing Emphasis table as well as the Speaking Table without duplicating the testing statements.

We will leave this approach to the readers to implement the application as a project or an exercise.

14.9 Execution Steps

We need to follow the steps below to test the project.

- Step #1: npm install
- Step #2: npm start
- Step #3: Bring up a browser and enter “localhost:3000”
- Step #4: Click ‘Edit’ to modify the course data you choose.
- Step #5: Click ‘Save”

14.10 Exercise

Complete the project with Gen Ed requirements for the Writing Emphasis and Speaking Emphasis tables. To make the project more challenging, please implement the prefix for some courses as a predefined field that is not allowed to be changed. Users can change the prefixes for other courses except for the specific course. Hint: You need to disable the prefix for these courses.

Chapter 15 React Routing

One of the disadvantages for an SPA is that the "return to previous page" button on a browser in general is no longer available. For example, if you browse a long article by scrolling through many screen pages that are displayed using an SPA application, and occasionally, you want to go back to the prior "page" by hitting the "return" button, it does not work. Once you hit the "return" button, you are out of the SPA application. You may wonder if there is a solution to this issue. A short answer is "yes." In this lesson, we will introduce the concept of React Routing to illustrate this solution.

15.1 Why React Routing?

When we develop a server-side rendering application, the server parses the URL and determines which controller function is associated with this URL and executes the routing function. However, when we develop a single-page application, this controller feature is absent. As a result, the go back button on a browser is no longer bringing back the previous screen as there is only a single page. For example, if you bring up a Google map and shift the location to a different location, there is no way to return to the previous location; when a user hits the go back button, the application exits.

In some situations when the displayed data can be distinguished by hitting a button, it is appealing to be able to jump from one screen to another based on the specific URL. That's the main concept of React Routing. According to the main website for React Router, published by Remix Software, Inc., React Router enables client-side routing. React Router provides many features including Nested Routes, Dynamic Segments, Ranked Route Matching, Relative Links, Data Loading, Redirects, Pending Navigation UI, Skeleton UI with '<Suspense>', Data Mutations, Data Revalidation, Busy Indicators, Optimistic UI, Data Fetchers, Race Condition handling, Error Handling, Scroll Restoration, Web Standard API, Standard Params, and more features. We will only demonstrate how to use the basic feature of React routing.

15.2 React Router Version 6

As the book is written, the latest version is React router version 6. We will only focus on the basics of client-side routing supported by React Router with the first example taken from the Remix Software website. For detailed explanation about the other features, please refer to the Remix website document copyrighted by Remix.

15.2.1 Use React-Router-Dom Module

The first example demonstrates how to use the BrowserRouter module in the package react-router-dom to define a route as a link. First, we need to add the following code in app.js:

```
import './App.css';
import React, { Component } from "react";

import {
      BrowserRouter
} from "react-router-dom";

export default class App extends Component {
      render() {
            return (
            <BrowserRouter>
            <ReactRouter />
            </BrowserRouter>
            )
      }

}
```

In the ReactRouter.js file, we can add the following:

```
import React from "react";
import {
      createBrowserRouter,
      RouterProvider,
      Route,
      Link,
} from "react-router-dom";
const router = createBrowserRouter([
      {
            path: "/",
            element: (
                  <div>
                  <h1>Hello World</h1>
                  <Link to="about">About Us</Link>
                  </div>
            ),
      },
      {
            path: "about",
            element: <div>About</div>,
      },
]);
```

Since this example application is used to illustrate how to use the React router version 6, the application only displays one link: “About Us.”

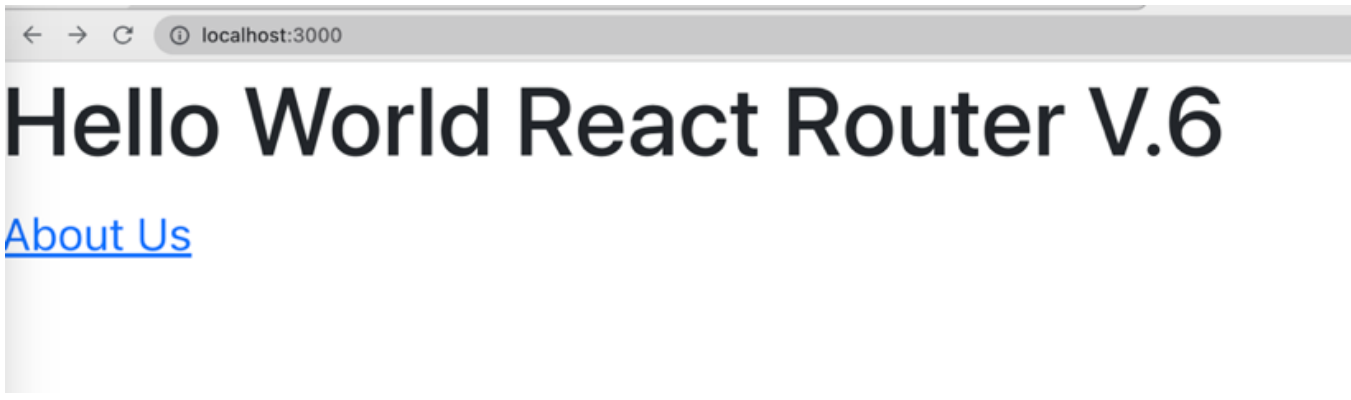


Figure 15-1. Execution Result.

15.2.2 Example of Gen Ed Course Table

In this next example, we will develop an application using the module react-router-dom to generate client-side routes. Our goal is to generate a simplified Gen Ed Course Table for illustrating the basic React routing feature.

15.2.3 Routing Without Using a Data Store

We will use the scaffolding command “npx create-react-app” to generate a skeleton and then modify the package.json, App.js, and GenEdTables.js files. We will illustrate these required changes to the generic project structure in the following sections.

15.2.4 File Structure

In this project file structure, you use only one set of CourseDisplay.js to display three tables of courses. You need to provide different properties for each table.

1. Since we will use React Routing Version 6, we need to check the package.json file for the following: (1) the version for react-router module is at least 6.3.0; (2) the version for the react-router-dom is at least 6.3.0.

```
      "react-router": "^6.14.1",
      "react-router-dom": "^6.14.1",
```

2. Change component *App* in App.js to import the <GenEdTables /> and add <BrowserRouter> to enclose the highest-level component <GenEdTables />.

```
import './App.css';
import React, { Component } from "react";
import GenEdTables from "./GenEdTables";
import {
      BrowserRouter
} from "react-router-dom";

export default class App extends Component {
      render() {
```

```
            return (
                  <BrowserRouter>
                        <GenEdTables />
                  </BrowserRouter>
            )
      }
}
```

3. The next step is to write the code for the <GenEdTables> component. We start from adding the import statements below:

```
      import React, { Component } from 'react';
      import { CourseDisplay } from './CourseDisplay';
      import {Routes, Route, NavLink} from "react-router-dom";
```

4. Then, we need to change the render () function with the <Route> tags enclosed in the <Routes> tag in the GenEdTables.js file. For the purpose of illustrating the main structure, we intentionally removed the style attribute.

```
render() {
      return (
      <div>
            <div>
            <h1>Display GenEd Courses</h1>
            </div>
            <nav >
            <div>
            <NavLink to="/Academic">Academic Foundation</NavLink>
            | {" "}
            <NavLink to="/Distributive">Distributive</NavLink>
            | {" "}
            <NavLink to="/Others">Other Courses</NavLink>
            </div>
            </nav>
            <Routes>
            <Route path="/Academic" element={<CourseDisplay
            name="Academic Foundation"
            course={ this.state.Acourses }
            saveCallback={ c => this.saveData("Acourses", c) }
            deleteCallback={
            c => this.deleteData("Acourses", c) }/>} />

            <Route path="/Distributive" element={<CourseDisplay
            name="Distributive Requirement"
            course={ this.state.Dcourses }
            saveCallback={ c => this.saveData("Dcourses", c) }
            deleteCallback={
            c => this.deleteData("Dcourses", c) }/>} />
```

```
            <Route path='Others' element={ <CourseDisplay
            name="Additional Requirement"
            course={ this.state.Ocourses }
            saveCallback={ c => this.saveData("Ocourses", c) }
            deleteCallback={
            c => this.deleteData("Ocourses", c) }/>}/>
            </Routes>
      </div>)
}
```

15.2.5 Execution Result

When the project is started on a web browser using “locahost:3000,” the screen displays the following with three routing tabs:

When the “Academic Foundation” tab is clicked, the screen displays the three tabs “Academic Foundation”, “Distributive”, and “Other Courses”.

When the “Distributive” tab is clicked, the Distributive Requirement courses are displayed.

When the “Other Courses” tab is clicked, the Additional requirement including the Writing Emphasis and Speaking Emphasis requirements are displayed. For the purpose of illustrating the concept of routing, the conditional rendering is ignored.

15.3 Exercises

1. Add the display function for the FYE (First-Year-Experience) course prefix. Note: The First-Year-Experience course is one of a list of courses taken by any incoming freshman as a way to prepare students for the university environment.
2. Define a variable as “let flag = true && (this.state.formData.prefix === ‘FYE’)and replace the “disabled” with “disabled={ flag }” in the rendering statement at the right place.
3. Add another two Tabs to display “Math” and “Major” course requirements. You may use the tabs “Math” and “Major” for your navigation tabs.

15.4 Additional Resources

1. Remix, “React Router", Accessed in April 2024.

Chapter 16 Redux State Management

The software Redux is a predictable state reservoir or a data store for JavaScript apps. We can use it with React, but Redux can be used with any other view libraries. Without using Redux, we must apply the principle of lifting state up to the common ancestor of all components sharing the state data. For some applications, this means defining all state variables in the root node of the DOM. With Redux, we define all state variables in a centralized data store. All components can access, update, or read the state values via Redux API functions. In this chapter, we will discuss the following topics: Redux, Detailed Concepts, and Redux Toolkit.

16.1 Introduction to Redux

In Figure 16-1, the relationship between a React component and a Redux Data Store is illustrated. Since Redux is "predictable" because the data stored in Redux will be immutable, the only way to make changes to the value is going through the function *useDispatch()* to dispatch an action. A component calls the event handler to trigger the function useDispatch() to be invoked. This call will trigger a slice of a *Reducer* to be executed. A slice of a reducer specifies (1) the name of the state variable, (2) the name of the reducer, and (3) the initial value.

When the state data are needed, the function *useSelector()* should be called can be used to return the current value. Prior to that, we need to use the function *createSlice()* to create a slice of the data store for the future accessing operations.

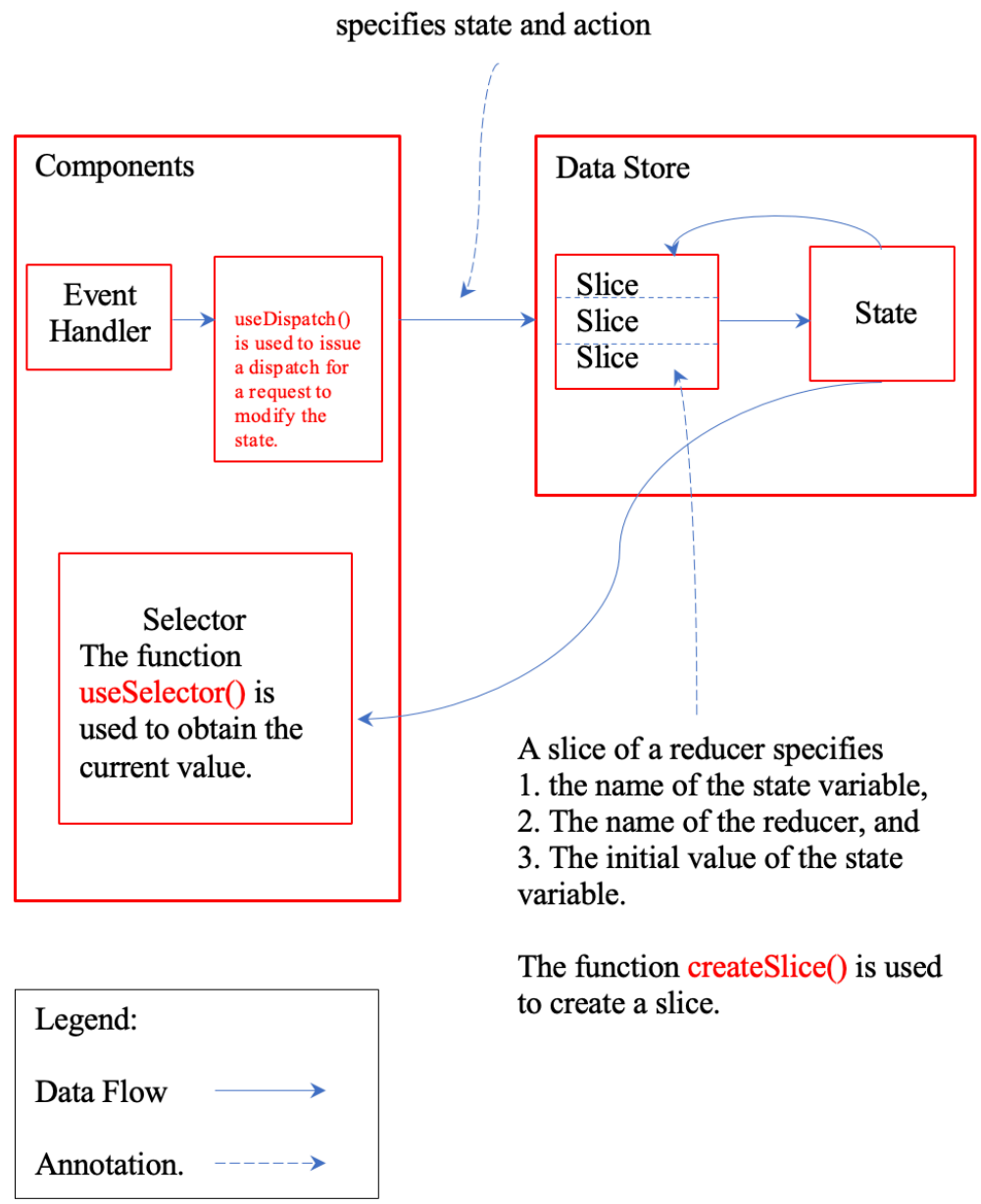


Figure 16-1. Redux Data Flow and Concepts. (Figure 16-1 Accessible Version)

16.2 Detailed Concepts

To use Redux API, we need to understand what some Redux nomenclature means, and the major rules to be complied with.

1. Components – We need to use *useDispatch(action, state)* to send a request for modifying the state variables in a Redux data store. We can call *useSelector()* to retrieve the state values.

2. Dispatch – Sends a request of changes to change the values of state variables.

3. Reducer – Receives a request and performs the changes.

4. Slice – Defines part of the data store information including the name of the reducer, the initial data of the state variables, and the names of the state variables.

5. Data flow is one way – Updating and accessing data can go only one way.

16.3 Software Dependencies

While Redux Toolkit provides some organized file structures for the Redux State Management development, it also comes with the issue of software dependencies. If your versions of Node.js, React, React-DOM, react-redux, and react-scripts are not working together, you may encounter many startling error messages. The following are the latest versions of software that work together when running the examples in this book.

The package.json contains the following:

```
{
"name": "1-redux-example1",
"version": "0.1.0",
"private": true,
"dependencies": {
      "@reduxjs/toolkit": "^1.9.1",
      "@testing-library/jest-dom": "^4.2.4",
      "@testing-library/react": "^9.3.2",
      "@testing-library/user-event": "^7.1.2",
      "react": "^18.2.0",
      "react-dom": "^18.2.0",
      "react-redux": "^8.0.5",
      "react-scripts": "^5.0.1"
      },
"scripts": {
      "start": "react-scripts start",
      "build": "react-scripts build",
      "test": "react-scripts test",
      "eject": "react-scripts eject"
```

```
      },
"eslintConfig": {
      "extends": "react-app"
      },
"browserslist": {
      "production": [
            ">0.2%",
            "not dead",
            "not op_mini all"
            ],
      "development": [
            "last 1 chrome version",
            "last 1 firefox version",
            "last 1 safari version"
            ]
      } // end of browsers list
} // end of the package.json file
```

The current node.js is V.18.12.0. If you are running V19.4.0 or higher, you may need to run the following command to downgrade node.js:

>npm install v18.12.0

The react-scripts may need to be upgraded to v. 5.0.1.

>npm install react-scripts@latest

The react-redux may need to be upgraded to 8.0.5.

>npm install react-redux@^8.0.5

16.4 Example - Redux Toolkit

Redux Toolkit provides a scaffolding template for the use of data stores with React, that is, React Redux Toolkit. We will use a sample program to illustrate how to develop React Components and access Redux. The example (copied from the Redux GitHub Toolkit [17]) is used under the MIT license.

16.4.1 Procedure

We will use a Redux toolkit to define a Redux store. The procedure as suggested in the Usage Guide [18] is listed below with the API functions involved, so beginners can understand the basics.

1. Install redux – Enter the following command:

```
      >npm install @reduxjs/toolkit react-redux
```

2. Create a Redux data store – Add the statement in component App:

```
      configureStore( {reducer:{}} )
```

3. Provide a Redux store to React – Add this statement in App:

```
<Provider store= “store”><App /></Provider>
```

4. Create a reducer slice using the API function createSlice().

5. Add the slicer Reducers to the data store –

```
import { configureStore } from '@reduxjs/toolkit';

import counterReducer from '../features/counter/counterSlice';
import creditReducer from '../features/credit/creditSlice';

export const store = configureStore({
      reducer: {
            counter: counterReducer,
            credit:creditReducer
      },
});
```

6. Use the function *useDispatch()* to modify the state value and the function *useSelector()* to retrieve the current values of the state variables.

16.4.2 Source Code

16.4.2.1 Provide the Redux Store To React

Create a file index.js inside the folder src:

```
import ReactDOM from ‘react-dom/client’;

var root = ReactDOM.createRoot(document.getElementById(‘root’));
root.render(
      <React.StrictMode>
            <Provider store={store}>
                  <App />
            </Provider>
      </React.StrictMode>
);
```

16.4.2.2 Create a Redux State Slice.

```
const initialState = {
      value: 0,
      status: 'idle',
};

// The function below is called a thunk and allows us to perform
      async logic. It
```

```
// can be dispatched like a regular action:
      `dispatch(incrementAsync(10))`. This
// will call the thunk with the `dispatch` function as the first
      argument. Async
// code can then be executed, and other actions can be dispatched.
      Thunks are
// typically used to make async requests.
export const incrementAsync = createAsyncThunk(
      'counter/fetchCount',
      async (amount) => {
      const response = await fetchCount(amount);
      // The value we return becomes the `fulfilled` action payload
      return response.data;
      }
);

export const counterSlice = createSlice({
      name: 'counter',
      initialState,
      //The reducers field lets us define reducers
      reducers: {
            increment: (state) => {
            // Redux Toolkit allows us to write "mutating" logic
      state.value += 1;
            },

            decrement: (state) => {
            state.value -= 1;
            },

            incrementByAmount: (state, action) => {
            state.value += action.payload;
      },
},

      extraReducers: (builder) => {
      builder
      .addCase(incrementAsync.pending, (state) => {
      state.status = 'loading';
      })
      .addCase(incrementAsync.fulfilled, (state, action) => {
      state.status = 'idle';
      state.value += action.payload;
      });
      },
      });
```

```
export const { increment, decrement, incrementByAmount } =
      counterSlice.actions;
```

16.4.2.3 Add Slice Reducers to the Store

```
export const store = configureStore({
      reducer: {
            counter: counterReducer,
      },
});
```

16.4.2.4 Use Redux State and Actions in React Components

```
import React, { useState } from 'react';
import { useSelector, useDispatch } from 'react-redux';
import {
      decrement,
      increment,
      incrementByAmount,
      incrementAsync,
      incrementIfOdd,
      selectCount,
      } from './counterSlice';
import styles from './Counter.module.css';
export function Counter() {
      const count = useSelector(selectCount);
      const dispatch = useDispatch();
      const [incrementAmount, setIncrementAmount] = useState('2');
      const incrementValue = Number(incrementAmount) || 0;
      return (
      <div>
      <div className={styles.row}>
            <button
            className={styles.button}
            aria-label="Decrement value"
            onClick={() => dispatch(decrement())}>
             -
            </button>
            <span className={styles.value}>{count}</span>
            <button
            className={styles.button}
            aria-label="Increment value"
            onClick={() => dispatch(increment())}>
            +
            </button>
      </div>
```

```
      <div className={styles.row}>
            <input
            className={styles.textbox}
            aria-label="Set increment amount"
            value={incrementAmount}
            onChange={(e) => setIncrementAmount(e.target.value)}
            />

            <button
            className={styles.button}
            onClick={() =>
      dispatch(incrementByAmount(incrementValue))}>
            Add Amount
            </button>

            <button
            className={styles.asyncButton}
            onClick={() => dispatch(incrementAsync(incrementValue))}>
            Add Async
            </button>

            <button
            className={styles.button}
            onClick={() => dispatch(incrementIfOdd(incrementValue))}>
            Add If Odd
            </button>
      </div>
      </div>
      ); // end of return
} //end of Counter()
```

16.4.2.5 Execution Results

A numeric state in a Redux data store and options to modify the data including *Add Amoun*t, *Add Asynchronously*, and *Add if Odd*, i.e., only does the addition when the number is an odd number.

16.5 Example – Define a Redux Store With Multiple State Variables

An App is used to illustrate adding a student name and the number of transferred credits to the InputStudentData component, calling a dispatch() call from the DisplayStudentData component, and using the useSelector() function to obtain the current values of the state variables stored in the data store.

16.5.1 Create a Redux Store

```
import { configureStore } from '@reduxjs/toolkit';

import creditReducer from '../features/student/studentSlice';
import studentReducer from '../features/student/studentSlice';

export const store = configureStore({
      reducer: {
            credit: creditReducer,
            student: studentReducer
      },
}); // end of configureStore()
```

16.5.2 Create a Reducer

```
import { createSlice } from '@reduxjs/toolkit';

const initialState = {
      credit: 0,
      student: '',
};

export const studentSlice = createSlice({
      name: 'student',
      initialState,

      reducers: {
            setCredit: (state,actions) => {
            state.credit = actions.payload;
            },
            setStudent: (state, actions) => {
            state.student = actions.payload;
            }
      }, // end of reducers
}); end of createSlice()

export const { setCredit, setStudent } = studentSlice.actions;
export const selectCredit = (state) => state.credit.value;
export default studentSlice.reducer;
```

16.5.3 Use a Class Component To Access the Data Store

We need to call the hook function useSelector() in the file InputStudentData.js to obtain the values for the state variables studentName and studentCredit.

```
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { setStudent, setCredit } from "../features/student/
     studentSlice";
import styles from './student.module.css';

export function InputStudentData() {
     const {studentName} = useSelector((state) => state.student);
     const {studentCredit} = useSelector((state) => state.credit);
     const dispatch = useDispatch();

     return (
          <div className="container-fluid p-4">
          <div className="row bg-info text-white p-2">
          Enter Student Name:
          <input
          className="styles.textbox"
          type="text"
          aria-label="Enter student name"
          value={studentName}
          onChange={(e) => dispatch(setStudent(e.target.value))} />
          </div>

          <div className="row bg-info text-white p-2">
          Enter Transferred Credits:
          <input
          className={styles.textbox}
          aria-label="Enter transferred credits"
          type="number"
          value={studentCredit}
          onChange={(e) => dispatch(setCredit(e.target.value))} />
          </div>
          </div>
     ); // end of return()
} //end of InputStudentData()
```

The component DisplayStudentData must call the useSelector() function to access the data store. Here is the source code.

```
import React from 'react';
import { useSelector } from 'react-redux';
```

```
export function DisplayStudentData() {

      const {credit} = useSelector((state) => state.credit);
      const {student} = useSelector((state) => state.student);

      return (
            <div>
            <h1 className="bg-primary text-white text-center">
            The total transferred credits for {student} is {credit}.
            </h1>
            </div>
      );
}
```

16.5.4 Execution Result

A Redux Data Store is used for storing the name of the student and the total transfer credits.

The browser displays two lines. Line 1 displays two textbox fields with the labels of “Entering Student Name” and “Enter Transfer Credits.” The second line displays a text “The total transferred credits for <name> is <number>.” Here the <name> and <number> are what the user enters. For example, if the name is John and the total number of credits is 40, the text display “The total transferred credits for John is 40.”

16.6 Managing State Using Hooks

As React continues to evolve, React version 18 releases new approaches for state management. Without using the hooks for managing state variables, we may confront with two issues. First, if we only need to store only a few state variables in a data store, Redux data store may be considered over-killed. Second, while the hook useState is a great way to create a state variable, developing a React component with several event handlers as separate functions can become overwhelming.

To resolve the previous two issues, React supports the state management with the hook setReducer() and the function dispatch() as another way to ‘centralize’ the various event handlers in one function. In the next example, we will modify the project sample that we used previously in the TextInput Project to illustrate the data store accessing operations using the hook setReducer and the function dispatch(). In this example we will use one reducer function to change the state values for the username and the user-input text depending on the ‘actions’ such as ‘changeName’ or ‘changeText.’

16.6.1 Define the TextMain Component

We will use the following steps to define the *TextMain* component:

1. Declare the state variables: *filterText* and *filterName*,
2. Declare the *reducer* function,
3. Define the event handlers, and
4. Finish the rendering with a return statement.

16.6.1.1 Declare the State Variables

First, we need to import the required module useReducer.

```
import {useReducer} from 'react';
```

The TextMain defines the state variables name and text with setReducer() instead of useState() below:

```
const [{filterName, filterText}, dispatch] =
useReducer( reducer, {filterName: "", filterText: ""});
```

In this code, the object {filterName, filterText} is used to store two state variables enclosed in curly braces. The useReducer() function accepts two arguments: the reducer function, and the initial values for the two state variables represented as a JSON object.

16.6.1.2 Declaring the Reducer Function

We will declare a single reducer function to modify the values of username and user-input text, respectively, depending on the type of action. If the action.type is changeText, the text value will be changed. If the action.type is changeName, the name value will be changed.

The reducer function accepts two parameters: the current state and the action. When the current state is passed to the reducer, the new state value will be returned. The source code is shown below:

```
function reducer (state, action) {
      console.log('In TextMain - reducer');
      switch ( action.type ) {
       case "changeText":
            console.log('type: changeText '+action.value);
            return {...state, filterText: action.value};
       case "changeName":
            console.log('type:changeName '+action.value);
            return {...state, filterName: action.value};
       default:
            throw new Error(`${action.type} is invalid`)
      }
}
```

16.6.1.3 Define the Event Handlers

The next step is to define two event handlers in a centralized file. One is for changing the value of the text to the new text; another is for changing the name value to the new name. The new value for the text will be passed in as a parameter to the handleTextChange() function; the new value of the name will be passed to the handleNameChange(). Then, the dispatch() function will be used to accept one parameter with the action type and the new value as a grouped together as a JSON object.

```
function handleTextChange(text) {
      console.log('++ Inside TextMain - handleTextChange' );
      dispatch({
            type: 'changeText',
            value: text,
      });
}
function handleNameChange(name) {
      console.log('++ Inside TextMain - handleNameChange' );
      dispatch({
            type: 'changeName',
            value: name,
      });
}
```

16.6.1.4 Complete the Rendering Statement With a Return

The last step in the TextMain component declares the rendering operation.

```
return (
      <div>
            <TextInput
            text={filterText}
            name={filterName}
            onFilterNameChange={handleNameChange}
            onFilterTextChange={handleTextChange}
            />

            <TextDisplay
            text={filterText}
            name={filterName}
            />
      </div>
);
```

This concludes the definition of TextMain component which becomes the common ancestor of two other components: TextInput and TextDisplay.

16.6.2 Define the InputText Component

We will illustrate the content of the *InputText* component in several parts:

1. Define the function component with the properties: name, text, *onFilterNameChange*, and *onFilterTextChange*.

2. Add the event handlers: *onFilterNameChange* and *onFilterTextChange*.

3. Complete the rendering with the *return* statement.

16.6.2.1 Declare the Function Component

We define the props and the event handlers in the function component as follows:

```
export default function TextInput({name, text,
      onFilterNameChange, onFilterTextChange}) {

      const handleFilterTextChange = (e) => {
            console.log ('handleFilterTextChange ' + e.target.value);
            onFilterTextChange(e.target.value)
      }
      const handleFilterNameChange = (e) => {
      console.log('handleFilterNameChange' + e.target.value);
      onFilterNameChange(e.target.value)
      }
} // end of TextInput
```

16.6.2.2 Complete the Rendering

We will use a return statement to complete the rendering activity.

```
return (
      <form className="bg-primary text-white text-center p-2 m-2">

      Please enter one line of text <br/>
      <input
            type="text"
            placeholder="Enter text..."
            value={text}
            onChange={handleFilterTextChange}
      />

      <br/>
      Please enter your name <br/>
      <input
            type="name"
            placeholder="Enter name..."
```

```
            value={name}
            onChange={handleFilterNameChange}
      />
      </form>
);
```

This concludes the definition of the TextInput component.

16.6.3 Define the TextDisplay Component

The function component TextDisplay is straight-forward.

```
import React from 'react';

export default function TextDisplay( {text, name}) {
      console.log("inside TextDisplay");
      console.log(text);
      console.log(name);
      return (
            <h4 className="bg-primary text-white text-center p-2
      m-2">
            Hello {name} <br/>
            You just entered: {text}
            </h4>
      )

}
```

16.6.4 Execution Result

The first screen shows two text fields as two instances of the <input> tag. One is for a user to enter one line of text, and another is for a user to enter the username. Then, the bottom part of the screen shows the label using the text and the username the user just entered.

Epilogue: Where Do We Go From Here?

There are several directions one may take toward becoming a professional UI programmer. First, you need to know that there are other view libraries or frameworks available. For example, Vue.js is a JavaScript library that may be mixed with React.js. Angular (used to be Angular.js) is a framework that works without using React or Vue.

Second, you also need to know that React is good for developing isomorphic web apps. But there are other JavaScript libraries that are good for the development of Server-Side Rendering web apps. For example, Next.js and Storybook are two popular ones.

As a professional UI developer, you will need to become familiar with these features and learn how to optimize your web application development.

Acknowledgement

The author would like to thank the past students who took the course entitled User Interfaces. Your participations and comments make teaching a rewarding experience.

The author would like to thank Hunter Musselman for proofreading the draft of the e-textbook.

Also, the author would like to thank the funding provided by the Teaching and Learning Center under a Department of Education grant (PA-ADOPT), and the well-prepared training and guidance provided by Marc Drumm and Hannah Glatt. PA-ADOPT is an initiative funded by the US Department of Education seeking to create a collection of high-quality eTextbooks that are freely disseminated using the principles of Open Educational Resources (OER).

Most important of all, the author is indebted to Dr. Eleanor Shevlin and her students who provided invaluable comments in copyediting.

Appendix: Selected Figure Descriptions

Figure 4-1 Project 3-Jsx File Structure-1

```
v 3-jsx
      > node_modules
      v public
            App.js
            index.html
      v server
            .server.js
      v src
            .babelrc
            App.jsx
      commands.md
      package-lock.json
      package.json
      README.md
```

Return to Figure 4-1 in Chapter 4

Figure 14-1 Writing Emphasis Table

This web form is used to display the Writing Emphasis Table. We need the user to enter the student name and the total transfer credit to complete the table.

Student Name: Mary Smith

Total Transfer Credits: 42

Writing Emphasis for Mary Smith

ID	Description	Semester	Prefix	Number	Grade	Editing
1	Writing1		ENG	368/371		Edit
2	Writing2					Edit

Return to Figure 14-1 in Chapter 14

Figure 16-1 Redux Data Flow and Concepts

This diagram illustrates the relationship between a React component and the Redux Data Store. A component calls the event handler to trigger the function useDispatch() to be invoked. This call will trigger a slice of a Reducer to be executed. A slice of a reducer specifies (1) the name of the state variable, (2) the name of the reducer, and (3) the initial value.

When the state data are needed, the function useSelector() should be called can be used to return the current value. Prior to that, we need to use the function createSlice() to create a slice of the data store for the future accessing operations.

Return to Figure 16-1 in Chapter 16

Index

J

L

M

N

O

P

R

S

T

U

V

W

Bibliography

(1) S. Smith, "Characteristics of Modern Web Applications," Microsoft Corporation, 2022.

(2) Meta Open Source, "React Docs," Accessed in July 2023. [Online]. Available: http://reactjs.org.

(3) Wikipedia Foundation Inc., "React (A JavaScript Library)," [Online]. Available: https://en.wikipedia.org/wiki/React. [Accessed in August 2023].

(4) Microsoft, "Create an ASP.NET Core app with React in Visual Studio," Available: https://docs.microsoft.com/en-us/visualstudio/javascript/tutorial-asp-net-core-with-react. [Accessed in August 2023].

(5) IETF Working Group, "HTTP RFC 2616," IETF, 1999. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2616.html. [Accessed in August 2022].

(6) reactjs.org, "How to Upgrade to React 18," 2022. [Online]. Available: https://reactjs.org/blog/2022/03/08/react-18-upgrade-guide.html. [Accessed in August 2022].

(7) B. Krajka, "The difference between Virtual DOM and DOM," 2015. [Online]. Available: https://reactkungfu.com/2015/10/the-difference-between-virtual-dom-and-dom/. [Accessed in August 2023].

(8) reactjs.org, Meta Platforms, Inc., 2023. [Online]. Available: https://reactjs.org/docs/events.html. [Accessed in September 2023].

(9) reactjs.org, "Synthetic Events," Meta Platform, Inc., [Online]. Available: https://reactjs.org/docs/events.html#form-events. [Accessed in August 2023].

(10)"HTML Input Tag," w3schools.com, [Online]. Available: https://www.w3schools.com/tags/tag_input.asp. [Accessed in September 2023].

(11)"Types for the HTML input tag," w3schools.com, [Online]. Available: https://www.w3schools.com/tags/att_input_type.asp. [Accessed in September 2023].

(12)"Event Attributes," w3schools.com, [Online]. Available: https://www.w3schools.com/tags/ref_eventattributes.asp. [Accessed in September 2023].

(13)"Form Events," w3school.com, [Online]. Available: https://www.w3schools.com/tags/ref_eventattributes.asp. [Accessed in September 2022].

(14)W. Maj, "React Component Lifecycle Methods Diagram Online Version," [Online]. Available: https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/. [Accessed in September 2023].

(15)D. Abramov, "Interactive React Lifecycle Methods diagram," April 2018. [Online]. Available: https://twitter.com/dan_abramov/status/981712092611989509. [Accessed in September 2023].

(16)reactjs.org, "reactjs.org," Meta Platforms, Inc., 2023. [Online]. Available: https://react.dev/reference/react-dom/components[Accessed in October 2023].

(17)D. A. a. D. authors, 2015-2022. [Online]. Available: https://redux-toolkit.js.org/. [Accessed in September 2023].

(18)D. A. a. d. authors, "Redux Usage Guide," 2015-2022. [Online] [Accessed in September 2023].. Available: https://redux-toolkit.js.org/usage/usage-guide. [Accessed in September 2022].

(19)V. Hiran, "Why Reactjs? Reasons to choose for your next project," August 2022. [Online]. Available: https://codetoart.com/blog/why-reactjs-reason-to-choose-for-your-next-project. [Accessed in September 2022].

(20)2022 Meta Platforms, Inc., "reactjs.org," Meta Platforms, Inc., 2022. [Online]. Available: https://reactjs.org/docs/hooks-intro.html. [Accessed in August 2022].

(21)storybook.s.org, "Guides," Chromatic Softwware, Inc., [Online]. Available: https://storybook.js.org/docs/react/get-started/introduction. [Accessed in October 2022]