**React JS Functional Programming concept:**

**Function Declaration vs Function Expression**

Function Declaration supports Hoisting where as Function expression does not.

// Invoking the function before it's declared

hey();

// Function Declaration

function hey() {

 alert("hey!");

}

/\*This works. You’ll see the alert appear in the browser. It works because the function is

hoisted, or moved up, to the top of the file’s scope. Trying the same exercise with a

function expression will cause an error:\*/

// Invoking the function before it's declared

hey();

// Function Expression

const hey = function() {

 alert("hey!");

};

//TypeError: hey is not a function

* In JavaScript functions are variables.
* const log = message => { console.log(message); };
* Since functions are variables, we can add them to objects:

const obj = {

message: "They can be added to objects like variables",

log(message) {

console.log(message);

}

};

obj.log(obj.message); // They can be added to objects like variables

**Higher Order Functions:** : functions that either take or return other functions.

const example = message => {

    return function(msg) {

        message(msg.toUpperCase()+ "")

    }

}

const createScream = logger => message => { logger(message.toUpperCase() + "!!!"); };

Functional programming is part of declarative programming means it is concerned about what should happen over defining how it should happen which is imperative programming.

Imperative focusing on how to achieve hyphen in string

const string = "Restaurants in Hanalei";

const urlFriendly = "";

for (var i = 0; i < string.length; i++) {

 if (string[i] === " ") {

 urlFriendly += "-";

 } else {

 urlFriendly += string[i];

 }

}

console.log(urlFriendly); // "Restaurants-in-Hanalei

Declarative or Functional Programming:

**const string = "Restaurants in Hanalei";**

**const urlFriendly = string.replace(/ /g, "-");**

**console.log(urlFriendly);**

**React is Declarative:**

Now lets take a look how to construct DOM declaratively using a React component.

**Single Page Application:**

In an SPA, the browser initially loads one HTML document. As users navigate through the site, they actually stay on the same page. JavaScript destroys and creates a new user interface as the user interacts with the application. It may feel as though you’re jumping from page to page, but you’re actually still on the same HTML page, and JavaScript is doing the heavy lifting.

**How React Works:**

* We need two libraries React and React DOM. React is the library to create views. ReactDOM is the library used to actually render the UI in the browser.
* const root = ReactDOM.createRoot(document.getElementById('root'))
* root.render(
* <React.StrictMode>
* <BrowserRouter>
* <AuthProvider>
* <App />
* </AuthProvider>
* </BrowserRouter>
* </React.StrictMode>
* )

**React Elements:**

React Elements HTML is simply a set of instructions that a browser follows when constructing the DOM. The elements that make up an HTML document become DOM elements when the browser loads HTML and renders the user interface.

**React is a library that’s designed to update the browser DOM for us.** We no longer have to be concerned with the complexities associated with building high-performing SPAs because React can do that for us. With React, we do not interact with the DOM API directly. Instead, we provide instructions for what we want React to build, and React will take care of rendering and reconciling the elements we’ve instructed it to create.

The browser DOM is made up of DOM elements. Similarly, the React DOM is made up of React elements. DOM elements and React elements may look the same, but they’re quite different. A React element is a description of what the actual DOM element should look like. In other words, React elements are the instructions for how the browser DOM should be created.

We can create a React element to represent an h1 using React.createElement:

**React.createElement("h1", { id: "recipe-0" }, "Baked Salmon");**

**A React element is just a JavaScript literal that tells React how to construct the DOM element.**

**ReactDOM**

Once we’ve created a React element, we’ll want to see it in the browser. ReactDOM contains the tools necessary to render React elements in the browser. ReactDOM is where we’ll find the render method

const dish = React.createElement("h1", null, "Baked Salmon");

const dessert = React.createElement("h2", null, "Coconut Cream Pie");

ReactDOM.render([dish, dessert], document.getElementById("root"));

This will render both of these elements as siblings inside of the root container.

**Passing Data from Parent to Child:**

In the case of sharing data from parent to child component, we use props. Props data is shared by the parent component and Child component cannot be changed as they are read-only.

**Parent.js**

return (

<div>

<div class="row m-4 ">

<h3>Parent To Child</h3>

**<Child dataList={dataList} />**

</div>

</div>

);

**Child.js**

const Child = (props) => {

return props.dataList.map((item, i) => {

return (

**Passing Data from Child to Parent:**

For Parent, component creates a callback Function and its Function gets the data from the child component.

Pass the callback function in the parent component as a prop to the child component.

The child component calls the parent callback function using props.

[**https://dev.to/turpp/passing-data-between-react-components-af7**](https://dev.to/turpp/passing-data-between-react-components-af7)

**JSX Rules:**

JSX combines the JS from JavaScript and the X from XML. It is a JavaScript extension that allows us to define React elements using a tag-based syntax directly within our JavaScript code.

Sometimes JSX is confused with HTML because they look similar**. JSX is just another way of creating React elements**, so you don’t have to pull your hair out looking for the missing comma in a complex createElement call.

**Important**:

React is to help us how DOM should be created, and we don’t interact directly with browser DOM so react creates elements which defines browser DOM.

So basically React helps us in achieving Single Page Application

So creating react elements using react.createElements is tedious process so JSX helps us in creating react elements in JavaScript itself using tag based syntax.

**JSX Expression:**

To assign some value = {props.name} like task={props.name} which is called JavaScript expression and it can include objects, arrays and functions as well.

Evaluation The JavaScript that’s added in between the curly braces will get evaluated. This means that operations such as concatenation or addition will occur. This also means that functions found in JavaScript expressions will be invoked:

**<h2>{"Hello" + title}</h2>**

<h2>{title.toLowerCase().replace}</h2>

To write some condition lets say you want to iterate over array of elements then you can write inside closed braces {}

{tasks.map((task) => (

            <Task key={task.id} task={task} onDelete={onDeleteTask}></Task>

          ))}

When child component tries to access callback function which came from Parent then we can write using arrow functions

onClick={() => onDelete(task.id)}

**JSX looks clean and readable, but it can’t be interpreted with a browser. All JSX must be converted into createElement calls. Luckily, there’s an excellent tool for this task: Babel.**

**Babel:** Which will help us to convert JSX to JavaScript.

**Webpack:**

JavaScript/React code is being compiled by something like webpack is vital

Webpack is billed as a module bundler. A module bundler takes all of our different files (JavaScript, LESS, CSS, JSX, ESNext, and so on) and turns them into a single file. The two main benefits of bundling are modularity and network performance.

**Code splitting** Splits up your code into different chunks that can be loaded when you need them. Sometimes these are called rollups or layers; the aim is to break up code as needed for different pages or devices.

**Minification** Removes whitespace, line breaks, lengthy variable names, and unnecessary code to reduce the file size. Feature Flagging Sends code to one or more—but not all—environments when testing out fea‐ tures.

**Hot Module Replacement (HMR)** Watches for changes in source code. Changes only the updated modules immediately.

* **Webpack.config.json**
* var path = require("path");
* module.exports = {
* entry: "./src/index.js",
* output: {
* path: path.join(\_\_dirname, "dist", "assets"),
* filename: "bundle.js"
* }
* module: {
* rules: [{ test: /\.js$/, exclude: /node\_modules/, loader: "babel-loader" }]
* }
* };

Here if we want to override webpack configuration then we can create file called webpack.config.js

We can specify the entry file name which is index.js

We can specify loaders like babel-loader in module

**Source Mapping**

Bundling our code into a single file can cause some setbacks when it comes time to debug the application in the browser. We can eliminate this problem by providing a source map. A source map is a file that maps a bundle to the original source files. With webpack, all we have to do is add a couple lines to our webpack.config.js file. 94 | Chapter 5: React with JSX //webpack.config.js with source mapping

**module.exports = {**

**...**

**devtool: "#source-map" // Add this option for source mapping**

**};**

Setting the devtool property to '#source-map' tells webpack that you want to use source mapping. The next time you run webpack, you’ll see that two output files are generated and added to the dist folder: the original bundle.js and bundle.js.map.

**Create-react-app**

import Statements We’re using import statements, which are not presently supported by most browsers or by Node.js. The reason import statements work is that Babel will convert them into require('module/ path'); statements in our final code. The require function is how CommonJS modules are typically loaded.

**The starting file for our Recipes app is index.js. It imports React, ReactDOM, and the App.js file.**

create-react-app my-project

This will create a React project in that directory with just three dependencies: React, ReactDOM, and react-scripts. react-scripts was also created by Facebook and is where the real magic happens. It installs Babel, ESLint, webpack, and more so that you don’t have to configure them manually.

**Pure Components:**

A pure component is a function component that does not contain state and will render the same user interface given the same props.