1.4. Introduction to React.js framework for frontend development

React.js is a **JavaScript framework** originally developed at Facebook in 2013 and designed for building user interfaces. It is a **declarative** framework, declaring what should be displayed instead of how. Declarative views make the code more predictable and easier to debug. Many websites have been developed using React since its release. React allows the development of *reusable User Interface (UI) components*, *Single Page Application (SPA)*, and *separation of front-end and back-end development*. Components are reusable code like libraries. React was developed with the purpose of enabling the developer to create UIs using **pure JavaScript** instead of introducing new template languages (to generate and render HTML pages).

A Single Page Application is a web application or website that interacts with the user by dynamically rewriting the current web page with new data from the web server instead of a web browser loading entire new pages. The goal is a faster transition that makes the website feel more like a native application. In a SPA, a page refresh never occurs; instead, all necessary HTML, CSS, and JavaScript code are either retrieved by the browser with a single page load, or the appropriate resources are dynamically loaded and added to the page as necessary, usually in response to user actions.

Angular is another JavaScript framework developed by Google and designed for similar purposes.

Study guide

- React JSX
- Components and the lifecycle of components
- Properties (props)
- Various React Hooks such as “useState”, “useEffect”, etc.
- Rendering in React, VirtualDOM vs. HTML DOM
- Client-side routing
- Client-side rendering
- Testing
- Building and deployment of React app

Some tutorial sites referenced in this chapter

- https://reactjs.org/docs/getting-started.html is an official site for React.
- https://reactjs.org/tutorial/tutorial.html for a simple tutorial
- https://www.w3schools.com/REACT/DEFAULT.ASP for simple examples
- Full react tutorial by The Net Ninja
- React crash course by Traversy Media

A simple React page

Create a JavaScript file called “first.js” including the following line of code:

```
//first.js
ReactDOM.render(<h1>Hello world! This is my first React page.</h1>, document.getElementById("root"));
```

Create a HTML file, e.g., “first.html” that includes “first.js” in <script> tag as follow:

```
<!DOCTYPE html>
<html lang="en">
<head>
```

```
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <script crossorigin src="https://unpkg.com/react@17/umd/react.development.js"></script>
    <script crossorigin src="https://unpkg.com/react-dom@17/umd/react-dom.development.js"></script>
    <script crossorigin src="https://unpkg.com/babel-standalone@6/babel.min.js"></script>
</head>
<body>
    <div id="root"></div>
    <script src="first.js" type="text/babel"></script>
</body>
</html>
```

Open the HTML file “first.html” using a browser without installing any react modules or setting up any web server. You will be able to see a message on your browser because the above HTML file includes all the necessary JavaScript files to interpret React code in “first.js”. This means that the **browser executes React code** after compilation and **renders the HTML page on the client-side (client-side rendering)**. That’s why **React programming** is considered **frontend development**.

What is client-side rendering and server-side rendering?

Client-side rendering means that rendering occurs on the client-side. With client-side rendering, the browser makes a request for a web page, the server responds, and the browser reads it in an HTML file and makes requests for any additional resources defined in it (CSS, JavaScript, images, etc.). Once the JavaScript is loaded and executed, it renders the HTML file on the browser screen. React is used for client-side rendering. **Server-side rendering** (SSR) is a technique for rendering a web page on the server and then sending it to the client (e.g., browser). SSR is useful in situations where the client has a slow Internet connection and then rendering of the whole page on the client-side takes too much time and SEO (search engine optimization) of your React application (since it removes the burden of rendering JavaScript off of search engine bots) at the cost of the server. For server-side rendering, Node.js, Express.js, or Next.js (basically web application frameworks) can be used.

Create another file “myreact.js” including the following React code without any HTML tags:

```
//myreact.js
import React from 'react';
import ReactDOM from 'react-dom/client';
const myElement = <h1>Hello, world! This is my first React page.</h1>;
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```

Since this is a JavaScript file, you will not be able to open it using a browser. How about using the Node command by executing “myreact.js” using the following Node command in a terminal?

```
node myreact.js
```

You will receive an error message from the execution since “myreact.js” use several libraries or components that are not part of standard JavaScript.

Which libraries do I have to install, and how can I set up a development environment for React applications?

React application development requires several libraries, a React compiler, and toolchains. Some useful and commonly used React toolchains include:

- babel or TypeScript for compiler
- create-react-app for SPA-oriented toolchain
- serve for a lightweight web server
- express or next.js for server-side rendering
- gatsby for static content-oriented toolchain

Refer to this page or that page for all the required libraries to install and set up.

For easy setup for React application development using create-react-app tool

1) Open a terminal, create a new project folder for a React application, and change the current directory to the project directory.
2) Type the following commands to create a React application 'myreact'.
   npx create-react-app myreact
   npm start

'npx create-react-app myreact' command will set up everything necessary to run a React application, including some default files. If you do not have create-react-app, you can install it by: 'npm install -g create-react-app'.

*If 'npx create-react-app' command doesn't work (possibly due to the previous installation), clean up the cache using yarn and try it again:

```
npm install yearn
yarn cache clean
npx create-react-app myreact
```

'npm start' command will start a development server and open the default page. To verify if the server is started (if it is not started automatically), open a browser and type '**localhost:3000**', then it will open the React page "App.js" in '**src**' folder. The default port is 3000.

npm vs **npx** vs **yarn**

- **npm** (Node Package Manager) is the **dependency or package manager** that helps install packages both globally and locally (in 'node_modules' directory). If you wish to run a package, you have to specify that package in your **package.json** (by changing "**scripts**" key) and install it locally. **To run a package**, enter 'npm run your-package-name'.
- **npx** (Node Package Execute) allows a package execution **without installing** the package. It is basically a npm package runner, so if any packages aren't already installed, it will install them automatically. This feature is useful when you want to take a look at a specific package and try out some commands, which is not possible with npm. If npx is not installed, install it 'npm install -g npx'. To execute a package, 'npx your-package-name'. You can run code directly from GitHub, 'npx URL'.
- **yarn** is yet another package manager that is known to be faster.

What is create-react-app?

Create-react-app is a tool to create a SPA that is officially supported by the React team. The script generates the required files and folder to start the React app and run it on the browser. Under the hood, create-react-app uses **Babel** (JavaScript compiler) and "**webpack**" (bundler for JavaScript files).

'npx create-react-app' command **will create the following files and folders by default:**

- README.md is a file that includes helpful tips and links for learning.
- **node_modules** is a folder that includes all of the dependency-related code.
- **package.json** is a file that manages the **app dependencies** (required modules) included in 'node_modules' folder and the scripts to run your app. When you look into the **package.json** file, **react-scripts** (/node_modules/react-scripts/bin/) are scripts to run the build tools required to transform React JSX syntax into plain JavaScript. You can create package.json 'npm init' if package.json is not created.
- **.gitignore** is a file to exclude files and folders by Git. A git repository is created whenever you create a new project with 'create-react-app'.
- **public** is a folder to store static files.
- **src** is a folder that contains the source code.

Review the following files before you write your own React code:

- public/index.html (the page template)
- src/index.js (the JavaScript entry point)
- src/App.js (App components)
- src/App.css (CSS for App)

index.html file is a template page for the default app created using the create-react-app tool.

```
  -->
  <title>React App</title>
</head>
<body>
  <noscript>You need to enable JavaScript to run this app.</noscript>
  <div id="root"></div>
  <!--
```

All the React code will be injected into division. So, this is a mounting point for React app. **Mounting** means adding nodes to the DOM.

index.js file will **mount** the "App" component that was imported to the DOM and **render** it to "root" element.

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

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

The imported "React" is for component-related tasks. The "**react-dom/client**" is for rendering the components in the DOM. There is also a component called "react-dom/server" that enables you to render components to static markup. It is typically used on a Node server.

App.js file is the main file for React components. This file exports a component "**App**".

```
import logo from './logo.svg';
import './App.css';
function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>Edit <code>src/App.js</code> and save to reload.</p>
        <a className="App-link" href="https://reactjs.org" target="_blank" rel="noopener noreferrer">
          Learn React
        </a>
      </header>
    </div>
  );
}
export default App;
```

This is going to be the root component of this application. All the other components that will be embedded in this file should return JSX. Ultimately, both index.js and App.js will be used to create a DOM tree for index.html that contains "root" container. Finally, the browser will render the DOM that includes all the elements.

Note that style class tag should be "**className**" instead of "class" to avoid any conflict.

How does 'npm start' command open the default React page?

To understand the sequence of commands, open the default page App.js, open the "package.json" file, and look for "start": "react-scripts start" in the scripts section. When you type "npm start", "react-scripts" will start "start.js" which will start "index.js", which also starts "App.js".

Why does "npm start" start a web server for the React application?

The web server is only used during the development for the purpose of easy testing of React applications. This server is not necessary when the application is deployed.

Key elements of React application

- **React elements** that are the smallest unit in JavaScript representation of HTML DOM. React API, React.createElement is used to create React elements.
- **JSX** (JavaScript XML) that is a JavaScript extension to design UI. JSX is an XML based, extensible language supporting HTML syntax. JSX can be compiled to React Elements and used to create UI. With JSX, you can use conditional statements in an HTML page (**for conditional rendering**).

- **React component** that is the primary building block of the React application. It uses React elements and JSX to design its UI. It is basically a JavaScript class (extends the "React.component" class) or a pure JavaScript function. React component has properties, state management, life cycle, and event handler.

React elements and JSX

Consider the following HTML file with React code embedded in <script> tag:

```
<!DOCTYPE html>
<html>
   <head><title>React Application</title></head>
   <body>
      <div id="react-app"></div>
      <script src="https://unpkg.com/react@17/umd/react.development.js" crossorigin></script>
      <script src="https://unpkg.com/react-dom@17/umd/react-dom.development.js" crossorigin></script>
      <script language="JavaScript">
         element = React.createElement('h1', {}, 'from React element created!')
         ReactDOM.render(element, document.getElementById('react-app'));
      </script>
   </body>
</html>
```

In this HTML file, **React.createElement** takes arguments (tag, attributes, content) and creates React elements that is JavaScript representation of DOM. **ReactDOM.render** takes arguments (HTML code, HTML element), creates a **virtual DOM** including the following HTML tags, and synchronizes it with the actual HTML DOM used by the browser for final rendering.

<h1>from React element created!</h1>

This process occurs when the browser executes the React code (basically JavaScript code after compilation). The browser will render the HTML DOM for the HTML page.

The following React code using **JSX** will produce the equivalent HTML tags to the previous code:

```
<!DOCTYPE html>
<html>
   <head><title>React Application</title></head>
   <body>
      <div id="react-app"></div>
      <script src="https://unpkg.com/react@17/umd/react.development.js" crossorigin></script>
      <script src="https://unpkg.com/react-dom@17/umd/react-dom.development.js"
crossorigin></script>
      <script src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
      <script type="text/babel">
         ReactDOM.render(
            <div><h1>from React element created!</h1></div>,
            document.getElementById('react-app')
         );
```

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

JSX is a syntactic extension of JavaScript that allows you to directly write HTML within JavaScript code in React. JSX file is also like an **HTML template** that allows changing the content dynamically. The file extension for JSX file can be either **.js** or **jsx**. In the above example, ReactDOM.render takes HTML tags as an argument. This means that **JSX is simply syntactic sugar for creating specific JavaScript objects** as JSX code will be transformed to **React.createElement()**. In React, all the HTML tags are converted to JavaScript code since the browser cannot understand React code. JSX is very convenient to the developers and can improve their coding productivity as generating HTML using JavaScript is a very long and tedious job.

React components

Components are reusable codes that can be a building block for larger components or applications. React components allow developers to break down the UI into independent and reusable codes and think of each component in isolation for easier development and maintenance. For example, components can be created for NavBar, SideBar, table, lists, dropdowns, select buttons, etc. You can find many pre-built UI components (e.g., material design). React apps can also include a **third-party component** for a specific purpose, such as routing, animation, state management, etc. React has two types of components, **Class components** and **Function components**.

Class component will look like this:

```
//App.js
class Car extends React.Component {
    render() {
        return <h1>This is a class component.</h1>;
    }
}
export default Car;
```

Functional component will look like this:

```
//App.js
function Car() {
    return <h1>This is a function component.</h1>;
}
export default Car;
```

Components contain an **HTML template** and program logic to dynamically change the HTML and render to the DOM. So, a component **requires a render() method** to return HTML (or DOM).

Class component has render() function. In React, a function can be a component, which is not surprising since a function in JavaScript is an object. A **Functional component does not** include **render() function**. However, everything defined within the function body is the render function that ultimately returns JSX that should be rendered using **ReactDOM.render()**.

VS Code shortcut keys for React template codes

Open a new React file, type “r” character in VS Code editor panel, see a list of templates (e.g., rcc, rce, rcep, etc.), choose one of them, and press enter. VS Code will create a Rect template code for you.

Variables in JSX to dynamically change the content

```
//App.js
import './App.css';
function App() {
  const title = "Welcome";
  const person = {name: 'Bob', age: 30}; //this is an object
  const goog = "http://www.google.com";
  return (
    <div className='App'>
      <div className='content'>
        <h3> {title} </h3>
        <p> {person.name} is {Math.random() * person.age} years of old.</p>
        <a href= {goog}>Google</a>
      </div>
    </div>
  );
}
export default App;
```

Conditional template or rendering

```
function Mailbox(props) {
  const unreadMessages = props.unreadMessages;
  return (
    <div>
      <h1>Hello!</h1>
      {unreadMessages.length > 0 &&
        <h2>
          You have {unreadMessages.length} unread messages.
        </h2>
      }
    </div>
  );
}
const messages = ['React', 'Re: React', 'Re:Re: React'];
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Mailbox unreadMessages={messages} />);
```

Review more examples for conditional rendering here.

Multiple components combined

To create a React app, you can create multiple components, each of which represents a building block of a web page. All the React components **should be structured in a tree** format with **one root component**. React app

starts with a single root component. The root component is built using one or more components. Each component can be nested and composed with other components to any level, logically creating a component tree. For example, an app with a root component (App.js), Navbar component (Navbar.js), Sidebar component (Sidebar.js), and MainBody component (MainBody.js) should form a tree structure (or be nested) like this:

```
//App.js
import Navar from "./Navbar";
import Sidevar from "./Sidebar";
import MainBody from "./MainBody";
function App() {
  return (
    <div className="App">
      <Navar />
      <Sidevar />
      <div className="content">
        <MainBody />
      </div>
    </div>
  );
}
export default App;
```

React Props to pass arguments to React components for code reuse

Props (properties) in JavaScript is an **object** (key-value pair). So, even if you pass multiple parameters, you can access it using the Props object. Likewise, Props are **arguments** that can be passed into React components **through HTML attributes**. Like in JavaScript, you can also pass functions to React Props as a function is an object.

The following examples show how to pass arguments to components:

```
//App.js
import ReactDOM from 'react-dom/client';
function Car(props) {
  return <h2>My car is {props.model}!</h2>;
}
const myElement = <Car model="Mustang"/>; //like a function call passing a parameter value “Mustang”
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
export default Car;
```

Passing parameters using map function

```
//App.js
import ReactDOM from 'react-dom/client';
function Car(props) {
  return <li>{props.brand}</li>;
}
function AllMyCars() {
```

```
  const cars = ['Ford', 'Mercedes', 'TESLA'];
  return (
    <>
        <h1>Here are all my cars:</h1>
        <ul>{cars.map((car) => <Car brand={car} />)}</ul>
    </>
  );
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<AllMyCars />);
export default AllMyCars;
```

Event and event handling

The following example shows how to use variables to dynamically change the HTML content and event handling in React.

```
//App.js
import './App.css';
function App() {
  let title = "Welcome";
  const person = {name: 'Bob', age: 30}; //this is an object
  const goog = "http://www.google.com";

  const clickHandler1 = () => {
    console.log("Event 1 was fired.");
  }
  const clickHandler2 = (msg) => {
    console.log(msg);
  }
  const clickChangeTitle = () => {
    title = "Hello";
    console.log("Hello");
  }
  return (
    <div className='App'>
        <h3> {title}</h3>
        <p> {person.name} is {Math.random() * person.age} years of old.</p>
        <a href= {goog}>Google</a><hr/>
        <button onClick={clickHandler1()}>Handler1</button>
        <button onClick={clickHandler2("Is event 2 fired?")}>Handler2</button>
        <button onClick={clickChangeTitle}>Change to Hello</button><br/>
        <button onClick={clickHandler1}>Handler3</button>
        <button onClick={() =>{clickHandler2("Is event 2 fired through inline?")}}>Handler4</button>
    </div>
  );
}
export default App;
```

{variable_name} will be replaced by the actual value evaluated from the variable. If a variable is a function call, it will replace it with the return value of the function and load the HTML. If the variable is the name of a function (in this case, a function pointer), the function will be called when you click the button.

To see the console messages printed by onClick, open the developer's debugger tool, then shows the following messages:

```
Event 1 was fired.
Is event 2 fired?
Event 1 was fired.
Is event 2 fired?
>
```

Why does it print the same message twice? It's because `<React.StrictMode>` index.js. **<StrictMode>** is a tool for highlighting potential problems in an application, only used by the developers. It is not rendered any visible UI, but it updates the state twice. If you remove the tag in "index.js" file, the above program will print only one message from each of "Handler1" and "Handler2" buttons.

If you click "Handler3" or "Handler4" button, it will print the corresponding message **once only when you click it**.

If you click the buttons "Handler1", "Handler2", and "Change to Hello," nothing happens. That's because those three events were already fired, and the final rendering was done. Note that React is a **declarative framework**; therefore, **React renders the component only once**. Components will get **re-rendered only when the state or props are changed** and **when the parent component is re-rendered**.

To correct this problem, we use a **React Hook**, called "**useState**". Here is an example code that utilizes useState Hook:

```
//App.js
import './App.css';
import { useState } from "react"; //importing useState Hook
function App() {
  //let title = "Welcome";
  const [title, setTitle] = useState("Welcome"); //initializing state value
  const person = {name: 'Bob', age: 30}; //this is an object
  const goog = "http://www.google.com";
  const clickHandler1 = () => {
    console.log("Event 1 was fired.");
  }
  const clickHandler2 = (msg) => {
    console.log(msg);
  }
  const clickChangeTitle = () => {
    //title = "Hello";
    setTitle("Hello"); //resetting new state value
  }
  return (
    <div className='App'>
        <h3> {title}</h3>
        <p> {person.name} is {Math.random() * person.age} years of old.</p>
        <a href= {goog}>Google</a><hr/>
        <button onClick={clickHandler1()}>Handler1</button>
```

```
        <button onClick={clickHandler2("Is event 2 fired?")}>Handler2</button>
        <button onClick={clickChangeTitle}>Change to Hello</button><br/>
        <button onClick={clickHandler1}>Handler3</button>
        <button onClick={() =>{clickHandler2("Is event 2 fired through inline?")}}>Handler4</button>
    </div>
  );
}
export default App;
```

Now click "Change to Hello" button and see if the title is changed to "Hello" from "Welcome".

Destructuring assignment in JavaScript
import {useState} from "react" is an example of **destructuring assignment** to useState from "react" as useState is a named export.

Destructuring assignment in JavaScript makes it possible to unpack values from arrays or properties from objects into distinct variables, e.g., the following code will display 'ford':

```
const cars = ['tesla', 'ford'];
const [car1, car2] = cars;
console.log(car2);
```

What is "state" in React?

The **state** is **a built-in React object** that is used to **contain data about the component**. The state object can be accessed using this.state and updated using this.setState(). A component's state can change over time; **whenever it changes, the component re-renders**. The change in component state can happen as a response to user actions or system-generated events. These changes determine the behavior of the component and how it will render.

State in Class component
The state object, "**this.state**" in a Class component is initialized in the constructor function. The state object can store multiple properties. The method "**this.setState**" is used to change the value of the state object. State changes in your application will be applied to the virtual DOM first. When the virtual DOM gets updated, React compares it to a previous snapshot of the virtual DOM and then only updates what has changed in the real DOM.

An example using the state object in a Class component

```
//App.js
import React from 'react';
import ReactDOM from 'react-dom/client';

class Car extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      brand: "Ford",
      model: "Mustang",
    };
```

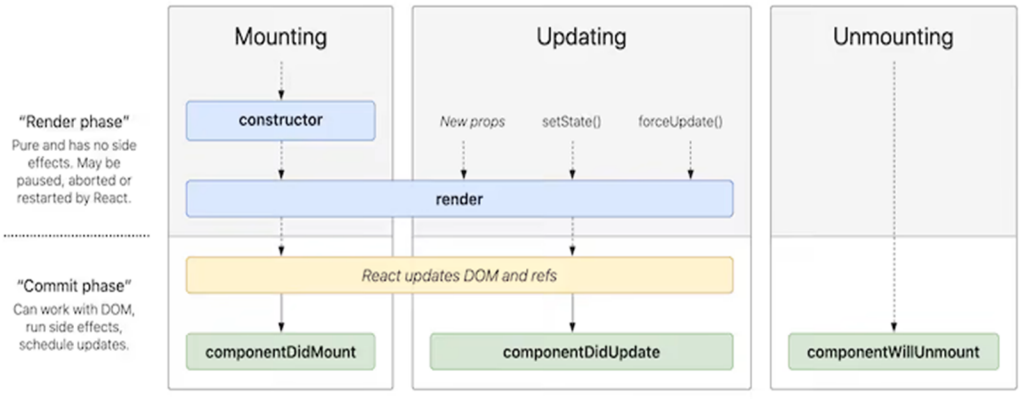
```
  }
  changeModel = () => {
    this.setState({model: "Explorer"});
  }
  render() {
    return (
      <div>
        <h1>My car is {this.state.brand} {this.state.model}.</h1>
        <button type="button" onClick={this.changeModel}>Change model</button>
      </div>
    );
  }
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Car />);
export default Car
```

This page explains the component state in detail with more examples. This page introduces the concept of state and lifecycle in a React component.

Lifecycle of React components

Each component in React has a lifecycle that you can monitor and manipulate during its three main phases:

- **Mounting phase** puts HTML elements into the **VirtualDOM**. During this phase, built-in methods, *constructor()* with the "props" argument, *getDerivedStateFromProps()*, *render()*, and *componentDidMount()* can be called. The *render()* method outputs the HTML to the virtual DOM. The *render()* method is required and will always be called while others are optional. Those built-in methods are called in the order, e.g., *componentDidMount()* is called after render().
- **Updating phase** updates a component when there is any change in the component's state and props. Built-in methods, *getDerivedStateFromProps()*, *shouldComponentUpdate()*, *render()*, *getSnapshotBeforeUpdate()*, and *componentDidUpdate(0* are called. Again, the *render()* is required while the others are optional.
- **Unmounting phase** removes a component from the virtual DOM (unmounting). One built-in method, *componentWillUnmount()* is called when a component is unmounted.



State in Function component

Since the React 16.8, React allows **function components to access state and other React features,** called "React Hooks". This means that, with Hooks, you don't have to use class components to manage the state in React.

What is "Hooks" in React?

The "**React Hooks**" are special functions that let you "hook into" **lifecycle React features from the function component**. Hooks allow you to use state and other features **without using the Class component**. Some of the **built-in Hooks** are useState, useEffect, useContext, useRef, useReducer, useCallback, useMemo, etc.

Why do we need to use state in React app?

We've already seen a situation where we want to dynamically change the content before rendering, which is not possible once rendering is completed by the browser. We should be able to access and change the component states before rendering. Hook state is the new way of declaring, setting, and retrieving a state in a function component using **useState()** functional component.

Rules of Hooks

- Only call Hooks at the top level (not in nested code such as loops, conditions, or another function body)
- Only use in React functions, not React classes

Different ways of creating, setting, and using states

```
import {useState} from "react";
import ReactDOM from "react-dom/client";
function MyCar () {
    const [color, setColor] = useState("Blue");
    const [carType, setType] = useState("SUV");
    const [car, setBrand] = useState({
        brand: "TSLA",
        year: "20222"
    })
    return (
      <>
        <h1>I want to buy a {color} {car.year} {car.brand} {carType}.</h1>
        <button type="button" onClick={() => setType("Sedan")}>Change to Sedan</button>
        <button type="button" onClick={() => setColor("Red")}>Change to Red</button>
        <button type="button" onClick={() => setBrand({brand: "Ford", year: "2022"})}>Change to Ford</button>
      </>
    )
}
const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(<MyCar />);
export default MyCar;
```

Calling **React.useState** inside a function component generates a state associated with that component. The **useState()** returns a state value and a function to update the state. Both useState() and setState() are

asynchronous functions. Therefore, they do not update the state immediately, instead, they are queued to update the state object.

Common use cases of useState and props
Refer to the Tutorial #11-13 for common use cases of useState and props to reuse React components.

What if I want to keep track of the state of a component and execute a code whenever the component state changes or the major component events (such as component mount, update, and unmount) **in the component lifecycle occur, as shown below?**

You can **call the built-in component lifecycle methods** such as componentDidMount(), componentDidUpdate(), componentWillUnmount(), etc. The following code shows an example of calling some of those methods:

```
//App.js
import React from "react";
class App extends React.Component {
    componentDidMount() {
      // Runs after the first render() lifecycle
    }
    render() {
      console.log('Render lifecycle')
      return <h1>Hello</h1>;
    }
  }
  export default App;
```

For functional components, you can use **useEffect Hook**. The following example code using the useEffect will do the same job as the above code.

```
//App.js
import React from "react";
function App() {
    React.useEffect(() => {
      // Runs after the first render() lifecycle
    }, []);
    return <h1>Hello</h1>;
}
export default App;
```

The useEffect Hook

The **useEffect** is another React Hook. The useEffect is an alternative for the lifecycle method in class components in a functional component. The useEffect does not use components lifecycle methods available in class components such as componentDidMount(), componentDidUpdate(0, and componentWillUnmount(), but it is equivalent to them.

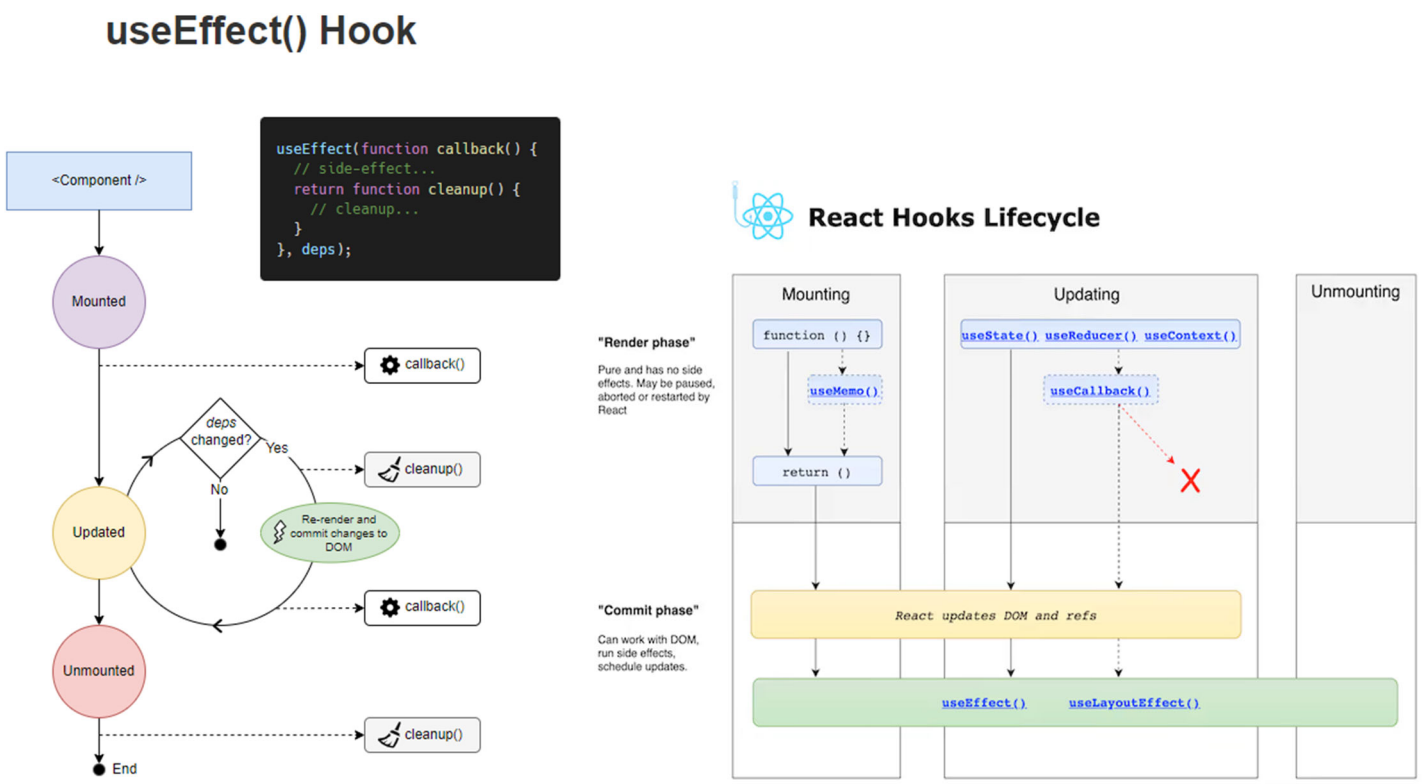
The useEffect() can be used to execute a code during the lifecycle of the component **like an event handler** for the state changes and the lifecycle changes of a component, instead of on specific user interactions.

Using useEffect within a function component is like telling React that the component needs to do something **after render**. React will remember the function (effect) you passed and call it later **after** performing the DOM updates. So, the useEffect() **will not interfere with the component rendering**.

The following code shows an example use case for useEffect:

```
//App.js
import { useState, useEffect } from 'react';

const App = () => {
    const [input, setInput] = useState('');
    const [isValid, setIsValid] = useState(false);

    const inputHandler = (e) => {
        setInput(e.target.value);
    };

    useEffect(() => {
        if (input.length < 5 || /\d/.test(input)) {
            setIsValid(false);
        } else {
            setIsValid(true);
        }
    }, [input]);

    return (
        <div className='App'>
            <h3>Running on state change: validating input field</h3>
            <form>
                <label htmlFor="input">Write more than 5 non numerical characters.</label><br />
                <input type="text" id="input" autoComplete="off" onChange={inputHandler} style={{ height:
'1.5rem', width: '20rem', marginTop: '1rem' }} />
            </form>
            <p><span style={isValid ? { backgroundColor: 'lightgreen', padding: '.5rem' } : { backgroundColor:
'lightpink', padding: '.5rem' }}>{isValid ? 'Valid input' : 'Input not valid'}</span></p>
        </div>
    );
 };
export default App;
```

Placing **useEffect() inside the component** lets us access the state variable. The following diagrams show **how useEffect() Hook works** in the component lifecycle:



The useEffect Hook allows you to **perform side effects** (an action) in your components. In programming (in computer science), an expression or function is said to have a side effect if it modifies some state variables outside its local environment, causing more outcomes than you originally intended. For example, when you pass a reference (pointer) argument to a function and the value of the argument is modified within the function, the function call creates a side effect since the argument value will be changed after the function call. So, **side effects are not predictable** because they are actions that are performed outside the (component) world.

Asynchronous tasks
The **useEffect Hook** is used **to handle side-effects caused by a component's actions** such as fetching data from APIs, attaching event listeners to the document, manipulating the DOM itself, etc. Those actions are generally **asynchronous tasks**, which means **unpredictable completion of the tasks**.

Some useful use cases for useEffect()

- Running only once on mount (when the component is first created): fetch API data, reading from local storage, interacting with the browser APIs (e.g., document or window)
- After state or prop value changes (updating the DOM): validating input field, live filtering, triggering animation, etc.
- Registering events (setting up a subscription) and deregistering event listeners
- Using time functions (eg., setTimeout or setInterval)
- Cleaning up the side effects for the application's performance when the component is destroyed. For example, if you use a timer using the setTimeout(), you need to clean it up by invoking the clearTimeout().

The syntax of useEffect()
useEffect(<callback>, <dependency>)

- <callback> is the function to be executed right after changes were pushed to DOM.
- <dependency> is an optional array of dependencies (state variables). The usEffect() executes <callback> function only if the dependencies have changed between rendering.

Several ways of defining useEffect()

```
useEffect(() => {
    // Runs every time the component renders
})

useEffect(() => {
    // Runs only on initial render
}, []) // Optional second argument: dependency array

useEffect(() => {
    // Runs only when 'apiData' changes
}, [apiData])
```

The first syntax has no dependency variable.
The second syntax has an empty dependency [].
The third syntax has one or more dependency variables passed.

The useEffect without dependencies will execute the useEffect function for every state change.

```
//App.js
import { useState, useEffect } from "react";
import ReactDOM from "react-dom/client";

function Timer() {
  const [count, setCount] = useState(0);
  useEffect(() => {
    setTimeout(() => {
      setCount((count) => count + 1);
    }, 1000);
  });
  return <h1>I have rendered {count} times!</h1>;
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Timer />);
export default Timer;
```

The useEffect will execute only once if an empty dependency parameter [] is provided.

An example with a "prop" dependency variable 'name':

```
//App.js
import ReactDOM from "react-dom/client";
import {useEffect} from "react";

function Greet({ name }) {
    const message = `My name is ${name}.`; // Calculates output. Note that the use of `backquote`
    useEffect(() => {
```

```
        document.title = `Greetings to ${name}`; // Side-effect!
    }, [name]);
    return <div>{message}</div>;       // Calculates output
}
const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(<Greet name="Bob" />)
export default Greet;
```

An example of useEffect that is dependent on one "state" variable:

```
//App.js
import { useState, useEffect } from "react";
import ReactDOM from "react-dom/client";
function Counter() {
  const [count, setCount] = useState(0);
  const [calculation, setCalculation] = useState(0);
  useEffect(() => {
    setCalculation(() => count * 2);
  }, [count]); // <- add the count variable here

  return (
    <>
      <p>Count: {count}</p>
      <button onClick={() => setCount((c) => c + 1)}>+</button>
      <p>Calculation: {calculation}</p>
    </>
  );
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Counter />);
export default Counter;
```

The useEffect() for clean up

```
useEffect(() => {
  /* do something */
  const cleanup = () => {
    /* delete something */
  }
  return cleanup
})
```

An example use case of the useEffect() for clean up when the component is destroyed:

```
useEffect(() => {
  const id = setInterval(() => {
    console.log("This is logged to console.");
  }, 5000);
  const cleanup = () => clearInterval(id)
  return cleanup
}, []);
```

Preventing unwanted component re-rendering using useMemo
Components will get re-rendered only when the state or props are changed and when the parent component is re-rendered. If the child component is re-rendered without any change in its props or state, it could be prevented using hooks called React.memo. **React.memo** is a high-order component that memorizes the result; e.g., React will skip rendering of that component and reuse the last rendered result. For example, assume a function component named "Child" like this:

```
function Child() {
        …
}
export default React.memo(Child);
```

"export default React.memo(Child)" will prevent unwanted re-rendering of components.

Other Hooks:

- **useContext** is a way to manage a state globally by sharing states between nested components.
- **useRef** allows you to persist values between renders. It can be used to track application renders.
- **useReducer** is similar to the useState, but it allows for custom state logic when you need to keep track of multiple states.
- **useCallback** returns a memorized (caching) callback function.
- **useMemo** returns a memorized value.
- **Custom Hooks** are Hooks that may consist of one or more Hook. For example, when you have component logic that needs to be used by multiple components, we can extract that logic to a custom Hook. Custom Hooks begin with "use", e.g., "**useFetch**".

Understanding rendering in React

When an HTML document is loaded into a browser, the browser creates a DOM tree that represents the HTML document and renders it on the browser's window. When there is any change in the same HTML document, the elements in the corresponding DOM should be changed as well. This process is costly (in terms of the browser's performance), especially when an application becomes complex (due to lots of changes occurring on the HTML pages).

The **VirtualDOM** created in React is a strategy to solve the performance problem. The VirtualDOM is created using a React library "ReactDOM". The VirtualDOM represents the actual DOM tree (used by the browser) in memory. React performs calculations using the state and props, decides which elements of the actual DOM should be changed, and try to sync it with the actual DOM for final rendering. For example, when React app calls ReactDOM.render method by passing the UI elements (created using either JSX or React.createElement) to render the UI elements, it processes the JSX or React element and creates the VirtualDOM. The Virtual DOM will be synchronized (or merged) with the actual HTML DOM, and rendered.

Three steps process of rendering

- **Rendering/re-rendering**: When any stage changes occur in a component, React will collect all the components from the root of your application to change the VirtualDOM for re-rendering, without updating the actual DOM yet. JSX code will be transformed to React.createElement().

- **Reconciliation**: After the re-rendering step, React has two versions of React.createElement output. React will determine using a heuristic algorithm which elements need to be represented again.
- **Commit**: After React calculated all the changes, react-dom for the browser and react-native for the mobile platform makes the modification of the actual DOM to the browser or mobile app. After this step, the VirtualDOM and the actual DOM are synchronized.

React Forms to handle the user request

Just like HTML forms, React uses forms to allow users to send a request. Form data is usually handled by the components (meaning that all the data is stored in the component state). You can control changes by adding event handlers such as onChange attribute. You can use the **useState** Hook to keep track of each input value for validation.

The following component, "MyForm" gets data from a user form.

```
//MyForm.js
import {useState} from "react";

function MyForm () {
  const [inputs, setInputs] = useState({});
  const handleChange = (event) => {
    const name = event.target.name;
    const value = event.target.value;
    setInputs(values => ({...values, [name]: value}))
  }
  const handleSubmit = (event) => {
      event.preventDefault(); //will prevent refreshing the page
      alert('A form was submitted: ' + event.target[0].value);
  }
  return (
      <>
      <form id='form1' onSubmit={handleSubmit}>
        <h3>User form</h3>
        <label>Name: <input type="text" name="name" value={inputs.name || ""} onChange={handleChange} />
        </label>
        <input type="submit" value="Submit" />
      </form>
      </>
  );
}
export default MyForm;
```

To store the form data, you use the **useState Hook**. The form data can be accessed via event object, **event.target.value**. You can use it to validate the data and send it to the server later when necessary. In the above example, a separate event handler, "handleChange" was defined. You can also use an inline event to set the value in a useState variable like this:

<input type="text" required onChange={(e)=>setTitle(e.target.value)/>

In the above program, the preventDefault() method is used to prevent the default submit action (sending the form and refreshing the page). **The preventDefault() method prevents** the browser from taking the **default action** that belongs to the event. The preventDefault() method will cancel the event if it is cancelable. Note that not all events are cancelable. This method can be useful when you want to prevent the browser from, for example, submitting a form when the user clicks a "submit" button since it does not meet the required criteria.

In the "App.js", we import "MyForm" component.

```
//App.js
import React from 'react';
import MyForm from './MyForm'

function App() {
  return (
    <div className="App">
      <MyForm />
    </div>
  );
}
export default App;
```

Making GET requests to fetch data from an API in React

There are multiple ways to fetch data from an API, e.g., JavaScript fetch API and libraries such as **Axios**.

State variables to store fetched data
When you request data, you must prepare a state to store the data upon return. If the data does not load, you must provide a state to manage the loading stage to improve the user experience and another state to manage the error. So, you typically need three state variables like this:

```
const [data, setData] = useState(null);
const [loading, setLoading] = useState(null);
const [error, setError] = useState(null);
```

Using fetch API and the useEffect()
JavaScript fetch API has the following syntax:

```
fetch(url, {
        method: "GET" //default, so we can ignore
})
```

Since you typically want to make a data request to fill the state only once when the component is first created, the code will look something like this:

```
const [data, setData] = useState([])

useEffect(() => {
```

```
  fetch(url)
    .then(res => res.json())
    .then(data => setData(data))
}, [])
```

The **fetch()** method in JavaScript returns a **Promise** that is either resolved or rejected. If the promise is resolved, you handle the response using **.then()**. At this stage, the returned data is a response object. The **res.json()** will return the body as a promise with JSON format (you can use other formats). The second .then() is called when the first .then is resolved. At this time, you store the data received from the API in the state variable "data". If not resolved (rejected), we can catch it using **.catch**() like this:

```
.catch((err) => {
        console.log(err.message):
});
```

You can also add a conditional statement to check the status of response right before res.json() like this:

```
if (!res.ok) {
        throw new Error(`An HTTP error status: ${res.status}`);
}
return res.json();
```

For more information about the Fetch API, you can review this page.

A custom React Hook (useFetch)
Assuming the following useEffect will be used in other places, save the following code in "useFetch.js" to create a custom Hook:

```
//useFetch.js
import { useState, useEffect } from "react";

const useFetch = (url) => {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(null);
  const [error, setError] = useState(null);
  useEffect(() => {
    fetch(url)
      .then((res) => {
        if (!res.ok) {
          throw new Error (`HTTP error: ${res.status}`);
        }
        console.log(res); //to see the response object
        return res.json();
      })
      .then((data) => setData(data))
      .catch((err) => {
        console.log(err.message);
        setError(err.message)
```

```
        setData(null);
      })
      .finally(() => { //to settles promise
        setLoading(false);
      });
  }, [url]);
  return [data];
};
export default useFetch;
```

Import “useFetch.js” in the following code. In this example, you will fetch data from an external API (from a free JSON placeholder) that returns JSON data.

```
//App.js
import useFetch from "./useFetch";

const App = () => {
    const [data] = useFetch("https://jsonplaceholder.typicode.com/todos");
    return (
      <div className="App">
        <h3>JSON data fetched</h3>
        {data && data.map((item) => {
            return <p key={item.id}>{item.title}</p>;
         })}
      </div>
    );
  };
export default App;
```

Using the async/await syntax

The previous example uses the pure promise syntax. The async/await syntax is a more elegant method to get data.

You can rewrite useFetch.js using async/await syntax like this:

```
//useFetch.js
import { useState, useEffect } from "react";

const useFetch = (url) => {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(null);
  const [error, setError] = useState(null);
  useEffect(() => {
    const getData = async () => {
      try {
        const res = await fetch(url);
        if (!res.ok) {
          throw new Error (`HTTP error: ${res.status}`);
```

```
        }
        console.log(res); //to see the response object
        let jsonData = await res.json();
        setData(jsonData);
        setError(null);
      } catch(err) {
        setError(err.message);
        setData(null);
      } finally {
        setLoading(false);
      }
    }
    getData();
  }, []);
  return [data];
};
export default useFetch;
```

Using the fetch API in a class component using componentDidMount()

```
//App.js
import React from "react";
import './App.css';
class App extends React.Component {
    constructor(props) {
        super(props);
        this.state = {
            items: [],
            DataisLoaded: false
        };
    }
    // ComponentDidMount is used to execute the code
    componentDidMount() {
        fetch("https://jsonplaceholder.typicode.com/users")
            .then((res) => res.json())
            .then((json) => {
                this.setState({
                    items: json,
                    DataisLoaded: true
                });
            })
    }
    render() {
        const { DataisLoaded, items } = this.state;
        if (!DataisLoaded) return <div>
            <h1> Pleses wait some time.... </h1> </div> ;
```

```
        return (
          <div className = "App">
            <h1> Fetch data from an api in react </h1>  {
                items.map((item) => (
                <ol key = { item.id } >
                    User_Name: { item.username },
                    Full_Name: { item.name },
                    User_Email: { item.email }
                    </ol>
                ))
            }
          </div>
        );
    }
}
export default App;
```

Another async/await version using AbortController for a useEffect cleanup

```
//App.js
import { useEffect, useState } from "react";

function App() {
  const [users, setUsers] = useState(null);
  const [isPending, setIsPending] = useState(true);
  const [error, setError] = useState(null);
  const url = "https://jsonplaceholder.typicode.com/todos/";
  const abortCont = new AbortController();

  const getData = async () => {
    try {
      const res = await fetch(url, {signal:abortCont.signal});
      if (!res.ok) {
        throw new Error (`HTTP error: ${res.status}`);
      }
      let jsonUsers = await res.json();
      setIsPending(false);
      setError(null);
      setUsers(jsonUsers);
      console.log(res);
    } catch (err) {
      if (err.name === 'AbortError') {
        console.log("Fetch aborted")
      } else {
        setIsPending(false);
        setError(err.message);
```

```
        console.log(err.message);
      }
    } finally {
      console.log("Done!") //just printing a console message, but you can process any remaining work.
    }
    //useEffect cleanup
    return () => abortCont.abort(); //to abort fetch if the component is unmounted
  };

  useEffect(() => { // comment out this line to test the useEffect
    getData();
  }, [url]); // comment out this line

  return (
    <div className="app">
      {users && users.map((user) => (
          <div className="item-container">
            Id:{user.id} <div className="title">Title:{user.title}</div>
          </div>
        ))}
    </div>
  );
}
export default App;
```

The above program shows an example of useEffect cleanup using AbortController. Consider a situation where this program runs after the component is unmounted. Obviously, you will get a fetch error. Another situation is when you want to abort the request for some reasons, e.g., it takes too much time. How can you handle this situation in fetching data? The solution for this situation is to **use the** abort() method **to abort the fetch request**. The abort() method of the AbortController interface abort a DOM request before it has been completed. The return statement in useEffect() is an example of **useEffect cleanup**.

```
return () => abortCont.abort();
```

To understand the need for useEffect, comment out the lines for useEffect() in the above program, open the developer's debugging tool from your browser, and see what's happening. Not using useEffect will cause the fetch() method to get called infinitely. That's because each fetch() changes the state variable "users" which in turn causes the component to re-rendering. That also causes the state change, and so on. What if you don't use the state variable? If you don't use the useState Hook, we've already seen that you won't be able to access the state during the component cycle because once the rendering is done, React won't re-render since no state has been changed.

Fetching data using the third-party library Axios

Axios is a promise-based HTTP client that connects to an endpoint. Unlike fetch() method, the response returned from this library contains the JSON format. It also has the advantage of robust error handling, so, we don't need to check and throw an error like we did earlier with the fetch() method. The following codes show the difference:

//with fetch API

```
fetch(url)
.then((res) => res.json())
.then((data) => console.log(data))
.catch((error) => console.log(error));
```

//With Axios

```
axios.get(url) //matching the HTTP methods
.then((res) => console.log(res))
.catch((error) => console.log(error));
```

To use Axios, install axios package by 'npm i axios --save' and try the following example:

```
//App.js
import axios from "axios";
import { useState, useEffect } from "react";

export default function App() {
    const [data, setData] = useState(null);
    const [loading, setLoading] = useState(null);
    const [error, setError] = useState(null);
    const getData = async () => {
        try {
            const res = await axios.get("https://jsonplaceholder.typicode.com/posts?_limit=10");
            setData(res.data);
            setError(null);
        } catch (err) {
            setError(err.message);
            setData(null);
        } finally {
            setLoading(false);
        }
    };

    useEffect(() => {
       getData();
    }, [])

    return (
        <div>
            <h1>API posts using axios</h1>
            {loading && <div>Loading...</div>}
            {error && (<div>`Error: {error}`</div>)}
            <ul>{data && data.map(({id, title}) => (
                <li key={id}> <h4>{title}</h4></li>
```

```
            ))}
            </ul>
        </div>
    );
}
```

For more information about Axios package, refer to this page.

Using the useFetch custom Hook from react-fetch-hook library
To use react-fetch-hook, install it package by 'npm i react-fetch-hook --save'.

```
//App.js
import useFetch from "react-fetch-hook";

export default function App() {
  const { isLoading, data, error } = useFetch(
    "https://jsonplaceholder.typicode.com/posts?_limit=10"
  );
  return (
    <div className="App">
      <h1>API Posts</h1>
      {isLoading && <div>A moment please...</div>}
      {error && (
        <div>{`There is a problem fetching the post data - ${error}`}</div>
      )}
      <ul>
        {data && data.map(({ id, title }) => (
            <li key={id}><h3>{title}</h3></li>
          ))}
      </ul>
    </div>
  );
}
```

For more information about react-fetch-hook, refer to this page.

So far, we have installed several additional libraries for different purposes.

How do I install all the required dependencies with npm for production?

'npm install' will resolve all the required dependencies form the package.json file and install them in the 'node_modules' folder. **Make sure you add 'node_modules' in .gitignore file** to avoid uploading all the files in this directory. You can manually edit 'dependencies' field in package.json file manually or use "**--save**" option whenever you install modules to save the dependency entry in package.json file.

Routing using React Router

Typically, most applications or websites have multiple pages. How do we serve multiple pages by the user request? We have to add a routing feature to our application. **Routing** is the ability to move different pages or parts of an application when a user enters a URL or clicks an element (e.g., button, link, etc.) of the page. For example, assume there are three pages in different directories hosted on "http://example.com" as follows:

/index.html /about/about.html /user/login.html

Server-side routing
When the user enters the URL "http://example.com/about.html", typically, the web server will find "about.html" in "about" directory, send it to the browser, and the browser load (render) the page. If the user request other pages, the server will find the other pages, send them to the browser, and the browser will **load the whole new page**. Here the ability process (or ability) to find the right pages is called routing in the server-side or server-side routing (typically coincides with **server-side rendering**).

Client-side routing
Server-side routing requires lots of communication (request and response) between the browser and the server, which impacts the scalability of the application. What if all the required page elements are already loaded in the browser? In this case, there is no need to communicate with the server any longer except for some data stored in the server. This approach will improve the overall performance as well as the scalability. A Single Page Application (SPA) works like this as it loads only a single page in the browser, then updates the body content of the page via APIs without loading whole new pages from the server (typically coincide with **client-side rendering**). This approach provides the users with faster and smooth responses as well as a performance gain in the overall application. The main disadvantage of this approach (or client-side rendering) is that search engine optimization (SEO) is required to do more work, while with server-side rendering, all the HTML content is present in the source code, allowing search engine easier crawling.

Routing in React app
React is a framework for SPA. When the user makes an initial request (index.html), the server finds the file, "index.html", and sends it with a compiled JavaScript bundle (created from "build") to the browser. From that point on, for any additional user requests, with React Router, React will intercept the request to load the proper component and handle the routing using **React Router** within the browser without requesting to the server except for the data on the server-side. React Router enables the navigation among views of various components in a React application and allows changing the browser URL. React Router is a fully featured client- and server-side routing library for React. It can run on the client-side, server-side (with node.js), or on React Native.

Install "react-router-dom" package
```
npm i -D react-router-dom --save
```

"-D" is the shortcut for "--save-dev". With this option, the package will appear in your devDependencies, which means that the package will not be installed if you do "npm install". More information about package installation can be found here.

Import the react-router component
```
Import {BrowserRouter, Routes, Route} from "react-router-dom";
```

To create an application with multiple page routes, create a folder named "**pages**" under "src" folder, add several JavaScript files for React components. We will use our Router in index.js as follow:

```
//index.js
import ReactDOM from "react-dom/client";
import { BrowserRouter as Router, Routes, Route } from "react-router-dom";
import Layout from "./pages/Layout";
import Home from "./pages/Home";
import Blogs from "./pages/Blogs";
import Contact from "./pages/Contact";
import NoPage from "./pages/NoPage";

export default function App() {
  return (
    <Router>
      <Routes>
        <Route path="/" element={<Layout />}>
          <Route index element={<Home />} />
          <Route path="blogs" element={<Blogs />} />
          <Route path="contact" element={<Contact />} />
          <Route path="*" element={<NoPage />} />
        </Route>
      </Routes>
    </Router>
  );
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<App />);
```

The above example shows we can wrap the routing paths in a page. Try to test this page with different endpoints (different URLs) or review this page to understand how it works.

<Route> can be nested. The first <Route> has a path of "/" and renders the "Layout" component. The nested <Route> inherits and adds to the parent <Route>. The "Home" component route does not have a path but has an "index" attribute. That specifies this route as the default route for the parent route, "/". Setting the "path" to "*" will act as a catch-all for any undefined URLs (which is good for an error page).

Router Link

A **<Link>** in React is an element that lets you navigate to another page by clicking it like <a> in HTML, but internally. Anytime we link to an internal path of a page, <Link> is used instead of <a href="">. A relative <Link to=""> value (that doesn't begin with "/") resolves relative to the parent route. This means that it builds upon the URL path that was matched by the route that rendered that <Link>. In <Link to="value">, the value can be dynamically changed, e.g., **<Link to={user.id}>**. To use <Link>, we have to import it from "react-router-dom":

import {Link} from "react-router-dom";

Handling 404 status code for a requested page not found

You can create a component to handle 404 status code like this:

```
//NotFound.js
import { Link } from "react-router-dom"

const NotFound = () => {
  return (
    <div className="not-found">
      <h2>Sorry</h2>
      <p>That page cannot be found</p>
      <Link to="/">Back to the home...</Link>
    </div>
  );
}
export default NotFound;
```

Then, you import it in the main page, e.g., "App.js" like this, catching all the remaining endpoints:

import NotFound from './NotFound';
...
<Route path="*"> <NotFound /></Route>
...
export default App;

Additional React-Router Hooks

- **useHistory** lets you access the history instance used by React Router. The history instance created by React Router uses a stack (called history stack), that stores all the entries the user has visited. The history.push("/path") adds the given "path" or URL to the history stack, which results in redirecting the user to the given path. This method is very useful when you want to return to a specific page from other pages. There are other methods and properties for useHistory Hook.
- **useParams** lets you access the parameters of the current route.
- **useLocation** returns the location object that represents the current URL.
- **useRouteMatch** returns a match object that contains all the information like how the current URL matched with the Route path.

Review this page for more information about the React router.

Making a POST request in React

In the previous example of fetch(), we used the GET method. You can also use the POST method with fetch API to make a request to the server. The GET and POST are different types of HTTP request methods. The HTTP protocol supports several other request methods. With the GET request method, data is being requested from a specific resource (represented by a URL or query string while with the POST request method, data is being sent through the body of the request to a specific resource.

The POST method is widely used to submit form data to the server, especially to submit a large amount of data or binary data. The POST method requires additional parameters with the request, such as method, headers, etc. The following code shows an example of the POST method:

```
//App.js
import { useState } from "react";

function App() {
    const [name, setName] = useState('Enter your name here');

    const handleSubmit = (e) => {
      e.preventDefault();
      const jsonName = {name};
      const url = "http://localhost:3000";
      //const url = "https://jsonplaceholder.typicode.com/posts";
      fetch(url, {
        method: 'POST',
        headers: {"Accept": "application/json", "Content-Type": "application/json" },
        body: JSON.stringify(jsonName)
      }).then(() => {
        console.log("Name " + name + ' is posted.');
      });
    }

    return (
      <div className="App">
        <h2>Posting a name</h2>
        <form onSubmit={handleSubmit}>
          <input type="text" required value={name}
            onChange={(e) => setName(e.target.value)}
            onFocus={(e) => setName('')}
          />
          <button>Submit</button>
        </form>
      </div>
    );
  }

export default App;
```

Posting data in JSON format is common. The **JSON.stringify()** method converts a JavaScript object or value to a JSON string. For example, console.log(JSON.stringify({x: 1, y: 3})); will print "{"x": 1, "y": 3}".

However, if you open the developer's tool to see the console message from the fetch() API using the POST method, you will notice the status code "404 (Not Found)". If you replace the URL "localhost" with "jsonplaceholder" (commented out in the above program), the problem will be fixed. Why? We will discuss the reason and how to resolve this issue later in this lesson when we discuss APIs and backend development.

Refer to this page for more information about HTTP response status codes.

Learning more complex use cases of React from example projects

- Example React projects here
- More example projects

How can I test and build my React app for continuous integration?

The create-react-app also makes it simple to test your React app as it includes all the packages you need to run tests using the React Testing Library (@testing-library/react). A basic test is included in the file 'App.test.js' in 'src' folder. You can run tests with the command 'npm run test'. It will test all files with the extension '.test.js'.

How can I bundle all JavaScript and React files and build them for production?

Webpack is a module bundler. It relies on a dependency graph and modularization that describe how modules relate to each other using the references (require or import statements) between files. This is especially useful for Single Page Applications (SPAs) as it allows downloading only one bundled file, e.g., 'bundle.js'.
To use Webpack, install the package if it is not installed:

```
npm install webpack-cli --save-dev
```

When you are ready to deploy to production, bundle the app into static files and create an optimized build of your app in the build folder for production.

```
npm run build
```

This command will create an optimized product build for a project and will output what files it has generated and how large each file is since the size of JavaScript impacts the performance. It will create a bundle in '**build**' directory.

To run the React app locally using the package '**serve**' to detect any errors before pushing live to the Web:

```
npx serve
```

How can I deploy my React application?

Pushing a react app to GitHub

1) Open .gitignore file and make sure you add '/node_modules' to avoid pushing all the React modules.
2) Create a project repository on GitHub, e.g., 'myreact'.
3) Copy the address of the remote repository.
4) Move the current directory to 'myreact' (e.g., cd myreact or choose 'myreact' folder in VS code).
5) Open a Terminal and create a local repository for 'myreact' by:

   ```
   git init
   git remote add origin 'address_of_remote_repository'
   git add .
   git commit -m 'any message'
   git push -u origin master
   ```

   If you want to change the name of the local branch name from 'master' to 'main', type 'git branch -m master main'.

Or you create a repository on GitHub first, then clone it in your local project directory.

Deploying React app to Heroku cloud from GitHub repository

1) Push the code of your application to the GitHub repository.
2) **Create a new app at Heroku**, e.g., 'myreact-heroku'
3) **Add a buildpack at Heroku** from **settings** by entering the URL 'https://github.com/mars/create-react-app-buildpack'.
4) Choose 'GitHub' as a deployment method.
5) Connect Heroku to Github at Heroku by entering the GitHub repository name, 'myreact-heroku'. If connected, now you can deploy any branch you want to deploy. Let's choose the main (or master) branch.
6) Choose an option for automatic or manual. Let's choose 'Automatic deploys' and deploy.

Deploying React app using Heroku CLI:

- Login to Heroku, 'heroku login'
- Connect the local repository to the remote Heroku repository, 'git:remote -a myreact-heroku'
- Assuming a Heroku application is already created, push the code to Heroku, 'git push Heroku master'

Deploying React app to AWS

Refer to the tutorials for deploying React applications to AWS using AWS Amplify.

Can I create a mobile application using React?

React Native lets you build real mobile applications using only JavaScript (not web pages). It uses the same design as React. React Native uses native components instead of web components as building blocks (https://reactnative.dev/docs/tutorial). **To create a mobile app, install the following packages**:

- create-react-native-app
- react-native-cli
- Android Studio