# UNIT -4

# React

## Page Setup:

To set up a React page, you'll need to follow a series of steps. Here's a basic outline of the process:

1. Install Node.js: React applications require Node.js, so make sure you have it installed on your system. You can download the latest version from the official Node.js website (https://nodejs.org).
2. Create a new React project: Open a terminal or command prompt and navigate to the directory where you want to create your React project. Run the following command to create a new project using CreateReactApp: **npx create-react-app my-app**

   Replace "my-app" with the desired name of your project. This command will set up a new React project with the necessary files and dependencies**.**
3. Navigate to the project directory: After the project is created, change to the project directory by running the following command.

   **cd my-app**

   Replace "my-app" with the desired name of your project.
4. Start the development server: To launch the development server and see your React page in action, run the following command**:**

   **npm start**

   This command will start the development server and open your React page in your default browser. You can make changes to your code, and the page will automatically reload to reflect the updates.
5. Customize your React page: Open your project in a code editor of your choice. The main file you'll work with is **src/App.js**, which contains the root component of your React application. You can modify this file or create additional components in the **src** directory to build your desired page.
6. Add CSS and styles: React supports various ways to style your components, including CSS files, inline styles, or CSS-in-JS libraries like styled-components. You can add styles to your components based on your preferences.
7. Deploy your React page: Once you've built your React page, you can deploy it to a hosting platform of your choice. Popular options include Netlify, Vercel, GitHub Pages, and AWS Amplify. Each platform has its own deployment process, so refer to their documentation for detailed instructions.

## React Elements

React elements are the basic building blocks of React applications. They are JavaScript objects that represent a virtual representation of a DOM element, which can be rendered to the actual DOM by React.

- 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 element is the smallest **renderable** unit available in React. We can render such elements using **ReactDom.**
- The browser DOM is made up of DOM elements. Similarly, the React DOM is made up of React elements.
- React elements are different from DOM elements as React elements are simple **Javascript** objects and are efficient to create.

**Here are some important notes about React elements**:

- React elements are lightweight and efficient: as they do not represent actual DOM nodes until they are rendered, and can be created and manipulated entirely in memory. This makes them a key part of React's performance optimizations.
- React elements are immutable: Once an element is created, you cannot change its properties. Instead, you must create a new element if you want to change its appearance.
- React elements are declarative: You tell React what you want to see, and it handles the rest. React updates the DOM to match the state of your components.
- React elements are JSX: JSX is a syntax extension for JavaScript that allows you to write HTML-like code in your JavaScript files. JSX is not required to use React, but it is the recommended way to write React components.
- React elements have a type and props: The type of a React element is a string or a function that returns another React element. The props are an object that contains the properties and values that you pass to the element.
- React elements are rendered to the DOM: To render a React element to the DOM, you must use the **ReactDOM.render()** method.
- React elements can be composed: You can compose multiple React elements to create more complex components. This makes it easy to build reusable UI components.

Overall, React elements are a powerful way to build user interfaces in a declarative, composable, and efficient manner.

## Creating React Elements:

To create React elements, you can use the createElement method or JSX syntax.

**1. Using createElement()** :

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

Parameters:

- The first argument defines the type of element we want to create. **type:** The type argument must be a valid React component type. For example, it could be a tag name string (such as 'div' or 'span'), or a React component (a function, a class, or a special component like Fragment) In this case, we want to create an h1 element.
- The second argument represents the element's properties. **props**: The props argument must either be an object or null. If you pass null, it will be treated the same as an empty object. React

will create an element with props matching the props you have passed. In this case, This h1 currently has an id of recipe-0.

- The third argument represents the element's children: any nodes that are inserted between the opening and closing tag (in this case, just some text).

2. **Using JSX syntax**:

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

1.**JSX** is a syntax extension for JavaScript that allows you to write HTML-like code in your JavaScript files. The above code will be compiled to the same "createElement()" call. It is commonly used in React to define the structure and appearance of components.

2.Import React: To use JSX, you need to import the React library at the beginning of your file. You can do this by adding the following line of code**: import React from 'react'**;

3.Component Rendering: JSX allows you to render React components as elements. You can create a component using a function or a class and then use it within JSX by simply invoking it as a tag.

For example:

```
function MyComponent() {

 return <div>Hello, World!</div>;

}

ReactDOM.render(<MyComponent />, document.getElementById('root'));
```

4. Self-Closing Tags: Just like in HTML, you can use self-closing tags in JSX for elements that don't have any children. For example:

**<img src="image.jpg" alt="An image" />**

5. Expressions in JSX: You can embed JavaScript expressions within JSX using curly braces {}. This allows you to dynamically compute values or render conditional content.

For example:

```
const name = 'John';

<div>Hello, {name}!</div>
```

6. Styling in JSX: You can apply inline styles to JSX elements using the style attribute. The style value should be an object where the keys are CSS properties in camelCase. For example:

```
const style = {            color: 'red',    fontSize: '16px'

};

<div style={style}>Styled text</div>
```

7. Class and ID Attributes: In JSX, the class attribute is replaced by className and the id attribute remains the same. This is because class is a reserved keyword in JavaScript. For example:

```
<div className="my-class" id="my-id">Element with class and id</div>
```

8.Commenting in JSX: You can add comments within JSX by wrapping them in curly braces and using JavaScript-style comments. For example:

```
<div>
        {/* This is a JSX comment */}

        <span>Hello, World!</span>

</div>
```

## ReactDOM:

DOM (Document Object Model).

ReactDOM is a package that provides DOM specific methods that can be used at the top level of a web app to enable an efficient way of managing DOM elements of the web page. ReactDOM provides the developers with an API containing the following methods and a few more.

- render()
- findDOMNode()
- unmountComponentAtNode()
- hydrate()
- createPortal()

ReactDOM contains the tools necessary to render React elements in the browser. ReactDOM is where we'll find the render method.

ReactDom is a separate library from React itself. While React provides the core functionality for building user interfaces, ReactDom provides the functionality for rendering those interfaces to the browser.

Overall, ReactDom is a crucial part of the React ecosystem that enables developers to render their React components to the browser and interact with the DOM.

**Pre-requisite:** To use the ReactDOM in any React web app we must first import ReactDOM from the react-dom package by using the following code snippet:

import ReactDOM from 'react-dom' **render()**

**Function:**

This is one of the most important methods of ReactDOM. This function is used to render a single React Component or several Components wrapped together in a Component or a div element. This function uses the efficient methods of React for updating the DOM by being able to change only a subtree, efficient diff methods, etc.

**Syntax:** ReactDOM.render(element, container, callback)

**Parameters**: This method can take a maximum of three parameters as described below.

- **element:** This parameter expects a JSX expression or a React Element to be rendered.
- **container:** This parameter expects the container in which the element has to be rendered.

- **callback:** This is an optional parameter that expects a function that is to be executed once the render is complete.

**Return Type:** This function returns a reference to the component or null if a stateless component was rendered.

## findDOMNode() Function:

This function is generally used to get the DOM node where a particular React component was rendered. This method is very less used like the following can be done by adding a ref attribute to each component itself.

**Syntax**: ReactDOM.findDOMNode(component)

**Parameters**: This method takes a single parameter component that expects a React Component to be searched in the Browser DOM.

**Return Type:** This function returns the DOM node where the component was rendered on success otherwise null. **unmountComponentAtNode() Function:**

This function is used to unmount or remove the React Component that was rendered to a particular container. As an example, you may think of a notification component, after a brief amount of time it is better to remove the component making the web page more efficient.

**Syntax:** ReactDOM.unmountComponentAtNode(container)

**Parameters**: This method takes a single parameter container which expects the DOM container from which the React component has to be removed.

**Return Type:** This function returns true on success otherwise false. **hydrate() Function:**

The hydrate method is similar to render, but it is used for server-side rendering. It takes an existing HTML page with server-rendered content and adds interactivity by hydrating it with React components.

**Syntax:** ReactDOM.hydrate(element, container, callback)

**Parameters**: This method can take a maximum of three parameters as described below.

- **element:** This parameter expects a JSX expression or a React Component to be rendered.
- **container:** This parameter expects the container in which the element has to be rendered.
- **callback:** This is an optional parameter that expects a function that is to be executed once the render is complete.

**Return Type:** This function attempts to attach event listeners to the existing markup and returns a reference to the component or null if a stateless component was rendered.

**createPortal() Function:**

Usually, when an element is returned from a component's render method, it's mounted on the DOM as a child of the nearest parent node which in some cases may not be desired. Portals allow us to render a component into a DOM node that resides outside the current DOM hierarchy of the parent component.

**Syntax:** ReactDOM.createPortal(child, container)

**Parameters**: This method takes two parameters as described below.

- **child:** This parameter expects a JSX expression or a React Component to be rendered.
- **container:** This parameter expects the container in which the element has to be rendered. **Return Type:** This function returns nothing.

**Important Points to Note:**

- ReactDOM.render() replaces the child of the given container if any. It uses a highly efficient diff algorithm and can modify any subtree of the DOM.

- findDOMNode() function can only be implemented upon mounted components thus Functional components can not be used in findDOMNode() method.
- ReactDOM uses observables thus provides an efficient way of DOM handling.
- ReactDOM can be used on both the client-side and server-side

Anything related to rendering elements to the DOM is found in the ReactDOM pack-age. In versions of React earlier than React 16, you could only render one element tothe DOM. Today, it's possible to render arrays well.

**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.

## Children:

In React, the **ReactDOM** package is responsible for rendering React elements into the DOM (Document Object Model). When rendering elements using **ReactDOM.render(),** the elements can have child elements. Here are some important notes regarding children in React:

1. Children in JSX: In JSX, the child elements are defined within the opening and closing tags of a parent element.

   For example:
   ```
   <ParentComponent>
   <ChildComponent1 />
               <ChildComponent2 />
   </ParentComponent>
   ```
   In this example, ChildComponent1 and ChildComponent2 are the children of ParentComponent.

2. Accessing children in a component: To access the children of a component, you can use the props.children property within the component. It allows you to work with the children as an array-like object.

   For example:
   ```
     function ParentComponent(props) {
     return (
           <div>
           {props.children}
       </div>
     );
   }
   ```
   In this case, the ParentComponent simply renders its children within a div.

3. Passing data to children: You can pass data from a parent component to its children using props.For example:

   ```
   function ParentComponent() {
     const data = "Hello from parent";

     return (
       <div>
   ```

```
    <ChildComponent message={data} />
   </div>
  );
}

function ChildComponent(props) {
return <p>{props.message}</p>;
}
```
Here, the ParentComponent passes the data as the message prop to the ChildComponent, which then renders it.

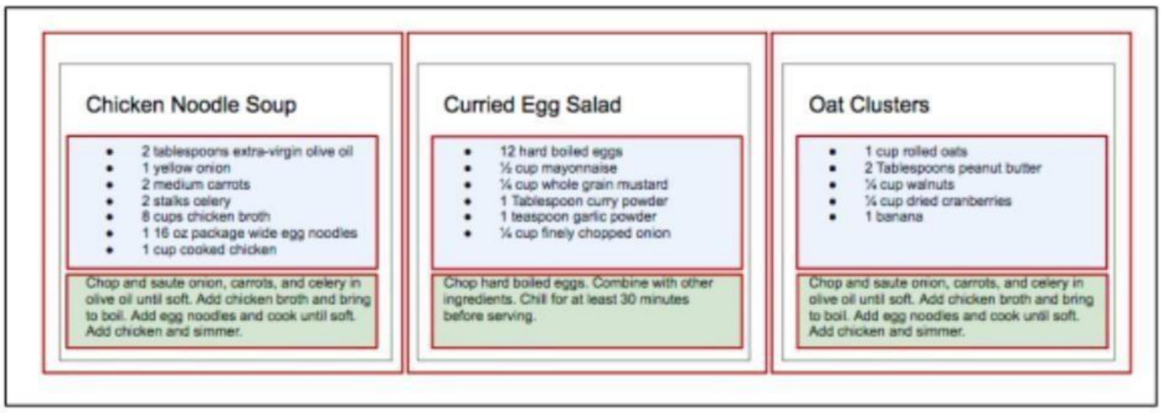
4. Conditional rendering of children: You can conditionally render children based on certain conditions. This can be done using JavaScript expressions and logical operators. For example:

```
        function ParentComponent(props) {
return (
                        <div>
                          {props.showChildren && (
                              <div>
                                  <ChildComponent1 />
                                  <ChildComponent2 />
                              </div>
                           )}
                        </div>
                  );
        }
```
In this case, the children components will only be rendered if the **showChildren** prop is **true**.

## REACT COMPONENTS

In React, we describe each of these parts as a component. Components allow us to reuse the same structure, and then we can populate those structures with different sets of data.

When considering a user interface you want to build with React, look for opportunities to break down your elements into reusable pieces. For example, the recipes in Figure 4-3 have a title, ingredients list, and instructions. All are part of a larger recipe or app component. We could create a component for each of the highlighted parts:  ingredients, instructions, and so on.



Chicken Noodle Soup

- 2 tablespoons extra-virgin olive oil
- 1 yellow onion
- 2 medium carrots
- 2 stalks celery
- 8 cups chicken broth
- 1 16 oz package wide egg noodles
- 1 cup cooked chicken

Chop and saute onion, carrots, and celery in olive oil until soft. Add chicken broth and bring to boil. Add egg noodles and cook until soft. Add chicken and simmer.

Curried Egg Salad

- 12 hard boiled eggs
- ½ cup mayonnaise
- ¼ cup whole grain mustard
- 1 Tablespoon curry powder
- 1 teaspoon garlic powder
- ¼ cup finely chopped onion

Chop hard boiled eggs. Combine with other ingredients. Chill for at least 30 minutes before serving.

Oat Clusters

- 1 cup rolled oats
- 2 Tablespoons peanut butter
- ¼ cup walnuts
- ¼ cup dried cranberries
- 1 banana

Chop and saute onion, carrots, and celery in olive oil until soft. Add chicken broth and bring to boil. Add egg noodles and cook until soft. Add chicken and simmer.

*Figure 4-3. Each component is outlined:* `App`, `IngredientsList`, `Instructions`

Think about how scalable this is. If we want to display one recipe, our component structure will support this. If we want to display 10,000 recipes, we'll just create 10,000 new instances of that component.

We'll create a component by writing a function. That function will return a reusable part of a user interface. Let's create a function that returns an unordered list of ingredients.

This time, we'll make dessert with a function called IngredientsList:

```
function IngredientsList() { return React.createElement(
"ul",
{ className: "ingredients" },
React.createElement("li", null, "1 cup unsalted butter"),
React.createElement("li", null, "1 cup crunchy peanut butter"),
React.createElement("li", null, "1 cup brown sugar"),
React.createElement("li", null, "1 cup white sugar"),
React.createElement("li", null, "2 eggs"),
React.createElement("li", null, "2.5 cups all purpose flour"),
React.createElement("li", null, "1 teaspoon baking powder"),
React.createElement("li", null, "0.5 teaspoon salt")
);
}
ReactDOM.render(
React.createElement(IngredientsList, null, null), document.getElementById("root")
);
```

The component's name is IngredientsList, and the function outputs elements that look like this:

```
<IngredientsList>
<ul className="ingredients">
<li>1 cup unsalted butter</li>
<li>1 cup white sugar</li>
<li>2 eggs</li>
<li>2.5 cups all purpose flour</li>
<li>1 teaspoon baking powder</li>
<li>0.5 teaspoon salt</li>
</ul>
</IngredientsList>
```

Creating our component this way will make the component more flexible. Whether the items array is one item or a hundred items long, the component will render each as a list item.

Another adjustment we can make here is to reference the items array from React props. Instead of mapping over the global items, we'll make items available on the props object. Start by passing props to the function, then mapping over props.items:

```
function IngredientsList(props) {
return React.createElement(
"ul",
{ className: "ingredients" },
props.items.map((ingredient, i) =>
React.createElement("li", { key: i }, ingredient)
)
);
}
```

We could also clean up the code a bit by destructuring items from props:

```
function IngredientsList({ items }) { return React.createElement(
"ul",
{ className: "ingredients" }, items.map((ingredient,
i) =>
React.createElement("li", { key: i }, ingredient)
) );
}
```

Everything that's associated with the UI for IngredientsList is encapsulated into one component. Everything we need is right there.

**React Elements as JSX**

Facebook's React team released JSX when they released React to provide a concise syntax for creating complex DOM trees with attributes. They also hoped to make React more readable like HTML and XML. In JSX, an element's type is specified with a tag. The tag's attributes represent the properties. The element's children can be added between the opening and closing tags.

You can also add other JSX elements as children. If you have an unordered list, you can add child list item elements to it with JSX tags. It looks very similar to HTML:

```
<ul>
<li>1 lb Salmon</li>
<li>1 cup Pine Nuts</li>
<li>2 cups Butter Lettuce</li>
<li>1 Yellow Squash</li>
<li>1/2 cup Olive Oil</li>
<li>3 Cloves of Garlic</li>
</ul>
```

JSX works with components as well. Simply define the component using the class name. We pass an array of ingredients to the IngredientsList as a property with JSX, as shown in Figure 5-1.

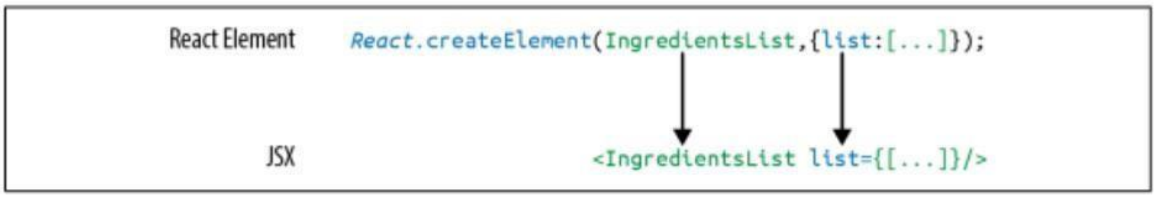

React Element
React.createElement(IngredientsList,{list:[...]});
JSX
<IngredientsList list={[...]}/>

*Figure 5-1. Creating the IngredientsList with JSX*

When we pass the array of ingredients to this component, we need to surround it with curly braces. This is called a JavaScript expression, and we must use these when passing JavaScript values to components as properties. Component

properties will take two types: either a string or a JavaScript expression. JavaScript

expressions can include arrays, objects, and even functions. In order to include

them, you must sur- round them in curly braces.

**JSX TIPS**

JSX might look familiar, and most of the rules result in syntax that's similar to HTML.

However, there are a few considerations you should understand when working with JSX.

**Nested components**

JSX allows you to add components as children of other components. For example,

inside the IngredientsList, we can render another component called Ingredient

multiple times:

```
<IngredientsList>
<Ingredient />
<Ingredient />
</IngredientsList>
```
**className**

Since class is a reserved word in JavaScript, className is used to define the class attribute

instead:

```
<h1 className="fancy">Baked Salmon</h1>
```

**JavaScript expressions**

JavaScript expressions are wrapped in curly braces and indicate where variables

will be evaluated and their resulting values returned. For example, if we want to

display the value of the title property in an element, we can insert that value using a

Java-

Script expression. The variable will be evaluated and its value returned:

```
<h1>{title}</h1>
```

Values of types other than string should also appear as JavaScript expressions:

```
<input type="checkbox" defaultChecked={false} />
```

**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:

```
<h1>{"Hello" + title}</h1>
<h1>{title.toLowerCase().replace}</h1>
```

Mapping Arrays with JSX

JSX is JavaScript, so you can incorporate JSX directly inside of JavaScript functions. For example, you can map an array to JSX elements:

```
<ul>
{props.ingredients.map((ingredient, i) => (
<li key="{i}">{ingredient}</li>
))}
</ul>
```

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

## Overview

**Many software langauges require you to compile your source code.JavaScript interpreted language that is the browser interprets the code as text, so there's no need to compile JavaScript.**

However, not all browsers support the latest JavaScript syntax, andno browser supports JSX syntax. Since we want to use the latest features of Java- Script along with JSX, we're going to need a way toconvert our fancy source code into something that the browser caninterpret. This process is called compiling, and it's what Babel is designed to do.

## Definition

1.Babel helps developers to write modern Javascript code and transpile it into a version that can run in all environments, including modern browsers and older ones.

2.Babel is a toolchain that is mainly used to convert ECMAScript 2015+ code intobackwards compatible version of JavaScript in current and older browsers or environments.

## Introduction

The first version of the project was called 6to5, and it was released in September 2014. 6to5 was a tool that could beused to convert ES6 syntax to ES5 syntax, which was more widely supported by web browsers.

As the project grew, it aimed to be a platform to support allof the latest changes in ECMAScript. It also grew to support converting JSX into JavaScript. The project was renamedBabel in February 2015.

Babel is used in production at Facebook, Netflix, PayPal,Airbnb, and more.

## Working with Babel

There are many ways of working with Babel. The easiest way toget started is to include a link to the Babel CDN directly in your HTML, which will compile any code in script blocks that have a type of “text/babel.

Babel will compile the source code on the client before runningit. Although this may not be the best solution for production, it’s a great way to get started with JSX.

## Example:

```
<!DOCTYPE html>
<html>
  <head>
  <meta charset="utf-8" />
  <title>React Examples</title>
  </head>
  <body>
  <div id="root"></div>
  <!-- React Library & React DOM -->
```

```
   <script src="https://unpkg.com/react@16.8.6/umd/react.development.js">
   </script>
   <script
   src="https://unpkg.com/react-dom@16.8.6/umd/reactdom.development.js">
   </script>
   <script src="https://unpkg.com/@babel/standalone/babel.min.js">
   </script>
   <script type="text/babel">
 // JSX code here. Or link to separate JavaScript file that containsJSX.
   </script>
   </body>
 </html>
```

## Need of Babel :

The need for Babel is as follows:

**1.Compatibility:**

Babel allows developers to write modern JavaScript code that is compatible with all environments, including older browsers thatdo not support the latest JavaScript syntax.

**2.Latest syntax:**

Babel in react allows developers to use the latest language features, such as ECMAScript 6 (ES6) and beyond, making iteasier to write code that is more efficient and easier to read.

**3.Debugging:**

Babel makes it easier for developers to debug their code and find and fix issues, making the development process faster andmore efficient.

**4.Customization:**

Babel in react is designed to be pluggable, which means developers can add or remove plugins to customize its functionality, making it a more flexible tool for a wide range of projects.

**5.Type annotations:**

Babel supports type annotations, making it easier for developersto write code with a clear understanding of the data types used in their applications. This helps to catch type-related errors earlyin the development process.

## Recipes as JSX

- JSX provides us with a nice, clean way to express React elements in our code that makes  sense to us and is immediatelyreadable by developers.
- The drawback of JSX is that it’s not readable by the browser. Beforeour code can be interpreted by the browser, it needs to be converted from JSX into JavaScript.
- This data array contains two recipes, and this represents ourapplication’s current state:

```
const data = [
        {
        name: "Baked
        Salmon",ingredients: [
        { name: "Salmon", amount: 1, measurement: "l lb" },
        { name: "Pine Nuts", amount: 1, measurement: "cup" },
        { name: "Butter Lettuce", amount: 2, measurement: "cups" },
        { name: "Yellow Squash", amount: 1, measurement: "med" },
        { name: "Olive Oil", amount: 0.5, measurement: "cup" },
        { name: "Garlic", amount: 3, measurement: "cloves" }
        ],
        steps: [
        "Preheat the oven to 350 degrees.",
        "Spread the olive oil around a glass baking dish.",
        "Add the yellow squash and place in the oven for 30 mins.","Add
        the salmon, garlic, and pine nuts to the dish.",
        "Bake for 15 minutes.",
        "Remove from oven. Add the lettuce and serve."

        ]

        },

        {

        name: "Fish

        Tacos",

        ingredients: [

        { name: "Whitefish", amount: 1, measurement: "l lb" },

        { name: "Cheese", amount: 1, measurement: "cup" },

        { name: "Iceberg Lettuce", amount: 2, measurement: "cups" },

        { name: "Tomatoes", amount: 2, measurement: "large" },

        { name: "Tortillas", amount: 3, measurement: "med" }

        ],

        steps: [

        "Cook the fish on the grill until cooked through.",

        "Place the fish on the 3 tortillas.",

        "Top them with lettuce, tomatoes, and cheese."

        ]

                }

        ];
```

- The data is expressed in an array of two JavaScript objects.
- We can create a UI for these recipes with two components: a Menu component for listing the recipes and a Recipe componentthat describes the UI for each recipe.

- It’s the Menu component that we’ll render to the DOM. We’llpass our data to the Menu component as a property called recipes:

```
// The data, an array of Recipe objects

const data = [ ... ];

// A function component for an individual Recipe

function Recipe (props) {

...

}

// A function component for the Menu of Recipes

function Menu (props) {

...

}

// A call to ReactDOM.render to render our Menu into the current DOM

ReactDOM.render(

<Menu recipes={data} title="Delicious Recipes" />,

document.getElementById("root")

);
```

A header element, an h1 element, and a div.recipes element are used to describe the DOM for our menu. The value for the titleproperty will be displayed as text within the h1:

```
function Menu(props)

{return (

<article>

<header>

<h1>{props.title}</h1>
</header>

<div className="recipes" />

</article>

);

}
```

- Inside of the div.recipes element, we add a component for eachrecipe:

```
<div className="recipes">

{props.recipes.map((recipe, i) => (

<Re

cipe

key

={i

}

name={recipe.name}

ingredients={recipe.ingredients}

steps={recipe.steps}

/>

))}

</div>
```

- You could also refactor this to use spread syntax. The JSX spread operator works like the object spread operator. It will add each field ofthe recipe object as a property of the Recipe component. The syntax here will supply all properties to the component:

```
{

props.recipes.map((recipe, i) => <Recipe key={i} {...recipe} />);

}
```

 We can use object destructuring to scope the variables to this function. This allows us to access the title and recipes variables directly,no longer having to prefix them with props:

```
function Menu({ title, recipes }) {
return (

<article>
<header>
<h1>{title}</h1>
</header>
<div className="recipes">
{recipes.map((recipe, i) => (
<Recipe key={i} {...recipe} />
))}
</div>
</article>
);
}
```

 Now let’s code the component for each individual recipe:

```
function Recipe({ name, ingredients, steps }) {
return (

<section id={name.toLowerCase().replace(/ /g, "-")}>
<h1>{name}</h1>
<ul className="ingredients">
{ingredients.map((ingredient, i) => (
<li key={i}>{ingredient.name}</li>
))}
</ul>
<section className="instructions">
<h2>Cooking Instructions</h2>
{steps.map((step, i) => (
<p key={i}>{step}</p>
))}
</section>
</section>
);
}
```

 **The complete code for the application should look like this:**const data = [

```
{
name: "Baked Salmon",
ingredients: [
{ name: "Salmon", amount: 1, measurement: "l lb" },
{ name: "Pine Nuts", amount: 1, measurement: "cup" },
{ name: "Butter Lettuce", amount: 2, measurement: "cups" },
{ name: "Yellow Squash", amount: 1, measurement: "med" },
```

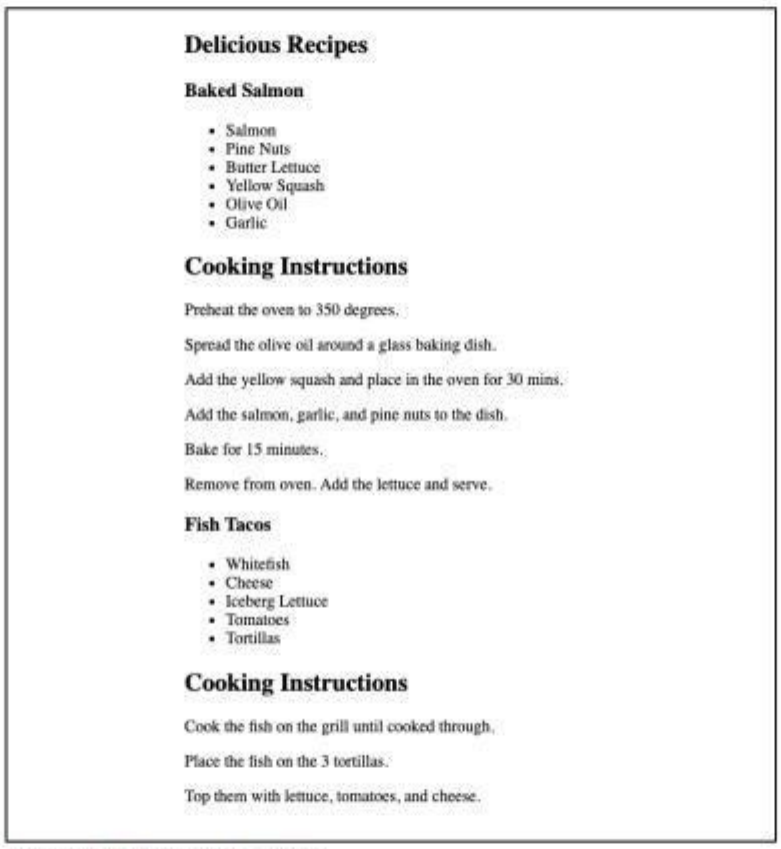
```
{ name: "Olive Oil", amount: 0.5, measurement: "cup" },
{ name: "Garlic", amount: 3, measurement: "cloves" }
],
steps: [
"Preheat the oven to 350 degrees.",
"Spread the olive oil around a glass baking dish.",
"Add the yellow squash and place in the oven for 30 mins.","Add the
salmon, garlic, and pine nuts to the dish.",
"Bake for 15 minutes.",
"Remove from oven. Add the lettuce and serve."
]
},
{
name: "Fish
Tacos",
ingredients: [
{ name: "Whitefish", amount: 1, measurement: "l lb" },
{ name: "Cheese", amount: 1, measurement: "cup" },
{ name: "Iceberg Lettuce", amount: 2, measurement: "cups" },
{ name: "Tomatoes", amount: 2, measurement: "large" },
{ name: "Tortillas", amount: 3, measurement: "med" }
],
steps: [
"Cook the fish on the grill until hot.",
"Place the fish on the 3 tortillas.",
"Top them with lettuce, tomatoes, and cheese."
]
}
];
function Recipe({ name, ingredients, steps }) {
return (
<section id={name.toLowerCase().replace(/ /g, "-")}>
<h1>{name}</h1>
<ul  className="ingredients">
```

```
{ingredients.map((ingredient, i) => (
<li key={i}>{ingredient.name}</li>
))}
</ul>
<section className="instructions">
<h2>Cooking Instructions</h2>
{steps.map((step, i) => (
<p key={i}>{step}</p>
))}
</section>
</section>
);
}
function Menu({ title, recipes }) {
return (
<article>
<header>
```

```
<h1>{title}</h1>
</header>
<div className="recipes">
{recipes.map((recipe, i) => (
<Recipe key={i} {...recipe} />
))}
```

```
</div>
</article>
);
}
ReactDOM.render(
<Menu recipes={data} title="Delicious Recipes" />,document.getElementById("root")
);
```

- When we run this code in the browser, React will construct a UI using ourinstructions with the recipe data as shown in Figure: 5.2



Delicious Recipes

Baked Salmon

- Salmon
- Pine Nuts
- Butter Lettuce
- Yellow Squash
- Olive Oil
- Garlic

Cooking Instructions

Preheat the oven to 350 degrees.

Spread the olive oil around a glass baking dish.

Add the yellow squash and place in the oven for 30 mins.

Add the salmon, garlic, and pine nuts to the dish.

Bake for 15 minutes.

Remove from oven. Add the lettuce and serve.

Fish Tacos

- Whitefish
- Cheese
- Iceberg Lettuce
- Tomatoes
- Tortillas

Cooking Instructions

Cook the fish on the grill until cooked through.

Place the fish on the 3 tortillas.

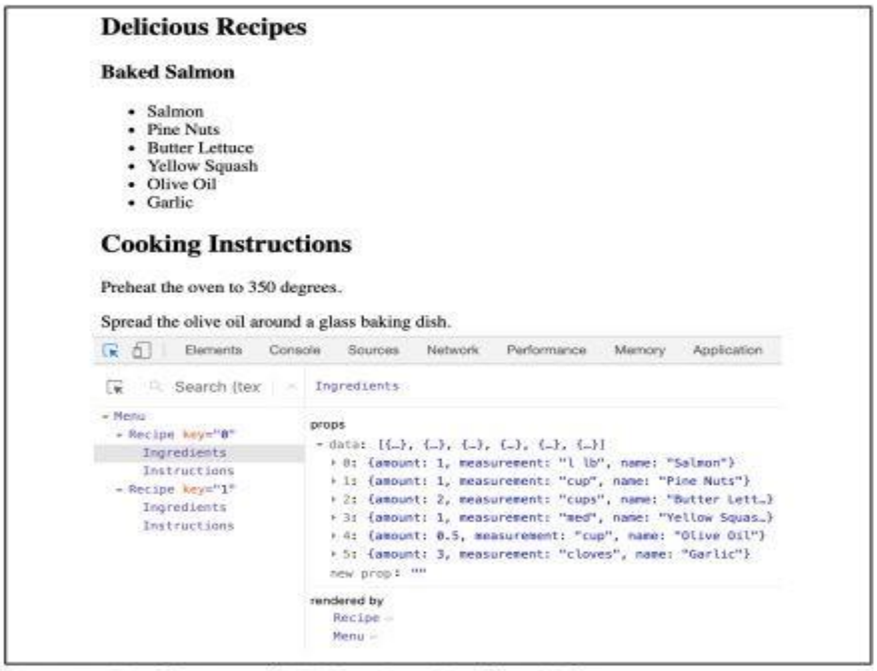
Top them with lettuce, tomatoes, and cheese.

*Figure 5-2. Delicious Recipes output*

- If you’re using Google Chrome and have the React Developer Tools Extension installed, you can take a look at the present state of the component tree. To do this, open the developer tools and select the Components tab, as shown in Figure5-3.

If we change the recipes array and rerender our Menu component, React will change this DOM as efficiently as possible.

Here we can see the Menu and its child elements. The data array contains two objects for recipes, and we have two Recipe elements. Each Recipe element has properties for the recipe name, ingredients, and steps. The ingredients and steps arepassed down to their own components as data.



Delicious Recipes

Baked Salmon

- Salmon
- Pine Nuts
- Butter Lettuce
- Yellow Squash
- Olive Oil
- Garlic

Cooking Instructions

Preheat the oven to 350 degrees.

Spread the olive oil around a glass baking dish.

```
- Menu
  - Recipe key="0"
      Ingredients
      Instructions
  - Recipe key="1"
      Ingredients
      Instructions
```

```
Ingredients

props
- data: [{…}, {…}, {…}, {…}, {…}, {…}]
  ▸ 0: {amount: 1, measurement: "l lb", name: "Salmon"}
  ▸ 1: {amount: 1, measurement: "cup", name: "Pine Nuts"}
  ▸ 2: {amount: 2, measurement: "cups", name: "Butter Lett…}
  ▸ 3: {amount: 1, measurement: "med", name: "Yellow Squas…}
  ▸ 4: {amount: 0.5, measurement: "cup", name: "Olive Oil"}
  ▸ 5: {amount: 3, measurement: "cloves", name: "Garlic"}
  new prop: ""

rendered by
  Recipe
  Menu
```

*Figure 5-3. Resulting virtual DOM in React Developer Tools*

## React Fragments

React Fragment is a feature in React that allows you to return multiple elements from a React component by allowing you to groupa list of children without adding extra nodes to the DOM.

### Example:

```
function Cat({ name }) {
  return <h1>The cat's name is {name}</h1>;
}
ReactDOM.render(<Cat name="Jungle" />,
document.getElementById("root"));
```

This will give output as: The cat’s name is Jungle

- But if you want to print multiple statements then:

function Cat({ name }) {return (

```
    <h1>The cat's name is {name}</h1>
    <p>He's good.</p>
    );
  }
```

Immediately, we'll see an error in the console that reads AdjacentJSX elements must be wrapped in an enclosing tag and recommends using a fragment.

React will not render two or more adjacent or sibling elements asa component, so we used to have to wrap these in an enclosing tag like a div.

To return multiple elements from a React component, you'llneed to wrap the element in a root element.

- This is where fragments come into play! If we use a React fragment, we can mimic the behavior of a wrapper withoutactually creating a new tag.

➢ **wrapping the adjacent tags, the h1 and p, with a React.Fragmenttag:**

```
function Cat({ name }) {return (
    <React.Fragment>
    <h1>The cat's name is {name}</h1>
    <p>He's good.</p>
    </React.Fragment>
    );
  }
```

**You also can use a fragment shorthand to make this look evencleaner:**

```
function Cat({ name}){
   return (
    <>
      <h1>The cat's name is {name}  </h1>
       <p>He's good.</p>
    </>
  );
}
```

➢ **Using a Div Tag:**

```
<div id="root">
  <h1>The cat's name is Jungle</h1>
  <p>He's good</p>
</div>
```

Fragments are a relatively new feature of React and do away with the need for extra wrapper tags that can pollute the DOM.

# React State Management

A component tree is a hierarchy of components that data was able to flow through as properties. Properties are half of the picture. State is the other half. The state of a React application is driven by data that has the ability to change. Introducing state to the recipe application could make it possible for chefs to create new recipes, modify existing recipes, and remove old ones. State and properties have a relationship with
each other. When we work with React applications, we gracefully compose components that are tied together based on this relationship. When the state of a

component tree changes, so do the properties. The new data flows through the tree, causing specific leaves and branches to render to reflect the new content.

## Building Forms:

A web developer usually collects large amounts of information from users with forms, we can build a lot of form components with React.All of the HTML form elements that are available to the DOM are also available as React elements.

Sample syntax (To render a form with JSX):

```
<form>
  <input type="text" placeholder="color title..." required />
  <input type="color" required />
  <button>ADD</button>
</form>
```

This form element has three child elements: two input elements and a button. The first input element is a text input that will be used to collect the title value for new colors. The second input element is an HTML color input that will allow users to pick a color from a color wheel. Basic HTML form validation is used to mark both inputs as required. The ADD button is used to add a new color.

In React, there are two ways of handling form data:

- **Controlled Components:** In this approach, form data is handled by React through the use of hooks such as the useState hook.

- **Uncontrolled Components:** Form data is handled by the Document Object Model (DOM) rather than by React. The DOM maintains the state of form data
and updates it based on user input.

## Using Refs- (Uncontrolled Components)

Uncontrolled components in React refer to form elements whose state is not managed by React. Instead, their state is handled by the browser's DOM.To get the value of an uncontrolled form element, you can use a feature called "ref". "Refs" provide a way to access the current value of DOM elements. You can create a "ref" using the useRef hook, then attach it to the form element you want to access. This allows you to retrieve the current value of an element at any time, without needing to manage its state in your React component.

To build a form component in React there are many patterns, one of the patterns involves accessing the DOM node directly using a React feature called refs.

In React, a ref is an object that stores values for the lifetime of a component. There are several use cases that involve using refs. A DOM node can be directly accessed
with a ref.

React provides us with a **useRef hook** that we can use to create a ref. Building the AddColorForm component using this hook:

```
import React, { useRef } from "react";
export default function AddColorForm({ onNewColor = f => f }) {
const txtTitle = useRef();  const hexColor = useRef();  const submit = e => {
... }  return (...)
}
```

Creating two refs using the useRef hook.

- txtTitle ref - reference to text input in form to collect the color title.

- hexColor ref - to access hexadecimal color values from HTML color input.

```
return (
 <form onSubmit={submit}>
 <input ref={txtTitle} type="text" placeholder="color title..." required />
 <input ref={hexColor} type="color" required />
 <button>ADD</button>
 </form>
 );
}
```

This creates a current field on the ref object that references the DOM element directly which provides access to the DOM element that captures its value. When the user submits this form by clicking the ADD button, it will invoke the submit function:

```
const submit = e => {
   e.preventDefault();
   const title = txtTitle.current.value;     const
color = hexColor.current.value;
onNewColor(title, color);     txtTitle.current.value = "";
hexColor.current.value = "";
  };
```

This is imperative code as the AddColorForm is now an uncontrolled component because it uses the DOM to save the form values. Sometimes using uncontrolled component can get us out of problems. For instance, one may want to share access to a form and its values with code outside of React. However, a controlled component is a better approach.

## Controlled Components

In a controlled component, the from values are managed by React and not the DOM. They do not require us to use refs. They do not require us to write imperative code.

Adding features like robust form validation is much easier when working with a controlled component.

To modify the AddColorForm control is given over the form's state:

```
import React, { useState } from "react";
export default function AddColorForm({ onNewColor = f => f }) {
const [title, setTitle] = useState("");  const [color, setColor] = useState("#000000");  const
submit = e => { ... };  return ( ... );
}
```

Instead of using refs, we can save the values for the title and color using React state. Creating variables for title and color and defining the functions can be used to change state: setTitle and setColor.

```
<form onSubmit={submit}>
   <input           value={title}           onChange={event  =>  setTitle(event.target.value)}
type="text"
   placeholder="color title..."
required     />     <input     value={color}
   onChange={event => setColor(event.target.value)}
type="color"     required
   />
   <button>ADD</button>
  </form>
  }
```

To submit the form, the state values for title and color are simply passed to the onNewColor function property as arguments when we invoke it. The setTitle and setColor functions can be used to reset the values after the new color has been passed to the parent component:

```
const submit = e => {
 e.preventDefault();
onNewColor(title,    color);        setTitle("");
setColor("");
};
```

## Creating Custom Hooks

**Hooks:** React hooks lets to use state in an application and other React features without writing a class.

For a large form with a lot of input elements, these two lines of code are generallyused:

```
value={title}
onChange={event => setTitle(event.target.value)}
```

Creating our own custom hook - useInput hook where we can abstract away the redundancy involved with creating controlled form inputs:

```
import { useState } from "react"; export const
useInput = initialValue => {   const [value, setValue] =
useState(initialValue); return [
 { value, onChange: e => setValue(e.target.value) },
 () => setValue(initialValue)
 ]; };
```

Using the hook inside of the AddColorForm:

```
import React from "react"; import { useInput } from "./hooks"; export
default function AddColorForm({ onNewColor = f => f }) { const
[titleProps, resetTitle] = useInput(""); const [colorProps, resetColor] =
useInput("#000000"); const submit = event => { ...
} return ( ... )
}
```

The titleProps and colorProps are used to be spread into their corresponding input elements:

```
return (
 <form onSubmit={submit}>
 <input
 {...titleProps}  type="text"   placeholder="color title..."
required
 />
 <input {...colorProps} type="color" required />
 <button>ADD</button>
 </form>
);
}
```

Spreading these properties from our custom hook is better than pasting them. Hook is used to create controlled form inputs without worrying about the underlying implementation details.

To submit the form:

```
const submit = event => {
event.preventDefault();
onNewColor(titleProps.value,      colorProps.value);            resetTitle();
resetColor();
};
```

Hooks are designed to be used inside of React components. We can compose hooks within other hooks because eventually the customized hook will be used inside of a component

**Adding Colors to State**

Both the controlled form component and the uncontrolled from component pass the values for title and color to the parent component via the onNewColor function. The parent doesn't care whether we used a controlled component or an  uncontrolled component; it only wants the values for the new color.

## Adding the AddColorForm:

```
import React, { useState } from "react"; import
colorData  from  "./color-data.json";  import  ColorList  from
"./ColorList.js"; import
AddColorForm from "./AddColorForm"; import
{ v4 } from "uuid"; export default function App()
{  const [colors, setColors] = useState(colorData);
return (  <>
 <AddColorForm
 onNewColor={(title, color) => {  const
newColors = [
 ...colors,
{   id: v4(),
rating:    0,
title,   color
}  ];
 setColors(newColors);
 }}
 />
```

```
 <ColorList .../>
 </>
 );
}
```

Users can now add new colors to the list, remove colors from the list, and rate any existing color on that list.

## Incorporating Data

Data is the most imporatant element in applications.The user interface components store data. Applications are filled with data from the internet. We collect, create, and send new data to the internet. The value of our applications is not the components themselves, it’s the data that flows through those components.

## Requesting Data

In JavaScript, we use fetch to make an HTTP request

```
fetch(`https://api.github.com/users/moonhighway`)
 .then(response => response.json())
 .then(console.log)
 .catch(console.error);
```

The fetch function returns a promise which making an asynchronous request to a specific URL. It takes time for that request to traverse the internet and respond with information. When it does, that information is passed to a callback using the .then(callback) method.

GitHub will respond to this request with a JSON object

Another way of working with promises is to use async/await. Since fetch returns a promise, we can await a fetch request inside of an async function:  async function requestGithubUser(githubLogin)

```
{   try {
 const response = await fetch(
 `https://api.github.com/users/${githubLogin}`
 );
 const userData = await response.json();
console.log(userData);  } catch (error) {  console.error(error);
 }
}
```

## Sending Data with a Request

A lot of requests require us to upload data with the request.To collect information about a user in order to create an account, or to update their account, we use a POST request when we're creating data and a PUT request when we're modifying
it.

The second argument of the fetch function allows us to pass an object of options that fetch can use when creating our HTTP request:

```
fetch("/create/user", {
 method: "POST",
 body: JSON.stringify({ username, password, bio })
});
```

## Uploading Files with fetch

Uploading files requires a different type of HTTP request: a multipart formdata
request. This type of request tells the server that a file or multiple files are located in the body of the request. To make this request in JavaScript, pass a FormData object in the body of our request:

```
const formData = new FormData();
formData.append("username", "moontahoe");
formData.append("fullname", "Alex Banks");
forData.append("avatar", imgFile); fetch("/create/user",
{  method: "POST",  body: formData
});
```

## Authorized Requests

Authorization is required to obtain personal or sensitive data. Additionally, authorization is almost always required for users to take action on the server with POST, PUT, or DELETE requests.

Users typically identify themselves with each request by adding a unique token to the request that a service can use to identify the user which is usually added as the Authorization header.

```
fetch(`https://api.github.com/users/${login}`, {
```

```
method: "GET",  headers: {
 Authorization: `Bearer ${token}`
 }
});
```

Fetching data from within a React component requires us to orchestrate the useState and useEffect hooks.

- The useState hook is used to store the response in state

- The useEffect hook is used to make the fetch request import React, { useState, useEffect } from "react"; function GitHubUser({ login }) { const [data, setData] = useState(); useEffect(() => { if (!login) return; fetch(`https://api.github.com/users/${login}`)

```
 .then(response => response.json())
 .then(setData)
 .catch(console.error);
 }, [login]);   if  (data)     return  <pre>{JSON.stringify(data,  null,
2)}</pre>;  return
null;
}
export default function App() {
 return <GitHubUser login="moonhighway" />;
}
```

## CREATE A FETCH HOOK

In React, a `fetch` hook is a custom hook that is used to perform network requests, such as fetching data from an API or making HTTP requests. The `fetch` hook simplifies the process of making network requests and managing the state of the data.

Example-1:

**How you can create a `fetch` hook in a React application:**

1. First, create a new file called `useFetch.js` in your project.
2. In the `useFetch.js` file, import the `useState` and `useEffect` hooks from the React library. These hooks will be used to manage the state of the data.
3. Create the `useFetch` function. This function will take a URL as an argument and return an object with two properties: `data` and `error`.

**.js file**

```
import { useState, useEffect } from 'react';

function useFetch(url) {   const [data, setData]

= useState(null);   const [error, setError] =

useState(null);


  useEffect(() => {     const

fetchData = async () => {

     try {

       const response = await fetch(url);

const data = await response.json();

setData(data);       } catch (error) {

setError(error);

      }

    };

    fetchData();

  }, [url]);

  return { data, error };
 }
```

4.Inside the `useFetch` function, use the `useState` hook to create two state variables: `data` and `error`. The `data` state variable will store the response data, and the `error` state variable will store any errors that occur during the network request.

5.Use the `useEffect` hook to perform the network request. Inside the `useEffect` hook, create an asynchronous function called `fetchData`. This function will use the `fetch` API to make a network request to the specified URL.

6.Inside the `fetchData` function, use a `try`/`catch` block to handle any errors that occur during the network request. If the request is successful, use the `setData` function to update the `data` state variable with the response data. If there is an error, use the `setError` function to update the `error` state variable with the error.

7.Finally, call the `fetchData` function inside the `useEffect` hook, passing in the `url` argument. Add the `url` argument to the dependency array of the `useEffect` hook to ensure that the hook is re-run whenever the URL changes.

8.Return an object from the `useFetch` function with two properties: `data` and `error`. These properties will be used to access the response data and any errors that occur during the network request.

**Now, you can use the `useFetch` hook in any component in your React application to fetch data from an API:**

**.js file**

```
import useFetch from './useFetch';


function MyComponent() {    const { data, error } =
useFetch('https://api.example.com/data');


  if (error) {
    return <div>Error: {error.message}</div>;
  }


  if (!data) {
    return <div>Loading...</div>;
  }
  return (
    <div>
      <h1>{data.title}</h1>
      <p>{data.description}</p>
```

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

In this example, the `useFetch` hook is used to fetch data from the `https://api.example.com/data` URL. The `data` and `error` properties are then used to render the component, depending on the state of the network request.

Example-2:

We know that a request is either pending, successful, or failed. We can reuse the logic that's necessary for making a fetch request by creating a custom hook. We'll call this hook useFetch, and we can use it in components across our application whenever we need to make a fetch request:

**.js file**

```
import React, { useState, useEffect } from "react";
export function useFetch(uri) { const [data, setData]
= useState(); const [error, setError] = useState();
const [loading, setLoading] = useState(true);
useEffect(() => { if (!uri) return; fetch(uri)
.then(data => data.json())
.then(setData)
.then(() => setLoading(false))
.catch(setError);
}, [uri]);
return {
loading, data,
error
}; }
```

This custom hook was created by composing the useState and useEffect hooks. The three states of a fetch request are represented in this hook: pending, success, and error. When the request is pending, the hook will return true for loading. When the request is successful and data is retrieved, it will be passed to the component from this hook. If something goes wrong, then this hook will return the error. All three of these states are managed inside of the useEffect hook. This hook is invoked every time the value for uri changes. If there's no uri, the fetch request is not made. When there's a uri, the fetch request begins. If the request is successful, we pass the resulting JSON to the setData function, changing the state value for data. After that, we then change the state value for loading to false because the request was successful (i.e., it's no longer pending). Finally, if anything goes wrong, we catch it and pass it to setError, which changes the state value for error.

Now we can use this hook to make fetch requests within our components. Anytime the values for loading, data, or error change, this hook causes the GitHubUser com- ponent to rerender with those new values:

**Now, you can use the `useFetch` hook in any component in your React application to fetch data from an API:**

**.js file**

```
function GitHubUser({ login }) { const {
loading, data, error } = useFetch(
`https://api.github.com/users/${login}`
);

if (loading) return <h1>loading...</h1>;
if (error)
return <pre>{JSON.stringify(error, null, 2)}</pre>; return
(
<div className="githubUser">
<img
src={data.avatar_url}
alt={data.login} style={{ width:
200 }}
/>
<div>
<h1>{data.login}</h1>
{data.name && <p>{data.name}</p>}
{data.location && <p>{data.location}</p>}
</div>
</div>
); }
```

Although the component now has less logic, it still handles all three states.

## CREATING A FETCH COMPONENT

In React, you can create a fetch component to fetch data from an API using the `fetch` API. Here's an example of how you can create a fetch component in a React application:

1. First, create a new file called `FetchComponent.js` in your project.
2. In the `FetchComponent.js` file, create a new React component called `FetchComponent`.
3. Inside the `FetchComponent` function, use the `useState` hook to create a state variable called `data` and set its initial value to `null`.
4. Use the `useEffect` hook to perform the network request. Inside the `useEffect` hook, create an asynchronous function called `fetchData`. This function will use the `fetch` API to make a network request to the specified URL.
5. Inside the `fetchData` function, use the `fetch` API to make a network request to the specified URL. Use the `setData` function to update the `data` state variable with the response data.
6. Call the `fetchData` function inside the `useEffect` hook. Pass an empty array (`[]`) as the second argument to the `useEffect` hook to ensure that the hook is only run once, when the component is mounted.
7. Finally, use the `data` state variable to render the component. If the `data` variable is `null`, render a loading message. Otherwise, render the response data.

~~~
```javascript import React, { useState, useEffect } from 'react';
function FetchComponent() {    const [data, setData] =
useState(null);   useEffect(() => {     const fetchData = async ()
=> {       const response = await
~~~

```
fetch('https://api.example.com/data');       const data = await
response.json();       setData(data);
    };
    fetchData();
  }, []);
  if (!data) {
    return <div>Loading...</div>;
  }
  return (
    <div>
      <h1>{data.title}</h1>
      <p>{data.description}</p>
    </div>
  );
}
export default FetchComponent;
```

In this example, the `FetchComponent` component is used to fetch data from the `https://api.example.com/data` URL. The `data` state variable is then used to render the component, depending on the state of the network request. If the `data` variable is `null`, a loading message is displayed. Otherwise, the response data is displayed.

You can then use the `FetchComponent` component in any other component in your React application to fetch data from an API:

**.js file**

```
//The `FetchComponent` component is used inside the `MyComponent` component. When the `MyComponent` component is rendered, the `FetchComponent` component will be mounted and will fetch data from the API.


import FetchComponent from './FetchComponent';


function MyComponent() {
return <FetchComponent />;
}
```

## HANDLING MULTIPLE REQUESTS

Once we start making requests for data from the internet, we won’t be able to stop. More often than not, we need to make several HTTP requests to obtain all the data required to hydrate our application. For example, we’re currently asking GitHub to provide information about a user’s account. We’ll also need to obtain information about that user’s repositories. Both of these data points are obtained by making sepa- rate HTTP requests.

GitHub users typically have many repositories. Information about a user’s reposito- ries is passed as an array of objects. We’re going to create a special custom hook called useIterator that will allow us to iterate through any array of objects:

```
export const useIterator = ( items
= [],
initialIndex = 0
) => {
const [i, setIndex] = useState(initialIndex);
const prev = () => { if (i === 0) return
setIndex(items.length - 1); setIndex(i - 1);
```

```
 };
const next = () => { if (i === items.length - 1)
return setIndex(0); setIndex(i + 1);
};
return [items[i], prev, next];
};
```

This hook will allow us to cycle through any array. Because it returns items inside of an array, we can take advantage of array destructuring to give these values names that make sense:

```
const [letter, previous, next] = useIterator([
"a",
"b",
```

"c"

]);

In this case, the initial letter is “b.” If the user invokes next, the component will rerender, but this time, the value for letter will be “b.” Invoke next two more times, and the value for letter will once again be “a” because this iterator circles back around to the first item in the array instead of letting the index go out of bounds. The useIterator hook takes in an array of items and an initial index. The key value to this iterator hook is the index, i, which was created with the useState hook. i is used to identify the current item in the array. This hook returns the current item, item[i], as well as functions for iterating through that array: prev and next. Both the prev and next functions either decrement or increment the value of i by invoking setIndex. This action causes the hook to rerender with a new index.

## REACT ROUTER:

React Router is a standard library for routing in React. It enables the navigation among views of various components in a React Application, allows changing the browser URL, and keeps the UI in sync with the URL.

Let us create a simple application to React to understand how the React Router works. The application will contain three components: home component, about a component, and contact component. We will use React Router to navigate between these components.

**Setting up the React Application:** Create a React application using create-react-app.
Command: npx create-react-app project_name

**Installing React Router:** React Router can be installed via npm in your React application. Follow the steps given below to install Router in your React application:

Step 1: cd into your project directory.
Step 2: To install the React Router use the following command: npm install – -
save react-router-dom or npm i react-router-dom

**Adding React Router Components:** The main Components of React Router are:

**BrowserRouter**: BrowserRouter is a router implementation that uses the HTML5 history API(pushState, replaceState and the popstate event) to keep your UI in sync with the URL. It is the parent component that is used to store all of the other components.

**Routes:** It’s a new component introduced in the v6 and a upgrade of the component.

The main advantages of Routes over Switch are: Relative s and s

Routes are chosen based on the best match instead of being traversed in order.

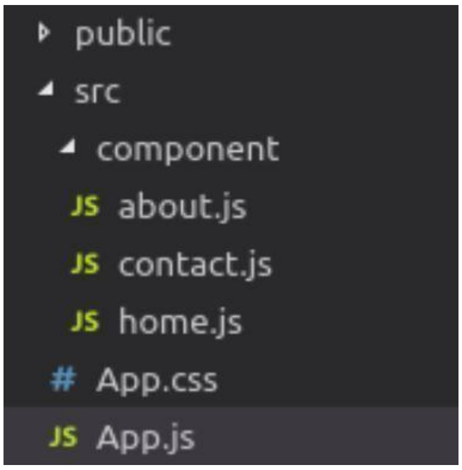
**Route**: Route is the conditionally shown component that renders some UI when its path matches the current URL.

**Link:** Link component is used to create links to different routes and implement navigation around the application. It works like HTML anchor tag.

To add React Router components in your application, open your project directory in the editor you use and go to app.js file. Now, add the below given code in app.js.

```
import {
   BrowserRouter as Router,
   Routes,
   Route,
   Link
} from 'react-router-dom';
```

**Using React Router:** To use React Router, let us first create few components in the react application. In your project directory, create a folder named component inside the src folder and now add 3 files named home.js, about.js and contact.js to the component folder. directory_structure



```
public
src
  component
    about.js
    contact.js
    home.js
  App.css
  App.js
```

Adding code to the 3 Components : **Home.js**

```
import React from 'react'; function Home (){ return
<h1>Welcome to the world of Geeks!</h1>
}     export     default
Home;
```

**About.js**

```
import React from 'react';
function About () {
return <div>
                  <h2>React Components</h2>
```

```
        </div> }
export default About;
```

**Contact.js**

```
import React from 'react'; function Contact (){ return <address>          Prasad V
Potluri Siddhartha Institute Of Technology </address>
}
export default Contact;
```

**Adding all the React Components in App.js**

```
import React, { Component } from 'react';
import { BrowserRouter as Router,Routes, Route, Link } from 'react-router-dom';
import Home from './component/home'; import About from './component/about'; import
Contact from './component/contact'; import './App.css';
class App extends Component {
render()    {
        return (
        <Router>
                <div className="App">
                        <ul className="App-header"> <li>
                                <Link to="/">Home</Link>
                        </li>
                        <li>
                                <Link to="/about">About Us</Link>
                        </li>
                        <li>
                                <Link to="/contact">Contact Us</Link>
                        </li>
                        </ul>
                <Routes>
                                <Route exact path='/' element={< Home />}></Route>
                                <Route exact path='/about' element={< About
                                        />}></Route>
                                <Route exact path='/contact' element={< Contact
                                        />}></Route>
                </Routes>
                </div>
        </Router>
        );
} } export default

App;
```

Now, we can click on the links Home, About, Contact and navigate to different components.

## MEMOZING VALUES:

React Memo is a higher-order component that wraps around a component to memoize the rendered output and avoid unnecessary renderings. This improves performance because it memoizes the result and skips rendering to reuse the last rendered result.

To achieve memoization, React provides PureComponent, memo, useMemo , and useCallback. Memoization improves performance by saving function results when the same prop is provided, minimizing the number of re-renderings.

**How do you memoize a React component?**

When a component is wrapped in React. memo() , React renders the component and memoizes the result. Before the next render, if the new props are the same, React reuses the memoized result, skipping the next rendering. export const MemoizedMovie = React.

useMemo() is a function that returns a memoized value of a passed in resourceintensive function. It is very useful in optimizing the performance of a React component by eliminating repeating heavy computations.

```
export function Movie({ title, releaseDate }) {
  return (
    <div>
      <div>Movie title: {title}</div>
      <div>Release date: {releaseDate}</div>
    </div>
  );
}       export       const       MemoizedMovie       =

React.memo(Movie);
```

React.memo(Movie) returns a new memoized component MemoizedMovie.

MemoizedMovie outputs the same content as the original Movie component, but with one difference — MemoizedMovie render is memoized. React reuses the memoized content as long as title and releaseDate props are the same between renderings:

```
// First render - MemoizedMovie IS INVOKED.
<MemoizedMovie title="Heat"
  releaseDate="December 15, 1995"
/>
// Second render - MemoizedMovie IS NOT INVOKED.
```

```
<MemoizedMovie title="Heat"
  releaseDate="December 15, 1995"
/>
```

## INTRO TO WEBPACK:

**Webpack:** Webpack is a static module bundler used for JavaScript applications. Since webpack understands only JavaScript and JSON files, It transforms front-end assets such as HTML, CSS, and images into valid modules if the corresponding loaders are included. While Processing your application webpack internally builds a dependency graph that maps every module your project needs and produces one or more output bundles.

**Why need webpack in React?**

Webpack simplifies the build process for React applications. It allows developers to manage dependencies and bundle files with ease. Webpack can also optimize code and assets for production, reducing the file size of the application and improving performance.

- Some core concepts of webpack are:
  - Entry
  - Output
  - Loaders
  - Plugins
  - Mode

**Entry:** An entry point defines which module webpack should use to start building out its internal dependency graph. The entry point's default value is ./src/index.js, but in the webpack configuration., you can specify a different or multiple entry points by setting an entry property within this file.

**Output:** The output property indicates webpack where to emit the bundles it creates and tells the way to name these files. By default, its value is ./dist/main.js for the main output file and it is ./dist folder for any other generated file, but we can change this part of the process by specifying an output field in our configuration.

**Loaders:** Since webpack only understands JavaScript and JSON files. Loaders process other types of files and after that, it converts them into the valid modules which can be consumed by our application, and add them to the dependency graph.

Loaders preprocess the other type of files and them to the bundle, Loaders have two properties in webpack configuration through which they achieve this:

The test property: It is used to identify which file or files should be transformed by the respective loader. Usually, a regular expression is used to identify the file or files which should be transformed.

The use property: It is used to indicate which loader should be used to do the transforming.

**Plugins:** While loaders are used to preprocess certain types of modules, plugins can be used to carry out a wider range of tasks like an injection of environment variables, asset management, and bundle optimization.

In order to use a plugin, we have to require() it and add it to the plugins array. Plugins can be customized through options. Since a plugin can be used multiple times in a configuration for different purposes, we need to create an instance of it by calling it with the new operator.

**Mode:** We can enable webpack's built-in optimizations that correspond to each environment by setting the mode parameter to either development, production, or none. Its default value is production. **Filename: webpack.config.js**

```
var HtmlWebpackPlugin = require('html-webpack-plugin'); var UglifyJsPlugin =
require('webpack/lib/optimize/UglifyJsPlugin'); var CommonsChunkPlugin =
require('webpack/lib/optimize/CommonsChunkPlugin');

module.exports = { entry: { vendor:
   ['babel-polyfill', 'lodash'], main:
   './src/main.js'
   }, output:
   {
      path: './dist', filename:
      'bundle.js'
   },
   module: {
      loaders: [
         { test: /\.jsx?$/, loader: 'babel', exclude: /node_modules/,
            query: { plugins: ['transform-runtime'], presets: ['es2015'] }
         },
         { test: /\.hbs$/, loader: 'handlebars' }
      ] },
   plugins: [
      new HtmlWebpackPlugin({ title:
         'Intro to Webpack',
         template: 'src/index.html'
      }),
      new UglifyJsPlugin({ beautify: false, mangle:
         { screw_ie8 : true },
            compress: { screw_ie8: true, warnings: false
            }, comments: false
        }),
        new CommonsChunkPlugin(/* chunkName= */"vendor", /* filename=
*/"vendor.bundle.js")
    ]
};
```