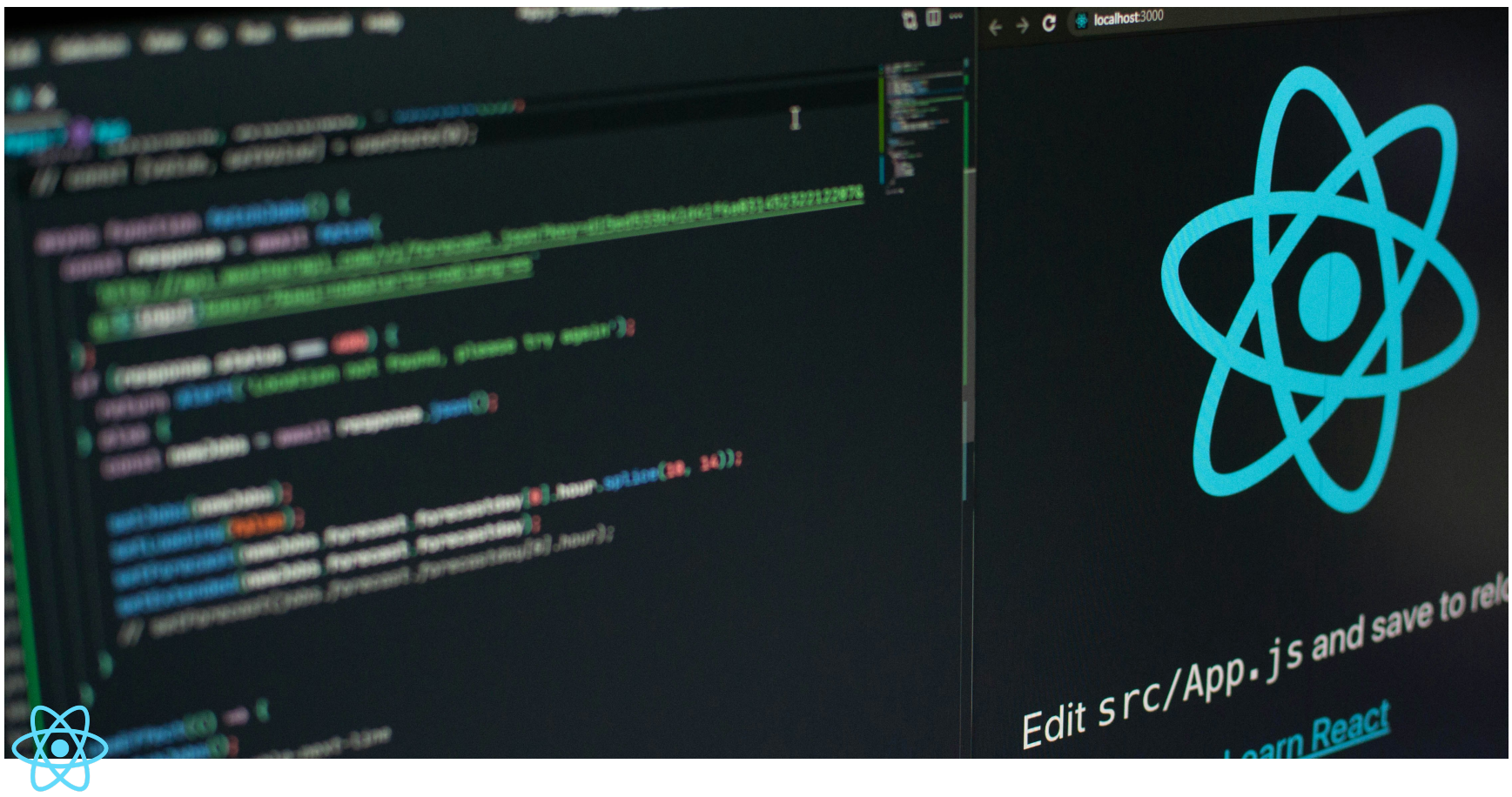


React JS Essentials

The library for web and native user interfaces

> 📝 Danniel Libor - 2024

Request access here

GitHub Repository

Table of contents

Introduction

🤔 What is React?

React is a JavaScript library created by Jordan Walke, Software Engineer at Facebook.

React lets you build user interfaces out of individual pieces called components.

It is designed to let you seamlessly combine components written by independent people, teams, and organizations.

Initial Release to the Public (React 0.3.0) was in July 2013.

React was first used in 2011 for Facebook's Newsfeed feature.

Current version of React is 18.0.0 (April 2022).

🧐 Why React?

React lets you build both web apps and native apps using the same skills.

With React, you can be a web *and* a native developer. Your team can ship to many platforms without sacrificing the user experience.

> 💡 While that may make it sound like it's easy to just translate your app from React to React Native, **it's not**. You essentially need to rebuild everything in React Native. React Native is a fine framework for native mobile apps.

Parameter	Angular	React	Vue
Initial Release	2016	2011	2014
Support	Google	Facebook	Community
Type	Framework	Library	Framework
Size	Medium	Small	Very small
Language	TypeScript	JavaScript	JavaScript
Performance	Good	Good	Good
Data Binding	Both	Unidirectional	Bidirectional
Learning Curve	Steep	Easy	Easy
Popular Websites	Paypal, Samsung, Upwork	Netflix, Twitter, Amazon	Alibaba, Grammarly, GitLab

Pros of React

React has established itself as a key library for building dynamic and responsive web applications for the following reasons:

- Component-Based Architecture
- Enhanced Customization in Development
- Future-Proof Choice for Developers
- Redux for State Management

Cons of React

While React provides plenty of advantages to developers of varying skill levels, it’s not without its respective drawbacks, including the following:

- Complex Concepts and Advanced Patterns
- Integration Complexity with Other Technologies
- Barrier for Non-JavaScript Developers
- Not a Full-Fledged Framework
- Code bloat

- Reduced performance on legacy devices and weak networks

▼ 😎 What Should You Already Know?

Before starting with React, you should have intermediate experience in:

- HTML
- CSS
- JavaScript

You should also have some experience with the new JavaScript features introduced in ECMAScript 6 (ES6).

Getting Started

▼ 🛠️ Tools You Need

- ☐ Visual Studio Code
  - ☐ vscode-icons
  - ☐ Code Spell Checker
  - ☐ Auto Rename Tag
  - ☐ Bracket Pair Color DLW
  - ☐ indent-rainbow
  - ☐ Document This
  - ☐ Colonize
  - ☐ DotENV
  - ☐ GitLens — Git supercharged

  - ☐ CSS Peek
  - ☐ HTML CSS Support
  - ☐ JavaScript (ES6) code snippets
  - ☐ ES7+ React/Redux/React-Native snippets
  - ☐ Import Cost
  - ☐ npm Intellisense
  - ☐ Tailwind CSS IntelliSense
  - ☐ Prettier - Code formatter
- ☐ Node
- ☐ Git
- ☐ Browser

▼ 📌 React Directly in HTML

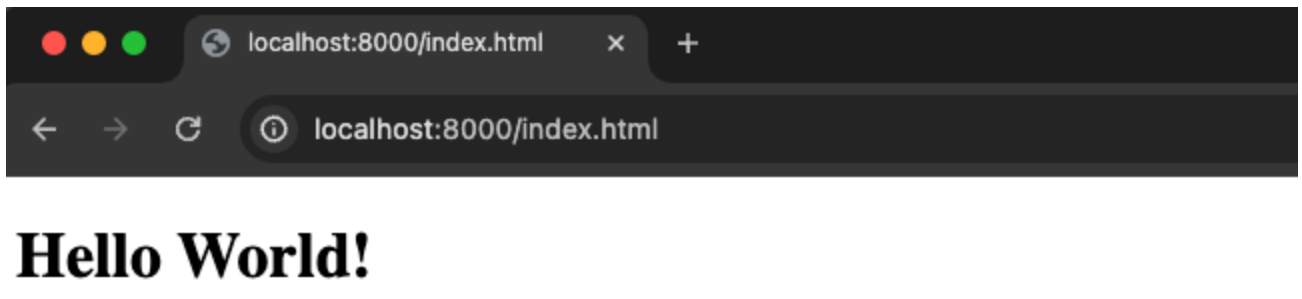
To get an overview of what React is, you can write React code directly in HTML.

> 💡 This way of using React can be OK for testing purposes, but for production you will need to set up a **React environment**.

Create a html file named `index.html` .

```
<!DOCTYPE html>
<html>
  <head>
    <script src="https://unpkg.com/react@18/umd/react.development.js" crossorigin></sc
    <script src="https://unpkg.com/react-dom@18/umd/react-dom.development.js" crossori
    <script src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
  </head>
  <body>

    <div id="root"></div>

    <script type="text/babel">
      function Hello() {
        return <h1>Hello World!</h1>;
      }

      const container = document.getElementById('root');
      const root = ReactDOM.createRoot(container);
      root.render(<Hello />)
    </script>

  </body>
</html>
```

Open your browser and type `localhost:3000` in the address bar.



React's goal is in many ways to render HTML in a web page.

React renders HTML to the web page by using a function called `createRoot()` and its method `render()`.

The createRoot Function

The `createRoot()` function takes one argument, an HTML element.

The purpose of the function is to define the HTML element where a React component should be displayed.

The render Method

The `render()` method is then called to define the React component that should be rendered.

You'll notice a single `<div>` in the body of this file. This is where our React application will be rendered.

The result is displayed in the `<div id="root">` element.

> 💡 Note that the element id does not have to be called "root", but this is the standard convention.

▼ The Root Node

The root node is the HTML element where you want to display the result.

It is like a *container* for content managed by React.

It does NOT have to be a `<div>` element and it does NOT have to have the `id='root'`

▼ 📌 React JSX

- Stands for JavaScript XML.
- Allows us to write HTML in React.
- Makes it easier to write and add HTML in React.
- Allows us to write HTML elements directly within JavaScript and place them in the DOM without any `createElement()` and/or `appendChild()` methods.
- Converts HTML tags into react elements.

> 💡 You are not required to use JSX, but JSX makes it easier to write React applications.

With JSX:

```
const myElement = <h1>I Love JSX!</h1>;

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```

Without JSX:

```
const myElement = React.createElement('h1', {}, 'I do not use JSX!');

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```

▼ Expressions in JSX

With JSX you can write expressions inside curly braces `{ }`.

The expression can be a React variable, or property, or any other valid JavaScript expression. JSX will execute the expression and return the result:

```
const myElement = <h1>React is {5 + 5} times better with JSX</h1>;
```

▼ Inserting a Large Block of HTML

To write HTML on multiple lines, put the HTML inside parentheses:

```
const myElement = (
  <ul>
    <li>Apples</li>
    <li>Bananas</li>
    <li>Cherries</li>
  </ul>
);
```

▼ One Top Level Element

The HTML code must be wrapped in *ONE* top level element.

So if you like to write *two* paragraphs, you must put them inside a parent element, like a `div` element.

```
const myElement = (
  <div>
    <p>I am a paragraph.</p>
    <p>I am a paragraph too.</p>
  </div>
);
```

💡 Alternatively, you can use a "fragment" to wrap multiple lines. This will prevent unnecessarily adding extra nodes to the DOM. A fragment looks like an empty HTML tag: `<></>`.

▼ Attribute class

The `class` attribute is a much used attribute in HTML, but since JSX is rendered as JavaScript, and the `class` keyword is a reserved word in JavaScript, you are not allowed to use it in JSX.

JSX solved this by using `className` instead. When JSX is rendered, it translates `className` attributes into `class` attributes.

```
const myElement = <h1 className="myclass">Hello World</h1>;
```

▼ Conditions

React supports `if` statements, but not *inside* JSX.

To be able to use conditional statements in JSX, you should put the `if` statements outside of the JSX, or you could use a ternary expression instead.

Option 1:

```
const x = 5;
let text = "Goodbye";
if (x < 10) {
  text = "Hello";
}

const myElement = <h1>{text}</h1>;
```

Option 2:

```
const x = 5;

const myElement = <h1>{(x) < 10 ? "Hello" : "Goodbye"}</h1>;
```

▼ 📌 Setting up a React Environment

Run this command to create a React application named `my-react-app`

```
npx create-react-app my-react-app
```

The `create-react-app` will set up everything you need to run a React application.

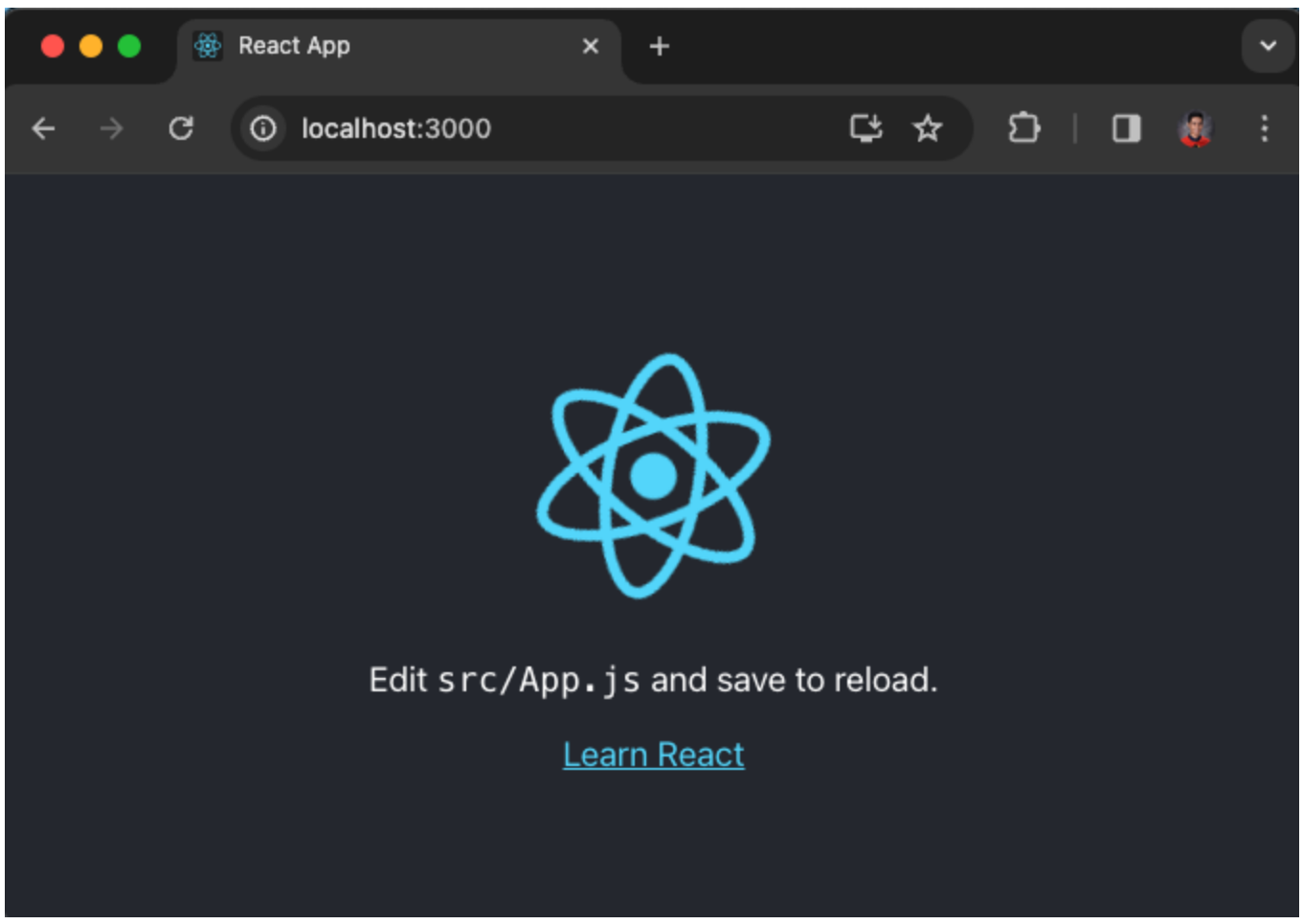
Run this command to move to the `my-react-app` directory:

```
cd my-react-app
```

Run this command to run the React application `my-react-app`:

```
npm start
```

A new browser window will pop up with your newly created React App! If not, open your browser and type `localhost:3000` in the address bar.



Vite

Next Generation Frontend Tooling

- 💡 Instant Server Start
- ⚡ Lightning Fast HMR
- 🛠️ Rich Features
- 📦 Optimized Build
- 🔩 Universal Plugin Interface
- 🔑 Fully Typed APIs

Vite (French word for "quick", pronounced `/vit/`, like "veet") is a new breed of frontend build tooling that significantly improves the frontend development experience. It consists of two major parts:

- A dev server that serves your source files over native ES modules, with rich built-in features and astonishingly fast Hot Module Replacement (HMR).
- A build command that bundles your code with Rollup, pre-configured to output highly optimized static assets for production.

In addition, Vite is highly extensible via its Plugin API and JavaScript API with full typing support.

```
npm create vite@latest
```

▼ 📌 React Components

Components are like functions that return HTML elements.

Components are independent and reusable bits of code. They serve the same purpose as JavaScript functions, but work in isolation and return HTML.

When creating a React component, the component's name *MUST* start with an upper case letter.

```
function Fruit() {
  return <h2>Hi, I am a Fruit!</h2>;
}

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

Now your React application has a component called Recipe, which returns an `<h2>` element.

React is all about re-using code, and it is recommended to split your components into separate files.

To do that, create a new file with a `.js` file extension and put the code inside it.

> 💡 Note that the filename must start with an uppercase character.

`Fruit.js`

```
function Fruit() {
  return <h2>Hi, I am a Fruit!</h2>;
}

export default Fruit;
```

`index.js`

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

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

We can refer to components inside other components:

```
import Component1 from './Component1.js';

function Fruit() {
  return (
      <>
          <h2>Hi, I am a Fruit!</h2>
          <Component1 />
      </>
  );
}

export default Fruit;
```

▼ 📌 React Props

Components can be passed as `props`, which stands for properties.

Props are like function arguments, and you send them into the component as attributes.

```
function Fruit(props) {
  return <h2>I am a {props.color} Fruit!</h2>;
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Fruit color="red"/>);
```

If you have a variable to send, and not a string as in the example above, you just put the variable name inside curly brackets.

```
function Fruit(props) {
  return <h2>I am a {props.color} Fruit!</h2>;
}

const colorName = "red";
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Fruit color={ colorName }/>);
```

Or if it was an object.

```
function Fruit(props) {
  return <h2>I am a {props.info.name} Fruit!</h2>;
}

const fruitInfo = { name: "Mango", color: "Yellow" };
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Fruit info={ fruitInfo }/>);
```

▼ 📌 React Events

Just like HTML DOM events, React can perform actions based on user events.

React has the same events as HTML: click, change, mouseover etc.

React events are written in camelCase syntax: `onClick` instead of `onclick`.

React event handlers are written inside curly braces: `onClick={shoot}` instead of `onclick="shoot()"`.

React

```
<button onClick={shoot}>Take the Shot!</button>
```

HTML

```
<button onclick="shoot()">Take the Shot!</button>
```

Example

```
function Football() {
  const shoot = () => {
    alert("Great Shot!");
  }

  return (
```

```
    <button onClick={shoot}>Take the shot!</button>);
}

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

To pass an argument to an event handler, use an arrow function.

```
function Football() {
  const shoot = (a) => {
    alert(a);
  }

  return (
    <button onClick={() => shoot("Goal!")}>Take the shot!</button>);
}

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

Event handlers have access to the React event that triggered the function.

```
function Football() {
  const shoot = (a, b) => {
    alert(b.type);
    /*
    'b' represents the React event that triggered the function,
    in this case the 'click' event
    */
  }

  return (
    <button onClick={(event) => shoot("Goal!", event)}>Take the shot!</button>);
}

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

▼ 📌 React Conditionals

In React, you can conditionally render components.

There are several ways to do this.

We can use the `if` JavaScript operator to decide which component to render.

```
function Goal(props) {
  const isGoal = props.isGoal;
  if (isGoal) {
    return <MadeGoal/>;
  }
  return <MissedGoal/>;
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Goal isGoal={false} />);
```

Another way to conditionally render a React component is by using the `&&` operator.

```
function Basket(props) {
  const fruits = props.fruits;
  return (
    <>
      <h1>Basket</h1>
      {fruits.length > 0 &&
        <h2>
          You have {fruits.length} fruits in your basket.
        </h2>}
    </>);
}

const fruits = ['Apple', 'Mango', 'Orange'];
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Basket fruits={fruits} />);
```

If `cars.length > 0` is equates to true, the expression after `&&` will render.

Another way to conditionally render elements is by using a ternary operator.

```
function Goal(props) {
  const isGoal = props.isGoal;
  return (
    <>
      { isGoal ? <MadeGoal/> : <MissedGoal/> }
    </>);
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Goal isGoal={false} />);
```

▼ 📌 React Lists

In React, you will render lists with some type of loop.

The JavaScript `map()` array method is generally the preferred method.

```
function Fruit(props) {
  return <li>{ props.name }</li>;
}

function Basket() {
  const fruits = ['Apple', 'Mango', 'Orange'];
  return (
    <>
      <ul>
        {fruits.map((fruit, index) => <Fruit key={index} name={fruit} />)}
      </ul>
    </>);
}

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

▼ 📌 React Forms

Just like in HTML, React uses forms to allow users to interact with the web page.

You add a form with React like any other element.

```
function Register() {
  return (
    <form>
      <label>Enter your name:
        <input type="text" />
      </label>
    </form>)
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Register />);
```

This will work as normal, the form will submit and the page will refresh.

But this is generally not what we want to happen in React.

We want to prevent this default behavior and let React control the form.

Handling forms is about how you handle the data when it changes value or gets submitted.

In HTML, form data is usually handled by the DOM.

In React, form data is usually handled by the components.

You can control the submit action by adding an event handler in the `onSubmit` attribute for the `<form>`

```
import ReactDOM from 'react-dom/client';

function Register() {
  const handleSubmit = (event) => {
    event.preventDefault();
    alert('You have just submitted a form!')
  }

  return (
    <form onSubmit={handleSubmit}>
      <label>Enter your name:
        <input type="text" />
      </label>
      <input type="submit" />
    </form>)
}

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

▼ 📌 React Router

Create React App doesn't include page routing.

React Router is the most popular solution.

To add React Router in your application, run this in the terminal from the root directory of the application

```
npm i -D react-router-dom
```

To create an application with multiple page routes, let's first start with the file structure.

Within the `src` folder, we'll create a folder named `pages` with several files.

Each file will contain a very basic React component.

▼ `src\index.js`

```
import ReactDOM from "react-dom/client";
import { BrowserRouter, Routes, Route } from "react-router-dom";
import Layout from "./pages/Layout";
import Login from "./pages/Login";
import Register from "./pages/Register";
import Dashboard from "./pages/Dashboard";
import NoPage from "./pages/NoPage";

export default function App() {
  return (
    <BrowserRouter>
      <Routes>
          <Route path="login" element={<Login />} />
        <Route path="register" element={<Register />} />

        <Route path="/" element={<Layout />}>
          <Route index element={<Dashboard />} />
          <Route path="*" element={<NoPage />} />
        </Route>
      </Routes>
    </BrowserRouter>);
}

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

▼ `src\pages\Layout.js`

```
import { Outlet, Link } from "react-router-dom";

const Layout = () => {
  return (
    <>
      <nav>
        <ul>
          <li>
            <Link to="/">Dashboard</Link>
          </li>
          <li>
            <Link to="/login">Login</Link>
          </li>
          <li>
            <Link to="/register">Register</Link>
          </li>
        </ul>
      </nav>

      <Outlet />
    </>
  );
};

export default Layout;
```

▼ `src\pages\Dashboard.js`

```
const Dashboard = () => {
  return <h1>Dashboard</h1>;
};

export default Dashboard;
```

▼ `src\pages\Login.js`

```
const Login = () => {
  return <h1>Login</h1>;
};

export default Login;
```

▼ `src\pages\Register.js`

```
const Register = () => {
  return <h1>Register</h1>;
};

export default Register;
```

▼ `src\pages\NoPage.js`

```
const NoPage = () => {
  return <h1>404</h1>;
};

export default NoPage;
```

We wrap our content first with `<BrowserRouter>`.

Then we define our `<Routes>`. An application can have multiple `<Routes>`. Our basic example only uses one.

`<Route>`s can be nested. The first `<Route>` has a path of `/` and renders the `Layout` component.

The nested `<Route>`s inherit and add to the parent route.

The `Dashboard` component route does not have a path but has an `index` attribute. That specifies this route as the default route for the parent route, which is `/`.

Setting the `path` to `*` will act as a catch-all for any undefined URLs. This is great for a 404 error page.

The `Layout` component has `<Outlet>` and `<Link>` elements.

The `<Outlet>` renders the current route selected.

`<Link>` is used to set the URL and keep track of browsing history.

Anytime we link to an internal path, we will use `<Link>` instead of `<a href="">`.

The "layout route" is a shared component that inserts common content on all pages, such as a navigation menu.

▼ 📌 React CSS Styling

There are many ways to style React with CSS, this tutorial will take a closer look at three common ways:

▼ Inline styling

To style an element with the inline style attribute, the value must be a JavaScript object.

Since the inline CSS is written in a JavaScript object, properties with hyphen separators, like `background-color`, must be written with camel case syntax.

Use `backgroundColor` instead of `background-color` .

```
const Login = () => {
  return <h1 style={{backgroundColor: "lightblue"}}>Login</h1>;
};

export default Login;
```

💡 In JSX, JavaScript expressions are written inside curly braces, and since JavaScript objects also use curly braces, the styling in the example above is written inside two sets of curly braces `{{}}`.

You can also create an object with styling information, and refer to it in the style attribute.

```
const Header = () => {
  const myStyle = {
    color: "white",
    backgroundColor: "DodgerBlue",
    padding: "10px",
    fontFamily: "Sans-Serif"
  };
  return (
    <>
      <h1 style={myStyle}>Hello Style!</h1>
      <p>Add a little style!</p>
    </>);
}
```

▼ CSS stylesheets

You can write your CSS styling in a separate file, just save the file with the `.css` file extension, and import it in your application.

Create a new file called "App.css" and insert some CSS code in it.

```
body {
  background-color: #282c34;
  color: white;
  padding: 40px;
  font-family: Sans-Serif;
  text-align: center;
}
```

`index.js`

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

💡 You can call the file whatever you like, just remember the correct file extension.

▼ CSS Modules

Another way of adding styles to your application is to use CSS Modules.

CSS Modules are convenient for components that are placed in separate files.

> 💡 The CSS inside a module is available only for the component that imported it, and you do not have to worry about name conflicts.

Create a new file called "my-style.module.css" and insert some CSS code in it.

```
.bigblue {
  color: DodgerBlue;
  padding: 40px;
  font-family: Sans-Serif;
  text-align: center;
}
```

Import the stylesheet in your component.

```
import styles from './my-style.module.css';

const Dashboard = () => {
  return <h1 className={styles.bigblue}>Dashboard</h1>;
}

export default Dashboard;
```

▼ Tailwind CSS

All of the components in Tailwind UI are designed for the latest version of Tailwind CSS, which is currently Tailwind CSS v3.4. To make sure that you are on the latest version of Tailwind, update via npm:

```
npm install -D tailwindcss postcss autoprefixer
```

```
npx tailwindcss init -p
```

Install `tailwindcss` and its peer dependencies, then generate your `tailwind.config.js` and `postcss.config.js` files.

Add the paths to all of your template files in your `tailwind.config.js` file.

```
/** @type {import('tailwindcss').Config} */
export default {
  content: [
    "./index.html",
    "./src/**/*.{js,ts,jsx,tsx}",
  ],
  theme: {
    extend: {},
  },
  plugins: [],
}
```

Add the `@tailwind` directives for each of Tailwind's layers to your `./src/index.css` file.

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

Tailwind UI for React depends on **Headless UI** to power all of the interactive behavior and **Heroicons** for icons, so you'll need to add these two libraries to your project:

```
npm install @headlessui/react @heroicons/react
```

> 💡 These libraries and Tailwind UI itself all require React >= 16.

React Hooks

🪝 What is a Hook?

Hooks allow function components to have access to state and other React features.

Hooks allow us to "hook" into React features such as state and lifecycle methods.

There are 3 rules for hooks:

- Hooks can only be called inside React function components.
- Hooks can only be called at the top level of a component.
- Hooks cannot be conditional

🪝 React `useState` Hook

The React `useState` Hook allows us to track state in a function component.

State generally refers to data or properties that need to be tracking in an application.

To use the `useState` Hook, we first need to `import` it into our component.

```
import { useState } from "react";
```

We initialize our state by calling `useState` in our function component.

`useState` accepts an initial state and returns two values:

- The current state.
- A function that updates the state.

```
import { useState } from "react";

function Register() {
  const [name, setName] = useState("");
  const [email, setEmail] = useState("");

    return (
        <>
            <p>My name is {name}</p>
            <p>My email is {email}</p>
        </>
    )
}

export default Register;
```

The first value, `color`, is our current state.

The second value, `setColor`, is the function that is used to update our state.

Lastly, we set the initial state to an empty string: `useState("")` .

We can now include our state anywhere in our component.

> 💡 These names are variables that can be named anything you would like.

To update our state, we use our state updater function.

```
// ...

<button type="button" onClick={() => setName("John Doe")}>
  Insert Name
</button>

// ...
```

The `useState` Hook can be used to keep track of strings, numbers, booleans, arrays, objects, and any combination of these!

We could create multiple state Hooks to track individual values.

```
// ...
const [name, setName] = useState("John Doe");
const [email, setEmail] = useState("johndoe@mail.test");
const [password, setPassword] = useState("P@ssw0rd");
const [password_confirmation, setPasswordConfirmation] = useState("P@ssw0rd");
// ...

return (
    // ...
    <input value={name}/>
    // ...
)

// ...
```

Or, we can just use one state and include an object instead!

```
// ...
const [form, setForm] = useState({
  name: "John Doe",
  email: "johndoe@mail.test",
  password: "P@ssw0rd",
  password_confirmation: "P@ssw0rd"
});

// ...

return (
    // ...
    <input value={form.name}/>
    // ...
)
```

```
// ...
```

Since we are now tracking a single object, we need to reference that object and then the property of that object when rendering the component. (Ex: `form.name`)

When state is updated, the entire state gets overwritten.

What if we only want to update the email of our form?

If we only called `setForm({email: "janedoe@mail.test"})`, this would remove the name, password and password_confirmation from our state.

We can use the JavaScript spread operator to help us.

```
// ...
const updateEmail = () => {
  setForm(previousState => {
    return { ...previousState, email: "janedoe@mail.test" }
  });
}
// ...

return (
    // ...
    <button type="button" onClick={updateEmail}>
    Update Email
  </button>
  // ...
)

// ...
```

Because we need the current value of state, we pass a function into our `setForm` function. This function receives the previous value.

We then return an object, spreading the `previousState` and overwriting only the color.

🪝 React `useEffect` Hook

The `useEffect` Hook allows you to perform side effects in your components.

Some examples of side effects are: fetching data, directly updating the DOM, and timers.

`useEffect` accepts two arguments. The second argument is optional.

`useEffect(<function>, <dependency>)`

Use `setTimeout()` to count 1 second after initial render:

```
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've rendered {count} times!</h1>;
```

```
}

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

But wait!! It keeps counting even though it should only count once!

`useEffect` runs on every render. That means that when the count changes, a render happens, which then triggers another effect.

This is not what we want. There are several ways to control when side effects run.

We should always include the second parameter which accepts an array. We can optionally pass dependencies to `useEffect` in this array.

1. No dependency passed:

```
useEffect(() => {
  //Runs on every render
});
```

2. An empty array:

```
useEffect(() => {
  //Runs only on the first render
}, []);
```

3. Props or state values:

```
useEffect(() => {
  //Runs on the first render
  //And any time any dependency value changes
}, [prop, state]);
```

So, to fix this issue, let's only run this effect on the initial render.

Only run the effect on the initial render:

```
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);
  }, []); // <- add empty brackets here

  return <h1>I've rendered {count} times!</h1>;
}

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

Here is an example of a `useEffect` Hook that is dependent on a variable. If the `count` variable updates, the effect will run again:

```
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 />);
```

If there are multiple dependencies, they should be included in the `useEffect` dependency array.

🪝 React `useContext` Hook

React Context is a way to manage state globally.

It can be used together with the `useState` Hook to share state between deeply nested components more easily than with `useState` alone.

State should be held by the highest parent component in the stack that requires access to the state.

To illustrate, we have many nested components. The component at the top and bottom of the stack need access to the state.

To do this without Context, we will need to pass the state as "props" through each nested component. This is called "prop drilling".

```
import { useState } from "react";

function Component1() {
  const [user, setUser] = useState("Jesse Hall");

  return (
    <>
      <h1>{`Hello ${user}!`}</h1>
      <Component2 user={user} />
    </>);
}

function Component2({ user }) {
  return (
    <>
      <h1>Component 2</h1>
      <Component3 user={user} />
    </>);
}
```

```
function Component3({ user }) {
  return (
    <>
      <h1>Component 3</h1>
      <h2>{`Hello ${user} again!`}</h2>
    </>);
}

export default Component1;
```

To create context, you must Import `createContext` and initialize it.

Next we'll use the Context Provider to wrap the tree of components that need the state Context.

Wrap child components in the Context Provider and supply the state value.

```
import { useState, createContext, useContext } from "react"

const UserContext = createContext();

function Component1() {
  const [user, setUser] = useState("Jesse Hall");

  return (
    <UserContext.Provider value={user}>
      <h1>{`Hello ${user}!`}</h1>
      <Component2 />
    </UserContext.Provider>
  );
}

function Component2() {
  return (
    <>
      <h1>Component 2</h1>
      <Component3 />
    </>
  );
}

function Component3() {
  const user = useContext(UserContext);

  return (
    <>
      <h1>Component 3</h1>
      <h2>{`Hello ${user} again!`}</h2>
    </>
  );
}

export default Component1;
```

▼ 🪝 React `useRef` Hook

The `useRef` Hook allows you to persist values between renders.

It can be used to store a mutable value that does not cause a re-render when updated.

It can be used to access a DOM element directly.

If we tried to count how many times our application renders using the `useState` Hook, we would be caught in an infinite loop since this Hook itself causes a re-render.

To avoid this, we can use the `useRef` Hook.

Use `useRef` to track application renders.

```
// ...
const [inputValue, setInputValue] = useState("");
const count = useRef(0);

useEffect(() => {
  count.current = count.current + 1;
});

return (
  <>
    <input
      type="text"
      value={inputValue}
      onChange={(e) => setInputValue(e.target.value)}
    />
    <h1>Render Count: {count.current}</h1>
  </>
);
// ...
```

`useRef()` only returns one item. It returns an Object called `current`.

When we initialize `useRef` we set the initial value: `useRef(0)`.

> 💡 It's like doing this: `const count = {current: 0}`. We can access the count by using `count.current`.

In general, we want to let React handle all DOM manipulation.

But there are some instances where `useRef` can be used without causing issues.

In React, we can add a `ref` attribute to an element to access it directly in the DOM.

Use `useRef` to focus the input:

```
// ...
const inputElement = useRef();

const focusInput = () => {
  inputElement.current.focus();
};

return (
  <>
    <input type="text" ref={inputElement} />
    <button onClick={focusInput}>Focus Input</button>
  </>
);
// ...
```

The `useRef` Hook can also be used to keep track of previous state values.

This is because we are able to persist `useRef` values between renders.

Use `useRef` to keep track of previous state values:

```
// ...
const [inputValue, setInputValue] = useState("");
const previousInputValue = useRef("");

useEffect(() => {
  previousInputValue.current = inputValue;
}, [inputValue]);

return (
  <>
    <input
        type="text"
        value={inputValue}onChange={(e) => setInputValue(e.target.value)}
      />
    <h2>Current Value: {inputValue}</h2>
    <h2>Previous Value: {previousInputValue.current}</h2>
  </>
);
// ...
```

This time we use a combination of `useState`, `useEffect`, and `useRef` to keep track of the previous state.

In the `useEffect`, we are updating the `useRef` current value each time the `inputValue` is updated by entering text into the input field.

🪝 React `useReducer` Hook

The `useReducer` Hook is similar to the `useState` Hook.

It allows for custom state logic.

If you find yourself keeping track of multiple pieces of state that rely on complex logic, `useReducer` may be useful.

The useReducer Hook accepts two arguments. `useReducer(<reducer>, <initialState>)`

The `reducer` function contains your custom state logic and the `initialState` can be a simple value but generally will contain an object.

The `useReducer` Hook returns the current `state` and a `dispatch` method.

```
import { useReducer } from "react";
import ReactDOM from "react-dom/client";

const initialTodos = [
  {
    id: 1,
    title: "Todo 1",
    complete: false,
  },
  {
    id: 2,
    title: "Todo 2",
    complete: false,
  },
];

const reducer = (state, action) => {
```

```
  switch (action.type) {
    case "COMPLETE":
      return state.map((todo) => {
        if (todo.id === action.id) {
          return { ...todo, complete: !todo.complete };
        } else {
          return todo;
        }
      });
    default:
      return state;
  }
};

function Todos() {
  const [todos, dispatch] = useReducer(reducer, initialTodos);

  const handleComplete = (todo) => {
    dispatch({ type: "COMPLETE", id: todo.id });
  };

  return (
    <>
      {todos.map((todo) => (
        <div key={todo.id}>
          <label>
            <input
              type="checkbox"
              checked={todo.complete}
              onChange={() => handleComplete(todo)}
            />
            {todo.title}
          </label>
        </div>
      ))}
    </>
  );
}

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

This is just the logic to keep track of the todo complete status.

All of the logic to add, delete, and complete a todo could be contained within a single `useReducer` Hook by adding more actions.

▼ ❗ React Memo

Using `memo` will cause React to skip rendering a component if its props have not changed.

This can improve performance.

In this example, the `Todos` component re-renders even when the todos have not changed.

`index.js`

```
import { useState } from "react";
import ReactDOM from "react-dom/client";
import Todos from "./Todos";

const App = () => {
```

```
  const [count, setCount] = useState(0);
  const [todos, setTodos] = useState(["todo 1", "todo 2"]);

  const increment = () => {
    setCount((c) => c + 1);
  };

  return (
    <>
      <Todos todos={todos} />
      <hr />
      <div>
        Count: {count}
        <button onClick={increment}>+</button>
      </div>
    </>);
};

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

`Todos.js`

```
const Todos = ({ todos }) => {
  console.log("child render");
  return (
    <>
      <h2>My Todos</h2>
      {todos.map((todo, index) => {
        return <p key={index}>{todo}</p>;
      })}
    </>);
};

export default Todos;
```

When you click the increment button, the `Todos` component re-renders.

If this component was complex, it could cause performance issues.

To fix this, we can use `memo`.

Use `memo` to keep the `Todos` component from needlessly re-rendering.

Wrap the `Todos` component export in `memo`:

`index.js`

```
import { useState } from "react";
import ReactDOM from "react-dom/client";
import Todos from "./Todos";

const App = () => {
  const [count, setCount] = useState(0);
  const [todos, setTodos] = useState(["todo 1", "todo 2"]);

  const increment = () => {
    setCount((c) => c + 1);
  };
```

```
  return (
    <>
      <Todos todos={todos} />
      <hr />
      <div>
        Count: {count}
        <button onClick={increment}>+</button>
      </div>
    </>);
};

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

`Todos.js`

```
import { memo } from "react";

const Todos = ({ todos }) => {
  console.log("child render");
  return (
    <>
      <h2>My Todos</h2>
      {todos.map((todo, index) => {
        return <p key={index}>{todo}</p>;
      })}
    </>);
};

export default memo(Todos);
```

Now the `Todos` component only re-renders when the `todos` that are passed to it through props are updated.

▼ 🪝 React `useCallback` Hook

The React `useCallback` Hook returns a memoized callback function.

Think of memoization as caching a value so that it does not need to be recalculated.

This allows us to isolate resource intensive functions so that they will not automatically run on every render.

The `useCallback` Hook only runs when one of its dependencies update.

This can improve performance.

One reason to use `useCallback` is to prevent a component from re-rendering unless its props have changed.

In this example, you might think that the `Todos` component will not re-render unless the `todos` change:

`index.js`

```
import { useState } from "react";
import ReactDOM from "react-dom/client";
import Todos from "./Todos";

const App = () => {
  const [count, setCount] = useState(0);
  const [todos, setTodos] = useState([]);

  const increment = () => {
    setCount((c) => c + 1);
  };
  const addTodo = () => {
```

```
    setTodos((t) => [...t, "New Todo"]);
  };

  return (
    <>
      <Todos todos={todos} addTodo={addTodo} />
      <hr />
      <div>
        Count: {count}
        <button onClick={increment}>+</button>
      </div>
    </>);
};

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

`Todos.js`

```
import { memo } from "react";

const Todos = ({ todos, addTodo }) => {
  console.log("child render");
  return (
    <>
      <h2>My Todos</h2>
      {todos.map((todo, index) => {
        return <p key={index}>{todo}</p>;
      })}
      <button onClick={addTodo}>Add Todo</button>
    </>);
};

export default memo(Todos);
```

You will notice that the `Todos` component re-renders even when the `todos` do not change.

Why does this not work? We are using `memo`, so the `Todos` component should not re-render since neither the `todos` state nor the `addTodo` function are changing when the count is incremented.

This is because of something called "referential equality".

Every time a component re-renders, its functions get recreated. Because of this, the `addTodo` function has actually changed.

To fix this, we can use the `useCallback` hook to prevent the function from being recreated unless necessary.

Use the `useCallback` Hook to prevent the `Todos` component from re-rendering needlessly:

`index.js`

```
import { useState, useCallback } from "react";
import ReactDOM from "react-dom/client";
import Todos from "./Todos";

const App = () => {
  const [count, setCount] = useState(0);
  const [todos, setTodos] = useState([]);

  const increment = () => {
    setCount((c) => c + 1);
```

```
  };
  const addTodo = useCallback(() => {
    setTodos((t) => [...t, "New Todo"]);
  }, [todos]);

  return (
    <>
      <Todos todos={todos} addTodo={addTodo} />
      <hr />
      <div>
        Count: {count}
        <button onClick={increment}>+</button>
      </div>
    </>);
};

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

`Todos.js`

```
import { memo } from "react";

const Todos = ({ todos, addTodo }) => {
  console.log("child render");
  return (
    <>
      <h2>My Todos</h2>
      {todos.map((todo, index) => {
        return <p key={index}>{todo}</p>;
      })}
      <button onClick={addTodo}>Add Todo</button>
    </>);
};

export default memo(Todos);
```

Now the `Todos` component will only re-render when the `todos` prop changes.

🪝 React `useMemo` Hook

The React `useMemo` Hook returns a memoized value.

The `useMemo` Hook only runs when one of its dependencies update.

This can improve performance.

The `useMemo` and `useCallback` Hooks are similar. The main difference is that `useMemo` returns a memoized value and `useCallback` returns a memoized function.

The `useMemo` Hook can be used to keep expensive, resource intensive functions from needlessly running.

In this example, we have an expensive function that runs on every render.

When changing the count or adding a todo, you will notice a delay in execution.

A poor performing function. The `expensiveCalculation` function runs on every render:

```
import { useState } from "react";
import ReactDOM from "react-dom/client";

const App = () => {
```

```
  const [count, setCount] = useState(0);
  const [todos, setTodos] = useState([]);
  const calculation = expensiveCalculation(count);

  const increment = () => {
    setCount((c) => c + 1);
  };
  const addTodo = () => {
    setTodos((t) => [...t, "New Todo"]);
  };

  return (
    <div>
      <div>
        <h2>My Todos</h2>
        {todos.map((todo, index) => {
          return <p key={index}>{todo}</p>;
        })}
        <button onClick={addTodo}>Add Todo</button>
      </div>
      <hr />
      <div>
        Count: {count}
        <button onClick={increment}>+</button>
        <h2>Expensive Calculation</h2>
        {calculation}
      </div>
    </div>);
};

const expensiveCalculation = (num) => {
  console.log("Calculating...");
  for (let i = 0; i < 1000000000; i++) {
    num += 1;
  }
  return num;
};

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

To fix this performance issue, we can use the `useMemo` Hook to memoize the `expensiveCalculation` function. This will cause the function to only run when needed.

We can wrap the expensive function call with `useMemo`.

The `useMemo` Hook accepts a second parameter to declare dependencies. The expensive function will only run when its dependencies have changed.

In the following example, the expensive function will only run when `count` is changed and not when todo's are added.

```
import { useState, useMemo } from "react";
import ReactDOM from "react-dom/client";

const App = () => {
  const [count, setCount] = useState(0);
  const [todos, setTodos] = useState([]);
  const calculation = useMemo(() => expensiveCalculation(count), [count]);
```

```
  const increment = () => {
    setCount((c) => c + 1);
  };
  const addTodo = () => {
    setTodos((t) => [...t, "New Todo"]);
  };

  return (
    <div>
      <div>
        <h2>My Todos</h2>
        {todos.map((todo, index) => {
          return <p key={index}>{todo}</p>;
        })}
        <button onClick={addTodo}>Add Todo</button>
      </div>
      <hr />
      <div>
        Count: {count}
        <button onClick={increment}>+</button>
        <h2>Expensive Calculation</h2>
        {calculation}
      </div>
    </div>);
};

const expensiveCalculation = (num) => {
  console.log("Calculating...");
  for (let i = 0; i < 1000000000; i++) {
    num += 1;
  }
  return num;
};

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

▼ 🪝 React Custom Hooks

Hooks are reusable functions.

When you have component logic that needs to be used by multiple components, we can extract that logic to a custom Hook.

Custom Hooks start with "use". Example: `useFetch`.

In the following code, we are fetching data in our `Home` component and displaying it.

We will use the JSONPlaceholder service to fetch fake data. This service is great for testing applications when there is no existing data.

Use the JSONPlaceholder service to fetch fake "todo" items and display the titles on the page:

`index.js`

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

const Home = () => {
  const [data, setData] = useState(null);

  useEffect(() => {
    fetch("https://jsonplaceholder.typicode.com/todos")
```

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

  return (
    <>
      {data &&
        data.map((item) => {
          return <p key={item.id}>{item.title}</p>;
        })}
    </>);
};

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

The fetch logic may be needed in other components as well, so we will extract that into a custom Hook.

Move the fetch logic to a new file to be used as a custom Hook:

useFetch.js

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

const useFetch = (url) => {
  const [data, setData] = useState(null);

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

  return [data];
};

export default useFetch;
```

index.js

```
import ReactDOM from "react-dom/client";
import useFetch from "./useFetch";

const Home = () => {
  const [data] = useFetch("https://jsonplaceholder.typicode.com/todos");

  return (
    <>
      {data &&
        data.map((item) => {
          return <p key={item.id}>{item.title}</p>;
        })}
    </>);
};

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

We have created a new file called `useFetch.js` containing a function called `useFetch` which contains all of the logic needed to fetch our data.

We removed the hard-coded URL and replaced it with a `url` variable that can be passed to the custom Hook.

Lastly, we are returning our data from our Hook.

In `index.js`, we are importing our `useFetch` Hook and utilizing it like any other Hook. This is where we pass in the URL to fetch data from.

Now we can reuse this custom Hook in any component to fetch data from any URL.

State Management using Redux

Redux is a JS library for predictable and maintainable global state management.

It helps you write applications that behave consistently, run in different environments (client, server, and native), and are easy to test. On top of that, it provides a great developer experience, such as live code editing combined with a time traveling debugger.

You can use Redux together with React, or with any other view library. It is tiny (2kB, including dependencies), but has a large ecosystem of addons available.

Redux is a state container that helps manage the state of your application in a consistent and predictable manner. It follows a few key principles:

1. **Single Source of Truth**:
   - The state of your entire application is stored in a single JavaScript object known as the "store".
   - This makes it easier to manage and debug the state of your application, as all the data is centralized.
2. **State is Read-Only**:
   - The only way to change the state is to emit an "action", which is a plain JavaScript object that describes what happened.
   - This ensures that the state can only be changed in a controlled manner.
3. **Changes are Made with Pure Functions**:
   - To specify how the state tree is transformed by actions, you write pure functions called "reducers".
   - A reducer takes the previous state and an action, and returns the next state.

Key Concepts

1. **Store**:
   - The store holds the entire state of the application. It is created using `createStore` function from Redux.
   - Example:

```
import { createStore } from 'redux';
import rootReducer from './reducers';

const store = createStore(rootReducer);
```

2. **Actions**:

- Actions are plain JavaScript objects that represent events happening in the application.
- An action must have a `type` property and can have additional data.
- Example:

```
const increment = {
  type: 'INCREMENT',
};
```

```
const addTodo = {
  type: 'ADD_TODO',
  payload: {
    text: 'Learn Redux',
  },
};
```

3. **Reducers**:

- Reducers are functions that specify how the state changes in response to an action.
- A reducer takes the current state and an action as arguments and returns a new state.
- Example:

```
const initialState = { count: 0 };

function counterReducer(state = initialState, action) {
  switch (action.type) {
    case 'INCREMENT':
      return { ...state, count: state.count + 1 };
    case 'DECREMENT':
      return { ...state, count: state.count - 1 };
    default:
      return state;
  }
}
```

4. **Dispatch**:

- The `dispatch` function is used to send actions to the store.
- This triggers the reducer to update the state based on the action.
- Example:

```
store.dispatch({ type: 'INCREMENT' });
```

5. **Selectors**:

- Selectors are functions that extract specific pieces of data from the state.
- They help in keeping the state shape decoupled from the component logic.
- Example:

```
const selectCount = (state) => state.count;

const count = selectCount(store.getState());
```

Step-by-Step Example

1. **Install Redux and React-Redux**

Install Redux and React-Redux using npm:

```
npm install redux react-redux
```

2. **Create Redux Files**

Create a new folder structure for your Redux files:

```
src/
  └── redux/
```

```
        ├── actions.js
        ├── reducers.js
        └── store.js
```

3. **Define Actions**

Create the `actions.js` file to define the action types and action creators:

`src/redux/actions.js`:

```
// Action Types
export const INCREMENT = 'INCREMENT';
export const DECREMENT = 'DECREMENT';

// Action Creators
export const increment = () => ({
  type: INCREMENT,
});

export const decrement = () => ({
  type: DECREMENT,
});
```

4. **Define Reducers**

Create the `reducers.js` file to define the reducer function:

`src/redux/reducers.js`:

```
import { INCREMENT, DECREMENT } from './actions';

const initialState = {
  count: 0,
};

const counterReducer = (state = initialState, action) => {
  switch (action.type) {
    case INCREMENT:
      return {
        ...state,
        count: state.count + 1
      };
    case DECREMENT:
      return {
        ...state,
        count: state.count - 1
      };
    default:
      return state;
  }
};

export default counterReducer;
```

5. **Configure the Store**

Create the `store.js` file to configure the Redux store:

`src/redux/store.js`:

```
import { createStore } from 'redux';
import counterReducer from './reducers';
```

```
const store = createStore(counterReducer);

export default store;
```

6. **Set Up the Provider**

Wrap your application with the `Provider` component to make the Redux store available to your components:

`src/index.js`:

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import store from './redux/store';
import App from './App';
import './index.css';

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
);
```

7. **Create the Counter Component**

Create a Counter component that connects to the Redux store:

`src/components/Counter.js`:

```
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { increment, decrement } from '../redux/actions';

const Counter = () => {
  const count = useSelector((state) => state.count);
  const dispatch = useDispatch();

  return (
    <div className="flex flex-col items-center justify-center
    min-h-screen bg-gray-100">
      <h1 className="text-4xl font-bold mb-4">Counter: {count}</h1>
      <div>
        <button
          className="bg-blue-500 text-white px-4 py-2 rounded-lg m-2"
          onClick={() => dispatch(increment())}
        >
          Increment
        </button>
        <button
          className="bg-red-500 text-white px-4 py-2 rounded-lg m-2"
          onClick={() => dispatch(decrement())}
        >
          Decrement
        </button>
      </div>
    </div>
  );
};

export default Counter;
```

8. **Update the App Component**

Update the App component to include the Counter component:

`src/App.js`:

```
import React from 'react';
import Counter from './components/Counter';

function App() {
  return (
    <div className="App">
      <Counter />
    </div>
  );
}

export default App;
```

Introduction to Next.js

Next.js is a React framework for building full-stack web applications. You use React Components to build user interfaces, and Next.js for additional features and optimizations.

Under the hood, Next.js also abstracts and automatically configures tooling needed for React, like bundling, compiling, and more. This allows you to focus on building your application instead of spending time with configuration.

Whether you're an individual developer or part of a larger team, Next.js can help you build interactive, dynamic, and fast React applications.

Used by some of the world's largest companies, Next.js enables you to create full-stack web applications by extending the latest React features, and integrating powerful Rust-based JavaScript tooling for the fastest builds.

Some of the main Next.js features include:

Feature	Description
Routing	A file-system based router built on top of Server Components that supports layouts, nested routing, loading states, error handling, and more.
Rendering	Client-side and Server-side Rendering with Client and Server Components. Further optimized with Static and Dynamic Rendering on the server with Next.js. Streaming on Edge and Node.js runtimes.
Data Fetching	Simplified data fetching with async/await in Server Components, and an extended `fetch` API for request memoization, data caching and revalidation.
Styling	Support for your preferred styling methods, including CSS Modules, Tailwind CSS, and CSS-in-JS
Optimizations	Image, Fonts, and Script Optimizations to improve your application's Core Web Vitals and User Experience.
TypeScript	Improved support for TypeScript, with better type checking and more efficient compilation, as well as custom TypeScript Plugin and type checker.

Contact Me

🌐 LinkedIn

✉️ Email: dlibor.dev@gmail.com

📞 09569354075