**Name: Gurjit Singh Sid:N01634963**

**Web Development And Framework 2**

**Learn React Foundations - Chapter 1- 7  
  
Chapter 1**

**About React and Next.js**Next.js is a flexible React framework that gives you building blocks to create fast, full-stack web applications.  
  
[**Building blocks of a web application**](https://nextjs.org/learn/react-foundations/what-is-react-and-nextjs#building-blocks-of-a-web-application)

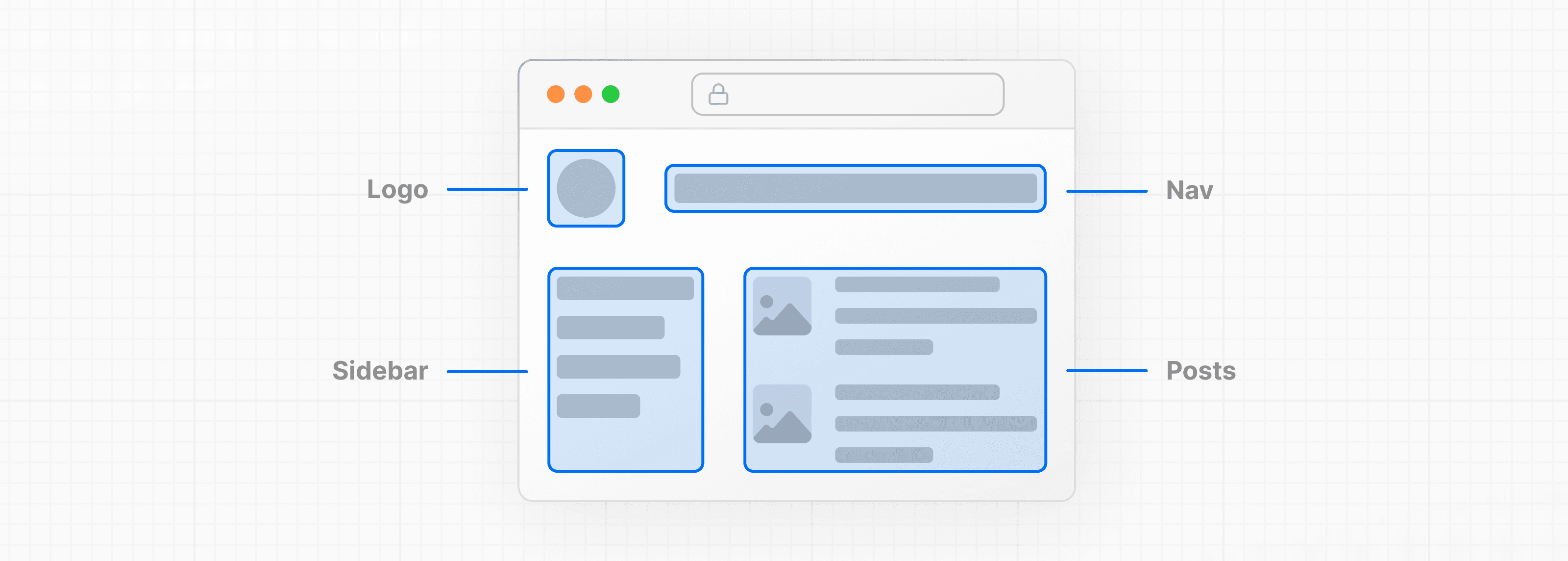
There are a few things you need to consider when building modern applications. Such as:

* **User Interface**- how users will consume and interact with your application.
* **Routing -** how users navigate between different parts of your application.
* **Data Fetching -** where your data lives and how to get it.
* **Rendering -** when and where you render static or dynamic content.
* **Integrations -** what third-party services you use (for CMS, auth, payments, etc.) and how you connect to them.
* **Infrastructure -** where you deploy, store, and run your application code (serverless, CDN, edge, etc.).
* **Performance -** how to optimize your application for end-users.
* **Scalability -** how your application adapts as your team, data, and traffic grow.
* **Developer Experience -** your team's experience building and maintaining your application.

For each part of your application, you will need to decide whether you will build a solution yourself or use other tools, such as packages, libraries, and frameworks.  
  
  
[**What is React?**](https://nextjs.org/learn/react-foundations/what-is-react-and-nextjs#what-is-react)

[React](https://react.dev/) is a JavaScript **library** for building **interactive user interfaces**.

By user interfaces (UI), we mean the elements that users see and interact with on-screen.



By library, we mean React provides helpful functions (APIs) to build UI, but leaves it up to the developer where to use those functions in their application.

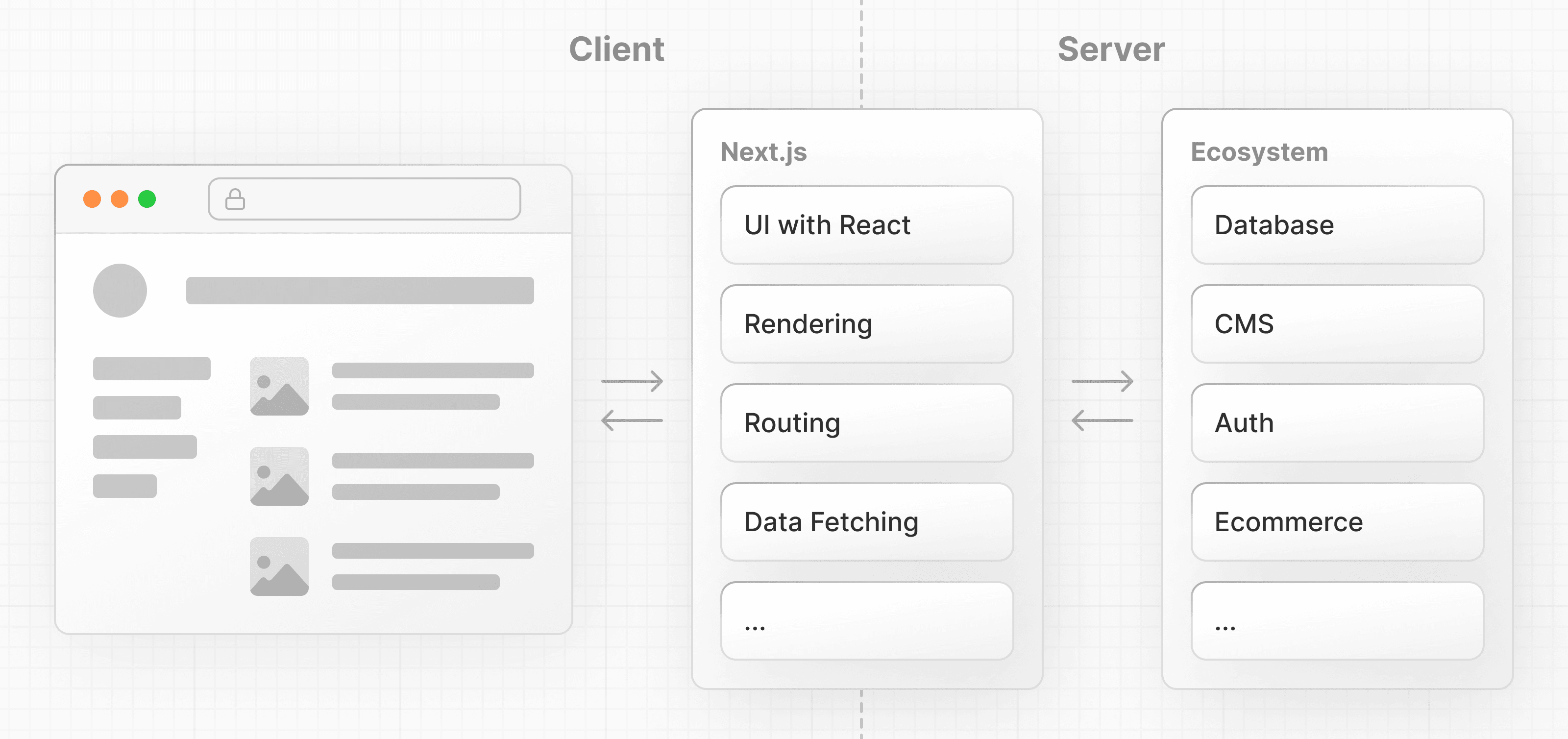
Part of React's success is that it is relatively unopinionated about the other aspects of building applications. This has resulted in a flourishing ecosystem of third-party tools and solutions, including Next.js.

It also means, however, that building a complete React application from the ground up requires some effort. Developers need to spend time configuring tools and reinventing solutions for common application requirements.

[**What is Next.js?**](https://nextjs.org/learn/react-foundations/what-is-react-and-nextjs#what-is-nextjs)

Next.js is a React **framework** that gives you building blocks to create web applications.

By framework, we mean Next.js handles the tooling and configuration needed for React, and provides additional structure, features, and optimizations for your application.



You can use React to build your UI, then incrementally adopt Next.js features to solve common application requirements such as routing, data fetching, and caching - all while improving the developer and end-user experience.

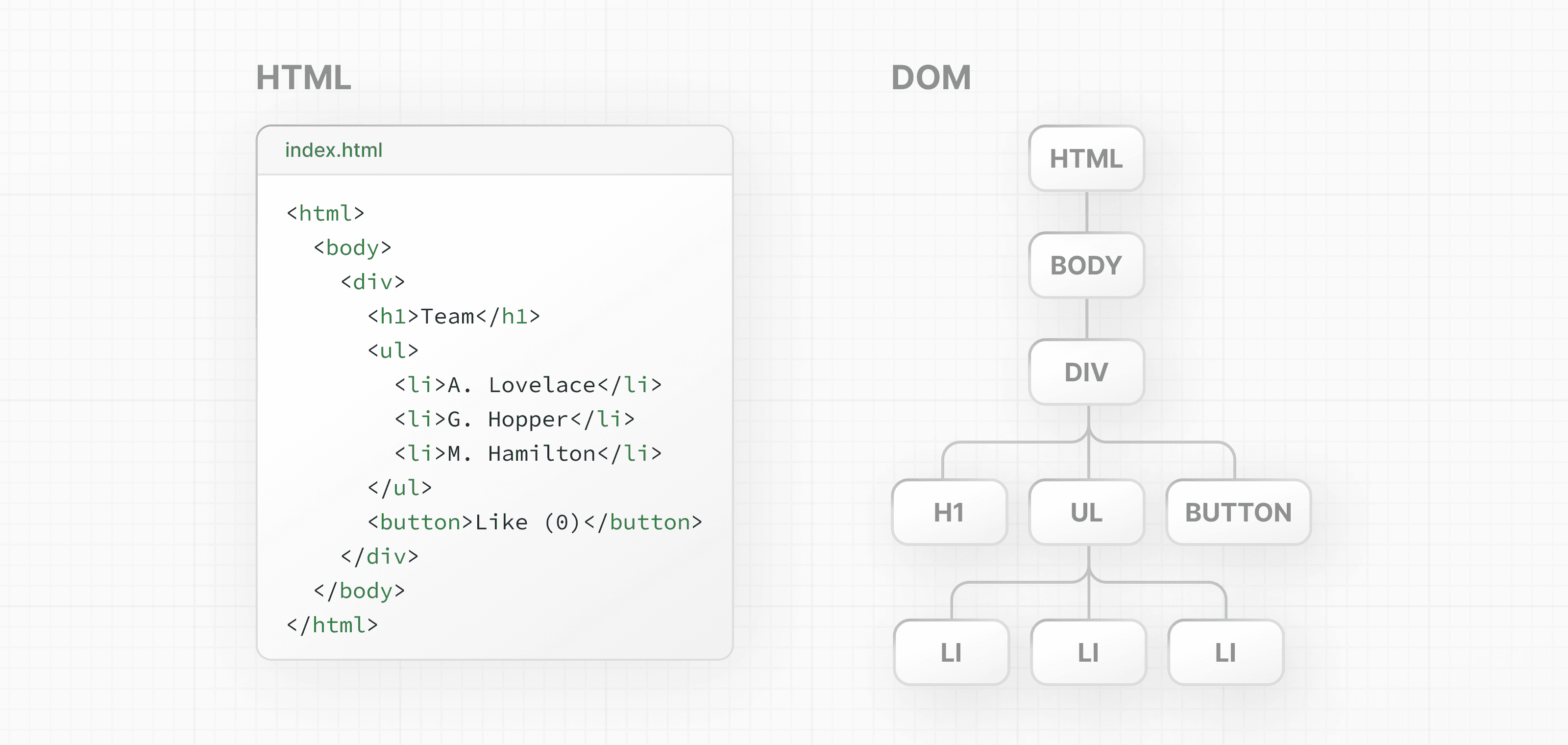
Whether you're an individual developer or part of a larger team, you can use React and Next.js to build fully interactive, highly dynamic, and performant web applications.

In the next chapters, we will discuss how you can get started with React and Next.js.

**Chapter 2**

**Rendering User Interfaces (UI)**

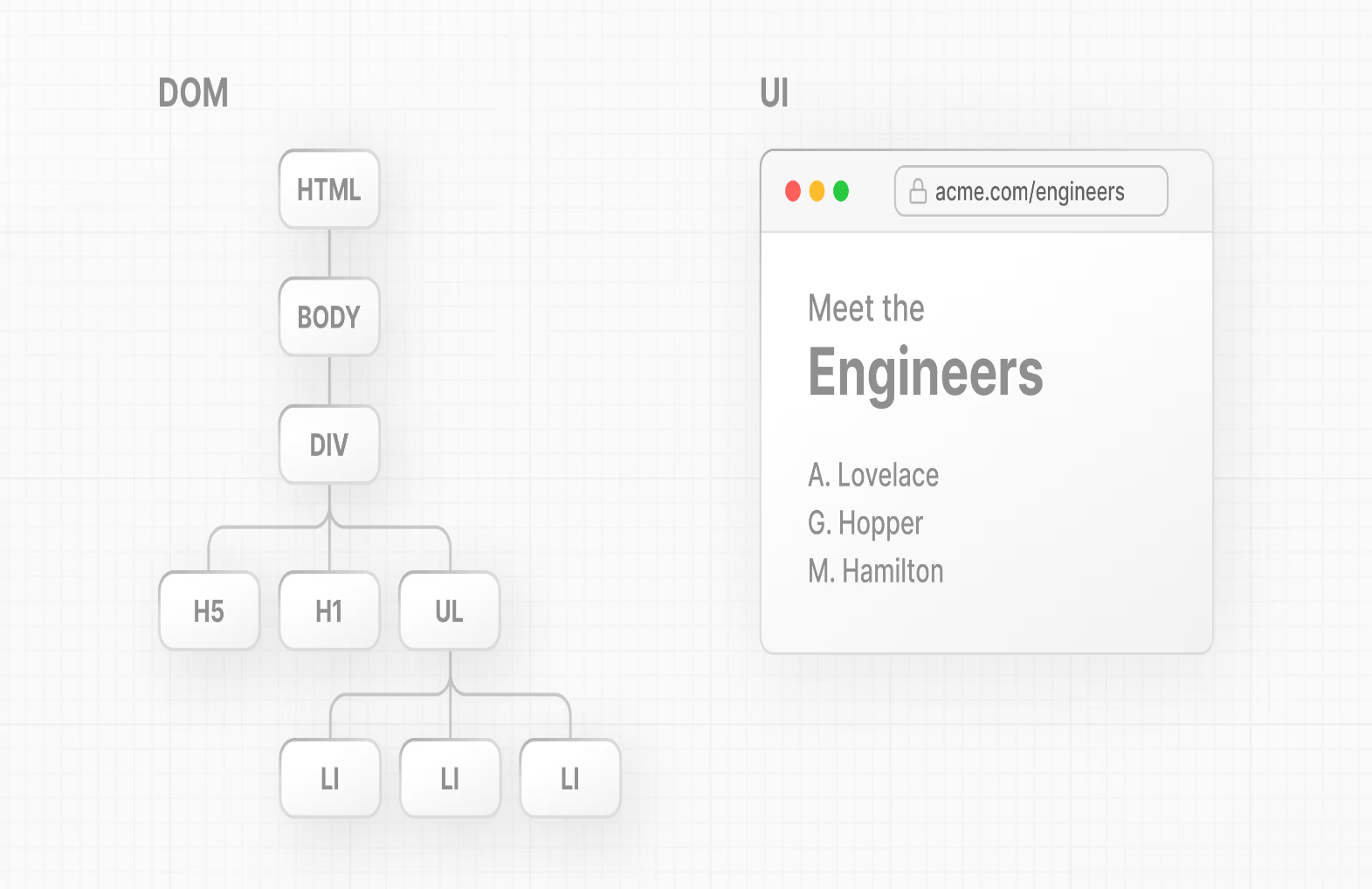
To understand how React works, we first need a basic understanding of how browsers interpret your code to create (or render) user interfaces (UI).

When a user visits a web page, the server returns an HTML file to the browser that may look like this: 

The browser then reads the HTML and constructs the Document Object Model (DOM).

[**What is the DOM?**](https://nextjs.org/learn/react-foundations/rendering-ui#what-is-the-dom)

The DOM is an object representation of the HTML elements. It acts as a bridge between your code and the user interface, and has a tree-like structure with parent and child relationships.



You can use DOM methods and JavaScript, to listen to user events and [manipulate the DOM](https://developer.mozilla.org/docs/Learn/JavaScript/Client-side_web_APIs/Manipulating_documents) by selecting, adding, updating, and deleting specific elements in the user interface. DOM manipulation allows you to not only target specific elements, but also change their style and content.

**Chapter 3**

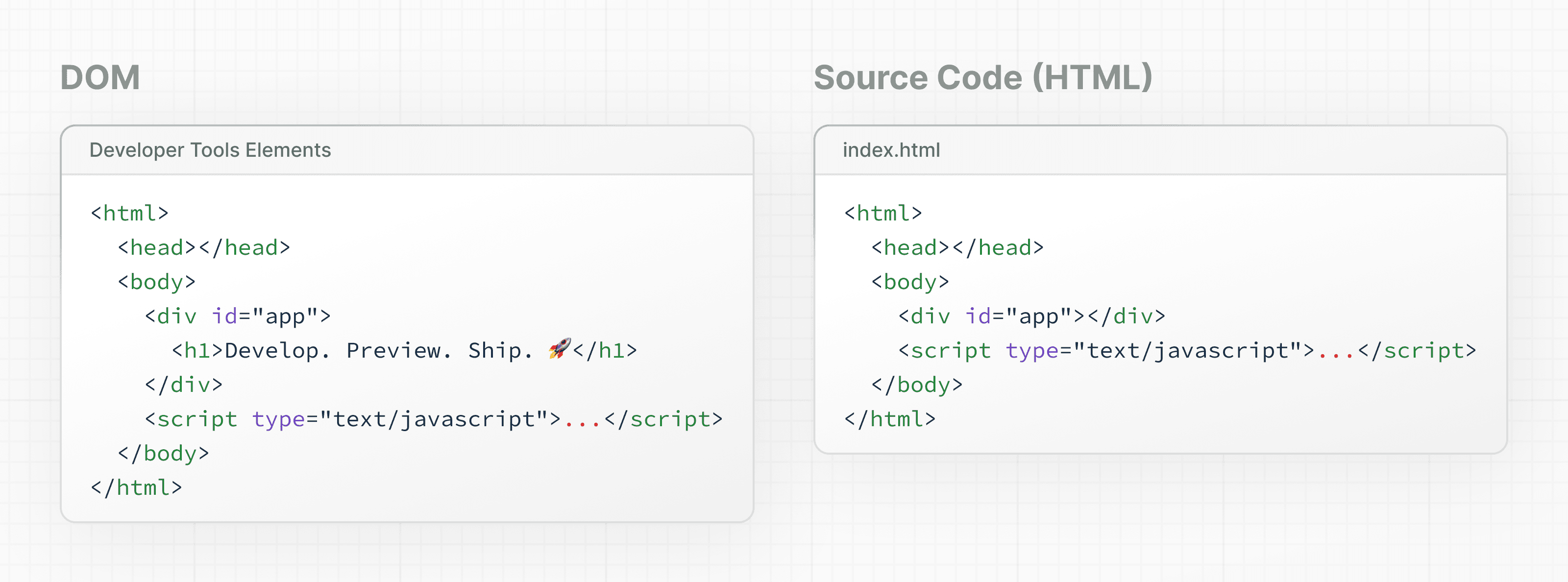
**Updating UI with JavaScript**

In this chapter, we'll start building out our project by using JavaScript and DOM methods to add an h1 tag to your project.  
A screenshot of a computer

AI-generated content may be incorrect.

[**HTML vs. the DOM**](https://nextjs.org/learn/react-foundations/updating-ui-with-javascript#html-vs-the-dom)

If you look at the DOM elements inside your [browser developer tools](https://developer.mozilla.org/docs/Learn/Common_questions/Tools_and_setup/What_are_browser_developer_tools), you will notice the DOM includes the <h1> element. The DOM of the page is different from the source code - or in other words, the original HTML file you created.



This is because the HTML represents the **initial page content**, whereas the DOM represents the **updated page content** which was changed by the JavaScript code you wrote.

[**Imperative vs. declarative programming**](https://nextjs.org/learn/react-foundations/updating-ui-with-javascript#imperative-vs-declarative-programming)

The code above is a good example of **imperative** **programming.** You're writing the steps for **how** the user interface should be updated. But when it comes to building user interfaces, a declarative approach is often preferred because it can speed up the development process. Instead of having to write DOM methods, it would be helpful if developers were able to declare **what** they want to show (in this case, an h1 tag with some text).

In other words, **imperative programming** is like giving a chef step-by-step instructions on how to make a pizza. **Declarative programming** is like ordering a pizza without being concerned about the steps it takes to make the pizza. 🍕

[React](https://react.dev/) is a popular declarative library that you can use build user interfaces.

[**React: A declarative UI library**](https://nextjs.org/learn/react-foundations/updating-ui-with-javascript#react-a-declarative-ui-library)

As a developer, you can tell React what you want to happen to the user interface, and React will figure out the steps of **how** to update the DOM on your behalf.

Chapter 4

**Getting Started with React**

To use React in your newly created project, load two React scripts from an external website called [unpkg.com](https://unpkg.com/):

* **react** is the core React library.
* **react-dom** provides DOM-specific methods that enable you to use React with the DOM.

## [What is JSX?](https://nextjs.org/learn/react-foundations/getting-started-with-react#what-is-jsx)

JSX is a syntax extension for JavaScript that allows you to describe your UI in a familiar HTML-like syntax. The nice thing about JSX is that apart from following [three JSX rules](https://react.dev/learn/writing-markup-with-jsx#the-rules-of-jsx), you don't need to learn any new symbols or syntax outside of HTML and JavaScript.

But browsers don't understand JSX out of the box, so you'll need a JavaScript compiler, such as a [Babel](https://babeljs.io/), to transform your JSX code into regular JavaScript.

You can start to see how using React enables you to cut down a lot of repetitive code.

And this is exactly what React does, it's a library that contains reusable snippets of code that perform tasks on your behalf - in this case, updating the UI.

[**Essential JavaScript for React**](https://nextjs.org/learn/react-foundations/getting-started-with-react#essential-javascript-for-react)

While you can learn JavaScript and React at the same time, being familiar with JavaScript can make the process of learning React easier.

In the next sections, you will be introduced to some core concepts of React from a JavaScript perspective. Here's a summary of the JavaScript topics that will be mentioned:

* [Functions](https://developer.mozilla.org/docs/Web/JavaScript/Guide/Functions) and [Arrow Functions](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Functions/Arrow_functions)
* [Objects](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Object)
* [Arrays and array methods](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Array)
* [Destructuring](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment)
* [Template literals](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Template_literals)
* [Ternary Operators](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Operators/Conditional_Operator)
* [ES Modules and Import / Export Syntax](https://developer.mozilla.org/docs/Web/JavaScript/Guide/Modules)

While this course does not dive into JavaScript, it's good practice to stay up to date with the latest versions of JavaScript. But if you don't feel proficient in JavaScript yet, don't let this hinder you from starting to build with React!

**Chapter 5**

**Building UI with Components**

[**React core concepts**](https://nextjs.org/learn/react-foundations/building-ui-with-components#react-core-concepts)

There are three core concepts of React that you'll need to be familiar with to start building React applications. These are:

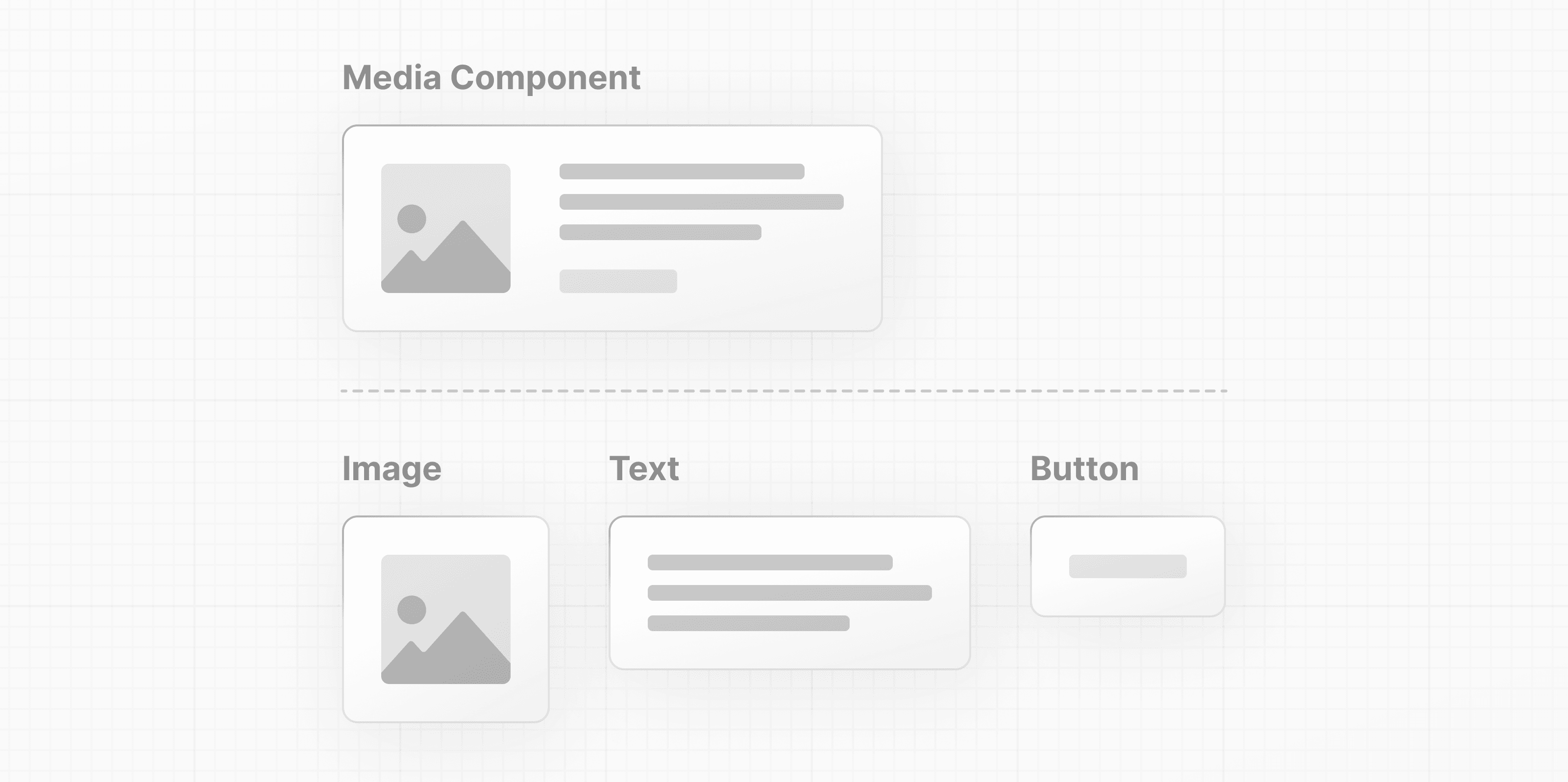
* Components
* Props
* State

In the next chapters, we will go through these concepts and provide resources where you can continue learning them. After you're familiar with these concepts, we'll then show you how to install Next.js and use newer React features such as Server and Client Components.

[**Components**](https://nextjs.org/learn/react-foundations/building-ui-with-components#components)

User interfaces can be broken down into smaller building blocks called **components**.

Components allow you to build self-contained, reusable snippets of code. If you think of components as **LEGO bricks**, you can take these individual bricks and combine them together to form larger structures. If you need to update a piece of the UI, you can update the specific component or brick.



This modularity allows your code to be more maintainable as it grows because you can add, update, and delete components without touching the rest of our application.

The nice thing about React components is that they are just JavaScript. Let's see how you can write a React component, from a JavaScript perspective:

[**Creating components**](https://nextjs.org/learn/react-foundations/building-ui-with-components#creating-components)

In React, components are **functions.** Inside your script tag, create a new function called header:

<script type="text/jsx">

const app = document.getElementById("app")

function header() {

}

const root = ReactDOM.createRoot(app);

root.render(<h1>Develop. Preview. Ship.</h1>);

</script>

A component is a function that **returns UI elements**. Inside the return statement of the function, you can write JSX:

<script type="text/jsx">

const app = document.getElementById("app")

function header() {

return (<h1>Develop. Preview. Ship.</h1>)

}

const root = ReactDOM.createRoot(app);

root.render(<h1>Develop. Preview. Ship.</h1>);

</script>

To render this component to the DOM, pass it as the first argument in the root.render() method:

<script type="text/jsx">

const app = document.getElementById("app")

function header() {

return (<h1>Develop. Preview. Ship.</h1>)

}

const root = ReactDOM.createRoot(app);

root.render(header);

</script>

But, wait a second. If you try to run the code above in your browser, you'll get an error. To get this to work, there are two things you have to do:

First, React components should be capitalized to distinguish them from plain HTML and JavaScript:

function Header() {

return <h1>Develop. Preview. Ship.</h1>;

}

const root = ReactDOM.createRoot(app);

// Capitalize the React Component

root.render(Header);

Second, you use React components the same way you'd use regular HTML tags, with angle brackets <>:

function Header() {

return <h1>Develop. Preview. Ship.</h1>;

}

const root = ReactDOM.createRoot(app);

root.render(<Header />);

function Header() {

return <h1>Develop. Preview. Ship.</h1>;

}

const root = ReactDOM.createRoot(app);

root.render(<Header />);

If you try to run the code in your browser again, you'll see your changes.

[**Nesting components**](https://nextjs.org/learn/react-foundations/building-ui-with-components#nesting-components)

Applications usually include more content than a single component. You can **nest** React components inside each other like you would regular HTML elements.

In your example, create a new component called HomePage:

index.html

function Header() { return <h1>Develop. Preview. Ship.</h1>;} function HomePage() { return <div></div>;} const root = ReactDOM.createRoot(app);root.render(<Header />);

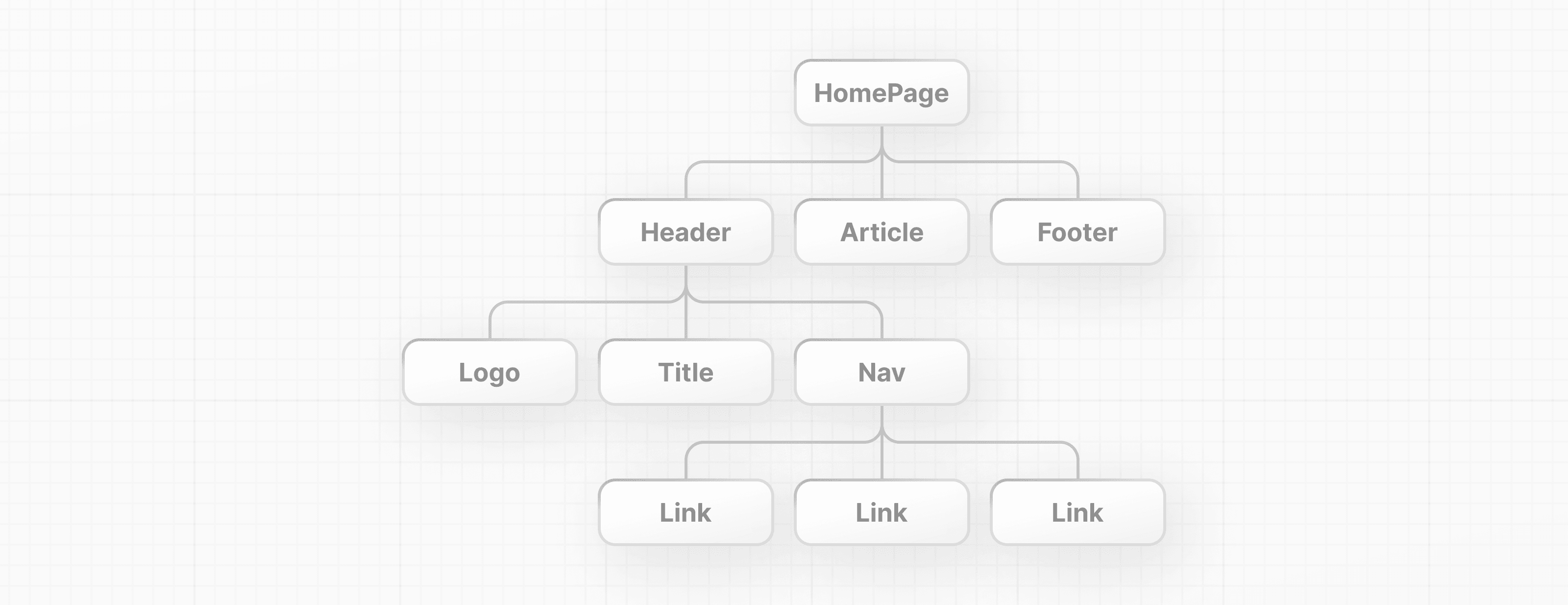
Then nest the <Header> component inside the new <HomePage>component:

index.html

function Header() { return <h1>Develop. Preview. Ship.</h1>;} function HomePage() { return ( <div> {/\* Nesting the Header component \*/} <Header /> </div> );} const root = ReactDOM.createRoot(app);root.render(<Header />);

[**Component trees**](https://nextjs.org/learn/react-foundations/building-ui-with-components#component-trees)

You can keep nesting React components this way to form component trees.



For example, your top-level HomePage component could hold a Header, an Article, and a Footer Component. And each of those components could in turn have their own child components and so on. For example, the Header component could contain a Logo, Title and Navigation component.

This modular format allows you to reuse components in different places inside your app.

In your project, since <HomePage> is now your top-level component, you can pass it to the root.render() method:

index.html

function Header() { return <h1>Develop. Preview. Ship.</h1>;} function HomePage() { return ( <div> <Header /> </div> );} const root = ReactDOM.createRoot(app);root.render(<HomePage />);

“Script”:  
<!DOCTYPE html>

<html>

<head>

  <title>React App</title>

  <!-- Add Babel for JSX support -->

  <script src="https://unpkg.com/@babel/standalone/babel.min.js"></script>

  <!-- Add React and ReactDOM -->

  <script src="https://unpkg.com/react@18/umd/react.development.js"></script>

  <script src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"></script>

</head>

<body>

  <div id="app"></div>

  <!-- Your React script here -->

  <script type="text/babel">

    const app = document.getElementById("app");

    function Header() {

      return <h1>Develop. Preview. Ship.</h1>;

    }

    function HomePage() {

      return (

        <div>

          <Header />

        </div>

      );

    }

    const root = ReactDOM.createRoot(app);

    root.render(<HomePage />);

  </script>

</body>

</html>

**Explanation:**

This is a basic HTML page that runs a small React app right inside the browser.

Here’s what’s happening, step by step:

1. HTML Setup  
   At the top, we set up a basic HTML page with a title “React App.”
2. Scripts for React  
   We include three important libraries from the internet:
   * Babel → Helps the browser understand JSX (which looks like HTML inside JavaScript).
   * React → The main library for building UI components.
   * ReactDOM → Helps React talk to the actual web page.
3. App Container  
   There’s a <div> with the ID app — this is where our React app will appear on the page.
4. React Code Section  
   Inside a script tag (type="text/babel"), we write our React code:
   * We grab the <div> from the page using document.getElementById("app").
   * We create a Header component — it’s just a small piece of the UI that returns a headline (<h1>Develop. Preview. Ship.</h1>).
   * We create a HomePage component that uses the Header component inside a <div>.
   * Finally, we tell React to render the HomePage inside the app div by creating a React root and calling root.render(<HomePage />).

In short, this code sets up a small React app that displays a headline on the web page.  
  
**Chapter 6**

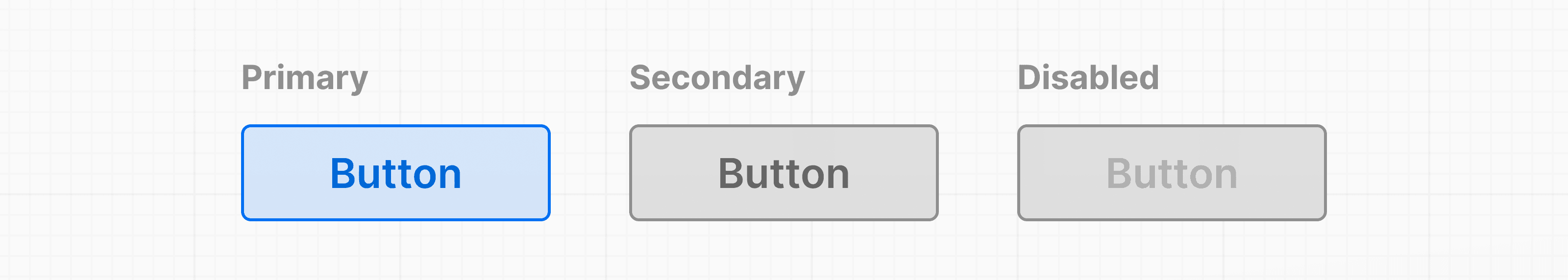
**Displaying Data with Props**

So far, if you were to reuse your <Header /> component, it would display the same content both times.

But what if you want to pass different text or you don't know the information ahead of time because you're fetching data from an external source?

Regular HTML elements have attributes that you can use to pass pieces of information that change the behavior of those elements. For example, changing the src attribute of an <img> element changes the image that is shown. Changing the href attribute of an <a> tag changes the destination of the link.

In the same way, you can pass pieces of information as properties to React components. These are called props. Take for instance, the possible variations of a button:



Similar to a JavaScript function, you can design components that accept custom arguments (or props) that change the component's behavior or what is visibly shown when it's rendered to the screen. Then, you can pass down these props from parent components to child components.

**Note:** In React, data flows down the component tree. This is referred to as *one-way data flow*. State, which will be discussed in the next chapter, can be passed from parent to child components as props.

[**Using props**](https://nextjs.org/learn/react-foundations/displaying-data-with-props#using-props)

In your HomePage component, you can pass a custom title prop to the Header component, just like you'd pass HTML attributes:  
  
  
index.html

function HomePage() { return ( <div> <Header title="React" /> </div> );}

And Header, the child component, can accept those props as its first **function parameter**:

index.html

function Header(props) { return <h1>Develop. Preview. Ship.</h1>;}

If you console.log() props, you can see that it's an **object** with a title property.

index.html

function Header(props) { console.log(props); // { title: "React" } return <h1>Develop. Preview. Ship.</h1>;}

Since props is an object, you can use [**object destructuring**](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment) to explicitly name the values of props inside your function parameters:

index.html

function Header({ title }) { console.log(title); // "React" return <h1>Develop. Preview. Ship.</h1>;}

Then you can replace the content of the <h1> tag with your title variable.

index.html

function Header({ title }) { console.log(title); return <h1>title</h1>;}

If you open your file in the browser, you will see that it is displaying the actual word "title". This is because React thinks you're intending to render a plain text string to the DOM.

You need a way to tell React that this is a JavaScript variable.

[**Using variables in JSX**](https://nextjs.org/learn/react-foundations/displaying-data-with-props#using-variables-in-jsx)

To use the title prop, add **curly braces** {}. These are a special JSX syntax that allows you to write regular JavaScript directly inside your JSX markup.

index.html

function Header({ title }) { console.log(title); return <h1>{title}</h1>;}

You can think of curly braces as a way to enter "JavaScript land" while you are in "JSX land". You can add any **JavaScript expression** (something that evaluates to a single value) inside curly braces. For example:

1. An **object property** with dot notation:

example.js

function Header(props) { return <h1>{props.title}</h1>;}

1. A **template literal**:

example.js

function Header({ title }) { return <h1>{`Cool ${title}`}</h1>;}

1. The **returned value of a function**:

example.js

function createTitle(title) { if (title) { return title; } else { return 'Default title'; }} function Header({ title }) { return <h1>{createTitle(title)}</h1>;}

1. Or **ternary operators**:

example.js

function Header({ title }) { return <h1>{title ? title : 'Default Title'}</h1>;}

You can now pass any string to your title prop, or, if you used the ternary operator, you could even not pass a title prop at all, since you've accounted for the default case in your component:

example.js

function Header({ title }) { return <h1>{title ? title : 'Default title'}</h1>;} function HomePage() { return ( <div> <Header /> </div> );}

Your component now accepts a generic title prop which you can reuse in different parts of your application. All you need to do is change the title string:

index.html

function HomePage() { return ( <div> <Header title="React" /> <Header title="A new title" /> </div> );}

[**Iterating through lists**](https://nextjs.org/learn/react-foundations/displaying-data-with-props#iterating-through-lists)

It's common to have data that you need to show as a list. You can use array methods to manipulate your data and generate UI elements that are identical in style but hold different pieces of information.

Add the following array of names to your HomePage component:

index.html

function HomePage() { const names = ['Ada Lovelace', 'Grace Hopper', 'Margaret Hamilton']; return ( <div> <Header title="Develop. Preview. Ship." /> <ul> {names.map((name) => ( <li>{name}</li> ))} </ul> </div> );}

You can then use the array.map() method to iterate over the array and use an **arrow function** to map a name to a list item:

index.html

function HomePage() { const names = ['Ada Lovelace', 'Grace Hopper', 'Margaret Hamilton']; return ( <div> <Header title="Develop. Preview. Ship." /> <ul> {names.map((name) => ( <li>{name}</li> ))} </ul> </div> );}

Notice how you've used curly braces to weave in and out of "JavaScript" and "JSX" land.

If you run this code, React will give us a warning about a missing key prop. This is because React needs something to uniquely identify items in an array so it knows which elements to update in the DOM.

You can use the names for now since they are currently unique, but it's recommended to use something guaranteed to be unique, like an item ID.

index.html

function HomePage() { const names = ['Ada Lovelace', 'Grace Hopper', 'Margaret Hamilton']; return ( <div> <Header title="Develop. Preview. Ship." /> <ul> {names.map((name) => ( <li key={name}>{name}</li> ))} </ul> </div> );}

**Additional Resources:**

* [Passing props to a component](https://react.dev/learn/passing-props-to-a-component)
* [Rendering lists](https://react.dev/learn/rendering-lists)
* [Conditional rendering](https://react.dev/learn/conditional-rendering)

**Chapter 7**

**Adding Interactivity with State**

Let's explore how React helps us add interactivity with state and event handlers.

As an example, let's create a "Like" button inside your HomePage component. First, add a button element inside the return() statement:

index.html

function HomePage() { const names = ['Ada Lovelace', 'Grace Hopper', 'Margaret Hamilton']; return ( <div> <Header title="Develop. Preview. Ship." /> <ul> {names.map((name) => ( <li key={name}>{name}</li> ))} </ul> <button>Like</button> </div> );}

[Listening to events](https://nextjs.org/learn/react-foundations/updating-state#listening-to-events)

To make the button do something when clicked, you can use the onClick event:

index.html

function HomePage() { // ... return ( <div> {/\* ... \*/} <button onClick={}>Like</button> </div> );}

In React, event names are camelCased. The onClick event is one of many possible events you can use to respond to user interaction. For example, you can use onChange for input fields or onSubmit for forms.

[Handling events](https://nextjs.org/learn/react-foundations/updating-state#handling-events)

You can define a function to "handle" events whenever they are triggered. Create a function before the return statement called handleClick():

index.html

function HomePage() { // ... function handleClick() { console.log("increment like count") } return ( <div> {/\* ... \*/} <button onClick={}>Like</button> </div> ) }

Then, you can call the handleClick function when the onClick event is triggered:

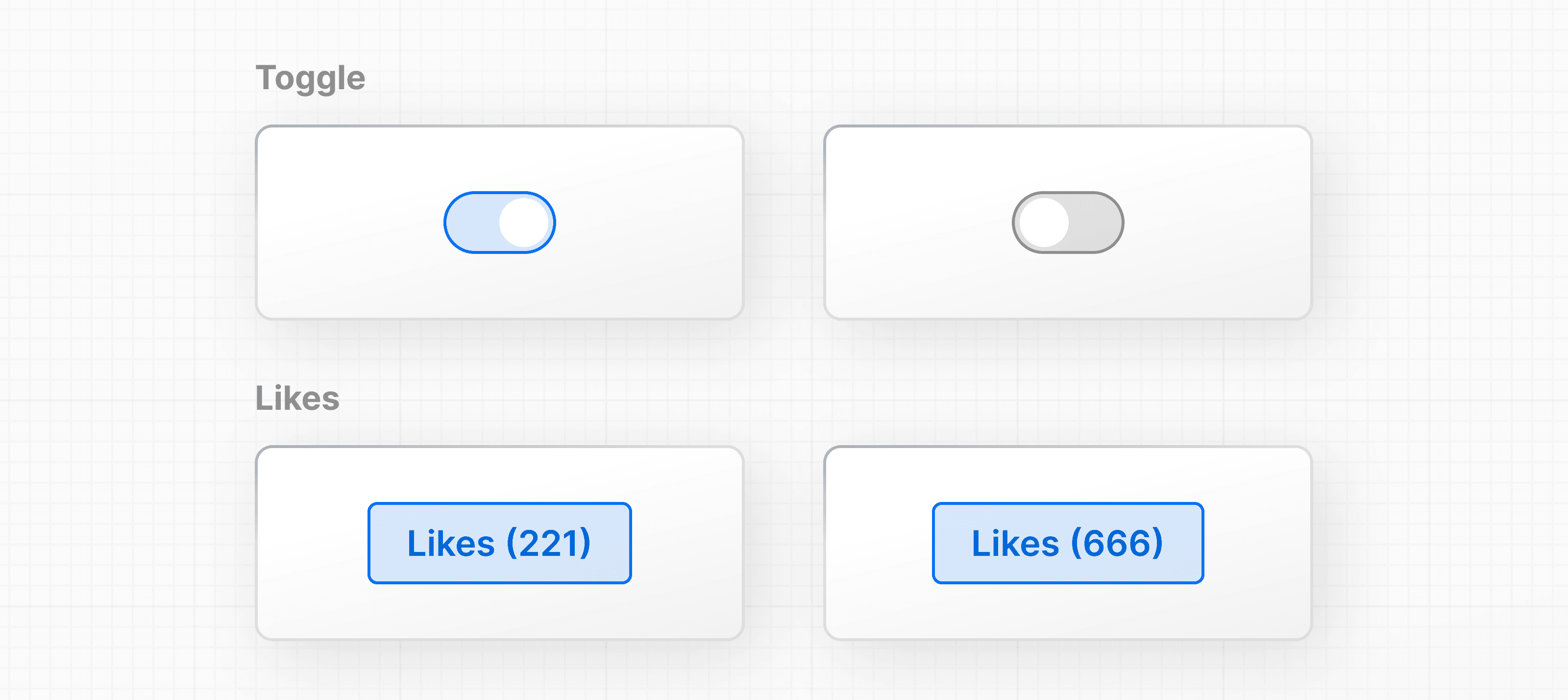
index.html

function HomePage() { // ... function handleClick() { console.log('increment like count'); } return ( <div> {/\* ... \*/} <button onClick={handleClick}>Like</button> </div> );}

Try running this in your browser. Notice in your developer tools how the log output increases.

[State and hooks](https://nextjs.org/learn/react-foundations/updating-state#state-and-hooks)

React has a set of functions called [hooks](https://react.dev/learn). Hooks allow you to add additional logic such as state to your components. You can think of state as any information in your UI that changes over time, usually triggered by user interaction.



You can use *state* to store and increment the number of times a user has clicked the "Like" button. In fact, the React hook used to manage state is called: useState()

Add useState() to your project. It returns an array, and you can access and use those array values inside your component using **array destructuring**:

index.html

function HomePage() { // ... const [] = React.useState(); // ...}

The first item in the array is the state value, which you can name anything. It's recommended to name it something descriptive:

index.html

function HomePage() { // ... const [likes] = React.useState(); // ...}

The second item in the array is a function to update the value. You can name the update function anything, but it's common to prefix it with set followed by the name of the state variable you're updating:

index.html

function HomePage() { // ... const [likes, setLikes] = React.useState(); // ...}

You can also take the opportunity to add the initial value of your likes state to 0:

index.html

function HomePage() { // ... const [likes, setLikes] = React.useState(0);}

Then, you can check the initial state is working by using the state variable inside your component.

index.html

function HomePage() { // ... const [likes, setLikes] = React.useState(0); // ... return ( // ... <button onClick={handleClick}>Like({likes})</button> );}

Finally, you can call your state updater function, setLikes in your HomePage component, let's add it inside the handleClick() function you previously defined:

index.html

function HomePage() { // ... const [likes, setLikes] = React.useState(0); function handleClick() { setLikes(likes + 1); } return ( <div> {/\* ... \*/} <button onClick={handleClick}>Likes ({likes})</button> </div> );}

Clicking the button will now call the handleClick function, which calls the setLikes state updater function with a single argument of the current number of likes + 1.

**Note**: Unlike props which are passed to components as the first function parameter, the state is initiated and stored within a component. You can pass the state information to children components as props, but the logic for updating the state should be kept within the component where state was initially created.

[**Managing state**](https://nextjs.org/learn/react-foundations/updating-state#managing-state)

This was only an introduction to state, and there's more you can learn about managing state and data flow in your React applications. To learn more, we recommend you go through the [Adding Interactivity](https://react.dev/learn/adding-interactivity) and [Managing State](https://react.dev/learn/managing-state) sections in the React documentation.

**Additional Resources:**

* [State: A component's memory](https://react.dev/learn/state-a-components-memory)
* [Meet your first hook](https://react.dev/learn/state-a-components-memory#meet-your-first-hook)
* [Responding to Events](https://react.dev/learn/responding-to-events)