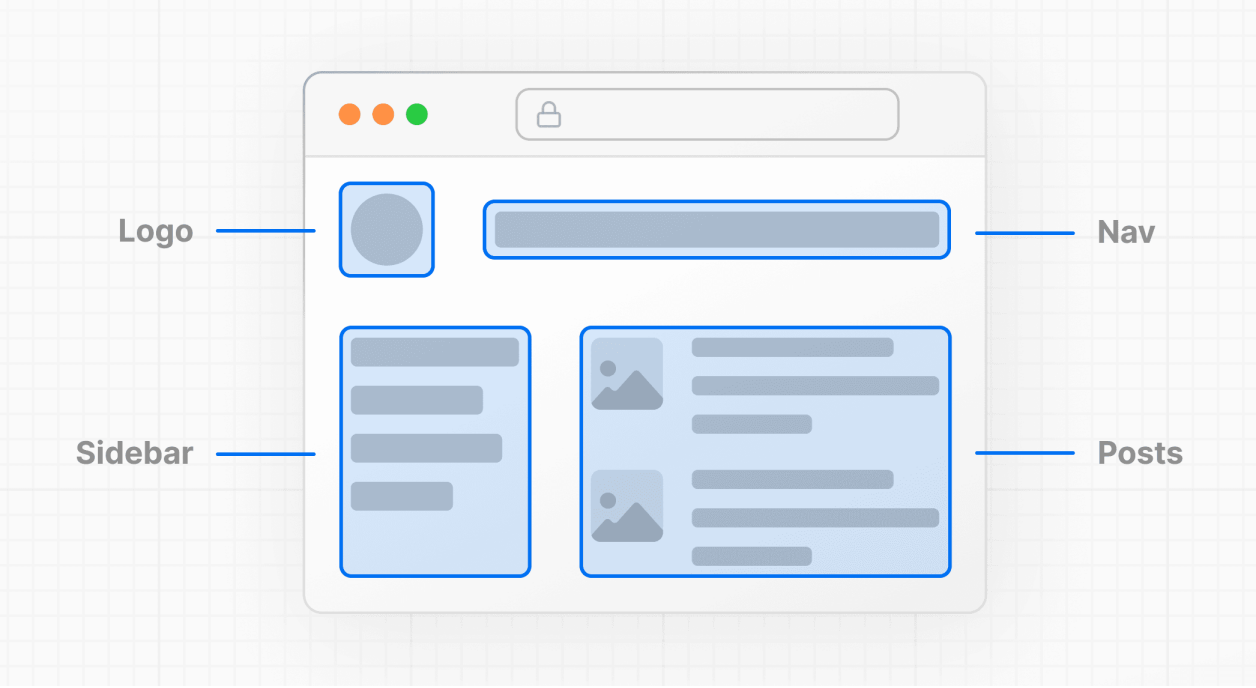
React and Next.js

NTOU CSE
2025

https://nextjs.org/learn/react-foundations/what-is-react-and-nextjs

What is React?[1]

- React 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.

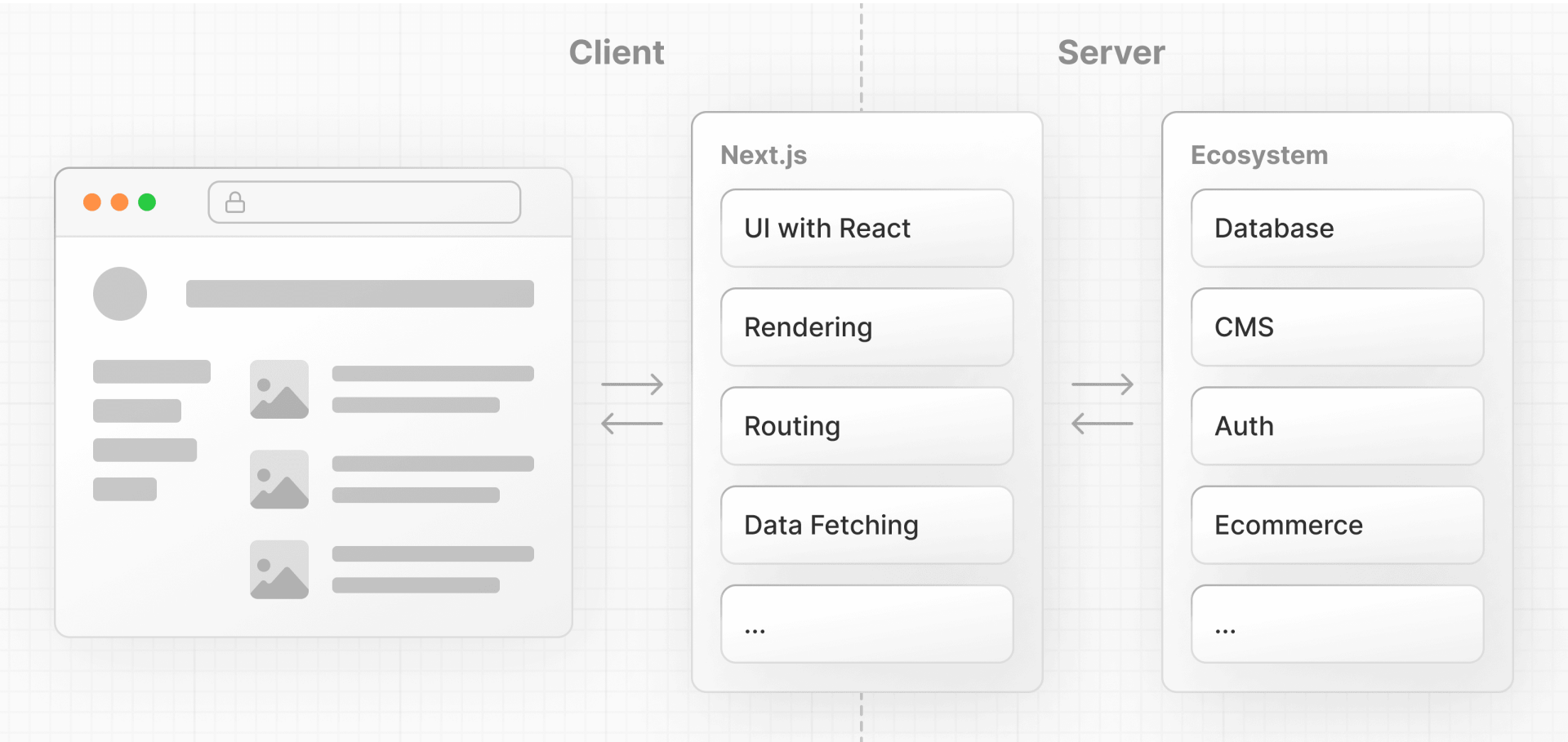


What is React?2

- 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.
- 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?

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



Imperative vs. Declarative Programming

- Traditional DOM-based code 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,
- **Declarative programming** 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).

Getting Started with React1

- To use React in your newly created project, load two React scripts from an external website called unpkg.com:
  - react is the core React library.
  - react-dom provides DOM-specific methods that enable you to use React with the DOM.

Getting Started with React2

- Instead of directly manipulating the DOM with plain JavaScript:
  - Remove the DOM methods that you had added previously,
  - Add the ReactDOM.createRoot() method to target a specific DOM element and create a root to display your React Components in.
  - Add the root.render() method to render your React code to the DOM.

Getting Started with React3


```
<div id="app"></div>
<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>
<script>
    const app = document.getElementById('app');
    const root = ReactDOM.createRoot(app);
    root.render(<h1>Develop. Preview. Ship.</h1>);
</script>
```

If you try to run this code in the browser, you will get a syntax error:

Uncaught SyntaxError: expected expression, got '<'

This is because <h1>...</h1> is not valid Javascript. This piece of code is JSX.

What is JSX?

- JSX is a syntax extension for JavaScript that allows you to describe your UI in a familiar HTML-like syntax.
  - Browsers don't understand JSX out of the box, so you'll need a JavaScript compiler, such as a Babel, to transform your JSX code into regular JavaScript.
- To add Babel to your project, copy and paste the following script in your index.html file:

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

Check c4-03-react-babel.html

Imperative vs. Declarative Programming

Imperative:

```
<script type="text/javascript">
  const app = document.getElementById('app');
  const header = document.createElement('h1');
  const text = 'Develop. Preview. Ship.';
  const headerContent = document.createTextNode(text);
  header.appendChild(headerContent);
  app.appendChild(header);
</script>
```

Declarative:

```
<script type="text/jsx">
  const domNode = document.getElementById('app');
  const root = ReactDOM.createRoot(domNode);
  root.render(<h1>Develop. Preview. Ship.</h1>);
</script>
```

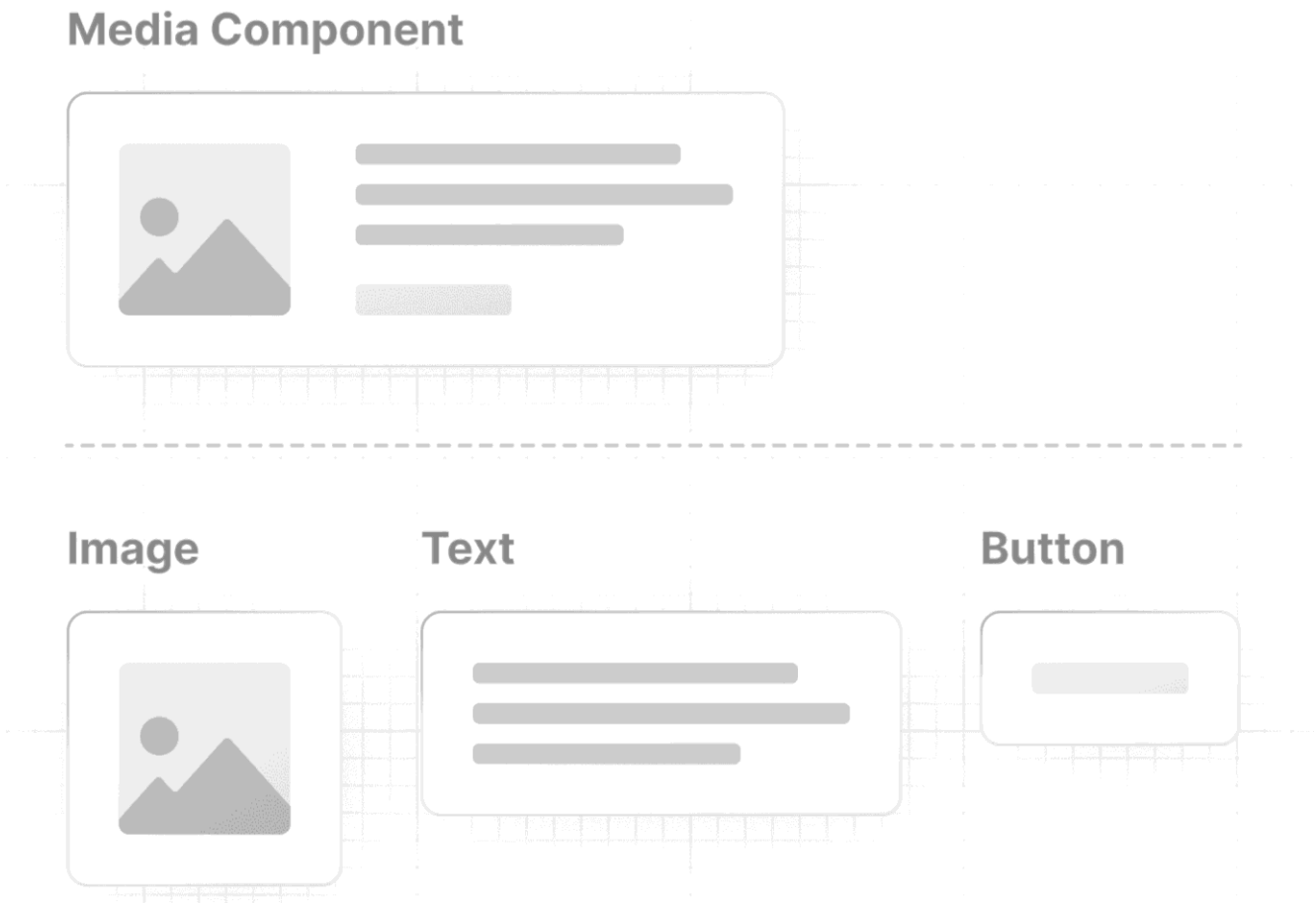
Building UI with Components

- There are three core concepts of React that you'll need to be familiar with to start building React applications.
  - Components
  - Props
  - State

Components1

- 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.

Components2



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.

Components3

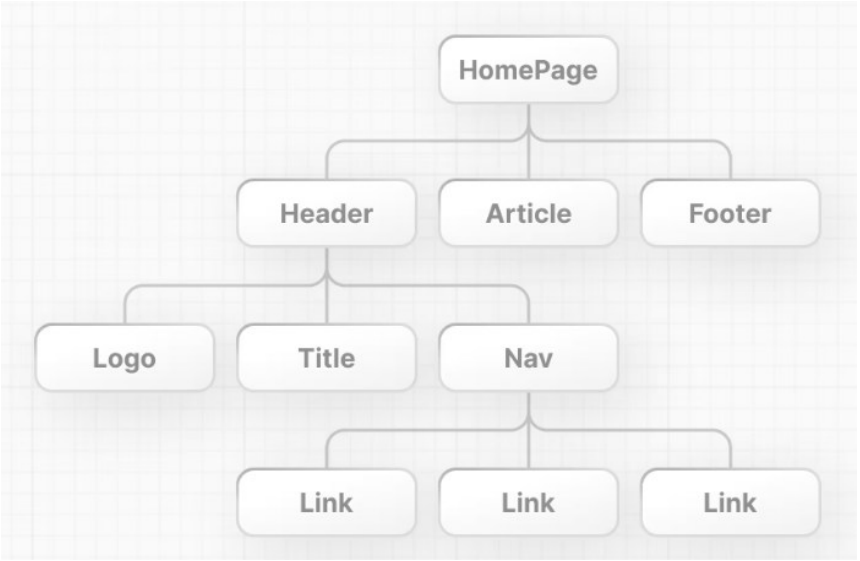
- In React, components are **functions**.
  - A component is **a function that returns UI elements.** Inside the return statement of the function.
- Two component rules:
  - React components should be capitalized to distinguish them from plain HTML and JavaScript.
  - Use React components the same way you'd use regular HTML tags, with angle brackets <>

Components4


```
<script type="text/jsx">
    const domNode = document.getElementById('app');

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

    const root = ReactDOM.createRoot(app);
    root.render(<Header />);
</script>
```

Nesting Components[1]

- Applications usually include more content than a single component.
  - You can nest React components inside each other like you would regular HTML elements.
- You can keep nesting React components this way to form component trees.

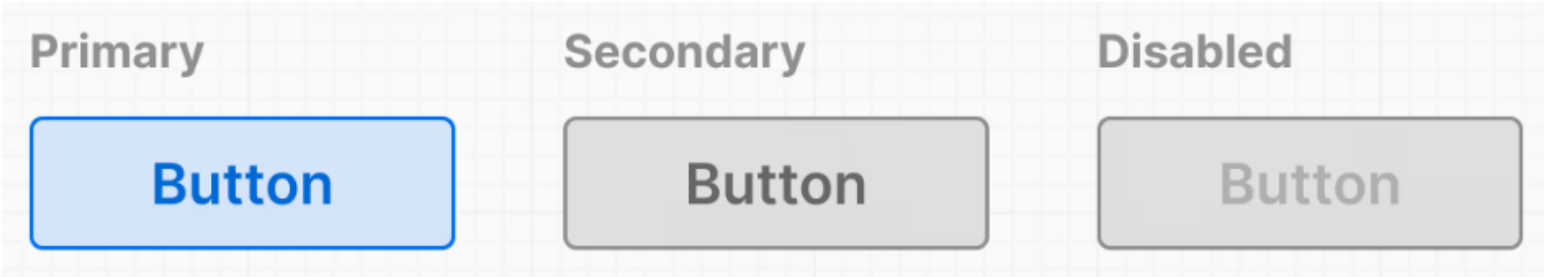


Nesting Components2


```
<script type="text/jsx">
    function Header() {
        return <h1>Develop. Preview. Ship.</h1>;
    }

    function HomePage() {
        return (
            <div>
                {/* Nesting the Header component */}
                <Header />
                <hr/>
            </div>
        );
    }

    const root = ReactDOM.createRoot(app);
    root.render(<HomePage />);
</script>
```

Displaying Data with Props1

- If you were to reuse component, it would display the same content both times.
- 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.



Displaying Data with Props2

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

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

Displaying Data with Props3

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

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

- Using variables in JSX
  - **Curly braces {}** are a special JSX syntax that allows you to write regular JavaScript directly inside your JSX markup.

Displaying Data with Props4

- You can add any JavaScript expression (something that evaluates to a single value) inside curly braces.
  - An object property with dot notation:

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

  - A template literal:

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

Displaying Data with Props 5

- The returned value of a function:

```
function createTitle(title) {
  if (title) {
    return title;
  } else {
    return 'Default title';
  }
}

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

- Ternary operators:

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

Destructuring

- The destructuring syntax is a JavaScript syntax that makes it possible to unpack values from arrays, or properties from objects, into distinct variables.
  - It can be used in locations that receive data (such as the left-hand side of an assignment or anywhere that creates new identifier bindings).

```
let a, b, rest;
[a, b] = [10, 20];

console.log(a);
// Expected output: 10

console.log(b);
// Expected output: 20

[a, b, ...rest] = [10, 20, 30, 40, 50];

console.log(rest);
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring

Iterating Through Lists[1]

- It's common to have data to show as a list.
  - Use the array.map() method to iterate over the array and use an arrow function to map a name to a list item

```
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>
    );
}
```

Iterating Through Lists 2

- **key** prop: React needs something to uniquely identify items in an array so it knows which elements to update in the DOM.

```
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>
    );
}
```

Handling Events

- You can define a function to "handle" events whenever they are triggered.
  - For example, you can create a function before the return statement called handleClick().
  - Then, you can call the handleClick function when the onClick event is triggered:

```
function handleClick() {
  console.log("increment like count")
}

return (
  <div>
    {/* ... */}
    <button onClick={handleClick}>Like</button>
  </div>
);
```

State and Hooks[1]

- React has a set of functions called **hooks**.
- 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**.

State and Hooks2

- The React hook used to manage state is called: **useState()**.
  - It returns an array, and you can access and use those array values inside your component using **array destructuring**.

```
const [likes, setLikes] = React.useState(0);

function handleClick() {
  setLikes(likes + 1);
}
```

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.

From React to Next.js

- While React excels at building UI, it does take some work to independently build that UI into a fully functioning scalable application.
- There are also newer React features, like Server and Client Components, that require a framework.
  - **Next.js** handles much of the setup and configuration and has additional features to help you build React applications.

Installing Next.js1

- When you use Next.js in your project, you do not need to load the react and react-dom scripts from unpkg.com anymore.
  - Instead, you can install these packages locally using npm or your preferred package manager.
- In your terminal, run the following command in the root of your project:

  npm install react@latest react-dom@latest next@latest

- Once the installation is complete, you should be able to see your project dependencies listed inside your package.json file.

Installing Next.js2

- Delete the following code for the HTML:
  - The <html> and <body> tags.
  - The element with the id of app.
  - The react and react-dom scripts since you've installed them with NPM.
  - The Babel script because Next.js has a compiler that transforms JSX into valid JavaScript browsers can understand.
  - The <script type="text/jsx"> tag.
  - The document.getElementById() and ReactDom.createRoot() methods.
  - The React. part of the React.useState(0) function

Installing Next.js3

- Create a new folder called app and move the index.js file inside it.
- Rename your HTML file to page.js.
  - This will be the main page of your application.
- Add export default to your <HomePage> component to help Next.js distinguish which component to render as the main component of the page.

Installing Next.js4

- Add a "next dev" script to your package.json file:

```
{
  "scripts": {
    "dev": "next dev"
  },
  "dependencies": {
    "next": "^16.0.10",
    "react": "^19.2.3",
    "react-dom": "^19.2.3"
  }
}
```

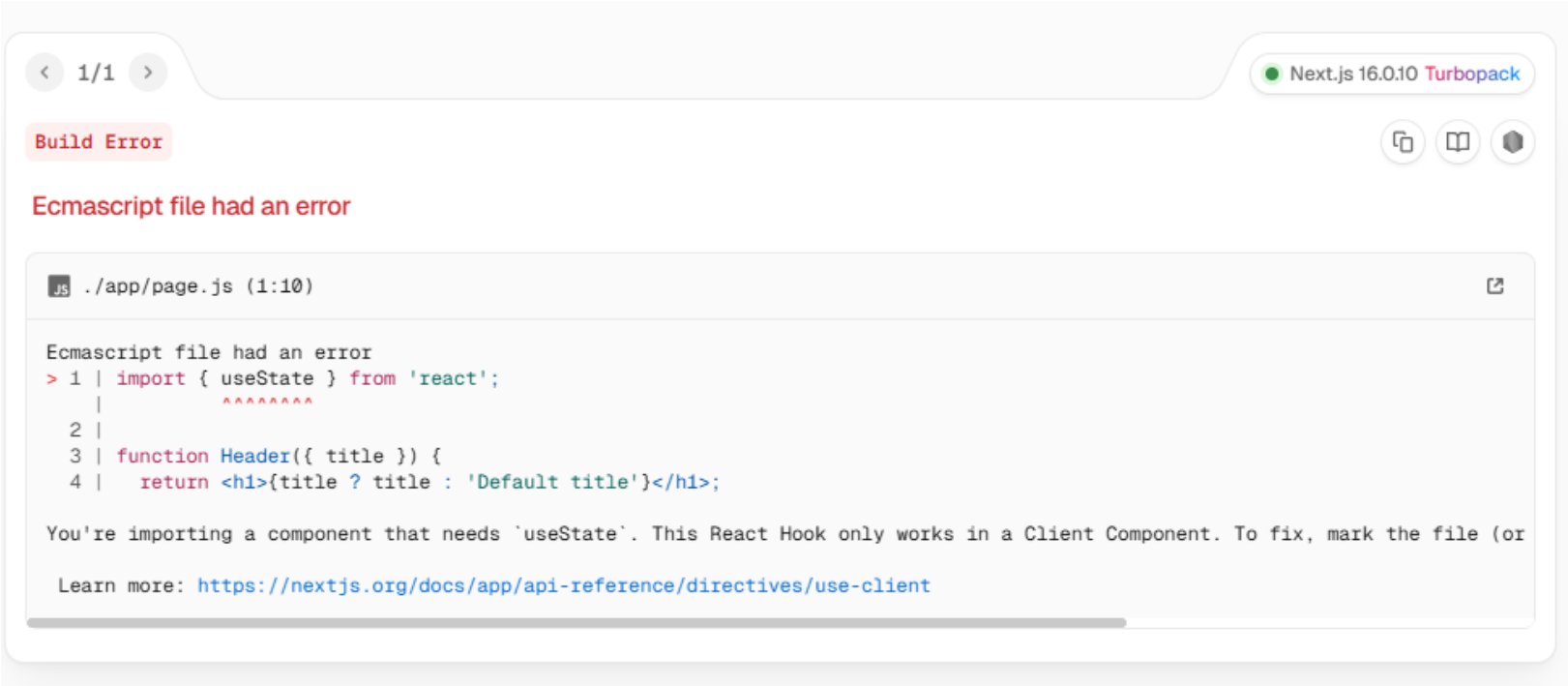
Installing Next.js5

- Add “layout.js” in the app folder:

```
export const metadata = {
    title: 'Next.js',
    description: 'Generated by Next.js',
};

export default function RootLayout({ children }) {
    return (
        <html lang="en">
            <body>{children}</body>
        </html>
    );
}
```

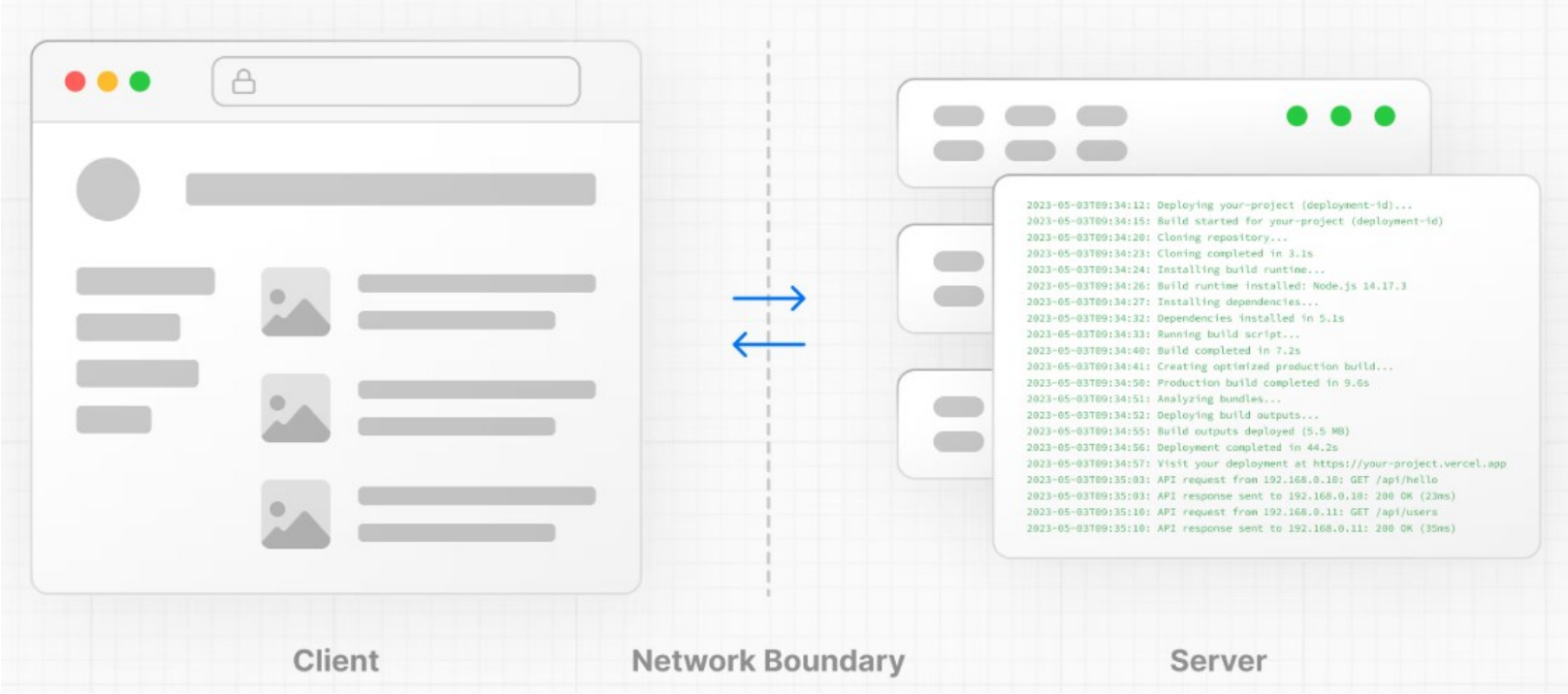
Installing Next.js6

- Run *npm run dev* in your terminal
- Navigate to http://localhost:3000:



Server and Client Environments[1]

- In the context of web applications:



Server and Client Environments2

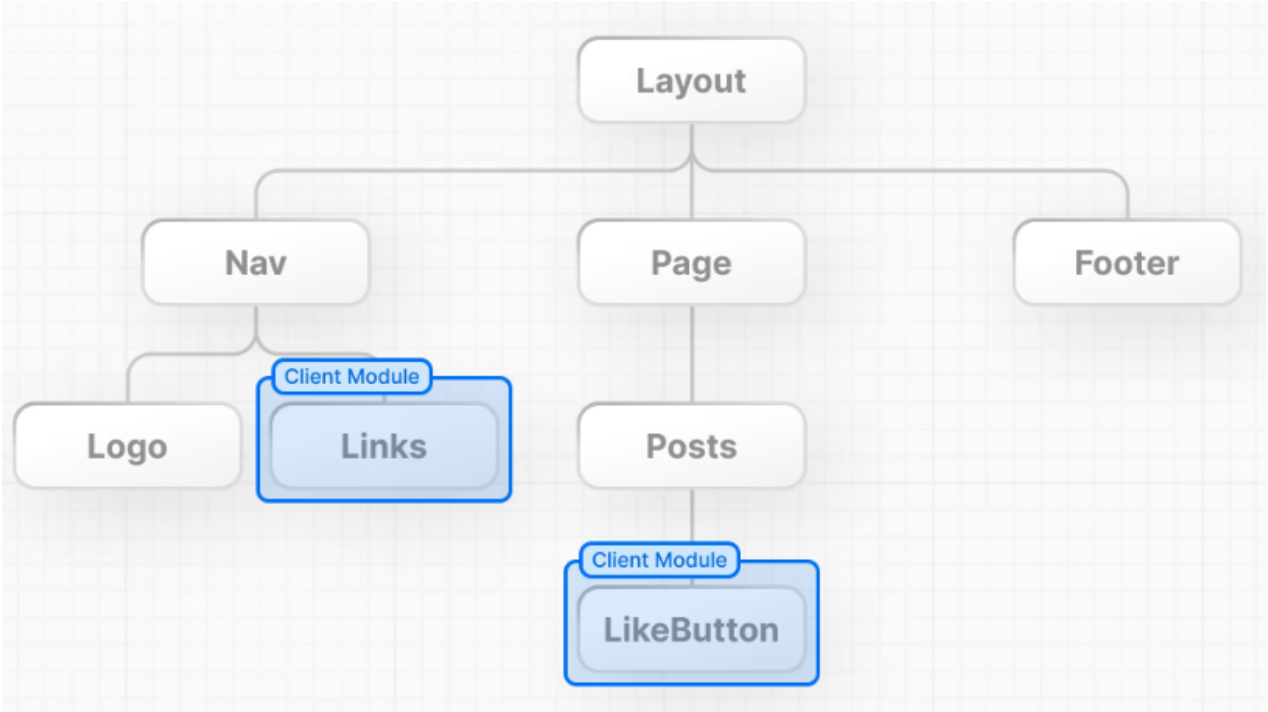
- The client refers to the browser on a user's device that sends a request to a server for your application code.
  - It then turns the response it receives from the server into an interface the user can interact with.
- The server refers to the computer in a data center that stores your application code, receives requests from a client, does some computation, and sends back an appropriate response.

Network Boundary1

- The Network Boundary is a conceptual line that separates the different environments.
- In React, you choose where to place the network boundary in your component tree.
  - You can fetch data and render a user's posts on the server (using Server Components).
  - You can render the interactive LikeButton for each post on the client (using Client Components).

Network Boundary2

- Another example: You can create a Nav component that is rendered on the server and shared across pages, but if you want to show an active state for links, you can render the list of Links on the client.



Network Boundary3

- The components are split into two module graphs:
  - The server module graph (or tree) contains all the Server Components that are rendered on the server
  - The client module graph (or tree) contains all Client Components.
- After Server Components are rendered, a special data format called the **React Server Component Payload (RSC)** is sent to the client.
  - The RSC payload contains:
    - The rendered result of Server Components.
    - Placeholders (or holes) for where Client Components should be rendered and references to their JavaScript files.

Using Client Components1

- Next.js **uses Server Components by default**.
  - To improve your application's performance
- Looking back at the error in your browser, Next.js is warning you that you're trying to useState inside a Server Component.
  - You can fix this by moving the interactive "Like" button to a Client Component.

Using Client Components2

- Create a new LikeButton component.
- Add the React **'use client'** directive at the to tell React to render the component on the client.

```
'use client';

import { useState } from 'react';

export default function LikeButton() {
    const [likes, setLikes] = useState(0);

    function handleClick() {
        setLikes(likes + 1);
    }

    return <button onClick={handleClick}>Like ({likes})</button>;
}
```

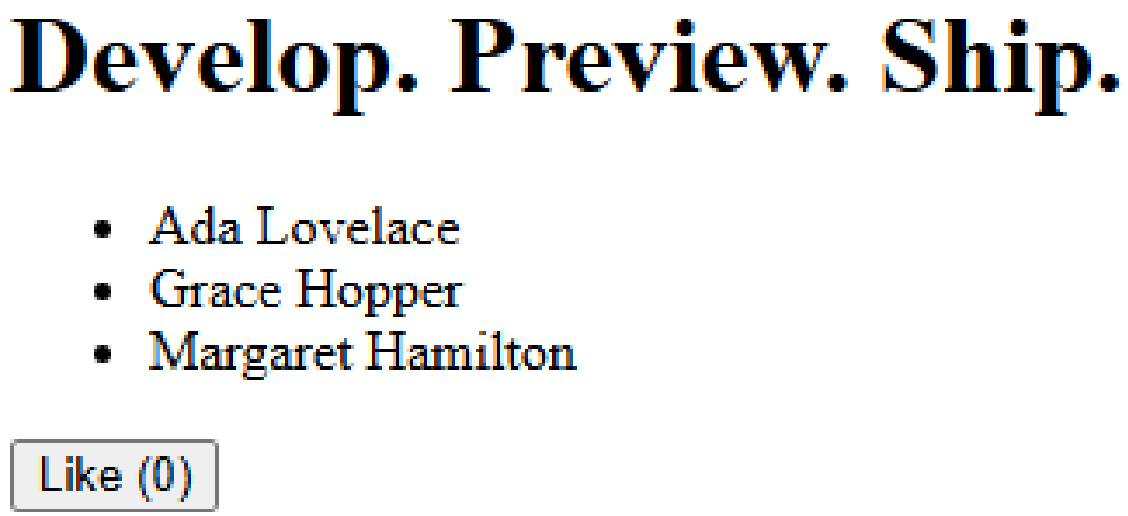
Using Client Components 3

- Import the LikeButton component into your page

```
import LikeButton from './like-button';

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

export default function HomePage() {
  const names = ['Ada Lovelace', 'Grace Hopper', 'Margaret Hamilton'];

  return (
    <div>
      …
      <LikeButton />
    </div>
  );
}
```

Using Client Components 4

- Run *npm run dev* in your terminal
- Navigate to http://localhost:3000:



Creating a Default NEXT.JS App

- Using pnpm
  - npm install -g pnpm
  - pnpm create next-app my-next-app
  - cd my-next-app
  - pnpm dev
- Using npm
  - npx create-next-app@latest my-next-app
  - cd my-next-app
  - npm run dev

Any Question?