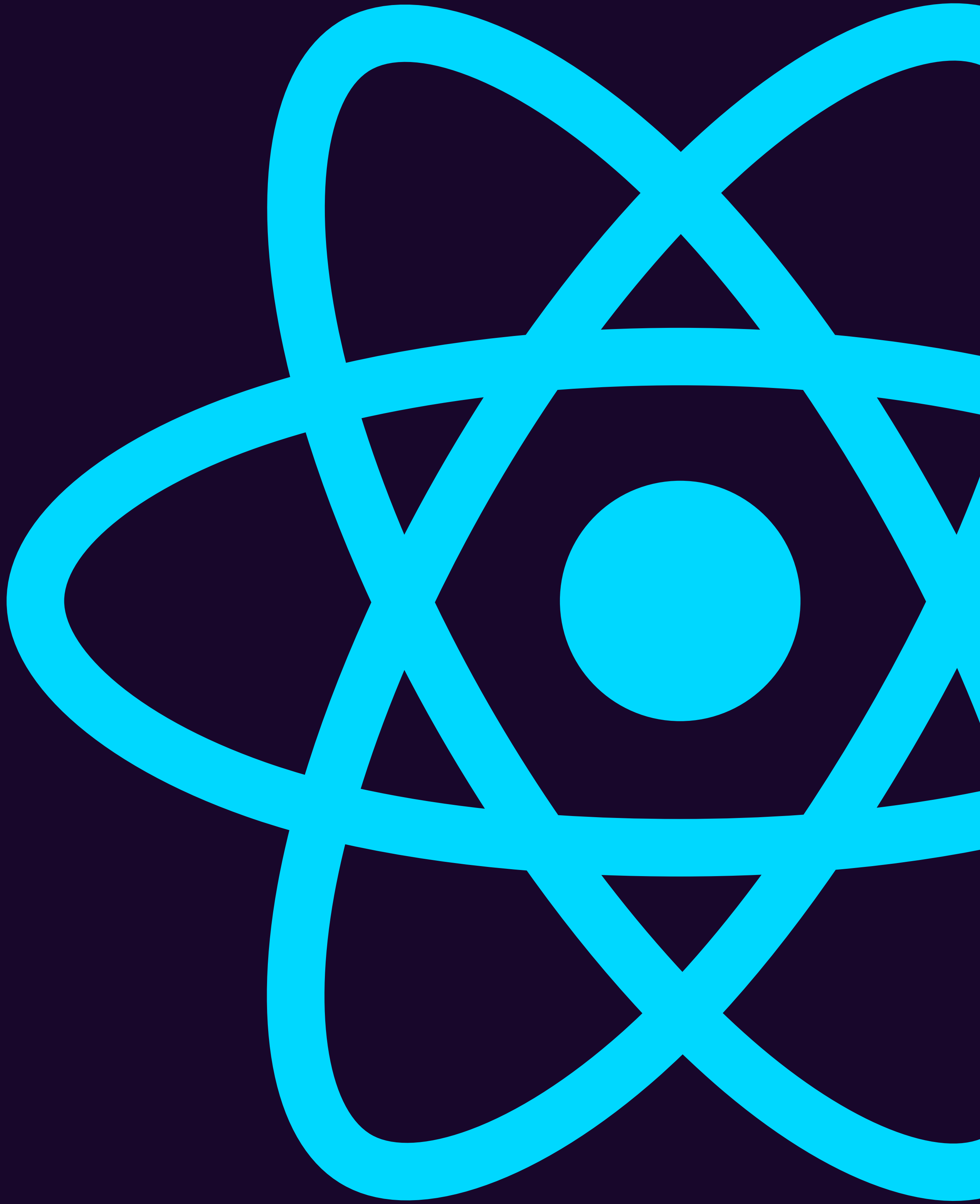
NJACK IITP

INTRODUCING

REACTJS

www.react.dev

MAY 2023



AGENDA

- INTRODUCTION TO REACT
- TOOLS AND SETUP
- INSTALLING REACT
- JSX
- COMPONENTS IN REACT
- STATES IN REACT

Introduction to React

React is javascript library used for creating web and native user interfaces (aka frontend framework)

Fast

Modular

Scalable

Flexible

Popular

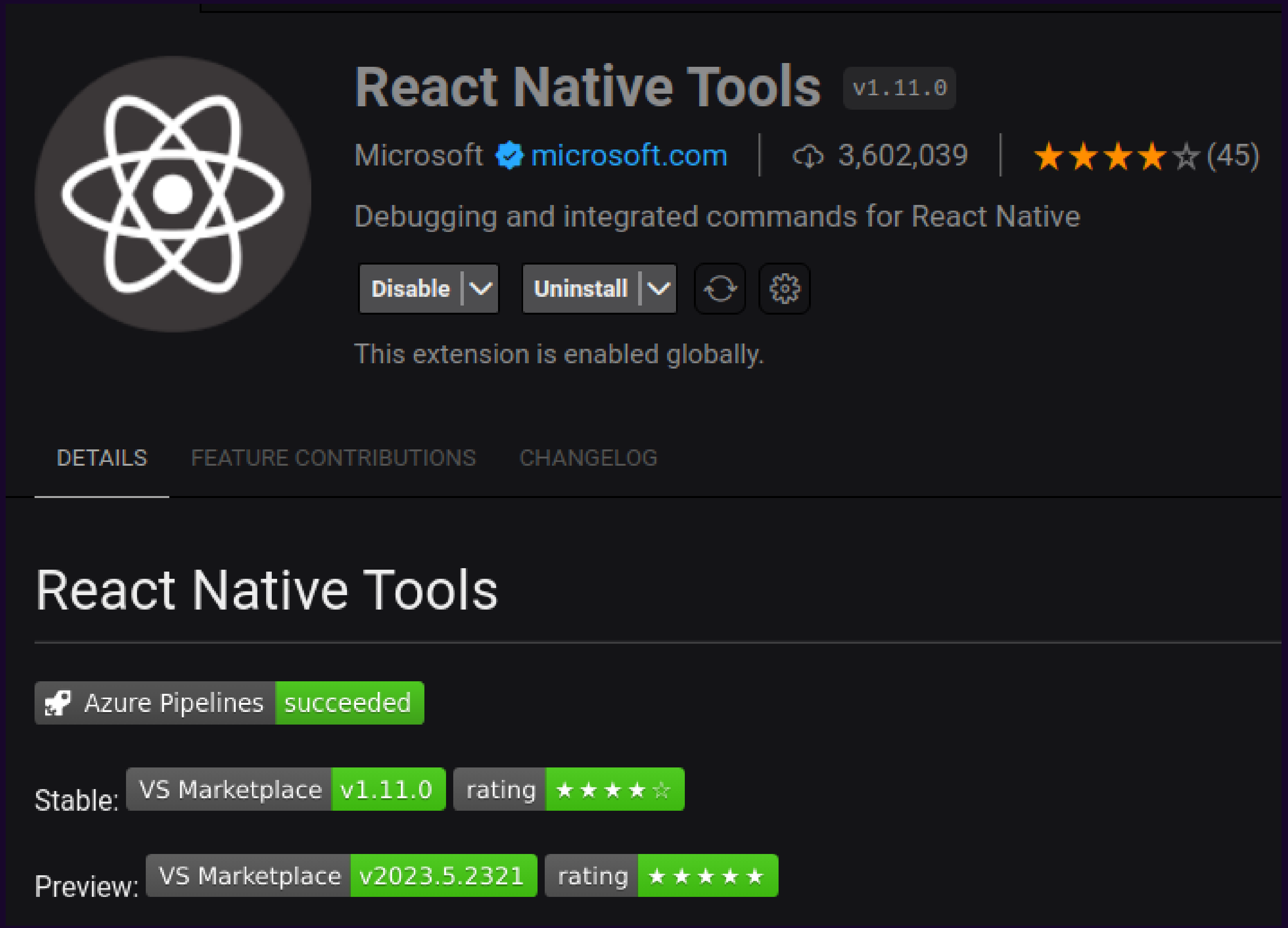
Requirements

- Knowledge of basic HTML and Javascript is a requirement
- No prior knowledge of react or any libraries/frameworks required

Set up tools

The required tools are:

- VS Code (IDE)
- Chrome/Firefox/Edge
- Nodejs v12+
- Extensions
  - ES7+ React/Redux/React-Native snippets



React Native Tools v1.11.0

Microsoft microsoft.com | 3,602,039 | ★★★★☆ (45)

Debugging and integrated commands for React Native

This extension is enabled globally.

React Native Tools

Azure Pipelines succeeded

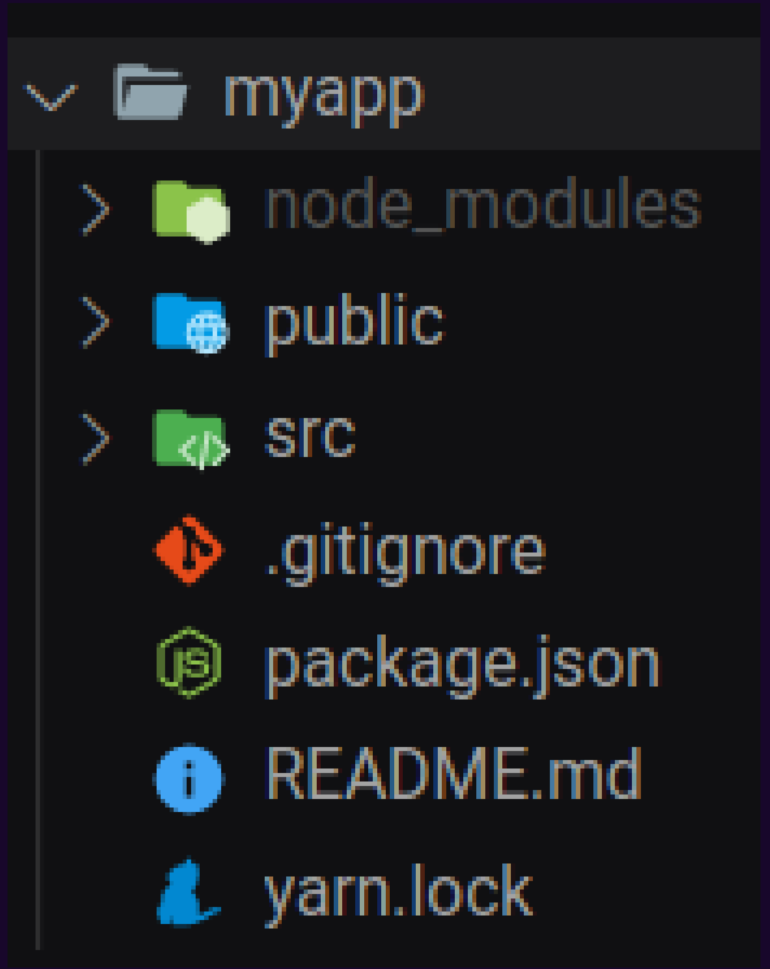
Stable: VS Marketplace v1.11.0 rating ★★★★☆

Preview: VS Marketplace v2023.5.2321 rating ★★★★★

Installing React

To start a new react project, run the following commands in terminal

- `npm init -y`
- `npx create-react-app myApp or yarn create react-app myapp`
- `cd myapp`
- `npm run start or yarn start`



```
myapp
  node_modules
  public
  src
  .gitignore
  package.json
  README.md
  yarn.lock
```

Note: React does not officially recommed create-react-app cli, but it is easier for beginners to learn. For production grade applications, use nextjs, gatsby, or remix, all of which are production ready frameworks.

Build Systems for React

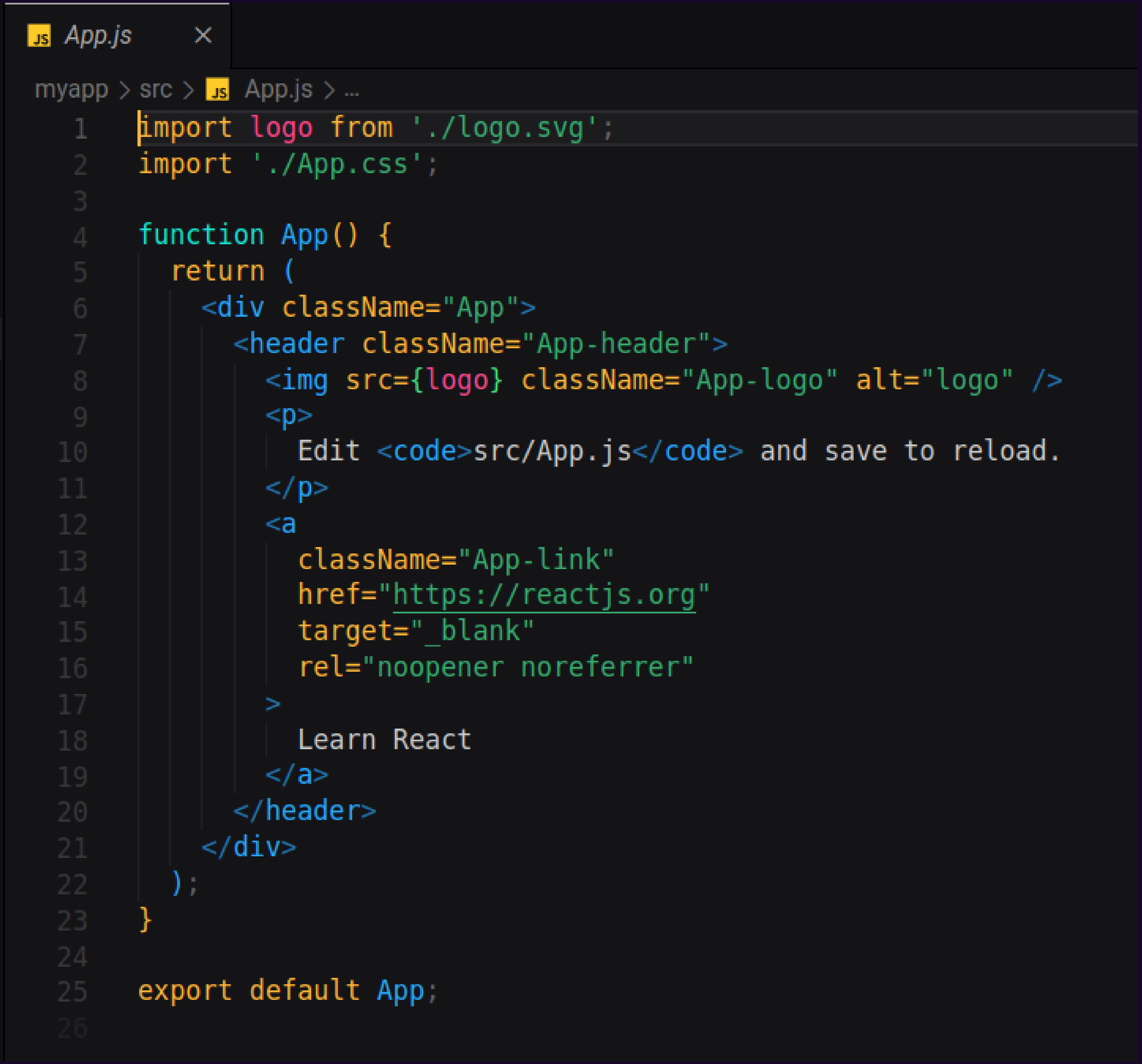
- Create-react-app configures a frontend build pipeline using **Webpack** module bundler, **babel** compiler, and **ESLint** code linter
- While React can be used without a build pipeline, it is recommended to use one for productivity. A build pipeline consists of:
    - **Package managers**, like yarn or npm
    - **Bundler**, like Webpack or Browserify
    - **Compiler** like Babel (to support modern ES6 and JSX syntax)
    - To install only core react libraries, without the entire pipeline, run: npm i react react-dom or yarn add react react-dom
- Another option is to use react by **CDN**
- Using custom build systems are often important for production grade applications, so here is a guide to setting up a react build system: https://dev.to/ivadyhabimana/how-to-create-a-react-app-without-using-create-react-app-a-step-by-step-guide-30nl

JSX

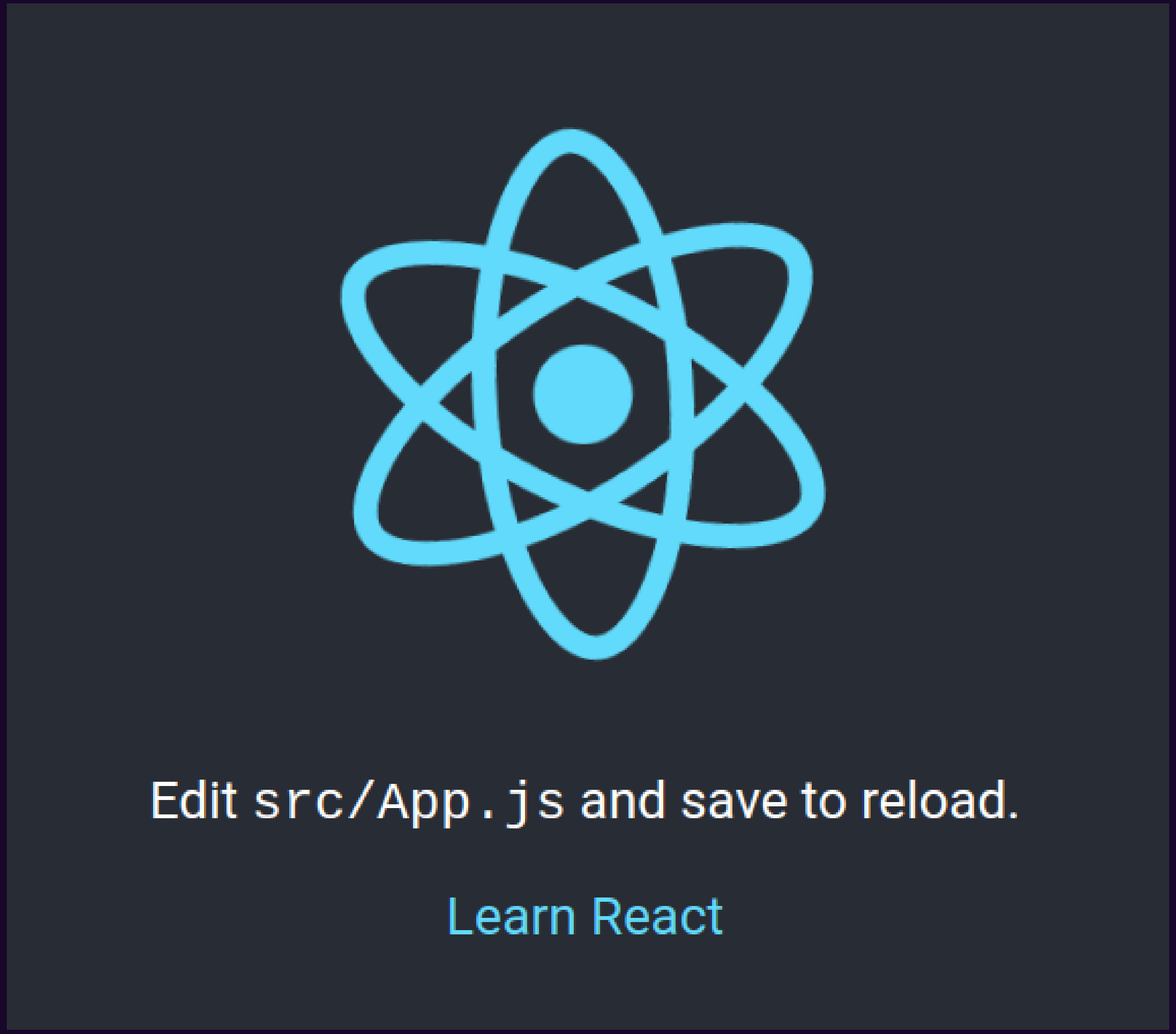
The non-standard way of writing HTML and Javascript

JSX is a syntax extension for JavaScript that lets you write HTML-like markup inside a JavaScript file. Although there are other ways to write components, most React developers prefer the conciseness of JSX, and most codebases use it.

JSX and React are two separate things. They’re often used together, but you can **use them independently** of each other. JSX is a syntax extension, while React is a JavaScript library.



```jsx
import logo from './logo.svg';
import './App.css';

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

export default App;
```



Rules of JSX

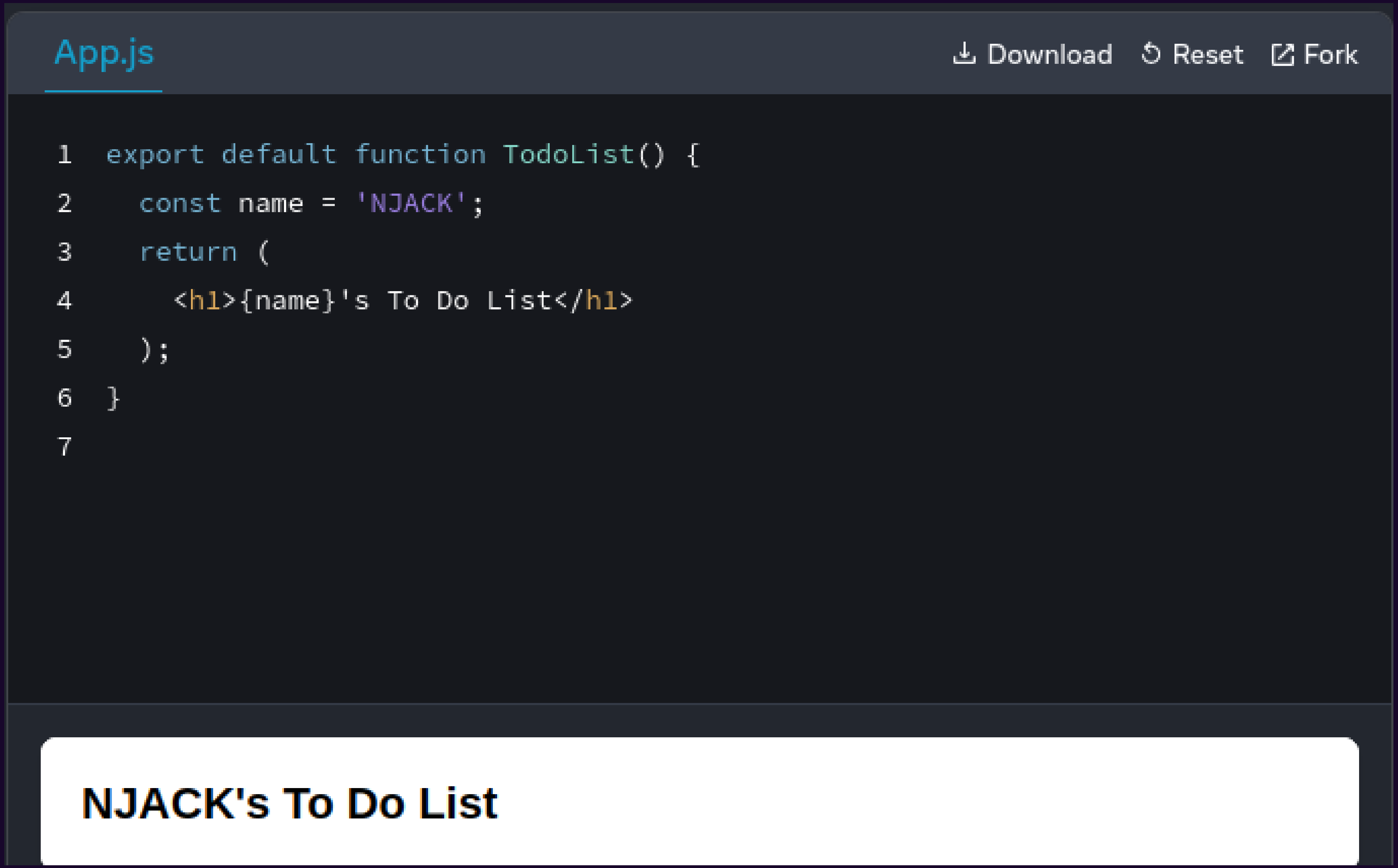
- Return a single root element
- Close all the tags
- camelCase most of the things (class becomes className, stroke-width becomes strokeWidth, as javascript does not allow dashes in attribute names)

visit https://react.dev/learn/writing-markup-with-jsx#the-rules-of-jsx for details

Tip: Use a HTML to JSX converter to initially understand how JSX syntax works:
https://transform.tools/html-to-jsx

Javascript in JSX

Javascript statements can be inserted into JSX using curly brackets {}

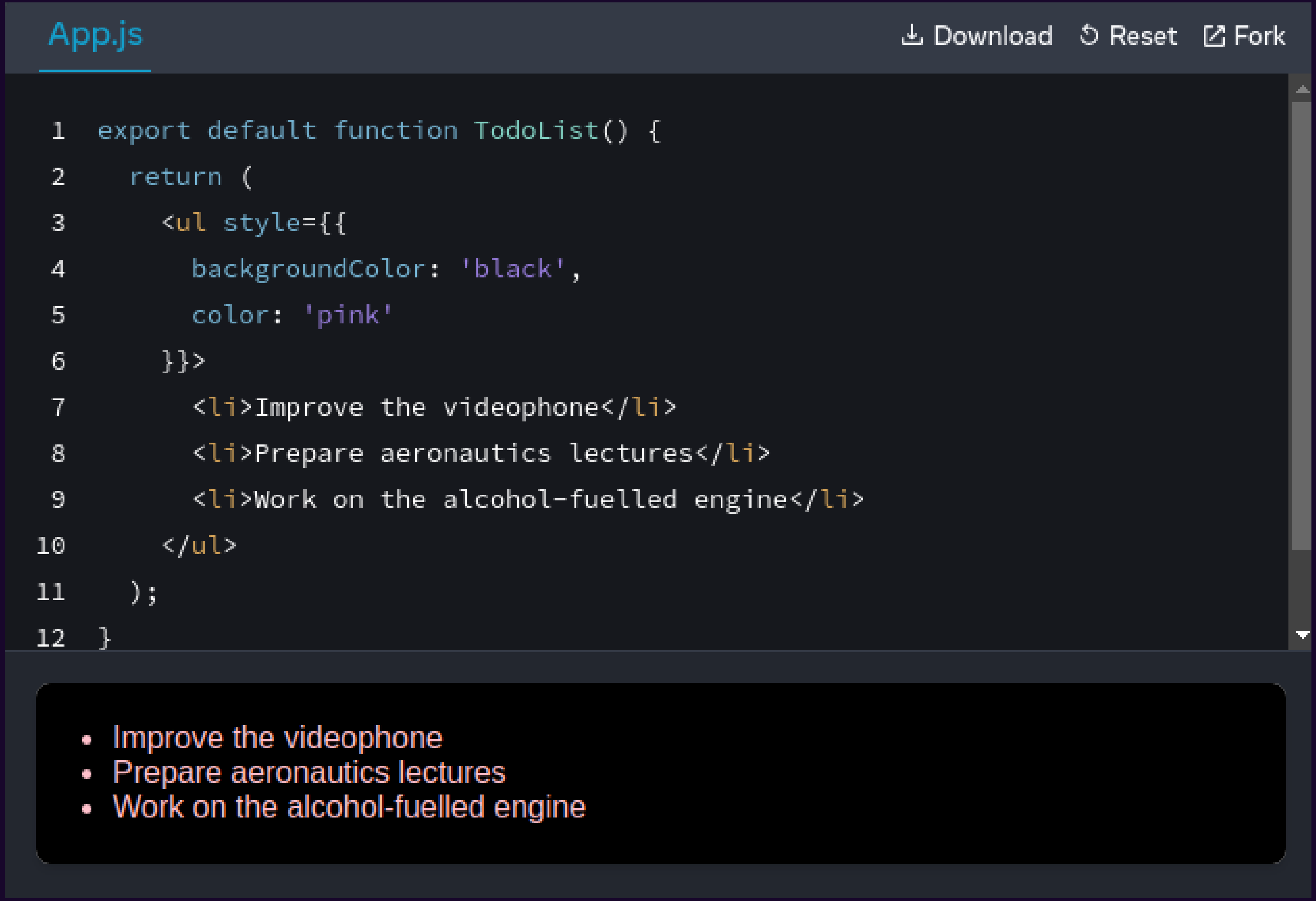


```
export default function TodoList() {
  const name = 'NJACK';
  return (
    <h1>{name}'s To Do List</h1>
  );
}
```

NJACK's To Do List

CSS in JSX

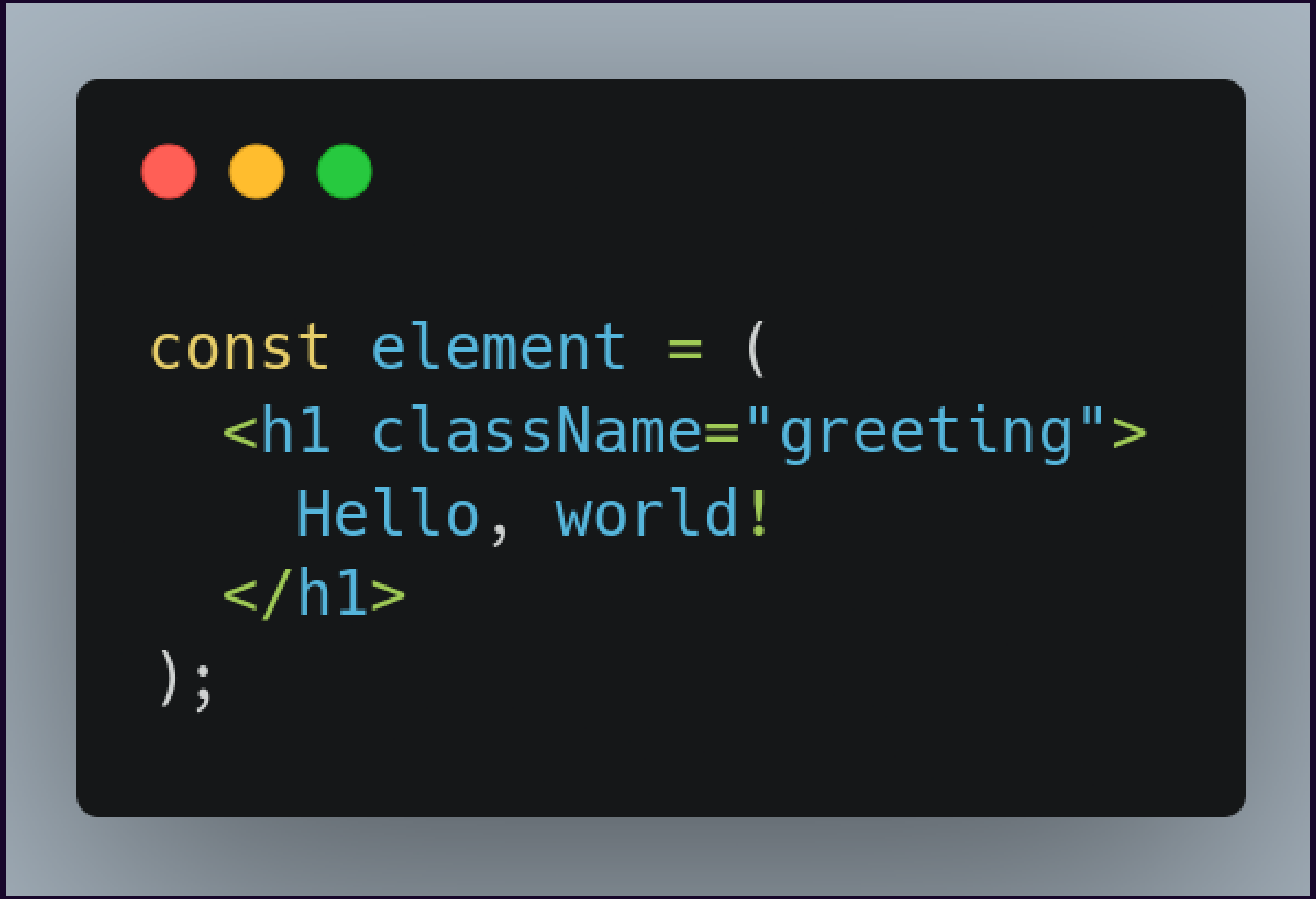
CSS statements can be inserted into JSX using double curly brackets {}, as they are represented as javascript objects.



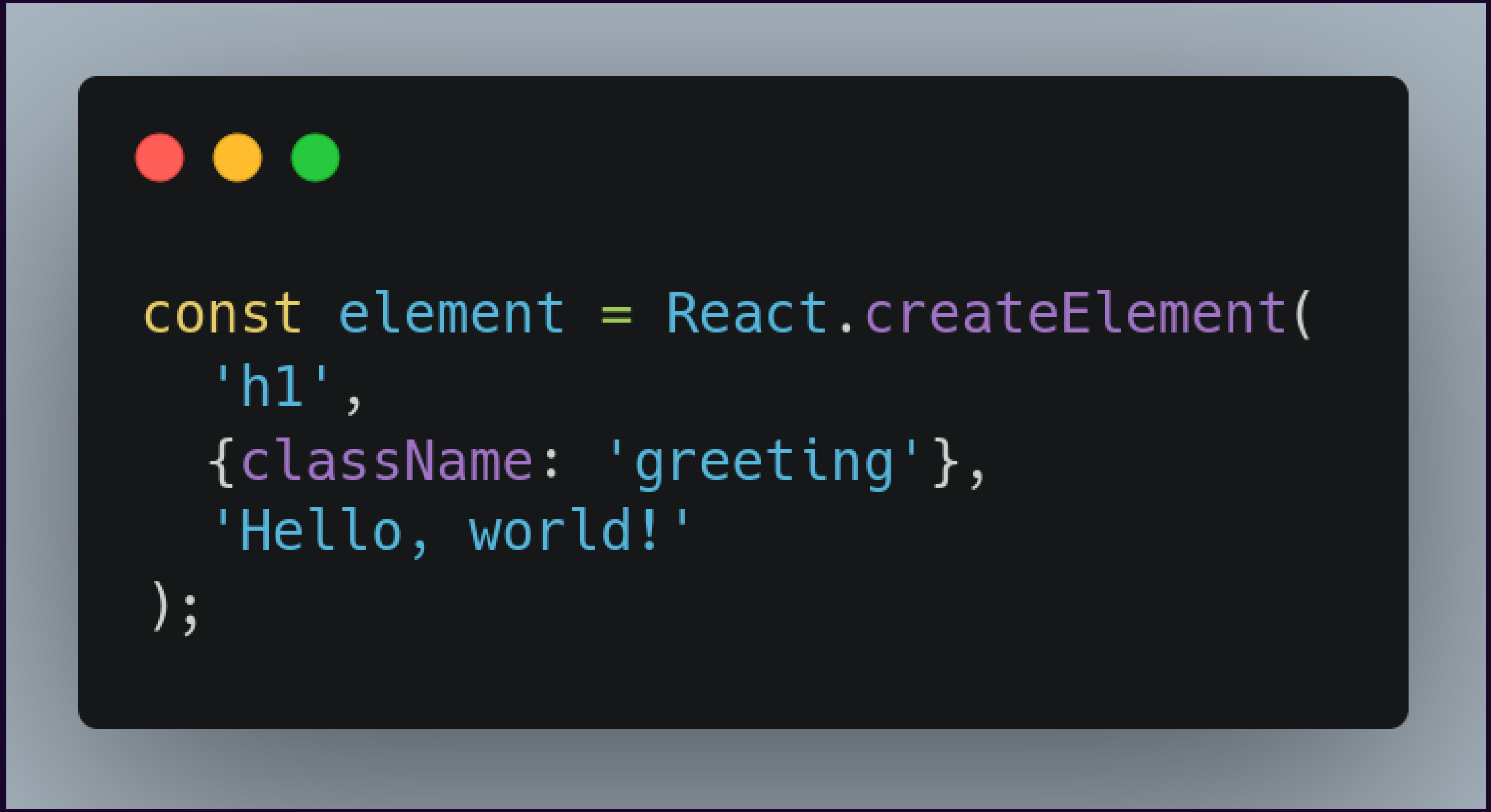
```
export default function TodoList() {
  return (
    <ul style={{
      backgroundColor: 'black',
      color: 'pink'
    }}>
      <li>Improve the videophone</li>
      <li>Prepare aeronautics lectures</li>
      <li>Work on the alcohol-fuelled engine</li>
    </ul>
  );
}
```

- Improve the videophone
- Prepare aeronautics lectures
- Work on the alcohol-fuelled engine

React can be used without JSX too



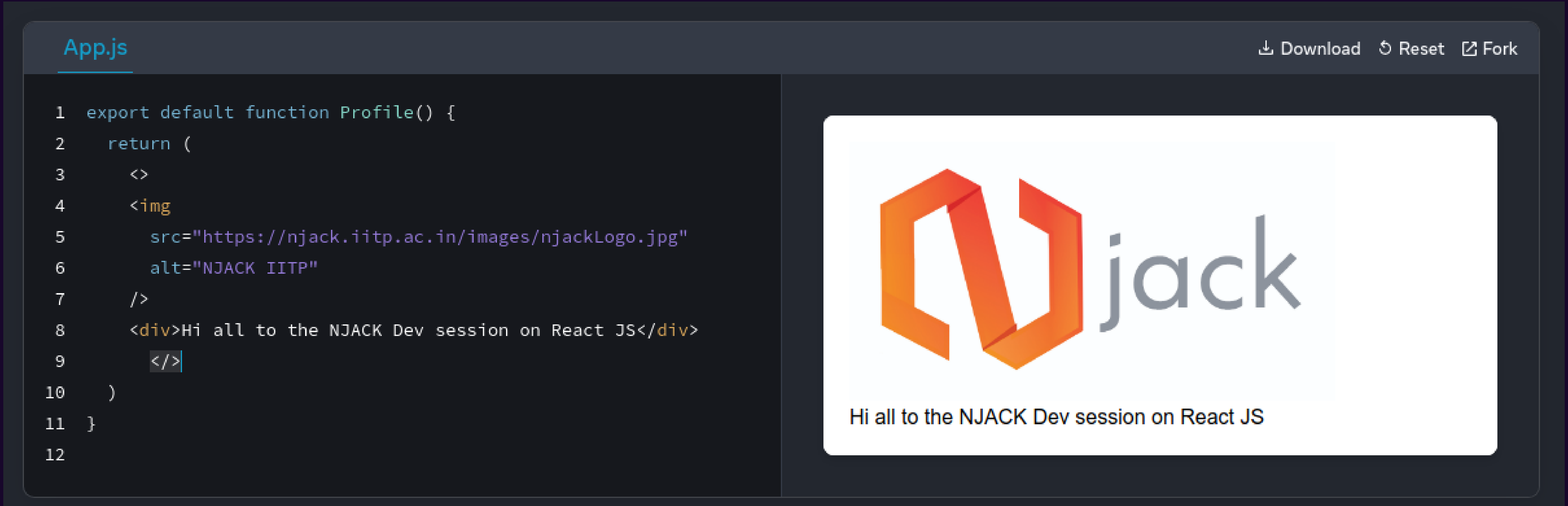
```
const element = (
  <h1 className="greeting">
    Hello, world!
  </h1>
);
```



```
const element = React.createElement(
  'h1',
  {className: 'greeting'},
  'Hello, world!'
);
```

Components

- Components are reusable UI elements for your App
- In a React app, every piece of UI is a component.
- React components are regular JavaScript functions except:
  - Their names always begin with a capital letter.
  - They return JSX markup.



```
export default function Profile() {
  return (
    <>
    <img
      src="https://njack.iitp.ac.in/images/njackLogo.jpg"
      alt="NJACK IITP"
    />
    <div>Hi all to the NJACK Dev session on React JS</div>
      </>
  )
}
```

Hi all to the NJACK Dev session on React JS

Steps to build Component

Step1: Export the Component

The export default prefix is a standard JavaScript syntax (not specific to React). It lets you mark the main function in a file so that you can later import it from other files.

Step2: Define the function

With function Profile() { } you define a JavaScript function with the name Profile. React components are regular JavaScript functions, but their names must start with a capital letter or they won’t work!

Step3: Add the markup

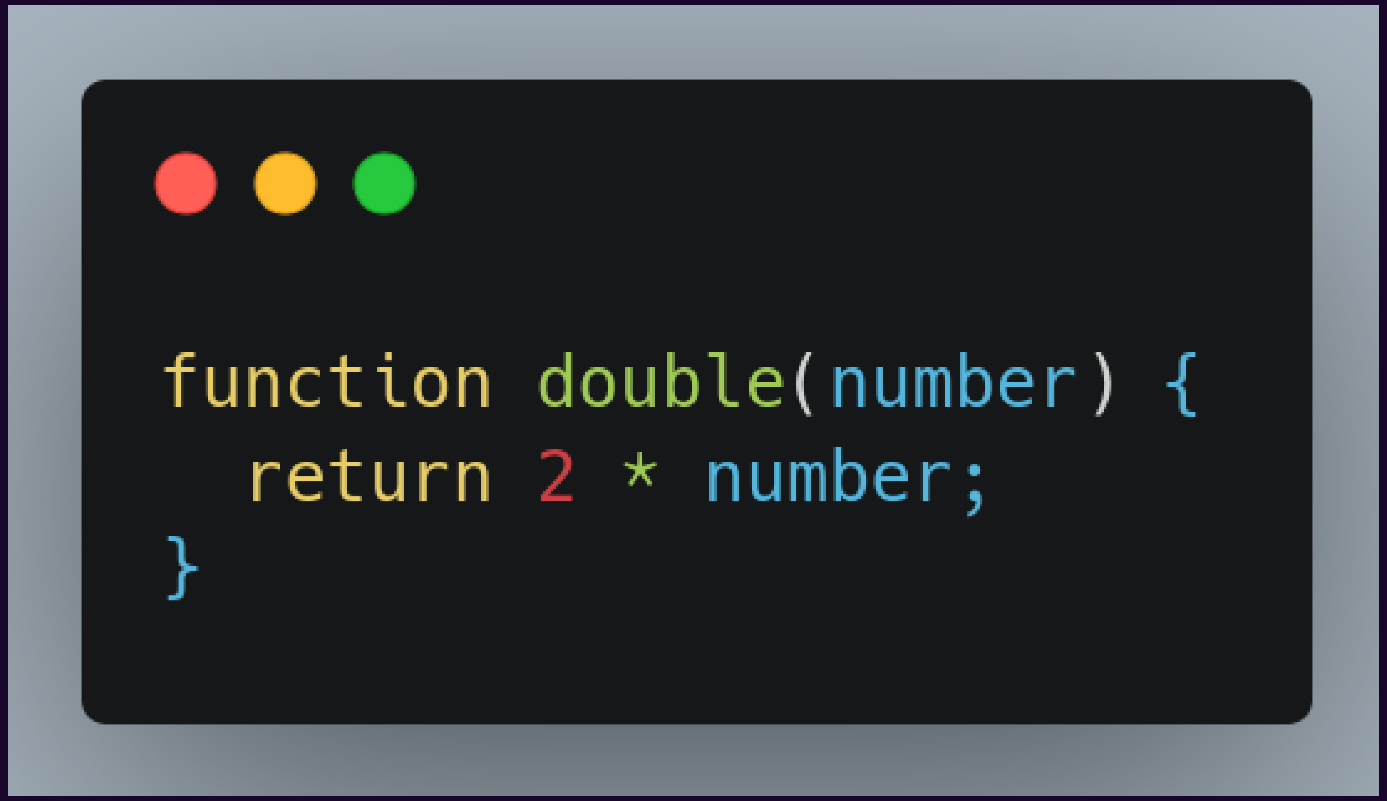
Add the JSX that needs to be returned. But if your markup isn’t all on the same line as the return keyword, you must wrap it in a pair of parentheses.

React components must be pure

A pure function is a function with following characteristics

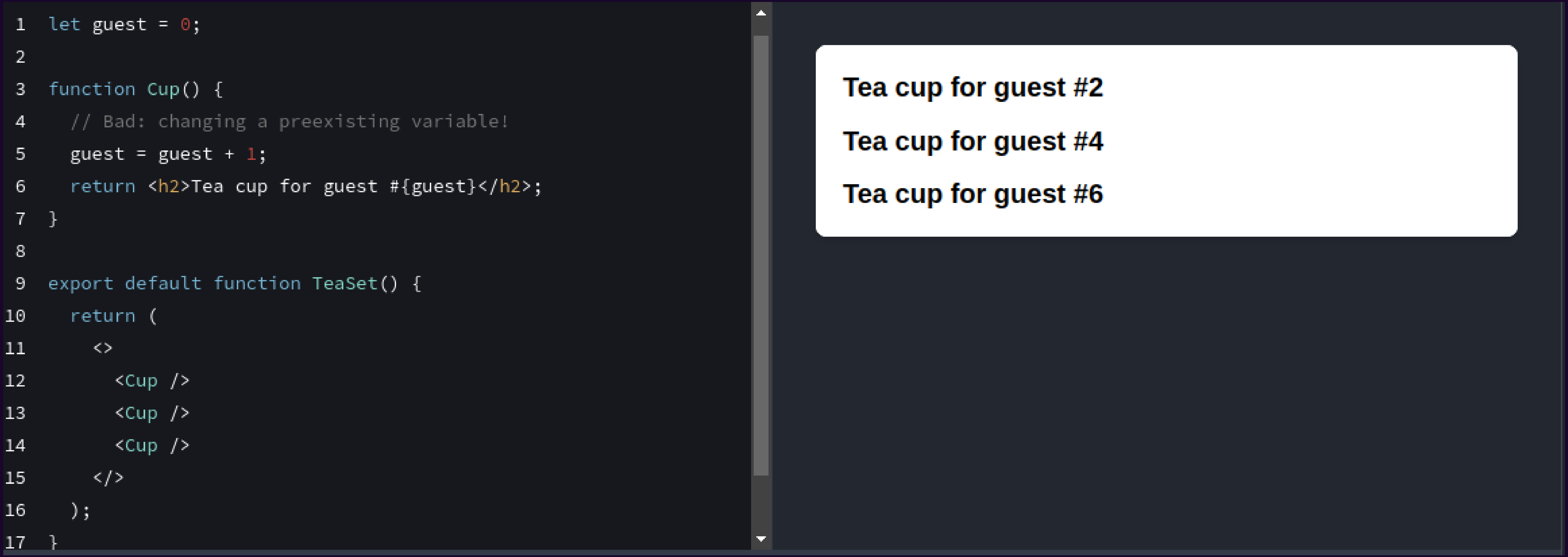
- It does not change any objects or variables that existed before it was called
- Given the same inputs, a pure function should always return the same result

React is designed around this concept. React assumes that every component you write is a pure function. This means that React components you write must always return the same JSX given the same inputs



```
function double(number) {
  return 2 * number;
}
```

Example of impure component



```
let guest = 0;

function Cup() {
  // Bad: changing a preexisting variable!
  guest = guest + 1;
  return <h2>Tea cup for guest #{guest}</h2>;
}

export default function TeaSet() {
  return (
    <>
      <Cup />
      <Cup />
      <Cup />
    </>
  );
}
```

```
Tea cup for guest #2
Tea cup for guest #4
Tea cup for guest #6
```

Using React's "Strict Mode"

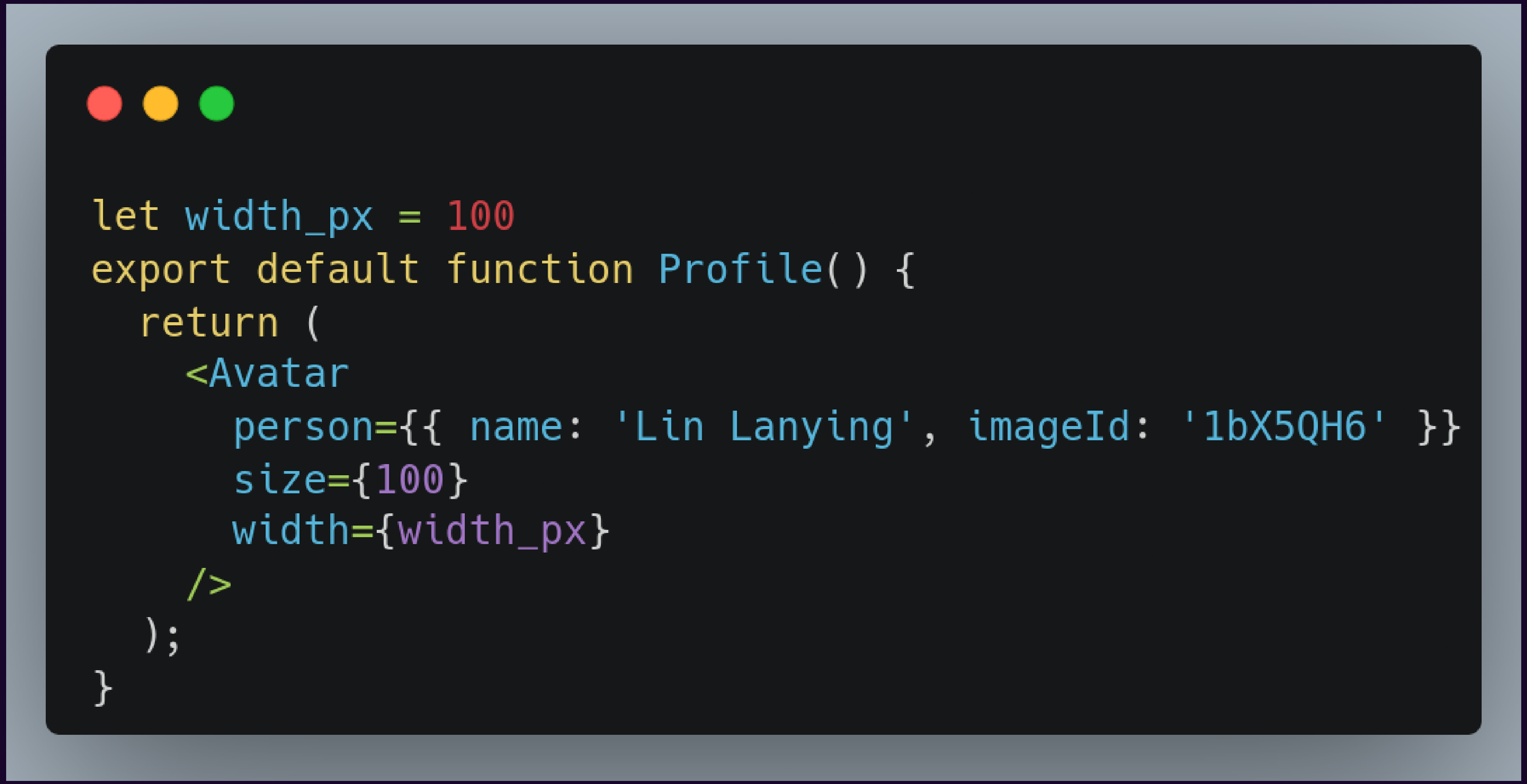
React's "Strict Mode" can be used to detect impure calculations during development.

React offers a “Strict Mode” in which it calls each component’s function twice during development. By calling the component functions twice, Strict Mode helps find components that break these rules.

There are ways in which we can update the render within a react component, which we will be discussing later.

"props" in react

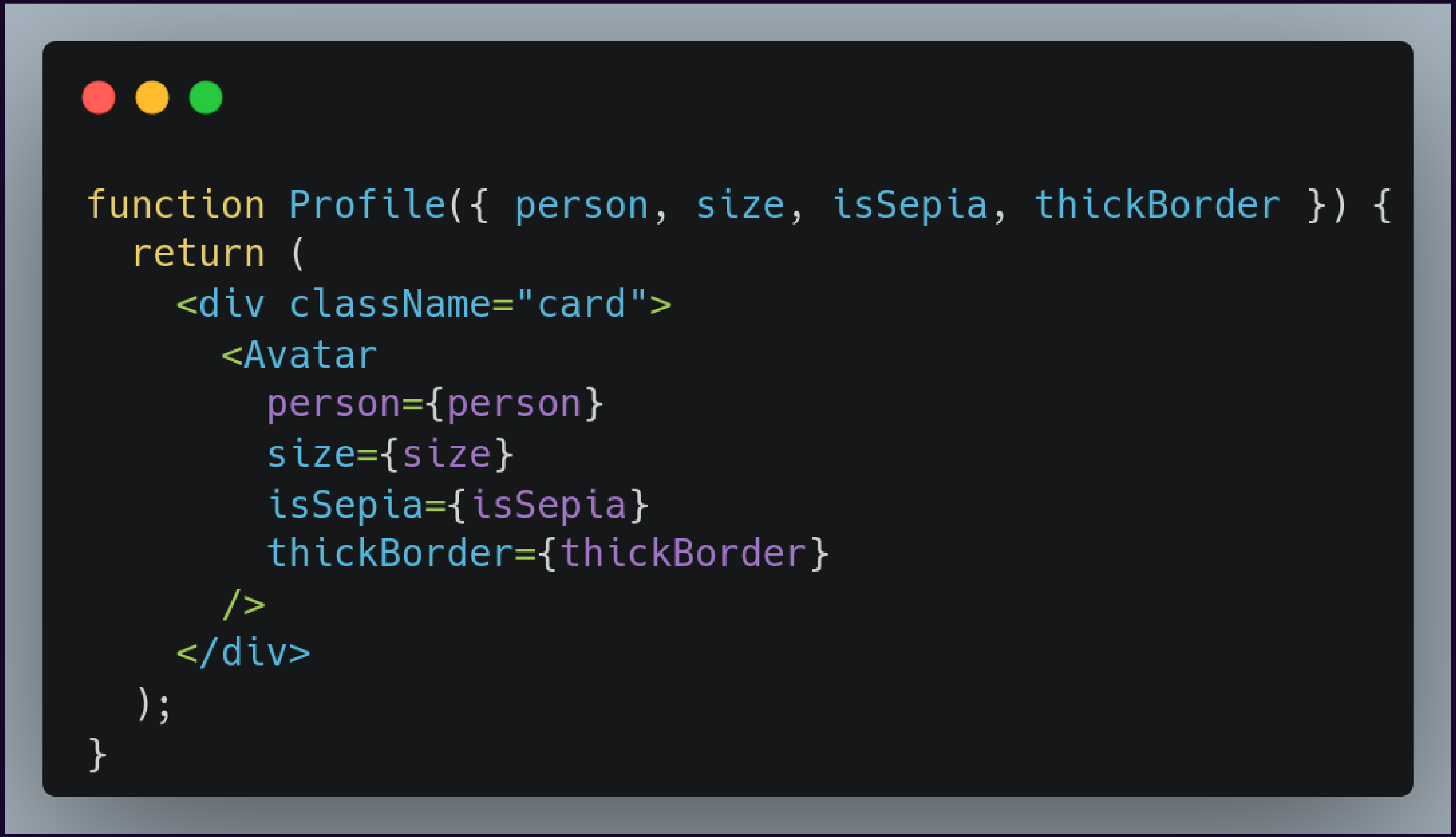
React components use props to communicate with each other. Every parent component can pass some information to its child components by giving them props. Props might remind you of HTML attributes, but you can pass any JavaScript value through them, including objects, arrays, and functions.



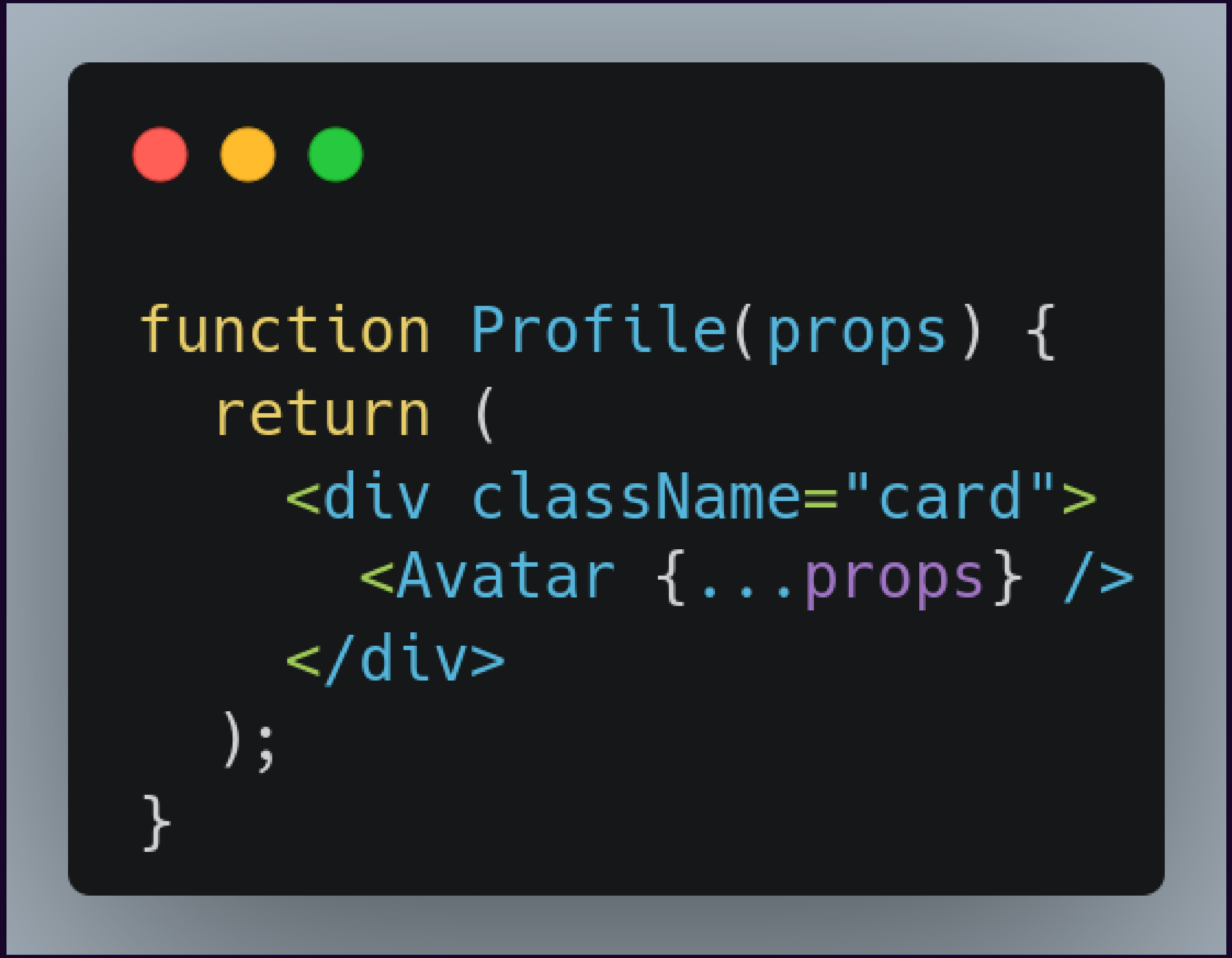
```
let width_px = 100
export default function Profile() {
  return (
    <Avatar
      person={{ name: 'Lin Lanying', imageId: '1bX5QH6' }}
      size={100}
      width={width_px}
    />
  );
}
```

Forwarding props with the JSX spread syntax

Sometimes, passing props gets very repetitive:

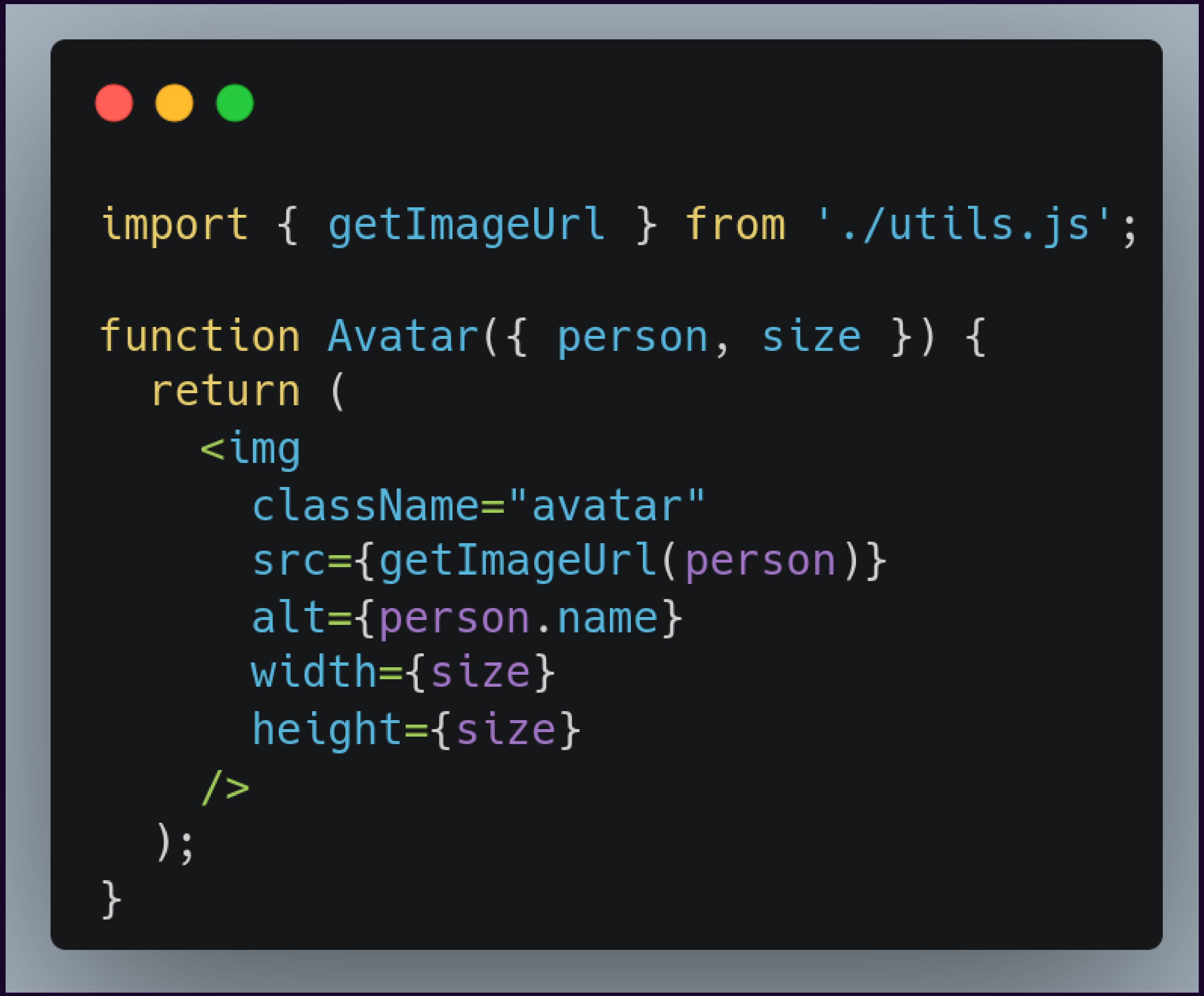


```jsx
function Profile({ person, size, isSepia, thickBorder }) {
  return (
    <div className="card">
      <Avatar
        person={person}
        size={size}
        isSepia={isSepia}
        thickBorder={thickBorder}
      />
    </div>
  );
}
```

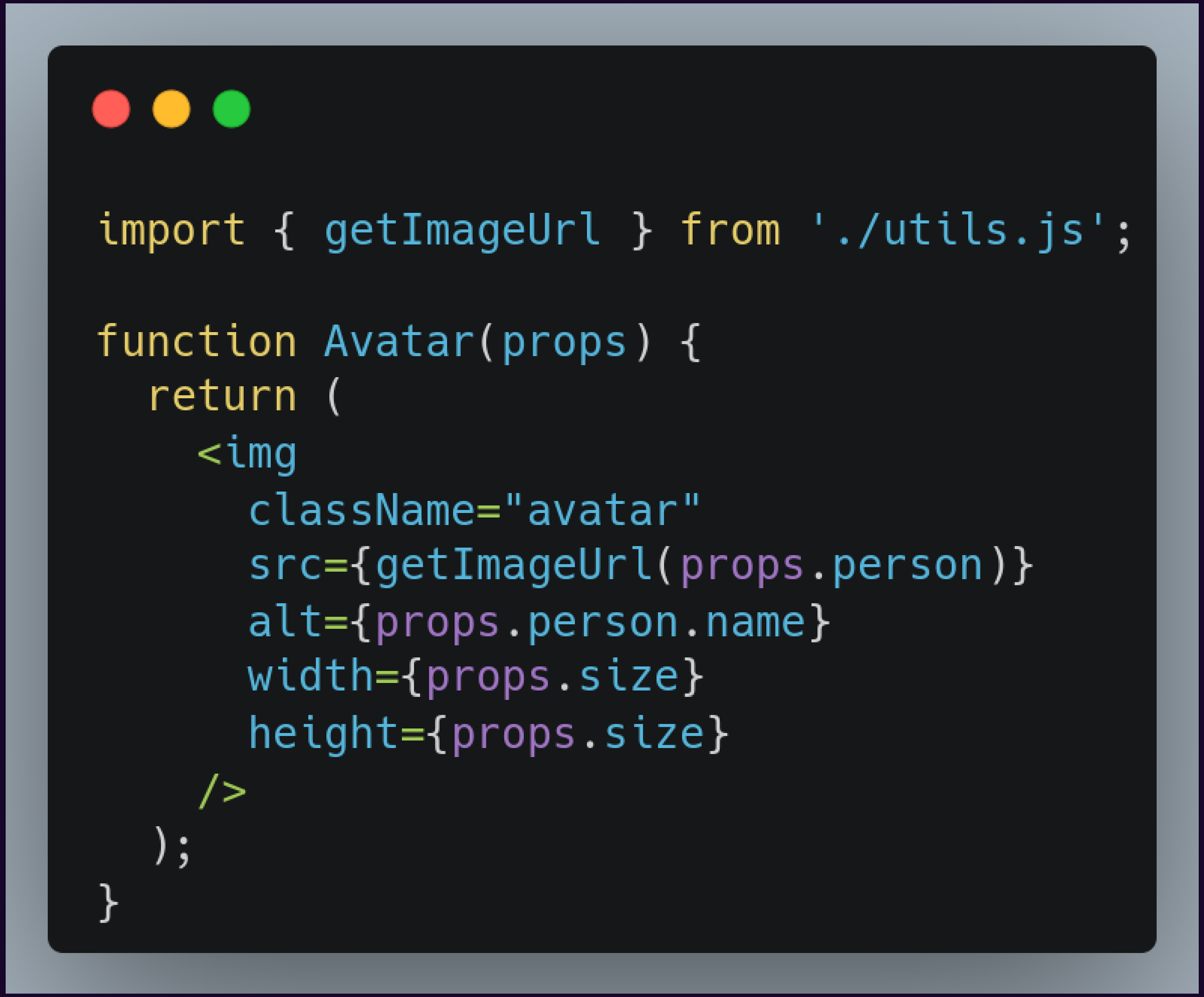


```jsx
function Profile(props) {
  return (
    <div className="card">
      <Avatar {...props} />
    </div>
  );
}
```

Reading props from child



```jsx
import { getImageUrl } from './utils.js';

function Avatar({ person, size }) {
  return (
    <img
      className="avatar"
      src={getImageUrl(person)}
      alt={person.name}
      width={size}
      height={size}
    />
  );
}
```



```jsx
import { getImageUrl } from './utils.js';

function Avatar(props) {
  return (
    <img
      className="avatar"
      src={getImageUrl(props.person)}
      alt={props.person.name}
      width={props.size}
      height={props.size}
    />
  );
}
```

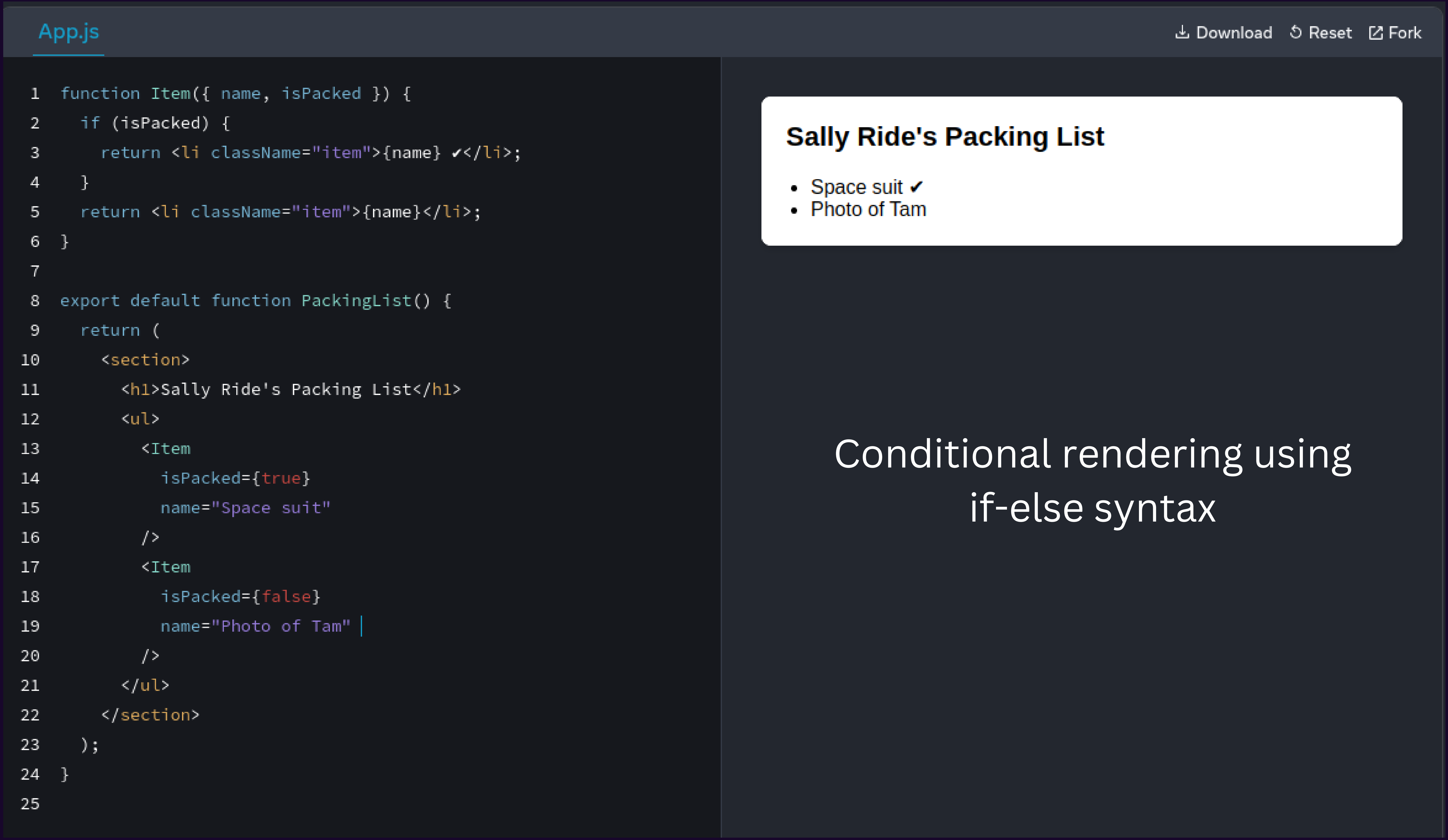
Props changing over time

A component may receive different props over time. Props are not always static! Props reflect a component’s data at any point in time, rather than only in the beginning.

However, props are immutable—a term from computer science meaning “unchangeable”. When a component needs to change its props (for example, in response to user interaction or new data), it will have to “ask” its parent component to pass it different props—a new object! Its old props will then be cast aside, and eventually the JavaScript engine will reclaim the memory taken by them.

Conditional Rendering

Your components will often need to display different things depending on different conditions. In React, you can conditionally render JSX using JavaScript syntax like if statements, &&, and ? : operators.


App.js

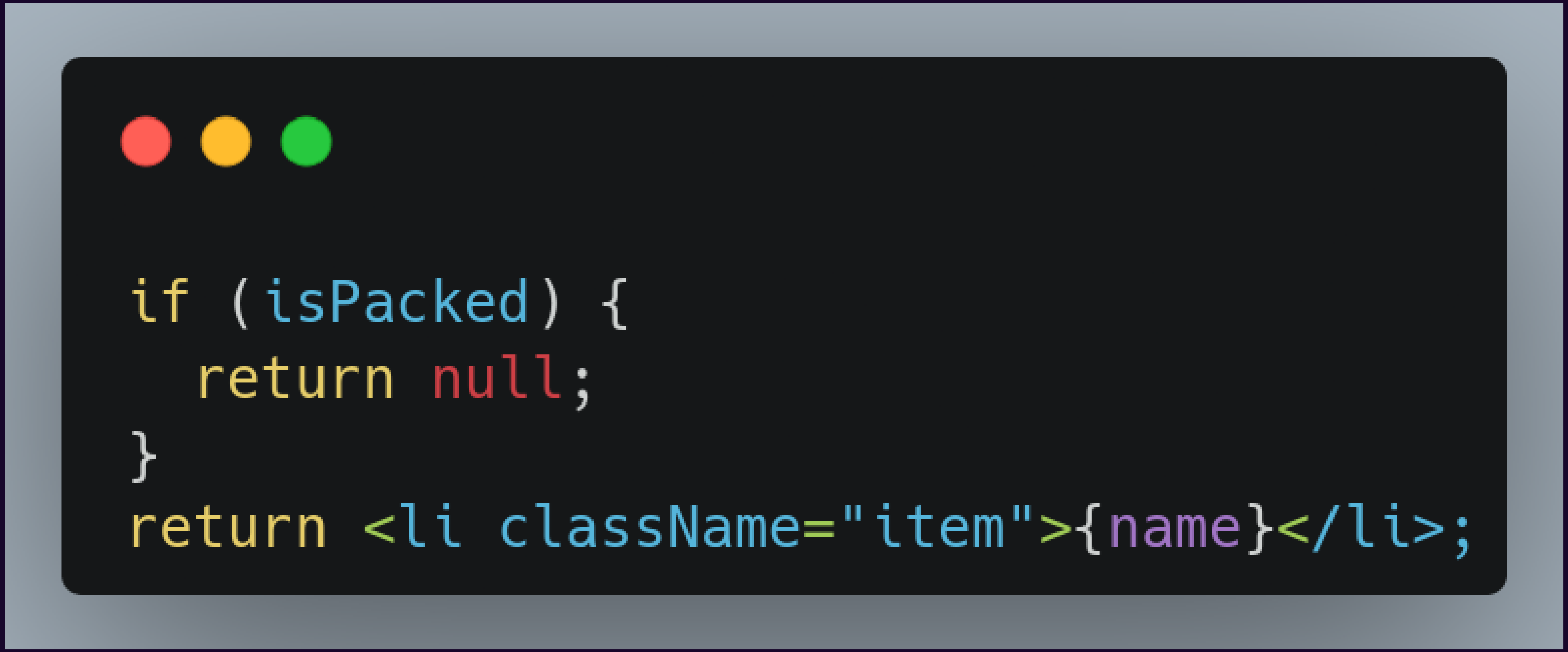
```
function Item({ name, isPacked }) {
  if (isPacked) {
    return <li className="item">{name} ✔</li>;
  }
  return <li className="item">{name}</li>;
}

export default function PackingList() {
  return (
    <section>
      <h1>Sally Ride's Packing List</h1>
      <ul>
        <Item
          isPacked={true}
          name="Space suit"
        />
        <Item
          isPacked={false}
          name="Photo of Tam"
        />
      </ul>
    </section>
  );
}

```

Sally Ride's Packing List

- Space suit ✔
- Photo of Tam

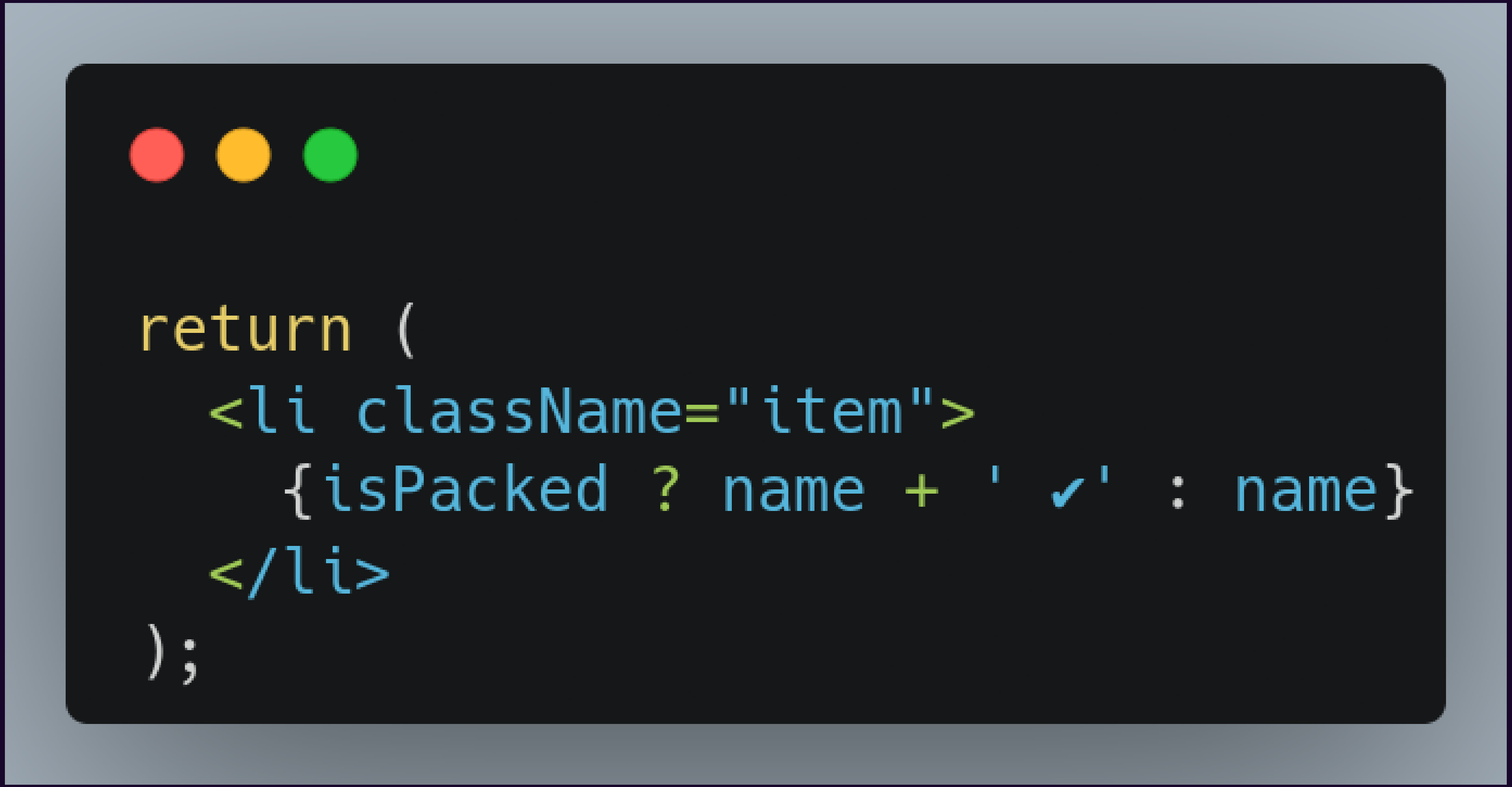
Conditional rendering using if-else syntax

Sometimes we need to return nothing on some condition



```
if (isPacked) {
  return null;
}
return <li className="item">{name}</li>;
```

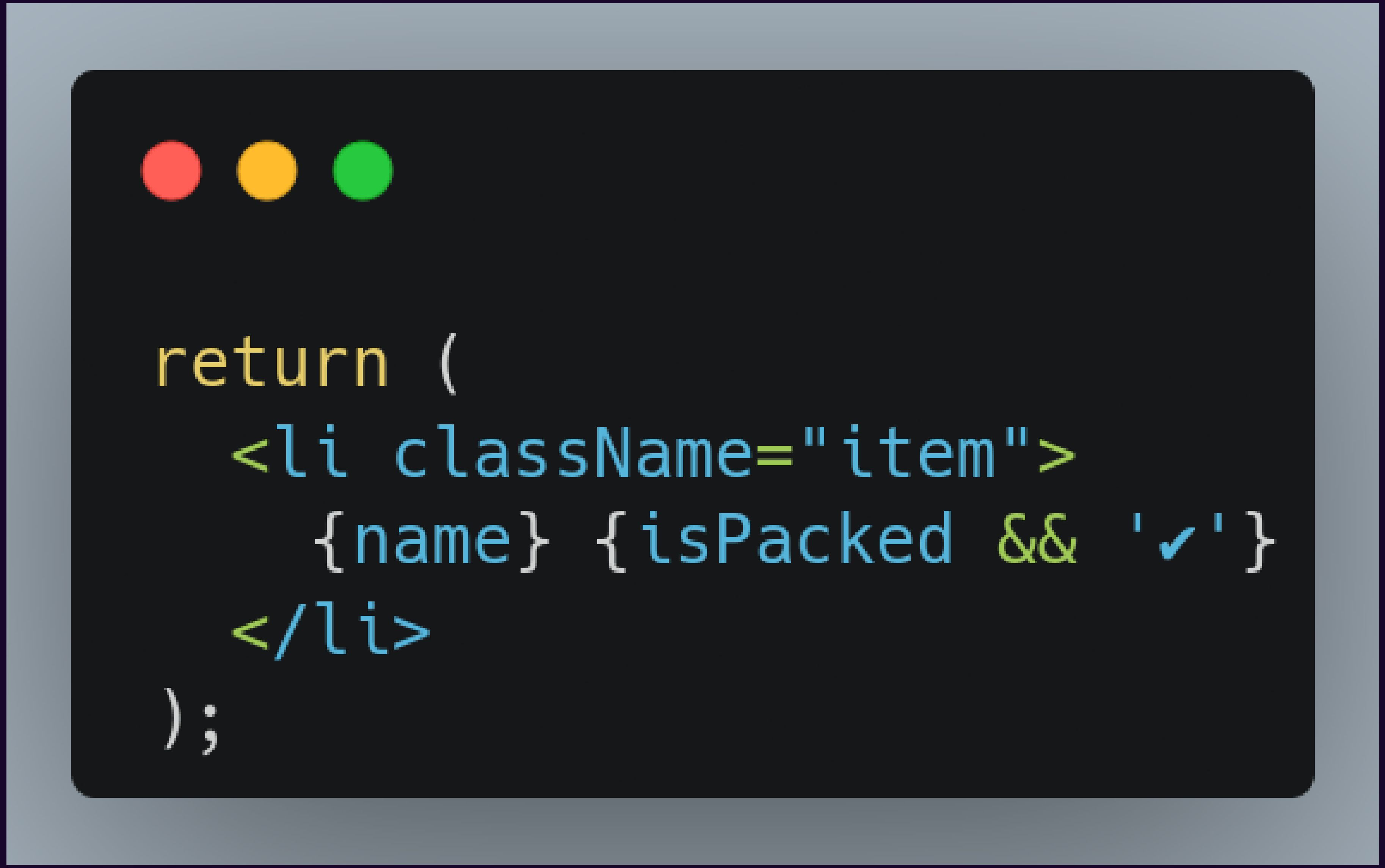
Using javascript ternary operator



```
return (
  <li className="item">
    {isPacked ? name + ' ✔' : name}
  </li>
);
```

You can read it as “if isPacked is true, then (?) render name + ' ✔',
otherwise (:) render name”.

Using javascript && operator

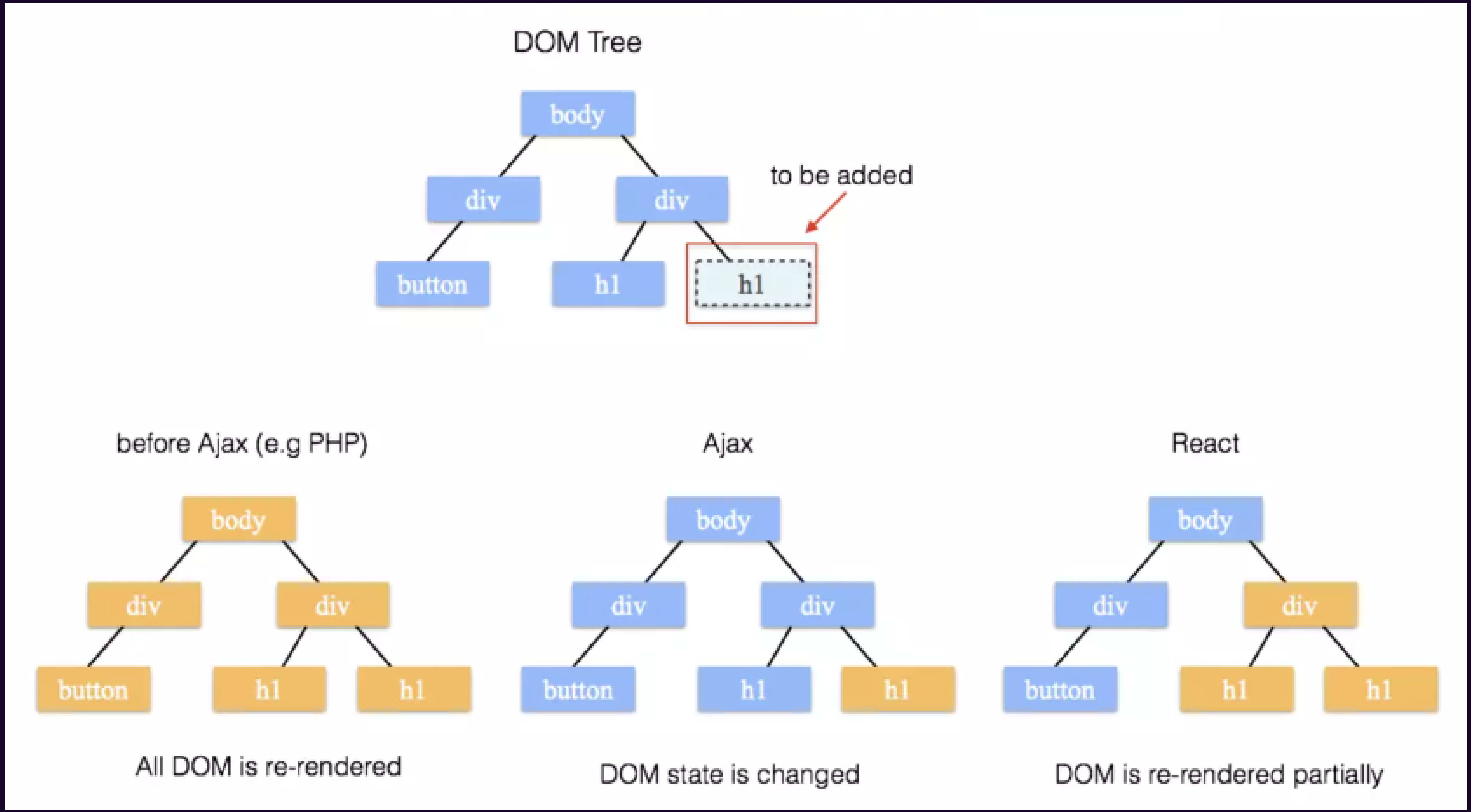


```
return (
  <li className="item">
    {name} {isPacked && '✔'}
  </li>
);
```

You can read this as “if isPacked, then (&&) render the checkmark, otherwise, render nothing”.

Updating a rendered element

React elements are immutable. Thus the only way to update a UI is to re-render the components entirely. React achieves this by creating a virtual-dom tree.



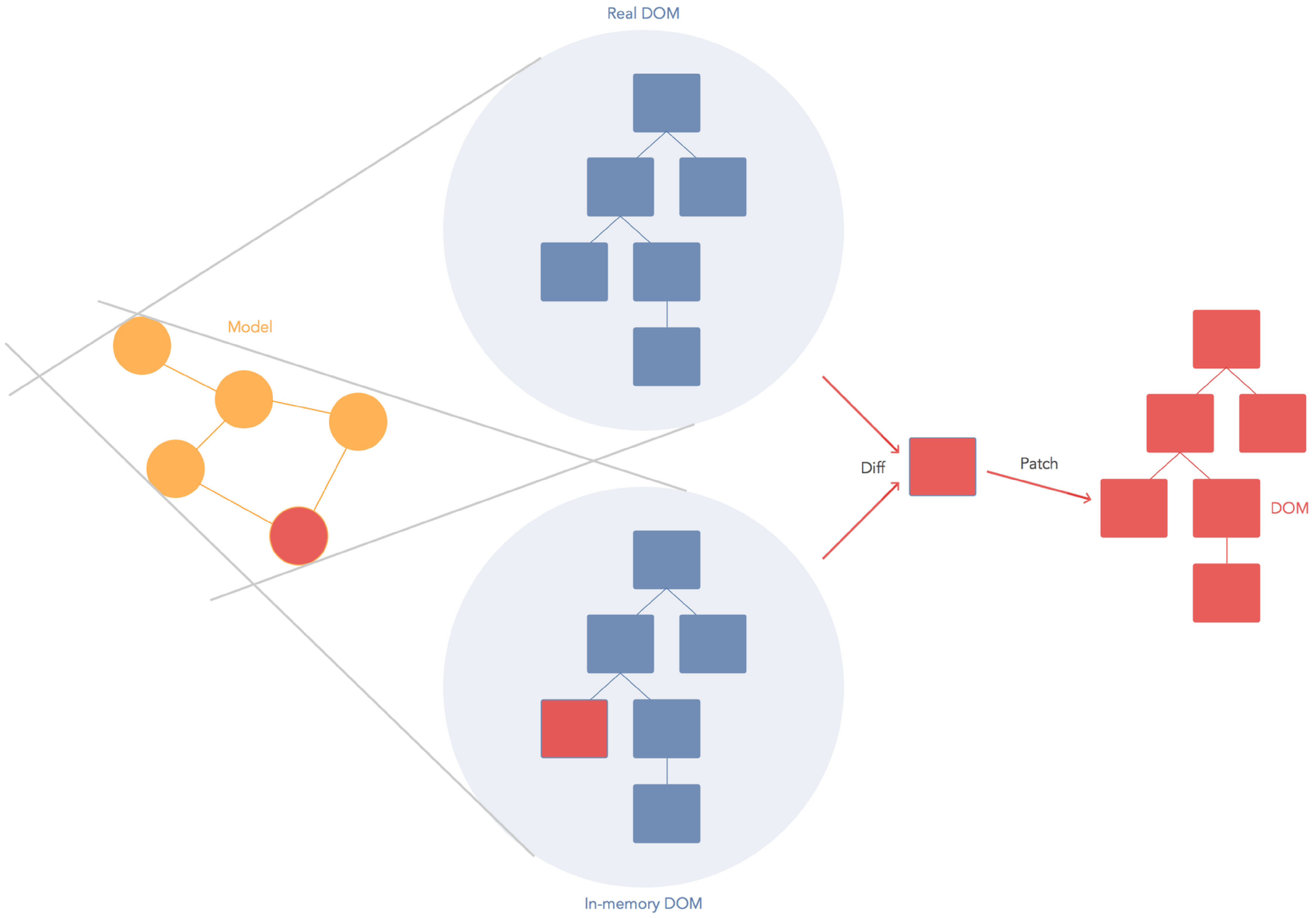
React virtual dom tree

React only updates what's necessary

React only updates DOM elements that have changed. This means if we re-render a component twice in a row, the second render will do nothing.

React uses a heuristic diffing algorithm to find out portions of the virtual dom that has been updated, which has the complexity of O(n), while state-of-the-art algorithms have O(n^3) complexity.

Read more: https://legacy.reactjs.org/docs/reconciliation.html


Real DOM
Model
Diff
Patch
DOM
In-memory DOM

States in React

React provides a declarative way to manipulate the UI. Instead of manipulating individual pieces of the UI directly, you describe the different states that your component can be in, and switch between them in response to the user input.

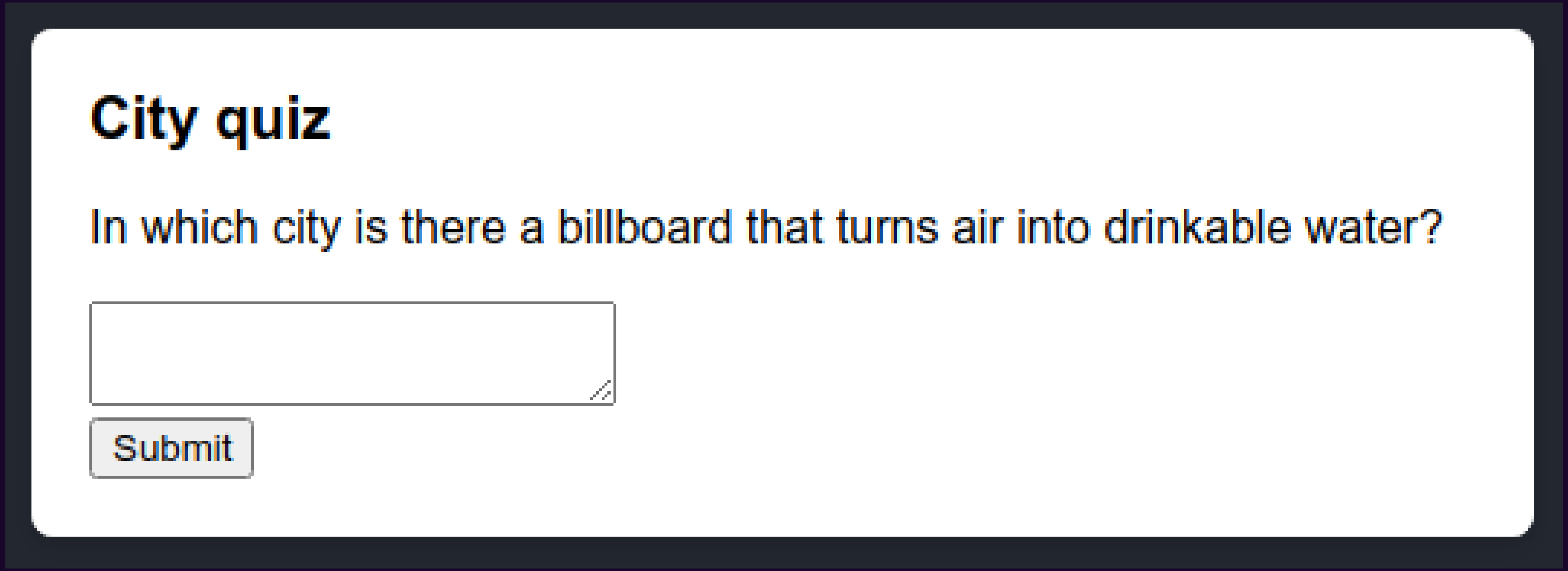
How to create UI with states in React

1. Identify your component’s different visual states
2. Determine what triggers those state changes
3. Represent the state in memory using useState
4. Remove any non-essential state variables
5. Connect the event handlers to set the state

Step 1: Identify your component’s different visual states

First, you need to visualize all the different “states” of the UI the user might see:

- **Empty**: Form has a disabled “Submit” button.
- **Typing**: Form has an enabled “Submit” button.
- **Submitting**: Form is completely disabled. Spinner is shown.
- **Success**: “Thank you” message is shown instead of a form.
- **Error**: Same as Typing state, but with an extra error message.



City quiz

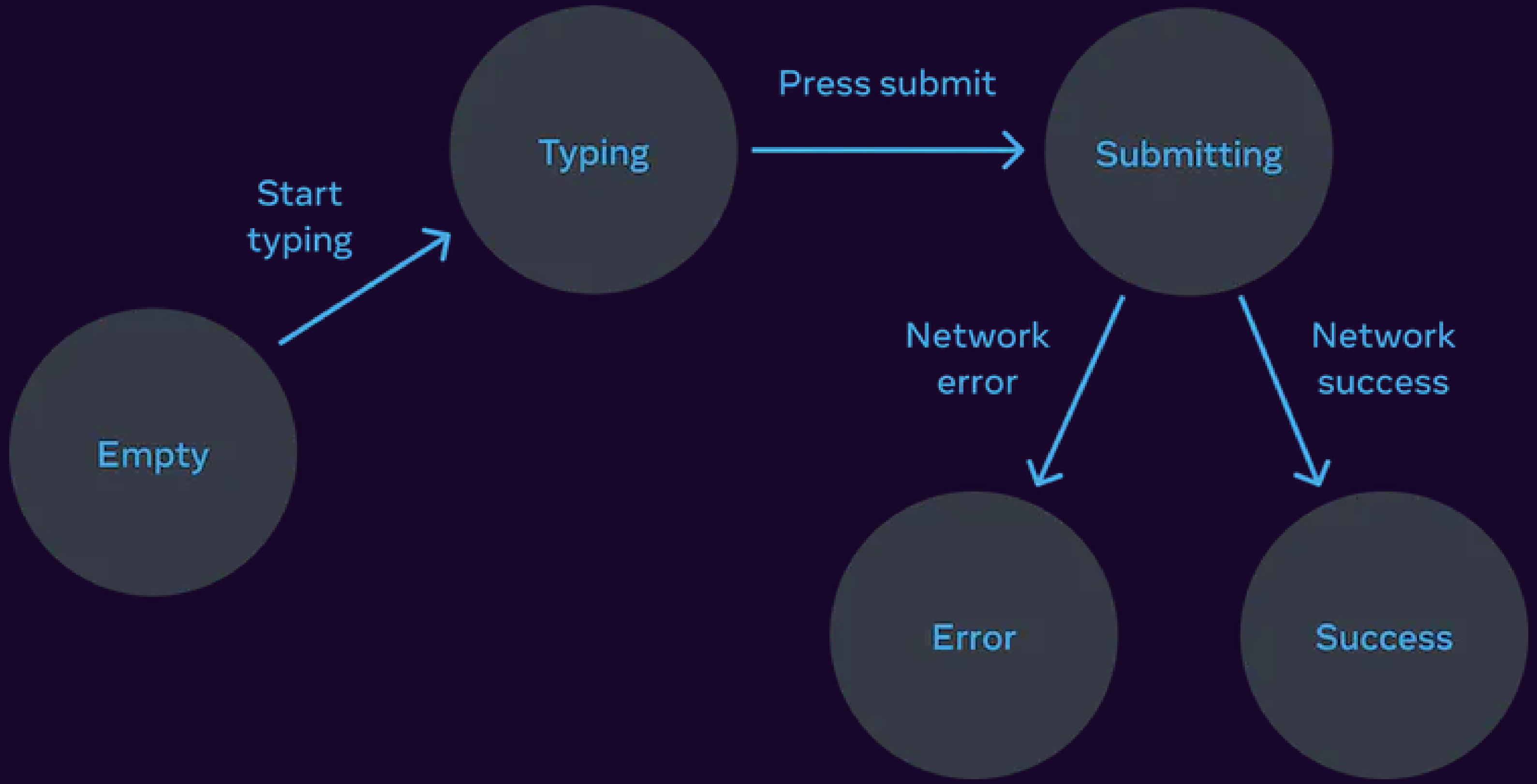
In which city is there a billboard that turns air into drinkable water?

Step 2: Determine what triggers those state changes

You can trigger state updates in response to two kinds of inputs:

- **Human inputs**, like clicking a button, typing in a field, navigating a link.
- **Computer inputs**, like a network response arriving, a timeout completing, an image loading.

To help visualize this flow, try drawing each state on paper as a labeled circle, and each change between two states as an arrow. You can sketch out many flows this way and sort out bugs long before implementation.



Step 3: Represent the state in memory with useState

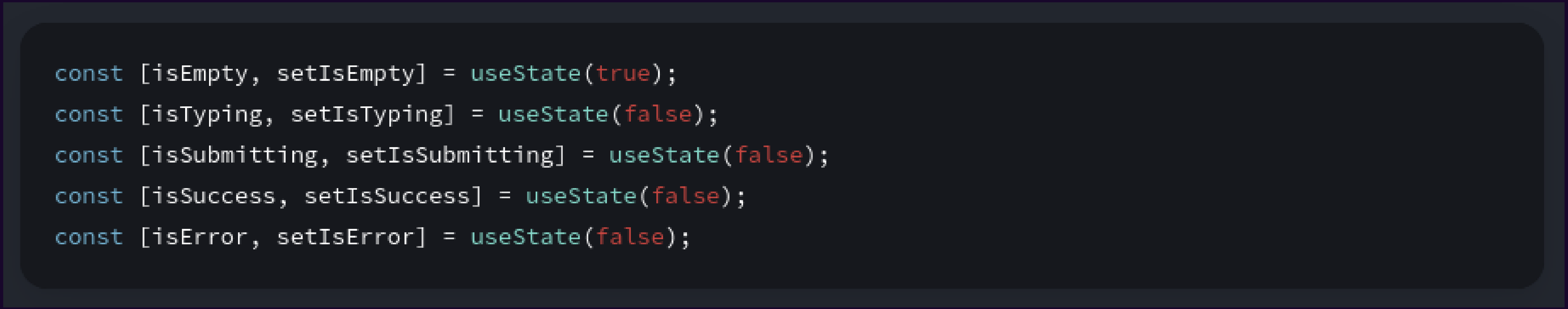
Next you’ll need to represent the visual states of your component in memory with useState. Simplicity is key: each piece of state is a “moving piece”, and you want as few “moving pieces” as possible. More complexity leads to more bugs!

Start with the state that absolutely must be there. For example, you’ll need to store the answer for the input, and the error (if it exists) to store the last error:

```
const [answer, setAnswer] = useState('');
const [error, setError] = useState(null);
```

Then, you’ll need a state variable representing which one of the visual states that you want to display. There’s usually more than a single way to represent that in memory, so you’ll need to experiment with it.

If you struggle to think of the best way immediately, start by adding enough state that you're definitely sure that all the possible visual states are covered:



```
const [isEmpty, setIsEmpty] = useState(true);
const [isTyping, setIsTyping] = useState(false);
const [isSubmitting, setIsSubmitting] = useState(false);
const [isSuccess, setIsSuccess] = useState(false);
const [isError, setIsError] = useState(false);
```

Your first idea likely won't be the best, but that's ok—refactoring state is a part of the process!

Step 4: Remove any non-essential state variables

You want to avoid duplication in the state content so you're only tracking what is essential. Spending a little time on refactoring your state structure will make your components easier to understand, reduce duplication, and avoid unintended meanings.

Here are some questions you can ask about your state variables:

Does this state cause a paradox? For example, *isTyping* and *isSubmitting* can't both be true. A paradox usually means that the state is not constrained enough. There are four possible combinations of two booleans, but only three correspond to valid states. To remove the "impossible" state, you can combine these into a status that must be one of three values: 'typing', 'submitting', or 'success'.

- **Is the same information available in another state variable already?** Another paradox: isEmpty and isTyping can’t be true at the same time. By making them separate state variables, you risk them going out of sync and causing bugs. Fortunately, you can remove isEmpty and instead check answer.length === 0.

- **Can you get the same information from the inverse of another state variable?** isError is not needed because you can check error !== null instead.

After this clean-up, you’re left with 3 (down from 7!) essential state variables:

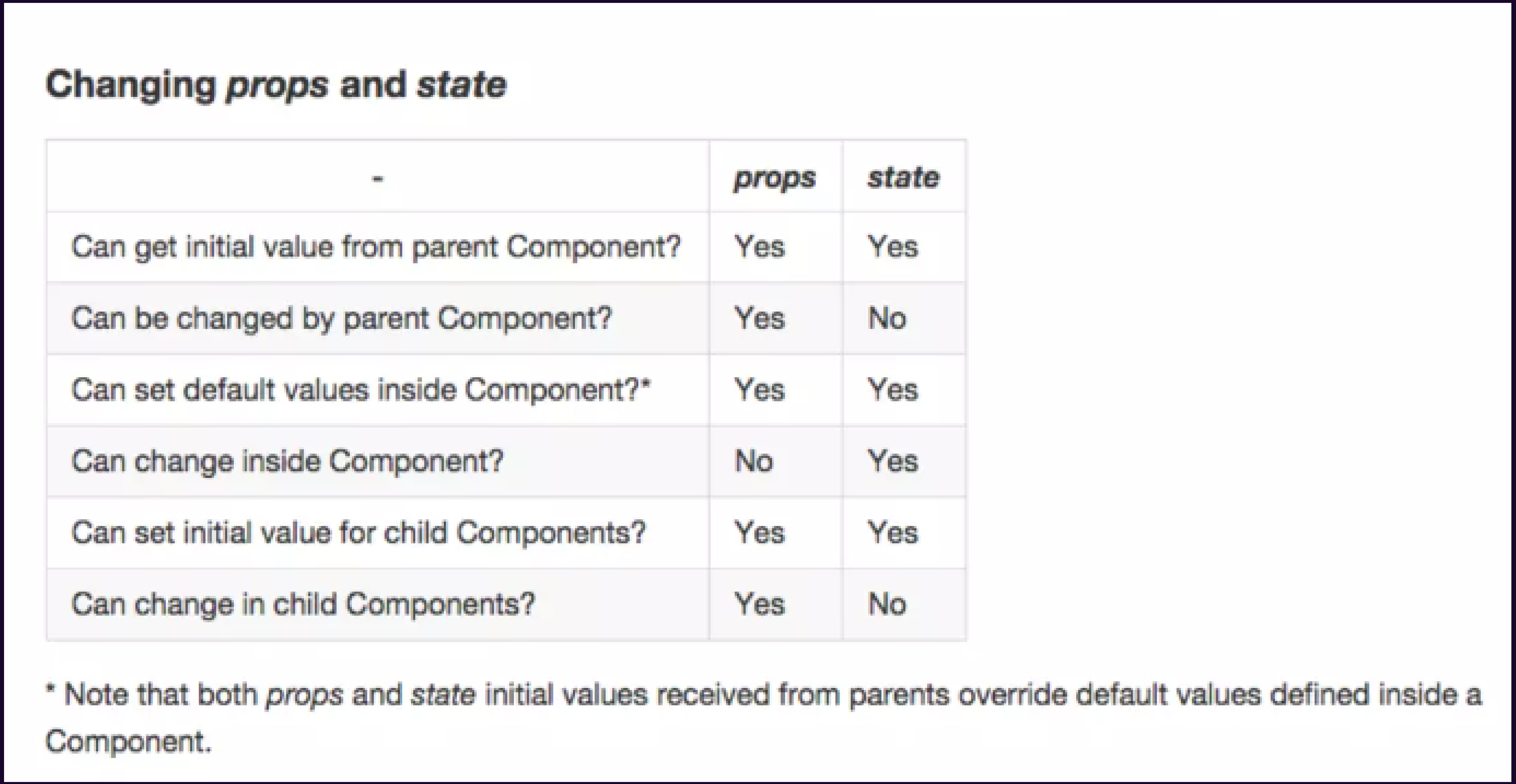
```
const [answer, setAnswer] = useState('');
const [error, setError] = useState(null);
const [status, setStatus] = useState('typing'); // 'typing', 'submitting', or 'success'
```

Step 5: Connect the event handlers to set state

Some suggestions regarding states

- **Group related state.** If you always update two or more state variables at the same time, consider merging them into a single state variable.
- **Avoid contradictions in state.** When the state is structured in a way that several pieces of state may contradict and “disagree” with each other, you leave room for mistakes. Try to avoid this.
- **Avoid redundant state.** If you can calculate some information from the component’s props or its existing state variables during rendering, you should not put that information into that component’s state.
- **Avoid duplication in state.** When the same data is duplicated between multiple state variables, or within nested objects, it is difficult to keep them in sync. Reduce duplication when you can.
- **Avoid deeply nested state.** Deeply hierarchical state is not very convenient to update. When possible, prefer to structure state in a flat way.

Props vs state



Changing *props* and *state*

-	*props*	*state*
Can get initial value from parent Component?	Yes	Yes
Can be changed by parent Component?	Yes	No
Can set default values inside Component?*	Yes	Yes
Can change inside Component?	No	Yes
Can set initial value for child Components?	Yes	Yes
Can change in child Components?	Yes	No

* Note that both *props* and *state* initial values received from parents override default values defined inside a Component.

Live Demo

Simple TODO App with CRUD functionality