

Session 2 — Building Game Components

You're about to build your first custom React component and unlock the power of reusable UI building blocks — the secret to fast, scalable development in React. This guide walks you through creating a `GameButton` component, understanding props, and using React developer tools. Ready to build your first component? Let's go!

Table of Contents

- [Understanding React's Approach](#)
- [Creating Your First Component](#)
- [Understanding Props](#)
- [Adding Click Functionality](#)
- [Styling with Variants](#)
- [Reusing Your Component](#)
- [Installing React DevTools](#)
- [Essential Terms](#)
- [Ask the AI](#)

Accessing Your Codespace

Visit github.com/codespaces to relaunch your Codespace from Session 1.

Understanding React's Approach

Why did swapping `<StartHere />` for `<SplashScreen />` feel so effortless? It's all about React's approach to building UIs.

With vanilla JavaScript, you write lots of repetitive code to update the page. React works differently: you build self-contained components, and React handles all the messy details of getting them on screen and keeping them updated.

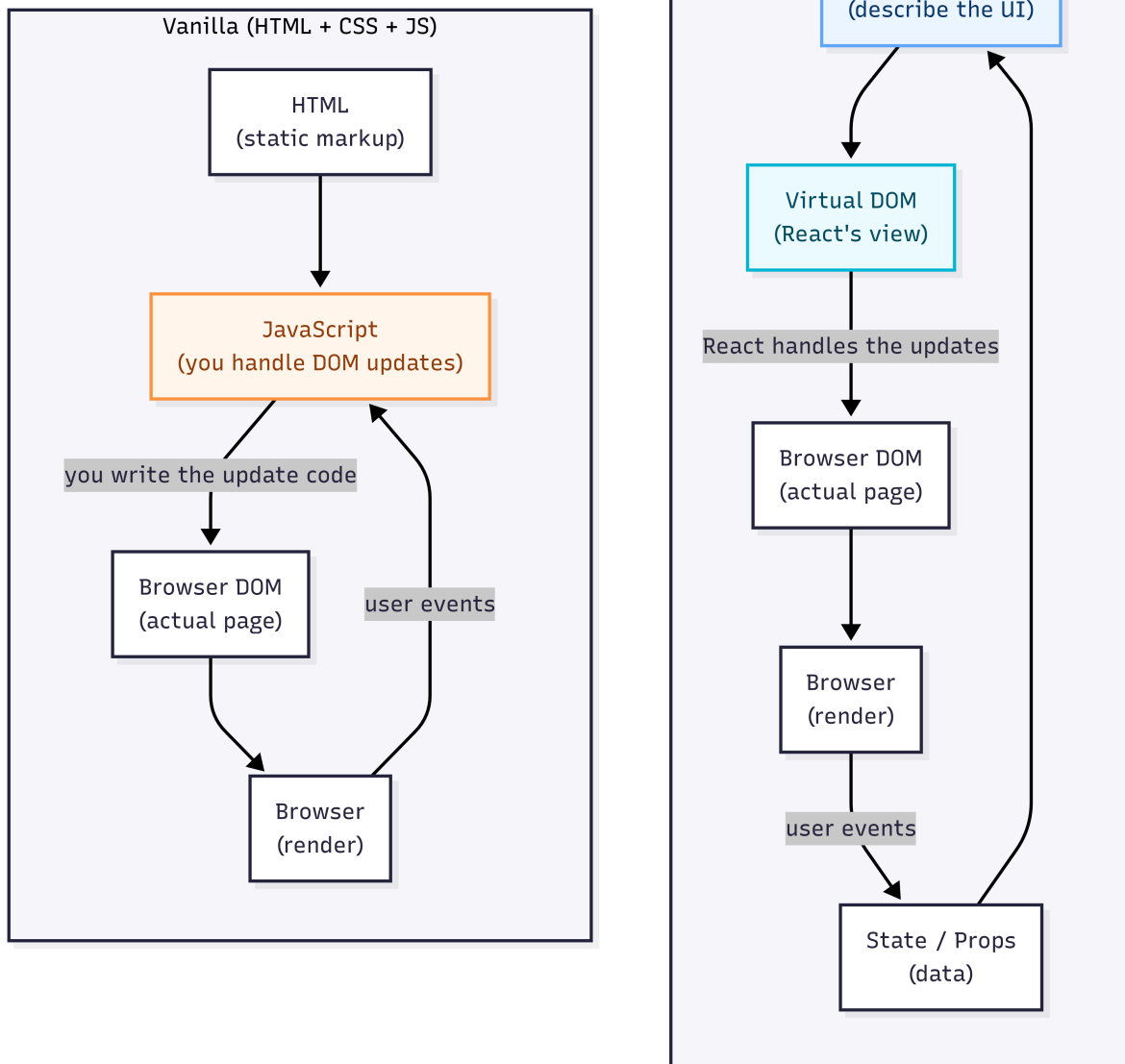



Figure: Vanilla JavaScript vs React — Why Components Make Development Easier

That's why swapping components felt so smooth. You weren't just editing code—you were shaping the UI with reusable building blocks.

Now let's build your first custom component and see that power in action.

Creating Your First Component

 **Goal:** Build a reusable GameButton component and learn how to export, import, and use custom components.

Step 1: Create the component file

Right-click `src/components` → New File → name it `GameButton.jsx`

Step 2: Write the component structure

Create the basic component function that returns a button element, establishing the foundation for your reusable GameButton.

File: `src/components/GameButton.jsx`

```
export default function GameButton() {  
  return <button>Start Adventure</button>;  
}
```

Step 3: Import and use the component

Import your new GameButton component into SplashScreen and add it to the JSX to see it render on the page.

File: `src/components/SplashScreen.jsx`

```
import GameLogo from "../GameLogo";  
+import GameButton from "../GameButton";  
  
export default function SplashScreen() {  
  return (  
    <div className="splash-screen">  
      <GameLogo />  
-    <div className="splash-buttons"></div>  
+    <div className="splash-buttons">  
+      <GameButton />  
+    </div>  
    </div>  
  );  
}
```

Step 4: Test your component

Run `npm run dev` if not already running.

✓ You should see: Your custom button appears on the splash screen!

💡 Export and Import Pattern

Components are the heart of React — reusable UI elements that combine markup, styling, and logic. Think of them as your own custom HTML tags. The `.jsx` file extension means you're writing JSX, a special syntax that looks like HTML but is actually JavaScript. When you create a component, export it with `export default` so it can be shared across your project. Then bring it into other files with `import`.

🏆 Bonus Challenge

Try changing the button text in `GameButton.jsx` and watch it update instantly thanks to Hot Module Replacement!

📦 Understanding Props

🎯 **Goal:** Make your `GameButton` flexible by accepting custom text through props.

Step 1: Add text prop to `GameButton`

Modify the component to accept a `text` prop using destructuring, replacing the hardcoded button text with a dynamic value.

File: `src/components/GameButton.jsx`

```
// Add text parameter in curly braces
export default function GameButton({ text }) {
  // Replace hardcoded text with {text}
  return <button>{text}</button>;
}
```

Step 2: Pass the text prop from `SplashScreen`

Provide the button text from the parent component by passing it as a prop, demonstrating how data flows from parent to child.

File: `src/components/SplashScreen.jsx`

```
export default function SplashScreen() {
  return (
    <div className="splash-screen">
      <GameLogo />
      <div className="splash-buttons">
        <GameButton text="Start Adventure" />
        {/* ↑ Add text prop with value */}
      </div>
    </div>
  );
}
```

✓ You should see: Your button now shows custom text!

💡 Parent-to-Child Data Flow

Props let parent components pass data to child components — just like function parameters. This makes your components flexible and reusable. The `{ text }` syntax is called **destructuring** — it pulls out just the values you need from the **props** object, keeping your code clean and readable.

🖱️ Adding Click Functionality

🎯 **Goal:** Make your button interactive by adding click handlers through props.

Step 1: Add `onClick` prop to `GameButton`

Extend the component to accept an `onClick` function prop and attach it to the button element, enabling interactive behavior.

File: `src/components/GameButton.jsx`

```
// Add onClick parameter
export default function GameButton({ text, onClick }) {
  // Add onClick to button element
  return <button onClick={onClick}>{text}</button>;
}
```

Step 2: Pass click handler from SplashScreen

Provide a function that will execute when the button is clicked, using an arrow function to display an alert.

File: `src/components/SplashScreen.jsx`

```
<div className="splash-buttons">
  <GameButton
    text="Start Adventure"
    onClick={() => alert('Start Game!')}
  />
  { /* ↑ Add onClick prop */ }
</div>
```

Step 3: Test the GameButton

Click “Start Adventure” on your splash screen.

✓ You should see: A browser alert with the message “Start Game!” appears!

Giving Components Different Behaviors

Functions as props are like giving your components different personalities. Your `GameButton` can do different things depending on where you use it — same button, different actions. It’s a key pattern in React for building interactive apps.

Styling with Variants

 **Goal:** Add visual variety to your buttons using CSS classes and default parameters.

Step 1: Add variant prop and dynamic className

Add a `variant` prop with a default value to control button styling, then create a dynamic `className` that combines the base class with the variant.

File: `src/components/GameButton.jsx`

```
// Add variant parameter with default value
export default function GameButton({ text, onClick, variant = "primary" }) {
  // Add this line: create buttonClass variable
  const buttonClass = `game-button ${variant}`;

  return (
    <button className={buttonClass} onClick={onClick}>
      {/* ↑ Update to use className */}
      {text}
    </button>
  );
}
```

Step 2: Use the variant prop in SplashScreen

Pass the `variant` prop to specify which button style to use, demonstrating how props control component appearance.

File: `src/components/SplashScreen.jsx`


```
<div className="splash-buttons">
  <GameButton
    text="Start Adventure"
    onClick={() => alert('Start Game!')}
    variant="primary"
  />
  {/* ↑ Add variant prop */}
</div>
```

✓ You should see: Your button now has the primary styling with a vibrant color!

💡 Building Dynamic Class Names

`className` is React's version of the HTML `class` attribute. We use a **template literal** to build a dynamic class name like `game-button primary`. This matches the styles already defined in your project. The `variant` prop lets you switch between styles like `primary` and `secondary`, and **default parameters** like `variant = "primary"` ensure your component still works even if no variant is passed.

Reusing Your Component

 **Goal:** Experience the power of component reusability by adding a second button with different props.

File: `src/components/SplashScreen.jsx`

Add a second `GameButton` with different prop values to demonstrate how the same component can be reused with different configurations.

```
<div className="splash-buttons">
  <GameButton
    text="Start Adventure"
    onClick={() => alert('Start Game!')}
    variant="primary"
  />
  { /* ↑ Existing button */ }

  <GameButton
    text="Credits"
    onClick={() => alert('Show Credits')}
    variant="secondary"
  />
  { /* ↑ Add this button */ }
</div>
```

✓ **You should see:** Two different buttons using the same component!

Write Once, Use Everywhere

Component reusability is React's superpower. You wrote the `GameButton` code once, but now you can use it anywhere in your app with different props. Thanks to your stylesheet, each variant (`primary`, `secondary`) automatically applies the right look — no extra styling needed.

Bonus Challenge

Try adding a third `GameButton` with `variant="primary"` and `text="Instructions"` to see how easy it is to scale your UI!

Installing React DevTools

 **Goal:** Install and explore React DevTools to inspect your components and props.

Step 1: Install the browser extension

Choose your browser and install React DevTools:

Browser	Installation Link	Notes
Chrome	Chrome Web Store	Most popular choice
Firefox	Firefox Add-ons	Great alternative
Edge	Edge Add-ons	Windows default
Safari	Manual installation required	Advanced users only

Step 2: Open and explore DevTools

1. Press `F12` or right-click → Inspect
 2. Find the Components tab (next to Console, Network, etc.)
 3. Click on components in the tree to see their props
 4. Find your GameButton component and inspect its props
- ✓ **You should see:** The text, onClick, and variant props displayed in the DevTools panel!

Real-Time Component Inspection

React DevTools gives you X-ray vision into your app. You can inspect components, props, and state in real time — essential for debugging and understanding how your app works under the hood.

Essential Terms

Quick reference for all the React concepts you just learned:

Term	Definition	Why it matters
 props	Data passed from parent to child components.	Props let you customize components and pass data around your app — essential for reusable components.
 className	React's version of the HTML <code>class</code> attribute for applying CSS styles.	Use <code>className</code> instead of <code>class</code> because <code>class</code> is a reserved word in JavaScript.
 destructuring	Extracting values from objects/arrays into variables, like <code>{ text, onClick }</code> from props.	Makes your code cleaner by avoiding repetitive <code>props.text</code> , <code>props.onClick</code> syntax.
 template literal	String interpolation using backticks and <code>\${}</code> for dynamic strings.	Perfect for creating dynamic CSS classes like <code>`game-button \${variant}`</code> .
 default parameters	Fallback values for function parameters, like <code>variant = "primary"</code> .	Ensures your components work even when some props aren't provided.
 React DevTools	Browser extension for inspecting React component trees, props, and state.	Essential debugging tool — like X-ray vision for your React app.

Ask the AI — Building Game Components

You just created your first reusable React component with props, styling, and click handlers — excellent work!

Now let's deepen your understanding of components, props, and the React development workflow. Here are the most impactful questions to ask your AI assistant about today's

session:

- What makes React components reusable and why is that important?
- How do props work in React and why are they read-only?
- How do template literals work and why are they perfect for dynamic CSS classes?
- What is interpolation in JSX and can you show me examples?
- How does JSX let me write HTML-like code inside JavaScript?
- Can I pass functions as props? How does that work and why is it powerful?
- What can I do with React DevTools that I can't do with regular browser DevTools?