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

☁️ 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.

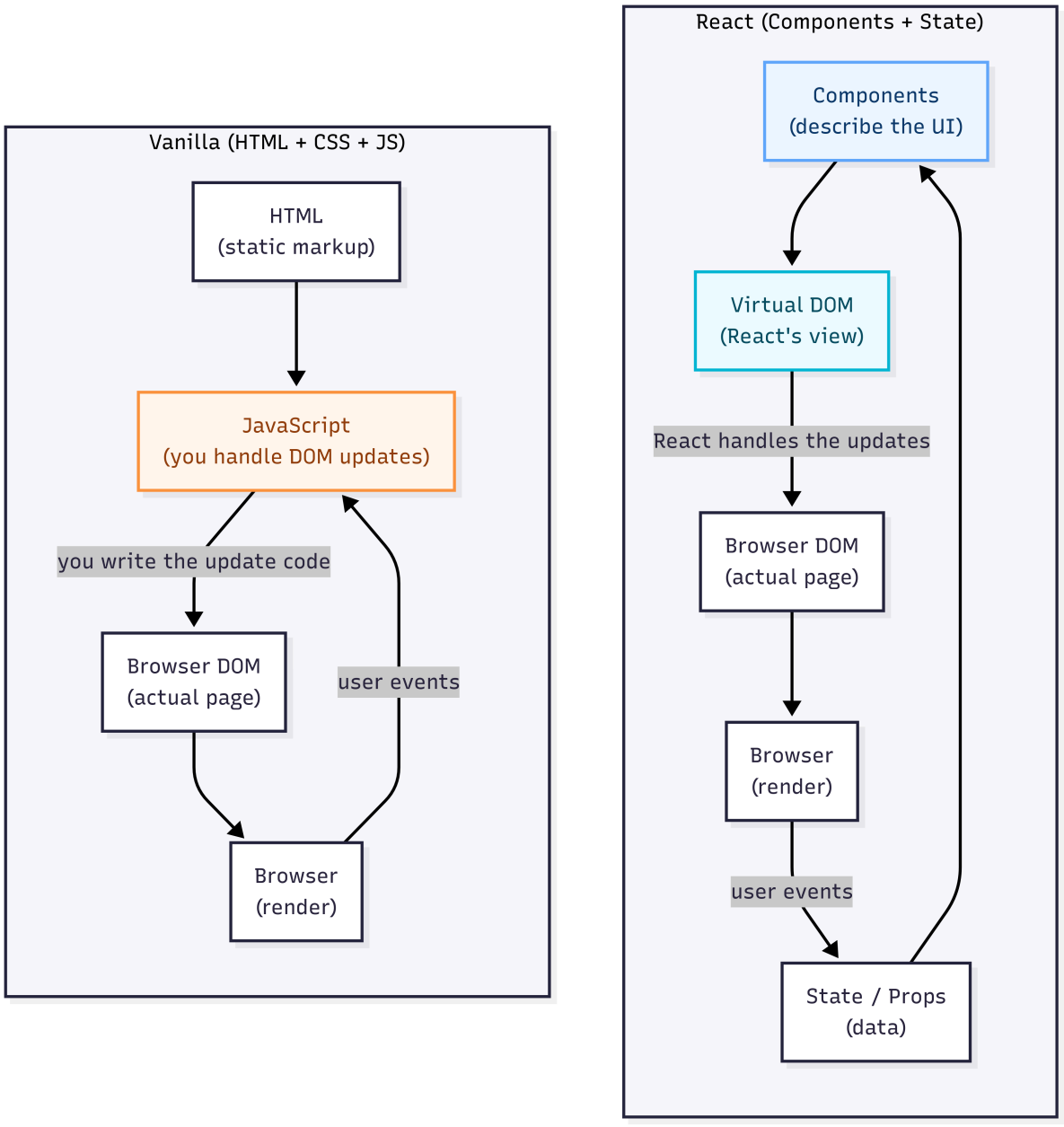

Figure: Vanilla JavaScript vs React — Why Components Make Development Easier

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.

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>
  );
}
```

> 💡 Understanding Diff Syntax
>
> One common way to show code changes is using a diff format. Lines starting with `+` are additions, while lines starting with `-` are deletions. Lines without symbols remain unchanged.

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

Make your button display custom text by accepting a `text` prop and using it in place of the hardcoded text.

File: `src/components/GameButton.jsx`

```
// [1] Add text parameter
export default function GameButton({ text }) {
  // [2] Use text in JSX
  return <button>{text}</button>;
}
```

> 💡 **Understanding Props and Destructuring**
>
> 1. **Add text parameter**: Destructure `text` from the props object in the function signature
> 2. **Use text in JSX**: Display the dynamic value with `{text}` instead of hardcoded text
>
> Destructuring `{ text }` is shorthand for `const text = props.text`. It pulls the value directly from the props object, making your code cleaner and more readable.

Step 2: Pass text prop from SplashScreen

Pass the button text from the parent component as a prop.

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. SplashScreen (the parent) passes `text="Start Adventure"` down to GameButton (the child), which receives it and displays it.

🖱️ Adding Click Functionality

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

Step 1: Add onClick prop to GameButton

Make your button interactive by accepting an `onClick` prop and attaching it to the button element.

File: `src/components/GameButton.jsx`

```
// [1] Add onClick parameter
export default function GameButton({ text, onClick }) {
  // [2] Attach to button element
  return <button onClick={onClick}>{text}</button>;
}
```

> 💡 Understanding Click Handlers
>
> 1. **Add onClick parameter**: Destructure `onClick` from props alongside `text`
> 2. **Attach to button element**: Pass `onClick` to the button's `onClick` attribute
>
> Functions can be passed as props just like any other data. When the button is clicked, React calls the function you passed from the parent component, enabling interactive behavior.

Step 2: Pass click handler from SplashScreen

Pass a function that will execute when the button is clicked.

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.

> 💡 **Functions as Props**
>
> Passing functions as props lets you define custom behavior for components. When you click the button, the `onClick` function you provided from SplashScreen is executed, showing the alert. This pattern allows components to be highly flexible and interactive.

🎨 Styling with Variants

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

Step 1: Add variant styling to GameButton

Add a `variant` parameter with a default value, create a variable that combines the base class with the variant, then update the button to use this dynamic className.

File: `src/components/GameButton.jsx`

```
// [1] Add variant parameter with default value
export default function GameButton({ text, onClick, variant = "primary" }) {
  // [2] Create buttonClass variable
  const buttonClass = `game-button ${variant}`;

  return (
    <button className={buttonClass} onClick={onClick}>
      {text}
    </button>
    {/* ↑ [3] Update to use className */}
  );
}
```

> 💡 Understanding Dynamic Styling
>
> 1. **Add variant parameter**: Accept `variant` prop with default value `"primary"`
> 2. **Create buttonClass variable**: Use template literal to combine base class with variant
> 3. **Update to use className**: Apply the dynamic class to the button element
>
> **className** is React's version of the HTML `class` attribute (since `class` is a reserved word in JavaScript). Template literals (backticks with `${}`) let you build strings dynamically — here we combine `game-button` with the variant to create class names like `game-button primary`. The `variant` prop lets you switch between styles, and the default parameter ensures the component works even without explicitly passing a variant.

Step 2: Pass variant prop from SplashScreen

Pass the `variant` prop to specify which button style to use.

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!

🔄 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.

```jsx
<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 credits 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.

🔍 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 `class` attribute for applying CSS styles.	Use `className` instead of `class` because `class` is a reserved word in JavaScript.

📤 destructuring	Extracting values from objects/arrays into variables, like `{ text, onClick }` from props.	Makes your code cleaner by avoiding repetitive `props.text`, `props.onClick` syntax.
🔤 template literal	String interpolation using backticks and `${}` for dynamic strings.	Perfect for creating dynamic CSS classes like `` `game-button ${variant}` ``.
⚙️ default parameters	Fallback values for function parameters, like `variant = "primary"`.	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?**