

Components & JSX

The React Native mantra: Everything is a component.

Components

The building blocks of any React Native application.

What is a component?

At its core, a **component** is a function that returns UI.

Rather than writing HTML, you write functions that return **JSX**.

Every button, card, header, modal — everything — is a component.

Example Component:

```
function App() {  
  return (  
    <Text>Hello!</Text>  
  );  
}
```

The returned JSX is wrapped in parentheses, and the function itself is just plain JavaScript.

Once you understand components, everything else falls into place; because **state**, **hooks**, **lists**, and **navigation** all live inside components.

JSX is *not* HTML. Though it resembles HTML, JSX gets transformed depending on the platform. A `<Text>` element may render as a native iOS UILabel, an Android TextView, or `<p>` on web.

Example

We can build entire screens using this pattern:

- Make a function
- Return JSX
- Put styling inside the JSX
- Combine components like LEGO bricks

This is how React Native apps grow.

```
function WelcomeCard() {
  return (
    <View style={{ padding: 20, backgroundColor: "black" }}>
      <Text style={{ color: "white" }}>Welcome!</Text>
      <Text style={{ color: "gray" }}>Glad you're here.</Text>
    </View>
  );
}
```

Props & Reusable Components

Props are inputs to a component.

They allow you to pass data *into* a component so it can behave differently depending on what it receives.

Props are just **function parameters**.

Example in plain JavaScript:

```
function greet(name) {
  console.log("Hello " + name);
}

greet("HotelDangerous");
```

Components are identical in spirit:

```
function Greeting(props) {
  return <Text>Hello {props.name}</Text>;
}

<Greeting name="HotelDangerous" />;
```

React passes `{ name: "HotelDangerous" }` into your component as `props`.

Why are Props Important?

Props let you reuse the same component in many situations.

Imagine we want a pink button to use throughout the app. Rather than rewriting the same boilerplate every time, we create a single component and give it a prop for the text.

```
function PinkButton(props) {  
  return (  
    <TouchableOpacity style={{ backgroundColor: "#C80085", padding: 10 }}>  
      <Text>{props.text}</Text>  
    </TouchableOpacity>  
  );  
}  
  
<PinkButton text="Save" />  
<PinkButton text="Delete" />  
<PinkButton text="Add Medicine" />
```

Same component, different text. This is the core of component reuse.

Deconstructing Props

You'll often see props **destructured** to make the code cleaner.

```
function PinkButton({ text }) {  
  return (  
    <TouchableOpacity style={{ backgroundColor: "#C80085", padding: 10 }}>  
      <Text>{text}</Text>  
    </TouchableOpacity>  
  );  
}  
  
<PinkButton text="Save" />  
<PinkButton text="Delete" />  
<PinkButton text="Add Medicine" />
```

Instead of writing `props.text`, we extract the `text` property directly:

`props = { text: "..." } → function PinkButton({ text })`.

Props are read-only.

You cannot modify props inside a component. Use state for that.

State (`useState`)

We know that components return UI, props pass data *into* components, and that props cannot be changed within the component. Now we cover the flip side. **State** is data inside a component that React *remembers*. Similar to props, how our app behaves may -and probably does- depend on the *current state*.

What is State?

State is data that belongs to a component and can change over time. While props are data passed from outside a component, state is “memory” inside a component. State allows our apps to be dynamic. With it we can track user input, expand and collapse sections, count clicks, update lists, toggle buttons, and much more.

The `useState` Hook

This is the most important line of code in React:

```
const [value, setValue] = useState(initialValue);
```

Where:

- `value` is the current state
- `setValue` is a function that updates the state
- `useState()` is a hook that creates component memory
- `initialValue` is the starting value

For example, if we have a counter and want to start at zero:

```
const [count, setCount] = useState(0);
```

React now remembers the number for this component.

In React, you **never** update state directly. You will never write:

```
count = count + 1;
```

Because React must know when the value changes so it can re-render the UI. Instead, we **always** write:

```
setCount(count + 1);
```

This notifies React that the component should re-render with new data.

Example: Incrementing a Count When a Button Is Pressed

```
function Counter() {  
  const [count, setCount] = useState(0);  
  return (  
    <View>  
      <Text>{count}</Text>  
      <Button title="Increment" onPress={() => setCount(count + 1)} />  
    </View>  
  );  
}
```

1. User presses the button
 2. `setCount(count + 1)` runs
 3. React re-renders
 4. UI updates and shows the new count
-

What Are Hooks?

Hooks are special React functions that let components store values, run code at specific times, and respond to changes. They allow React to remember data across re-renders without you manually tracking anything.

When state updates, React uses hooks to recall previous values, compute the new UI, and update the screen automatically.

Events and Interactivity

Previously, we learned about *state*. We know that state is data that is remembered by a component and we know that the component re-renders when the state changes. But what causes the state to change? **Events**.

What is an Event?

An **event** occurs whenever the user interacts with a component. For example: the user taps a button, types input, opens or closes a modal, swipes, scrolls, selects a time, or

toggles a switch. Interactive components have an *event handler* which runs a function whenever an event happens. The most common event is `onPress`.

When the user taps a button, React calls the function given to that component. For example, a button that prints "Press me" whenever you tap it:

```
<TouchableOpacity onPress={() => console.log("Press me")}>
  <Text>Tap me!</Text>
</TouchableOpacity>
```

Interaction = State + Events

When a user presses a button that fires an event, an **interaction** has occurred. For each interaction, the following steps happen:

1. User triggers an event
2. Event calls a function
3. Function updates the state
4. React re-renders the component

Note:

Arrow functions are common in React and look like:

```
onPress={() => setCount(count + 1)}
```

These functions allow us to run code *later*—such as when an event occurs—rather than immediately when the component loads.

Events Can Receive Arguments

Suppose we plan to have many pink buttons in our app, each with different text and behavior. We can create a reusable UI component that accepts both a label and a function as arguments:

```
// Pink Button Style
const pinkButtonStyle = {
  backgroundColor: "#C80085",
  padding: 12,
  borderRadius: 10,
  alignItems: "center",
  justifyContent: "center"
};
```

```
// Pink Button Component
function PinkButton({ label, onPress }) {
  return (
    <TouchableOpacity onPress={onPress} style={pinkButtonStyle}>
      <Text>{label}</Text>
    </TouchableOpacity>
  );
}

// ...Using it later
<PinkButton label="Save" onPress={() => setSave(true)} />
```

The `style` details are not important right now.

Controlled Inputs

A **controlled input** is a text field whose value is *fully* controlled by React through state. This means the text the user sees **comes from state only**. Every time the user types:

1. The input fires an event
2. State updates
3. React re-renders
4. The input displays the updated state

```
const [name, setName] = useState("");

<TextInput
  value={name}
  onChangeText={newValue => setName(newValue)}
/>
```

Without controlled inputs, we would not be able to read what the user typed, validate it, submit forms, reset fields, or build flows like “Add Medicine.” Controlled inputs are essential.

How `TextInput` Works

Returning to the example above:

- `value` represents the current text from state
- `onChangeText` receives the latest text the user typed

- setName(newValue) updates state
- React re-renders and the value updates the visible text

We don't read the input directly-- **we read the state that mirrors the input.**

Example: Controlled Username Input

Below is a corrected and fully working version of the component.
Your original snippet had a few issues:

- <textInput> should be <TextInput>
- The placeholder had a typo
- onChangeText must accept the new text
- You displayed "username" literally instead of using {username}

Here is the fixed version:

```
function UsernameInput() {
  const [username, setUsername] = useState("");

  return (
    <View>
      <TextInput
        value={username}
        onChangeText={newValue => setUsername(newValue)}
        placeholder="Enter username"
      />
      <Text>Current username: {username}</Text>
    </View>
  );
}
```

This is the standard controlled-input pattern in React Native.