# React Crash Course: From Vanilla JavaScript to Component-Based Development

This crash course will guide you through React's core concepts using the chatbot example, showing how React transforms DOM manipulation into declarative, component-based development.

# Why React?


Jobsdb

**Jobsdb – Hong Kong's no. 1 jobs, employment, career and recruitment site**

Search, browse and apply the latest Admin & HR, Banking, IT, Sales, Marketing and many other jobs in Hong Kong. Start your job search in Hong Kong at Jobsdb.com.

# Why React: Key Metrics & Popularity

React's standing in the web development ecosystem is exceptionally strong, as evidenced by its widespread adoption and community engagement. Here's how it compares to a major alternative:

| Metric | React | Angular |
| --- | --- | --- |
| Developer Usage | 40.6% | 17.1% |
| GitHub Stars | 213K+ | 90K+ |
| Weekly NPM Downloads | ~19M | ~2.8M |
| UK Job Openings | ~52,000 | ~23,000 |
| Developer Satisfaction | 62.2% | 53.4% |

These figures underscore React's dominance, making it a highly desirable skill for developers and a robust choice for building modern web applications.

# What is React?

React is a JavaScript library for building user interfaces. Instead of manually manipulating the DOM (like document.createElement or element.appendChild), React uses a **declarative approach** where you describe what the UI should look like, and React handles the updates.

### Vanilla JavaScript (Imperative):

```
// You tell the browser HOW to do something
const messageDiv = document.createElement('div');
messageDiv.textContent = 'Hello World';
messageDiv.className = 'message';
chatContainer.appendChild(messageDiv);
```

### React (Declarative):

```
// You tell React WHAT you want
<div className="message">Hello World</div>
```

Hello World

React automatically figures out how to update the DOM when your data changes.

# JSX: Writing HTML in JavaScript

JSX lets you write HTML-like syntax directly in JavaScript. It's not actually HTML—it gets compiled to JavaScript function calls.

## JSX Syntax Rules

```
// JSX looks like HTML but has some differences
function Greeting() {
  const name = "Alice";
  return (
    <div className="greeting">  {/* className instead of class */}
      <h1>Hello, {name}!</h1>
      <p>Welcome to React</p>
    </div>
  );
}
```

**Key Differences from HTML:**

- Use className instead of class
- Use {} to embed JavaScript expressions
- Self-closing tags must end with / (like `<input />`)
- All JSX must return a single parent element

## In Our Chatbot

```
 return (
<div className="chat-container">
<div className="chat-messages" ref={messagesRef}>
          {/* JavaScript expression to render messages */}
          {
            messages.map((message, index) => (
                <div
                  key={index}
                  className={`message ${message.sender === "user" ? "user message" : "bot-message"}`}
                >
                  {message.text}
                </div>
                    ))
           }
</div>
      {/* Rest of the component */}
</div>
);
```

# Components: Building Blocks of React

Components are reusable pieces of UI. Think of them as custom HTML elements that you create.

## Functional Components

```
function Chatbot() {
  // Component logic goes here
  return (
    // JSX goes here
  );
}
```

## Why Components?

### Reusability

Write once, use anywhere

### Organization

Keep related code together

### Maintainability

Easier to debug and update

## Component Hierarchy

You can nest components inside other components:

```
function Message({ text, sender }) {
return (
<div className={`message ${sender}-message`}>
        {text}
</div>
   );
}
function Chatbot() {
return (
<div className="chat-container">
<Message text="Hello!" sender="bot" />
<Message text="Hi there!" sender="user" />
</div>
   );
}
```

# State Management with useState

State is data that can change over time. In vanilla JavaScript, you might store this in variables. In React, you use the useState hook.

## Basic useState

```
const [count, setCount] = React.useState(0);
       ↑       ↑                   ↑
    current  function to          initial
    value    update value         value
```

## In Our Chatbot

```
function Chatbot() {
  // State for storing all chat messages
  const [messages, setMessages] = React.useState([
    { text: "Bot says: How can I help?", sender: "bot" }
  ]);

  // State for the current input value
  const [input, setInput] = React.useState("");
}
```

## Updating State

### Wrong Way (Don't Mutate State):

```
// This won't trigger a re-render!
messages.push(newMessage);
```

### Right Way (Create New State):

```
// This creates a new array and triggers a re-render
setMessages([...messages, newMessage]);
```

## Why Immutability Matters

React compares the old state with the new state to decide if it needs to re-render. If you mutate the existing state, React won't detect the change.

# Event Handling in React

React wraps native DOM events in SyntheticEvents, which work the same way but are consistent across browsers.

## Basic Event Handling

```
function Button() {
const handleClick = () => {
console.log("Button clicked!");
    };
return <button onClick={handleClick}>Click me</button>;
}
```

## Event Handler Patterns

```
// Inline arrow function (good for simple logic)
<button onClick={() => console.log("Clicked")}>Click</button>
// Function reference (good for reusable logic)
<button onClick={handleClick}>Click</button>
// Passing parameters
 <button onClick={() => handleDelete(itemId)}>Delete</button>
```

# Rendering Lists and Keys

When rendering arrays in React, each item needs a unique key prop to help React track changes efficiently.

## Basic List Rendering

```
const fruits = ["apple", "banana", "orange"];
return (
<ul>
      {fruits.map((fruit, index) => (
<li key={index}>{fruit}</li>
      ))}
</ul>
);
```

## In Our Chatbot

```
{messages.map((message, index) => (
<div
key={index}  // Unique identifier for each message
className={`message ${message.sender === "user" ? "user-
message" : "bot
message"}`}
   >
      {message.text}
</div>
))}
```

## Why Keys Matter

- **Identify which items have changed**
- **Optimize re-rendering performance**
- **Maintain component state correctly**

**Best Practice:**

Use unique, stable IDs when possible:

```
// Better than using index
{messages.map((message) => (
<div key={message.id}>  // Assuming each message has a unique ID
      {message.text}
</div>
))}
```

# Controlled Components

A controlled component's value is managed by React state, rather than the DOM directly. This ensures that React is always the single source of truth for the input's value.

## Uncontrolled (Vanilla JavaScript)

```
const input = document.getElementById('user-input');
const value = input.value;  // DOM controls the value
```

Here, the DOM directly controls the input's value. You read it when needed.

## Controlled (React)

```
const [input, setInput] = React.useState("");
<input
value={input}  // React state controls the value
onChange={(e) => setInput(e.target.value)}  // Update state on change
/>
```

React state manages the input's value. Any change updates the state, triggering a re-render.

# Benefits of Controlled Components

Controlled components bring predictability and easier management of form data in your React applications.

- **Single Source of Truth**

  The React state is always the definitive value for the input, ensuring consistency across your application.

- **Real-time Validation**

  You can validate user input as they type, providing immediate feedback and improving the user experience.

- **Conditional Rendering**

  Show or hide elements, or enable/disable features, based on the current input value.

- **Simplified Form Submission**

  Easily access all form values from the component's state when submitting, without direct DOM manipulation.

## In Our Chatbot

Here's how the user input field in our chatbot example is implemented as a controlled component:

```
<input
  id="user-input"
  type="text"
  value={input}  // Controlled by input state
  onChange={(e) => setInput(e.target.value)}  // Update state on every keystroke
  onKeyPress={handleKeyPress}
  placeholder="Type your message..."
/>
```

# `useRef` and DOM References

Sometimes you need direct access to DOM elements, for example, to manage focus, text selection, or media playback. In React, the `useRef` hook provides a way to create a mutable reference that persists across component re-renders.

## Basic `useRef` Example

```
function FocusInput() {
  const inputRef = React.useRef(null);

  const focusInput = () => {
    // Direct DOM manipulation
    inputRef.current.focus();
  };

  return (
    <div>
      <input ref={inputRef} />
      <button onClick={focusInput}>Focus Input</button>
    </div>
  );
}
```

## In Our Chatbot

```
// Reference to the chat messages container
const messagesRef = React.useRef(null);

// Use the reference for auto-scrolling
React.useEffect(() => {
  if (messagesRef.current) {
    messagesRef.current.scrollTop = messagesRef.current.scrollHeight;
  }
}, [messages]);

// Attach ref to the element in the JSX
<div className="chat-messages" ref={messagesRef}>
  {/* messages */}
</div>
```

## When to Use `useRef`

- **Accessing DOM Elements**

  For operations like focusing an input, playing media, measuring element dimensions, or triggering animations.

- **Integrating with Third-Party Libraries**

  When working with libraries that directly manipulate the DOM (e.g., charting libraries, canvas animations).

- **Storing Mutable Values**

  To hold any mutable value that needs to persist across re-renders but doesn't trigger a component update when changed.

- **What NOT to Use It For**

  Do not use `useRef` for storing state that affects the component's rendering. For that, always use `useState`.

# useEffect and Side Effects

useEffect handles side effects like API calls, timers, DOM manipulation, or cleanup.

## Basic useEffect

```
React.useEffect(() => {
  // Side effect code
  console.log("Component rendered or updated");

  // Optional cleanup function
  return () => {
    console.log("Cleanup");
  };
}, []); // Dependency array
```

## Dependency Array Patterns

1

### Runs on every render

```
React.useEffect(() => {
  console.log("Every render");
});
```

2

### Runs only once (on mount)

```
React.useEffect(() => {
  console.log("Only on mount");
}, []);
```

3

### Runs when specific values change

```
React.useEffect(() => {
  console.log("When count changes");
}, [count]);
```

## In Our Chatbot

```
// Auto-scroll when messages update
React.useEffect(() => {
  if (messagesRef.current) {
    messagesRef.current.scrollTop = messagesRef.current.scrollHeight;
  }
}, [messages]); // Only run when messages array changes
```

## Common Use Cases

### API calls

```
React.useEffect(() => {
  fetch('/api/data')
    .then(response => response.json())
    .then(data => setData(data));
}, []);
```

### Timers

```
React.useEffect(() => {
  const timer = setInterval(() => {
    setTime(new Date());
  }, 1000);
  return () => clearInterval(timer); // Cleanup
}, []);
```

### Event listeners

```
React.useEffect(() => {
 const handleResize = () => {
 setWindowWidth(window.innerWidth);
 };
 window.addEventListener('resize',
handleResize);
 return () =>
window.removeEventListener('resize',
handleResize);
}, []);
```

# React Component Lifecycle

Understanding the React component lifecycle helps you know when and how to execute code at different stages of a component's existence. In functional components, we use hooks to tap into these lifecycle events.

## Component Lifecycle Phases

Every React component goes through three main phases:

### Mounting

Component is being created and inserted into the DOM

### Updating

Component is being re-rendered as a result of changes to props or state

### Unmounting

Component is being removed from the DOM

# Key Takeaways

### Declarative vs Imperative

React lets you describe what you want, not how to achieve it

### Components

Break UI into reusable, self-contained pieces

### State

Use useState for data that changes over time

### Events

Handle user interactions with event handlers

### Lists

Use map() and key props to render dynamic lists

### Controlled Components

Let React control form inputs through state

### References

Use useRef for direct DOM access when needed

### Side Effects

Use useEffect for operations outside of rendering

## React's Mental Model

> Think of React components as functions that take props (input) and return JSX (output). When state changes, React calls your function again with the new state, compares the output, and updates only what changed in the DOM.

This makes your UI predictable and easier to debug because you always know exactly what your component will render based on its current state and props.