Complete React Tutorial: From Basics to Advanced

📚 Table of Contents

---

1. React Fundamentals

What is React?

React is a JavaScript library for building user interfaces. It lets you create reusable components that manage their own state.

Key Concepts:

- **Components**: Building blocks of React apps
- **Props**: Data passed from parent to child
- **State**: Data that changes over time
- **Hooks**: Functions that let you use React features

```javascript
// This is a React component
function App() {
  return <h1>Hello World</h1>;
}
```

---

2. JSX - JavaScript XML

JSX lets you write HTML-like code in JavaScript.

```javascript
// JSX Example
const element = <h1>Hello, world!</h1>;

// JSX with expressions
const name = "John";
const greeting = <h1>Hello, {name}!</h1>;

// JSX with attributes
const image = <img src="photo.jpg" alt="My photo" />;

// Multi-line JSX (must have ONE root element)
const card = (
  <div className="card">
    <h1>Title</h1>
    <p>Description</p>
  </div>
);
```

In Our Code:

```javascript
return (
  <div className="min-h-screen bg-gradient-to-br from-purple-600 to-blue-600">
    {/* This is JSX - looks like HTML but it's JavaScript */}
    <h1 className="text-4xl">Multiplayer Tic-Tac-Toe</h1>
  </div>
);
```

Note: Use `className` instead of `class` (because `class` is a JavaScript keyword)

---

3. Components

Components are reusable pieces of UI.

Function Components (Modern Way):

```javascript
```

```javascript
// Simple component
function Welcome() {
  return <h1>Welcome!</h1>;
}

// Component with logic
function App() {
  const greeting = "Hello!";
  return <h1>{greeting}</h1>;
}
```

In Our Code:

```javascript
function App() {
  // Component logic here
  const [ws, setWs] = useState(null);

  // Component UI here
  return (
    <div>...</div>
  );
}

export default App; // Make it available to other files
```

---

4. Props (Properties)

Props pass data from parent to child components.

```javascript
```

```javascript
// Parent component
function App() {
  return <Greeting name="Alice" age={25} />;
}

// Child component receives props
function Greeting(props) {
  return <h1>Hello, {props.name}! You are {props.age}.</h1>;
}

// Destructuring props (cleaner way)
function Greeting({ name, age }) {
  return <h1>Hello, {name}! You are {age}.</h1>;
}
```

Props are READ-ONLY

javascript

```javascript
// ❌ WRONG - Never modify props
function Greeting({ name }) {
  name = "Changed"; // Don't do this!
  return <h1>Hello, {name}</h1>;
}

// ✅ CORRECT - Use state if you need to change values
function Greeting({ name }) {
  const [displayName, setDisplayName] = useState(name);
  return <h1>Hello, {displayName}</h1>;
}
```

---

5. State - useState Hook

State is data that CHANGES over time. When state changes, React re-renders the component.

Basic useState:

javascript

```
import { useState } from "react";

function Counter() {
  // [currentValue, functionToUpdateIt] = useState(initialValue)
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>
        Increment
      </button>
    </div>
  );
}
```

Multiple State Variables:

javascript

```
function Form() {
  const [name, setName] = useState("");
  const [age, setAge] = useState(0);
  const [email, setEmail] = useState("");

  return (
    <form>
      <input value={name} onChange={(e) => setName(e.target.value)} />
      <input value={age} onChange={(e) => setAge(e.target.value)} />
      <input value={email} onChange={(e) => setEmail(e.target.value)} />
    </form>
  );
}
```

In Our Code:

javascript

```
// WebSocket connection state
const [ws, setWs] = useState(null);

// Room and player state
const [roomId, setRoomId] = useState("");
const [joined, setJoined] = useState(false);
const [player, setPlayer] = useState(null);

// Game state
const [board, setBoard] = useState(Array(9).fill(null));
const [currentTurn, setCurrentTurn] = useState("X");
const [winner, setWinner] = useState(null);

// UI state
const [status, setStatus] = useState("Enter a room ID to start");
const [playerCount, setPlayerCount] = useState(0);
```

State with Objects/Arrays:

javascript

```
// ❌ WRONG - Direct mutation
const [board, setBoard] = useState([1, 2, 3]);
board[0] = 5; // Don't mutate directly!

// ✅ CORRECT - Create new array
setBoard([5, 2, 3]); // New array
setBoard([...board, 4]); // Spread operator to add item

// For objects
const [user, setUser] = useState({ name: "John", age: 25 });
setUser({ ...user, age: 26 }); // Update specific field
```

---

6. Effects - useEffect Hook

useEffect runs side effects (code that interacts with the outside world).

Basic useEffect:

javascript

```javascript
import { useEffect } from "react";

function App() {
  const [count, setCount] = useState(0);

  // Runs after EVERY render
  useEffect(() => {
    console.log("Component rendered");
  });

  // Runs ONCE when component mounts
  useEffect(() => {
    console.log("Component mounted");
  }, []); // Empty dependency array

  // Runs when 'count' changes
  useEffect(() => {
    console.log(`Count changed to ${count}`);
  }, [count]); // Dependency array with count

  return <button onClick={() => setCount(count + 1)}>Count: {count}</button>;
}
```

Cleanup Function:

javascript

```javascript
useEffect(() => {
  // Setup
  const timer = setInterval(() => {
    console.log("Tick");
  }, 1000);

  // Cleanup (runs when component unmounts)
  return () => {
    clearInterval(timer);
  };
}, []);
```

In Our Code:

javascript

```
useEffect(() => {
  // Setup: Connect to WebSocket server
  const socket = new WebSocket("ws://localhost:8080");

  socket.onopen = () => {
    console.log("Connected to server");
    setStatus("Connected! Enter a room ID to join or create a room");
  };

  socket.onmessage = (event) => {
    const data = JSON.parse(event.data);
    // Handle incoming messages
    switch (data.type) {
      case "joined":
        setJoined(true);
        setPlayer(data.player);
        break;
      // ... more cases
    }
  };

  socket.onclose = () => {
    console.log("Disconnected from server");
    setStatus("Disconnected from server");
  };

  setWs(socket);

  // Cleanup: Close connection when component unmounts
  return () => {
    if (socket.readyState === WebSocket.OPEN) {
      socket.close();
    }
  };
}, []); // Empty array = run once on mount
```

Why empty dependency array?

- We only want to connect to WebSocket ONCE when the app starts
- If we omit it, we'd create a new connection on every render (bad!)
- The cleanup function closes the connection when we leave

---

7. Refs - useRef Hook

Refs let you "remember" values that DON'T cause re-renders when changed.

Use Cases:

1. Access DOM elements directly
2. Store values that persist across renders but don't trigger re-renders

```javascript
```

```
import { useRef } from "react";

// Example 1: Focus an input
function LoginForm() {
  const inputRef = useRef(null);

  const focusInput = () => {
    inputRef.current.focus(); // Access DOM element
  };

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

// Example 2: Store previous value
function Counter() {
  const [count, setCount] = useState(0);
  const prevCountRef = useRef();

  useEffect(() => {
    prevCountRef.current = count; // Update ref (no re-render)
  });

  const prevCount = prevCountRef.current;

  return (
    <div>
      <p>Current: {count}, Previous: {prevCount}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}
```

In Our Code:

```
javascript
```

```
const inputRef = useRef(null);

return (
  <input
    ref={inputRef} // Attach ref to input element
    type="text"
    value={roomId}
    onChange={(e) => setRoomId(e.target.value)}
    onKeyPress={(e) => e.key === "Enter" && joinRoom()}
  />
);
```

useState vs useRef:

- useState: Changes cause re-render ✅ Use for UI data
- useRef: Changes DON'T cause re-render ✅ Use for DOM access, timers, etc.

---

8. Event Handling

React events are similar to DOM events but with camelCase naming.

Basic Events:

javascript

```
function EventExamples() {
  const handleClick = () => {
    console.log("Clicked!");
  };

  const handleChange = (event) => {
    console.log("Input value:", event.target.value);
  };

  const handleSubmit = (event) => {
    event.preventDefault(); // Prevent page refresh
    console.log("Form submitted");
  };

  return (
    <div>
      <button onClick={handleClick}>Click Me</button>

      <input onChange={handleChange} />

      <form onSubmit={handleSubmit}>
        <button type="submit">Submit</button>
      </form>
    </div>
  );
}
```

Event Object:

javascript

```
function Input() {
  const handleKeyPress = (e) => {
    console.log("Key pressed:", e.key);
    console.log("Target element:", e.target);
    console.log("Current value:", e.target.value);

    if (e.key === "Enter") {
      console.log("Enter pressed!");
    }
  };

  return <input onKeyPress={handleKeyPress} />;
}
```

In Our Code:

```javascript
// Button click
<button onClick={joinRoom}>Join Room</button>

// Input change
<input
  value={roomId}
  onChange={(e) => setRoomId(e.target.value)}
  onKeyPress={(e) => e.key === "Enter" && joinRoom()}
/>

// Game board click
<button
  onClick={() => makeMove(index)} // Arrow function to pass argument
  disabled={!!cell || !!winner || currentTurn !== player}
>
  {cell}
</button>
```

Inline vs Named Handlers:

```javascript
// Inline (good for simple logic)
<button onClick={() => setCount(count + 1)}>+</button>

// Named (better for complex logic)
const increment = () => {
  setCount(count + 1);
  console.log("Incremented");
};
<button onClick={increment}>+</button>
```

---

9. Conditional Rendering

Show different UI based on conditions.

Method 1: if/else (Outside JSX)

```javascript
```

```
function Greeting({ isLoggedIn }) {
  if (isLoggedIn) {
    return <h1>Welcome back!</h1>;
  } else {
    return <h1>Please sign in.</h1>;
  }
}
```

Method 2: Ternary Operator (Inside JSX)

javascript

```
function Greeting({ isLoggedIn }) {
  return (
    <div>
      {isLoggedIn ? (
        <h1>Welcome back!</h1>
      ) : (
        <h1>Please sign in.</h1>
      )}
    </div>
  );
}
```

Method 3: Logical && (Show or Nothing)

javascript

```
function Mailbox({ unreadMessages }) {
  return (
    <div>
      <h1>Mailbox</h1>
      {unreadMessages.length > 0 && (
        <h2>You have {unreadMessages.length} unread messages.</h2>
      )}
    </div>
  );
}
```

In Our Code:

javascript

```
return (
  <div>
    <h1>Multiplayer Tic-Tac-Toe</h1>

    {!joined ? (
      // Show room joining UI
      <div className="space-y-4">
        <input placeholder="Enter room name" />
        <button onClick={joinRoom}>Join Room</button>
      </div>
    ) : (
      // Show game board UI
      <div className="space-y-6">
        <div>Room: {roomId}</div>
        <div>Game board here...</div>

        {winner && (
          <button onClick={resetGame}>Play Again</button>
        )}
      </div>
    )}
  </div>
);
```

Breakdown:

- `!joined ? ... : ...` - Show join screen OR game screen
- `{winner && <button>...}` - Only show "Play Again" if there's a winner

---

10. Lists & Keys

Render arrays of data efficiently.

Basic List:

```javascript
```

```
function TodoList() {
  const todos = ["Buy milk", "Walk dog", "Write code"];

  return (
    <ul>
      {todos.map((todo, index) => (
        <li key={index}>{todo}</li>
      ))}
    </ul>
  );
}
```

Why Keys Matter:

javascript

```
// ❌ BAD: Using index as key (can cause bugs)
{items.map((item, index) => (
  <div key={index}>{item.name}</div>
))}

// ✅ GOOD: Using unique ID
{items.map((item) => (
  <div key={item.id}>{item.name}</div>
))}
```

Keys help React:

- Identify which items changed
- Determine which items to re-render
- Maintain component state correctly

In Our Code:

javascript

```
<div className="grid grid-cols-3 gap-2">
  {board.map((cell, index) => (
    <button
      key={index} // Index is OK here because board order never changes
      onClick={() => makeMove(index)}
      className={/* dynamic classes */}
    >
      {cell}
    </button>
  ))}
</div>
```

Why index is OK here:

- The board has exactly 9 cells
- Cells never get reordered, added, or removed
- Index is stable

---

11. WebSocket Integration (Advanced)

WebSockets enable real-time, two-way communication.

How It Works:

```
Client                    Server
  |                         |
  |------ Connect (ws://) ------->|
  |<----- Connection OK ----------|
  |                         |
  |------ Send Message ---------->|
  |<----- Send Message ----------|
  |                         |
  |------ Close Connection ------>|
```

In Our Code:

1. Connect to Server

javascript

```
useEffect(() => {
  const socket = new WebSocket("ws://localhost:8080");

  socket.onopen = () => {
    console.log("Connected!");
  };

  setWs(socket); // Store in state

  return () => socket.close(); // Cleanup
}, []);
```

2. Listen for Messages

javascript

```
socket.onmessage = (event) => {
  const data = JSON.parse(event.data); // Parse JSON

  switch (data.type) {
    case "joined":
      setJoined(true);
      setPlayer(data.player);
      break;

    case "gameState":
      setBoard(data.board);
      setCurrentTurn(data.currentTurn);
      setWinner(data.winner);
      break;

    case "error":
      setStatus(`Error: ${data.message}`);
      break;
  }
};
```

3. Send Messages

javascript

```
const joinRoom = () => {
  if (ws && ws.readyState === WebSocket.OPEN) {
    ws.send(JSON.stringify({
      type: "join",
      roomId: roomId.trim()
    }));
  }
};

const makeMove = (index) => {
  ws.send(JSON.stringify({
    type: "move",
    roomId,
    index
  }));
};
```

4. Handle Disconnection

javascript

```
socket.onclose = () => {
  console.log("Disconnected");
  setStatus("Disconnected from server");
};
```

---

12. Advanced Patterns

Pattern 1: Guard Clauses (Early Returns)

javascript

```
const makeMove = (index) => {
  // Check conditions and return early if invalid
  if (!joined || playerCount < 2) {
    setStatus("Waiting for opponent...");
    return; // Stop execution
  }

  if (board[index] || winner) return;

  if (currentTurn !== player) {
    setStatus("Not your turn!");
    return;
  }

  // If we get here, move is valid
  ws.send(JSON.stringify({ type: "move", roomId, index }));
};
```

Pattern 2: Dynamic Classes

javascript

```
<button
  className={`
    h-24 text-4xl font-bold rounded-lg transition-all
    ${cell
      ? cell === "X"
        ? "bg-blue-500 text-white"
        : "bg-red-500 text-white"
      : "bg-gray-200 hover:bg-gray-300"
    }
    ${!cell && !winner && currentTurn === player && playerCount === 2
      ? "cursor-pointer transform hover:scale-105"
      : "cursor-not-allowed opacity-75"
    }
  `}
>
  {cell}
</button>
```

Breakdown:

- Base classes: `h-24 text-4xl font-bold...`
- Conditional color: `${cell ? ... : ...}`
- Conditional interaction: `${!cell && ... ? ... : ...}`

Pattern 3: State Management

```javascript
// Related state kept together
const [ws, setWs] = useState(null);
const [roomId, setRoomId] = useState("");
const [joined, setJoined] = useState(false);

// Could be improved with useReducer for complex state:
const [state, dispatch] = useReducer(reducer, initialState);

// But useState is fine for our app!
```

Pattern 4: Derived State (Don't Store What You Can Calculate)

```javascript
// ❌ BAD: Storing derived state
const [items, setItems] = useState([1, 2, 3]);
const [count, setCount] = useState(3); // Duplicate!

// ✅ GOOD: Calculate on render
const [items, setItems] = useState([1, 2, 3]);
const count = items.length; // Derived!
```

Pattern 5: Controlled Components

```javascript
// Input controlled by React state
const [value, setValue] = useState("");

<input
  value={value} // React controls the value
  onChange={(e) => setValue(e.target.value)} // Update state
/>
```

🎯 Complete Code Explanation

Let's trace through a full user interaction:

1. User Opens App

```javascript
```

```
// App mounts
useEffect(() => {
  // Connect to WebSocket
  const socket = new WebSocket("ws://localhost:8080");
  setWs(socket);
}, []); // Runs once
```

2. User Enters Room ID & Clicks "Join"

javascript

```
// User types "room1"
<input
  value={roomId} // "room1"
  onChange={(e) => setRoomId(e.target.value)} // Updates state
/>

// User clicks "Join Room"
const joinRoom = () => {
  ws.send(JSON.stringify({
    type: "join",
    roomId: "room1"
  }));
};
```

3. Server Responds

javascript

```
socket.onmessage = (event) => {
  const data = JSON.parse(event.data);
  // data = { type: "joined", player: "X", playerCount: 1 }

  setJoined(true);          // joined = true
  setPlayer("X");           // player = "X"
  setPlayerCount(1);        // playerCount = 1
  setStatus("You are Player X. Waiting for opponent...");
};
```

4. UI Re-renders

javascript

```
// joined = true, so show game UI
{!joined ? (
  <div>Join Room</div>
) : (
  <div>
    {/* Game board appears! */}
    Room: room1
    Players: 1/2
    You are: Player X
  </div>
)}
```

5. User Makes Move

javascript

```
// User clicks cell 4
const makeMove = (4) => {
  // Validation
  if (currentTurn !== player) return; // Is it my turn?
  if (board[4]) return; // Is cell empty?

  // Send move to server
  ws.send(JSON.stringify({
    type: "move",
    roomId: "room1",
    index: 4
  }));
};
```

6. Server Broadcasts Game State

javascript

```
socket.onmessage = (event) => {
  const data = JSON.parse(event.data);
  // data = {
  //   type: "gameState",
  //   board: [null, null, null, null, "X", null, null, null, null],
  //   currentTurn: "O",
  //   winner: null
  // }

  setBoard(data.board);          // Update board
  setCurrentTurn(data.currentTurn); // Switch turn
  setStatus("Player O's turn");    // Update status
};
```

7. Board Re-renders

javascript

```
{board.map((cell, index) => (
  <button key={index}>
    {cell}  {/* Shows "X" in cell 4 */}
  </button>
))}
```

🚀 React Best Practices

1. Keep Components Small

javascript

```
// ❌ BAD: One huge component
function App() {
  // 500 lines of code...
}

// ✅ GOOD: Split into smaller components
function App() {
  return (
    <div>
      <Header />
      <GameBoard />
      <StatusBar />
    </div>
  );
}
```

2. Use Meaningful Names

javascript

```
// ❌ BAD
const [x, setX] = useState(false);
const [d, setD] = useState([]);

// ✅ GOOD
const [isJoined, setIsJoined] = useState(false);
const [board, setBoard] = useState([]);
```

3. Extract Complex Logic

javascript

```
// ❌ BAD: Logic in JSX
<button onClick={() => {
  if (!joined) return;
  if (board[index]) return;
  ws.send(JSON.stringify({ type: "move", index }));
}}>
  Click
</button>

// ✅ GOOD: Extract to function
const handleMove = (index) => {
  if (!joined || board[index]) return;
  ws.send(JSON.stringify({ type: "move", index }));
};

<button onClick={() => handleMove(index)}>Click</button>
```

4. Avoid Unnecessary Re-renders

javascript

```
// Use React.memo for expensive components
const ExpensiveComponent = React.memo(({ data }) => {
  // Only re-renders if 'data' changes
  return <div>{/* Expensive rendering */}</div>;
});
```

5. Handle Errors

javascript

```
useEffect(() => {
  const socket = new WebSocket("ws://localhost:8080");

  socket.onerror = (error) => {
    console.error("WebSocket error:", error);
    setStatus("Connection error");
  };

  // ...
}, []);
```

📝 Summary

React Core Concepts:

1. **Components** - Building blocks
2. **JSX** - HTML-like syntax
3. **Props** - Pass data down
4. **State** - Data that changes
5. **Effects** - Side effects
6. **Events** - User interactions
7. **Conditional Rendering** - Show/hide UI
8. **Lists** - Render arrays

Our App Architecture:

```
App Component
├── State Management (useState)
│   ├── WebSocket connection
│   ├── Room & player info
│   ├── Game board
│   └── UI status
├── Side Effects (useEffect)
│   ├── Connect to WebSocket
│   ├── Listen for messages
│   └── Cleanup on unmount
├── Event Handlers
│   ├── joinRoom()
│   ├── makeMove()
│   ├── resetGame()
│   └── leaveRoom()
└── UI Rendering
    ├── Join screen (conditional)
    └── Game screen (conditional)
        ├── Room info
        ├── Status message
        ├── Game board (list)
        └── Action buttons
```

Key Takeaways:

- React is **declarative**: Describe what UI should look like, React handles updates
- **State changes** trigger re-renders
- **useEffect** for side effects (API calls, WebSocket, timers)

- **Keys** help React identify list items
- **Hooks** must be called at the top level (not in loops/conditions)

You've now learned React from basics to building a real-time multiplayer game! 🎉