React Essentials

A modern JavaScript library for building user interfaces

Welcome to React!

What you'll learn today:

- **React Fundamentals** - Core concepts and architecture
- **JSX** - Writing React elements
- **Components** - Building reusable UI pieces
- **Props & State** - Data flow in React
- **Hooks** - Modern React patterns
- **Event Handling** - User interactions

Course Overview

🎯 Learning Objectives

By the end of this session, you will be able to:

- Understand React's component-based architecture
- Write JSX to describe UI elements
- Create reusable React components
- Manage component state and props
- Use React hooks for modern development
- Handle user events and form interactions

⏱️ Session Structure

- **Introduction** (15 min) - React basics and concepts
- **JSX Deep Dive** (30 min) - Syntax and patt

Prerequisites

✅ What you should know:

- **JavaScript fundamentals** (ES6+ syntax)
- **HTML & CSS** basics
- **DOM manipulation** concepts
- **Modern web development** concepts

📚 Helpful but not required:

- **TypeScript** experience
- **Node.js** and **npm** familiarity
- **Build tools** (Webpack, Vite)
- **Version control** (Git)

Why React?

🚀 Industry Standard

- **Most popular** frontend framework
- **High demand** in job market
- **Large ecosystem** of libraries and tools
- **Strong community** support

🎯 Perfect for Learning

- **Declarative** programming model
- **Component-based** architecture
- **Excellent documentation**
- **Rich learning resources**

Getting Started

🛠️ Development Environment

```
# Create a new React project
npx create-react-app my-app
cd my-app
npm start
```

📦 Key Tools

- **Node.js** - JavaScript runtime
- **npm/yarn** - Package managers
- **VS Code** - Recommended editor
- **React Developer Tools** - Browser extension

🌐 Online Resources

React Essentials

A modern JavaScript library for building user interfaces

What is React?

React is a **declarative, efficient, and flexible** JavaScript library for building user interfaces.

Core Concepts:

- **Component-Based**: Build encapsulated components that manage their own state
- **Virtual DOM**: Efficient rendering through a lightweight representation of the actual DOM
- **Declarative**: Describe what you want, React handles the DOM updates
- **Learn Once, Write Anywhere**: Use React for web, mobile, and desktop

React Architecture

Traditional DOM

Direct DOM manipulation

Slow updates

Complex state management

React Virtual DOM

Virtual DOM diffing

Efficient updates

Component state

History

React's Evolution

Year	Milestone	Key Features
2011	Created at Facebook	Internal use for Facebook Ads
2013	Open-sourced	Released to public
2015	React Native	Mobile development
2016	React Fiber	New reconciliation algorithm
2018	React Hooks	Functional components with state
2020	React 18	Concurrent features, Suspense

Key Contributors:

- **Jordan Walke** - Original creator

Why Use React?

🚀 Performance

- Virtual DOM for efficient updates
- Optimized rendering algorithms
- Minimal DOM manipulation

🧩 Component Reusability

- Build once, use everywhere
- Composable architecture
- Easy to maintain and test

🌐 Ecosystem

- Massive community support

React vs Other Frameworks

Feature	React	Vue	Angular
Learning Curve	Moderate	Easy	Steep
Performance	Excellent	Good	Good
Ecosystem	Massive	Growing	Large
Mobile	React Native	NativeScript	Ionic
Backing	Meta	Community	Google

React Advantages:

- **Flexibility**: Minimal opinions, maximum freedom
- **Community**: Largest JavaScript ecosystem
- **Jobs**: High demand in job market

React Use Cases

🌐 Web Applications

- Single Page Applications (SPAs)
- Progressive Web Apps (PWAs)
- E-commerce platforms
- Social media applications

📱 Mobile Applications

- **React Native** for iOS and Android
- Cross-platform development
- Native performance

🖥️ Desktop Applications

Popular React Applications

Social Media

- Facebook
- Instagram
- Twitter (X)
- LinkedIn

Entertainment

- Netflix
- Discord
- Twitch
- Spotify

React Development Tools

🛠️ Essential Tools

- **Create React App** - Quick project setup
- **React Developer Tools** - Browser extension
- **ESLint** - Code quality
- **Prettier** - Code formatting

📦 Build Tools

- **Webpack** - Module bundling
- **Vite** - Fast development server
- **Babel** - JavaScript transpilation
- **TypeScript** - Type safety

Simple React Example

<script type="text/babel"> const { useState } = React; function SimpleCounter() { const [count, setCount] = useState(0); return (

Simple React Component

Count: **{count}**

<button onClick={() => setCount(count + 1)} style={{ padding: '10px 20px', margin: '5px', backgroundColor: '#4CAF50', color: 'white', border: 'none', borderRadius: '4px', cursor: 'pointer' }} > Increment </button> <button onClick={() => setCount(count - 1)} style={{ padding: '10px 20px', margin: '5px', backgroundColor: '#f44336', color: 'white', border: 'none', borderRadius: '4px', cursor: 'pointer' }} > Decrement </button>
); } ReactDOM.createRoot(document.getElementById('simple-react-demo')).render(<SimpleCounter />); </script> <script src="https://unpkg.com/react@18/umd/react.development.js" crossorigin></script>

Component Architecture

Component Tree

```
App
├── Header
├── Sidebar
└── Main
├── Card
├── Card
└── Card
```

Data Flow

Parent → Props → Child

Child → Events → Parent

State Management:

React Learning Path

📚 Beginner Level

1. **JSX Syntax** - Writing React elements
2. **Components** - Building reusable UI pieces
3. **Props** - Passing data between components
4. **State** - Managing component data
5. **Event Handling** - User interactions

🚀 Intermediate Level

1. **Hooks** - useState, useEffect, useContext
2. **Conditional Rendering** - Dynamic UI
3. **Lists & Keys** - Rendering collections
4. **Forms** - Controlled components

JSX (JavaScript XML)

JSX

JSX is a syntax extension for JavaScript that looks like HTML but compiles down to JavaScript.

It's used with React to describe the UI in a more readable and declarative way.

Why use JSX

- Easier to visualize UI compared to React.createElement().
- Makes component code more intuitive and closer to HTML, which web developers already know.

JSX Syntax

JSX looks like HTML but compiles to JS.

Example JSX:

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

Compiles to:

```
const element = React.createElement("h1", null, "Hello, world!");
```

JSX Embedding Expressions

You can embed any JavaScript expression in JSX by wrapping it in curly braces `{}`.

```
const name = "John Doe";
const element = <h1>Hello, {name}!</h1>;

// You can also use expressions
const user = { firstName: "John", lastName: "Doe" };
const greeting = (
  <h1>
    Hello, {user.firstName} {user.lastName}!
  </h1>
);

// Function calls work too
function formatName(user) {
  return user.firstName + " " + user.lastName;
}
const formattedGreeting = <h1>Hello, {formatName(user)}!</h1>;
```

JSX as an expression

JSX can be stored in variables, passed to functions, passed to other components and returned from functions.

```
// Store in variables
const element = <h1>Hello, world!</h1>;

// Use in conditionals
function getGreeting(user) {
  if (user) {
    return <h1>Hello, {formatName(user)}!</h1>;
  }
  return <h1>Hello, Stranger.</h1>;
}

// Use in loops
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) => (
    <li key={number.toString()}>{number}</li>
```

Conditional Rendering in JSX

JSX supports conditional rendering using JavaScript expressions.

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

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

Looping in JSX

You can render lists of elements using JavaScript's `map()` function.

```
// Basic list rendering
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) => (
  <li key={number.toString()}>{number}</li>
));

// In a component
function NumberList({ numbers }) {
  return (
    <ul>
      {numbers.map((number) => (
        <li key={number.toString()}>{number}</li>
      ))}
    </ul>
  );
}

// With filtering
function TodoList({ todos }) {
  return (
    <ul>
      {todos
        .filter((todo) => !todo.completed)
        .map((todo) => (
          <li key={todo.id}>{todo.text}</li>
        ))}
    </ul>
```

Event Handling in JSX

JSX uses camelCase for event names and passes functions as event handlers.

```
// Basic event handling
function Button() {
  function handleClick() {
    alert("Button clicked!");
  }

  return <button onClick={handleClick}>Click me</button>;
}

// With parameters
function Button({ id, text }) {
  function handleClick(id) {
    console.log(`Button ${id} clicked`);
  }

  return <button onClick={() => handleClick(id)}>{text}</button>;
}

// Form handling
function NameForm() {
  const [value, setValue] = useState("");

  function handleSubmit(event) {
    event.preventDefault();
    alert("A name was submitted: " + value);
  }

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        value={value}
        onChange={(e) => setValue(e.target.value)}
      />
      <button type="submit">Submit</button>
```

JSX Rules

- **Single Parent Element**: JSX must have exactly one parent element
- **Use `className` instead of `class`**: HTML attributes use camelCase
- **Self-closing tags are required**: `<input />` not `<input>`
- **Use `htmlFor` instead of `for`**: For label elements
- **Use `onClick` instead of `onclick`**: Event handlers use camelCase

```
// ❌ Wrong - multiple parent elements
function WrongComponent() {
  return (
    <h1>Title</h1>
    <p>Paragraph</p>
  );
}

// ✅ Correct - single parent element
function CorrectComponent() {
  return (
    <div>
      <h1>Title</h1>
      <p>Paragraph</p>
    </div>
```

JSX vs HTML Differences

HTML	JSX
`class="container"`	`className="container"`
`<input>`	`<input />`
`for="name"`	`htmlFor="name"`
`onclick="handleClick()"`	`onClick={handleClick}`
`style="color: red"`	`style={{color: 'red'}}`

```
// HTML style
<div class="container" onclick="handleClick()">
  <label for="name">Name:</label>
  <input type="text" id="name">
</div>

// JSX style
```

Interactive JSX Demo

<script type="text/babel"> const { useState } = React; function JSXDemo() { const [name, setName] = useState("World"); const [items, setItems] = useState(['React', 'JSX', 'Components']); const [newItem, setNewItem] = useState(''); const addItem = () => { if (newItem.trim()) { setItems([...items, newItem]); setNewItem(''); } }; return (

Interactive JSX Example

<label htmlFor="name-input">Your name: </label> <input id="name-input" value={name} onChange={(e) => setName(e.target.value)} placeholder="Enter your name" style={{ marginLeft: '10px', padding: '5px' }} />
Hello, **{name}**!

Dynamic List:

    {items.map((item, index) => (
    {item}

JSX Best Practices

✅ Do's

- **Use meaningful component names** (PascalCase)
- **Always include keys** when rendering lists
- **Use fragments** to avoid unnecessary wrapper divs
- **Extract complex logic** into separate functions
- **Use proper event handling** (prevent default, stop propagation)

❌ Don'ts

- **Don't use array index as key** (unless list is static)
- **Don't put too much logic in JSX**
- **Don't forget to handle loading/error states**
- **Don't use inline styles for complex styling**

Common JSX Patterns

1. Conditional Rendering

```
{
  isLoading ? <Spinner /> : <Content />;
}
```

2. List Rendering

```
{
  items.map((item) => <Item key={item.id} {...item} />);
}
```

3. Fragment Usage

```
<>
  <Header />
  <Main />
```

JSX Performance Tips

- **Use React.memo()** for expensive components
- **Avoid creating objects/functions in render**
- **Use useCallback for event handlers**
- **Use useMemo for expensive calculations**

```
// ✅ Good - memoized component
const ExpensiveComponent = React.memo(({ data }) => {
  return <div>{/* expensive rendering */}</div>;
});

// ✅ Good - memoized callback
function Parent() {
  const handleClick = useCallback(() => {
    // handle click
  }, []);

  return <Child onClick={handleClick} />;
```

Next Steps

- **Components**: Building reusable UI pieces
- **Props**: Passing data between components
- **State**: Managing component data
- **Hooks**: Using React's built-in hooks
- **Event Handling**: Responding to user interactions

Questions & Practice

Try building a simple component using JSX!

```jsx
function TodoItem({ todo, onToggle }) {
  return (
    <div
      style={{
        textDecoration: todo.completed ? "line-through" : "none",
        cursor: "pointer",
      }}
    >
      <input
        type="checkbox"
        checked={todo.completed}
        onChange={() => onToggle(todo.id)}
      />
      {todo.text}
    </div>
  );
}
```

React Components

Components

Components are the building blocks of React applications. They let you split the UI into independent, reusable pieces.

What are Components?

Components are **functions or classes** that return JSX. They can be:

- **Reusable** - Use the same component multiple times
- **Composable** - Combine components to build complex UIs
- **Isolated** - Each component manages its own logic and styling

```
// Function Component
function Welcome() {
  return <h1>Hello, World!</h1>;
}

// Arrow Function Component
const Welcome = () => {
  return <h1>Hello, World!</h1>;
};

// Using the component
function App() {
  return (
```

Component Styles

1. Inline Styles

```
function StyledComponent() {
  return (
    <div
      style={{
        backgroundColor: "blue",
        color: "white",
        padding: "20px",
        borderRadius: "8px",
        fontSize: "18px",
      }}
    >
      Styled with inline styles
    </div>
  );
}
```

2. CSS Classes

Props (Properties)

Props are how components receive data from their parent components.

Basic Props

```
function Greeting(props) {
  return <h1>Hello, {props.name}!</h1>;
}

// Using the component
<Greeting name="John" />
<Greeting name="Jane" />
```

Destructuring Props

```
function Greeting({ name, age, city }) {
  return (
    <div>
      <h1>Hello, {name}!</h1>
```

Props Example

<script type="text/babel"> const { useState } = React; function UserCard({ name, email, role, avatar }) { return (

{name}

{name}

Email: {email}

Role: {role}

); } function PropsDemo() { const [users] = useState([{ name: "John Doe", email: "john@example.com", role: "Developer", avatar: "https://via.placeholder.com/60/4CAF50/FFFFFF?text=JD" }, { name: "Jane Smith", email: "jane@example.com", role: "Designer", avatar: "https://via.placeholder.com/60/2196F3/FFFFFF?text=JS" }]); return (

State

State allows components to manage their own data that can change over time.

useState Hook

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

function Counter() {
  const [count, setCount] = useState(0);
  const [name, setName] = useState("");

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

      <input
        value={name}
        onChange={(e) => setName(e.target.value)}
        placeholder="Enter your name"
      />
```

State Example

```
<script type="text/babel"> const { useState } = React; function TodoList() { const [todos,
setTodos] = useState([ { id: 1, text: 'Learn React', completed: false }, { id: 2, text: 'Build a
project', completed: false }, { id: 3, text: 'Deploy to production', completed: false } ]);
const [newTodo, setNewTodo] = useState(''); const addTodo = () => { if (newTodo.trim())
{ setTodos(prevTodos => [ ...prevTodos, { id: Date.now(), text: newTodo, completed: false
} ]); setNewTodo(''); } }; const toggleTodo = (id) => { setTodos(prevTodos =>
prevTodos.map(todo => todo.id === id ? { ...todo, completed: !todo.completed } : todo
) ); }; return (
```

Todo List with State

```
<input value={newTodo} onChange={(e) => setNewTodo(e.target.value)}
placeholder="Add new todo" style={{ marginRight: '10px', padding: '5px' }} /> <button
onClick={addTodo}>Add</button>
```

Hooks

Hooks are functions that let you "hook into" React state and lifecycle features from function components.

useState Hook

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

function Example() {
  // Declare a state variable
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Click me</button>
    </div>
  );
}
```

More Hooks

useContext Hook

```
import { createContext, useContext, useState } from "react";

// Create a context
const ThemeContext = createContext();

// Provider component
function ThemeProvider({ children }) {
  const [theme, setTheme] = useState("light");

  return (
    <ThemeContext.Provider value={{ theme, setTheme }}>
      {children}
    </ThemeContext.Provider>
  );
}

// Consumer component
function ThemedButton() {
  const { theme, setTheme } = useContext(ThemeContext);

  return (
    <button onClick={() => setTheme(theme === "light" ? "dark" : "light")}>
      Current theme: {theme}
    </button>
```

Hooks Example

```
<script type="text/babel"> const { useState, useEffect, useRef } = React; function
HooksDemo() { const [count, setCount] = useState(0); const [windowWidth,
setWindowWidth] = useState(window.innerWidth); const inputRef = useRef(null); //
useEffect for window resize useEffect(() => { const handleResize = () =>
setWindowWidth(window.innerWidth); window.addEventListener('resize', handleResize);
// Cleanup function return () => window.removeEventListener('resize', handleResize); },
[]); // useEffect for document title useEffect(() => { document.title = `Count: ${count}`; },
[count]); const focusInput = () => { inputRef.current.focus(); }; return (
```

Hooks Demo

Count: {count}

```
<button onClick={() => setCount(count + 1)}>Increment</button>
Window width: {windowWidth}px
```

Event Handling

React events are named using camelCase and pass functions as event handlers.

Basic Event Handling

```
function Button() {
  const handleClick = () => {
    alert("Button clicked!");
  };

  return <button onClick={handleClick}>Click me</button>;
}
```

Event with Parameters

```
function Button({ id, text }) {
  const handleClick = (id, event) => {
    console.log(`Button ${id} clicked`);
    console.log("Event:", event);
```

Event Handling Example

<script type="text/babel"> const { useState } = React; function EventsDemo() { const [mousePosition, setMousePosition] = useState({ x: 0, y: 0 }); const [keyPressed, setKeyPressed] = useState(''); const [formData, setFormData] = useState({ name: '', email: '' }); const handleMouseMove = (e) => { setMousePosition({ x: e.clientX, y: e.clientY }); }; const handleKeyPress = (e) => { setKeyPressed(e.key); }; const handleFormChange = (e) => { const { name, value } = e.target; setFormData(prev => ({ ...prev, [name]: value })); }; const handleSubmit = (e) => { e.preventDefault(); alert(`Form submitted: ${JSON.stringify(formData)}`); }; return (

Event Handling Demo

Mouse Position:

X: {mousePosition.x}, Y: {mousePosition.y}

Move your mouse over this area!

Component Best Practices

✅ Do's

- Use descriptive component names (PascalCase)
- Keep components small and focused
- Extract reusable logic into custom hooks
- Use proper prop validation
- Handle loading and error states

❌ Don'ts

- Don't create components that are too large
- Don't put business logic in components
- Don't forget to clean up effects
- Don't mutate state directly

Component Composition

Children Prop

```
function Card({ children, title }) {
  return (
    <div className="card">
      <h3>{title}</h3>
      {children}
    </div>
  );
}

// Usage
<Card title="User Profile">
  <p>This content goes inside the card</p>
  <button>Action</button>
</Card>;
```

Render Props

Next Steps

- **Advanced Hooks**: useMemo, useCallback, useReducer
- **Context API**: Global state management
- **Performance**: React.memo, optimization techniques
- **Testing**: Unit and integration testing
- **Routing**: React Router for navigation

Practice Exercise

Build a simple component that combines all concepts:

```
function ProductCard({ product, onAddToCart }) {
  const [quantity, setQuantity] = useState(1);
  const [isHovered, setIsHovered] = useState(false);

  const handleAddToCart = () => {
    onAddToCart(product, quantity);
    setQuantity(1);
  };

  return (
    <div
      className="product-card"
      onMouseEnter={() => setIsHovered(true)}
      onMouseLeave={() => setIsHovered(false)}
    >
      <img src={product.image} alt={product.name} />
      <h3>{product.name}</h3>
      <p>${product.price}</p>
      <input
        type="number"
        value={quantity}
        onChange={(e) => setQuantity(parseInt(e.target.value))}
        min="1"
      />
      <button onClick={handleAddToCart}>Add to Cart</button>
    </div>
  );
}

---

<!-- class: invert -->

## Summary

### 🎯 **What We've Covered**

#### **React Fundamentals**
- ✅ Component-based architecture
- ✅ Virtual DOM and performance
- ✅ Declarative programming model
- ✅ Cross-platform development

#### **JSX & Components**
- ✅ JSX syntax and expressions
- ✅ Component creation and composition
- ✅ Props and state management
- ✅ Event handling patterns

#### **Modern React Patterns**
- ✅ Functional components with hooks
- ✅ useState and useEffect
- ✅ Component styling approaches
- ✅ Best practices and optimization

---

## Key Takeaways

### 🧩 **Component-Based Architecture**
- **Reusable** components for maintainable code
- **Composable** design for complex UIs
- **Isolated** state and logic management

### 🎨 **JSX Benefits**
- **Declarative** UI description
- **JavaScript power** in markup
- **Component composition** made easy

### ⚡ **Modern React**
- **Hooks** for functional components
- **Performance** optimization techniques
- **Developer experience** improvements

---

## Next Steps

### 📚 **Continue Learning**
- **Advanced Hooks** - useMemo, useCallback, useReducer
- **Context API** - Global state management
- **React Router** - Navigation and routing
- **Testing** - Unit and integration tests
- **Performance** - Optimization techniques

### 🛠️ **Practice Projects**
- **Todo App** - State management practice
- **Weather App** - API integration
- **E-commerce Site** - Complex component structure
- **Social Media Clone** - Full-stack application

### 🌐 **Resources**
- **Official Docs** - react.dev
- **Community** - React Discord, Reddit
- **Courses** - Udemy, Pluralsight, Frontend Masters
- **Practice** - CodeSandbox, CodePen, GitHub

---

## Q&A Session

### 🤔 **Common Questions**
- How does React compare to Vue/Angular?
- When should I use class vs functional components?
- What's the best way to manage global state?
- How do I optimize React performance?

### 💡 **Tips for Success**
- **Start small** - Build simple components first
- **Practice regularly** - Code daily if possible
- **Read the docs** - React documentation is excellent
- **Join communities** - Learn from others
- **Build projects** - Apply what you learn

---

<!-- class: lead -->

## Thank You!

### 🎉 **Congratulations!**
You've completed the React Essentials course!

### **Stay Connected**
```