React Complete Tutorial: From Python Developer to React Pro

Welcome! As a Python developer, you'll find React's component-based approach similar to how you might structure classes and modules. Let's dive deep into each topic.

---

1. Component-Based Architecture

1.1 What is Component-Based Architecture?

Think of React components like Python functions or classes - they're reusable pieces of code that take inputs (props) and return outputs (JSX). Just as you might have a `User` class in Python with methods, React has components that encapsulate UI logic and appearance.

Simple Definition: A component is a JavaScript function that returns JSX (HTML-like syntax) describing what should appear on screen.

1.2 The Mental Model

```
Traditional Web Development:
HTML + CSS + JavaScript = One big interconnected file

React Way:
App Component
├── Header Component
├── TaskList Component
│   ├── TaskItem Component
│   ├── TaskItem Component
│   └── TaskItem Component
└── Footer Component
```

1.3 Function Components vs Class Components

Function Components (Modern Approach)

```jsx
// Simple function component
function Welcome() {
  return <h1>Hello, World!</h1>;
}

// Arrow function component (more common)
const Welcome = () => {
  return <h1>Hello, World!</h1>;
};

// With props (like function parameters)
const TaskItem = (props) => {
  return <div>{props.taskName}</div>;
};

// Destructured props (cleaner)
const TaskItem = ({ taskName, isCompleted }) => {
  return (
    <div>
      <span>{taskName}</span>
      {isCompleted && <span>✅</span>}
    </div>
  );
};
```

Class Components (Legacy, but good to know)

```jsx
import React, { Component } from 'react';

class TaskItem extends Component {
  render() {
    return (
      <div>
        <span>{this.props.taskName}</span>
        {this.props.isCompleted && <span>✅</span>}
      </div>
    );
  }
}
```

1.4 Props: The Component's Parameters

Props are like function parameters in Python. They're read-only data passed from parent to child components.

```jsx
// Parent Component
const App = () => {
  const tasks = [
    { id: 1, name: "Learn React", completed: false },
    { id: 2, name: "Build a project", completed: false }
  ];

  return (
    <div>
      <h1>My Task App</h1>
      <TaskList tasks={tasks} />
    </div>
  );
};

// Child Component
const TaskList = ({ tasks }) => {
  return (
    <div>
      {tasks.map(task => (
        <TaskItem
          key={task.id}
          taskName={task.name}
          isCompleted={task.completed}
        />
      ))}
    </div>
  );
};
```

1.5 Component Composition and Nesting

Composition means building complex UIs by combining simpler components, like building complex Python programs by combining functions and classes.

jsx

```jsx
// Atomic components (smallest units)
const Button = ({ onClick, children }) => (
  <button onClick={onClick}>{children}</button>
);

const Input = ({ value, onChange, placeholder }) => (
  <input
    value={value}
    onChange={onChange}
    placeholder={placeholder}
  />
);

// Molecule component (combines atoms)
const AddTaskForm = ({ onAddTask }) => {
  return (
    <form onSubmit={onAddTask}>
      <Input placeholder="Enter task..." />
      <Button>Add Task</Button>
    </form>
  );
};

// Organism component (combines molecules)
const TaskManager = () => {
  return (
    <div>
      <AddTaskForm onAddTask={handleAddTask} />
      <TaskList tasks={tasks} />
    </div>
  );
};
```

1.6 How React Processes Components Behind the Scenes

1. **Virtual DOM Creation**: React creates a virtual representation of your component tree
2. **Diffing Algorithm**: Compares new virtual DOM with previous version
3. **Reconciliation**: Updates only the changed parts in the real DOM
4. **Re-rendering**: Triggers when props or state changes

```jsx
// React's mental model:
// 1. Props change → Component re-runs → New JSX returned
// 2. React compares old JSX vs new JSX
// 3. Updates only what changed in actual DOM
```

1.7 Best Practices and Common Mistakes

✅ Best Practices:

```jsx
// 1. Always use PascalCase for component names
const TaskItem = () => { /* ... */ };

// 2. Keep components small and focused (Single Responsibility)
const TaskItem = ({ task, onToggle, onDelete }) => (
  <div>
    <span>{task.name}</span>
    <button onClick={() => onToggle(task.id)}>Toggle</button>
    <button onClick={() => onDelete(task.id)}>Delete</button>
  </div>
);

// 3. Use destructuring for cleaner props
const TaskItem = ({ task: { id, name, completed }, onToggle }) => {
  // ...
};

// 4. Always provide keys for lists
{tasks.map(task => (
  <TaskItem key={task.id} task={task} />
))}
```

❌ Common Mistakes:

```jsx
// 1. Don't modify props directly
const BadComponent = ({ tasks }) => {
  tasks.push(newTask); // ❌ Never mutate props!
  return <div>{tasks.length}</div>;
};

// 2. Don't forget keys in lists
{tasks.map(task => (
  <TaskItem task={task} /> // ❌ Missing key prop
))}

// 3. Don't use array index as key for dynamic lists
{tasks.map((task, index) => (
  <TaskItem key={index} task={task} /> // ❌ Can cause bugs
))}
```

---

2. State Management (useState)

2.1 What is State?

State is like instance variables in a Python class - it's data that belongs to a component and can change over time. When state changes, React automatically re-renders the component.

Analogy: Think of state like a variable in a Python class that, when changed, automatically calls a method to update the display.

```python
# Python analogy
class TaskManager:
    def __init__(self):
        self.tasks = []  # This is like state

    def add_task(self, task):
        self.tasks.append(task)
        self.update_display()  # Manual update needed

# React does the update_display() automatically!
```

2.2 Basic useState Syntax

```jsx
import React, { useState } from 'react';

const TaskManager = () => {
  // useState returns [currentValue, setterFunction]
  const [tasks, setTasks] = useState([]);
  const [inputValue, setInputValue] = useState('');

  // Multiple state variables
  const [isLoading, setIsLoading] = useState(false);
  const [error, setError] = useState(null);

  return (
    <div>
      <p>Total tasks: {tasks.length}</p>
      <input
        value={inputValue}
        onChange={(e) => setInputValue(e.target.value)}
      />
    </div>
  );
};
```

2.3 How State Updates Work Behind the Scenes

The React Rendering Cycle:

1. **Initial Render**: Component runs, useState returns initial value
2. **State Update**: You call setter function (e.g., `setTasks(newTasks)`)
3. **Re-render Scheduled**: React schedules a re-render
4. **Re-render**: Component function runs again with new state value
5. **DOM Update**: React updates the actual DOM with changes

jsx

```jsx
const Counter = () => {
  const [count, setCount] = useState(0);

  console.log('Component rendered with count:', count);

  const increment = () => {
    console.log('Before setState:', count);
    setCount(count + 1);
    console.log('After setState (still old value):', count); // Still 0!
  };

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

2.4 State Updates are Batched

React batches multiple state updates in the same event handler for performance:

jsx

```jsx
const TaskForm = () => {
  const [name, setName] = useState('');
  const [priority, setPriority] = useState('low');
  const [category, setCategory] = useState('personal');

  const handleSubmit = () => {
    setName('New Task');     // Batched
    setPriority('high');     // Batched
    setCategory('work');     // Batched
    // All three updates trigger only ONE re-render
  };

  return (
    <form onSubmit={handleSubmit}>
      {/* form fields */}
    </form>
  );
};
```

2.5 Async State Updates and Stale State

The Problem: State updates are asynchronous, leading to stale state issues:

```jsx
const Counter = () => {
  const [count, setCount] = useState(0);

  const badIncrement = () => {
    setCount(count + 1); // Uses current count (0)
    setCount(count + 1); // Still uses old count (0) - Result: count becomes 1, not 2!
  };

  const goodIncrement = () => {
    setCount(prev => prev + 1); // Uses previous value
    setCount(prev => prev + 1); // Uses updated previous value - Result: count becomes 2!
  };

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={badIncrement}>Bad Increment</button>
      <button onClick={goodIncrement}>Good Increment</button>
    </div>
  );
};
```

2.6 Complex State Management

Working with Objects:

jsx

```jsx
const UserProfile = () => {
  const [user, setUser] = useState({
    name: '',
    email: '',
    preferences: {
      theme: 'light',
      notifications: true
    }
  });

  const updateName = (newName) => {
    // ❌ Wrong - mutates state
    user.name = newName;
    setUser(user);

    // ✅ Correct - creates new object
    setUser(prevUser => ({
      ...prevUser,
      name: newName
    }));
  };

  const updatePreferences = (key, value) => {
    setUser(prevUser => ({
      ...prevUser,
      preferences: {
        ...prevUser.preferences,
        [key]: value
      }
    }));
  };

  return (
    <div>
      <input
        value={user.name}
        onChange={(e) => updateName(e.target.value)}
      />
    </div>
  );
};
```

Working with Arrays:

jsx

```jsx
const TaskList = () => {
  const [tasks, setTasks] = useState([]);

  const addTask = (newTask) => {
    // ✅ Add to end
    setTasks(prev => [...prev, newTask]);
  };

  const removeTask = (taskId) => {
    // ✅ Filter out task
    setTasks(prev => prev.filter(task => task.id !== taskId));
  };

  const updateTask = (taskId, updates) => {
    // ✅ Map and update specific task
    setTasks(prev =>
      prev.map(task =>
        task.id === taskId
          ? { ...task, ...updates }
          : task
      )
    );
  };

  const toggleTask = (taskId) => {
    setTasks(prev =>
      prev.map(task =>
        task.id === taskId
          ? { ...task, completed: !task.completed }
          : task
      )
    );
  };

  return (
    <div>
      {tasks.map(task => (
        <div key={task.id}>
          <span>{task.name}</span>
          <button onClick={() => toggleTask(task.id)}>
            {task.completed ? 'Undo' : 'Complete'}
          </button>
          <button onClick={() => removeTask(task.id)}>Delete</button>
        </div>
```

```
    ))}
    </div>
  );
};
```

2.7 Common Pitfalls and Debugging

Pitfall 1: Infinite Re-renders

jsx

```jsx
const BadComponent = () => {
  const [count, setCount] = useState(0);

  // ❌ This causes infinite re-renders
  setCount(count + 1);

  return <div>{count}</div>;
};

// ✅ Fix: Put state updates in event handlers or useEffect
const GoodComponent = () => {
  const [count, setCount] = useState(0);

  const handleClick = () => {
    setCount(count + 1);
  };

  return (
    <div>
      {count}
      <button onClick={handleClick}>Increment</button>
    </div>
  );
};
```

Pitfall 2: Not Using Functional Updates

```jsx
const TodoApp = () => {
  const [todos, setTodos] = useState([]);

  const addMultipleTodos = () => {
    // ❌ Wrong - both use same initial state
    setTodos([...todos, { id: 1, text: 'Task 1' }]);
    setTodos([...todos, { id: 2, text: 'Task 2' }]); // Only this one will work

    // ✅ Correct - uses functional updates
    setTodos(prev => [...prev, { id: 1, text: 'Task 1' }]);
    setTodos(prev => [...prev, { id: 2, text: 'Task 2' }]);
  };

  return (
    <button onClick={addMultipleTodos}>Add Tasks</button>
  );
};
```

---

3. Effect Hook (useEffect)

3.1 What are Side Effects?

In functional programming (like React), a **side effect** is any operation that affects something outside the function's scope or has observable interaction with the outside world.

Examples of Side Effects:

- API calls
- DOM manipulation outside React
- Timers (setTimeout, setInterval)
- Subscriptions (WebSocket connections)
- Logging to console
- Local storage operations

Analogy: Think of useEffect like Python's `__init__` method combined with cleanup methods - it handles initialization, cleanup, and reactions to changes.

3.2 Basic useEffect Syntax

jsx

```jsx
import React, { useState, useEffect } from 'react';

const TaskManager = () => {
  const [tasks, setTasks] = useState([]);

  // Basic useEffect - runs after every render
  useEffect(() => {
    console.log('Component rendered or updated');
  });

  // Effect with dependency array - runs only once
  useEffect(() => {
    console.log('Component mounted');
  }, []); // Empty dependency array

  // Effect with dependencies - runs when dependencies change
  useEffect(() => {
    console.log('Tasks changed:', tasks);
  }, [tasks]); // Runs when tasks state changes

  return <div>Task Manager</div>;
};
```

3.3 The useEffect Execution Timeline

Understanding when useEffect runs is crucial:

jsx

```jsx
const ExampleComponent = () => {
  const [count, setCount] = useState(0);

  console.log('1. Component function runs');

  useEffect(() => {
    console.log('3. useEffect runs (after DOM update)');
  });

  console.log('2. About to return JSX');

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

// Order of execution:
// 1. Component function runs
// 2. About to return JSX
// 3. DOM gets updated
// 4. useEffect runs
```

3.4 Dependency Arrays Explained

The dependency array is React's way of knowing when to re-run your effect:

jsx

```jsx
const TaskManager = () => {
  const [tasks, setTasks] = useState([]);
  const [filter, setFilter] = useState('all');
  const [user, setUser] = useState(null);

  // 1. No dependency array - runs after EVERY render
  useEffect(() => {
    console.log('Runs after every render');
  });

  // 2. Empty dependency array - runs ONLY ONCE (on mount)
  useEffect(() => {
    console.log('Runs only once when component mounts');
    fetchUserData();
  }, []);

  // 3. With dependencies - runs when ANY dependency changes
  useEffect(() => {
    console.log('Runs when tasks or filter changes');
    filterAndSortTasks();
  }, [tasks, filter]);

  // 4. Multiple effects for different concerns
  useEffect(() => {
    // Handle user-related side effects
    if (user) {
      updateUserLastSeen();
    }
  }, [user]);

  return <div>Task Manager</div>;
};
```

3.5 API Calls with useEffect

Here's how to properly make API calls in React:

jsx

```jsx
const TaskManager = () => {
  const [tasks, setTasks] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  // Fetch tasks when component mounts
  useEffect(() => {
    const fetchTasks = async () => {
      try {
        setLoading(true);
        setError(null);

        const response = await fetch('/api/tasks');
        if (!response.ok) {
          throw new Error(`HTTP error! status: ${response.status}`);
        }

        const data = await response.json();
        setTasks(data);
      } catch (err) {
        setError(err.message);
        console.error('Failed to fetch tasks:', err);
      } finally {
        setLoading(false);
      }
    };

    fetchTasks();
  }, []); // Empty dependency array - run once on mount

  // Re-fetch when specific conditions change
  useEffect(() => {
    const refetchTasks = async () => {
      // Re-fetch logic here
    };

    // Only refetch if user is logged in
    if (user?.isLoggedIn) {
      refetchTasks();
    }
  }, [user?.isLoggedIn]); // Dependency on user login status

  if (loading) return <div>Loading tasks...</div>;
  if (error) return <div>Error: {error}</div>;
```

```
  return (
    <div>
      {tasks.map(task => (
        <div key={task.id}>{task.name}</div>
      ))}
    </div>
  );
};
```

3.6 Cleanup Functions

Cleanup functions prevent memory leaks and unwanted side effects:

jsx

```
const LiveTaskUpdates = () => {
  const [tasks, setTasks] = useState([]);

  useEffect(() => {
    // Setup: Create subscription/timer/listener
    const interval = setInterval(() => {
      fetchLatestTasks().then(setTasks);
    }, 5000);

    const handleVisibilityChange = () => {
      if (document.hidden) {
        console.log('Tab is hidden, pause updates');
      }
    };

    document.addEventListener('visibilitychange', handleVisibilityChange);

    // Cleanup function - runs when:
    // 1. Component unmounts
    // 2. Dependencies change (before next effect runs)
    return () => {
      clearInterval(interval);
      document.removeEventListener('visibilitychange', handleVisibilityChange);
      console.log('Cleaned up subscriptions');
    };
  }, []); // Run once on mount

  return <div>{tasks.length} tasks</div>;
};
```

3.7 Advanced useEffect Patterns

Pattern 1: Debounced API Calls

jsx

```jsx
const SearchTasks = () => {
  const [searchTerm, setSearchTerm] = useState('');
  const [results, setResults] = useState([]);

  useEffect(() => {
    if (!searchTerm.trim()) {
      setResults([]);
      return;
    }

    // Debounce: Wait 500ms after user stops typing
    const timeoutId = setTimeout(async () => {
      try {
        const response = await fetch(`/api/search?q=${searchTerm}`);
        const data = await response.json();
        setResults(data);
      } catch (error) {
        console.error('Search failed:', error);
      }
    }, 500);

    // Cleanup: Cancel previous timeout if searchTerm changes
    return () => clearTimeout(timeoutId);
  }, [searchTerm]);

  return (
    <div>
      <input
        value={searchTerm}
        onChange={(e) => setSearchTerm(e.target.value)}
        placeholder="Search tasks..."
      />
      {results.map(task => (
        <div key={task.id}>{task.name}</div>
      ))}
    </div>
  );
};
```

Pattern 2: Conditional Effects

jsx

```jsx
const TaskSync = ({ user, isOnline }) => {
  const [tasks, setTasks] = useState([]);
  const [lastSyncTime, setLastSyncTime] = useState(null);

  // Only sync when user is logged in AND online
  useEffect(() => {
    if (!user || !isOnline) {
      return; // Early return - no cleanup needed
    }

    const syncTasks = async () => {
      try {
        const response = await fetch('/api/sync', {
          headers: { 'Authorization': `Bearer ${user.token}` }
        });
        const syncedTasks = await response.json();
        setTasks(syncedTasks);
        setLastSyncTime(new Date());
      } catch (error) {
        console.error('Sync failed:', error);
      }
    };

    syncTasks();
    const syncInterval = setInterval(syncTasks, 30000); // Sync every 30s

    return () => clearInterval(syncInterval);
  }, [user, isOnline]); // Re-run when user or online status changes

  return (
    <div>
      <p>Last sync: {lastSyncTime?.toLocaleTimeString()}</p>
      {tasks.length} tasks synced
    </div>
  );
};
```

3.8 Common Mistakes and Debugging

Mistake 1: Missing Dependencies

```jsx
const TaskManager = () => {
  const [tasks, setTasks] = useState([]);
  const [userId, setUserId] = useState(null);

  // ❌ Wrong - missing userId dependency
  useEffect(() => {
    fetchUserTasks(userId).then(setTasks);
  }, []); // ESLint will warn about this

  // ✅ Correct - include all dependencies
  useEffect(() => {
    if (userId) {
      fetchUserTasks(userId).then(setTasks);
    }
  }, [userId]);

  return <div>{tasks.length} tasks</div>;
};
```

Mistake 2: Infinite Loops

```jsx
const BadComponent = () => {
  const [data, setData] = useState([]);

  // ❌ Creates infinite loop
  useEffect(() => {
    setData([...data, 'new item']); // Changes data, triggers effect again!
  }, [data]);

  // ✅ Fixed version
  useEffect(() => {
    setData(prev => [...prev, 'new item']);
  }, []); // No dependency on data
};
```

Mistake 3: Not Handling Cleanup

```jsx
// ❌ Memory leak - subscription not cleaned up
const BadComponent = () => {
  useEffect(() => {
    const subscription = websocket.subscribe(handleMessage);
    // Missing cleanup!
  }, []);
};

// ✅ Proper cleanup
const GoodComponent = () => {
  useEffect(() => {
    const subscription = websocket.subscribe(handleMessage);

    return () => {
      subscription.unsubscribe();
    };
  }, []);
};
```

---

4. API Calls (fetch vs axios)

4.1 Understanding HTTP Requests in React

In React, API calls are side effects, so they should be made in:

1. **useEffect** for data fetching on component mount/update
2. **Event handlers** for user-triggered actions (form submissions, button clicks)

Python Analogy: It's like using the `requests` library in Python, but with additional considerations for React's rendering cycle.

4.2 Fetch API - The Native Approach

Fetch is built into modern browsers and provides a promise-based API for making HTTP requests:

jsx

```jsx
const TaskManager = () => {
  const [tasks, setTasks] = useState([]);
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState(null);

  // GET Request - Fetch all tasks
  const fetchTasks = async () => {
    try {
      setLoading(true);
      setError(null);

      const response = await fetch('/api/tasks', {
        method: 'GET',
        headers: {
          'Content-Type': 'application/json',
          'Authorization': 'Bearer your-token'
        }
      });

      // Check if response is successful
      if (!response.ok) {
        throw new Error(`HTTP error! status: ${response.status}`);
      }

      const data = await response.json();
      setTasks(data);
    } catch (err) {
      setError(err.message);
      console.error('Fetch error:', err);
    } finally {
      setLoading(false);
    }
  };

  // POST Request - Add new task
  const addTask = async (taskData) => {
    try {
      const response = await fetch('/api/tasks', {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json',
        },
        body: JSON.stringify(taskData)
      });
```

```
    if (!response.ok) {
      throw new Error(`Failed to create task: ${response.status}`);
    }

    const newTask = await response.json();
    setTasks(prev => [...prev, newTask]);
  } catch (err) {
    setError(err.message);
  }
};

// PUT Request - Update existing task
const updateTask = async (taskId, updates) => {
  try {
    const response = await fetch(`/api/tasks/${taskId}`, {
      method: 'PUT',
      headers: {
        'Content-Type': 'application/json',
      },
      body: JSON.stringify(updates)
    });

    if (!response.ok) {
      throw new Error(`Failed to update task: ${response.status}`);
    }

    const updatedTask = await response.json();
    setTasks(prev =>
      prev.map(task =>
        task.id === taskId ? updatedTask : task
      )
    );
  } catch (err) {
    setError(err.message);
  }
};

// DELETE Request - Remove task
const deleteTask = async (taskId) => {
  try {
    const response = await fetch(`/api/tasks/${taskId}`, {
      method: 'DELETE'
    });

    if (!response.ok) {
      throw new Error(`Failed to delete task: ${response.status}`);
```

```
    }

    setTasks(prev => prev.filter(task => task.id !== taskId));
  } catch (err) {
    setError(err.message);
  }
};

// Fetch tasks on component mount
useEffect(() => {
  fetchTasks();
}, []);

return (
  <div>
    {loading && <p>Loading...</p>}
    {error && <p>Error: {error}</p>}
    {tasks.map(task => (
      <div key={task.id}>
        <span>{task.name}</span>
        <button onClick={() => updateTask(task.id, { completed: !task.completed })}>
          Toggle
        </button>
        <button onClick={() => deleteTask(task.id)}>Delete</button>
      </div>
    ))}
  </div>
);
};
```

4.3 Axios - The Popular Alternative

Axios is a third-party library that provides a more feature-rich API:

jsx

```jsx
import axios from 'axios';

const TaskManagerWithAxios = () => {
  const [tasks, setTasks] = useState([]);
  const [loading, setLoading] = useState(false);

  // Configure axios defaults
  useEffect(() => {
    axios.defaults.baseURL = 'https://api.example.com';
    axios.defaults.headers.common['Authorization'] = 'Bearer your-token';

    // Request interceptor
    axios.interceptors.request.use(config => {
      console.log('Making request to:', config.url);
      return config;
    });

    // Response interceptor
    axios.interceptors.response.use(
      response => response,
      error => {
        console.error('API Error:', error.response?.data);
        return Promise.reject(error);
      }
    );
  }, []);

  // GET Request with Axios
  const fetchTasks = async () => {
    try {
      setLoading(true);
      const response = await axios.get('/tasks');
      setTasks(response.data); // Axios automatically parses JSON
    } catch (error) {
      console.error('Error fetching tasks:', error);
    } finally {
      setLoading(false);
    }
  };

  // POST Request with Axios
  const addTask = async (taskData) => {
    try {
      const response = await axios.post('/tasks', taskData);
```

```
    setTasks(prev => [...prev, response.data]);
  } catch (error) {
    if (error.response?.status === 400) {
      console.error('Validation error:', error.response.data);
    }
  }
};

// PUT Request with Axios
const updateTask = async (taskId, updates) => {
  try {
    const response = await axios.put(`/tasks/${taskId}`, updates);
    setTasks(prev =>
      prev.map(task =>
        task.id === taskId ? response.data : task
      )
    );
  } catch (error) {
    console.error('Update failed:', error);
  }
};

return <div>{/* Component JSX */}</div>;
};
```

4.4 Fetch vs Axios Comparison

Feature	Fetch	Axios
Browser Support	Modern browsers	All browsers
Size	Built-in (0kb)	~13kb
JSON Handling	Manual `.json()`	Automatic
Error Handling	Only network errors	HTTP + Network errors
Request/Response Interceptors	No	Yes
Request Timeout	Manual with AbortController	Built-in
Request Cancellation	AbortController	CancelToken

4.5 Advanced Error Handling Patterns

jsx

```jsx
const RobustTaskManager = () => {
  const [tasks, setTasks] = useState([]);
  const [error, setError] = useState(null);
  const [retryCount, setRetryCount] = useState(0);
  const [loading, setLoading] = useState(false);

  // Generic API call wrapper with retry logic
  const apiCall = async (apiFunction, maxRetries = 3) => {
    let attempt = 0;

    while (attempt < maxRetries) {
      try {
        setError(null);
        const result = await apiFunction();
        setRetryCount(0); // Reset retry count on success
        return result;
      } catch (err) {
        attempt++;
        setRetryCount(attempt);

        // Don't retry on client errors (4xx)
        if (err.response?.status >= 400 && err.response?.status < 500) {
          throw err;
        }

        // Don't retry on last attempt
        if (attempt >= maxRetries) {
          throw err;
        }

        // Exponential backoff: wait 1s, then 2s, then 4s
        await new Promise(resolve =>
          setTimeout(resolve, Math.pow(2, attempt) * 1000)
        );
      }
    }
  };

  // Fetch with retry logic
  const fetchTasks = async () => {
    try {
      setLoading(true);

      await apiCall(async () => {
```

```
      const response = await fetch('/api/tasks');
      if (!response.ok) {
        const errorData = await response.json();
        throw new Error(errorData.message || 'Failed to fetch tasks');
      }
      const data = await response.json();
      setTasks(data);
    });
  } catch (err) {
    setError({
      message: err.message,
      type: 'fetch_error',
      canRetry: err.response?.status >= 500
    });
  } finally {
    setLoading(false);
  }
};

// Specific error handling for different scenarios
const handleApiError = (error, operation) => {
  const errorMap = {
    400: 'Invalid request data',
    401: 'Please log in to continue',
    403: 'You don\'t have permission to perform this action',
    404: 'Resource not found',
    429: 'Too many requests. Please try again later',
    500: 'Server error. Please try again',
    503: 'Service temporarily unavailable'
  };

  const message = errorMap[error.response?.status] || 'An unexpected error occurred';

  setError({
    message,
    operation,
    status: error.response?.status,
    canRetry: error.response?.status >= 500
  });
};

return (
  <div>
    {loading && <p>Loading tasks...</p>}

    {error && (
      <div className="error-banner">
```

```
          <p>{error.message}</p>
          {error.canRetry && (
            <button onClick={fetchTasks}>
              Retry ({retryCount > 0 ? `Attempt ${retryCount + 1}` : 'Try Again'})
            </button>
          )}
        </div>
      )}

      {tasks.map(task => (
        <div key={task.id}>{task.name}</div>
      ))}
    </div>
  );
};
```

4.6 Request Cancellation and Cleanup

Prevent memory leaks when components unmount during API calls:

jsx

```jsx
const TaskManager = () => {
  const [tasks, setTasks] = useState([]);
  const [loading, setLoading] = useState(false);

  useEffect(() => {
    // Create AbortController for request cancellation
    const abortController = new AbortController();

    const fetchTasks = async () => {
      try {
        setLoading(true);

        const response = await fetch('/api/tasks', {
          signal: abortController.signal // Pass abort signal
        });

        if (!response.ok) {
          throw new Error('Failed to fetch');
        }

        const data = await response.json();

        // Check if component is still mounted
        if (!abortController.signal.aborted) {
          setTasks(data);
        }
      } catch (error) {
        // Ignore AbortError when component unmounts
        if (error.name !== 'AbortError') {
          console.error('Fetch error:', error);
        }
      } finally {
        if (!abortController.signal.aborted) {
          setLoading(false);
        }
      }
    };

    fetchTasks();

    // Cleanup: Cancel request if component unmounts
    return () => {
      abortController.abort();
    };
```

```
  }, []);

  return (
    <div>
      {loading ? <p>Loading...</p> : tasks.map(task =>
        <div key={task.id}>{task.name}</div>
      )}
    </div>
  );
};
```

4.7 Custom Hooks for API Calls

Create reusable hooks for common API patterns:

jsx

```jsx
// Custom hook for data fetching
const useApi = (url, options = {}) => {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState(null);

  const execute = async (overrideUrl) => {
    try {
      setLoading(true);
      setError(null);

      const response = await fetch(overrideUrl || url, {
        headers: { 'Content-Type': 'application/json' },
        ...options
      });

      if (!response.ok) {
        throw new Error(`HTTP error! status: ${response.status}`);
      }

      const result = await response.json();
      setData(result);
      return result;
    } catch (err) {
      setError(err.message);
      throw err;
    } finally {
      setLoading(false);
    }
  };

  return { data, loading, error, execute };
};

// Usage in components
const TaskList = () => {
  const {
    data: tasks,
    loading,
    error,
    execute: fetchTasks
  } = useApi('/api/tasks');

  const { execute: createTask } = useApi('/api/tasks', { method: 'POST' });
```

```
  // Fetch tasks on mount
  useEffect(() => {
    fetchTasks();
  }, []);

  const handleAddTask = async (taskData) => {
    try {
      await createTask('/api/tasks', {
        method: 'POST',
        body: JSON.stringify(taskData)
      });
      // Refetch tasks after adding
      fetchTasks();
    } catch (error) {
      console.error('Failed to add task:', error);
    }
  };

  if (loading) return <div>Loading...</div>;
  if (error) return <div>Error: {error}</div>;

  return (
    <div>
      {tasks?.map(task => (
        <div key={task.id}>{task.name}</div>
      ))}
    </div>
  );
};
```

5. Forms & Event Handling

5.1 Understanding React Forms

React forms work differently from traditional HTML forms. Instead of letting the browser manage form state, React takes control through **controlled components**.

Analogy: Traditional HTML forms are like self-managing objects, while React forms are like objects where you explicitly control every property change.

5.2 Controlled vs Uncontrolled Components

Controlled Components (Recommended)

React controls the form input value through state:

jsx

```jsx
const ControlledForm = () => {
  const [formData, setFormData] = useState({
    taskName: '',
    priority: 'medium',
    dueDate: '',
    description: ''
  });

  const handleInputChange = (e) => {
    const { name, value, type, checked } = e.target;

    setFormData(prev => ({
      ...prev,
      [name]: type === 'checkbox' ? checked : value
    }));
  };

  const handleSubmit = (e) => {
    e.preventDefault(); // Prevent page reload
    console.log('Form submitted:', formData);

    // Process form data
    createTask(formData);

    // Reset form
    setFormData({
      taskName: '',
      priority: 'medium',
      dueDate: '',
      description: ''
    });
  };

  return (
    <form onSubmit={handleSubmit}>
      <div>
        <label htmlFor="taskName">Task Name:</label>
        <input
          type="text"
          id="taskName"
          name="taskName"
          value={formData.taskName} // Controlled by React state
          onChange={handleInputChange}
          required
```

```
      />
    </div>

    <div>
      <label htmlFor="priority">Priority:</label>
      <select
        id="priority"
        name="priority"
        value={formData.priority}
        onChange={handleInputChange}
      >
        <option value="low">Low</option>
        <option value="medium">Medium</option>
        <option value="high">High</option>
      </select>
    </div>

    <div>
      <label htmlFor="dueDate">Due Date:</label>
      <input
        type="date"
        id="dueDate"
        name="dueDate"
        value={formData.dueDate}
        onChange={handleInputChange}
      />
    </div>

    <div>
      <label htmlFor="description">Description:</label>
      <textarea
        id="description"
        name="description"
        value={formData.description}
        onChange={handleInputChange}
        rows={4}
      />
    </div>

    <button type="submit">Add Task</button>
  </form>
 );
};
```

Uncontrolled Components (Less Common)

The DOM handles the form state, React just reads values when needed:

jsx

```jsx
import { useRef } from 'react';

const UncontrolledForm = () => {
  const formRef = useRef();
  const taskNameRef = useRef();
  const priorityRef = useRef();

  const handleSubmit = (e) => {
    e.preventDefault();

    // Get values directly from DOM
    const formData = {
      taskName: taskNameRef.current.value,
      priority: priorityRef.current.value
    };

    console.log('Uncontrolled form data:', formData);

    // Reset form
    formRef.current.reset();
  };

  return (
    <form ref={formRef} onSubmit={handleSubmit}>
      <input
        ref={taskNameRef}
        type="text"
        placeholder="Task name"
        defaultValue="" // Use defaultValue, not value
      />

      <select ref={priorityRef} defaultValue="medium">
        <option value="low">Low</option>
        <option value="medium">Medium</option>
        <option value="high">High</option>
      </select>

      <button type="submit">Add Task</button>
    </form>
  );
};
```

5.3 Event Handling Deep Dive

React uses **SyntheticEvents** - a wrapper around native DOM events that provides consistent behavior across browsers:

jsx

```jsx
const EventHandlingExamples = () => {
  const [message, setMessage] = useState('');

  // Basic event handler
  const handleClick = (e) => {
    console.log('Event object:', e);
    console.log('Native event:', e.nativeEvent);
    console.log('Target element:', e.target);
    console.log('Current target:', e.currentTarget);
  };

  // Event handler with parameters
  const handleButtonClick = (taskId, action) => {
    return (e) => {
      e.stopPropagation(); // Prevent event bubbling
      console.log(`${action} task ${taskId}`);
    };
  };

  // Input event handler
  const handleInputChange = (e) => {
    const { name, value, type } = e.target;
    console.log(`Input ${name} changed to: ${value}`);
    setMessage(value);
  };

  // Keyboard event handler
  const handleKeyDown = (e) => {
    if (e.key === 'Enter' && e.ctrlKey) {
      console.log('Ctrl+Enter pressed');
      e.preventDefault();
      // Submit form or perform action
    }

    if (e.key === 'Escape') {
      setMessage('');
    }
  };

  // Focus/blur event handlers
  const handleFocus = (e) => {
    console.log('Input focused');
    e.target.style.backgroundColor = '#f0f0f0';
  };
```

```
  const handleBlur = (e) => {
    console.log('Input blurred');
    e.target.style.backgroundColor = '';
  };

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

      <button onClick={handleButtonClick('task-1', 'delete')}>
        Delete Task 1
      </button>

      <input
        name="message"
        value={message}
        onChange={handleInputChange}
        onKeyDown={handleKeyDown}
        onFocus={handleFocus}
        onBlur={handleBlur}
        placeholder="Type something..."
      />

      <p>Message: {message}</p>
    </div>
  );
};
```

5.4 Form Validation

Client-Side Validation

jsx

```jsx
const ValidatedTaskForm = () => {
  const [formData, setFormData] = useState({
    taskName: '',
    email: '',
    dueDate: ''
  });

  const [errors, setErrors] = useState({});
  const [touched, setTouched] = useState({});

  // Validation rules
  const validateField = (name, value) => {
    switch (name) {
      case 'taskName':
        if (!value.trim()) return 'Task name is required';
        if (value.length < 3) return 'Task name must be at least 3 characters';
        return '';

      case 'email':
        const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
        if (!value.trim()) return 'Email is required';
        if (!emailRegex.test(value)) return 'Please enter a valid email';
        return '';

      case 'dueDate':
        if (!value) return 'Due date is required';
        if (new Date(value) < new Date()) return 'Due date cannot be in the past';
        return '';

      default:
        return '';
    }
  };

  // Validate entire form
  const validateForm = () => {
    const newErrors = {};
    Object.keys(formData).forEach(field => {
      const error = validateField(field, formData[field]);
      if (error) newErrors[field] = error;
    });
    setErrors(newErrors);
    return Object.keys(newErrors).length === 0;
  };
```

```
const handleInputChange = (e) => {
  const { name, value } = e.target;

  // Update form data
  setFormData(prev => ({
    ...prev,
    [name]: value
  }));

  // Clear error when user starts typing
  if (touched[name] && errors[name]) {
    const error = validateField(name, value);
    setErrors(prev => ({
      ...prev,
      [name]: error
    }));
  }
};

const handleBlur = (e) => {
  const { name, value } = e.target;

  // Mark field as touched
  setTouched(prev => ({
    ...prev,
    [name]: true
  }));

  // Validate on blur
  const error = validateField(name, value);
  setErrors(prev => ({
    ...prev,
    [name]: error
  }));
};

const handleSubmit = async (e) => {
  e.preventDefault();

  // Mark all fields as touched
  const allTouched = Object.keys(formData).reduce((acc, key) => {
    acc[key] = true;
    return acc;
  }, {});
  setTouched(allTouched);
```

```
  // Validate entire form
  if (!validateForm()) {
    console.log('Form has errors');
    return;
  }

  try {
    // Submit form data
    await createTask(formData);
    console.log('Task created successfully');

    // Reset form
    setFormData({ taskName: '', email: '', dueDate: '' });
    setErrors({});
    setTouched({});
  } catch (error) {
    console.error('Failed to create task:', error);
  }
};

return (
  <form onSubmit={handleSubmit} noValidate>
    <div className="field">
      <label htmlFor="taskName">Task Name:</label>
      <input
        type="text"
        id="taskName"
        name="taskName"
        value={formData.taskName}
        onChange={handleInputChange}
        onBlur={handleBlur}
        className={errors.taskName ? 'error' : ''}
      />
      {touched.taskName && errors.taskName && (
        <span className="error-message">{errors.taskName}</span>
      )}
    </div>

    <div className="field">
      <label htmlFor="email">Email:</label>
      <input
        type="email"
        id="email"
        name="email"
        value={formData.email}
        onChange={handleInputChange}
```

```
        onBlur={handleBlur}
        className={errors.email ? 'error' : ''}
      />
      {touched.email && errors.email && (
        <span className="error-message">{errors.email}</span>
      )}
    </div>

    <div className="field">
      <label htmlFor="dueDate">Due Date:</label>
      <input
        type="date"
        id="dueDate"
        name="dueDate"
        value={formData.dueDate}
        onChange={handleInputChange}
        onBlur={handleBlur}
        className={errors.dueDate ? 'error' : ''}
      />
      {touched.dueDate && errors.dueDate && (
        <span className="error-message">{errors.dueDate}</span>
      )}
    </div>

    <button
      type="submit"
      disabled={Object.keys(errors).some(key => errors[key])}
    >
      Create Task
    </button>
  </form>
  );
};
```

5.5 Advanced Form Patterns

Dynamic Forms

jsx

```jsx
const DynamicTaskForm = () => {
  const [tasks, setTasks] = useState([
    { id: 1, name: '', priority: 'medium' }
  ]);

  const addTask = () => {
    const newTask = {
      id: Date.now(),
      name: '',
      priority: 'medium'
    };
    setTasks(prev => [...prev, newTask]);
  };

  const removeTask = (taskId) => {
    setTasks(prev => prev.filter(task => task.id !== taskId));
  };

  const updateTask = (taskId, field, value) => {
    setTasks(prev =>
      prev.map(task =>
        task.id === taskId
          ? { ...task, [field]: value }
          : task
      )
    );
  };

  const handleSubmit = (e) => {
    e.preventDefault();
    console.log('All tasks:', tasks);
  };

  return (
    <form onSubmit={handleSubmit}>
      <h3>Task List</h3>

      {tasks.map((task, index) => (
        <div key={task.id} className="task-row">
          <span>Task {index + 1}:</span>

          <input
            type="text"
            value={task.name}
```

```
            onChange={(e) => updateTask(task.id, 'name', e.target.value)}
            placeholder="Task name"
          />

          <select
            value={task.priority}
            onChange={(e) => updateTask(task.id, 'priority', e.target.value)}
          >
            <option value="low">Low</option>
            <option value="medium">Medium</option>
            <option value="high">High</option>
          </select>

          {tasks.length > 1 && (
            <button
              type="button"
              onClick={() => removeTask(task.id)}
            >
              Remove
            </button>
          )}
        </div>
      ))}

      <button type="button" onClick={addTask}>
        Add Another Task
      </button>

      <button type="submit">Submit All Tasks</button>
    </form>
  );
};
```

5.6 Form with File Upload

jsx

```jsx
const TaskFormWithFiles = () => {
  const [formData, setFormData] = useState({
    taskName: '',
    attachments: []
  });

  const handleFileChange = (e) => {
    const files = Array.from(e.target.files);

    // Validate files
    const validFiles = files.filter(file => {
      const isValidType = ['image/jpeg', 'image/png', 'application/pdf'].includes(file.type);
      const isValidSize = file.size <= 5 * 1024 * 1024; // 5MB limit

      if (!isValidType) {
        alert(`Invalid file type: ${file.name}`);
        return false;
      }

      if (!isValidSize) {
        alert(`File too large: ${file.name}`);
        return false;
      }

      return true;
    });

    setFormData(prev => ({
      ...prev,
      attachments: [...prev.attachments, ...validFiles]
    }));
  };

  const removeFile = (fileIndex) => {
    setFormData(prev => ({
      ...prev,
      attachments: prev.attachments.filter((_, index) => index !== fileIndex)
    }));
  };

  const handleSubmit = async (e) => {
    e.preventDefault();

    // Create FormData for file upload
```

```
  const submitData = new FormData();
  submitData.append('taskName', formData.taskName);

  formData.attachments.forEach((file, index) => {
    submitData.append(`attachment${index}`, file);
  });

  try {
    const response = await fetch('/api/tasks', {
      method: 'POST',
      body: submitData // Don't set Content-Type header for FormData
    });

    if (response.ok) {
      console.log('Task created with attachments');
      setFormData({ taskName: '', attachments: [] });
    }
  } catch (error) {
    console.error('Upload failed:', error);
  }
};

return (
  <form onSubmit={handleSubmit}>
    <div>
      <label>Task Name:</label>
      <input
        type="text"
        value={formData.taskName}
        onChange={(e) => setFormData(prev => ({
          ...prev,
          taskName: e.target.value
        }))}
      />
    </div>

    <div>
      <label>Attachments:</label>
      <input
        type="file"
        multiple
        accept="image/*,.pdf"
        onChange={handleFileChange}
      />
    </div>

    {formData.attachments.length > 0 && (
```

```
      <div>
        <h4>Selected Files:</h4>
        {formData.attachments.map((file, index) => (
          <div key={index} className="file-item">
            <span>{file.name} ({(file.size / 1024).toFixed(1)} KB)</span>
            <button
              type="button"
              onClick={() => removeFile(index)}
            >
              Remove
            </button>
          </div>
        ))}
      </div>
    )}

    <button type="submit">Create Task</button>
  </form>
 );
};
```

5.7 Common Form Mistakes and Best Practices

❌ **Common Mistakes:**

jsx

```jsx
// 1. Forgetting to prevent default form submission
const BadForm = () => {
  const handleSubmit = (e) => {
    // ❌ Page will reload without e.preventDefault()
    console.log('Form submitted');
  };

  return <form onSubmit={handleSubmit}>...</form>;
};

// 2. Directly mutating state
const BadForm2 = () => {
  const [formData, setFormData] = useState({ name: '', items: [] });

  const addItem = () => {
    formData.items.push('new item'); // ❌ Direct mutation
    setFormData(formData); // Won't trigger re-render
  };
};

// 3. Not handling edge cases
const BadForm3 = () => {
  const [email, setEmail] = useState('');

  const handleSubmit = (e) => {
    e.preventDefault();
    // ❌ No validation or error handling
    submitEmail(email);
  };
};
```

✅ **Best Practices:**

jsx

```jsx
const GoodForm = () => {
  const [formData, setFormData] = useState({
    name: '',
    email: '',
    items: []
  });
  const [isSubmitting, setIsSubmitting] = useState(false);
  const [errors, setErrors] = useState({});

  const handleSubmit = async (e) => {
    e.preventDefault(); // ✅ Always prevent default

    // ✅ Validate before submitting
    if (!validateForm()) return;

    try {
      setIsSubmitting(true); // ✅ Show loading state
      await submitForm(formData);

      // ✅ Reset form after successful submission
      setFormData({ name: '', email: '', items: [] });
      setErrors({});
    } catch (error) {
      // ✅ Handle errors gracefully
      setErrors({ submit: 'Failed to submit form' });
    } finally {
      setIsSubmitting(false);
    }
  };

  const addItem = () => {
    // ✅ Immutable state update
    setFormData(prev => ({
      ...prev,
      items: [...prev.items, 'new item']
    }));
  };

  return (
    <form onSubmit={handleSubmit}>
      {/* Form fields with proper validation */}
      <button
        type="submit"
        disabled={isSubmitting || Object.keys(errors).length > 0}
```

```
      >
        {isSubmitting ? 'Submitting...' : 'Submit'}
      </button>
    </form>
  );
};
```

Summary and Next Steps

You've now learned the core React concepts that every frontend developer needs to master:

1. **Component-Based Architecture**: Breaking UIs into reusable, composable pieces
2. **State Management (useState)**: Managing component data and triggering re-renders
3. **Effect Hook (useEffect)**: Handling side effects and component lifecycle
4. **API Calls**: Fetching and sending data with proper error handling
5. **Forms & Event Handling**: Creating interactive, validated user interfaces

Key Takeaways:

- **Think in Components**: Break your UI into small, focused, reusable pieces
- **State Drives UI**: When state changes, React automatically updates the display
- **Effects Handle Side Effects**: Use useEffect for API calls, subscriptions, and cleanup
- **Controlled Components**: Let React manage form state for better control and validation
- **Always Handle Errors**: Network requests can fail, so always have error handling

Practice Exercises:

1. Build a complete Todo app with add/edit/delete/filter functionality
2. Create a user registration form with validation
3. Implement a search feature with debounced API calls
4. Build a dashboard that fetches data from multiple APIs

These fundamentals will serve as the foundation for more advanced React concepts like custom hooks, context API, and state management libraries. Keep practicing, and remember that React's power comes from combining these simple concepts in sophisticated ways!