

Complete JavaScript and React.js Learning Path: Beginner to Professional

Introduction

Welcome to your comprehensive guide for mastering JavaScript and React.js! This curriculum is designed as a progressive journey that will take you from fundamental concepts to professional-level understanding. Each section builds upon the previous one, creating a solid foundation of knowledge and skills.

Whether you're refreshing your memory or filling in knowledge gaps, this guide will help you develop both conceptual understanding and practical implementation skills. Let's begin!

Learning Path Overview

1. **JavaScript Fundamentals** - Core language features and syntax
 2. **Advanced JavaScript** - Deeper concepts including asynchronous programming
 3. **Modern JavaScript** - ES6+ features and contemporary practices
 4. **Browser JavaScript** - DOM manipulation and Web APIs
 5. **JavaScript Tools & Practices** - Development environment, testing, and optimization
 6. **React Fundamentals** - Core React concepts and component architecture
 7. **React State Management** - Approaches from simple to complex
 8. **Advanced React Patterns** - Professional techniques and patterns
 9. **React Ecosystem** - Routing, data fetching, and related tools
 10. **Professional React Development** - Testing, optimization, and deployment
-

PART 1: JAVASCRIPT FUNDAMENTALS

JavaScript Foundations

Conceptual Foundations

JavaScript is a high-level, interpreted programming language that conforms to the ECMAScript specification. Originally created to make web pages interactive, it has evolved into a versatile language that powers complex web applications, server-side development, mobile apps, and more.

Mental Model: Think of JavaScript as a bridge between static web pages and dynamic user interactions. If HTML is the skeleton and CSS is the appearance, JavaScript is the nervous system that enables a website to respond to user actions.

Historical Context: Created in 1995 by Brendan Eich at Netscape, JavaScript was designed in just 10 days. Despite its name, it's not related to Java. It was originally called Mocha, then LiveScript, before being renamed JavaScript as a marketing decision. It's standardized as ECMAScript (ES), with major revisions including ES5 (2009), ES6/ES2015, and yearly updates since.

Common Misconceptions:

- JavaScript is NOT related to Java beyond the name
- JavaScript is NOT just for simple website animations - it's a full-featured programming language
- Modern JavaScript is NOT slow or inefficient - browser engines have become highly optimized
- JavaScript is NOT poorly designed - while it has quirks from its rapid development, modern JavaScript has addressed many early criticisms

Setting Up Your Environment

To begin working with JavaScript, you need a minimal setup:

1. **Text Editor or IDE:** Visual Studio Code is highly recommended

- Download from: <https://code.visualstudio.com/>
- Recommended extensions:
 - ESLint (linting)
 - Prettier (code formatting)
 - JavaScript (ES6) code snippets
 - Live Server (for running HTML/JS files locally)

2. **Browser with Developer Tools:**

- Chrome, Firefox, or Edge all have excellent developer tools
- Access by right-clicking on a page and selecting "Inspect" or pressing F12

3. **Node.js (for running JavaScript outside the browser):**

- Download from: <https://nodejs.org/>
- This includes npm (Node Package Manager) for installing JavaScript libraries

First JavaScript Program:

Create a file named `hello.js`:

```
// My first JavaScript program
console.log('Hello, World!');

// Variables and basic operations
let name = 'JavaScript Learner';
let daysStudying = 1;
let message = `Hello, ${name}! You've been studying for ${daysStudying} day(s).`;

console.log(message);

// This will output:
// Hello, World!
// Hello, JavaScript Learner! You've been studying for 1 day(s).
```

Run this code:

- In the browser: Create an HTML file, include this script, and open it

- In Node.js: Open a terminal, navigate to the file location, and run `node hello.js`

Variables, Data Types, and Operators

Conceptual Foundations

Variables in JavaScript are containers for storing data values. Unlike some languages, JavaScript is dynamically typed, meaning variables can hold different types of data throughout their lifecycle.

Mental Model: Think of variables as labeled boxes that can hold different types of content. The box (variable) stays the same, but what you put inside it (the value) can change.

Common Misconceptions:

- JavaScript doesn't require declaring variable types - it infers them
- Variable declarations with `var` have function scope, while `let` and `const` have block scope
- `const` prevents reassignment but doesn't make objects immutable

Practical Implementation

Variable Declaration:

```
// Three ways to declare variables
var oldWay = 'This has function scope'; // Pre-ES6, avoid when possible
let modern = 'This has block scope'; // Preferred for variables that change
const constant = 'This cannot be reassigned'; // Preferred when possible

// Variable naming follows camelCase by convention
let firstName = 'John';
let isActive = true;

// This will throw an error:
// const myValue = 5;
// myValue = 10; // Error: Assignment to constant variable

// But objects and arrays declared with const can still be modified:
const person = { name: 'Alice' };
person.name = 'Bob'; // This works - the object reference hasn't changed
// person = { name: "Charlie" }; // This would error - reassignment not allowed
```

Data Types:

```
// Primitive data types:

// 1. String - text data
let name = 'JavaScript';
let singleQuotes = 'Also works';
let templateLiteral = `Hello, ${name}`; // Template literals allow embedded
expressions
```

```
// 2. Number - numeric data (no separate integer/float types)
let integer = 42;
let float = 3.14;
let scientific = 2.998e8; // Scientific notation
let infinity = Infinity;
let notANumber = NaN; // Result of invalid operations

// 3. Boolean - logical data
let isTrue = true;
let isFalse = false;

// 4. Undefined - default value of uninitialized variables
let undefinedVar;
console.log(undefinedVar); // undefined

// 5. Null - intentional absence of value
let emptyValue = null;

// 6. Symbol - unique identifiers (ES6)
let uniqueId = Symbol('description');

// 7. BigInt - for integers larger than Number can represent (ES2020)
let hugeNumber = 9007199254740991n;

// Object - collection of properties
let person = {
  name: 'Alice',
  age: 30,
  isStudent: false,
};

// Array - ordered collection (specialized object)
let colors = ['red', 'green', 'blue'];

// Function - callable object
let greet = function () {
  return 'Hello!';
};

// Checking types
console.log(typeof 'string'); // "string"
console.log(typeof 42); // "number"
console.log(typeof true); // "boolean"
console.log(typeof undefined); // "undefined"
console.log(typeof null); // "object" (this is a historical bug)
console.log(typeof {}); // "object"
console.log(typeof []); // "object" (arrays are objects)
console.log(typeof function () {}); // "function"

// Type coercion - JavaScript will convert types automatically
console.log('5' + 2); // "52" (number converted to string)
console.log('5' - 2); // 3 (string converted to number)
console.log(5 + true); // 6 (true converted to 1)
console.log(5 + false); // 5 (false converted to 0)
```

```
console.log(Boolean(0)); // false
console.log(Boolean('')); // false
console.log(Boolean('0')); // true (non-empty string)
```

Operators:

```
// Arithmetic operators
let sum = 5 + 3; // 8
let difference = 5 - 3; // 2
let product = 5 * 3; // 15
let quotient = 5 / 3; // 1.6666...
let remainder = 5 % 3; // 2
let exponent = 5 ** 3; // 125 (53)

// Increment and decrement
let count = 0;
count++; // count is now 1
count--; // count is now 0

// Assignment operators
let x = 5;
x += 3; // x = x + 3 (8)
x -= 2; // x = x - 2 (6)
x *= 4; // x = x * 4 (24)
x /= 3; // x = x / 3 (8)

// Comparison operators
console.log(5 == '5'); // true (equal value, different type)
console.log(5 === '5'); // false (strict equality - different type)
console.log(5 != '5'); // false (not equal value)
console.log(5 !== '5'); // true (strict inequality - different type)
console.log(5 > 3); // true
console.log(5 >= 5); // true
console.log(3 < 5); // true
console.log(3 <= 3); // true

// Logical operators
console.log(true && false); // false (logical AND)
console.log(true || false); // true (logical OR)
console.log(!true); // false (logical NOT)

// Optional chaining (ES2020)
const user = {
  profile: {
    name: 'Alice',
  },
};
console.log(user.profile?.name); // "Alice"
console.log(user.settings?.theme); // undefined (no error)

// Nullish coalescing (ES2020)
const value = null;
```

```
const defaultValue = value ?? 'default'; // "default"
// Different from || which returns fallback for any falsy value:
console.log('' || 'fallback'); // "fallback"
console.log('' ?? 'fallback'); // "" (empty string is not nullish)
```

Control Flow

Conceptual Foundations

Control flow determines the order in which statements are executed in a program. JavaScript provides several constructs to control this flow based on conditions, allowing for decision-making and repetition.

Mental Model: Think of control flow as a roadmap with various paths and decision points. The program follows different routes based on conditions it encounters, like a traveler following signs at intersections.

Common Misconceptions:

- The `switch` statement uses strict equality (==) for comparisons
- `break` is usually necessary in `switch` cases to prevent fall-through
- The difference between `for...in` (for object properties) and `for...of` (for iterable values)

Practical Implementation

Conditional Statements:

```
// Basic if statement
let temperature = 75;

if (temperature > 80) {
  console.log("It's hot outside!");
}

// if...else statement
if (temperature > 80) {
  console.log("It's hot outside!");
} else {
  console.log("It's not too hot today.");
}

// if...else if...else chain
if (temperature > 80) {
  console.log("It's hot outside!");
} else if (temperature > 60) {
  console.log("It's a nice day!");
} else if (temperature > 40) {
  console.log("It's a bit chilly.");
} else {
  console.log("It's cold outside!");
}

// Ternary operator - compact conditional expression
```

```
let message = temperature > 80 ? "It's hot!" : "It's not too hot.";
console.log(message);

// Switch statement - evaluates an expression against multiple cases
let day = 'Monday';

switch (day) {
  case 'Monday':
    console.log('Start of the work week');
    break;
  case 'Friday':
    console.log('End of the work week');
    break;
  case 'Saturday':
  case 'Sunday':
    console.log('Weekend!');
    break;
  default:
    console.log('Midweek');
    break;
}

// Without break, execution "falls through" to the next case
let fallThroughExample = 'A';

switch (fallThroughExample) {
  case 'A':
    console.log('A was matched');
    // No break, so execution continues to case "B"
  case 'B':
    console.log('B code executed');
    break;
  case 'C':
    console.log('C code executed');
    break;
}
// Output:
// "A was matched"
// "B code executed"
```

Loops and Iteration:

```
// Basic for loop
for (let i = 0; i < 5; i++) {
  console.log(`Iteration ${i}`);
}

// Loop with multiple variables
for (let i = 0, j = 10; i < 5; i++, j--) {
  console.log(`i = ${i}, j = ${j}`);
}
```

```
// while loop - executes as long as condition is true
let count = 0;
while (count < 5) {
  console.log(`Count is ${count}`);
  count++;
}

// do...while loop - always executes at least once
let num = 10;
do {
  console.log(`Number is ${num}`);
  num--;
} while (num > 5);

// Breaking out of loops
for (let i = 0; i < 10; i++) {
  if (i === 5) {
    break; // Immediately exit the loop
  }
  console.log(`Iteration ${i}`);
}

// Skipping iterations
for (let i = 0; i < 10; i++) {
  if (i % 2 === 0) {
    continue; // Skip to next iteration
  }
  console.log(`Odd number: ${i}`);
}

// for...in loop - iterates over enumerable properties of an object
const person = {
  name: 'Alice',
  age: 30,
  occupation: 'Engineer',
};

for (let key in person) {
  console.log(`${key}: ${person[key]}`);
}

// for...of loop - iterates over iterable objects (arrays, strings, etc.)
const colors = ['red', 'green', 'blue'];

for (let color of colors) {
  console.log(color);
}

// Iterating over a string
for (let char of 'Hello') {
  console.log(char);
}

// forEach method for arrays
```

```
colors.forEach((color, index) => {
  console.log(`Color at position ${index}: ${color}`);
});
```

Truthy and Falsy Values:

```
// In JavaScript, values are inherently truthy or falsy
// Falsy values:
console.log(Boolean(false)); // false
console.log(Boolean(0)); // false
console.log(Boolean('')); // false
console.log(Boolean(null)); // false
console.log(Boolean(undefined)); // false
console.log(Boolean(NaN)); // false

// Everything else is truthy:
console.log(Boolean(true)); // true
console.log(Boolean(1)); // true
console.log(Boolean('hello'))); // true
console.log(Boolean([])); // true
console.log(Boolean({})); // true
console.log(Boolean(function () {})); // true

// This allows for shorthand conditionals:
let username = '';
if (username) {
  console.log('Username is provided');
} else {
  console.log('Username is empty');
}

// Short-circuit evaluation
// In OR (||) expressions, returns first truthy value or last value
let defaultName = username || 'Guest';
console.log(defaultName); // "Guest"

// In AND (&&) expressions, returns first falsy value or last value
let isAdult = true;
let canProceed = isAdult && username; // false (username is "")
console.log(canProceed);
```

Functions

Conceptual Foundations

Functions are one of the fundamental building blocks in JavaScript. A function is a reusable block of code designed to perform a particular task. Functions allow for code organization, reusability, and abstraction.

Mental Model: Think of functions as recipes. They take ingredients (parameters), follow a set of instructions (the function body), and produce a dish (the return value). You can use the same recipe multiple times with

different ingredients to create variations of the dish.

Historical Context: JavaScript functions have evolved significantly. Originally, they were primarily defined using function declarations and expressions. ES6 introduced arrow functions, providing a more concise syntax and lexical `this` binding.

Common Misconceptions:

- Functions don't have to return a value explicitly (they return `undefined` by default)
- Arrow functions don't have their own `this` context, unlike traditional functions
- Function parameters are local variables, even if they have the same name as variables outside the function
- Function declarations are hoisted, while function expressions are not

Practical Implementation

Function Declarations and Expressions:

```
// Function declaration
function greet(name) {
  return `Hello, ${name}!`;
}

console.log(greet('Alice')); // "Hello, Alice!"

// Function expression
const sayGoodbye = function (name) {
  return `Goodbye, ${name}!`;
};

console.log(sayGoodbye('Bob')); // "Goodbye, Bob!"

// Arrow function (ES6)
const welcome = (name) => {
  return `Welcome, ${name}!`;
};

// Simplified arrow function (for single expressions)
const welcome2 = (name) => `Welcome, ${name}!`;

console.log(welcome('Charlie')); // "Welcome, Charlie!"
console.log(welcome2('Diana')); // "Welcome, Diana!"

// Function with default parameters (ES6)
function greetWithDefault(name = 'Guest') {
  return `Hello, ${name}!`;
}

console.log(greetWithDefault()); // "Hello, Guest!"
console.log(greetWithDefault('Edward')); // "Hello, Edward!"

// Rest parameters (ES6) - collect remaining arguments into an array
```

```
function sum(...numbers) {
  return numbers.reduce((total, num) => total + num, 0);
}

console.log(sum(1, 2, 3, 4, 5)); // 15

// Function that returns another function (Higher-order function)
function createMultiplier(factor) {
  return function (number) {
    return number * factor;
  };
}

const double = createMultiplier(2);
const triple = createMultiplier(3);

console.log(double(5)); // 10
console.log(triple(5)); // 15

// Immediately Invoked Function Expression (IIFE)
// Creates a private scope that doesn't pollute the global namespace
(function () {
  const privateVar = 'I am private';
  console.log(privateVar);
})();

// This would result in an error - privateVar is not accessible:
// console.log(privateVar);
```

Function Parameters and Arguments:

```
// Basic parameters
function add(a, b) {
  return a + b;
}

console.log(add(5, 3)); // 8

// More arguments than parameters
function logFirstTwo(a, b) {
  console.log(a, b);
}

logFirstTwo(1, 2, 3, 4); // 1 2 (extra arguments are ignored)

// Fewer arguments than parameters
function multiply(a, b, c) {
  return a * b * c;
}

console.log(multiply(2, 3)); // NaN (missing parameter is undefined)
```

```
// Using the arguments object (older approach)
function sum() {
  let total = 0;
  for (let i = 0; i < arguments.length; i++) {
    total += arguments[i];
  }
  return total;
}

console.log(sum(1, 2, 3, 4)); // 10

// Destructuring parameters (ES6)
function displayPerson({ name, age, occupation = 'Unknown' }) {
  console.log(` ${name} is ${age} years old and works as ${occupation}`);
}

const person = {
  name: 'Alice',
  age: 30,
  occupation: 'Engineer',
};

displayPerson(person); // "Alice is 30 years old and works as Engineer"
displayPerson({ name: 'Bob', age: 25 }); // "Bob is 25 years old and works as Unknown"
```

Closures:

```
// A closure is a function that remembers its lexical scope,
// even when the function is executed outside that scope.

function createCounter() {
  let count = 0; // This variable is enclosed in the closure

  return function () {
    count += 1;
    return count;
  };
}

const counter = createCounter();
console.log(counter()); // 1
console.log(counter()); // 2
console.log(counter()); // 3

// Each call to createCounter() creates a new closure with its own count variable
const counter2 = createCounter();
console.log(counter2()); // 1 (independent of the first counter)

// Practical example: Creating private variables
function createBankAccount(initialBalance) {
  let balance = initialBalance; // Private variable
```

```

return {
  deposit: function (amount) {
    balance += amount;
    return `Deposited ${amount}. New balance: ${balance}`;
  },
  withdraw: function (amount) {
    if (amount > balance) {
      return 'Insufficient funds';
    }
    balance -= amount;
    return `Withdrew ${amount}. New balance: ${balance}`;
  },
  getBalance: function () {
    return `Current balance: ${balance}`;
  },
};
}

const account = createBankAccount(100);
console.log(account.getBalance()); // "Current balance: 100"
console.log(account.deposit(50)); // "Deposited 50. New balance: 150"
console.log(account.withdraw(30)); // "Withdrew 30. New balance: 120"
console.log(account.withdraw(200)); // "Insufficient funds"

// balance is not directly accessible:
// console.log(account.balance); // undefined

```

Higher-Order Functions:

```

// Higher-order functions either:
// 1. Accept functions as arguments
// 2. Return functions as results
// 3. Or both

// Array's built-in HOFs:

// map() - Creates a new array by transforming each element
const numbers = [1, 2, 3, 4, 5];
const doubled = numbers.map((num) => num * 2);
console.log(doubled); // [2, 4, 6, 8, 10]

// filter() - Creates a new array with elements that pass a test
const evens = numbers.filter((num) => num % 2 === 0);
console.log(evens); // [2, 4]

// reduce() - Accumulates values into a single result
const sum = numbers.reduce((total, num) => total + num, 0);
console.log(sum); // 15

// forEach() - Executes a function on each element
numbers.forEach((num) => console.log(`Number: ${num}`));

```

```
// Custom higher-order function
function applyOperation(x, y, operation) {
  return operation(x, y);
}

const add = (a, b) => a + b;
const subtract = (a, b) => a - b;
const multiply = (a, b) => a * b;

console.log(applyOperation(5, 3, add)); // 8
console.log(applyOperation(5, 3, subtract)); // 2
console.log(applyOperation(5, 3, multiply)); // 15

// Function that creates custom operations
function createOperation(operator) {
  switch (operator) {
    case '+':
      return (a, b) => a + b;
    case '-':
      return (a, b) => a - b;
    case '*':
      return (a, b) => a * b;
    case '/':
      return (a, b) => a / b;
    default:
      return (a, b) => NaN;
  }
}

const divide = createOperation('/');
console.log(divide(10, 2)); // 5

// Function composition
function compose(f, g) {
  return function (x) {
    return f(g(x));
  };
}

const double = (x) => x * 2;
const square = (x) => x * x;

const doubleSquare = compose(double, square);
const squareDouble = compose(square, double);

console.log(doubleSquare(3)); // double(square(3)) = double(9) = 18
console.log(squareDouble(3)); // square(double(3)) = square(6) = 36
```

Function Context and **this** Keyword:

```
// 'this' refers to the execution context of a function
// Its value depends on how the function is called

// Method invocation - 'this' refers to the object
const person = {
  name: 'Alice',
  greet: function () {
    return `Hello, my name is ${this.name}`;
  },
};

console.log(person.greet()); // "Hello, my name is Alice"

// Function invocation - 'this' refers to global object (window in browser, global
// in Node)
// In strict mode, 'this' is undefined
function standalone() {
  console.log(this);
}

standalone(); // window or global object (or undefined in strict mode)

// Constructor invocation - 'this' refers to the new object
function Person(name) {
  this.name = name;
  this.sayHello = function () {
    return `Hello, my name is ${this.name}`;
  };
}

const alice = new Person('Alice');
console.log(alice.sayHello()); // "Hello, my name is Alice"

// call(), apply(), and bind() - explicitly set 'this'
function introduce(greeting, punctuation) {
  return `${greeting}, my name is ${this.name}${punctuation}`;
}

const bob = { name: 'Bob' };

// call() - pass 'this' and arguments individually
console.log(introduce.call(bob, 'Hi', '!')); // "Hi, my name is Bob!"

// apply() - pass 'this' and arguments as an array
console.log(introduce.apply(bob, ['Hello', '.'])); // "Hello, my name is Bob."

// bind() - creates a new function with 'this' permanently bound
const bobIntroduce = introduce.bind(bob);
console.log(bobIntroduce('Hey', '...')); // "Hey, my name is Bob..."

// Partially applied function with bind()
const bobSayHi = introduce.bind(bob, 'Hi');
console.log(bobSayHi('!')); // "Hi, my name is Bob!"
```

```
// Arrow functions don't have their own 'this'  
// They inherit 'this' from the enclosing scope  
const team = {  
    members: ['Alice', 'Bob'],  
    leader: 'Charlie',  
  
    // Regular function - 'this' changes with context  
    showMembersRegular: function () {  
        // 'this' here is the team object  
        return this.members.map(function (member) {  
            // 'this' here is undefined or global object  
            return `${member} reports to ${this.leader}`;  
        });  
    },  
  
    // Arrow function - 'this' inherits from parent scope  
    showMembersArrow: function () {  
        // 'this' here is the team object  
        return this.members.map((member) => {  
            // 'this' here is still the team object  
            return `${member} reports to ${this.leader}`;  
        });  
    },  
};  
  
// This will fail or show undefined for this.leader:  
try {  
    console.log(team.showMembersRegular());  
} catch (e) {  
    console.log('Error with regular function');  
}  
  
// This works correctly:  
console.log(team.showMembersArrow());  
// ["Alice reports to Charlie", "Bob reports to Charlie"]
```

Objects and Prototypes

Conceptual Foundations

Objects are one of JavaScript's most fundamental data structures. They're collections of key-value pairs where values can be any data type, including other objects and functions.

Mental Model: Think of objects as containers with labeled compartments. Each label (key) identifies a specific compartment that stores a value. This structure allows you to organize related data and functionality together.

Prototypes provide JavaScript's inheritance mechanism. Every JavaScript object has a private property that links to another object called its prototype. This prototype object has its own prototype, creating a chain until reaching an object with a null prototype.

Mental Model for Prototypes: Imagine a family tree of objects. When you look for a property on an object, JavaScript first checks the object itself. If it doesn't find it there, it checks the object's parent (prototype), then grandparent, and so on up the chain.

Common Misconceptions:

- Primitive values (strings, numbers) are not objects (though they have object wrappers)
- JavaScript uses prototypal inheritance, not class-based inheritance (classes in ES6 are syntactic sugar)
- Object properties are always accessible (not truly private without closures or newer techniques)
- The prototype chain can affect performance if it's too deep

Practical Implementation

Creating and Using Objects:

```
// Object literal notation
const person = {
  firstName: 'John',
  lastName: 'Doe',
  age: 30,
  hobbies: ['reading', 'cycling', 'cooking'],
  address: {
    street: '123 Main St',
    city: 'Anytown',
    country: 'USA',
  },
  isEmployed: true,

  // Method (function as a property)
  fullName: function () {
    return `${this.firstName} ${this.lastName}`;
  },

  // ES6 shorthand method syntax
  greet() {
    return `Hello, my name is ${this.fullName()}`;
  },
};

// Accessing object properties
console.log(person.firstName); // "John"
console.log(person['lastName']); // "Doe" (bracket notation)
console.log(person.address.city); // "Anytown"
console.log(person.fullName()); // "John Doe"
console.log(person.greet()); // "Hello, my name is John Doe"

// Adding new properties
person.email = 'john.doe@example.com';
person['phone'] = '555-1234';

// Modifying properties
person.age = 31;
```

```
// Deleting properties
delete person.isEmployed;

// Checking if a property exists
console.log('email' in person); // true
console.log(person.hasOwnProperty('phone')); // true
console.log('ssn' in person); // false

// Getting all keys of an object
console.log(Object.keys(person)); // ["firstName", "lastName", "age", "hobbies", "address", "fullName", "greet", "email", "phone"]

// Getting all values of an object
console.log(Object.values(person)); // [Array of values]

// Getting all entries (key-value pairs)
console.log(Object.entries(person)); // [Array of [key, value] arrays]
```

Object Creation Patterns:

```
// 1. Object literals (as shown above)
const book = {
  title: 'JavaScript: The Good Parts',
  author: 'Douglas Crockford',
  year: 2008,
};

// 2. Constructor functions (pre-ES6)
function Person(firstName, lastName, age) {
  // 'this' refers to the new object being created
  this.firstName = firstName;
  this.lastName = lastName;
  this.age = age;

  this.fullName = function () {
    return `${this.firstName} ${this.lastName}`;
  };
}

const john = new Person('John', 'Doe', 30);
console.log(john.fullName()); // "John Doe"

// 3. Object.create() - creates a new object with the specified prototype
const personPrototype = {
  greet() {
    return `Hello, my name is ${this.firstName} ${this.lastName}`;
  },
};

const jane = Object.create(personPrototype);
jane.firstName = 'Jane';
```

```

jane.lastName = 'Smith';
console.log(jane.greet()); // "Hello, my name is Jane Smith"

// 4. ES6 Classes (syntactic sugar over prototypes)
class Employee {
  constructor(firstName, lastName, position) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.position = position;
  }

  fullName() {
    return `${this.firstName} ${this.lastName}`;
  }

  describe() {
    return `${this.fullName()} works as a ${this.position}`;
  }
}

const alice = new Employee('Alice', 'Johnson', 'Developer');
console.log(alice.describe()); // "Alice Johnson works as a Developer"

// 5. Factory functions (return new objects without using 'new')
function createPerson(firstName, lastName, age) {
  return {
    firstName,
    lastName,
    age,
    fullName() {
      return `${this.firstName} ${this.lastName}`;
    },
  };
}

const bob = createPerson('Bob', 'Smith', 35);
console.log(bob.fullName()); // "Bob Smith"

```

Prototypes and Inheritance:

```

// Understanding the prototype chain
function Animal(name) {
  this.name = name;
}

// Adding a method to the prototype
Animal.prototype.makeSound = function () {
  return 'Some generic sound';
};

// Create an instance
const animal = new Animal('Generic Animal');

```

```
console.log(animal.makeSound()); // "Some generic sound"

// Create a Dog constructor that inherits from Animal
function Dog(name, breed) {
  // Call the parent constructor
  Animal.call(this, name);
  this.breed = breed;
}

// Set up the prototype chain
Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.constructor = Dog; // Fix the constructor property

// Override the makeSound method
Dog.prototype.makeSound = function () {
  return 'Woof!';
};

// Add a new method specific to dogs
Dog.prototype.fetch = function () {
  return `${this.name} is fetching.`;
};

const dog = new Dog('Buddy', 'Golden Retriever');
console.log(dog.name); // "Buddy"
console.log(dog.breed); // "Golden Retriever"
console.log(dog.makeSound()); // "Woof!"
console.log(dog.fetch()); // "Buddy is fetching."

// Verify the prototype chain
console.log(dog instanceof Dog); // true
console.log(dog instanceof Animal); // true
console.log(dog instanceof Object); // true

// ES6 class syntax for inheritance
class Animal2 {
  constructor(name) {
    this.name = name;
  }

  makeSound() {
    return 'Some generic sound';
  }
}

class Dog2 extends Animal2 {
  constructor(name, breed) {
    super(name); // Call parent constructor
    this.breed = breed;
  }

  makeSound() {
    return 'Woof!';
  }
}
```

```
fetch() {
  return `${this.name} is fetching.`;
}
}

const dog2 = new Dog2('Rex', 'German Shepherd');
console.log(dog2.makeSound()); // "Woof!"
```

Object Property Descriptors and Immutability:

```
// Property descriptors control how properties behave
const product = {};

// Define a property with specific attributes
Object.defineProperty(product, 'name', {
  value: 'Laptop',
  writable: true, // Can change the value
  enumerable: true, // Shows up in loops/Object.keys
  configurable: true, // Can be deleted/modified
});

// Define a read-only property
Object.defineProperty(product, 'id', {
  value: 'PROD-123',
  writable: false, // Cannot change the value
  enumerable: true,
  configurable: false, // Cannot be deleted/redefined
});

product.name = 'Desktop'; // Works
console.log(product.name); // "Desktop"

// This won't work (in strict mode it throws an error)
product.id = 'PROD-456';
console.log(product.id); // Still "PROD-123"

// Get property descriptor
console.log(Object.getOwnPropertyDescriptor(product, 'name'));

// Define multiple properties at once
Object.defineProperties(product, {
  price: {
    value: 999.99,
    writable: true,
    enumerable: true,
    configurable: true,
  },
  currency: {
    value: 'USD',
    writable: false,
    enumerable: true,
  }
});
```

```
    configurable: false,
  },
});

// Creating immutable objects
const settings = {
  theme: 'dark',
  notifications: {
    email: true,
    sms: false,
  },
};

// Make the object non-extensible (can't add properties)
Object.preventExtensions(settings);
settings.sound = 'on'; // Won't work
console.log(settings.sound); // undefined

// Make the object sealed (can't add/delete properties)
Object.seal(settings);
delete settings.theme; // Won't work
console.log(settings.theme); // "dark"

// Make the object frozen (can't add/delete/modify properties)
Object.freeze(settings);
settings.theme = 'light'; // Won't work
console.log(settings.theme); // Still "dark"

// Note: Freezing is shallow - nested objects aren't frozen
settings.notifications.email = false; // This works!
console.log(settings.notifications.email); // false

// Deep freeze function (recursive)
function deepFreeze(obj) {
  // Freeze properties before freezing self
  Object.keys(obj).forEach((prop) => {
    if (typeof obj[prop] === 'object' && obj[prop] !== null) {
      deepFreeze(obj[prop]);
    }
  });
  return Object.freeze(obj);
}

const config = {
  server: {
    port: 3000,
    host: 'localhost',
  },
};

deepFreeze(config);
config.server.port = 8080; // Won't work
console.log(config.server.port); // Still 3000
```

Modern Object Features (ES6+):

```
// Object destructuring
const user = {
  name: 'Alice',
  age: 30,
  location: {
    city: 'New York',
    country: 'USA',
  },
};

// Basic destructuring
const { name, age } = user;
console.log(name, age); // "Alice" 30

// Destructuring with different variable names
const { name: userName, age: userAge } = user;
console.log(userName, userAge); // "Alice" 30

// Destructuring nested objects
const {
  location: { city, country },
} = user;
console.log(city, country); // "New York" "USA"

// Default values
const { name: personName, role = 'User' } = user;
console.log(personName, role); // "Alice" "User"

// Rest operator with destructuring
const { name: customerName, ...details } = user;
console.log(customerName); // "Alice"
console.log(details); // { age: 30, location: { city: 'New York', country: 'USA' } }

// Object spread operator
const defaults = { theme: 'light', fontSize: 16 };
const userPreferences = { theme: 'dark' };

// Combine objects (newer values override)
const settings = { ...defaults, ...userPreferences };
console.log(settings); // { theme: 'dark', fontSize: 16 }

// Object property shorthand
const firstName = 'John';
const lastName = 'Doe';

// Instead of { firstName: firstName, lastName: lastName }
const shorthandPerson = { firstName, lastName };
console.log(shorthandPerson); // { firstName: 'John', lastName: 'Doe' }
```

```

// Computed property names
const propName = 'job';
const employee = {
  name: 'Bob',
  [propName]: 'Developer', // 'job' becomes the property name
  [`$${propName}Level`]: 'Senior', // 'jobLevel' becomes the property name
};
console.log(employee); // { name: 'Bob', job: 'Developer', jobLevel: 'Senior' }

// Object methods from ES6+
// Object.assign() - merge objects
const target = { a: 1, b: 2 };
const source = { b: 3, c: 4 };
const merged = Object.assign(target, source);
console.log(merged); // { a: 1, b: 3, c: 4 }
console.log(target); // { a: 1, b: 3, c: 4 } (modified!)

// Better way using spread (doesn't modify target)
const safelyMerged = { ...target, ...source };

// Object.is() - compare values
console.log(Object.is(5, 5)); // true
console.log(Object.is(5, '5')); // false
console.log(Object.is(NaN, NaN)); // true (unlike === which returns false)
console.log(Object.is(0, -0)); // false (unlike === which returns true)

// Object.fromEntries() - convert array of key-value pairs to object
const entries = [
  ['name', 'Charlie'],
  ['age', 40],
];
const entriesObject = Object.fromEntries(entries);
console.log(entriesObject); // { name: 'Charlie', age: 40 }

```

Arrays and Collections

Conceptual Foundations

Arrays in JavaScript are ordered collections of values. Unlike objects which use named properties, arrays use numeric indices. Arrays are a special type of object optimized for ordered data.

Mental Model: Think of arrays as ordered lists or sequences where each item has a numbered position (starting from 0). This makes them perfect for collections where order matters and you need to access elements by their position.

Common Misconceptions:

- JavaScript arrays can hold mixed data types (unlike some other languages)
- Arrays in JavaScript are dynamic (can grow or shrink)
- Arrays' length can be greater than the number of elements (sparse arrays)
- Arrays methods like map, filter, reduce don't modify the original array

Practical Implementation

Creating and Manipulating Arrays:

```
// Creating arrays
const numbers = [1, 2, 3, 4, 5];
const mixed = [42, 'hello', true, null, { name: 'Alice' }, [1, 2]];
const empty = [];
const preDefinedLength = new Array(5); // Creates array with 5 empty slots
const fromValues = Array.of(1, 2, 3); // Creates array with these values
const fromIterable = Array.from('hello'); // Creates array ['h', 'e', 'l', 'l', 'o']
const usingSpread = [...numbers, 6, 7]; // Creates [1, 2, 3, 4, 5, 6, 7]

// Accessing elements
console.log(numbers[0]); // 1 (first element)
console.log(numbers[numbers.length - 1]); // 5 (last element)
console.log(numbers.at(-1)); // 5 (last element using at() - ES2022)

// Modifying arrays
numbers[0] = 10; // Replace first element
numbers.push(6); // Add to end
numbers.unshift(0); // Add to beginning
numbers.pop(); // Remove from end and return it
numbers.shift(); // Remove from beginning and return it
numbers.splice(2, 1, 'a', 'b'); // Replace 1 element at index 2 with 'a' and 'b'

// Finding elements
console.log(numbers.indexOf(4)); // Returns index of first occurrence (or -1)
console.log(numbers.lastIndexOf(4)); // Returns index of last occurrence (or -1)
console.log(numbers.includes(5)); // Returns true if element exists

// Find objects or by condition
const users = [
  { id: 1, name: 'Alice' },
  { id: 2, name: 'Bob' },
  { id: 3, name: 'Charlie' },
];

// Find first element matching condition
const bob = users.find((user) => user.name === 'Bob');
console.log(bob); // { id: 2, name: "Bob" }

// Find index of first matching element
const charlieIndex = users.findIndex((user) => user.name === 'Charlie');
console.log(charlieIndex); // 2

// Filter creates new array with elements that pass test
const evenNumbers = numbers.filter(
  (num) => typeof num === 'number' && num % 2 === 0
);
console.log(evenNumbers);
```

```
// Transforming arrays
// Map creates new array by transforming each element
const doubled = numbers.filter((x) => typeof x === 'number').map((x) => x * 2);
console.log(doubled);

// Reduce to a single value (accumulate)
const sum = numbers
  .filter((x) => typeof x === 'number')
  .reduce((acc, val) => acc + val, 0);
console.log(sum);

// Flattening nested arrays
const nested = [
  [1, 2],
  [3, 4],
  [5, 6],
];
const flat = nested.flat(); // [1, 2, 3, 4, 5, 6]
console.log(flat);

// Deep flattening
const deepNested = [
  [1, [2, [3]]],
  [4, [5]],
];
const deepFlat = deepNested.flat(Infinity); // [1, 2, 3, 4, 5]
console.log(deepFlat);

// Sorting arrays
const names = ['Charlie', 'Alice', 'Bob'];
names.sort(); // Modifies the original array!
console.log(names); // ["Alice", "Bob", "Charlie"]

// Sort with custom comparison function
const scores = [40, 100, 1, 5, 25, 10];
scores.sort((a, b) => a - b); // Ascending order
console.log(scores); // [1, 5, 10, 25, 40, 100]

// Sorting objects
const students = [
  { name: 'Alice', score: 85 },
  { name: 'Bob', score: 92 },
  { name: 'Charlie', score: 78 },
];
students.sort((a, b) => b.score - a.score); // Sort by score descending
console.log(students[0].name); // "Bob" (highest score)

// Joining arrays
const arr1 = [1, 2, 3];
const arr2 = [4, 5, 6];
const combined = arr1.concat(arr2); // [1, 2, 3, 4, 5, 6]
const withSpread = [...arr1, ...arr2]; // Same result using spread
```

```
// Converting to/from strings
const fruits = ['apple', 'banana', 'cherry'];
const fruitsString = fruits.join(', '); // "apple, banana, cherry"
console.log(fruitsString);

const backToArray = fruitsString.split(', '); // ["apple", "banana", "cherry"]
console.log(backToArray);

// Checking if array
console.log(Array.isArray(fruits)); // true
console.log(Array.isArray('not array'))); // false
```

Array Iteration Methods:

```
const numbers = [1, 2, 3, 4, 5];

// forEach - executes a function for each element
numbers.forEach((num, index, array) => {
  console.log(`#${num} is at index ${index} in ${array}`);
});

// map - creates a new array with transformed elements
const squared = numbers.map((num) => num * num);
console.log(squared); // [1, 4, 9, 16, 25]

// filter - creates a new array with elements that pass a test
const evenNumbers = numbers.filter((num) => num % 2 === 0);
console.log(evenNumbers); // [2, 4]

// reduce - accumulates a value based on elements
const sum = numbers.reduce((total, num) => total + num, 0);
console.log(sum); // 15 (1+2+3+4+5)

// reduceRight - like reduce but from right to left
const concatenated = ['a', 'b', 'c'].reduceRight((acc, val) => acc + val, '');
console.log(concatenated); // "cba"

// every - tests if all elements pass a condition
const allPositive = numbers.every((num) => num > 0);
console.log(allPositive); // true

// some - tests if at least one element passes a condition
const hasEven = numbers.some((num) => num % 2 === 0);
console.log(hasEven); // true

// find - returns first element that passes a test
const firstEven = numbers.find((num) => num % 2 === 0);
console.log(firstEven); // 2

// findIndex - returns index of first element that passes a test
const firstEvenIndex = numbers.findIndex((num) => num % 2 === 0);
```

```

console.log(firstEvenIndex); // 1

// flatMap - combines map and flat
const sentences = ['Hello world', 'How are you?'];
const words = sentences.flatMap((sentence) => sentence.split(' '));
console.log(words); // ["Hello", "world", "How", "are", "you?"]

// Method chaining
const result = numbers
  .filter((num) => num % 2 === 0) // Keep even numbers
  .map((num) => num * 10) // Multiply each by 10
  .reduce((sum, num) => sum + num, 0); // Add them together
console.log(result); // 60 (2*10 + 4*10)

```

Array Destructuring and Spread:

```

// Array destructuring
const rgb = [255, 200, 100];
const [red, green, blue] = rgb;
console.log(red, green, blue); // 255 200 100

// Skip elements
const [first, , third] = [1, 2, 3];
console.log(first, third); // 1 3

// Default values
const [name = 'Guest', role = 'User'] = ['Alice'];
console.log(name, role); // "Alice" "User"

// Rest operator in destructuring
const [leader, ...teammates] = ['Alice', 'Bob', 'Charlie', 'David'];
console.log(leader); // "Alice"
console.log(teammates); // ["Bob", "Charlie", "David"]

// Swapping variables
let a = 1;
let b = 2;
[a, b] = [b, a];
console.log(a, b); // 2 1

// Nested destructuring
const colors = [
  ['red', '#FF0000'],
  ['green', '#00FF00'],
  ['blue', '#0000FF'],
];
const [[primaryColor, primaryHex], [secondaryColor, secondaryHex]] = colors;
console.log(primaryColor, primaryHex); // "red" "#FF0000"

// Function that returns multiple values via array
function getUser() {
  return ['Alice', 30, 'Developer'];
}

```

```

}

const [userName, userAge, userRole] = getUser();
console.log(userName, userAge, userRole); // "Alice" 30 "Developer"

// Spread operator with arrays
const originalArray = [1, 2, 3];
const copy = [...originalArray]; // Creates a shallow copy
const extended = [...originalArray, 4, 5]; // Add elements
const combined = [...originalArray, ...extended]; // Combine arrays

// Spreading into function arguments
function sum(a, b, c) {
  return a + b + c;
}
const values = [1, 2, 3];
console.log(sum(...values)); // 6

```

Other Collection Types:

```

// Set - collection of unique values
const uniqueNumbers = new Set([1, 2, 3, 3, 4, 4, 5]);
console.log(uniqueNumbers); // Set {1, 2, 3, 4, 5}
console.log(uniqueNumbers.size); // 5

// Adding and removing elements
uniqueNumbers.add(6);
uniqueNumbers.delete(1);
console.log(uniqueNumbers.has(3)); // true
console.log(uniqueNumbers.has(1)); // false

// Converting to/from arrays
const setArray = Array.from(uniqueNumbers);
console.log(setArray); // [2, 3, 4, 5, 6]
const newSet = new Set([7, 8, 9]);
const combinedArray = [...uniqueNumbers, ...newSet];

// Practical Set example: Remove duplicates
function removeDuplicates(array) {
  return [...new Set(array)];
}
console.log(removeDuplicates([1, 2, 2, 3, 4, 4, 5])); // [1, 2, 3, 4, 5]

// Map - collection of key-value pairs (any type as key)
const userRoles = new Map();

// Setting values
userRoles.set('alice', 'Admin');
userRoles.set('bob', 'Editor');
userRoles.set('charlie', 'Viewer');

// Getting values
console.log(userRoles.get('alice')); // "Admin"

```

```
console.log(userRoles.size); // 3
console.log(userRoles.has('bob')); // true

// Objects can be keys (unlike regular objects)
const userObject = { id: 1, name: 'Alice' };
userRoles.set(userObject, 'SuperAdmin');
console.log(userRoles.get(userObject)); // "SuperAdmin"

// Initialize with nested arrays
const ages = new Map([
  ['Alice', 28],
  ['Bob', 35],
  ['Charlie', 42],
]);

// Iteration with Map
ages.forEach((age, name) => {
  console.log(` ${name} is ${age} years old.`);
});

// Map iteration with destructuring
for (const [name, age] of ages) {
  console.log(`${name}: ${age}`);
}

// Get all keys, values, or entries
console.log([...ages.keys()]); // ["Alice", "Bob", "Charlie"]
console.log([...ages.values()]); // [28, 35, 42]
console.log([...ages.entries()]); // [[{"name": "Alice", "age": 28}, {"name": "Bob", "age": 35}, {"name": "Charlie", "age": 42}]

// WeakMap and WeakSet
// Similar to Map and Set, but:
// - Keys must be objects
// - Keys are weakly referenced (can be garbage collected)
// - Not iterable (no forEach, size, or entries())
// Good for associating data with objects without preventing garbage collection

const weakMap = new WeakMap();
let obj = { id: 1 };
weakMap.set(obj, 'metadata');
console.log(weakMap.get(obj)); // "metadata"

// If obj is later set to null, both obj and its entry in weakMap
// can be garbage collected
```

PART 2: ADVANCED JAVASCRIPT

Asynchronous JavaScript

Conceptual Foundations

Asynchronous programming is a technique that enables your program to start a potentially long-running task and still be able to be responsive to other events while that task runs, rather than having to wait until that task has finished. Once that task has finished, your program is presented with the result.

Mental Model: Think of asynchronous operations like ordering food at a restaurant. After placing your order (starting an operation), you don't wait idly at the counter; you sit at your table and do other things (continue execution). When your food is ready (operation completes), the waiter brings it to you (callback or promise resolution).

Historical Context: JavaScript began with callback functions as the primary means of handling asynchronous operations. As applications grew more complex, this led to "callback hell" - deeply nested callbacks that were hard to read and maintain. Promises were introduced to address this, followed by `async/await` syntax which made asynchronous code look and behave more like synchronous code.

Common Misconceptions:

- Asynchronous does not mean concurrent or multithreaded (JavaScript is still single-threaded)
- `await` doesn't stop the JavaScript engine from working; it only pauses the execution of the current function
- Promises don't execute in parallel by default (use `Promise.all()` for that)
- Setting a timeout of 0ms doesn't execute immediately; it waits for the call stack to clear

Practical Implementation

Callbacks:

```
// Callbacks are functions passed as arguments to other functions,
// to be executed after an operation completes

// Simple callback example
function fetchData(callback) {
    // Simulate network request with setTimeout
    setTimeout(() => {
        const data = { id: 1, name: 'JavaScript' };
        callback(null, data); // null means no error
    }, 1000);
}

// Using the callback
fetchData((error, data) => {
    if (error) {
        console.error('Error fetching data:', error);
        return;
    }
    console.log('Data received:', data);
});

console.log('This runs before the callback');

// Callback hell example (nested callbacks)
function getUser(userId, callback) {
```

```
setTimeout(() => {
  console.log('Getting user...');
  callback(null, { id: userId, name: 'Alice' });
}, 1000);
}

function getUserPosts(userId, callback) {
  setTimeout(() => {
    console.log('Getting posts...');
    callback(null, [
      { id: 1, title: 'Post 1' },
      { id: 2, title: 'Post 2' },
    ]);
  }, 1000);
}

function getPostComments(postId, callback) {
  setTimeout(() => {
    console.log('Getting comments...');
    callback(null, [
      { id: 1, text: 'Comment 1' },
      { id: 2, text: 'Comment 2' },
    ]);
  }, 1000);
}

// Nested callbacks - hard to read and maintain
getUser(1, (userError, user) => {
  if (userError) {
    console.error(userError);
    return;
  }

  console.log('User:', user);

  getUserPosts(user.id, (postsError, posts) => {
    if (postsError) {
      console.error(postsError);
      return;
    }

    console.log('Posts:', posts);

    getPostComments(posts[0].id, (commentsError, comments) => {
      if (commentsError) {
        console.error(commentsError);
        return;
      }

      console.log('Comments:', comments);
    });
  });
});
```

Promises:

```
// Promises represent a value that might be available now, later, or never

// Creating a promise
const dataPromise = new Promise((resolve, reject) => {
  // Simulate network request
  setTimeout(() => {
    const success = true;

    if (success) {
      resolve({ id: 1, name: 'JavaScript' }); // Fulfilled
    } else {
      reject(new Error('Failed to fetch data')); // Rejected
    }
  }, 1000);
});

// Using a promise
dataPromise
  .then((data) => {
    console.log('Success:', data);
    return data;
  })
  .catch((error) => {
    console.error('Error:', error.message);
  })
  .finally(() => {
    console.log('Promise completed (either success or failure)');
  });

console.log('This runs before the promise resolves');

// Converting callback functions to promises
function getUser(userId) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log('Getting user...');
      resolve({ id: userId, name: 'Alice' });
    }, 1000);
  });
}

function getUserPosts(userId) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log('Getting posts...');
      resolve([
        { id: 1, title: 'Post 1' },
        { id: 2, title: 'Post 2' },
      ]);
    }, 1000);
  });
}
```

```
});

}

function getPostComments(postId) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log('Getting comments...');
      resolve([
        { id: 1, text: 'Comment 1' },
        { id: 2, text: 'Comment 2' },
      ]);
    }, 1000);
  });
}

// Promise chaining - much cleaner than nested callbacks
getUser(1)
  .then((user) => {
    console.log('User:', user);
    return getUserPosts(user.id);
  })
  .then((posts) => {
    console.log('Posts:', posts);
    return getPostComments(posts[0].id);
  })
  .then((comments) => {
    console.log('Comments:', comments);
  })
  .catch((error) => {
    console.error('Error:', error);
  });

// Promise combinators
// Promise.all - waits for all promises to resolve (or first rejection)
Promise.all([getUser(1), getUserPosts(1), getPostComments(1)])
  .then(([user, posts, comments]) => {
    console.log('All data:', user, posts, comments);
  })
  .catch((error) => {
    console.error('One of the promises failed:', error);
  });

// Promise.race - resolves or rejects as soon as the first promise does
Promise.race([
  new Promise((resolve) => setTimeout(() => resolve('First'), 1000)),
  new Promise((resolve) => setTimeout(() => resolve('Second'), 500)),
]).then((result) => {
  console.log('Fastest result:', result); // "Second"
});

// Promise.allSettled - waits for all promises to settle (resolve or reject)
Promise.allSettled([
  Promise.resolve('Success'),
  Promise.reject('Failure'),
```

```

Promise.resolve('Another success'),
]).then((results) => {
  console.log(results);
  // [
  //   { status: "fulfilled", value: "Success" },
  //   { status: "rejected", reason: "Failure" },
  //   { status: "fulfilled", value: "Another success" }
  // ]
});

// Promise.any - resolves as soon as any promise resolves (rejects only if all
// reject)
Promise.any([
  new Promise((resolve, reject) => setTimeout(() => reject('Failure 1'), 1000)),
  new Promise((resolve, reject) => setTimeout(() => resolve('Success'), 2000)),
  new Promise((resolve, reject) => setTimeout(() => reject('Failure 2'), 3000)),
])
.then((result) => {
  console.log('First success:', result); // "Success"
})
.catch((error) => {
  console.log('All promises failed:', error);
});

```

Async/Await:

```

// Async/await is syntactic sugar over promises
// Allows asynchronous code to look and behave more like synchronous code

// Async function declaration
async function fetchUserData(userId) {
  try {
    // await pauses execution until the promise resolves
    const user = await getUser(userId);
    console.log('User:', user);

    const posts = await getUserPosts(user.id);
    console.log('Posts:', posts);

    const comments = await getPostComments(posts[0].id);
    console.log('Comments:', comments);

    return { user, posts, comments };
  } catch (error) {
    // Catches any errors in the try block
    console.error('Error in fetchUserData:', error);
    throw error; // Re-throw the error for the caller to handle
  }
}

// Calling an async function returns a promise
fetchUserData(1)

```

```
.then((data) => {
  console.log('All data:', data);
})
.catch((error) => {
  console.error('Caught outside:', error);
});

// Async arrow function
const getUserInfo = async (userId) => {
  const user = await getUser(userId);
  return user;
};

// Parallel operations with await and Promise.all
async function fetchAllData(userId) {
  try {
    // These will run in parallel, not sequentially
    const [user, posts, comments] = await Promise.all([
      getUser(userId),
      getUserPosts(userId),
      getPostComments(1),
    ]);

    console.log('User:', user);
    console.log('Posts:', posts);
    console.log('Comments:', comments);

    return { user, posts, comments };
  } catch (error) {
    console.error('Error:', error);
    throw error;
  }
}

// Looping with async/await
async function processArray(array) {
  // Sequential processing (each awaits completion before starting next)
  for (const item of array) {
    await processItem(item);
  }

  // Parallel processing
  const promises = array.map((item) => processItem(item));
  await Promise.all(promises);
}

async function processItem(item) {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log(`Processed ${item}`);
      resolve(item);
    }, 1000);
  });
}
```

```
// IIFE (Immediately Invoked Function Expression) with async
(async () => {
  try {
    const result = await fetchAllData(1);
    console.log('IIFE result:', result);
  } catch (error) {
    console.error('IIFE error:', error);
  }
})();

// Class methods can be async too
class UserService {
  async getUser(userId) {
    // Implementation
    return await fetch(`/api/users/${userId}`).then((res) => res.json());
  }
}
```

Event Loop and Microtasks:

```
// The event loop is JavaScript's mechanism for executing code asynchronously

console.log('1. Script start'); // Synchronous

// setTimeout callback goes to the Task Queue (Macrotask)
setTimeout(() => {
  console.log('2. setTimeout callback');
}, 0);

// Promise callbacks go to the Microtask Queue
Promise.resolve()
  .then(() => {
    console.log('3. Promise then 1');
  })
  .then(() => {
    console.log('4. Promise then 2');
  });

// Synchronous code executes immediately
console.log('5. Script end');

// Output order:
// 1. Script start
// 5. Script end
// 3. Promise then 1
// 4. Promise then 2
// 2. setTimeout callback

// This happens because:
// 1. Synchronous code runs first
// 2. Microtasks (Promises) run next, before the next Task
```

```
// 3. Tasks (setTimeout, events) run last

// More complex example
console.log('A');

setTimeout(() => {
  console.log('B');
}, 0);

Promise.resolve()
  .then(() => {
    console.log('C');

    // This creates a new microtask
    Promise.resolve().then(() => {
      console.log('D');
    });
  });

  // This creates a new Task
  setTimeout(() => {
    console.log('E');
  }, 0);
})

.then(() => {
  console.log('F');
});

console.log('G');

// Output:
// A
// G
// C
// D
// F
// B
// E
```

Real-world Asynchronous Patterns:

```
// Timeout promises
function timeout(ms) {
  return new Promise((resolve) => setTimeout(resolve, ms));
}

async function delayedGreeting(name) {
  await timeout(1000);
  return `Hello, ${name}!`;
}

// Promise with cancellation using AbortController
function fetchWithTimeout(url, options = {}, timeoutMs = 5000) {
```

```
const controller = new AbortController();
const { signal } = controller;

// Create a timeout that aborts the fetch
const timeoutId = setTimeout(() => controller.abort(), timeoutMs);

// Create the fetch promise with the abort signal
const fetchPromise = fetch(url, { ...options, signal });

// Return a promise that cleans up properly
return fetchPromise
  .then((response) => {
    clearTimeout(timeoutId);
    return response;
  })
  .catch((error) => {
    clearTimeout(timeoutId);
    if (error.name === 'AbortError') {
      throw new Error(`Request timed out after ${timeoutMs}ms`);
    }
    throw error;
  });
}

// Retry mechanism with exponential backoff
async function fetchWithRetry(url, options = {}, maxRetries = 3) {
  let retries = 0;

  while (true) {
    try {
      return await fetch(url, options);
    } catch (error) {
      retries++;

      if (retries >= maxRetries) {
        throw new Error(
          `Max retries (${maxRetries}) exceeded: ${error.message}`
        );
      }
    }

    // Exponential backoff with jitter
    const delay = Math.min(1000 * 2 ** retries, 10000) + Math.random() * 1000;
    console.log(`Retry ${retries} after ${delay}ms`);

    await timeout(delay);
  }
}

// Async utility for limiting concurrency
async function runWithConcurrencyLimit(tasks, limit = 5) {
  const results = [];
  const executing = new Set();

  tasks.forEach(task => {
    if (executing.size < limit) {
      executing.add(task);
      task().finally(() => executing.delete(task));
    } else {
      results.push(task);
    }
  });

  return Promise.all(results);
}
```

```
for (const [index, task] of tasks.entries()) {
  const promise = Promise.resolve().then(() => task());
  results[index] = promise;

  executing.add(promise);

  const clean = () => executing.delete(promise);
  promise.then(clean, clean);

  if (executing.size >= limit) {
    await Promise.race(executing);
  }
}

return Promise.all(results);
}

// Usage example
const tasks = Array.from({ length: 10 }, (_, i) => async () => {
  await timeout(Math.random() * 1000);
  return i;
});

runWithConcurrencyLimit(tasks, 3).then((results) =>
  console.log('Results:', results)
);
```

ES6+ Features and Modern JavaScript

Conceptual Foundations

ECMAScript 6 (ES6), also known as ECMAScript 2015, was a major update to JavaScript that introduced many new features. Since then, JavaScript has continued to evolve with yearly updates (ES2016, ES2017, etc.), each adding new capabilities to the language.

Mental Model: Think of modern JavaScript as a continuously improving toolkit. Each new version adds more specialized and powerful tools that help developers write cleaner, more expressive, and more efficient code.

Historical Context: Before ES6, JavaScript had not seen major language updates for many years. ES6 was a significant milestone that modernized the language, and subsequent yearly releases have continued to refine and extend JavaScript's capabilities.

Common Misconceptions:

- You can't use new features in older browsers (transpilers like Babel can convert modern code to compatible versions)
- ES6+ features are just syntactic sugar (many features bring genuine new capabilities)
- All new JavaScript features automatically work in all environments (always check compatibility)

Practical Implementation

ES6 Features:

```
// let and const for block-scoped variables (covered earlier)
// Arrow functions (covered earlier)

// Template literals
const name = 'JavaScript';
const greeting = `Hello, ${name}!`;
console.log(greeting); // "Hello, JavaScript!"

// Multi-line strings
const multiLine = `This is a
multi-line
string`;

// Default parameters
function greet(name = 'Guest') {
  return `Hello, ${name}!`;
}

// Rest and spread operators
// ...rest in function parameters
function collectArgs(first, ...rest) {
  console.log('First:', first);
  console.log('Rest:', rest);
}
collectArgs(1, 2, 3, 4); // First: 1, Rest: [2, 3, 4]

// ...spread for arrays and objects
const arr1 = [1, 2, 3];
const arr2 = [...arr1, 4, 5]; // [1, 2, 3, 4, 5]

const obj1 = { a: 1, b: 2 };
const obj2 = { ...obj1, c: 3 }; // { a: 1, b: 2, c: 3 }

// Destructuring assignment
const [first, second] = [1, 2];
const { a, b } = { a: 1, b: 2 };

// Enhanced object literals
const x = 1,
  y = 2;
// Property shorthand
const coords = { x, y };

// Method shorthand
const methods = {
  sayHello() {
    return 'Hello';
  },
};

// Computed property names
```

```
const propName = 'dynamicProp';
const obj = {
  [propName]: 'Dynamic value',
};

// Classes
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  sayHello() {
    return `Hello, my name is ${this.name}`;
  }

  // Getter
  get description() {
    return `${this.name}, ${this.age} years old`;
  }

  // Static method
  static createAnonymous() {
    return new Person('Anonymous', 0);
  }
}

// Inheritance
class Employee extends Person {
  constructor(name, age, company) {
    super(name, age);
    this.company = company;
  }

  sayHello() {
    return `${super.sayHello()} and I work at ${this.company}`;
  }
}

// Modules
// In file math.js:
// export function add(a, b) { return a + b; }
// export function subtract(a, b) { return a - b; }
// export const PI = 3.14159;

// In another file:
// import { add, subtract, PI } from './math.js';
// import * as math from './math.js';
// import { add as sum } from './math.js';

// Default export:
// export default function multiply(a, b) { return a * b; }
// import multiply from './math.js';
```

```
// Promises (covered in Asynchronous JavaScript)

// Symbol - unique identifiers
const sym1 = Symbol();
const sym2 = Symbol('description');
const sym3 = Symbol('description');
console.log(sym2 === sym3); // false - symbols are always unique

// Well-known symbols
const customObject = {
  [Symbol.iterator]: function* () {
    yield 1;
    yield 2;
    yield 3;
  },
};

for (const item of customObject) {
  console.log(item); // 1, 2, 3
}

// Map, Set, WeakMap, WeakSet (covered in Arrays and Collections)

// Iterators and Generators
function* simpleGenerator() {
  yield 1;
  yield 2;
  yield 3;
}

const generator = simpleGenerator();
console.log(generator.next()); // { value: 1, done: false }
console.log(generator.next()); // { value: 2, done: false }
console.log(generator.next()); // { value: 3, done: false }
console.log(generator.next()); // { value: undefined, done: true }

// Custom iterator
const iterableObject = {
  data: [1, 2, 3],
  [Symbol.iterator]() {
    let index = 0;
    return {
      next: () => {
        if (index < this.data.length) {
          return { value: this.data[index++], done: false };
        } else {
          return { value: undefined, done: true };
        }
      },
    };
  },
};

for (const item of iterableObject) {
```

```
    console.log(item); // 1, 2, 3
}
```

ES2016+ Features:

```
// ES2016 (ES7)
// Array.prototype.includes
const array = [1, 2, 3, 4, 5];
console.log(array.includes(3)); // true
console.log(array.includes(6)); // false

// Exponentiation operator
console.log(2 ** 3); // 8 (same as Math.pow(2, 3))

// ES2017 (ES8)
// Object.values
const obj = { a: 1, b: 2, c: 3 };
console.log(Object.values(obj)); // [1, 2, 3]

// Object.entries
console.log(Object.entries(obj)); // [["a", 1], ["b", 2], ["c", 3]]

// Object.getOwnPropertyDescriptors
console.log(Object.getOwnPropertyDescriptors(obj));

// String padding
console.log('5'.padStart(3, '0')); // "005"
console.log('Hello'.padEnd(10, '!')); // "Hello!!!!"

// Trailing commas in function parameter lists
function trailingComma(
  param1,
  param2,
  param3 // Valid in ES2017+
) {
  // Function body
}

// Async/await (covered in Asynchronous JavaScript)

// ES2018 (ES9)
// Rest/spread for objects
const { a, ...rest } = { a: 1, b: 2, c: 3 };
console.log(a); // 1
console.log(rest); // { b: 2, c: 3 }

// Promise.finally (covered in Asynchronous JavaScript)

// Asynchronous iteration
async function* asyncGenerator() {
  yield Promise.resolve(1);
  yield Promise.resolve(2);
```

```
yield Promise.resolve(3);
}

(async () => {
  for await (const value of asyncGenerator()) {
    console.log(value); // 1, 2, 3
  }
})();

// RegExp features
// Named capture groups
const dateRegex = /(?:<year>\d{4})-(?:<month>\d{2})-(?:<day>\d{2})/;
const match = dateRegex.exec('2023-04-15');
console.log(match.groups); // { year: "2023", month: "04", day: "15" }

// ES2019 (ES10)
// Array.prototype.flat
const nestedArray = [1, [2, [3, 4]]];
console.log(nestedArray.flat()); // [1, 2, [3, 4]]
console.log(nestedArray.flat(2)); // [1, 2, 3, 4]

// Array.prototype.flatMap
const sentences = ['Hello world', 'How are you'];
console.log(sentences.flatMap((s) => s.split(' '))); // ["Hello", "world", "How", "are", "you"]

// Object.fromEntries (inverse of Object.entries)
const entries = [
  ['a', 1],
  ['b', 2],
];
console.log(Object.fromEntries(entries)); // { a: 1, b: 2 }

// String.prototype.trimStart() and trimEnd()
const padded = ' Hello ';
console.log(padded.trimStart()); // "Hello "
console.log(padded.trimEnd()); // " Hello"

// ES2020 (ES11)
// Optional chaining
const user = {
  details: {
    name: 'Alice',
  },
};
console.log(user.details?.name); // "Alice"
console.log(user.settings?.theme); // undefined (no error)

// Nullish coalescing operator
const value = null;
console.log(value ?? 'default'); // "default"
console.log('' ?? 'default'); // "" (empty string is not nullish)

// Promise.allSettled (covered in Asynchronous JavaScript)
```

```
// BigInt
const bigNumber = 9007199254740991n; // n suffix creates a BigInt
console.log(bigNumber + 1n); // 9007199254740992n

// String.prototype.matchAll
const regex = /t(e)(st(\d?))/g;
const str = 'test1test2';
const matches = [...str.matchAll(regex)];
console.log(matches); // Array containing all matches and their groups

// Dynamic import
// import("./module.js").then(module => {
//   module.doSomething();
// });

// ES2021 (ES12)
// String.prototype.replaceAll
const string = 'test test test';
console.log(string.replaceAll('test', 'replaced')); // "replaced replaced
replaced"

// Promise.any (covered in Asynchronous JavaScript)

// Logical assignment operators
let x = 0;
// x |= 5; // Equivalent to: x = x || 5
console.log(x); // 5

let y = 0;
// y &&= 5; // Equivalent to: y = y && 5
console.log(y); // 0

let z = null;
// z ??= 5; // Equivalent to: z = z ?? 5
console.log(z); // 5

// Numeric separators
const billion = 1_000_000_000; // More readable
console.log(billion); // 1000000000

// ES2022 (ES13)
// Top-level await (in modules)
// const response = await fetch("https://api.example.com");
// const data = await response.json();

// Class fields
class Modern {
  // Public fields
  publicField = 'public';

  // Private fields
  #privateField = 'private';
```

```
// Static fields
static staticField = 'static';

// Private static fields
static #privateStaticField = 'private static';

// Public method
publicMethod() {
    return this.publicField;
}

// Private method
#privateMethod() {
    return this.#privateField;
}

// Static method
static staticMethod() {
    return this.staticField;
}

// Private static method
static #privateStaticMethod() {
    return this.#privateStaticField;
}

// Static blocks
static {
    // Initialize static properties
    this.initialized = true;
}
}

// Array.prototype.at
const array1 = [1, 2, 3, 4, 5];
console.log(array1.at(-1)); // 5 (last element)
console.log(array1.at(-2)); // 4 (second-to-last element)

// Object.hasOwn (safer than hasOwnProperty)
console.log(Object.hasOwn({ a: 1 }, 'a')); // true
console.log(Object.hasOwn({ a: 1 }, 'toString')); // false

// Regular Expression Match Indices
// const pattern = /test(?:<group>ing)/d;
// const result = pattern.exec("testing");
// console.log(result.indices); // [[0, 7], [4, 7]]
// console.log(result.indices.groups); // { group: [4, 7] }

// ES2023 (ES14)
// Array.prototype.findLast() and findLastIndex()
const numbers = [1, 2, 3, 4, 5, 2];
console.log(numbers.findLast((n) => n % 2 === 0)); // 2 (last even number)
console.log(numbers.findLastIndex((n) => n % 2 === 0)); // 5 (index of last even number)
```

```
// Hashbang grammar for executable JavaScript files
// #!/usr/bin/env node
// console.log("This is a Node.js script");
```

Modern JavaScript Best Practices:

```
// Use strict mode
'use strict';

// Prefer const over let, and let over var
const PI = 3.14159; // Won't change
let count = 0; // Will change

// Use template literals for string interpolation
const name = 'Alice';
const greeting = `Hello, ${name}!`;

// Use arrow functions for short callbacks
const numbers = [1, 2, 3, 4, 5];
const doubled = numbers.map((n) => n * 2);

// Use destructuring for cleaner code
const { firstName, lastName } = user;
const [first, ...rest] = array;

// Use spread for object/array cloning and merging
const clonedObj = { ...originalObj };
const merged = { ...obj1, ...obj2 };

// Use optional chaining for safer property access
const userTheme = user.preferences?.theme ?? 'default';

// Use modern iteration methods instead of for loops
// Instead of:
for (let i = 0; i < items.length; i++) {
    // Do something with items[i]
}

// Use:
items.forEach((item) => {
    // Do something with item
});

// Or for transforming:
const transformed = items.map((item) => transformItem(item));

// Use filter/find instead of loops with if conditions
const filtered = items.filter((item) => item.isValid);
const found = items.find((item) => item.id === 123);

// Use async/await for asynchronous code
```

```
async function fetchData() {
  try {
    const response = await fetch('/api/data');
    const data = await response.json();
    return data;
  } catch (error) {
    console.error('Error fetching data:', error);
    throw error;
  }
}

// Use modules for better code organization
// In module file:
export function helper() {
  // Implementation
}

// In main file:
import { helper } from './helpers.js';

// Use named exports for better refactoring and intellisense
// Instead of:
export default [
  method1,
  method2,
];

// Use:
export { method1, method2 };

// Default parameters instead of conditionals
function configure(options = {}) {
  const { debug = false, timeout = 1000, retries = 3 } = options;

  // Use parameters
}

// Short-circuit evaluation for conditionals
// Instead of:
if (isValid) {
  doSomething();
}

// Use:
isValid && doSomething();

// For default values:
const value = userInput || defaultValue;
const nullishValue = userInput ?? defaultValue;

// Use Array and Object methods for transformations
const sum = numbers.reduce((total, n) => total + n, 0);
const hasNegative = numbers.some((n) => n < 0);
const allPositive = numbers.every((n) => n > 0);
```

```
// Use Set for unique values
const uniqueValues = [...new Set(array)];

// Use Map for complex key-value associations
const userRoles = new Map();
userRoles.set(userObj, 'admin');

// Early returns for cleaner conditionals
function processUser(user) {
  if (!user) return null;
  if (!user.isActive) return 'Inactive user';

  // Process active user
  return 'Active user';
}
```

DOM Manipulation and Browser APIs

Conceptual Foundations

The Document Object Model (DOM) is a programming interface for HTML and XML documents. It represents the page so that programs can change the document structure, style, and content.

Mental Model: Think of the DOM as a tree of nodes representing the structure of a web page. Each element (paragraphs, divs, headings) becomes a node in this tree, which can be accessed, modified, added, or removed using JavaScript.

Historical Context: In the early days of the web, browsers had inconsistent implementations of JavaScript APIs, leading to cross-browser compatibility issues. Libraries like jQuery became popular for providing a consistent interface. Modern browsers now have standardized APIs, making direct DOM manipulation more reliable.

Common Misconceptions:

- DOM manipulation is always slow (only when done inefficiently)
- jQuery is required for cross-browser DOM manipulation (modern browsers have consistent APIs)
- The DOM is the same as HTML (the DOM is a live object model created from HTML)
- DOM changes are immediately reflected visually (some changes require layout/paint cycles)

Practical Implementation

DOM Selection and Traversal:

```
// Selecting single elements
const mainHeading = document.getElementById('main-heading');
const firstButton = document.querySelector('.btn');
const navLink = document.querySelector('nav a');

// Selecting multiple elements
```

```

const paragraphs = document.getElementsByTagName('p'); // HTMLCollection (live)
const buttons = document.getElementsByClassName('btn'); // HTMLCollection (live)
const listItems = document.querySelectorAll('ul.menu li'); // NodeList (not live)

// Difference between HTMLCollection and NodeList:
// - HTMLCollection is live (automatically updates when DOM changes)
// - NodeList is static (doesn't update)
// - NodeList has forEach method, HTMLCollection doesn't

// Convert collections to arrays for more methods
const paragraphsArray = Array.from(paragraphs);
const buttonsArray = [...buttons]; // Spread operator

// DOM traversal
const parent = listItems[0].parentNode; // or .parentElement for element nodes
const nextSibling = listItems[0].nextSibling; // Might be text node (whitespace)
const nextElementSibling = listItems[0].nextElementSibling; // Skip text nodes
const prevSibling = listItems[1].previousSibling;
const prevElementSibling = listItems[1].previousElementSibling;
const children = parent.children; // HTMLCollection of child elements
const childNodes = parent.childNodes; // NodeList of all child nodes (elements, text, comments)
const firstChild = parent.firstChild; // Might be text node
const firstElementChild = parent.firstElementChild; // First child that is an element
const lastChild = parent.lastChild;
const lastElementChild = parent.lastElementChild;

// Finding elements by relationship
const form = document.querySelector('form');
const formElements = form.elements; // All form controls

// Check if an element contains another
const isDescendant = parent.contains(listItems[0]); // true if direct or indirect child

// Get closest ancestor matching a selector
const closestSection = listItems[0].closest('section');

```

Manipulating DOM Elements:

```

// Changing content
const heading = document.querySelector('h1');
heading.textContent = 'New Heading'; // Text only
heading.innerText = 'Visible Text'; // Text considering CSS (not recommended)
heading.innerHTML = 'Heading with <em>emphasis</em>'; // HTML content (security risk with user input)

// Safer alternative to innerHTML for user input
function setTextSafely(element, text) {
  element.textContent = ''; // Clear existing content
  element.appendChild(document.createTextNode(text));
}

```

```

}

// Creating and adding elements
const paragraph = document.createElement('p');
paragraph.textContent = 'This is a new paragraph.';
document.body.appendChild(paragraph); // Add to end of body

// Insert at specific position
const container = document.querySelector('.container');
const referenceElement = container.querySelector('.reference');
container.insertBefore(paragraph, referenceElement); // Insert before reference

// Modern insertion methods
container.append(paragraph); // Add at end (allows multiple nodes and text)
container.prepend(paragraph); // Add at beginning
referenceElement.before(paragraph); // Insert before
referenceElement.after(paragraph); // Insert after
referenceElement.replaceWith(paragraph); // Replace

// Creating document fragments (for batch operations)
const fragment = document.createDocumentFragment();
for (let i = 0; i < 10; i++) {
  const li = document.createElement('li');
  li.textContent = `Item ${i}`;
  fragment.appendChild(li);
}
document.querySelector('ul').appendChild(fragment); // Only one DOM update

// Removing elements
paragraph.remove(); // Modern method
// Or
paragraph.parentNode.removeChild(paragraph); // Older compatible method

// Cloning elements
const clone = paragraph.cloneNode(true); // true for deep clone (with children)
container.appendChild(clone);

```

Working with Attributes and Properties:

```

const link = document.querySelector('a');

// Getting attributes
const href = link.getAttribute('href');
const id = link.id; // Direct property access for common attributes

// Setting attributes
link.setAttribute('href', 'https://example.com');
link.id = 'main-link'; // Direct property for common attributes
link.href = 'https://example.com'; // Property might be parsed (absolute URL)

// Checking attributes
const hasTarget = link.hasAttribute('target');

```

```

const hasHref = 'href' in link; // Check if property exists

// Removing attributes
link.removeAttribute('target');

// Data attributes
const element = document.querySelector('.user-card');
// <div class="user-card" data-user-id="123" data-status="active">...</div>
const userId = element.dataset.userId; // Camel-cased access to data-user-id
element.dataset.status = 'inactive'; // Sets data-status

// Classes
element.className = 'user-card highlighted'; // Replaces all classes
element.classList.add('selected'); // Add class
element.classList.remove('highlighted'); // Remove class
element.classList.toggle('expanded'); // Add if missing, remove if present
element.classList.replace('old-class', 'new-class'); // Replace one class with another
const hasClass = element.classList.contains('selected'); // Check for class

// Inline styles
element.style.color = 'red';
element.style.backgroundColor = 'lightblue'; // Camel case for CSS properties
element.style.cssText = 'color: red; background-color: lightblue;'; // Set multiple at once

// Get computed styles (actual applied styles)
const element = document.querySelector('.example');

// Get computed styles (actual applied styles after CSS)
const computed = window.getComputedStyle(element);
const fontSize = computed.fontSize; // "16px"
const display = computed.getPropertyValue('display'); // Alternative syntax

// Element dimensions and position
const rect = element.getBoundingClientRect(); // Relative to viewport
console.log(rect.width, rect.height, rect.top, rect.left);

// For entire page sizes and positions:
const pageHeight = document.documentElement.scrollHeight;
const viewportHeight = window.innerHeight;
const scrollPosition = window.scrollY || window.pageYOffset;

// Force layout recalculation (reflow) - use sparingly
element.offsetHeight; // Triggers layout calculation

```

Handling Events:

```

const button = document.querySelector('button');

// Basic event handling
button.onclick = function () {

```

```
console.log('Button clicked!');

// Better approach: addEventListener
button.addEventListener('click', function (event) {
  console.log('Button clicked!');
  console.log('Event object:', event);
});

// Using arrow functions and event object properties
button.addEventListener('click', (e) => {
  console.log(`Clicked at coordinates: ${e.clientX}, ${e.clientY}`);
  e.target.textContent = 'Clicked!'; // e.target is the element that was clicked
});

// Remove event listener (must reference same function)
const handler = () => console.log('Handled!');
button.addEventListener('click', handler);
button.removeEventListener('click', handler);

// Event propagation: capturing and bubbling
document.body.addEventListener(
  'click',
  () => {
    console.log('Body clicked (bubbling phase)');
  },
  false
); // false (default) = bubbling phase

document.body.addEventListener(
  'click',
  () => {
    console.log('Body clicked (capturing phase)');
  },
  true
); // true = capturing phase

const innerButton = document.querySelector('.inner-button');
innerButton.addEventListener('click', (e) => {
  console.log('Button clicked');
  e.stopPropagation(); // Stop event from bubbling up
});

// Preventing default behavior
const link = document.querySelector('a');
link.addEventListener('click', (e) => {
  e.preventDefault(); // Prevent navigation
  console.log('Link click prevented');
});

// Event delegation (efficient handling of multiple elements)
const list = document.querySelector('ul');
list.addEventListener('click', (e) => {
  // Check if clicked element is a list item
});
```

```
if (e.target.tagName === 'LI') {
  console.log('List item clicked:', e.target.textContent);
}

// Common events
// Mouse events
element.addEventListener('click', handleClick); // Click
element.addEventListener('dblclick', handleDblClick); // Double click
element.addEventListener('mousedown', handleMouseDown); // Mouse button pressed
element.addEventListener('mouseup', handleMouseUp); // Mouse button released
element.addEventListener('mousemove', handleMouseMove); // Mouse moved
element.addEventListener('mouseover', handleMouseOver); // Mouse entered element
element.addEventListener('mouseout', handleMouseOut); // Mouse left element
element.addEventListener('contextmenu', handleContextMenu); // Right click

// Keyboard events
document.addEventListener('keydown', (e) => {
  console.log('Key pressed:', e.key, e.code, e.keyCode);
});
document.addEventListener('keyup', handleKeyUp);
document.addEventListener('keypress', handleKeyPress); // Fired for keys that
produce character values

// Form events
const form = document.querySelector('form');
form.addEventListener('submit', (e) => {
  e.preventDefault(); // Prevent form submission
  console.log('Form submitted');
});

const input = document.querySelector('input');
input.addEventListener('focus', () => console.log('Input focused'));
input.addEventListener('blur', () => console.log('Input lost focus'));
input.addEventListener('input', () =>
  console.log('Input changed:', input.value)
);
input.addEventListener('change', () =>
  console.log('Input value committed', input.value)
);

// Document and window events
document.addEventListener('DOMContentLoaded', () => console.log('DOM loaded'));
window.addEventListener('load', () => console.log('Page fully loaded'));
window.addEventListener('resize', () => console.log('Window resized'));
window.addEventListener('scroll', () => console.log('Window scrolled'));
window.addEventListener('beforeunload', (e) => {
  // Confirm before leaving the page
  e.preventDefault();
  e.returnValue = ''; // Required for some browsers
  return ''; // Required for some browsers
});

// Custom events
```

```
const customEvent = new CustomEvent('userLogin', {
  detail: { username: 'Alice' },
  bubbles: true,
  cancelable: true,
});
document.dispatchEvent(customEvent);

// Listen for custom event
document.addEventListener('userLogin', (e) => {
  console.log('User logged in:', e.detail.username);
});
```

Working with Forms:

```
const form = document.querySelector('#login-form');
const usernameInput = form.elements.username; // Access by name attribute
const passwordInput = form.elements['password']; // Alternative syntax

// Form submission
form.addEventListener('submit', (e) => {
  e.preventDefault(); // Prevent actual form submission

  // Collect form data
  const formData = new FormData(form);

  // Different ways to access form data
  const username = usernameInput.value;
  const password = passwordInput.value;

  // Using FormData object
  console.log('Username:', formData.get('username'));
  console.log('Password:', formData.get('password'));
  console.log('Remember me:', formData.get('remember') === 'on');

  // Convert FormData to object
  const data = Object.fromEntries(formData.entries());
  console.log('Form data:', data);

  // Form validation
  let isValid = true;

  if (username.trim() === '') {
    showError(usernameInput, 'Username is required');
    isValid = false;
  }

  if (password.length < 8) {
    showError(passwordInput, 'Password must be at least 8 characters');
    isValid = false;
  }

  if (isValid) {
```

```
console.log('Form valid, submitting...');

// Submit form data to server
submitFormData(data);
}

});

function showError(input, message) {
  const formGroup = input.closest('.form-group');
  const errorElement = formGroup.querySelector('.error-message');

  if (errorElement) {
    errorElement.textContent = message;
  } else {
    const error = document.createElement('div');
    error.className = 'error-message';
    error.textContent = message;
    formGroup.appendChild(error);
  }

  input.classList.add('invalid');
}

function submitFormData(data) {
  // Example AJAX request
  fetch('/api/login', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify(data),
  })
  .then((response) => response.json())
  .then((result) => {
    console.log('Success:', result);
  })
  .catch((error) => {
    console.error('Error:', error);
  });
}

// Input validation on change
usernameInput.addEventListener('input', () => {
  if (usernameInput.value.trim() !== '') {
    usernameInput.classList.remove('invalid');
    const errorElement = usernameInput
      .closest('.form-group')
      .querySelector('.error-message');
    if (errorElement) {
      errorElement.textContent = '';
    }
  }
});

// Form reset
```

```
const resetButton = document.querySelector('#reset-button');
resetButton.addEventListener('click', () => {
  form.reset();

  // Clear all error messages
  form.querySelectorAll('.error-message').forEach((el) => {
    el.textContent = '';
  });

  form.querySelectorAll('.invalid').forEach((el) => {
    el.classList.remove('invalid');
  });
});
```

Browser APIs:

```
// Local Storage and Session Storage
// Local Storage: persists across browser sessions
localStorage.setItem('username', 'Alice');
const username = localStorage.getItem('username');
localStorage.removeItem('username');
localStorage.clear(); // Remove all items

// Session Storage: cleared when the session ends
sessionStorage.setItem('sessionId', '123456');
const sessionId = sessionStorage.getItem('sessionId');
sessionStorage.removeItem('sessionId');
sessionStorage.clear();

// Storing objects (must be serialized)
const user = { id: 1, name: 'Alice', roles: ['admin', 'user'] };
localStorage.setItem('user', JSON.stringify(user));

// Retrieving objects
const storedUser = JSON.parse(localStorage.getItem('user'));
console.log(storedUser.name, storedUser.roles); // Alice, ['admin', 'user']

// Cookies
document.cookie = 'username=Alice; max-age=3600; path=/'; // Set cookie
document.cookie = 'preference=dark; max-age=2592000; path=/'; // Expires in 30
days

// Parsing cookies
function getCookie(name) {
  const cookies = document.cookie.split('; ');
  const cookie = cookies.find((c) => c.startsWith(name + '='));
  return cookie ? cookie.split('=')[1] : null;
}

const usernameCookie = getCookie('username');
console.log('Username cookie:', usernameCookie);
```

```
// Set cookie with additional options
function setCookie(name, value, options = {}) {
  let cookie = `${name}=${value}`;

  if (options.maxAge) cookie += `; max-age=${options.maxAge}`;
  if (options.domain) cookie += `; domain=${options.domain}`;
  if (options.path) cookie += `; path=${options.path}`;
  if (options.secure) cookie += `; secure`;
  if (options.httpOnly) cookie += `; HttpOnly`;
  if (options.sameSite) cookie += `; SameSite=${options.sameSite}`;

  document.cookie = cookie;
}

setCookie('theme', 'dark', {
  maxAge: 2592000, // 30 days
  path: '/',
  sameSite: 'Strict',
});

// Delete cookie by setting expired date
function deleteCookie(name) {
  document.cookie = `${name}=; max-age=0; path=/`;
}

deleteCookie('username');

// Navigation and History API
// Navigate to a new page
// location.href = 'https://example.com';

// Reload the page
// location.reload();

// Parse URL components
console.log(location.hostname); // domain name: example.com
console.log(location.pathname); // path: /page.html
console.log(location.search); // query string: ?id=123
console.log(location.hash); // fragment: #section1

// History API
history.back(); // Go back one page
history.forward(); // Go forward one page
history.go(-2); // Go back two pages

// Add state to browser history without reloading
history.pushState({ page: 2 }, 'Page 2', '/page2');

// Replace current state
history.replaceState({ page: 3 }, 'Page 3', '/page3');

// Listen for popstate (triggered when navigating through history)
window.addEventListener('popstate', (event) => {
  console.log('State:', event.state);
```

```
updateUI(event.state);
});

// Fetch API (covered in Asynchronous JavaScript)

// Geolocation API
if ('geolocation' in navigator) {
  navigator.geolocation.getCurrentPosition(
    // Success callback
    (position) => {
      const { latitude, longitude } = position.coords;
      console.log(`Location: ${latitude}, ${longitude}`);
    },
    // Error callback
    (error) => {
      console.error('Error getting location:', error.message);
    },
    // Options
    {
      enableHighAccuracy: true,
      timeout: 5000,
      maximumAge: 0,
    }
  );
}

// Watch position (continuous updates)
const watchId = navigator.geolocation.watchPosition((position) => {
  console.log(
    `Updated location: ${position.coords.latitude},
${position.coords.longitude}`
  );
});

// Stop watching
// navigator.geolocation.clearWatch(watchId);
}

// Web Storage API (LocalStorage, SessionStorage covered above)

// IndexedDB - client-side database
const request = indexedDB.open('MyDatabase', 1);

request.onerror = function (event) {
  console.error('Database error:', event.target.error);
};

request.onupgradeneeded = function (event) {
  const db = event.target.result;

  // Create object store (table)
  const store = db.createObjectStore('users', { keyPath: 'id' });

  // Create indexes
  store.createIndex('name', 'name', { unique: false });
}
```

```
store.createIndex('email', 'email', { unique: true });
};

request.onsuccess = function (event) {
  const db = event.target.result;

  // Add data
  function addUser(user) {
    const transaction = db.transaction(['users'], 'readwrite');
    const store = transaction.objectStore('users');
    const request = store.add(user);

    request.onsuccess = function () {
      console.log('User added:', user);
    };

    transaction.oncomplete = function () {
      console.log('Transaction completed');
    };
  };

  transaction.onerror = function (event) {
    console.error('Transaction error:', event.target.error);
  };
}

// Get data
function getUser(id) {
  const transaction = db.transaction(['users']);
  const store = transaction.objectStore('users');
  const request = store.get(id);

  request.onsuccess = function () {
    console.log('User retrieved:', request.result);
  };
}

// Example usage
addUser({ id: 1, name: 'Alice', email: 'alice@example.com' });
getUser(1);
};

// Web Workers - run JavaScript in background threads
function startWorker() {
  if (typeof Worker !== 'undefined') {
    // Create a new worker
    const worker = new Worker('worker.js');

    // Send message to worker
    worker.postMessage({ data: [1, 2, 3, 4, 5], operation: 'sum' });

    // Receive message from worker
    worker.onmessage = function (event) {
      console.log('Result from worker:', event.data);
    };
  }
}
```

```
// Handle errors
worker.onerror = function (error) {
  console.error('Worker error:', error.message);
  worker.terminate();
};

// Terminate worker when done
// worker.terminate();
} else {
  console.error('Web Workers not supported');
}
}

// Example worker.js:
/*
self.onmessage = function(event) {
  const { data, operation } = event.data;

  if (operation === 'sum') {
    const result = data.reduce((sum, num) => sum + num, 0);
    self.postMessage(result);
  }
};

*/
// Intersection Observer - detect when elements enter/exit viewport
function observeElements() {
  const options = {
    root: null, // viewport
    rootMargin: '0px',
    threshold: 0.5, // 50% of element must be visible
  };

  const observer = new IntersectionObserver((entries, observer) => {
    entries.forEach((entry) => {
      if (entry.isIntersecting) {
        console.log('Element is visible:', entry.target);
        entry.target.classList.add('visible');
        // Optional: stop observing after it's visible
        // observer.unobserve(entry.target);
      } else {
        console.log('Element is not visible:', entry.target);
        entry.target.classList.remove('visible');
      }
    });
  }, options);

  // Start observing elements
  document.querySelectorAll('.observe-me').forEach((el) => {
    observer.observe(el);
  });
}
}
```

```
// MutationObserver - detect changes to DOM
function watchDOMChanges() {
  const targetNode = document.querySelector('#observed-container');

  const observer = new MutationObserver((mutations) => {
    mutations.forEach((mutation) => {
      if (mutation.type === 'childList') {
        console.log(
          'Child nodes changed:',
          mutation.addedNodes,
          mutation.removedNodes
        );
      } else if (mutation.type === 'attributes') {
        console.log(
          `Attribute ${mutation.attributeName} changed.`,
          targetNode.getAttribute(mutation.attributeName)
        );
      }
    });
  });

  const config = {
    childList: true, // Observe direct children changes
    attributes: true, // Observe attribute changes
    subtree: true, // Observe all descendants
    attributeOldValue: true, // Record old attribute values
    characterData: true, // Observe text content changes
    characterDataOldValue: true, // Record old text values
  };

  observer.observe(targetNode, config);

  // Later: stop observing
  // observer.disconnect();
}

// Canvas API - drawing graphics
function drawCanvas() {
  const canvas = document.querySelector('#myCanvas');
  const ctx = canvas.getContext('2d');

  // Set canvas size
  canvas.width = 500;
  canvas.height = 300;

  // Basic shapes
  ctx.fillStyle = 'blue';
  ctx.fillRect(10, 10, 100, 80); // x, y, width, height

  ctx.strokeStyle = 'red';
  ctx.lineWidth = 3;
  ctx.strokeRect(130, 10, 100, 80);

  // Paths
}
```

```
ctx.beginPath();
ctx.moveTo(250, 50);
ctx.lineTo(300, 90);
ctx.lineTo(350, 10);
ctx.closePath();
ctx.fillStyle = 'green';
ctx.fill();

// Circle
ctx.beginPath();
ctx.arc(400, 50, 40, 0, Math.PI * 2); // x, y, radius, startAngle, endAngle
ctx.fillStyle = 'purple';
ctx.fill();

// Text
ctx.font = '24px Arial';
ctx.fillStyle = 'black';
ctx.fillText('Hello Canvas', 180, 150);

// Gradients
const gradient = ctx.createLinearGradient(50, 180, 450, 180);
gradient.addColorStop(0, 'red');
gradient.addColorStop(0.5, 'green');
gradient.addColorStop(1, 'blue');

ctx.fillStyle = gradient;
ctx.fillRect(50, 180, 400, 80);

// Images
const img = new Image();
img.onload = function () {
  ctx.drawImage(img, 200, 220, 100, 70); // x, y, width, height
};
img.src = 'path/to/image.png'; // Set source after onload handler
}
```

Error Handling and Debugging

Conceptual Foundations

Error handling is the process of anticipating, detecting, and resolving errors in your code. Effective error handling improves the robustness of your applications and provides better user experiences.

Mental Model: Think of error handling as a safety net. Just as a safety net catches a falling trapeze artist, error handling catches unexpected issues before they crash your entire program. It allows you to fail gracefully and recover when possible.

Common Misconceptions:

- Error handling is just for production code (it's also essential for development)
- Try/catch blocks slow down code significantly (modern JavaScript engines optimize this)
- All errors should be caught (some errors, like syntax errors, can't be caught at runtime)

- `Console.log` is sufficient for debugging (proper debugging tools provide much more insight)

Practical Implementation

Error Types:

```
// Common built-in error types

// SyntaxError - code cannot be parsed
// const user = { name: "Alice" ; // Missing closing brace

// ReferenceError - reference to an undefined variable
try {
  console.log(undefinedVariable);
} catch (error) {
  console.error('Reference error:', error.message);
}

// TypeError - operation on a value of the wrong type
try {
  const num = 42;
  num.toUpperCase(); // Numbers don't have this method
} catch (error) {
  console.error('Type error:', error.message);
}

// RangeError - value outside valid range
try {
  const arr = new Array(-1); // Invalid array length
} catch (error) {
  console.error('Range error:', error.message);
}

// URIError - improper use of URI functions
try {
  decodeURIComponent('%'); // Invalid URI
} catch (error) {
  console.error('URI error:', error.message);
}

// EvalError - error in eval() function (rare in modern JS)

// Custom Error - for application-specific errors
class ValidationError extends Error {
  constructor(message, field) {
    super(message);
    this.name = 'ValidationError';
    this.field = field;
  }
}

try {
```

```
const age = -5;
if (age < 0) {
  throw new ValidationError('Age cannot be negative', 'age');
}
} catch (error) {
  if (error instanceof ValidationError) {
    console.error(`Validation error on field ${error.field}:`, error.message);
  } else {
    console.error('Unexpected error:', error);
  }
}

// Capturing error stack traces
function captureStackTrace() {
  try {
    throw new Error('Custom error');
  } catch (error) {
    console.log(error.stack);
  }
}
```

Try-Catch-Finally:

```
// Basic try-catch
try {
  // Code that might throw an error
  const data = JSON.parse('{"name": "Alice"}'); // Valid JSON
  console.log('Data parsed successfully:', data);
} catch (error) {
  // Handle error
  console.error('Error parsing JSON:', error.message);
}

// Try-catch-finally
function processFile(path) {
  let file = null;

  try {
    file = openFile(path);
    return processData(file.data);
  } catch (error) {
    console.error('Error processing file:', error);
    return null;
  } finally {
    // Runs regardless of try/catch outcome
    if (file) {
      file.close();
      console.log('File closed');
    }
  }
}
```

```

// Nested try-catch blocks
try {
  try {
    throw new Error('Inner error');
  } catch (innerError) {
    console.error('Inner catch:', innerError.message);
    throw innerError; // Re-throw to outer catch
  }
} catch (outerError) {
  console.error('Outer catch:', outerError.message);
}

// Try-catch with conditional catching
try {
  // Code that might throw different types of errors
  throwRandomError();
} catch (error) {
  if (error instanceof TypeError) {
    console.error('Type error:', error.message);
  } else if (error instanceof ReferenceError) {
    console.error('Reference error:', error.message);
  } else {
    console.error('Unknown error:', error);
  }
}

// Async/await error handling
async function fetchUserData() {
  try {
    const response = await fetch('/api/user');

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

    const data = await response.json();
    return data;
  } catch (error) {
    console.error('Error fetching user data:', error);
    // Re-throw or return fallback
    throw error; // Re-throw for caller to handle
    // return { name: 'Unknown', isDefault: true }; // Or return fallback
  }
}

```

Error Propagation and Handling Patterns:

```

// Error propagation
function validateInput(input) {
  if (typeof input !== 'string') {
    throw new TypeError('Input must be a string');
  }
}

```

```
if (input.length === 0) {
  throw new ValidationError('Input cannot be empty');
}

return input;
}

function processInput(input) {
try {
  // This function may throw, and we let it propagate up
  const validInput = validateInput(input);
  return `Processed: ${validInput}`;
} catch (error) {
  // Only handle specific errors, let others propagate
  if (error instanceof ValidationError) {
    console.warn('Validation issue:', error.message);
    return null;
  }
  throw error; // Re-throw errors we don't handle
}
}

function handleFormSubmit() {
try {
  const userInput = document.getElementById('userInput').value;
  const result = processInput(userInput);

  if (result) {
    displayResult(result);
  } else {
    displayError('Please provide a valid input');
  }
} catch (error) {
  console.error('Form submission error:', error);
  displayError('An unexpected error occurred');
}
}

// Error-first callback pattern (Node.js style)
function readFile(path, callback) {
// Simulate async file reading
setTimeout(() => {
  try {
    if (!path.endsWith('.txt')) {
      return callback(new Error('Only .txt files are supported'));
    }

    const data = 'File content';
    callback(null, data);
  } catch (error) {
    callback(error);
  }
}, 1000);
}
```

```

}

// Usage of error-first callback
readFile('example.txt', (error, data) => {
  if (error) {
    console.error('Error reading file:', error.message);
    return;
  }

  console.log('File data:', data);
});

// Promise error handling (continuation from Asynchronous section)
function fetchData(url) {
  return fetch(url)
    .then((response) => {
      if (!response.ok) {
        throw new Error(`HTTP error: ${response.status}`);
      }
      return response.json();
    })
    .catch((error) => {
      console.error('Fetch error:', error);
      throw error; // Re-throw to propagate to caller
    });
}

// Global error handling
// For uncaught exceptions in browser
window.addEventListener('error', (event) => {
  console.error('Global error:', event.error);
  // Send to error tracking service
  // reportErrorToAnalytics(event.error);
  event.preventDefault(); // Prevent browser's default error handling
});

// For unhandled promise rejections
window.addEventListener('unhandledrejection', (event) => {
  console.error('Unhandled promise rejection:', event.reason);
  // Send to error tracking service
  // reportErrorToAnalytics(event.reason);
  event.preventDefault();
});

```

Debugging Techniques:

```

// Using console methods
console.log('Basic log message');
console.info('Informational message');
console.warn('Warning message');
console.error('Error message');

```

```
// Structured data
console.table([
  { name: 'Alice', age: 30 },
  { name: 'Bob', age: 25 },
]);

// Grouping logs
console.group('User Details');
console.log('Name: Alice');
console.log('Age: 30');
console.groupEnd();

// Performance measurement
console.time('loop');
for (let i = 0; i < 1000000; i++) {
  // Some operation
}
console.timeEnd('loop');

// Count occurrences
function processItem(item) {
  console.count(`Processed item: ${item.type}`);
  // Process item
}

// Stack traces
console.trace('Tracing function calls');

// Conditional logging with assert
console.assert(1 === 2, 'This will show, 1 is not equal to 2');
console.assert(1 === 1, 'This will not show, condition is true');

// Debugger statement
function buggyFunction(input) {
  let result = input * 2;
  debugger; // Browser will pause execution here when dev tools are open
  return result;
}

// Setting breakpoints programmatically
// Place this where you want to break conditionally
if (someCondition) {
  debugger;
}

// Chrome DevTools commands in console
// $_ - last evaluated expression
// $0 - $4 - last inspected elements
// $(selector) - document.querySelector()
// $$(selector) - document.querySelectorAll()
// $x(path) - XPath selection
// copy() - copy to clipboard
// keys(obj), values(obj) - get object keys/values
// monitor(function) - log function calls
```

```
// monitorEvents(element, events) - log events
// profile(), profileEnd() - CPU profiling

// Using try/catch for debugging
function debugValue(value, label = 'Debug') {
  try {
    console.log(`#${label}:`, value);
    console.log(`Type:`, typeof value);

    if (value && typeof value === 'object') {
      console.log(`Properties:`, Object.keys(value));
      console.table(value);
    }
  }

  return value;
} catch (error) {
  console.error(`Error debugging ${label}:`, error);
  return value;
}
}

// Debug async code with async/await
async function debugAsync() {
  try {
    const response = await fetch('/api/data');
    debugValue(response, 'Response');

    const data = await response.json();
    debugValue(data, 'Data');

    return data;
  } catch (error) {
    console.error('Async error:', error);
    throw error;
  }
}
```

Common Debugging Scenarios:

```
// Debugging scope issues
function outerFunction() {
  const outerVar = 'I am outer';

  function innerFunction() {
    const innerVar = 'I am inner';
    console.log(outerVar); // Accessible (closure)
  }

  innerFunction();
  // console.log(innerVar); // ReferenceError - not accessible
}
```

```
// Debugging this context
const user = {
  name: 'Alice',
  greet: function () {
    console.log(`Hello, my name is ${this.name}`);
  },
  greetArrow: () => {
    console.log(`Arrow: Hello, my name is ${this.name}`); // 'this' is not from
user
  },
};

user.greet(); // Works: "Hello, my name is Alice"
user.greetArrow(); // Might not work as expected

const greetFn = user.greet;
// greetFn(); // "Hello, my name is undefined" ('this' is global/window)

// Debugging asynchronous code
function asyncDebug() {
  console.log('1. Before setTimeout');

  setTimeout(() => {
    console.log('2. Inside setTimeout');
  }, 0);

  Promise.resolve().then(() => {
    console.log('3. Inside Promise.then');
  });

  console.log('4. After setTimeout and Promise');
}

// Output: 1, 4, 3, 2 (due to event loop priorities)

// Memory leaks debugging
function createElements() {
  const elements = [];

  for (let i = 0; i < 10000; i++) {
    const el = document.createElement('div');
    el.textContent = `Element ${i}`;

    // Bad: Global reference prevents garbage collection
    // window.lastElement = el;

    // Memory leak: keeping reference to all elements
    elements.push(el);

    // Not appending to DOM, just creating elements
  }

  // Memory leak: not releasing the array
  return elements;
}
```

```
// To detect with DevTools:  
// 1. Take heap snapshot before  
// 2. Run function  
// 3. Take heap snapshot after  
// 4. Compare snapshots
```

JavaScript Modules and Bundling

Conceptual Foundations

JavaScript modules are a way to organize code into reusable, independent pieces. Each module encapsulates related code, hiding private implementation details and exposing a public API. Bundling is the process of combining multiple modules (and their dependencies) into fewer files for optimized delivery to the browser.

Mental Model: Think of modules like chapters in a book, each focused on a specific topic with references to other chapters when needed. Bundling is like creating a condensed version of the book that contains only the chapters you need, optimized for faster reading.

Historical Context: JavaScript originally had no built-in module system. Developers created patterns like the Immediately Invoked Function Expression (IIFE) and the Revealing Module Pattern to simulate modules. AMD (Asynchronous Module Definition) and CommonJS emerged as competing module formats, with CommonJS becoming popular in Node.js. ES2015 (ES6) finally introduced native JavaScript modules, which are now the standard.

Common Misconceptions:

- ES modules and CommonJS modules are interchangeable (they have different syntax and behaviors)
- Bundling is only for file size optimization (it also handles dependencies and compatibility)
- Modules are only useful for large projects (even small projects benefit from modular code)
- All browsers support ES modules (older browsers require transpilation and bundling)

Practical Implementation

Module Formats:

```
// Early module patterns (pre-ES6)  
  
// IIFE (Immediately Invoked Function Expression)  
const counter = (function () {  
    // Private variables  
    let count = 0;  
  
    // Return public API  
    return {  
        increment() {  
            count++;  
            return count;  
        },  
        decrement() {  
            count--;  
            return count;  
        },  
        reset() {  
            count = 0;  
            return count;  
        }  
    };  
});
```

```
        count--;
        return count;
    },
    getCount() {
        return count;
    },
},
());
})();

console.log(counter.getCount()); // 0
counter.increment();
console.log(counter.getCount()); // 1

// CommonJS (used in Node.js)
// In math.js:
function add(a, b) {
    return a + b;
}

function subtract(a, b) {
    return a - b;
}

module.exports = {
    add,
    subtract,
};

// In main.js:
const math = require('./math');
console.log(math.add(5, 3)); // 8

// AMD (Asynchronous Module Definition, used with RequireJS)
// In helper.js:
define([], function () {
    return {
        formatDate(date) {
            return date.toLocaleDateString();
        },
    };
});

// In app.js:
define(['./helper'], function (helper) {
    const formattedDate = helper.formatDate(new Date());
    return {
        getFormattedDate() {
            return formattedDate;
        },
    };
});

// UMD (Universal Module Definition - works in multiple environments)
(function (root, factory) {
```

```
if (typeof define === 'function' && define.amd) {
    // AMD
    define(['jquery'], factory);
} else if (typeof module === 'object' && module.exports) {
    // CommonJS
    module.exports = factory(require('jquery'));
} else {
    // Browser global
    root.myModule = factory(root.jQuery);
}
})(typeof self !== 'undefined' ? self : this, function ($) {
    return {
        name: 'myModule',
        doSomething() {
            return $('body').text();
        },
    };
});
```

ES Modules (ESM):

```
// Named exports and imports
// In math.js:
export function add(a, b) {
    return a + b;
}

export function subtract(a, b) {
    return a - b;
}

export const PI = 3.14159;

// In another file:
import { add, subtract, PI } from './math.js';
console.log(add(5, 3)); // 8
console.log(PI); // 3.14159

// Import all exports as a namespace
import * as math from './math.js';
console.log(math.add(5, 3)); // 8
console.log(math.PI); // 3.14159

// Default exports and imports
// In user.js:
export default class User {
    constructor(name) {
        this.name = name;
    }

    sayHello() {
        return `Hello, I'm ${this.name}`;
    }
}
```

```
    }

}

// In another file:
import User from './user.js';
const user = new User('Alice');
console.log(user.sayHello()); // "Hello, I'm Alice"

// Mixing default and named exports
// In utils.js:
export default function helper() {
    return 'I am the default export';
}

export function format(str) {
    return str.trim().toUpperCase();
}

export const VERSION = '1.0.0';

// In another file:
import helper, { format, VERSION } from './utils.js';
console.log(helper()); // "I am the default export"
console.log(format(' hello ')); // "HELLO"

// Re-exporting from another module
// In index.js:
export { add, subtract } from './math.js';
export { default as User } from './user.js';

// Dynamic imports (lazy loading)
button.addEventListener('click', async () => {
    try {
        // Load module on demand
        const { formatDate } = await import('./date-utils.js');
        const formatted = formatDate(new Date());
        console.log(formatted);
    } catch (error) {
        console.error('Failed to load module:', error);
    }
});

// Module side effects (code that runs when the module is imported)
// In logger.js:
console.log('Logger module initialized');
export function log(message) {
    console.log(`[LOG]: ${message}`);
}

// When importing logger.js, "Logger module initialized" will run
import { log } from './logger.js';

// Import and export aliases
import { add as sum, subtract as minus } from './math.js';
```

```
console.log(sum(5, 3)); // 8

// In aggregator.js:
import { format } from './utils.js';
export { format as formatString };
```

Module Patterns and Best Practices:

```
// Singleton pattern
// In singleton.js:
let instance = null;

class DatabaseConnection {
  constructor() {
    if (instance) {
      return instance;
    }

    instance = this;
    this.connected = false;
  }

  connect() {
    this.connected = true;
    return this;
  }
}

export default new DatabaseConnection();

// Usage:
import db from './singleton.js';
db.connect();

// Model-View separation
// In userModel.js:
export class UserModel {
  constructor(data) {
    this.id = data.id;
    this.name = data.name;
    this.email = data.email;
  }

  static async fetchById(id) {
    const response = await fetch(`/api/users/${id}`);
    const data = await response.json();
    return new UserModel(data);
  }

  update(changes) {
    Object.assign(this, changes);
    return this.save();
  }
}
```

```
}

async save() {
    // API call to update user
    return this;
}
}

// In userView.js:
import { UserModel } from './userModel.js';

export class UserView {
    constructor(containerId) {
        this.container = document.getElementById(containerId);
    }

    async renderUser(userId) {
        try {
            const user = await UserModel.fetchById(userId);
            this.container.innerHTML =
                `

<h2>${user.name}</h2>
                    <p>${user.email}</p>
                </div>
            `;
            return user;
        } catch (error) {
            this.renderError(error);
        }
    }
}

renderError(error) {
    this.container.innerHTML =
        `

Failed to load user: ${error.message}
        </div>
    `;
}

// Factory pattern
// In elementFactory.js:
export function createElement(type, props = {}, children = []) {
    const element = document.createElement(type);

    // Apply properties
    Object.entries(props).forEach(([key, value]) => {
        if (key === 'className') {
            element.className = value;
        } else if (key === 'style' && typeof value === 'object') {
            Object.assign(element.style, value);
        } else if (key.startsWith('on') && typeof value === 'function') {
            const eventName = key.slice(2).toLowerCase(); // 'onClick' -> 'click'
            element.addEventListener(eventName, value);
        }
    });
}


```

```
    } else {
      element.setAttribute(key, value);
    }
  });

// Add children
children.forEach((child) => {
  if (typeof child === 'string') {
    element.appendChild(document.createTextNode(child));
  } else {
    element.appendChild(child);
  }
});

return element;
}

// Repository pattern (data access layer)
// In userRepository.js:
export class UserRepository {
  async findById(id) {
    const response = await fetch(`/api/users/${id}`);
    if (!response.ok) {
      throw new Error(`Failed to fetch user: ${response.statusText}`);
    }
    return response.json();
  }

  async findAll() {
    const response = await fetch('/api/users');
    if (!response.ok) {
      throw new Error(`Failed to fetch users: ${response.statusText}`);
    }
    return response.json();
  }

  async create(userData) {
    const response = await fetch('/api/users', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(userData),
    });

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

    return response.json();
  }

  // Other CRUD operations...
}

// Service pattern
```

```
// In authService.js:  
import { UserRepository } from './userRepository.js';  
  
export class AuthService {  
  constructor() {  
    this.userRepository = new UserRepository();  
    this.currentUser = null;  
  }  
  
  async login(email, password) {  
    try {  
      const response = await fetch('/api/login', {  
        method: 'POST',  
        headers: { 'Content-Type': 'application/json' },  
        body: JSON.stringify({ email, password }),  
      });  
  
      if (!response.ok) {  
        throw new Error('Invalid credentials');  
      }  
  
      const { token, userId } = await response.json();  
      localStorage.setItem('auth_token', token);  
  
      this.currentUser = await this.userRepository.findById(userId);  
      return this.currentUser;  
    } catch (error) {  
      console.error('Login failed:', error);  
      throw error;  
    }  
  }  
  
  logout() {  
    localStorage.removeItem('auth_token');  
    this.currentUser = null;  
  }  
  
  async checkAuthStatus() {  
    const token = localStorage.getItem('auth_token');  
    if (!token) return null;  
  
    try {  
      const response = await fetch('/api/verify-token', {  
        headers: { Authorization: `Bearer ${token}` },  
      });  
  
      if (!response.ok) {  
        this.logout();  
        return null;  
      }  
  
      const { userId } = await response.json();  
      this.currentUser = await this.userRepository.findById(userId);  
      return this.currentUser;  
    }  
  }  
}
```

```

    } catch (error) {
      console.error('Auth check failed:', error);
      this.logout();
      return null;
    }
  }
}

// Using these patterns together
// In app.js:
import { UserView } from './userView.js';
import { AuthService } from './authService.js';

export async function initializeApp() {
  const authService = new AuthService();
  const userView = new UserView('user-container');

  // Check if user is already logged in
  const user = await authService.checkAuthStatus();

  if (user) {
    userView.renderUser(user.id);
  } else {
    // Show login form
    document.getElementById('login-form').style.display = 'block';
  }

  // Set up event listeners
  document
    .getElementById('login-form')
    .addEventListener('submit', async (e) => {
      e.preventDefault();
      const email = document.getElementById('email').value;
      const password = document.getElementById('password').value;

      try {
        const user = await authService.login(email, password);
        userView.renderUser(user.id);
        document.getElementById('login-form').style.display = 'none';
      } catch (error) {
        alert(`Login failed: ${error.message}`);
      }
    });
}

```

Module Bundling with Webpack:

```

// Basic webpack.config.js
const path = require('path');

module.exports = {
  entry: './src/index.js', // Entry point

```

```
output: {
  filename: 'bundle.js',
  path: path.resolve(__dirname, 'dist'),
},
mode: 'development', // or 'production'
module: {
  rules: [
    {
      test: /\.js$/,
      exclude: /node_modules/,
      use: {
        loader: 'babel-loader', // For transpiling modern JS
        options: {
          presets: ['@babel/preset-env'],
        },
      },
    },
    {
      test: /\.css$/,
      use: ['style-loader', 'css-loader'], // Handle CSS files
    },
    {
      test: /\.(png|svg|jpg|gif)$/,
      use: ['file-loader'], // Handle image files
    },
  ],
},
devServer: {
  contentBase: './dist',
  port: 8080,
},
plugins: [
  // Add plugins like HTMLWebpackPlugin, etc.
],
};

// Example NPM scripts in package.json
/*
{
  "scripts": {
    "build": "webpack",
    "start": "webpack serve",
    "build:prod": "webpack --mode=production"
  }
}
*/
// Using code splitting in webpack
// Dynamic import - webpack will create a separate chunk
import('./large-module.js').then((module) => {
  module.doSomething();
});

// Multiple entry points
```

```
/*
module.exports = {
  entry: {
    main: './src/index.js',
    admin: './src/admin.js'
  },
  output: {
    filename: '[name].bundle.js',
    path: path.resolve(__dirname, 'dist')
  }
}
*/
```

Other Bundlers and Build Tools:

```
// Rollup.js - config example
// rollup.config.js
import resolve from '@rollup/plugin-node-resolve';
import commonjs from '@rollup/plugin-commonjs';
import babel from '@rollup/plugin-babel';
import { terser } from 'rollup-plugin-terser';

export default {
  input: 'src/main.js',
  output: {
    file: 'dist/bundle.js',
    format: 'esm', // or 'cjs', 'iife', 'umd'
    sourcemap: true
  },
  plugins: [
    resolve(), // Locate modules using the Node resolution algorithm
    commonjs(), // Convert CommonJS modules to ES6
    babel({ babelHelpers: 'bundled' }),
    terser() // Minify
  ]
};

// Parcel - zero configuration bundler
// Just reference your entry HTML file:
// $ parcel index.html

// Snowpack - designed for modern web development
// snowpack.config.js
module.exports = {
  mount: {
    public: '/',
    src: '/dist'
  },
  plugins: [
    '@snowpack/plugin-react-refresh',
    '@snowpack/plugin-dotenv'
  ],
}
```

```
packageOptions: {  
    /* ... */  
},  
devOptions: {  
    port: 8080  
},  
buildOptions: {  
    /* ... */  
}  
};  
  
// Vite - next-generation frontend tooling  
// vite.config.js  
import { defineConfig } from 'vite';  
import react from '@vitejs/plugin-react';  
  
export default defineConfig({  
    plugins: [react()],  
    server: {  
        port: 3000  
    },  
    build: {  
        outDir: 'dist',  
        minify: 'terser'  
    }  
});
```

Performance Optimization and Memory Management

Conceptual Foundations

Performance optimization in JavaScript focuses on making your code run faster and use fewer resources.

Memory management involves efficiently allocating, using, and releasing memory to prevent leaks and reduce garbage collection overhead.

Mental Model: Think of JavaScript performance like maintaining a race car. Just as a race car needs regular tuning and optimization to maintain peak performance, JavaScript code requires thoughtful design and periodic optimization to run efficiently. Memory management is like resource management – ensuring you're not wasting fuel (memory) or leaving parts lying around (memory leaks).

Historical Context: JavaScript performance has evolved significantly. Early browsers had basic JavaScript engines with limited optimization. Modern engines like V8 (Chrome), SpiderMonkey (Firefox), and JavaScriptCore (Safari) use just-in-time (JIT) compilation to convert JavaScript into optimized machine code.

Common Misconceptions:

- Premature optimization is useful (it often wastes time and complicates code)
- Modern JavaScript engines optimize everything (they can't optimize certain patterns)
- All memory will be automatically garbage collected (references can prevent collection)
- More memory always means better performance (excessive memory use can degrade performance)

Practical Implementation

Performance Optimization Techniques:

```
// Avoid unnecessary DOM operations
// Bad approach: many separate DOM updates
function renderList(items) {
  const list = document.getElementById('list');

  // Clearing the list
  list.innerHTML = '';

  // Adding items one by one - causes multiple reflows
  items.forEach((item) => {
    list.innerHTML += `<li>${item}</li>`;
  });
}

// Better approach: batch DOM operations
function renderListOptimized(items) {
  const list = document.getElementById('list');

  // Create fragment (doesn't cause reflows)
  const fragment = document.createDocumentFragment();

  // Build list in memory
  items.forEach((item) => {
    const li = document.createElement('li');
    li.textContent = item;
    fragment.appendChild(li);
  });

  // Clear and update list with a single operation
  list.innerHTML = '';
  list.appendChild(fragment);
}

// Even better: only update what changed
function renderListWithDiff(newItems, oldItems = []) {
  const list = document.getElementById('list');
  const existingItems = list.querySelectorAll('li');

  // Update existing items
  for (let i = 0; i < Math.min(newItems.length, existingItems.length); i++) {
    if (newItems[i] !== oldItems[i]) {
      existingItems[i].textContent = newItems[i];
    }
  }

  // Remove extra items
  for (let i = newItems.length; i < existingItems.length; i++) {
    list.removeChild(existingItems[i]);
  }
}
```

```
}

// Add new items
if (newItems.length > existingItems.length) {
  const fragment = document.createDocumentFragment();

  for (let i = existingItems.length; i < newItems.length; i++) {
    const li = document.createElement('li');
    li.textContent = newItems[i];
    fragment.appendChild(li);
  }

  list.appendChild(fragment);
}

return newItems; // Return for next comparison
}

// Optimize loops
// Bad: accessing length on each iteration
function sumArray(arr) {
  let sum = 0;
  for (let i = 0; i < arr.length; i++) {
    sum += arr[i];
  }
  return sum;
}

// Better: cache the length
function sumArrayOptimized(arr) {
  let sum = 0;
  const length = arr.length;
  for (let i = 0; i < length; i++) {
    sum += arr[i];
  }
  return sum;
}

// Modern: array methods (may be less verbose but not always faster)
function sumArrayModern(arr) {
  return arr.reduce((sum, num) => sum + num, 0);
}

// Reduce object property lookups
// Bad: repeated lookups
function processUser(user) {
  console.log(user.profile.name);
  console.log(user.profile.email);
  updateProfile(user.profile.name, user.profile.email);
}

// Better: destructure or cache
function processUserOptimized(user) {
  const { name, email } = user.profile;
```

```
console.log(name);
console.log(email);
updateProfile(name, email);
}

// Optimize recursive functions
// Bad: recalculates the same values repeatedly
function fibonacci(n) {
  if (n <= 1) return n;
  return fibonacci(n - 1) + fibonacci(n - 2);
}

// Better: memoization
function fibonacciMemoized(n, memo = {}) {
  if (n <= 1) return n;
  if (memo[n]) return memo[n];

  memo[n] = fibonacciMemoized(n - 1, memo) + fibonacciMemoized(n - 2, memo);
  return memo[n];
}

// Or with closure
function createFibonacci() {
  const memo = {};

  return function fib(n) {
    if (n <= 1) return n;
    if (memo[n]) return memo[n];

    memo[n] = fib(n - 1) + fib(n - 2);
    return memo[n];
  };
}

const fib = createFibonacci();
console.log(fib(40)); // Fast calculation

// Avoid forced synchronous layouts (layout thrashing)
// Bad: reading then writing then reading layout properties
function badAnimation() {
  const elements = document.querySelectorAll('.animated');

  elements.forEach((el) => {
    const height = el.offsetHeight; // Read
    el.style.height = height * 2 + 'px'; // Write

    const width = el.offsetWidth; // Read - forces layout recalculation
    el.style.width = width * 2 + 'px'; // Write
  });
}

// Better: batch reads and writes
function goodAnimation() {
  const elements = document.querySelectorAll('.animated');
```

```
const dimensions = [];

// Batch all reads
elements.forEach((el) => {
  dimensions.push({
    height: el.offsetHeight,
    width: el.offsetWidth,
  });
});

// Batch all writes
elements.forEach((el, i) => {
  const { height, width } = dimensions[i];
  el.style.height = height * 2 + 'px';
  el.style.width = width * 2 + 'px';
});

// Debounce and throttle for performance
// Debounce: only execute after a quiet period
function debounce(func, wait) {
  let timeout;

  return function (...args) {
    const context = this;
    clearTimeout(timeout);

    timeout = setTimeout(() => {
      func.apply(context, args);
    }, wait);
  };
}

// Throttle: limit execution rate
function throttle(func, limit) {
  let inThrottle;

  return function (...args) {
    const context = this;

    if (!inThrottle) {
      func.apply(context, args);
      inThrottle = true;

      setTimeout(() => {
        inThrottle = false;
      }, limit);
    }
  };
}

// Usage
const debouncedResize = debounce(() => {
  console.log('Resized finished');
});
```

```
}, 300);

const throttledScroll = throttle(() => {
  console.log('Scroll event (throttled)');
}, 100);

window.addEventListener('resize', debouncedResize);
window.addEventListener('scroll', throttledScroll);

// Web Workers for CPU-intensive tasks
// main.js
function startWorker() {
  const worker = new Worker('worker.js');

  worker.postMessage({
    numbers: Array.from({ length: 10000000 }, (_, i) => i),
  });

  worker.onmessage = function (e) {
    console.log('Sum result:', e.data.result);
    worker.terminate();
  };
}

// worker.js
/*
self.onmessage = function(e) {
  const { numbers } = e.data;

  // Simulating CPU-intensive calculation
  const sum = numbers.reduce((acc, num) => acc + num, 0);

  self.postMessage({ result: sum });
};

*/
// Web Animations API for smooth animations
function animateElement() {
  const element = document.querySelector('.animated-box');

  element.animate(
    [
      // Keyframes
      { transform: 'translateX(0px)' },
      { transform: 'translateX(300px)' },
    ],
    {
      // Timing options
      duration: 1000,
      easing: 'ease-in-out',
      iterations: Infinity,
      direction: 'alternate',
    }
  );
}
```

```
}

// Virtual scrolling for long lists
function createVirtualScroller(container, itemHeight, totalItems, renderItem) {
  const viewportHeight = container.clientHeight;
  const visibleItems = Math.ceil(viewportHeight / itemHeight) + 2; // +2 for
buffer

  let startIndex = 0;
  let scrollTop = 0;

  // Create content height
  const content = document.createElement('div');
  content.style.height = `${totalItems * itemHeight}px`;
  content.style.position = 'relative';
  container.appendChild(content);

  function render() {
    // Clear content
    content.innerHTML = '';

    // Calculate visible range
    startIndex = Math.floor(scrollTop / itemHeight);
    const endIndex = Math.min(startIndex + visibleItems, totalItems);

    // Render only visible items
    for (let i = startIndex; i < endIndex; i++) {
      const itemEl = renderItem(i);
      itemEl.style.position = 'absolute';
      itemEl.style.top = `${i * itemHeight}px`;
      itemEl.style.height = `${itemHeight}px`;
      itemEl.style.width = '100%';
      content.appendChild(itemEl);
    }
  }

  // Listen for scroll events
  container.addEventListener('scroll', () => {
    scrollTop = container.scrollTop;
    render();
  });

  // Initial render
  render();

  return {
    scrollToIndex(index) {
      container.scrollTop = index * itemHeight;
    },
    refresh() {
      render();
    },
  };
}
```

Memory Management and Leak Prevention:

```
// Common memory leak: forgotten event listeners
// Bad: not removing event listeners
function createButton() {
  const button = document.createElement('button');
  button.textContent = 'Click me';

  // Event handler captures variables in closure
  button.addEventListener('click', function () {
    console.log('Button clicked');
  });

  return button;
}

// When you later remove the button from DOM, the event handler still exists in
// memory

// Better: clean up when done
function createButtonWithCleanup() {
  const button = document.createElement('button');
  button.textContent = 'Click me';

  const handleClick = function () {
    console.log('Button clicked');
  };

  button.addEventListener('click', handleClick);

  // Return cleanup function
  return {
    element: button,
    cleanup() {
      button.removeEventListener('click', handleClick);
    },
  };
}

const { element, cleanup } = createButtonWithCleanup();
document.body.appendChild(element);

// Later when removing
document.body.removeChild(element);
cleanup();

// Preventing circular references
// Bad: circular references can cause memory leaks
function createCircularReference() {
  const parent = {};
  const child = {};
  parent.child = child;
  child.parent = parent;
}
```

```
parent.child = child;
child.parent = parent; // Circular reference

return parent;
}

// Better: weak references for back-references
function createWeakReference() {
  const parent = {};
  const child = {};

  parent.child = child;

  // Use WeakMap for back-reference
  const parentRefs = new WeakMap();
  parentRefs.set(child, parent);

  return {
    parent,
    getParentOf(childObj) {
      return parentRefs.get(childObj);
    },
  };
}

// Using WeakSet and WeakMap to prevent leaks
// WeakSet: weakly holds objects (good for "has" checks)
const visitedObjects = new WeakSet();

function processObject(obj) {
  if (visitedObjects.has(obj)) {
    console.log('Already processed');
    return;
  }

  visitedObjects.add(obj);
  console.log('Processing object');
}

// WeakMap: weakly holds keys (good for metadata)
const objectMetadata = new WeakMap();

function addMetadata(obj, metadata) {
  objectMetadata.set(obj, metadata);
}

function getMetadata(obj) {
  return objectMetadata.get(obj);
}

// Proper cleanup when removing DOM elements
function setupComponent(container) {
  const button = document.createElement('button');
```

```
const animation = button.animate(
  [{ transform: 'scale(1)' }, { transform: 'scale(1.2)' }],
  { duration: 1000, iterations: Infinity }
);

const interval = setInterval(() => {
  console.log('Component check');
}, 1000);

const handleClick = () => console.log('Button clicked');
button.addEventListener('click', handleClick);

container.appendChild(button);

// Return cleanup function
return function cleanup() {
  // Stop animations
  animation.cancel();

  // Clear intervals/timeouts
  clearInterval(interval);

  // Remove event listeners
  button.removeEventListener('click', handleClick);

  // Remove from DOM
  if (container.contains(button)) {
    container.removeChild(button);
  }
};

}

// Using the cleanup
const cleanup = setupComponent(document.getElementById('container'));

// Later:
cleanup();

// Managing large data structures
// Bad: holding entire dataset in memory
function loadData(url) {
  fetch(url)
    .then((response) => response.json())
    .then((data) => {
      // Store entire dataset in memory
      window.appData = data; // Potentially very large
    });
}

// Better: pagination and virtualization
function loadDataPaginated(url, pageSize = 50) {
  let allData = []; // Not visible outside
  let currentPage = 0;
```

```
function loadPage(page) {
  return fetch(` ${url}?page=${page}&size=${pageSize}`).then((response) =>
  response.json()
);

}

return {
  async getPage(page) {
    currentPage = page;
    return loadPage(page);
  },
  async nextPage() {
    return this.getPage(currentPage + 1);
  },
  async prevPage() {
    if (currentPage > 0) {
      return this.getPage(currentPage - 1);
    }
    return null;
  },
};
}

// Memory-efficient data processing with generators
function* processLargeArray(array) {
  // Process elements one at a time without loading all results in memory
  for (const item of array) {
    yield transformItem(item);
  }
}

function transformItem(item) {
  // Expensive operation
  return item * 2;
}

// Usage
const largeArray = Array.from({ length: 1000000 }, (_, i) => i);

// Process without storing all results
const generator = processLargeArray(largeArray);

// Get results as needed
for (let i = 0; i < 10; i++) {
  console.log(generator.next().value);
}

// Releasing references to enable garbage collection
function processData() {
  // Large temporary data
  const hugeData = new Array(1000000).fill('data');

  // Do something with hugeData
  const result = hugeData.length;
```

```
// Explicitly clean up large objects when done
// hugeData = null; // This doesn't help with local variables

// Return result without references to large data
return result;
} // hugeData goes out of scope here and can be garbage collected

// Using object pools for frequently created objects
class ObjectPool {
    constructor(createFn, resetFn, initialSize = 10) {
        this.createFn = createFn;
        this.resetFn = resetFn;
        this.pool = [];

        // Pre-populate pool
        for (let i = 0; i < initialSize; i++) {
            this.pool.push(this.createFn());
        }
    }

    get() {
        if (this.pool.length === 0) {
            return this.createFn();
        }
        return this.pool.pop();
    }

    release(obj) {
        this.resetFn(obj);
        this.pool.push(obj);
    }

    size() {
        return this.pool.length;
    }
}

// Example: Particle system
const particlePool = new ObjectPool(
    // Create function
    () => ({ x: 0, y: 0, vx: 0, vy: 0, age: 0, active: false }),
    // Reset function
    (particle) => {
        particle.x = 0;
        particle.y = 0;
        particle.vx = 0;
        particle.vy = 0;
        particle.age = 0;
        particle.active = false;
    },
    100 // Initial size
);
```

```

function createParticle(x, y) {
  const particle = particlePool.get();
  particle.x = x;
  particle.y = y;
  particle.vx = Math.random() * 2 - 1;
  particle.vy = Math.random() * 2 - 1;
  particle.age = 0;
  particle.active = true;
  return particle;
}

function updateParticles(particles) {
  for (let i = particles.length - 1; i >= 0; i--) {
    const p = particles[i];
    p.age++;

    if (p.age > 100) {
      // Return to pool instead of creating new objects
      particlePool.release(p);
      particles.splice(i, 1);
    } else {
      p.x += p.vx;
      p.y += p.vy;
    }
  }
}

```

Performance Measurement and Profiling:

```

// Basic performance measurement
function measurePerformance(fn, iterations = 1000) {
  const start = performance.now();

  for (let i = 0; i < iterations; i++) {
    fn();
  }

  const end = performance.now();
  const time = end - start;

  console.log(`Executed ${iterations} iterations in ${time.toFixed(2)}ms`);
  console.log(`Average: ${((time / iterations).toFixed(4))}ms per iteration`);

  return time;
}

// Comparing function performance
function compareFunctions(
  fnA,
  fnB,
  name1 = 'A',
  name2 = 'B',
)

```

```
iterations = 1000
) {
  console.log(
    `Comparing ${name1} vs ${name2} (${iterations} iterations each):`
  );

  const timeA = measurePerformance(fnA, iterations);
  const timeB = measurePerformance(fnB, iterations);

  const faster = timeA < timeB ? name1 : name2;
  const factor = Math.max(timeA, timeB) / Math.min(timeA, timeB);

  console.log(`${faster} is ${factor.toFixed(2)}x faster`);
}

// Using console.time for simple timing
function complexOperation() {
  console.time('complexOperation');

  // Simulate expensive operation
  let sum = 0;
  for (let i = 0; i < 1000000; i++) {
    sum += i;
  }

  console.timeEnd('complexOperation');
  return sum;
}

// Using Performance API for more detailed measurements
function detailedPerfMeasurement() {
  // Mark start point
  performance.mark('operationStart');

  // Do something
  let sum = 0;
  for (let i = 0; i < 1000000; i++) {
    sum += i;
  }

  // Mark end point
  performance.mark('operationEnd');

  // Measure between marks
  performance.measure('operation', 'operationStart', 'operationEnd');

  // Get the measurement
  const measurements = performance.getEntriesByName('operation');
  console.log('Operation took', measurements[0].duration, 'ms');

  // Clean up
  performance.clearMarks();
  performance.clearMeasures();
}
```

```
// Measuring rendering performance
function checkPaintTiming() {
    // Get paint timing entries
    const paintTimings = performance.getEntriesByType('paint');

    paintTimings.forEach((timing) => {
        console.log(`#${timing.name}: ${timing.startTime.toFixed(2)}ms`);
    });
}

// Request animation frame timing
function measureAnimationFrame() {
    const timings = [];
    let frameCount = 0;
    let lastTime = performance.now();

    function frame(now) {
        // Calculate time since last frame
        const delta = now - lastTime;
        lastTime = now;

        timings.push(delta);
        frameCount++;

        if (frameCount < 60) {
            requestAnimationFrame(frame);
        } else {
            // Calculate statistics
            const average = timings.reduce((sum, t) => sum + t, 0) / timings.length;
            const max = Math.max(...timings);
            const min = Math.min(...timings);

            console.log('Animation Frame Timing:');
            console.log(`Average: ${average.toFixed(2)}ms (${(1000 / average).toFixed(2)} FPS)`);
            console.log(`Range: ${min.toFixed(2)}ms - ${max.toFixed(2)}ms`);
        }
    }

    requestAnimationFrame(frame);
}

// Memory usage monitoring (Chrome-specific)
function logMemoryUsage() {
    if (window.performance && window.performance.memory) {
        const memory = window.performance.memory;
        console.log(
            'Total JS heap size:',
            (memory.totalJSHeapSize / 1048576).toFixed(2),
            'MB'
        );
        console.log(

```

```
'Used JS heap size:',  
  (memory.usedJSHeapSize / 1048576).toFixed(2),  
  'MB'  
);  
console.log(  
  'JS heap size limit:',  
  (memory.jsHeapSizeLimit / 1048576).toFixed(2),  
  'MB'  
);  
} else {  
  console.log('Memory API not available');  
}  
}  
  
// Detect long tasks with PerformanceObserver  
function detectLongTasks() {  
  // Check if Long Tasks API is supported  
  if ('PerformanceObserver' in window) {  
    try {  
      const observer = new PerformanceObserver((list) => {  
        for (const entry of list.getEntries()) {  
          console.log('Long task detected:', entry.duration.toFixed(2), 'ms');  
        }  
      });  
  
      observer.observe({ entryTypes: ['longtask'] });  
      return observer;  
    } catch (e) {  
      console.log('Long Tasks API not supported');  
    }  
  }  
  return null;  
}  
  
// Creating a performance monitoring utility  
class PerformanceMonitor {  
  constructor() {  
    this.measurements = {};  
    this.observers = {};  
    this.enabled = true;  
  }  
  
  start(label) {  
    if (!this.enabled) return;  
  
    if (!this.measurements[label]) {  
      this.measurements[label] = [];  
    }  
  
    this.measurements[label].push({  
      start: performance.now(),  
      end: null,  
    });  
  }  
}
```

```
end(label) {
  if (!this.enabled || !this.measurements[label]) return;

  const measurement =
    this.measurements[label][this.measurements[label].length - 1];
  if (measurement && measurement.end === null) {
    measurement.end = performance.now();
    measurement.duration = measurement.end - measurement.start;
  }
}

getStats(label) {
  if (!this.measurements[label]) return null;

  const durations = this.measurements[label]
    .filter((m) => m.end !== null)
    .map((m) => m.duration);

  if (durations.length === 0) return null;

  const sum = durations.reduce((acc, d) => acc + d, 0);

  return {
    count: durations.length,
    total: sum,
    average: sum / durations.length,
    min: Math.min(...durations),
    max: Math.max(...durations),
  };
}

logStats(label = null) {
  if (label) {
    const stats = this.getStats(label);
    if (stats) {
      console.log(`Performance stats for '${label}':`);
      console.log(
        `Count: ${stats.count}, Total: ${stats.total.toFixed(2)}ms`;
      );
      console.log(
        `Avg: ${stats.average.toFixed(2)}ms, Min: ${stats.min.toFixed(
          2
        )}ms, Max: ${stats.max.toFixed(2)}ms`;
      );
    }
  } else {
    for (const key in this.measurements) {
      this.logStats(key);
    }
  }
}

reset(label = null) {
```

```
if (label) {
  delete this.measurements[label];
} else {
  this.measurements = {};
}

// Monitor long tasks
monitorLongTasks() {
  if ('PerformanceObserver' in window) {
    try {
      this.observers.longTasks = new PerformanceObserver((list) => {
        for (const entry of list.getEntries()) {
          console.warn(`Long task detected: ${entry.duration.toFixed(2)}ms`);
        }
      });
    }

    this.observers.longTasks.observe({ entryTypes: ['longtask'] });
  } catch (e) {
    console.log('Long Tasks API not supported');
  }
}

// Monitor frames per second
monitorFPS(duration = 5000) {
  let frames = 0;
  let startTime = performance.now();

  const countFrame = () => {
    frames++;

    const elapsed = performance.now() - startTime;

    if (elapsed < duration) {
      requestAnimationFrame(countFrame);
    } else {
      const fps = (frames / elapsed) * 1000;
      console.log(`Average FPS: ${fps.toFixed(1)}`);
    }
  };

  requestAnimationFrame(countFrame);
}

// Usage
const perfMonitor = new PerformanceMonitor();

function testPerformanceMonitor() {
  perfMonitor.start('operation');

  // Simulate work
  let sum = 0;
```

```
for (let i = 0; i < 1000000; i++) {
    sum += i;
}

perfMonitor.end('operation');

// Repeat a few times
perfMonitor.start('operation');
sum = 0;
for (let i = 0; i < 500000; i++) {
    sum += i;
}
perfMonitor.end('operation');

// Log results
perfMonitor.logStats('operation');
}
```

Testing JavaScript Applications

Conceptual Foundations

Testing is the process of verifying that your code behaves as expected. A comprehensive testing strategy helps catch bugs early, ensures code quality, and provides confidence when making changes to your codebase.

Mental Model: Think of testing like quality control in manufacturing. Just as a car manufacturer tests individual components and the assembled vehicle before shipping, developers test individual units of code and their integration to ensure the application works correctly before delivering to users.

Historical Context: Testing practices in JavaScript have evolved significantly. Early JavaScript had minimal testing. As applications grew more complex, testing frameworks emerged. Modern JavaScript applications often use a combination of unit, integration, and end-to-end tests, with continuous integration pipelines running tests automatically.

Common Misconceptions:

- Testing slows down development (in the long run, it speeds up development and reduces bugs)
- 100% code coverage means bug-free code (coverage doesn't guarantee quality or correctness)
- Only unit tests matter (a balanced approach with different test types is best)
- Manual testing is sufficient (automated tests are more reliable and repeatable)

Practical Implementation

Testing Types and Approaches:

```
// The main types of tests:

// 1. Unit Tests
// Test individual functions, classes, or components in isolation
// Focus on a single unit of functionality
```

```
// Dependencies are typically mocked or stubbed

// 2. Integration Tests
// Test how different units work together
// Verify interactions between components
// May involve testing API calls, database operations, etc.

// 3. End-to-End (E2E) Tests
// Test the application from a user's perspective
// Simulate user interactions like clicking, typing, etc.
// Test the entire system together

// 4. Other types:
// Performance tests
// Accessibility tests
// Security tests
// Visual regression tests
```

Testing with Jest:

```
// Jest is a popular JavaScript testing framework
// Example math.js module to test
function add(a, b) {
  return a + b;
}

function subtract(a, b) {
  return a - b;
}

function multiply(a, b) {
  return a * b;
}

function divide(a, b) {
  if (b === 0) {
    throw new Error('Cannot divide by zero');
  }
  return a / b;
}

module.exports = { add, subtract, multiply, divide };

// Example test file (math.test.js)
const { add, subtract, multiply, divide } = require('./math');

// Test suite
describe('Math functions', () => {
  // Individual test
  test('adds 1 + 2 to equal 3', () => {
    expect(add(1, 2)).toBe(3);
  });
})
```

```
test('subtracts 5 - 2 to equal 3', () => {
  expect(subtract(5, 2)).toBe(3);
});

test('multiplies 3 * 4 to equal 12', () => {
  expect(multiply(3, 4)).toBe(12);
});

test('divides 10 / 2 to equal 5', () => {
  expect(divide(10, 2)).toBe(5);
});

test('throws error when dividing by zero', () => {
  expect(() =>
    divide(10, 0)
  ).toThrow('Cannot divide by zero');
});
});

// Using test.each for parameterized tests
test.each([
  [1, 1, 2],
  [2, 2, 4],
  [3, 3, 6],
])('add(%i, %i) = %i', (a, b, expected) => {
  expect(add(a, b)).toBe(expected);
});

// Grouping related tests
describe('divide function', () => {
  test('divides positive numbers', () => {
    expect(divide(10, 2)).toBe(5);
  });

  test('divides negative numbers', () => {
    expect(divide(-10, 2)).toBe(-5);
  });

  test('divides zero', () => {
    expect(divide(0, 5)).toBe(0);
  });

  test('throws when dividing by zero', () => {
    expect(() => divide(10, 0)).toThrow();
  });
});

// Setup and teardown
let globalData;

beforeAll(() => {
  // Run once before all tests
  globalData = { key: 'value' };
});
```

```
console.log('Setting up test environment');
});

afterAll(() => {
  // Run once after all tests
  globalData = null;
  console.log('Cleaning up test environment');
});

beforeEach(() => {
  // Run before each test
  jest.resetAllMocks(); // Reset all mocks
});

afterEach(() => {
  // Run after each test
  // Clean up resources
});

// Async tests
test('async function returns correct data', async () => {
  const data = await fetchData();
  expect(data).toEqual({ success: true });
});

// Alternative async syntax with promises
test('async function returns correct data', () => {
  return fetchData().then((data) => {
    expect(data).toEqual({ success: true });
  });
});

// Testing for rejections
test('fetchData rejects when network fails', async () => {
  await expect(fetchData('bad-url')).rejects.toThrow();
});
```

Jest Matchers and Assertions:

```
// Common matchers in Jest

// Exact equality
test('two plus two is four', () => {
  expect(2 + 2).toBe(4);
});

// Object equality (deep equality)
test('object equality', () => {
  const data = { one: 1, two: 2 };
  expect(data).toEqual({ one: 1, two: 2 });
});
```

```
// Truthiness
test('truthiness checks', () => {
  const value = 'test';
  expect(value).toBeTruthy();
  expect('').toBeFalsy();
  expect(null).toBeNull();
  expect(undefined).toBeUndefined();
  expect(value).toBeDefined();
});

// Numbers
test('numeric comparisons', () => {
  const value = 2 + 2;
  expect(value).toBeGreaterThan(3);
  expect(value).toBeGreaterThanOrEqual(4);
  expect(value).toBeLessThan(5);
  expect(value).toBeLessThanOrEqual(4);

  // For floating point equality
  expect(0.1 + 0.2).toBeCloseTo(0.3);
});

// Strings
test('string matching', () => {
  expect('team').not.toMatch(/I/);
  expect('Christoph').toMatch(/stop/);

  // Case insensitive
  expect('HELLO WORLD').toMatch(/hello/i);
});

// Arrays and iterables
test('array contains item', () => {
  const shoppingList = ['milk', 'bread', 'eggs'];
  expect(shoppingList).toContain('milk');
  expect(new Set(shoppingList)).toContain('eggs');
});

// Exceptions
test('throws on invalid input', () => {
  expect(() => {
    validateInput(null);
  }).toThrow();

  // Test for specific error message
  expect(() => {
    validateInput(null);
  }).toThrow('Input cannot be null');

  // Test with regex
  expect(() => {
    validateInput(null);
  }).toThrow(/cannot be null/);
});
```

```
// Snapshots
test('renders correctly', () => {
  const tree = renderer.create(<Component />).toJSON();
  expect(tree).toMatchSnapshot();
});

// Partial object matching
test('user has correct properties', () => {
  const user = {
    name: 'John',
    age: 30,
    address: {
      city: 'New York',
      zip: '10001',
    },
  };

  expect(user).toMatchObject({
    name: 'John',
    address: { city: 'New York' },
  });
});

// Array containing objects
test('users array contains user with specific name', () => {
  const users = [
    { name: 'John', age: 30 },
    { name: 'Jane', age: 25 },
  ];

  expect(users).toContainEqual({ name: 'Jane', age: 25 });
});

// Custom matchers
expect.extend({
  toBeWithinRange(received, floor, ceiling) {
    const pass = received >= floor && received <= ceiling;
    if (pass) {
      return {
        message: () =>
          `expected ${received} not to be within range ${floor} - ${ceiling}`,
        pass: true,
      };
    } else {
      return {
        message: () =>
          `expected ${received} to be within range ${floor} - ${ceiling}`,
        pass: false,
      };
    }
  },
});
```

```
test('number is within range', () => {
  expect(25).toBeWithinRange(20, 30);
});
```

Mocking and Spies:

```
// Mocking functions
test('mock function called correctly', () => {
  const mockFn = jest.fn();

  doSomething(mockFn); // Function that should call the callback

  expect(mockFn).toHaveBeenCalled();
  expect(mockFn).toHaveBeenCalledTimes(1);
  expect(mockFn).toHaveBeenCalledWith('arg1', 'arg2');
});

// Mock implementation
test('mock implementation', () => {
  const mockFn = jest.fn(() => 'mocked return value');

  const result = mockFn();

  expect(result).toBe('mocked return value');
});

// Mock implementation once
test('mock implementation once', () => {
  const mockFn = jest
    .fn()
    .mockImplementationOnce(() => 'first call')
    .mockImplementationOnce(() => 'second call')
    .mockImplementation(() => 'default');

  expect(mockFn()).toBe('first call');
  expect(mockFn()).toBe('second call');
  expect(mockFn()).toBe('default');
  expect(mockFn()).toBe('default');
});

// Mocking return values
test('mock return values', () => {
  const mockFn = jest.fn();

  mockFn.mockReturnValueOnce(10).mockReturnValueOnce('x').mockReturnValue(true);

  expect(mockFn()).toBe(10);
  expect(mockFn()).toBe('x');
  expect(mockFn()).toBe(true);
  expect(mockFn()).toBe(true);
});
```

```
// Mocking modules
jest.mock('./userService'); // Mock the entire module

// Import the mocked module
const userService = require('./userService');

test('fetchUsers calls the correct API', () => {
    // Setup the mock implementation
    userService.getUsers.mockResolvedValue([
        { id: 1, name: 'John' },
        { id: 2, name: 'Jane' },
    ]);

    // Use the function that calls userService.getUsers()
    return getUserList().then((users) => {
        expect(users).toHaveLength(2);
        expect(users[0].name).toBe('John');
        expect(userService.getUsers).toHaveBeenCalledTimes(1);
    });
});

// Partial mocks
jest.mock('./utils', () => {
    const originalModule = jest.requireActual('./utils');

    return {
        ...originalModule,
        getRandomNumber: jest.fn(() => 42),
    };
});

// Spying on methods
test('spy on console.log', () => {
    const spy = jest.spyOn(console, 'log');

    logMessage('Hello');

    expect(spy).toHaveBeenCalledWith('Hello');

    // Restore the original implementation
    spy.mockRestore();
});

// Spying on object methods
test('spy on object method', () => {
    const user = {
        getName() {
            return 'John';
        },
    };

    // Spy on getName method
    const spy = jest.spyOn(user, 'getName');
```

```
user.getName();

expect(spy).toHaveBeenCalled();

// Restore original
spy.mockRestore();
});

// Mock timers
test('setTimeout gets called', () => {
  jest.useFakeTimers();

  const callback = jest.fn();

  setTimeout(callback, 1000);

  // Fast-forward until all timers have been executed
  jest.runAllTimers();

  expect(callback).toHaveBeenCalled();

  // Restore real timers
  jest.useRealTimers();
});

// Advancing timers by time
test('advance timers by time', () => {
  jest.useFakeTimers();

  const callback = jest.fn();

  setTimeout(callback, 1000);

  // Advance by 500ms
  jest.advanceTimersByTime(500);
  expect(callback).not.toHaveBeenCalled();

  // Advance by another 500ms
  jest.advanceTimersByTime(500);
  expect(callback).toHaveBeenCalled();

  jest.useRealTimers();
});
```

Testing Asynchronous Code:

```
// Function to test
async function fetchData() {
  const response = await fetch('https://api.example.com/data');
  if (!response.ok) {
    throw new Error('Network error');
  }
}
```

```
    return response.json();
}

// Testing promises with async/await
test('data is fetched correctly', async () => {
    // Mock the fetch function
    global.fetch = jest.fn(() =>
        Promise.resolve({
            ok: true,
            json: () => Promise.resolve({ id: 1, name: 'John' })
        })
    );
}

const data = await fetchData();

expect(data).toEqual({ id: 1, name: 'John' });
expect(fetch).toHaveBeenCalledWith('https://api.example.com/data');
expect(fetch).toHaveBeenCalledTimes(1);
});

// Testing for rejected promises
test('throws error when fetch fails', async () => {
    global.fetch = jest.fn(() =>
        Promise.resolve({
            ok: false,
        })
    );

    await expect(fetchData()).rejects.toThrow('Network error');
});

// Callback-based async code
function fetchDataWithCallback(callback) {
    setTimeout(() => {
        callback({ id: 1, name: 'John' });
    }, 1000);
}

test('callback is called with correct data', (done) => {
    function callback(data) {
        try {
            expect(data).toEqual({ id: 1, name: 'John' });
            done();
        } catch (error) {
            done(error);
        }
    }

    fetchDataWithCallback(callback);

    jest.runAllTimers();
});

// Using resolves/rejects matchers
```

```

test('data promise resolves to correct value', () => {
  global.fetch = jest.fn(() =>
    Promise.resolve({
      ok: true,
      json: () => Promise.resolve({ id: 1, name: 'John' })
    })
  );
  return expect(fetchData()).resolves.toEqual({ id: 1, name: 'John' });
});

// Testing async functions that return values
test('processData returns processed value', async () => {
  const mockData = [1, 2, 3];

  // Mock fetchData to return our test data
  jest.spyOn(api, 'fetchData').mockResolvedValue(mockData);
  // missing code here
});

```

// testing-dom.test.js

```

// Function to test function updateHeader(text) { const header = document.getElementById('header'); if (header) { header.textContent = text; } }

function toggleVisibility(id) { const element = document.getElementById(id); if (element) { if (element.style.display === 'none') { element.style.display = 'block'; } else { element.style.display = 'none'; } } }

// Test setup with Jest beforeEach() => { // Set up a DOM environment for testing document.body.innerHTML = <h1 id="header">Original Header</h1> <div id="content" style="display: block">Content here</div>; };

// Tests describe('DOM manipulation functions', () => { test('updateHeader changes header text', () => { // Call the function to test updateHeader('New Header Text');

```

```

    // Check the result
    const header = document.getElementById('header');
    expect(header.textContent).toBe('New Header Text');
  });

  test('toggleVisibility changes display style', () => {
    const content = document.getElementById('content');
    expect(content.style.display).toBe('block');

    // Toggle it once
    toggleVisibility('content');
  });

```

```
expect(content.style.display).toBe('none');

// Toggle it again
toggleVisibility('content');
expect(content.style.display).toBe('block');
});

test('toggleVisibility does nothing for non-existent element', () => {
    // This should not throw an error
    toggleVisibility('non-existent');
    // Just verify the code ran without error
    expect(true).toBe(true);
});
```

});

```
// Testing event listeners function setupButtonListener() { const button =
document.getElementById('myButton'); const output = document.getElementById('output');
```

```
button.addEventListener('click', () => {
    output.textContent = 'Button clicked!';
});
```

}

```
test('button click handler updates output text', () => { // Set up DOM document.body.innerHTML = <button
id="myButton">Click me</button> <div id="output"></div>;
```

```
// Initialize function to test
setupButtonListener();

// Verify initial state
const output = document.getElementById('output');
expect(output.textContent).toBe('');

// Simulate a click event
const button = document.getElementById('myButton');
button.click();

// Verify the handler was called
expect(output.textContent).toBe('Button clicked!');
```

});

Testing API Calls:

```
```javascript
// api.js (the module we want to test)
export async function fetchUsers() {
 const response = await fetch('https://api.example.com/users');
 if (!response.ok) {
 throw new Error('Failed to fetch users');
 }
 return response.json();
}

export async function createUser(userData) {
 const response = await fetch('https://api.example.com/users', {
 method: 'POST',
 headers: {
 'Content-Type': 'application/json'
 },
 body: JSON.stringify(userData)
 });

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

 return response.json();
}

// api.test.js
import { fetchUsers, createUser } from './api';

// Mock the global fetch function
global.fetch = jest.fn();

describe('API functions', () => {
 // Reset mocks between tests
 beforeEach(() => {
 fetch.mockClear();
 });

 test('fetchUsers returns user data', async () => {
 // Mock the fetch response
 fetch.mockResolvedValueOnce({
 ok: true,
 json: async () => [
 { id: 1, name: 'John' },
 { id: 2, name: 'Jane' }
]
 });
 // Call the function
 const users = await fetchUsers();

 // Verify fetch was called correctly
 expect(fetch).toHaveBeenCalledWith('https://api.example.com/users');
 });
}
```

```
expect(fetch).toHaveBeenCalledTimes(1);

 // Verify the result
 expect(users).toHaveLength(2);
 expect(users[0].name).toBe('John');
 expect(users[1].name).toBe('Jane');
});

test('fetchUsers throws error when request fails', async () => {
 // Mock a failed fetch
 fetch.mockResolvedValueOnce({
 ok: false,
 status: 404,
 statusText: 'Not Found'
 });

 // Verify that the function throws
 await expect(fetchUsers()).rejects.toThrow('Failed to fetch users');
});

test('createUser sends correct request and returns response', async () => {
 // Mock the successful response
 fetch.mockResolvedValueOnce({
 ok: true,
 json: async () => ({ id: 3, name: 'Bob' })
 });

 const userData = { name: 'Bob' };
 const newUser = await createUser(userData);

 // Verify fetch was called with correct arguments
 expect(fetch).toHaveBeenCalledWith(
 'https://api.example.com/users',
 {
 method: 'POST',
 headers: {
 'Content-Type': 'application/json'
 },
 body: JSON.stringify(userData)
 }
);

 // Verify the result
 expect(newUser).toEqual({ id: 3, name: 'Bob' });
});

// Alternative approach with jest-fetch-mock library
// First install: npm install --save-dev jest-fetch-mock
// Then in setupTests.js:
// import fetchMock from 'jest-fetch-mock';
// fetchMock.enableMocks();

// Then in your test:
```

```

import fetchMock from 'jest-fetch-mock';

describe('API with jest-fetch-mock', () => {
 beforeEach(() => {
 fetchMock.resetMocks();
 });

 test('fetchUsers uses fetch and returns users', async () => {
 fetchMock.mockResponseOnce(JSON.stringify([
 { id: 1, name: 'John' },
 { id: 2, name: 'Jane' }
]));

 const users = await fetchUsers();

 expect(fetchMock).toHaveBeenCalledWith('https://api.example.com/users');
 expect(users).toHaveLength(2);
 });

 test('createUser posts data and returns response', async () => {
 fetchMock.mockResponseOnce(JSON.stringify({ id: 3, name: 'Bob' }));

 const userData = { name: 'Bob' };
 const newUser = await createUser(userData);

 expect(fetchMock).toHaveBeenCalledWith(
 'https://api.example.com/users',
 expect.objectContaining({
 method: 'POST',
 body: JSON.stringify(userData)
 })
);
 expect(newUser).toEqual({ id: 3, name: 'Bob' });
 });
});

```

## Testing Best Practices:

```

// 1. Follow the AAA pattern (Arrange, Act, Assert)
test('calculateTotal returns correct sum', () => {
 // Arrange - set up test conditions
 const items = [
 { price: 10, quantity: 2 },
 { price: 5, quantity: 4 },
];

 // Act - execute the function being tested
 const total = calculateTotal(items);

 // Assert - verify the result
 expect(total).toBe(30); // (10*2 + 5*4)
});

```

```
// 2. Test both positive and negative cases
describe('validateEmail function', () => {
 // Positive test cases
 test('returns true for valid email addresses', () => {
 expect(validateEmail('user@example.com')).toBe(true);
 expect(validateEmail('user.name+tag@example.co.uk')).toBe(true);
 });

 // Negative test cases
 test('returns false for invalid email addresses', () => {
 expect(validateEmail('not-an-email')).toBe(false);
 expect(validateEmail('missing@domain')).toBe(false);
 expect(validateEmail('@missing-username.com')).toBe(false);
 expect(validateEmail('')).toBe(false);
 expect(validateEmail(null)).toBe(false);
 });
});

// 3. Use descriptive test and suite names
describe('User authentication', () => {
 describe('login function', () => {
 test('successfully logs in with valid credentials', () => {
 // Test implementation
 });

 test('returns error message with invalid password', () => {
 // Test implementation
 });

 test('returns error message when user does not exist', () => {
 // Test implementation
 });
 });
});

// 4. Keep tests independent and isolated
let user;

beforeEach(() => {
 // Reset state before each test
 user = { id: 1, name: 'Alice', isAdmin: false };
});

test('promoteToAdmin sets isAdmin to true', () => {
 promoteToAdmin(user);
 expect(user.isAdmin).toBe(true);
});

test('revokeAdmin sets isAdmin to false', () => {
 // This test doesn't depend on the previous test
 revokeAdmin(user);
 expect(user.isAdmin).toBe(false);
});
```

```
// 5. Test the public API, not implementation details
// Don't test private methods directly
// Instead, test the public methods that use them

// 6. Mock external dependencies
test('sendWelcomeEmail calls email service with correct data', () => {
 // Mock the email service
 const mockEmailService = {
 sendEmail: jest.fn(),
 };

 // Use dependency injection
 const userService = new UserService(mockEmailService);

 // Act
 userService.registerUser({
 email: 'user@example.com',
 name: 'New User',
 });

 // Assert
 expect(mockEmailService.sendEmail).toHaveBeenCalledWith(
 'user@example.com',
 'Welcome to our app!',
 expect.stringContaining('New User')
);
});

// 7. Add meaningful assertions
test('searchUsers filters by name', () => {
 const users = [
 { id: 1, name: 'Alice' },
 { id: 2, name: 'Bob' },
 { id: 3, name: 'Charlie' },
];

 const results = searchUsers(users, 'ali');

 // Multiple meaningful assertions
 expect(results).toHaveLength(1);
 expect(results[0].id).toBe(1);
 expect(results[0].name).toBe('Alice');
 expect(results).not.toContainEqual(expect.objectContaining({ name: 'Bob' }));
});

// 8. Test error handling
test('divide throws error when dividing by zero', () => {
 // Test that the function throws an error
 expect(() => {
 divide(10, 0);
 }).toThrow('Cannot divide by zero');

 // Test normal functionality still works
});
```

```

expect(divide(10, 2)).toBe(5);
});

// 9. Write readable tests
test('calculateDiscount applies correct discount', () => {
 // Given - a shopping cart with items worth $100
 const cart = {
 items: [
 { name: 'Item 1', price: 40 },
 { name: 'Item 2', price: 60 },
],
 };

 // When - a 20% discount is applied
 const discountedTotal = calculateDiscount(cart, 0.2);

 // Then - the total should be $80
 expect(discountedTotal).toBe(80);
});

// 10. Focus on behavior, not implementation
// Bad approach - testing implementation details
test('bad example - testing implementation', () => {
 const spy = jest.spyOn(Math, 'random');

 const randomId = generateRandomId();

 expect(spy).toHaveBeenCalled();
 spy.mockRestore();
});

// Better approach - testing behavior
test('good example - testing behavior', () => {
 const id1 = generateRandomId();
 const id2 = generateRandomId();

 // Test observable behavior
 expect(id1).not.toBe(id2);
 expect(typeof id1).toBe('string');
 expect(id1.length).toBeGreaterThan(0);
});

```

## Test-Driven Development (TDD):

```

// Test-Driven Development follows a cycle:
// 1. Write a failing test
// 2. Write minimal code to make the test pass
// 3. Refactor the code while keeping tests passing

// Example: Building a shopping cart with TDD

// Step 1: Write the first failing test

```

```
// shoppingCart.test.js
describe('ShoppingCart', () => {
 test('starts with zero items', () => {
 const cart = new ShoppingCart();
 expect(cart.getItemCount()).toBe(0);
 });
});

// Step 2: Write minimal implementation to pass this test
// shoppingCart.js
class ShoppingCart {
 constructor() {
 this.items = [];
 }

 getItemCount() {
 return 0; // Hardcoded to pass the test
 }
}

// Step 3: Add another failing test
test('addItem increases item count', () => {
 const cart = new ShoppingCart();
 cart.addItem({ id: 1, name: 'Product', price: 10 });
 expect(cart.getItemCount()).toBe(1);
});

// Step 4: Update implementation to pass both tests
class ShoppingCart {
 constructor() {
 this.items = [];
 }

 getItemCount() {
 return this.items.length;
 }

 addItem(item) {
 this.items.push(item);
 }
}

// Step 5: Add another failing test
test('getTotal returns the sum of item prices', () => {
 const cart = new ShoppingCart();
 cart.addItem({ id: 1, name: 'Product 1', price: 10 });
 cart.addItem({ id: 2, name: 'Product 2', price: 20 });
 expect(cart.getTotal()).toBe(30);
});

// Step 6: Update implementation
class ShoppingCart {
 constructor() {
 this.items = [];
 }
```

```
}

getItemCount() {
 return this.items.length;
}

addItem(item) {
 this.items.push(item);
}

getTotal() {
 return this.items.reduce((total, item) => total + item.price, 0);
}

// Step 7: Add test for removing items
test('removeItem removes an item from the cart', () => {
 const cart = new ShoppingCart();
 const item = { id: 1, name: 'Product', price: 10 };
 cart.addItem(item);
 expect(cart.getItemCount()).toBe(1);

 cart.removeItem(1); // Remove by ID
 expect(cart.getItemCount()).toBe(0);
});

// Step 8: Update implementation
class ShoppingCart {
 constructor() {
 this.items = [];
 }

 getItemCount() {
 return this.items.length;
 }

 addItem(item) {
 this.items.push(item);
 }

 removeItem(id) {
 this.items = this.items.filter((item) => item.id !== id);
 }

 getTotal() {
 return this.items.reduce((total, item) => total + item.price, 0);
 }
}

// Step 9: Add test for quantity
test('addItem with quantity adds multiple items', () => {
 const cart = new ShoppingCart();
 cart.addItem({ id: 1, name: 'Product', price: 10, quantity: 3 });
 expect(cart.getItemCount()).toBe(1);
```

```
expect(cart.getTotalQuantity()).toBe(3);
expect(cart.getTotal()).toBe(30);
});

// Step 10: Final implementation with all features
class ShoppingCart {
 constructor() {
 this.items = [];
 }

 getItemCount() {
 return this.items.length;
 }

 getTotalQuantity() {
 return this.items.reduce((total, item) => total + (item.quantity || 1), 0);
 }

 addItem(item) {
 // Check if item already exists
 const existingItem = this.items.find((i) => i.id === item.id);
 if (existingItem) {
 existingItem.quantity =
 (existingItem.quantity || 1) + (item.quantity || 1);
 } else {
 this.items.push({
 ...item,
 quantity: item.quantity || 1,
 });
 }
 }

 removeItem(id) {
 this.items = this.items.filter((item) => item.id !== id);
 }

 updateQuantity(id, quantity) {
 const item = this.items.find((i) => i.id === id);
 if (item) {
 item.quantity = quantity;
 if (item.quantity <= 0) {
 this.removeItem(id);
 }
 }
 }

 getTotal() {
 return this.items.reduce((total, item) => {
 return total + item.price * (item.quantity || 1);
 }, 0);
 }

 clearCart() {
 this.items = [];
 }
}
```

```
}
```

## 🏆 PART 3: REACT FUNDAMENTALS

### React Fundamentals and JSX

#### Conceptual Foundations

React is a JavaScript library for building user interfaces. It lets you compose complex UIs from small, isolated pieces of code called "components." React uses a declarative approach, where you describe how your UI should look at any given point in time, and React efficiently updates and renders the right components when your data changes.

**Mental Model:** Think of React as a blueprint system. You design reusable blueprints (components), specify what they should look like based on different inputs (props), and React builds the actual UI. When the inputs change, React automatically updates the structure according to your blueprint, without you having to manipulate the DOM directly.

**Historical Context:** React was created by Jordan Walke at Facebook and released in 2013. It addressed the challenges of building complex, interactive UIs with frequently changing data. Before React, developers had to manually manipulate the DOM, which was error-prone and inefficient. React introduced a new paradigm with the Virtual DOM for efficient updates and a component-based architecture.

#### Common Misconceptions:

- React is a framework (it's actually a library - it handles the view layer)
- React is only for web development (React can be used for mobile with React Native and other platforms)
- React is slow because of the Virtual DOM (the Virtual DOM is actually what makes React fast for most use cases)
- You need to rewrite your entire application to use React (React can be incrementally adopted)

### Setting Up a React Environment

#### Create React App (CRA):

```
Install Create React App globally (you only need to do this once)
npm install -g create-react-app

Create a new React project
npx create-react-app my-react-app

Navigate to the project directory
cd my-react-app
```

```
Start the development server
npm start
```

## Project Structure:

```
my-react-app/
 └── node_modules/ # Dependencies installed by npm
 └── public/ # Static files
 └── index.html # HTML template
 └── favicon.ico # Favicon
 └── ... # Other static assets
 └── src/ # Source code
 └── App.js # Main component
 └── App.css # Styles for App component
 └── index.js # Entry point
 └── index.css # Global styles
 └── ... # Other components and files
 └── package.json # Project configuration and dependencies
 └── README.md # Project documentation
```

## Basic React Component:

```
// App.js
import React from 'react';
import './App.css';

function App() {
 return (
 <div className="App">
 <header className="App-header">
 <h1>Hello, React!</h1>
 <p>Welcome to my first React application.</p>
 </header>
 </div>
);
}

export default App;
```

## JSX (JavaScript XML)

JSX is a syntax extension for JavaScript that lets you write HTML-like markup inside a JavaScript file. Although it looks like HTML, it's actually closer to JavaScript.

**Mental Model:** Think of JSX as a template language that has the full power of JavaScript. It's like HTML that can directly incorporate JavaScript expressions and logic.

```
// Basic JSX syntax
const element = <h1>Hello, world!</h1>

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

// JSX with attributes (use camelCase for HTML attributes)
const link = (

 React Documentation

);

// JSX with children
const container = (
 <div>
 <h1>Title</h1>
 <p>Paragraph text</p>
 </div>
);

// JSX must have one root element (or use fragments)
const validJSX = (
 <div>
 <h1>Title</h1>
 <p>Paragraph</p>
 </div>
);

// Using React Fragment to avoid extra divs
const withFragment = (
 <>
 <h1>Title</h1>
 <p>Paragraph</p>
 </>
);

// JSX with inline styles (uses objects, not strings)
const styledElement = (
 <div
 style={{
 color: 'blue',
 fontSize: '16px',
 marginTop: '20px',
 }}
 >
 Styled text
 </div>
);

// JSX with event handlers
const handleClick = () => alert('Button clicked!');
```

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

// JSX with conditional rendering
const isLoggedIn = true;
const greeting2 = (
 <div>{isLoggedIn ? <h1>Welcome back!</h1> : <h1>Please sign in.</h1>}</div>
);

// JSX with list rendering (each child should have a unique "key" prop)
const numbers = [1, 2, 3, 4, 5];
const listItems = (

 {numbers.map((number) => (
 <li key={number.toString()}>{number}
)))

);

// Self-closing tags must be closed
const img = ;
```

## Under the Hood: JSX Compilation

JSX code gets compiled to regular JavaScript by tools like Babel. For example:

```
// JSX code
const element = <h1 className="greeting">Hello, world!</h1>

// Compiles to this JavaScript code
const element = React.createElement(
 'h1',
 { className: 'greeting' },
 'Hello, world!'
);
```

## React Components

Components are the building blocks of React applications. They are reusable, self-contained pieces of code that encapsulate markup, styles, and behavior.

### Functional Components:

```
// Simple functional component
function Welcome(props) {
 return <h1>Hello, {props.name}</h1>;
}

// Arrow function component
const Welcome = (props) => {
```

```
return <h1>Hello, {props.name}</h1>;
};

// Simplified return with implicit return
const Welcome = (props) => <h1>Hello, {props.name}</h1>

// Using destructuring for props
const Welcome = ({ name, age }) => (
 <div>
 <h1>Hello, {name}</h1>
 <p>You are {age} years old.</p>
 </div>
);

// Component with default props
const Welcome = ({ name = 'Guest' }) => <h1>Hello, {name}</h1>
```

## Class Components:

```
// Basic class component
import React, { Component } from 'react';

class Welcome extends Component {
 render() {
 return <h1>Hello, {this.props.name}</h1>;
 }
}

// Class component with constructor
class Counter extends Component {
 constructor(props) {
 super(props); // Always call super with props in constructor
 this.state = { count: 0 };
 }

 render() {
 return <h1>Count: {this.state.count}</h1>;
 }
}

// Class component with methods
class Counter extends Component {
 constructor(props) {
 super(props);
 this.state = { count: 0 };

 // Binding method to access 'this'
 this.increment = this.increment.bind(this);
 }

 increment() {
 this.setState({ count: this.state.count + 1 });
 }
}
```

```
}

// Alternative: use arrow function (auto-binds 'this')
decrement = () => {
 this.setState({ count: this.state.count - 1 });
};

render() {
 return (
 <div>
 <h1>Count: {this.state.count}</h1>
 <button onClick={this.increment}>+</button>
 <button onClick={this.decrement}>-</button>
 </div>
);
}
}
```

## Component Composition:

```
// Creating nested components
function App() {
 return (
 <div>
 <Header />
 <MainContent />
 <Footer />
 </div>
);
}

function Header() {
 return (
 <header>
 <h1>My Website</h1>
 </header>
);
}

function MainContent() {
 return (
 <main>
 <h2>Welcome to my website!</h2>
 <p>This is the main content area.</p>
 </main>
);
}

function Footer() {
 return <footer>© 2025 My Website</footer>;
}
```

```
// Component with children props
function Card({ title, children }) {
 return (
 <div className="card">
 <h2>{title}</h2>
 <div className="card-content">
 {children}' '
 {/* This will render any content placed between <Card> tags */}
 </div>
 </div>
);
}

// Using the Card component with children
function App() {
 return (
 <div>
 <Card title="Welcome">
 <p>This is a card with some content.</p>
 <button>Click me</button>
 </Card>
 </div>
);
}
```

## Props and Data Flow

Props (short for "properties") are a way to pass data from parent to child components. Props are read-only and follow a one-way data flow.

**Mental Model:** Think of props like function arguments. Just as you pass arguments to a function to customize its behavior, you pass props to a component to customize what it renders.

```
// Passing props to a component
function App() {
 return (
 <div>
 <Welcome name="Alice" age={25} isAdmin={true} />
 <Welcome name="Bob" age={30} isAdmin={false} />
 </div>
);
}

// Receiving and using props
function Welcome(props) {
 return (
 <div>
 <h1>Hello, {props.name}!</h1>
 <p>Age: {props.age}</p>
 {props.isAdmin && <p>Admin privileges granted</p>}
 </div>
);
}
```

```
);

}

// Destructuring props
function Welcome({ name, age, isAdmin }) {
 return (
 <div>
 <h1>Hello, {name}!</h1>
 <p>Age: {age}</p>
 {isAdmin && <p>Admin privileges granted</p>}
 </div>
);
}

// Props with default values
function Welcome({ name = 'Guest', age = 'Unknown', isAdmin = false }) {
 return (
 <div>
 <h1>Hello, {name}!</h1>
 <p>Age: {age}</p>
 {isAdmin && <p>Admin privileges granted</p>}
 </div>
);
}

// Using the spread operator with props
function App() {
 const userProps = {
 name: 'Alice',
 age: 25,
 isAdmin: true,
 };

 return <Welcome {...userProps} />;
}

// Passing functions as props
function App() {
 const handleClick = () => {
 alert('Button clicked!');
 };

 return <Button onClick={handleClick} text="Click me" />;
}

function Button({ onClick, text }) {
 return <button onClick={onClick}>{text}</button>;
}

// PropTypes for type checking (requires prop-types package)
import PropTypes from 'prop-types';

function Welcome({ name, age, isAdmin }) {
 return (
```

```
<div>
 <h1>Hello, {name}!</h1>
 <p>Age: {age}</p>
 {isAdmin && <p>Admin privileges granted</p>}
</div>
);
}

Welcome.propTypes = {
 name: PropTypes.string.isRequired,
 age: PropTypes.number,
 isAdmin: PropTypes.bool,
};

Welcome.defaultProps = {
 age: 0,
 isAdmin: false,
};
```

## State and Lifecycle

State is private data controlled by a component. When state changes, React re-renders the component. Lifecycle methods (in class components) let you run code at specific points in a component's life.

**Mental Model:** Think of state as a component's memory. It's like the component's internal notepad where it keeps track of information that can change over time, like user input or fetched data.

### Class Component State:

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

class Counter extends Component {
 constructor(props) {
 super(props);
 // Initialize state
 this.state = {
 count: 0,
 lastClicked: null,
 };
 }

 // Incorrect way to update state (doesn't trigger re-rendering)
 // badIncrement() {
 // this.state.count += 1; // Don't modify state directly
 // }

 // Correct way to update state
 increment = () => {
 this.setState({ count: this.state.count + 1 });
 };
}
```

```
// setState with function (when new state depends on old state)
incrementSafe = () => {
 this.setState((prevState) => ({
 count: prevState.count + 1,
 lastClicked: 'increment',
 }));
};

// setState with callback after update
incrementWithCallback = () => {
 this.setState({ count: this.state.count + 1 }, () =>
 console.log('State updated, new count:', this.state.count)
);
};

render() {
 return (
 <div>
 <h1>Count: {this.state.count}</h1>
 <button onClick={this.increment}>Increment</button>
 <button onClick={this.incrementSafe}>Increment Safely</button>
 <button onClick={this.incrementWithCallback}>
 Increment with Callback
 </button>
 {this.state.lastClicked && (
 <p>Last button clicked: {this.state.lastClicked}</p>
)}
 </div>
);
}
}
```

## Functional Component State (with useState Hook):

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

function Counter() {
 // Initialize state with useState hook
 const [count, setCount] = useState(0);
 const [lastClicked, setLastClicked] = useState(null);

 // Update state with state setter function
 const increment = () => {
 setCount(count + 1);
 setLastClicked('increment');
 };

 // Update state based on previous state
 const incrementSafe = () => {
 setCount((prevCount) => prevCount + 1);
 setLastClicked('increment safe');
 };
}
```

```
return (
 <div>
 <h1>Count: {count}</h1>
 <button onClick={increment}>Increment</button>
 <button onClick={incrementSafe}>Increment Safely</button>
 {lastClicked && <p>Last button clicked: {lastClicked}</p>}
 </div>
);
}
```

## Lifecycle Methods (Class Components):

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

class LifecycleDemo extends Component {
 constructor(props) {
 super(props);
 this.state = {
 count: 0,
 data: null,
 error: null,
 };
 console.log('1. Constructor - Component is being initialized');
 }

 // Mounting phase
 componentDidMount() {
 console.log('3. ComponentDidMount - Component has been inserted into DOM');
 // Good place for initial data fetching
 this.fetchData();

 // Set up timers or intervals
 this.timer = setInterval(() => {
 this.setState((prevState) => ({ count: prevState.count + 1 }));
 }, 1000);
 }

 // Updating phase
 static getDerivedStateFromProps(nextProps, prevState) {
 console.log('2. GetDerivedStateFromProps - Before render');
 // Return new state or null
 return null;
 }

 shouldComponentUpdate(nextProps, nextState) {
 console.log('4. ShouldComponentUpdate - Should re-render?');
 // Return true to update, false to skip update
 return true;
 }

 getSnapshotBeforeUpdate(prevProps, prevState) {
```

```
console.log('6. GetSnapshotBeforeUpdate - Before DOM update');
// Return value to pass to componentDidUpdate
return window.scrollY;
}

componentDidUpdate(prevProps, prevState, snapshot) {
 console.log('7. ComponentDidUpdate - After DOM update', snapshot);

 // Check if specific prop or state changed
 if (prevProps.id !== this.props.id) {
 this.fetchData(); // Re-fetch data if id changed
 }
}

// Unmounting phase
componentWillUnmount() {
 console.log('8. ComponentWillUnmount - Before component removal');
 // Clean up resources
 clearInterval(this.timer);
}

// Error handling
static getDerivedStateFromError(error) {
 console.log('Error occurred in a child component');
 // Update state to display fallback UI
 return { error: error.message };
}

componentDidCatch(error, info) {
 console.log('Error details:', error, info);
 // Log the error to an error reporting service
}

// Custom methods
fetchData = async () => {
 try {
 const response = await fetch('https://api.example.com/data');
 const data = await response.json();
 this.setState({ data });
 } catch (error) {
 this.setState({ error: error.message });
 }
};

render() {
 console.log('5. Render - Building React elements');

 if (this.state.error) {
 return <div>Error: {this.state.error}</div>;
 }

 return (
 <div>
 <h1>Lifecycle Demo</h1>
 </div>
);
}
```

```
<p>Count: {this.state.count}</p>
{this.state.data ? (
 <div>Data: {JSON.stringify(this.state.data)}</div>
) : (
 <div>Loading data...</div>
)}
</div>
);
}
}
```

## Handling Events

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

```
// Basic event handling
function Button() {
 const handleClick = () => {
 alert('Button clicked!');
 };

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

// Event handling with parameters
function Button() {
 const handleClick = (text) => {
 alert(`Button says: ${text}`);
 };

 // Use an arrow function in the callback to pass parameters
 return <button onClick={() => handleClick('Hello!')}>Click me</button>;
}

// This would call handleClick immediately, not on click:
// return <button onClick={handleClick('Hello!')}>Click me</button>;
}

// Accessing the event object
function Form() {
 const handleChange = (event) => {
 console.log('Input value:', event.target.value);
 };

 return <input type="text" onChange={handleChange} />;
}

// Preventing default behavior
function Form() {
 const handleSubmit = (event) => {
 event.preventDefault(); // Prevent form submission
 console.log('Form submitted');
}
```

```
};

return (
 <form onSubmit={handleSubmit}>
 <input type="text" />
 <button type="submit">Submit</button>
 </form>
);
}

// Synthetic events
function Input() {
 const handleKeyDown = (event) => {
 // event is a SyntheticEvent (cross-browser wrapper)
 if (event.key === 'Enter') {
 console.log('Enter key pressed');
 }
 };
 return <input type="text" onKeyDown={handleKeyDown} />;
}

// Event delegation
function List() {
 const handleItemClick = (event) => {
 // Get the data attribute from the clicked item
 const id = event.target.dataset.id;
 console.log('Clicked item with ID:', id);
 };

 return (
 <ul onClick={handleItemClick}>
 <li data-id="1">Item 1
 <li data-id="2">Item 2
 <li data-id="3">Item 3

);
}

// Class component with bound event handlers
class Toggle extends React.Component {
 constructor(props) {
 super(props);
 this.state = { isOn: false };

 // Binding in constructor
 this.handleClick = this.handleClick.bind(this);
 }

 handleClick() {
 this.setState((prevState) => ({
 isOn: !prevState.isOn,
 }));
 }
}
```

```
render() {
 return (
 <button onClick={this.handleClick}>
 {this.state.isOn ? 'ON' : 'OFF'}
 </button>
);
}

// Class component with class field syntax (no binding needed)
class Toggle extends React.Component {
 state = { isOn: false };

 // Arrow function automatically binds 'this'
 handleClick = () => {
 this.setState((prevState) => ({
 isOn: !prevState.isOn,
 }));
 };

 render() {
 return (
 <button onClick={this.handleClick}>
 {this.state.isOn ? 'ON' : 'OFF'}
 </button>
);
 }
}
```

## Conditional Rendering

Conditional rendering allows you to render different UI based on certain conditions.

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

// Ternary operator
function UserGreeting({ isLoggedIn }) {
 return <h1>{isLoggedIn ? 'Welcome back!' : 'Please sign in.'}</h1>;
}

// Logical && operator (for rendering or nothing)
function Notification({ message }) {
 return <div>{message && <div className="alert">{message}</div>}</div>;
```

```
}

// Multiple conditions
function StatusMessage({ status }) {
 let message;

 if (status === 'loading') {
 message = <p>Loading...</p>;
 } else if (status === 'success') {
 message = <p className="success">Data loaded successfully!</p>;
 } else if (status === 'error') {
 message = <p className="error">Error loading data!</p>;
 } else {
 message = <p>Unknown status</p>;
 }

 return <div className="status-container">{message}</div>;
}

// Using switch statement
function StatusMessage({ status }) {
 let message;

 switch (status) {
 case 'loading':
 message = <p>Loading...</p>;
 break;
 case 'success':
 message = <p className="success">Data loaded successfully!</p>;
 break;
 case 'error':
 message = <p className="error">Error loading data!</p>;
 break;
 default:
 message = <p>Unknown status</p>;
 }

 return <div className="status-container">{message}</div>;
}

// Using object literal (map of components)
function StatusMessage({ status }) {
 const statusMessages = {
 loading: <p>Loading...</p>,
 success: <p className="success">Data loaded successfully!</p>,
 error: <p className="error">Error loading data!</p>,
 };

 return (
 <div className="status-container">
 {statusMessages[status] || <p>Unknown status</p>}
 </div>
);
}
```

```
// Conditional rendering with null (nothing renders)
function ConditionalComponent({ shouldRender }) {
 if (!shouldRender) {
 return null;
 }

 return <div>This will only render if shouldRender is true.</div>;
}

// Conditional CSS classes
function Button({ isPrimary }) {
 return (
 <button className={isPrimary ? 'btn btn-primary' : 'btn btn-secondary'}>
 Click me
 </button>
);
}

// Using template literals for multiple conditions
function Button({ isPrimary, isLarge, isDisabled }) {
 const className = `btn
 ${isPrimary ? 'btn-primary' : 'btn-secondary'}
 ${isLarge ? 'btn-large' : ''}
 ${isDisabled ? 'btn-disabled' : ''}
 `;

 return (
 <button className={className} disabled={isDisabled}>
 Click me
 </button>
);
}
```

## Lists and Keys

List rendering in React involves mapping an array to JSX elements. Keys help React identify which items have changed, been added, or been removed.

```
// Basic list rendering
function NumberList({ numbers }) {
 const listItems = numbers.map((number) => (
 <li key={number.toString()}>{number}
));

 return {listItems};
}

// Inline map in JSX
function NumberList({ numbers }) {
 return (
```

```

 {numbers.map((number) => (
 <li key={number.toString()}>{number}
)))

);
}

// List of objects with id as key
function UserList({ users }) {
 return (

 {users.map((user) => (
 <li key={user.id}>
 {user.name} - {user.email}

)))

);
}

// If no stable ID exists, use index as a last resort
function ItemList({ items }) {
 return (

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

);
}

// Keys must be unique among siblings, but not globally
function Blog({ posts }) {
 return (
 <div>
 {/* Each key only needs to be unique within this array */}
 {posts.map((post) => (
 <BlogPost key={post.id} post={post} />
)))
 {/* Same IDs can be used in a different array */}
 <h2>Featured Posts</h2>
 {posts
 .filter((post) => post.isFeatured)
 .map((post) => (
 <BlogPost key={post.id} post={post} />
)))
 </div>
);
}

// Using a component for list items
function TodoList({ todos, onToggle }) {
```

```
return (

 {todos.map((todo) => (
 <TodoItem key={todo.id} todo={todo} onToggle={onToggle} />
)))

);
}

function TodoItem({ todo, onToggle }) {
 return (

 <input
 type="checkbox"
 checked={todo.completed}
 onChange={() => onToggle(todo.id)}
 />
 <span
 style={{ textDecoration: todo.completed ? 'line-through' : 'none' }}
 >
 {todo.text}

);
}

// List with deletion
function TodoList({ todos, onDelete }) {
 return (

 {todos.map((todo) => (
 <li key={todo.id}>
 {todo.text}
 <button onClick={() => onDelete(todo.id)}>Delete</button>

)))

);
}

// Nested lists
function NestedList({ categories }) {
 return (

 {categories.map((category) => (
 <li key={category.id}>
 {category.name}
 {category.items.length > 0 && (

 {category.items.map((item) => (
 <li key={item.id}>{item.name}
)))

)})
))}

```

```

)}

);
}
```

## Forms and Controlled Components

In React, there are two approaches to handling form inputs: controlled components (React manages the form data) and uncontrolled components (DOM manages the form data).

**Mental Model:** Think of controlled components as having React as the "single source of truth" for form data. The form elements don't maintain their own state; instead, React holds their values in state and updates them via handlers.

### Controlled Components:

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

function SimpleForm() {
 // State for form values
 const [name, setName] = useState('');
 const [email, setEmail] = useState('');
 const [message, setMessage] = useState('');

 // Handler for form submission
 const handleSubmit = (event) => {
 event.preventDefault();
 console.log('Form submitted:', { name, email, message });
 // Process the form data (e.g., send to an API)

 // Reset form after submission
 setName('');
 setEmail('');
 setMessage('');
 };

 return (
 <form onSubmit={handleSubmit}>
 <div>
 <label htmlFor="name">Name:</label>
 <input
 type="text"
 id="name"
 value={name}
 onChange={(e) => setName(e.target.value)}
 required
 />
 </div>
 <div>
```

```
 <label htmlFor="email">Email:</label>
 <input
 type="email"
 id="email"
 value={email}
 onChange={(e) => setEmail(e.target.value)}
 required
 />
 </div>

 <div>
 <label htmlFor="message">Message:</label>
 <textarea
 id="message"
 value={message}
 onChange={(e) => setMessage(e.target.value)}
 rows="4"
 />
 </div>

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

## Multiple Input Types:

```
function ComplexForm() {
 // Single state object for all form values
 const [formData, setFormData] = useState({
 username: '',
 email: '',
 password: '',
 confirmPassword: '',
 gender: '',
 age: '',
 interests: [],
 agreeToTerms: false,
 });

 // Error state
 const [errors, setErrors] = useState({});

 // Generic change handler for most inputs
 const handleChange = (event) => {
 const { name, value, type, checked } = event.target;

 // For checkboxes, use the 'checked' property instead of 'value'
 setFormData((prevData) => ({
 ...prevData,
 [name]: type === 'checkbox' ? checked : value,
 }));
 };
}
```

```
};

// Special handler for interests (checkboxes that map to array)
const handleInterestChange = (event) => {
 const { value, checked } = event.target;

 setFormData((prevData) => {
 // If checked, add to array; if unchecked, remove from array
 if (checked) {
 return {
 ...prevData,
 interests: [...prevData.interests, value],
 };
 } else {
 return {
 ...prevData,
 interests: prevData.interests.filter(
 (interest) => interest !== value
),
 };
 }
 });
};

// Validate form and update errors state
const validateForm = () => {
 const newErrors = {};

 // Username validation
 if (!formData.username.trim()) {
 newErrors.username = 'Username is required';
 } else if (formData.username.length < 3) {
 newErrors.username = 'Username must be at least 3 characters';
 }

 // Email validation
 if (!formData.email.trim()) {
 newErrors.email = 'Email is required';
 } else if (!/\S+@\S+\.\S+/.test(formData.email)) {
 newErrors.email = 'Email is invalid';
 }

 // Password validation
 if (!formData.password) {
 newErrors.password = 'Password is required';
 } else if (formData.password.length < 6) {
 newErrors.password = 'Password must be at least 6 characters';
 }

 // Confirm password
 if (formData.password !== formData.confirmPassword) {
 newErrors.confirmPassword = 'Passwords do not match';
 }
}
```

```
// Terms agreement
if (!formData.agreeToTerms) {
 newErrors.agreeToTerms = 'You must agree to the terms';
}

setErrors(newErrors);
return Object.keys(newErrors).length === 0; // Return true if no errors
};

// Form submission handler
const handleSubmit = (event) => {
 event.preventDefault();

 // Validate form
 const isValid = validateForm();

 if (isValid) {
 console.log('Form submitted:', formData);
 // Submit to API, etc.

 // Clear form
 setFormData({
 username: '',
 email: '',
 password: '',
 confirmPassword: '',
 gender: '',
 age: '',
 interests: [],
 agreeToTerms: false,
 });
 setErrors({});
 } else {
 console.log('Form has errors', errors);
 }
};

return (
 <form onSubmit={handleSubmit}>
 {/* Text Input */}
 <div>
 <label htmlFor="username">Username:</label>
 <input
 type="text"
 id="username"
 name="username"
 value={formData.username}
 onChange={handleChange}
 />
 {errors.username && {errors.username}}
 </div>

 {/* Email Input */}
 <div>
```

```
<label htmlFor="email">Email:</label>
<input
 type="email"
 id="email"
 name="email"
 value={formData.email}
 onChange={handleChange}
/>
{errors.email && {errors.email}}
</div>

{/* Password Input */}
<div>
 <label htmlFor="password">Password:</label>
 <input
 type="password"
 id="password"
 name="password"
 value={formData.password}
 onChange={handleChange}
/>
 {errors.password && {errors.password}}
</div>

{/* Confirm Password */}
<div>
 <label htmlFor="confirmPassword">Confirm Password:</label>
 <input
 type="password"
 id="confirmPassword"
 name="confirmPassword"
 value={formData.confirmPassword}
 onChange={handleChange}
/>
 {errors.confirmPassword && (
 {errors.confirmPassword}
)}
</div>

{/* Radio Buttons */}
<div>
 <p>Gender:</p>
 <label>
 <input
 type="radio"
 name="gender"
 value="male"
 checked={formData.gender === 'male'}
 onChange={handleChange}
 />
 Male
 </label>
 <label>
 <input
```

```
 type="radio"
 name="gender"
 value="female"
 checked={formData.gender === 'female'}
 onChange={handleChange}
 />
 Female
</label>
<label>
 <input
 type="radio"
 name="gender"
 value="other"
 checked={formData.gender === 'other'}
 onChange={handleChange}
 />
 Other
</label>
</div>

{/* Select Dropdown */}
<div>
 <label htmlFor="age">Age Group:</label>
 <select
 id="age"
 name="age"
 value={formData.age}
 onChange={handleChange}
 >
 <option value="">Select age group</option>
 <option value="under18">Under 18</option>
 <option value="18-24">18-24</option>
 <option value="25-34">25-34</option>
 <option value="35-44">35-44</option>
 <option value="45+">45+</option>
 </select>
</div>

{/* Checkboxes (multiple) */}
<div>
 <p>Interests:</p>
 <label>
 <input
 type="checkbox"
 name="interests"
 value="sports"
 checked={formData.interests.includes('sports')}
 onChange={handleInterestChange}
 />
 Sports
 </label>
 <label>
 <input
 type="checkbox"
```

```
 name="interests"
 value="music"
 checked={formData.interests.includes('music')}
 onChange={handleInterestChange}
 />
 Music
</label>
<label>
 <input
 type="checkbox"
 name="interests"
 value="reading"
 checked={formData.interests.includes('reading')}
 onChange={handleInterestChange}
 />
 Reading
</label>
<label>
 <input
 type="checkbox"
 name="interests"
 value="travel"
 checked={formData.interests.includes('travel')}
 onChange={handleInterestChange}
 />
 Travel
</label>
</div>

/* Single Checkbox */
<div>
 <label>
 <input
 type="checkbox"
 name="agreeToTerms"
 checked={formData.agreeToTerms}
 onChange={handleChange}
 />
 I agree to the terms and conditions
 </label>
 {errors.agreeToTerms && (
 {errors.agreeToTerms}
)}
</div>

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

## Uncontrolled Components:

```
import React, { useRef } from 'react';

function UncontrolledForm() {
 // Create refs for form elements
 const nameInputRef = useRef(null);
 const emailInputRef = useRef(null);
 const messageInputRef = useRef(null);

 // Form submission handler
 const handleSubmit = (event) => {
 event.preventDefault();

 // Access values from refs
 const formData = {
 name: nameInputRef.current.value,
 email: emailInputRef.current.value,
 message: messageInputRef.current.value,
 };

 console.log('Form submitted:', formData);

 // Reset the form
 event.target.reset();
 };

 return (
 <form onSubmit={handleSubmit}>
 <div>
 <label htmlFor="name">Name:</label>
 <input
 type="text"
 id="name"
 ref={nameInputRef}
 defaultValue=""
 required
 />
 </div>

 <div>
 <label htmlFor="email">Email:</label>
 <input
 type="email"
 id="email"
 ref={emailInputRef}
 defaultValue=""
 required
 />
 </div>

 <div>
 <label htmlFor="message">Message:</label>
 <textarea id="message" ref={messageInputRef} defaultValue="" rows="4" />
 </div>
 </form>
);
}
```

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

## Component Architecture

### Conceptual Foundations

Component architecture is about how you structure and organize your React components to create maintainable, reusable, and efficient applications. Good component architecture follows principles like separation of concerns, single responsibility, and composition over inheritance.

**Mental Model:** Think of components as LEGO blocks. Each block has a specific purpose and can be combined with others to create complex structures. Well-designed components should be modular, reusable, and focused on doing one thing well.

### Common Patterns:

1. **Container/Presentational Pattern** - Separate data fetching and state management (container) from rendering UI (presentational).
2. **Composition Pattern** - Build complex UIs by composing simpler components together.
3. **Render Props Pattern** - Components share code by taking a function prop that returns a React element.
4. **Higher-Order Components (HOCs)** - Functions that take a component and return a new enhanced component.
5. **Compound Components** - Multiple components that work together to form a cohesive unit.

### Functional vs. Class Components

#### Class Components:

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

class UserProfile extends Component {
 constructor(props) {
 super(props);
 this.state = {
 isEditing: false,
 userData: {
 name: props.name || '',
 bio: props.bio || '',
 },
 };
}
```

```
// Lifecycle methods
componentDidMount() {
 console.log('Component mounted');
 // Fetch data, set up subscriptions, etc.
}

componentDidUpdate(prevProps, prevState) {
 // React to props or state changes
 if (prevProps.name !== this.props.name) {
 this.setState({
 userData: {
 ...this.state.userData,
 name: this.props.name,
 },
 });
 }
}

componentWillUnmount() {
 console.log('Component unmounting');
 // Clean up: remove listeners, cancel requests, etc.
}

// Class methods
toggleEdit = () => {
 this.setState((prevState) => ({
 isEditing: !prevState.isEditing,
 }));
};

handleChange = (event) => {
 const { name, value } = event.target;
 this.setState((prevState) => ({
 userData: {
 ...prevState.userData,
 [name]: value,
 },
 }));
};

handleSubmit = (event) => {
 event.preventDefault();
 // Save data
 this.props.onUpdate(this.state.userData);
 this.toggleEdit();
};

render() {
 const { isEditing, userData } = this.state;

 if (isEditing) {
 return (
 <form onSubmit={this.handleSubmit}>
 <div>

```

```

 <label htmlFor="name">Name:</label>
 <input
 type="text"
 id="name"
 name="name"
 value={userData.name}
 onChange={this.handleChange}
 />
 </div>
 <div>
 <label htmlFor="bio">Bio:</label>
 <textarea
 id="bio"
 name="bio"
 value={userData.bio}
 onChange={this.handleChange}
 />
 </div>
 <button type="submit">Save</button>
 <button type="button" onClick={this.toggleEdit}>
 Cancel
 </button>
</form>
);
}

return (
 <div className="user-profile">
 <h2>{userData.name}</h2>
 <p>{userData.bio}</p>
 <button onClick={this.toggleEdit}>Edit Profile</button>
 </div>
);
}
}

```

## Functional Components with Hooks:

```

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

function UserProfile(props) {
 // State hook
 const [isEditing, setIsEditing] = useState(false);
 const [userData, setUserData] = useState({
 name: props.name || '',
 bio: props.bio || '',
 });

 // Effect hook (componentDidMount and componentDidUpdate combined)
 useEffect(() => {
 console.log('Component mounted or updated');
 });
}

```

```
// This runs when component unmounts
return () => {
 console.log('Component unmounting');
 // Clean up: remove listeners, cancel requests, etc.
};

}, []); // Empty dependency array means it only runs on mount and unmount

// Effect with dependencies (runs when props.name changes)
useEffect(() => {
 setUserData((prevData) => ({
 ...prevData,
 name: props.name,
 }));
}, [props.name]);

// Event handlers
const toggleEdit = () => {
 setIsEditing((prevState) => !prevState);
};

const handleChange = (event) => {
 const { name, value } = event.target;
 setUserData((prevData) => ({
 ...prevData,
 [name]: value,
 }));
};

const handleSubmit = (event) => {
 event.preventDefault();
 props.onUpdate(userData);
 toggleEdit();
};

if (isEditing) {
 return (
 <form onSubmit={handleSubmit}>
 <div>
 <label htmlFor="name">Name:</label>
 <input
 type="text"
 id="name"
 name="name"
 value={userData.name}
 onChange={handleChange}
 />
 </div>
 <div>
 <label htmlFor="bio">Bio:</label>
 <textarea
 id="bio"
 name="bio"
 value={userData.bio}
 onChange={handleChange}
 ></textarea>
 </div>
 </form>
);
}

// Render the component
return (
 <div>
 <h1>User Profile</h1>
 <p>Name: {userData.name}</p>
 <p>Bio: {userData.bio}</p>
 </div>
);
```

```
 />
 </div>
 <button type="submit">Save</button>
 <button type="button" onClick={toggleEdit}>
 Cancel
 </button>
 </form>
);
}

return (
 <div className="user-profile">
 <h2>{userData.name}</h2>
 <p>{userData.bio}</p>
 <button onClick={toggleEdit}>Edit Profile</button>
 </div>
);
}
```

## Component Composition Patterns

### Basic Composition:

```
// Building complex components from simpler ones
function App() {
 return (
 <div className="app">
 <Header />
 <MainContent />
 <Footer />
 </div>
);
}

function Header() {
 return (
 <header>
 <Logo />
 <Navigation />
 <SearchBar />
 </header>
);
}

function MainContent() {
 return (
 <main>
 <Sidebar />
 <ArticleList />
 </main>
);
}
```

```

}

// And so on...

```

### Containment with children prop:

```

// Generic Card component that can contain anything
function Card({ title, children }) {
 return (
 <div className="card">
 <div className="card-header">
 <h2>{title}</h2>
 </div>
 <div className="card-body">{children}</div>
 </div>
);
}

// Using the Card component
function App() {
 return (
 <div>
 <Card title="User Profile">
 <p>Name: John Doe</p>
 <p>Email: john@example.com</p>
 <button>Edit Profile</button>
 </Card>

 <Card title="Recent Posts">

 Post 1
 Post 2
 Post 3

 </Card>
 </div>
);
}

```

### Multiple Children Slots:

```

// Layout component with multiple slots
function PageLayout({ header, sidebar, content, footer }) {
 return (
 <div className="page-layout">
 <header className="header">{header}</header>
 <div className="main">
 <aside className="sidebar">{sidebar}</aside>
 <main className="content">{content}</main>
 </div>
 {footer}
 </div>
);
}

```

```

 </div>
 <footer className="footer">{footer}</footer>
 </div>
)
}

// Using the PageLayout component
function App() {
 return (
 <PageLayout
 header={<h1>My Website</h1>}
 sidebar={
 <nav>

 Home

 About

 Contact

 </nav>
 }
 content={
 <article>
 <h2>Welcome to my website</h2>
 <p>This is the main content.</p>
 </article>
 }
 footer={<p>© 2025 My Website</p>}
 />
);
}
}

```

## Specialization:

```

// Generic Button component
function Button({ children, onClick, type = 'button', className = '' }) {
 return (
 <button type={type} onClick={onClick} className={`button ${className}`}>
 {children}
 </button>
);
}

// Specialized buttons that use the generic Button
function PrimaryButton(props) {
 return <Button {...props} className="button-primary" />;
}

```

```

function SecondaryButton(props) {
 return <Button {...props} className="button-secondary" />;
}

function DangerButton(props) {
 return <Button {...props} className="button-danger" />;
}

// Usage
function App() {
 return (
 <div>
 <PrimaryButton onClick={() => console.log('Primary clicked')}>
 Primary Action
 </PrimaryButton>

 <SecondaryButton onClick={() => console.log('Secondary clicked')}>
 Secondary Action
 </SecondaryButton>

 <DangerButton onClick={() => console.log('Danger clicked')}>
 Delete
 </DangerButton>
 </div>
);
}

```

## Render Props Pattern:

```

// Component with render prop
function DataFetcher({ url, render }) {
 const [data, setData] = useState(null);
 const [isLoading, setIsLoading] = useState(true);
 const [error, setError] = useState(null);

 useEffect(() => {
 async function fetchData() {
 setIsLoading(true);
 try {
 const response = await fetch(url);
 if (!response.ok) {
 throw new Error('Network response was not ok');
 }
 const result = await response.json();
 setData(result);
 setError(null);
 } catch (err) {
 setError(err.message);
 setData(null);
 } finally {
 setIsLoading(false);
 }
 }
 fetchData();
 }, [url]);
 render(data);
}

```

```
 }
 }

 fetchData();
}, [url]);

return render({ data, isLoading, error });
}

// Using the DataFetcher
function UserProfile({ userId }) {
 return (
 <DataFetcher
 url={`https://api.example.com/users/${userId}`}
 render={({ data, isLoading, error }) => {
 if (isLoading) return <p>Loading user data...</p>;
 if (error) return <p>Error: {error}</p>;
 if (!data) return <p>No user data available</p>;
 }}
 >
);
}

// Alternative with children as a function
function DataFetcher({ url, children }) {
 const [data, setData] = useState(null);
 const [isLoading, setIsLoading] = useState(true);
 const [error, setError] = useState(null);

 useEffect(() => {
 // Same fetching logic as above
 }, [url]);

 return children({ data, isLoading, error });
}

// Using with children as a function
function UserPosts({ userId }) {
 return (
 <DataFetcher url={`https://api.example.com/users/${userId}/posts`} >
 {({ data, isLoading, error }) => {
 if (isLoading) return <p>Loading posts...</p>;
 if (error) return <p>Error: {error}</p>;
 if (!data || data.length === 0) return <p>No posts available</p>;
 }}
 </DataFetcher>
);
}
```

```
<div>
 <h2>Posts</h2>

 {data.map((post) => (
 <li key={post.id}>{post.title}
)))

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

## Higher-Order Components (HOCs):

```
// HOC that adds loading state
function withLoading(WrappedComponent) {
 return function WithLoading(props) {
 const [isLoading, setIsLoading] = useState(props.isLoading || false);

 useEffect(() => {
 setIsLoading(props.isLoading || false);
 }, [props.isLoading]);

 if (isLoading) {
 return <div className="loading-spinner">Loading...</div>;
 }

 return <WrappedComponent {...props} />;
 };
}

// Component to be wrapped
function UserList({ users }) {
 return (

 {users.map((user) => (
 <li key={user.id}>{user.name}
)))

);
}

// Enhanced component with loading functionality
const UserListWithLoading = withLoading(UserList);

// Using the enhanced component
function App() {
 const [users, setUsers] = useState([]);
 const [isLoading, setIsLoading] = useState(true);
```

```

useEffect(() => {
 // Simulate fetching data
 setTimeout(() => {
 setUsers([
 { id: 1, name: 'Alice' },
 { id: 2, name: 'Bob' },
 { id: 3, name: 'Charlie' },
]);
 setIsLoading(false);
 }, 2000);
}, []);

return <UserListWithLoading users={users} isLoading={isLoading} />;
}

// Another HOC example: authentication
function withAuth(WrappedComponent) {
 return function WithAuth(props) {
 const isAuthenticated = useAuth(); // Custom hook that checks auth status

 if (!isAuthenticated) {
 return <Redirect to="/login" />;
 }

 return <WrappedComponent {...props} />;
 };
}

// Protected component
const ProtectedDashboard = withAuth(Dashboard);

```

## Compound Components:

```

// Tabs compound component
function Tabs({ children, defaultActiveIndex = 0 }) {
 const [activeIndex, setActiveIndex] = useState(defaultActiveIndex);

 // Clone children to inject activeIndex and setActiveIndex
 const enhancedChildren = React.Children.map(children, (child) => {
 return React.cloneElement(child, { activeIndex, setActiveIndex });
 });

 return <div className="tabs">{enhancedChildren}</div>;
}

function TabList({ children, activeIndex, setActiveIndex }) {
 // Map over children to add onClick handler
 const tabs = React.Children.map(children, (child, index) => {
 return React.cloneElement(child, {
 active: index === activeIndex,
 onClick: () => setActiveIndex(index),
 });
}

```

```
});

return <div className="tab-list">{tabs}</div>;
}

function Tab({ active, onClick, children }) {
 return (
 <div className={`tab ${active ? 'active' : ''}`} onClick={onClick}>
 {children}
 </div>
);
}

function TabPanels({ children, activeIndex }) {
 // Show only the active panel
 const activePanel = React.Children.toArray(children)[activeIndex];
 return <div className="tab-panels">{activePanel}</div>;
}

function TabPanel({ children }) {
 return <div className="tab-panel">{children}</div>;
}

// Export all components
Tabs.List = TabList;
Tabs.Tab = Tab;
Tabs.Panels = TabPanels;
Tabs.Panel = TabPanel;

// Usage
function App() {
 return (
 <Tabs defaultActiveIndex={0}>
 <Tabs.List>
 <Tabs.Tab>Tab 1</Tabs.Tab>
 <Tabs.Tab>Tab 2</Tabs.Tab>
 <Tabs.Tab>Tab 3</Tabs.Tab>
 </Tabs.List>
 <Tabs.Panels>
 <Tabs.Panel>
 <h2>Content for Tab 1</h2>
 <p>This is the first tab content.</p>
 </Tabs.Panel>
 <Tabs.Panel>
 <h2>Content for Tab 2</h2>
 <p>This is the second tab content.</p>
 </Tabs.Panel>
 <Tabs.Panel>
 <h2>Content for Tab 3</h2>
 <p>This is the third tab content.</p>
 </Tabs.Panel>
 </Tabs.Panels>
 </Tabs>
);
}
```

```
);
}
```

## Container/Presentational Pattern:

```
// Container component (handles data and state)
function UserListContainer() {
 const [users, setUsers] = useState([]);
 const [isLoading, setIsLoading] = useState(true);
 const [error, setError] = useState(null);

 useEffect(() => {
 async function fetchUsers() {
 try {
 setIsLoading(true);
 const response = await fetch('https://api.example.com/users');
 if (!response.ok) {
 throw new Error('Failed to fetch users');
 }
 const data = await response.json();
 setUsers(data);
 setError(null);
 } catch (err) {
 setError(err.message);
 } finally {
 setIsLoading(false);
 }
 }

 fetchUsers();
 }, []);

 const handleDeleteUser = async (userId) => {
 try {
 await fetch(`https://api.example.com/users/${userId}` , {
 method: 'DELETE',
 });
 setUsers(users.filter((user) => user.id !== userId));
 } catch (err) {
 setError('Failed to delete user');
 }
 };

 // Render the presentational component with data and handlers
 return (
 <UserList
 users={users}
 isLoading={isLoading}
 error={error}
 onDeleteUser={handleDeleteUser}
 />
);
}
```

```
}

// Presentational component (handles rendering UI)
function UserList({ users, isLoading, error, onDeleteUser }) {
 if (isLoading) {
 return <div className="loading">Loading users...</div>;
 }

 if (error) {
 return <div className="error">Error: {error}</div>;
 }

 if (users.length === 0) {
 return <div className="empty">No users found</div>;
 }

 return (
 <div className="user-list">
 <h2>Users</h2>

 {users.map((user) => (
 <li key={user.id} className="user-item">
 <div className="user-info">
 {user.name}
 {user.email}
 </div>
 <button onClick={() => onDeleteUser(user.id)}>Delete</button>

))}

 </div>
);
}
```

---

## 🏆 PART 4: REACT STATE MANAGEMENT

---

### State Management Approaches

#### Conceptual Foundations

State management is about how you organize and maintain the data that changes over time in your application. React offers several approaches for state management, from simple local state to complex global state libraries.

**Mental Model:** Think of state management as a filing system. You need to decide where to store different documents (data) so that they're accessible to the right people (components) when needed, with the right level of security (encapsulation) and organization.

**Historical Context:** React started with just component state. As applications grew more complex, patterns like lifting state up and flux architecture emerged. Eventually, libraries like Redux became popular for managing global state. React has since introduced Context API and hooks that provide more built-in options.

### Common Misconceptions:

- You always need Redux or a state management library (many apps can use built-in React features)
- Global state is always better than local state (local state is often simpler and more maintainable)
- State management is only about component data (it often includes UI state, cached data, form state, etc.)

### useState Hook

The useState hook is the fundamental way to add state to functional components in React.

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

// Basic counter with useState
function Counter() {
 // Declare a state variable with initial value
 const [count, setCount] = useState(0);

 return (
 <div>
 <p>You clicked {count} times</p>
 <button onClick={() => setCount(count + 1)}>Increment</button>
 <button onClick={() => setCount(count - 1)}>Decrement</button>
 <button onClick={() => setCount(0)}>Reset</button>
 </div>
);
}

// Multiple state variables
function UserForm() {
 const [name, setName] = useState('');
 const [email, setEmail] = useState('');
 const [age, setAge] = useState(0);

 return (
 <form>
 <div>
 <label>Name: </label>
 <input value={name} onChange={(e) => setName(e.target.value)} />
 </div>
 <div>
 <label>Email: </label>
 <input value={email} onChange={(e) => setEmail(e.target.value)} />
 </div>
 <div>
 <label>Age: </label>
 <input
 type="number"
 />
 </div>
 </form>
);
}
```

```
 value={age}
 onChange={(e) => setAge(parseInt(e.target.value) || 0)}
 />
 </div>
 </form>
);
}

// Object state
function ProfileForm() {
 const [profile, setProfile] = useState({
 name: '',
 email: '',
 bio: '',
 });

 // Update a single field in the object
 const handleChange = (e) => {
 const { name, value } = e.target;
 setProfile((prevProfile) => ({
 ...prevProfile, // Keep all other fields
 [name]: value, // Update just one field
 }));
 };

 return (
 <form>
 <div>
 <label>Name: </label>
 <input name="name" value={profile.name} onChange={handleChange} />
 </div>
 <div>
 <label>Email: </label>
 <input name="email" value={profile.email} onChange={handleChange} />
 </div>
 <div>
 <label>Bio: </label>
 <textarea name="bio" value={profile.bio} onChange={handleChange} />
 </div>
 </form>
);
}

// Array state
function TodoList() {
 const [todos, setTodos] = useState([]);
 const [newTodo, setNewTodo] = useState('');

 const addTodo = () => {
 if (newTodo.trim() === '') return;

 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
)
);
};

const deleteTodo = (id) => {
 setTodos((prevTodos) => prevTodos.filter((todo) => todo.id !== id));
};

return (
 <div>
 <h2>Todo List</h2>
 <div>
 <input
 value={newTodo}
 onChange={(e) => setNewTodo(e.target.value)}
 placeholder="Add a new todo"
 />
 <button onClick={addTodo}>Add</button>
 </div>

 {todos.map((todo) => (
 <li key={todo.id}>
 <input
 type="checkbox"
 checked={todo.completed}
 onChange={() => toggleTodo(todo.id)}
 />
 <span
 style={{
 textDecoration: todo.completed ? 'line-through' : 'none',
 }}
 >
 {todo.text}

 <button onClick={() => deleteTodo(todo.id)}>Delete</button>

))}

 </div>
);
}

// Lazy initialization (when initial state is expensive to compute)
function ExpensiveInitialState() {
 // This function only runs once during initial render
 const [state, setState] = useState(() => {
```

```
console.log('Computing initial state...');

// Expensive computation:
let result = 0;
for (let i = 0; i < 10000000; i++) {
 result += i;
}
return result;
});

return (
 <div>
 <p>Result of expensive computation: {state}</p>
 <button onClick={() => setState(0)}>Reset</button>
 </div>
);
}
```

## useReducer Hook

The useReducer hook is an alternative to useState for complex state logic. It's inspired by how Redux works.

```
import React, { useReducer } from 'react';

// Simple counter with useReducer
function counterReducer(state, action) {
 switch (action.type) {
 case 'INCREMENT':
 return { count: state.count + 1 };
 case 'DECREMENT':
 return { count: state.count - 1 };
 case 'RESET':
 return { count: 0 };
 default:
 return state;
 }
}

function Counter() {
 const [state, dispatch] = useReducer(counterReducer, { count: 0 });

 return (
 <div>
 <p>Count: {state.count}</p>
 <button onClick={() => dispatch({ type: 'INCREMENT' })}>Increment</button>
 <button onClick={() => dispatch({ type: 'DECREMENT' })}>Decrement</button>
 <button onClick={() => dispatch({ type: 'RESET' })}>Reset</button>
 </div>
);
}

// More complex example: Todo list with useReducer
```

```
function todoReducer(state, action) {
 switch (action.type) {
 case 'ADD_TODO':
 return {
 ...state,
 todos: [
 ...state.todos,
 {
 id: Date.now(),
 text: action.payload,
 completed: false,
 },
],
 };
 case 'TOGGLE_TODO':
 return {
 ...state,
 todos: state.todos.map((todo) =>
 todo.id === action.payload
 ? { ...todo, completed: !todo.completed }
 : todo
),
 };
 case 'DELETE_TODO':
 return {
 ...state,
 todos: state.todos.filter((todo) => todo.id !== action.payload),
 };
 case 'CLEAR_COMPLETED':
 return {
 ...state,
 todos: state.todos.filter((todo) => !todo.completed),
 };
 default:
 return state;
 }
}

function TodoApp() {
 const [state, dispatch] = useReducer(todoReducer, { todos: [] });
 const [text, setText] = useState('');

 const handleSubmit = (e) => {
 e.preventDefault();
 if (text.trim() === '') return;

 dispatch({ type: 'ADD_TODO', payload: text });
 setText('');
 };
}
```

```
return (
 <div>
 <h2>Todo List</h2>

 <form onSubmit={handleSubmit}>
 <input
 value={text}
 onChange={(e) => setText(e.target.value)}
 placeholder="Add a new todo"
 />
 <button type="submit">Add</button>
 </form>

 {state.todos.map((todo) => (
 <li key={todo.id}>
 <input
 type="checkbox"
 checked={todo.completed}
 onChange={() =>
 dispatch({
 type: 'TOGGLE_TODO',
 payload: todo.id,
 })
 }
 />
 <span
 style={{
 textDecoration: todo.completed ? 'line-through' : 'none',
 }}
 >
 {todo.text}

 <button
 onClick={() =>
 dispatch({
 type: 'DELETE_TODO',
 payload: todo.id,
 })
 }
 >
 Delete
 </button>

))}

 <button onClick={() => dispatch({ type: 'CLEAR_COMPLETED' })}>
 Clear Completed
 </button>
 </div>
);
}
```

```
// Initialization with init function
function init(initialCount) {
 return { count: initialCount };
}

function counterReducer(state, action) {
 switch (action.type) {
 case 'INCREMENT':
 return { count: state.count + 1 };
 case 'DECREMENT':
 return { count: state.count - 1 };
 case 'RESET':
 return init(action.payload);
 default:
 return state;
 }
}

function Counter({ initialCount = 0 }) {
 const [state, dispatch] = useReducer(counterReducer, initialCount, init);

 return (
 <div>
 <p>Count: {state.count}</p>
 <button onClick={() => dispatch({ type: 'INCREMENT' })}>+</button>
 <button onClick={() => dispatch({ type: 'DECREMENT' })}>-</button>
 <button
 onClick={() =>
 dispatch({
 type: 'RESET',
 payload: initialCount,
 })
 }
 >
 Reset
 </button>
 </div>
);
}
```

## Context API

The Context API provides a way to share values like themes, user data, or language preferences between components without having to explicitly pass props.

```
import React, { createContext, useContext, useState } from 'react';

// Step 1: Create a context
const ThemeContext = createContext();

// Step 2: Create a provider component
```

```
function ThemeProvider({ children }) {
 const [theme, setTheme] = useState('light');

 const toggleTheme = () => {
 setTheme((prevTheme) => (prevTheme === 'light' ? 'dark' : 'light'));
 };

 // Value object with state and functions
 const value = {
 theme,
 toggleTheme,
 };

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

// Step 3: Create a custom hook for using the context
function useTheme() {
 const context = useContext(ThemeContext);
 if (context === undefined) {
 throw new Error('useTheme must be used within a ThemeProvider');
 }
 return context;
}

// Step 4: Use the context in components
function ThemedButton() {
 const { theme, toggleTheme } = useTheme();

 return (
 <button
 onClick={toggleTheme}
 style={{
 backgroundColor: theme === 'light' ? '#fff' : '#333',
 color: theme === 'light' ? '#000' : '#fff',
 border: '1px solid',
 padding: '8px 16px',
 }}
 >
 Toggle Theme
 </button>
);
}

function ThemedHeader() {
 const { theme } = useTheme();

 return (
 <header
 style={{
 backgroundColor: theme === 'light' ? '#f5f5f5' : '#222',
 color: theme === 'light' ? '#333' : '#fff',
 }}
 >
 Header
 </header>
);
}
```

```
 padding: '1rem',
)}
>
<h1>Themed App</h1>
<p>Current theme: {theme}</p>
</header>
);
}

// Step 5: Wrap your app with the provider
function App() {
 return (
 <ThemeProvider>
 <div className="app">
 <ThemedHeader />
 <main style={{ padding: '1rem' }}>
 <ThemedButton />
 </main>
 </div>
 </ThemeProvider>
);
}

// More complex example: User authentication context
const AuthContext = createContext();

function AuthProvider({ children }) {
 const [user, setUser] = useState(null);
 const [loading, setLoading] = useState(false);
 const [error, setError] = useState(null);

 const login = async (email, password) => {
 try {
 setLoading(true);
 setError(null);

 // Simulate API call
 await new Promise((resolve) => setTimeout(resolve, 1000));

 if (email === 'user@example.com' && password === 'password') {
 const userData = { id: 1, name: 'John Doe', email };
 setUser(userData);
 return userData;
 } else {
 throw new Error('Invalid credentials');
 }
 } catch (err) {
 setError(err.message);
 throw err;
 } finally {
 setLoading(false);
 }
 };
}
```

```
const logout = async () => {
 try {
 setLoading(true);

 // Simulate API call
 await new Promise((resolve) => setTimeout(resolve, 500));

 setUser(null);
 } catch (err) {
 setError(err.message);
 } finally {
 setLoading(false);
 }
};

const register = async (name, email, password) => {
 try {
 setLoading(true);
 setError(null);

 // Simulate API call
 await new Promise((resolve) => setTimeout(resolve, 1500));

 // Simulate successful registration
 const userData = { id: Date.now(), name, email };
 setUser(userData);
 return userData;
 } catch (err) {
 setError(err.message);
 throw err;
 } finally {
 setLoading(false);
 }
};

const value = {
 user,
 loading,
 error,
 login,
 logout,
 register,
};

return <AuthContext.Provider value={value}>{children}</AuthContext.Provider>;
}

function useAuth() {
 const context = useContext(AuthContext);
 if (context === undefined) {
 throw new Error('useAuth must be used within anAuthProvider');
 }
 return context;
}
```

```
// Using the auth context
function LoginPage() {
 const { login, loading, error } = useAuth();
 const [email, setEmail] = useState('');
 const [password, setPassword] = useState('');

 const handleSubmit = async (e) => {
 e.preventDefault();
 try {
 await login(email, password);
 // Redirect after successful login
 } catch (err) {
 // Error is already captured in the context
 }
 };

 return (
 <form onSubmit={handleSubmit}>
 {error && <div className="error">{error}</div>}
 <div>
 <label>Email:</label>
 <input
 type="email"
 value={email}
 onChange={(e) => setEmail(e.target.value)}
 required
 />
 </div>
 <div>
 <label>Password:</label>
 <input
 type="password"
 value={password}
 onChange={(e) => setPassword(e.target.value)}
 required
 />
 </div>
 <button type="submit" disabled={loading}>
 {loading ? 'Logging in...' : 'Login'}
 </button>
 </form>
);
}

function Profile() {
 const { user, logout } = useAuth();

 if (!user) {
 return <p>Please login to view your profile</p>;
 }

 return (
 <div>
```

```
<h2>Profile</h2>
<p>Name: {user.name}</p>
<p>Email: {user.email}</p>
<button onClick={logout}>Logout</button>
</div>
);
}

// Using multiple contexts together
function App() {
 return (
 <AuthProvider>
 <ThemeProvider>
 <div className="app">
 <ThemedHeader />
 <main>
 <LoginPage />
 <Profile />
 <ThemedButton />
 </main>
 </div>
 </ThemeProvider>
 </AuthProvider>
);
}

// Context with useReducer for complex state management
const initialState = {
 cart: [],
 totalItems: 0,
 totalAmount: 0,
};

function cartReducer(state, action) {
 switch (action.type) {
 case 'ADD_TO_CART':
 const { product } = action.payload;
 // Check if product already exists in cart
 const existingItem = state.cart.find((item) => item.id === product.id);

 if (existingItem) {
 // Update quantity of existing item
 const updatedCart = state.cart.map((item) =>
 item.id === product.id
 ? { ...item, quantity: item.quantity + 1 }
 : item
);
 }

 return {
 ...state,
 cart: updatedCart,
 totalItems: state.totalItems + 1,
 totalAmount: state.totalAmount + product.price,
 };
}
```

```
 } else {
 // Add new item to cart
 const newItem = {
 ...product,
 quantity: 1,
 };

 return {
 ...state,
 cart: [...state.cart, newItem],
 totalItems: state.totalItems + 1,
 totalAmount: state.totalAmount + product.price,
 };
 }

 case 'REMOVE_FROM_CART':
 const { productId } = action.payload;
 const itemToRemove = state.cart.find((item) => item.id === productId);

 if (!itemToRemove) return state;

 // If quantity is 1, remove the item completely
 if (itemToRemove.quantity === 1) {
 return {
 ...state,
 cart: state.cart.filter((item) => item.id !== productId),
 totalItems: state.totalItems - 1,
 totalAmount: state.totalAmount - itemToRemove.price,
 };
 } else {
 // Decrease quantity
 const updatedCart = state.cart.map((item) =>
 item.id === productId
 ? { ...item, quantity: item.quantity - 1 }
 : item
);

 return {
 ...state,
 cart: updatedCart,
 totalItems: state.totalItems - 1,
 totalAmount: state.totalAmount - itemToRemove.price,
 };
 }

 case 'CLEAR_CART':
 return initialState;

 default:
 return state;
}

const CartContext = createContext();
```

```

function CartProvider({ children }) {
 const [state, dispatch] = useReducer(cartReducer, initialState);

 const addToCart = (product) => {
 dispatch({
 type: 'ADD_TO_CART',
 payload: { product },
 });
 };

 const removeFromCart = (productId) => {
 dispatch({
 type: 'REMOVE_FROM_CART',
 payload: { productId },
 });
 };

 const clearCart = () => {
 dispatch({ type: 'CLEAR_CART' });
 };

 const value = {
 cart: state.cart,
 totalItems: state.totalItems,
 totalAmount: state.totalAmount,
 addToCart,
 removeFromCart,
 clearCart,
 };
}

return <CartContext.Provider value={value}>{children}</CartContext.Provider>;
}

function useCart() {
 const context = useContext(CartContext);
 if (context === undefined) {
 throw new Error('useCart must be used within a CartProvider');
 }
 return context;
}

```

## Custom Hooks for Reusable State Logic

Custom hooks are a way to extract component logic into reusable functions. They let you reuse stateful logic between components without changing your component hierarchy.

```

import { useState, useEffect, useRef, useCallback } from 'react';

// Custom hook for form handling
function useForm(initialValues = {}) {

```

```
const [values, setValues] = useState(initialValues);
const [errors, setErrors] = useState({});
const [touched, setTouched] = useState({});
const [isSubmitting, setIsSubmitting] = useState(false);

// Update form values
const handleChange = (event) => {
 const { name, value } = event.target;
 setValues((prevValues) => ({
 ...prevValues,
 [name]: value,
 }));
};

// Track which fields have been touched
const handleBlur = (event) => {
 const { name } = event.target;
 setTouched((prevTouched) => ({
 ...prevTouched,
 [name]: true,
 }));
};

// Reset the form
const resetForm = () => {
 setValues(initialValues);
 setErrors({});
 setTouched({});
 setIsSubmitting(false);
};

// For handling form submission
const handleSubmit = (submitCallback, validateFn) => {
 return async (event) => {
 event.preventDefault();

 // Validate if validation function is provided
 if (validateFn) {
 const validationErrors = validateFn(values);
 setErrors(validationErrors);

 // If there are errors, don't submit
 if (Object.keys(validationErrors).length > 0) {
 // Mark all fields as touched to show all errors
 const allTouched = Object.keys(values).reduce((acc, key) => {
 acc[key] = true;
 return acc;
 }, {});
 setTouched(allTouched);
 return;
 }
 }

 setIsSubmitting(true);
 }
}
```

```
try {
 await submitCallback(values);
 resetForm();
} catch (error) {
 console.error('Form submission error:', error);
} finally {
 setIsSubmitting(false);
}
};

return {
 values,
 errors,
 touched,
 isSubmitting,
 handleChange,
 handleBlur,
 handleSubmit,
 resetForm,
 setValues,
 setErrors,
};
}

// Example of using the useForm hook
function SignupForm() {
 const {
 values,
 errors,
 touched,
 isSubmitting,
 handleChange,
 handleBlur,
 handleSubmit,
 resetForm,
 } = useForm({
 username: '',
 email: '',
 password: '',
 confirmPassword: '',
 });

 // Validation function
 const validate = (values) => {
 const errors = {};

 if (!values.username) {
 errors.username = 'Username is required';
 }

 if (!values.email) {
 errors.email = 'Email is required';
 }
 };
}
```

```
 } else if (!/\S+@\S+\.\S+/.test(values.email)) {
 errors.email = 'Email is invalid';
 }

 if (!values.password) {
 errors.password = 'Password is required';
 } else if (values.password.length < 8) {
 errors.password = 'Password must be at least 8 characters';
 }

 if (values.password !== values.confirmPassword) {
 errors.confirmPassword = 'Passwords do not match';
 }

 return errors;
};

// Submit handler
const submitForm = async (formValues) => {
 // Simulate API request
 await new Promise((resolve) => setTimeout(resolve, 1000));
 console.log('Form submitted:', formValues);
 alert('Signup successful!');
};

return (
 <form onSubmit={handleSubmit(submitForm, validate)}>
 <div>
 <label htmlFor="username">Username</label>
 <input
 id="username"
 name="username"
 type="text"
 value={values.username}
 onChange={handleChange}
 onBlur={handleBlur}
 />
 {touched.username && errors.username && (
 <div className="error">{errors.username}</div>
)}
 </div>

 <div>
 <label htmlFor="email">Email</label>
 <input
 id="email"
 name="email"
 type="email"
 value={values.email}
 onChange={handleChange}
 onBlur={handleBlur}
 />
 {touched.email && errors.email && (
 <div className="error">{errors.email}</div>
)}
 </div>
 </form>
)
```

```
)}
 </div>

 <div>
 <label htmlFor="password">Password</label>
 <input
 id="password"
 name="password"
 type="password"
 value={values.password}
 onChange={handleChange}
 onBlur={handleBlur}
 />
 {touched.password && errors.password && (
 <div className="error">{errors.password}</div>
)}
 </div>

 <div>
 <label htmlFor="confirmPassword">Confirm Password</label>
 <input
 id="confirmPassword"
 name="confirmPassword"
 type="password"
 value={values.confirmPassword}
 onChange={handleChange}
 onBlur={handleBlur}
 />
 {touched.confirmPassword && errors.confirmPassword && (
 <div className="error">{errors.confirmPassword}</div>
)}
 </div>

 <div className="buttons">
 <button type="submit" disabled={isSubmitting}>
 {isSubmitting ? 'Signing up...' : 'Sign Up'}
 </button>
 <button type="button" onClick={resetForm}>
 Reset
 </button>
 </div>
 </form>
);

}

// Custom hook for handling API requests
function useFetch(url, options = {}) {
 const [data, setData] = useState(null);
 const [loading, setLoading] = useState(false);
 const [error, setError] = useState(null);

 // Keep track of mounted state to avoid updating state after unmount
 const mounted = useRef(true);
```

```
useEffect(() => {
 return () => {
 mounted.current = false;
 };
}, []);

const fetchData = useCallback(async () => {
 setLoading(true);
 setError(null);

 try {
 const response = await fetch(url, options);

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

 const result = await response.json();

 if (mounted.current) {
 setData(result);
 setLoading(false);
 }
 } catch (error) {
 if (mounted.current) {
 setError(error.message);
 setLoading(false);
 }
 }
}, [url, options]);

useEffect(() => {
 fetchData();
}, [fetchData]);

// Function to refetch data manually
const refetch = () => {
 fetchData();
};

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

// Example of using the useFetch hook
function UserList() {
 const { data, loading, error, refetch } = useFetch(
 'https://api.example.com/users'
);

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

 return (
 <div>
```

```
<h2>Users</h2>
<button onClick={refetch}>Refresh</button>
{data && (

 {data.map((user) => (
 <li key={user.id}>{user.name}
)))

)}
</div>
);
}

// Custom hook for local storage
function useLocalStorage(key, initialValue) {
 // State to store our value
 const [storedValue, setStoredValue] = useState(() => {
 try {
 // Get from local storage by key
 const item = window.localStorage.getItem(key);
 // Parse stored json or if none return initialValue
 return item ? JSON.parse(item) : initialValue;
 } catch (error) {
 console.error(error);
 return initialValue;
 }
 });
}

// Return a wrapped version of useState's setter function
const setValue = (value) => {
 try {
 // Allow value to be a function so we have same API as useState
 const valueToStore =
 value instanceof Function ? value(storedValue) : value;
 // Save state
 setStoredValue(valueToStore);
 // Save to local storage
 window.localStorage.setItem(key, JSON.stringify(valueToStore));
 } catch (error) {
 console.error(error);
 }
};

return [storedValue, setValue];
}

// Example of using the useLocalStorage hook
function DarkModeToggle() {
 const [darkMode, setDarkMode] = useLocalStorage('darkMode', false);

 useEffect(() => {
 // Apply dark mode to the body
 if (darkMode) {
 document.body.classList.add('dark-mode');
 }
 });
}
```

```
 } else {
 document.body.classList.remove('dark-mode');
 }
 }, [darkMode]);

 return (
 <button onClick={() => setDarkMode(!darkMode)}>
 {darkMode ? 'Switch to Light Mode' : 'Switch to Dark Mode'}
 </button>
);
}

// Custom hook for media queries
function useMediaQuery(query) {
 const [matches, setMatches] = useState(false);

 useEffect(() => {
 const mediaQuery = window.matchMedia(query);

 // Set initial value
 setMatches(mediaQuery.matches);

 // Create event listener to update state
 const handler = (event) => setMatches(event.matches);

 // Add event listener
 mediaQuery.addEventListener('change', handler);

 // Clean up
 return () => mediaQuery.removeEventListener('change', handler);
 }, [query]);

 return matches;
}

// Example of using the useMediaQuery hook
function ResponsiveLayout() {
 const isMobile = useMediaQuery('(max-width: 768px)');

 return (
 <div className={`layout ${isMobile ? 'mobile' : 'desktop'}`}>
 <h2>Current Layout: {isMobile ? 'Mobile' : 'Desktop'}</h2>
 {isMobile ? (
 <div className="mobile-nav">
 <button>Menu</button>
 </div>
) : (
 <nav className="desktop-nav">
 Home
 About
 Contact
 </nav>
)}
 </div>
)
}
```

```
);

}

// Custom hook for handling clickOutside events
function useClickOutside(ref, callback) {
 useEffect(() => {
 function handleClickOutside(event) {
 if (ref.current && !ref.current.contains(event.target)) {
 callback();
 }
 }

 // Add event listener
 document.addEventListener('mousedown', handleClickOutside);

 // Clean up
 return () => {
 document.removeEventListener('mousedown', handleClickOutside);
 };
 }, [ref, callback]);
}

// Example of using the useClickOutside hook
function Dropdown({ items = [] }) {
 const [isOpen, setIsOpen] = useState(false);
 const dropdownRef = useRef(null);

 // Close dropdown when clicking outside
 useClickOutside(dropdownRef, () => {
 if (isOpen) setIsOpen(false);
 });

 return (
 <div className="dropdown" ref={dropdownRef}>
 <button onClick={() => setIsOpen(!isOpen)}>
 {isOpen ? 'Close Menu' : 'Open Menu'}
 </button>

 {isOpen && (
 <ul className="dropdown-menu">
 {items.map((item, index) => (
 <li key={index}>{item}
)));

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

## Redux

Redux is a predictable state container for JavaScript apps. It helps you write applications that behave consistently, run in different environments, and are easy to test.

```
// Core Redux concepts
// Store - holds the state
// Actions - describe what happened
// Reducers - how state changes in response to actions

// 1. Setting up Redux:
// Install packages: redux, react-redux, redux-thunk (for async actions)
// npm install redux react-redux redux-thunk

// 2. Create a simple Redux store:

// actions.js - Action types and action creators
export const ADD_TODO = 'ADD_TODO';
export const TOGGLE_TODO = 'TOGGLE_TODO';
export const DELETE_TODO = 'DELETE_TODO';
export const SET_FILTER = 'SET_FILTER';

export const FILTER_ALL = 'FILTER_ALL';
export const FILTER_ACTIVE = 'FILTER_ACTIVE';
export const FILTER_COMPLETED = 'FILTER_COMPLETED';

// Action creators
export const addTodo = (text) => ({
 type: ADD_TODO,
 payload: {
 id: Date.now(),
 text,
 completed: false
 }
});

export const toggleTodo = (id) => ({
 type: TOGGLE_TODO,
 payload: { id }
});

export const deleteTodo = (id) => ({
 type: DELETE_TODO,
 payload: { id }
});

export const setFilter = (filter) => ({
 type: SET_FILTER,
 payload: { filter }
});

// reducers.js - Reducers for todos and filter
import { combineReducers } from 'redux';
import {
```

```
ADD_TODO,
TOGGLE_TODO,
DELETE_TODO,
SET_FILTER,
FILTER_ALL
} from './actions';

// Todos reducer
const initialTodosState = [];

function todos(state = initialTodosState, action) {
 switch (action.type) {
 case ADD_TODO:
 return [...state, action.payload];

 case TOGGLE_TODO:
 return state.map(todo =>
 todo.id === action.payload.id
 ? { ...todo, completed: !todo.completed }
 : todo
);

 case DELETE_TODO:
 return state.filter(todo => todo.id !== action.payload.id);

 default:
 return state;
 }
}

// Filter reducer
const initialFilterState = FILTER_ALL;

function filter(state = initialFilterState, action) {
 switch (action.type) {
 case SET_FILTER:
 return action.payload.filter;

 default:
 return state;
 }
}

// Combine all reducers
const rootReducer = combineReducers({
 todos,
 filter
});

export default rootReducer;

// store.js - Configure the Redux store
import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';
```

```
import rootReducer from './reducers';

const store = createStore(
 rootReducer,
 applyMiddleware(thunk) // Add middleware
);

export default store;

// 3. Connecting Redux to React:

// index.js - Wrap the app with Redux Provider
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import store from './store';
import App from './App';
import './index.css';

ReactDOM.render(
 <Provider store={store}>
 <App />
 </Provider>,
 document.getElementById('root')
);

// 4. Using Redux in components:

// TodoList.js - Component connected to Redux
import React, { useState } from 'react';
import { useSelector, useDispatch } from 'react-redux';
import {
 addTodo,
 toggleTodo,
 deleteTodo,
 setFilter,
 FILTER_ALL,
 FILTER_ACTIVE,
 FILTER_COMPLETED
} from './actions';

function TodoList() {
 const [text, setText] = useState('');
 const todos = useSelector(state => state.todos);
 const currentFilter = useSelector(state => state.filter);
 const dispatch = useDispatch();

 const handleSubmit = (e) => {
 e.preventDefault();
 if (!text.trim()) return;
 dispatch(addTodo(text));
 setText('');
 };
}
```

```
// Filter todos based on the current filter
const getFilteredTodos = () => {
 switch (currentFilter) {
 case FILTER_ACTIVE:
 return todos.filter(todo => !todo.completed);
 case FILTER_COMPLETED:
 return todos.filter(todo => todo.completed);
 case FILTER_ALL:
 default:
 return todos;
 }
};

const filteredTodos = getFilteredTodos();

return (
 <div className="todo-list">
 <h2>Redux Todo List</h2>

 <form onSubmit={handleSubmit}>
 <input
 type="text"
 value={text}
 onChange={(e) => setText(e.target.value)}
 placeholder="Add a todo"
 />
 <button type="submit">Add</button>
 </form>

 <div className="filters">
 <button
 className={currentFilter === FILTER_ALL ? 'active' : ''}
 onClick={() => dispatch(setFilter(FILTER_ALL))}
 >
 All
 </button>
 <button
 className={currentFilter === FILTER_ACTIVE ? 'active' : ''}
 onClick={() => dispatch(setFilter(FILTER_ACTIVE))}
 >
 Active
 </button>
 <button
 className={currentFilter === FILTER_COMPLETED ? 'active' : ''}
 onClick={() => dispatch(setFilter(FILTER_COMPLETED))}
 >
 Completed
 </button>
 </div>

 {filteredTodos.map(todo => (
 <li key={todo.id} className={todo.completed ? 'completed' : ''}>
 <input
 type="checkbox"
 checked={todo.completed}
 onChange={() => dispatch(setTodoCompleted(todo.id, !todo.completed))}>
 </input>
 {todo.title}

))}

 </div>
)
```

```
 type="checkbox"
 checked={todo.completed}
 onChange={() => dispatch(toggleTodo(todo.id))}
 />
 {todo.text}
 <button onClick={() => dispatch(deleteTodo(todo.id))}>
 Delete
 </button>

)));

<div className="todo-count">
 <p>{todos.filter(todo => !todo.completed).length} items left</p>
</div>
</div>
);
}

// 5. Async actions with Redux-Thunk:

// asyncActions.js - Thunk action creators for async operations
export const FETCH_TODOS_REQUEST = 'FETCH_TODOS_REQUEST';
export const FETCH_TODOS_SUCCESS = 'FETCH_TODOS_SUCCESS';
export const FETCH_TODOS_FAILURE = 'FETCH_TODOS_FAILURE';

// Action creators for async fetch
export const fetchTodosRequest = () => ({
 type: FETCH_TODOS_REQUEST
});

export const fetchTodosSuccess = (todos) => ({
 type: FETCH_TODOS_SUCCESS,
 payload: { todos }
});

export const fetchTodosFailure = (error) => ({
 type: FETCH_TODOS_FAILURE,
 payload: { error }
});

// Thunk action creator for fetching todos
export const fetchTodos = () => {
 return async (dispatch) => {
 dispatch(fetchTodosRequest());

 try {
 const response = await fetch('https://api.example.com/todos');

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

 const data = await response.json();
 }
 }
};
```

```
 dispatch(fetchTodosSuccess(data));
 } catch (error) {
 dispatch(fetchTodosFailure(error.message));
 }
};

};

// Add the async states to the todos reducer
const initialTodosState = {
 items: [],
 loading: false,
 error: null
};

function todos(state = initialTodosState, action) {
 switch (action.type) {
 case FETCH_TODOS_REQUEST:
 return {
 ...state,
 loading: true,
 error: null
 };

 case FETCH_TODOS_SUCCESS:
 return {
 ...state,
 loading: false,
 items: action.payload.todos
 };

 case FETCH_TODOS_FAILURE:
 return {
 ...state,
 loading: false,
 error: action.payload.error
 };

 case ADD_TODO:
 return {
 ...state,
 items: [...state.items, action.payload]
 };

 case TOGGLE_TODO:
 return {
 ...state,
 items: state.items.map(todo =>
 todo.id === action.payload.id
 ? { ...todo, completed: !todo.completed }
 : todo
)
 };

 case DELETE_TODO:
```

```
 return {
 ...state,
 items: state.items.filter(todo => todo.id !== action.payload.id)
 };

 default:
 return state;
 }
}

// Use the async action in a component
function TodoListWithFetch() {
 const { items, loading, error } = useSelector(state => state.todos);
 const dispatch = useDispatch();

 useEffect(() => {
 dispatch(fetchTodos());
 }, [dispatch]);

 if (loading) {
 return <div>Loading todos...</div>;
 }

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

 return (
 <div>
 <h2>Todos from API</h2>

 {items.map(todo => (
 <li key={todo.id}>
 {todo.title || todo.text}

))}

 </div>
);
}

// 6. Redux Toolkit (a simpler way to use Redux)

// npm install @reduxjs/toolkit

// todoSlice.js - Using createSlice from Redux Toolkit
import { createSlice, createAsyncThunk } from '@reduxjs/toolkit';

// Create an async thunk for fetching todos
export const fetchTodos = createAsyncThunk(
 'todos/fetchTodos',
 async () => {
 const response = await fetch('https://api.example.com/todos');
 if (!response.ok) {
```

```
 throw new Error('Failed to fetch todos');
 }
 return response.json();
}
);

// Create a slice for todos
const todoSlice = createSlice({
 name: 'todos',
 initialState: {
 items: [],
 loading: false,
 error: null,
 filter: 'all'
 },
 reducers: {
 // Action creators will be generated for each reducer function
 addTodo: (state, action) => {
 state.items.push({
 id: Date.now(),
 text: action.payload,
 completed: false
 });
 },
 toggleTodo: (state, action) => {
 const todo = state.items.find(todo => todo.id === action.payload);
 if (todo) {
 todo.completed = !todo.completed;
 }
 },
 deleteTodo: (state, action) => {
 state.items = state.items.filter(todo => todo.id !== action.payload);
 },
 setFilter: (state, action) => {
 state.filter = action.payload;
 }
 },
 extraReducers: (builder) => {
 // Handle async action states
 builder
 .addCase(fetchTodos.pending, (state) => {
 state.loading = true;
 state.error = null;
 })
 .addCase(fetchTodos.fulfilled, (state, action) => {
 state.loading = false;
 state.items = action.payload;
 })
 .addCase(fetchTodos.rejected, (state, action) => {
 state.loading = false;
 state.error = action.error.message;
 });
 }
});
});
```

```
// Export actions and reducer
export const { addTodo, toggleTodo, deleteTodo, setFilter } = todoSlice.actions;
export default todoSlice.reducer;

// store.js - Configure the store with Redux Toolkit
import { configureStore } from '@reduxjs/toolkit';
import todoReducer from './todoSlice';

const store = configureStore({
 reducer: {
 todos: todoReducer
 }
});

export default store;

// Using the Redux Toolkit slice in a component
import React, { useState, useEffect } from 'react';
import { useSelector, useDispatch } from 'react-redux';
import {
 addTodo,
 toggleTodo,
 deleteTodo,
 setFilter,
 fetchTodos
} from './todoSlice';

function TodoListReduxToolkit() {
 const [text, setText] = useState('');
 const { items, loading, error, filter } = useSelector(state => state.todos);
 const dispatch = useDispatch();

 useEffect(() => {
 dispatch(fetchTodos());
 }, [dispatch]);

 const handleSubmit = (e) => {
 e.preventDefault();
 if (!text.trim()) return;
 dispatch(addTodo(text));
 setText('');
 };

 // Filter todos based on the current filter
 const getFilteredTodos = () => {
 switch (filter) {
 case 'active':
 return items.filter(todo => !todo.completed);
 case 'completed':
 return items.filter(todo => todo.completed);
 case 'all':
 default:
 return items;
 }
 };
}
```

```
 }

 };

 if (loading) {
 return <div>Loading todos...</div>;
 }

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

 const filteredTodos = getFilteredTodos();

 return (
 <div className="todo-list">
 <h2>Redux Toolkit Todo List</h2>

 <form onSubmit={handleSubmit}>
 <input
 type="text"
 value={text}
 onChange={(e) => setText(e.target.value)}
 placeholder="Add a todo"
 />
 <button type="submit">Add</button>
 </form>

 <div className="filters">
 <button
 className={filter === 'all' ? 'active' : ''}
 onClick={() => dispatch(setFilter('all'))}
 >
 All
 </button>
 <button
 className={filter === 'active' ? 'active' : ''}
 onClick={() => dispatch(setFilter('active'))}
 >
 Active
 </button>
 <button
 className={filter === 'completed' ? 'active' : ''}
 onClick={() => dispatch(setFilter('completed'))}
 >
 Completed
 </button>
 </div>

 {filteredTodos.map(todo => (
 <li key={todo.id} className={todo.completed ? 'completed' : ''}>
 <input
 type="checkbox"
 checked={todo.completed}
 />

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

```
 onChange={() => dispatch(toggleTodo(todo.id))}
 />
 {todo.text}
 <button onClick={() => dispatch(deleteTodo(todo.id))}>
 Delete
 </button>

))}

</div>
<);
>
```

## 🏆 PART 5: REACT ECOSYSTEM

### React Hooks In Depth

#### Conceptual Foundations

Hooks let you use state and other React features without writing a class. They were introduced in React 16.8 to solve several problems in React:

1. Difficulty in reusing stateful logic between components
2. Complex components becoming hard to understand
3. Classes being confusing for humans and machines

**Mental Model:** Think of hooks as special functions that let you "hook into" React's features. They allow functional components to remember things, perform side effects, and access React's lifecycle from simple, composable functions.

#### Built-in Hooks

##### **useState (State Management):**

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

function Counter() {
 // Basic useState with number
 const [count, setCount] = useState(0);

 // useState with callback for lazy initialization
 const [expensiveValue] = useState(() => {
 console.log('Computing initial state...');
 return computeExpensiveValue();
 });

 // useState with object
 const [user, setUser] = useState({ name: '', email: '' });
```

```

// Updating object state correctly
const handleNameChange = (e) => {
 setUser((prevUser) => ({
 ...prevUser,
 name: e.target.value,
 }));
};

// Functional updates when new state depends on old state
const increment = () => {
 setCount((prevCount) => prevCount + 1);
};

// Multiple state updates in one go
const incrementTwice = () => {
 setCount((prevCount) => prevCount + 1);
 setCount((prevCount) => prevCount + 1);
};

return (
 <div>
 <p>Count: {count}</p>
 <button onClick={increment}>Increment</button>
 <button onClick={incrementTwice}>Increment Twice</button>

 <div>
 <input
 value={user.name}
 onChange={handleNameChange}
 placeholder="Name"
 />
 <p>Name: {user.name}</p>
 </div>
 </div>
);
}

function computeExpensiveValue() {
 // Simulate expensive calculation
 let result = 0;
 for (let i = 0; i < 1000000; i++) {
 result += i;
 }
 return result;
}

```

## useEffect (Side Effects):

```

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

function DataFetcher() {

```

```
const [data, setData] = useState(null);
const [loading, setLoading] = useState(true);
const [error, setError] = useState(null);
const [count, setCount] = useState(0);

// Effect runs after every render (no dependency array)
useEffect(() => {
 console.log('Component rendered');
});

// Effect runs only on mount (empty dependency array)
useEffect(() => {
 console.log('Component mounted');

 // Cleanup function runs before component unmounts
 return () => {
 console.log('Component will unmount');
 };
}, []);

// Effect runs when count changes
useEffect(() => {
 console.log('Count changed:', count);

 // Set document title based on count
 document.title = `Count: ${count}`;

 // Cleanup function runs before next effect or unmount
 return () => {
 console.log('Cleaning up previous effect');
 };
}, [count]);

// Data fetching effect
useEffect(() => {
 let isMounted = true;

 const fetchData = async () => {
 setLoading(true);
 setError(null);

 try {
 const response = await fetch('https://api.example.com/data');

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

 const result = await response.json();

 // Only update state if component is still mounted
 if (isMounted) {
 setData(result);
 setLoading(false);
 }
 } catch (error) {
 setError(error.message);
 }
 };

 fetchData();
}, []);
}
```

```

 }
 } catch (err) {
 if (isMounted) {
 setError(err.message);
 setLoading(false);
 }
 }
};

fetchData();

// Cleanup function to handle component unmounting during fetch
return () => {
 isMounted = false;
};
}, []));

// Effect with event listener
useEffect(() => {
 const handleResize = () => {
 console.log('Window resized');
 };

 window.addEventListener('resize', handleResize);

 // Cleanup: remove event listener
 return () => {
 window.removeEventListener('resize', handleResize);
 };
}, []);

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

return (
<div>
 <h2>Data</h2>
 <pre>{JSON.stringify(data, null, 2)}</pre>
 <p>Count: {count}</p>
 <button onClick={() => setCount((c) => c + 1)}>Increment</button>
</div>
);
}

```

## useContext (Context API):

```

import React, { createContext, useContext, useState } from 'react';

// Create a context with default value
const ThemeContext = createContext('light');

function ThemedButton() {

```

```
// Use the context
const theme = useContext(ThemeContext);

return (
 <button
 style={{
 background: theme === 'dark' ? '#333' : 'fff',
 color: theme === 'dark' ? 'fff' : '#333',
 }}
 >
 I am styled based on theme: {theme}
 </button>
);
}

function ThemedSection() {
 // Nested component using context
 return (
 <div>
 <h3>Themed Section</h3>
 <ThemedButton />
 </div>
);
}

function App() {
 const [theme, setTheme] = useState('light');

 const toggleTheme = () => {
 setTheme((prevTheme) => (prevTheme === 'light' ? 'dark' : 'light'));
 };

 return (
 <div>
 <button onClick={toggleTheme}>Toggle Theme</button>

 {/* Provide the context value */}
 <ThemeContext.Provider value={theme}>
 <ThemedSection />
 </ThemeContext.Provider>
 </div>
);
}

// More complex example with multiple contexts and context with state and
functions
const UserContext = createContext();
const NotificationContext = createContext();

function UserProvider({ children }) {
 const [user, setUser] = useState(null);

 const login = (username) => {
 setUser({ username, lastLogin: new Date() });
 };
}
```

```
};

const logout = () => {
 setUser(null);
};

return (
 <UserContext.Provider value={{ user, login, logout }}>
 {children}
 </UserContext.Provider>
);
}

function NotificationProvider({ children }) {
 const [notifications, setNotifications] = useState([]);

 const addNotification = (message) => {
 const id = Date.now();
 setNotifications((prev) => [...prev, { id, message }]);

 // Auto-remove after 3 seconds
 setTimeout(() => {
 removeNotification(id);
 }, 3000);
 };

 const removeNotification = (id) => {
 setNotifications((prev) =>
 prev.filter((notification) => notification.id !== id)
);
 };

 return (
 <NotificationContext.Provider
 value={{ notifications, addNotification, removeNotification }}
 >
 {children}
 </NotificationContext.Provider>
);
}

// Custom hooks for using contexts
function useUser() {
 const context = useContext(UserContext);
 if (!context) {
 throw new Error('useUser must be used within a UserProvider');
 }
 return context;
}

function useNotifications() {
 const context = useContext(NotificationContext);
 if (!context) {
 throw new Error(
 'useNotifications must be used within a NotificationProvider'
);
 }
 return context;
}
```

```
'useNotifications must be used within a NotificationProvider'
);
}
return context;
}

// Using multiple contexts in a component
function ProfilePage() {
 const { user, logout } = useUser();
 const { addNotification } = useNotifications();

 const handleLogout = () => {
 logout();
 addNotification('You have been logged out');
 };

 if (!user) {
 return <LoginForm />;
 }

 return (
 <div>
 <h2>Welcome, {user.username}</h2>
 <p>Last login: {user.lastLogin.toString()}</p>
 <button onClick={handleLogout}>Logout</button>
 </div>
);
}

function LoginForm() {
 const [username, setUsername] = useState('');
 const { login } = useUser();
 const { addNotification } = useNotifications();

 const handleSubmit = (e) => {
 e.preventDefault();
 if (username.trim()) {
 login(username);
 addNotification(`Welcome back, ${username}!`);
 }
 };

 return (
 <form onSubmit={handleSubmit}>
 <input
 value={username}
 onChange={(e) => setUsername(e.target.value)}
 placeholder="Username"
 />
 <button type="submit">Login</button>
 </form>
);
}
```

```

function NotificationList() {
 const { notifications, removeNotification } = useNotifications();

 return (
 <div className="notifications">
 {notifications.map((notification) => (
 <div key={notification.id} className="notification">
 {notification.message}
 <button onClick={() => removeNotification(notification.id)}>
 Dismiss
 </button>
 </div>
))}
 </div>
);
}

function AppWithProviders() {
 return (
 <UserProvider>
 <NotificationProvider>
 <div className="app">
 <NotificationList />
 <ProfilePage />
 </div>
 </NotificationProvider>
 </UserProvider>
);
}

```

### useReducer (Complex State):

```

import React, { useReducer } from 'react';

// Simple counter reducer
function counterReducer(state, action) {
 switch (action.type) {
 case 'INCREMENT':
 return { count: state.count + 1 };
 case 'DECREMENT':
 return { count: state.count - 1 };
 case 'RESET':
 return { count: 0 };
 case 'SET':
 return { count: action.payload };
 default:
 throw new Error(`Unsupported action type: ${action.type}`);
 }
}

function Counter() {
 const [state, dispatch] = useReducer(counterReducer, { count: 0 });

```

```
return (
 <div>
 <p>Count: {state.count}</p>
 <button onClick={() => dispatch({ type: 'INCREMENT' })}>+</button>
 <button onClick={() => dispatch({ type: 'DECREMENT' })}>-</button>
 <button onClick={() => dispatch({ type: 'RESET' })}>Reset</button>
 <button onClick={() => dispatch({ type: 'SET', payload: 10 })}>
 Set to 10
 </button>
 </div>
);
}

// More complex reducer for a form
function formReducer(state, action) {
 switch (action.type) {
 case 'CHANGE_INPUT':
 return {
 ...state,
 inputs: {
 ...state.inputs,
 [action.field]: action.payload,
 },
 };
 case 'ADD_ITEM':
 return {
 ...state,
 items: [
 ...state.items,
 { id: Date.now(), text: state.inputs.itemText },
],
 inputs: {
 ...state.inputs,
 itemText: '',
 },
 };
 case 'REMOVE_ITEM':
 return {
 ...state,
 items: state.items.filter((item) => item.id !== action.payload),
 };
 case 'TOGGLE_ITEM':
 return {
 ...state,
 items: state.items.map((item) =>
 item.id === action.payload
 ? { ...item, completed: !item.completed }
 : item
),
 };
 case 'RESET_FORM':
 return {
 ...initialFormState,
 };
 }
}
```

```
 };
 default:
 return state;
 }
}

const initialFormState = {
 inputs: {
 itemText: '',
 searchTerm: '',
 },
 items: [],
};

function TodoApp() {
 const [state, dispatch] = useReducer(formReducer, initialFormState);

 const handleSubmit = (e) => {
 e.preventDefault();
 if (state.inputs.itemText.trim()) {
 dispatch({ type: 'ADD_ITEM' });
 }
 };

 // Filter items based on search term
 const filteredItems = state.items.filter((item) =>
 item.text.toLowerCase().includes(state.inputs.searchTerm.toLowerCase())
);

 return (
 <div>
 <h2>Todo List with useReducer</h2>

 <form onSubmit={handleSubmit}>
 <input
 value={state.inputs.itemText}
 onChange={(e) =>
 dispatch({
 type: 'CHANGE_INPUT',
 field: 'itemText',
 payload: e.target.value,
 })
 }
 placeholder="Add new item"
 />
 <button type="submit">Add</button>
 </form>

 <div>
 <input
 value={state.inputs.searchTerm}
 onChange={(e) =>
 dispatch({
 type: 'CHANGE_INPUT',
 field: 'searchTerm',
 payload: e.target.value,
 })
 }
 />
 </div>
 </div>
);
}

export default TodoApp;
```

```
 field: 'searchTerm',
 payload: e.target.value,
 })
}
placeholder="Search items"
/>
</div>

{filteredItems.map((item) => (
 <li key={item.id}>
 <input
 type="checkbox"
 checked={item.completed || false}
 onChange={() =>
 dispatch({
 type: 'TOGGLE_ITEM',
 payload: item.id,
 })
 }
 />
 <span
 style={{
 textDecoration: item.completed ? 'line-through' : 'none',
 }}
 >
 {item.text}

 <button
 onClick={() =>
 dispatch({
 type: 'REMOVE_ITEM',
 payload: item.id,
 })
 }
 >
 Delete
 </button>

))}

<button onClick={() => dispatch({ type: 'RESET_FORM' })}>Reset</button>
</div>
);
}

// Combining useReducer with useContext for global state
const TodoContext = createContext();

function TodoProvider({ children }) {
 const [state, dispatch] = useReducer(formReducer, initialFormState);

 return (

```

```
<TodoContext.Provider value={{ state, dispatch }}>
 {children}
</TodoContext.Provider>
);
}

function useTodo() {
 const context = useContext(TodoContext);
 if (!context) {
 throw new Error('useTodo must be used within a TodoProvider');
 }
 return context;
}

// Components using the global state
function AddTodoForm() {
 const { state, dispatch } = useTodo();

 const handleSubmit = (e) => {
 e.preventDefault();
 if (state.inputs.itemText.trim()) {
 dispatch({ type: 'ADD_ITEM' });
 }
 };

 return (
 <form onSubmit={handleSubmit}>
 <input
 value={state.inputs.itemText}
 onChange={(e) =>
 dispatch({
 type: 'CHANGE_INPUT',
 field: 'itemText',
 payload: e.target.value,
 })
 }
 placeholder="Add new item"
 />
 <button type="submit">Add</button>
 </form>
);
}

function TodoList() {
 const { state, dispatch } = useTodo();

 return (

 {state.items.map((item) => (
 <li key={item.id}>
 <input
 type="checkbox"
 checked={item.completed || false}
 onChange={() =>
 dispatch({
 type: 'TOGGLE_COMPLETED',
 id: item.id,
 })
 }
 />

))}

);
}
```

```

 dispatch({
 type: 'TOGGLE_ITEM',
 payload: item.id,
 })
 }
 />
 <span
 style={{
 textDecoration: item.completed ? 'line-through' : 'none',
 }}
 >
 {item.text}

 <button
 onClick={() =>
 dispatch({
 type: 'REMOVE_ITEM',
 payload: item.id,
 })
 }
 >
 Delete
 </button>

)
)

);
}
}

function TodoAppWithContext() {
 return (
 <TodoProvider>
 <div>
 <h2>Todo List with Context</h2>
 <AddTodoForm />
 <TodoList />
 </div>
 </TodoProvider>
);
}

```

## useCallback and useMemo (Performance Optimization):

```

import React, { useState, useCallback, useMemo } from 'react';

// Parent component
function ParentComponent() {
 const [count, setCount] = useState(0);
 const [text, setText] = useState('');

 // This function is recreated on every render
 const regularFunction = () => {

```

```
 console.log('Regular function called, count:', count);
};

// This function is memoized and only changes when count changes
const memoizedCallback = useCallback(() => {
 console.log('Callback function called, count:', count);
}, [count]);

// Expensive calculation
const expensiveCalculation = (num) => {
 console.log('Computing expensive calculation...');
 // Simulate expensive operation
 let result = 0;
 for (let i = 0; i < 1000000000; i++) {
 result += i;
 }
 return num * 2;
};

// This calculation runs on every render
const regularResult = expensiveCalculation(count);

// This calculation is memoized and only runs when count changes
const memoizedResult = useMemo(() => {
 return expensiveCalculation(count);
}, [count]);

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

 <input
 value={text}
 onChange={(e) => setText(e.target.value)}
 placeholder="Type something...">
 />

 <p>Text: {text}</p>

 {/* Child components with different types of functions */}
 <ChildComponent
 regularFunction={regularFunction}
 memoizedCallback={memoizedCallback}
 />

 <p>Regular Result: {regularResult}</p>
 <p>Memoized Result: {memoizedResult}</p>
 </div>
);

// Child component that re-renders when props change
function ChildComponent({ regularFunction, memoizedCallback }) {
```

```
console.log('Child component rendering');

return (
 <div>
 <button onClick={regularFunction}>Call Regular Function</button>
 <button onClick={memoizedCallback}>Call Memoized Callback</button>
 </div>
);
}

// Optimized child component that only re-renders when props actually change
const MemoizedChildComponent = React.memo(function ChildComponent({
 regularFunction,
 memoizedCallback,
}) {
 console.log('Memoized child component rendering');

 return (
 <div>
 <button onClick={regularFunction}>Call Regular Function</button>
 <button onClick={memoizedCallback}>Call Memoized Callback</button>
 </div>
);
});

// How to properly use useCallback with event handlers
function SearchComponent({ onSearch }) {
 const [query, setQuery] = useState('');

 // Bad: This creates a new function on every render
 const handleSearch = () => {
 onSearch(query);
 };

 // Good: This memoizes the function and only changes when query changes
 const memoizedHandleSearch = useCallback(() => {
 onSearch(query);
 }, [onSearch, query]);

 return (
 <div>
 <input
 value={query}
 onChange={(e) => setQuery(e.target.value)}
 placeholder="Search..." />
 <button onClick={memoizedHandleSearch}>Search</button>
 </div>
);
}

// Using useMemo for derived state
function UserList({ users, searchTerm }) {
 // This filtering happens on every render
```

```

const regularFilteredUsers = users.filter((user) =>
 user.name.toLowerCase().includes(searchTerm.toLowerCase())
);

// This filtering is memoized and only runs when users or searchTerm changes
const memoizedFilteredUsers = useMemo(() => {
 console.log('Filtering users...');
 return users.filter((user) =>
 user.name.toLowerCase().includes(searchTerm.toLowerCase())
);
}, [users, searchTerm]);

return (
 <div>
 <h3>User List</h3>

 {memoizedFilteredUsers.map((user) => (
 <li key={user.id}>{user.name}
))}

 </div>
);
}

```

## useRef (DOM References and Mutable Values):

```

import React, { useState, useRef, useEffect, useCallback } from 'react';

function TextInputWithFocusButton() {
 // Create a ref to store the input DOM node
 const inputRef = useRef(null);

 // Function to focus the input
 const focusInput = () => {
 // Access the current property to get the DOM node
 inputRef.current.focus();
 };

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

// Using useRef to store previous values
function Counter() {
 const [count, setCount] = useState(0);

 // Use ref to store previous count value
 const prevCountRef = useRef();

```

```
// Update ref after render
useEffect(() => {
 prevCountRef.current = count;
}, [count]);

// Get the previous value (undefined on first render)
const prevCount = prevCountRef.current;

return (
 <div>
 <h2>
 Current: {count}, Previous: {prevCount}
 </h2>
 <button onClick={() => setCount(count + 1)}>Increment</button>
 </div>
);
}

// Using useRef for interval management
function Timer() {
 const [seconds, setSeconds] = useState(0);
 const [isRunning, setIsRunning] = useState(false);

 // Store interval ID in a ref
 const intervalRef = useRef(null);

 // Start the timer
 const startTimer = () => {
 if (intervalRef.current !== null) return; // Prevent multiple intervals

 setIsRunning(true);
 intervalRef.current = setInterval(() => {
 setSeconds((s) => s + 1);
 }, 1000);
 };

 // Stop the timer
 const stopTimer = () => {
 if (intervalRef.current === null) return; // Nothing to stop

 clearInterval(intervalRef.current);
 intervalRef.current = null;
 setIsRunning(false);
 };

 // Reset the timer
 const resetTimer = () => {
 stopTimer();
 setSeconds(0);
 };

 // Clean up interval on unmount
 useEffect(() => {

```

```
return () => {
 if (intervalRef.current !== null) {
 clearInterval(intervalRef.current);
 }
};

}, []);
```

```
return (
 <div>
 <h2>Timer: {seconds} seconds</h2>
 {!isRunning ? (
 <button onClick={startTimer}>Start</button>
) : (
 <button onClick={stopTimer}>Stop</button>
)}
 <button onClick={resetTimer}>Reset</button>
 </div>
);
}
```

```
// Using useRef to avoid recreating objects
function SearchBar({ onSearch }) {
 const [query, setQuery] = useState('');

 // Store the latest query in a ref
 const latestQuery = useRef(query);

 // Update ref when query changes
 useEffect(() => {
 latestQuery.current = query;
 }, [query]);

 // Debounce search - only call onSearch after user stops typing
 useEffect(() => {
 const timeoutId = setTimeout(() => {
 // Use the ref value to ensure we have the latest query
 if (latestQuery.current) {
 onSearch(latestQuery.current);
 }
 }, 500);

 return () => clearTimeout(timeoutId);
 }, [query, onSearch]);
}

return (
 <input
 type="text"
 value={query}
 onChange={(e) => setQuery(e.target.value)}
 placeholder="Search..." />
);
}
```

```
// Using useRef with forwardRef to pass refs to custom components
const CustomInput = React.forwardRef((props, ref) => {
 return <input ref={ref} {...props} className="custom-input" />;
});

function FormWithCustomInput() {
 const inputRef = useRef(null);

 const focusInput = () => {
 inputRef.current.focus();
 };

 return (
 <div>
 <CustomInput ref={inputRef} placeholder="Type here..." />
 <button onClick={focusInput}>Focus</button>
 </div>
);
}

// Using useRef for imperative instance variables
function StopWatch() {
 const [time, setTime] = useState(0);
 const [isRunning, setIsRunning] = useState(false);

 // Store interval ID and start time
 const stopwatchRef = useRef({
 intervalId: null,
 startTime: 0,
 });

 const start = () => {
 if (!isRunning) return;

 setIsRunning(true);
 stopwatchRef.current.startTime = Date.now() - time;
 stopwatchRef.current.intervalId = setInterval(() => {
 setTime(Date.now() - stopwatchRef.current.startTime);
 }, 10);
 };

 const stop = () => {
 if (!isRunning) return;

 clearInterval(stopwatchRef.current.intervalId);
 stopwatchRef.current.intervalId = null;
 setIsRunning(false);
 };

 const reset = () => {
 stop();
 setTime(0);
 };
}
```

```

useEffect(() => {
 return () => {
 // Clean up on unmount
 if (stopwatchRef.current.intervalId) {
 clearInterval(stopwatchRef.current.intervalId);
 }
 };
}, []);

// Format time as mm:ss:ms
const formattedTime = () => {
 const ms = Math.floor((time % 1000) / 10);
 const seconds = Math.floor((time / 1000) % 60);
 const minutes = Math.floor((time / (1000 * 60)) % 60);

 return `${minutes.toString().padStart(2, '0')}:${seconds
 .toString()
 .padStart(2, '0')}:${ms.toString().padStart(2, '0')}`;
}();

return (
 <div>
 <h2>Stopwatch</h2>
 <div className="time">{formattedTime}</div>
 <div>
 {!isRunning ? (
 <button onClick={start}>Start</button>
) : (
 <button onClick={stop}>Stop</button>
)}
 <button onClick={reset}>Reset</button>
 </div>
 </div>
);
}

```

### useLayoutEffect (DOM Measurements and Updates):

```

import React, { useState, useRef, useEffect, useLayoutEffect } from 'react';

// Comparing useEffect and useLayoutEffect
function ComparisonExample() {
 const [width, setWidth] = useState(0);
 const [height, setHeight] = useState(0);
 const [useLayout, setUseLayout] = useState(false);

 const divRef = useRef(null);

 // Toggle between useEffect and useLayoutEffect
 const toggleHook = () => {
 setUseLayout((prev) => !prev);
 };
}

```

```
// useEffect runs asynchronously after render
useEffect(() => {
 if (!useLayout && divRef.current) {
 console.log('Running effect with useEffect');
 setWidth(divRef.current.offsetWidth);
 setHeight(divRef.current.offsetHeight);
 }
}, [useLayout]);

// useLayoutEffect runs synchronously before browser paint
useLayoutEffect(() => {
 if (useLayout && divRef.current) {
 console.log('Running effect with useLayoutEffect');
 setWidth(divRef.current.offsetWidth);
 setHeight(divRef.current.offsetHeight);
 }
}, [useLayout]);

return (
 <div>
 <h2>Comparing Effect Hooks</h2>
 <p>Currently using: {useLayout ? 'useLayoutEffect' : 'useEffect'}</p>
 <button onClick={toggleHook}>Toggle Hook</button>

 <div
 ref={divRef}
 style={{
 width: '50%',
 padding: '20px',
 border: '1px solid black',
 margin: '20px 0',
 }}
 >
 This div's dimensions are measured after render
 </div>

 <p>
 Width: {width}px, Height: {height}px
 </p>
 </div>
);

// Using useLayoutEffect to prevent flickering when positioning elements
function Tooltip() {
 const [isVisible, setIsVisible] = useState(false);
 const [position, setPosition] = useState({ top: 0, left: 0 });
 const buttonRef = useRef(null);
 const tooltipRef = useRef(null);

 const showTooltip = () => {
 setIsVisible(true);
 };
}
```

```
const hideTooltip = () => {
 setIsVisible(false);
};

// Calculate tooltip position before browser paint
useLayoutEffect(() => {
 if (isVisible && buttonRef.current && tooltipRef.current) {
 const buttonRect = buttonRef.current.getBoundingClientRect();
 const tooltipRect = tooltipRef.current.getBoundingClientRect();

 // Position tooltip above the button
 setPosition({
 top: buttonRect.top - tooltipRect.height - 10,
 left: buttonRect.left + buttonRect.width / 2 - tooltipRect.width / 2,
 });
 }
}, [isVisible]);

return (
 <div style={{ padding: '100px' }}>
 <button
 ref={buttonRef}
 onMouseEnter={showTooltip}
 onMouseLeave={hideTooltip}
 >
 Hover me
 </button>

 {isVisible && (
 <div
 ref={tooltipRef}
 style={{
 position: 'fixed',
 top: `${position.top}px`,
 left: `${position.left}px`,
 background: 'black',
 color: 'white',
 padding: '5px 10px',
 borderRadius: '4px',
 zIndex: 1000,
 opacity: 0.9,
 }}
 >
 This is a tooltip
 </div>
)}
 </div>
);

// Measuring and animating element dimensions with useLayoutEffect
function ResizingBox() {
 const [size, setSize] = useState(100);
```

```
const boxRef = useRef(null);
const [boxSize, setBoxSize] = useState({ width: 0, height: 0 });

// Update size from input
const handleSizeChange = (e) => {
 setSize(Number(e.target.value));
};

// Measure actual box size after size change
useLayoutEffect(() => {
 if (boxRef.current) {
 const { offsetWidth, offsetHeight } = boxRef.current;
 setBoxSize({
 width: offsetWidth,
 height: offsetHeight,
 });
 }
}, [size]);

return (
 <div>
 <h2>Resizing Box</h2>
 <input
 type="range"
 min="50"
 max="300"
 value={size}
 onChange={handleSizeChange}
 />
 <div
 ref={boxRef}
 style={{
 width: `${size}px`,
 height: `${size}px`,
 backgroundColor: 'skyblue',
 margin: '20px 0',
 display: 'flex',
 alignItems: 'center',
 justifyContent: 'center',
 transition: 'all 0.3s ease',
 }}
 >
 Box
 </div>
 <p>
 Actual size: {boxSize.width}px x {boxSize.height}px
 </p>
 </div>
);
}
```

### useImperativeHandle (Customizing Ref Exposure):

```
import React, { useRef, useImperativeHandle, forwardRef } from 'react';

// Custom input with exposed imperative methods
const CustomTextInput = forwardRef((props, ref) => {
 const inputRef = useRef(null);

 // Expose only the methods we want to expose
 useImperativeHandle(ref, () => ({
 focus: () => {
 inputRef.current.focus();
 },
 clear: () => {
 inputRef.current.value = '';
 },
 getValue: () => {
 return inputRef.current.value;
 },
 // Don't expose the actual input DOM node!
 }));
}

return <input ref={inputRef} {...props} />;
});

// Using the CustomTextInput with its exposed methods
function Form() {
 const inputRef = useRef(null);

 const handleFocus = () => {
 // Call the exposed focus method
 inputRef.current.focus();
 };

 const handleClear = () => {
 // Call the exposed clear method
 inputRef.current.clear();
 };

 const handleGetValue = () => {
 // Call the exposed getValue method
 alert(`Current value: ${inputRef.current.getValue()}`);
 };

 return (
 <div>
 <h2>Custom Input with useImperativeHandle</h2>
 <CustomTextInput ref={inputRef} placeholder="Type something..." />
 <div>
 <button onClick={handleFocus}>Focus</button>
 <button onClick={handleClear}>Clear</button>
 <button onClick={handleGetValue}>Get Value</button>
 </div>
 </div>
);
}
```

```
}

// Multiple refs example
const MultiInput = forwardRef((props, ref) => {
 const firstNameRef = useRef(null);
 const lastNameRef = useRef(null);
 const emailRef = useRef(null);

 useImperativeHandle(ref, () => ({
 focusFirstName: () => {
 firstNameRef.current.focus();
 },
 focusLastName: () => {
 lastNameRef.current.focus();
 },
 focusEmail: () => {
 emailRef.current.focus();
 },
 getAllValues: () => {
 return {
 firstName: firstNameRef.current.value,
 lastName: lastNameRef.current.value,
 email: emailRef.current.value,
 };
 },
 }));
}

return (
 <div>
 <div>
 <label>First Name:</label>
 <input ref={firstNameRef} placeholder="First Name" />
 </div>
 <div>
 <label>Last Name:</label>
 <input ref={lastNameRef} placeholder="Last Name" />
 </div>
 <div>
 <label>Email:</label>
 <input ref={emailRef} type="email" placeholder="Email" />
 </div>
 </div>
);
});

function ComplexForm() {
 const formRef = useRef(null);

 const handleFocusFirst = () => {
 formRef.current.focusFirstName();
 };

 const handleFocusLast = () => {
 formRef.current.focusLastName();
 };
}
```

```
};

const handleFocusEmail = () => {
 formRef.current.focusEmail();
};

const handleSubmit = (e) => {
 e.preventDefault();
 // Get all values from form
 const values = formRef.current.getAllValues();
 console.log('Form values:', values);
 alert(`Form submitted with: ${JSON.stringify(values)}`);
};

return (
 <form onSubmit={handleSubmit}>
 <h2>Complex Form with Multiple Refs</h2>
 <MultiInput ref={formRef} />
 <div>
 <button type="button" onClick={handleFocusFirst}>
 Focus First Name
 </button>
 <button type="button" onClick={handleFocusLast}>
 Focus Last Name
 </button>
 <button type="button" onClick={handleFocusEmail}>
 Focus Email
 </button>
 <button type="submit">Submit</button>
 </div>
 </form>
);
}

// Animation controller component
const AnimatedBox = forwardRef((props, ref) => {
 const boxRef = useRef(null);
 const [position, setPosition] = useState({ x: 0, y: 0 });

 // Define animation methods
 useImperativeHandle(ref, () => ({
 moveRight: (distance = 50) => {
 setPosition((prev) => ({ ...prev, x: prev.x + distance }));
 },
 moveLeft: (distance = 50) => {
 setPosition((prev) => ({ ...prev, x: prev.x - distance }));
 },
 moveUp: (distance = 50) => {
 setPosition((prev) => ({ ...prev, y: prev.y - distance }));
 },
 moveDown: (distance = 50) => {
 setPosition((prev) => ({ ...prev, y: prev.y + distance }));
 },
 reset: () => {
 setPosition({ x: 0, y: 0 });
 }
 }));
});
```

```
 setPosition({ x: 0, y: 0 });
 },
 getPosition: () => position,
)));

return (
 <div
 ref={boxRef}
 style={{
 width: '100px',
 height: '100px',
 backgroundColor: 'coral',
 position: 'relative',
 transform: `translate(${position.x}px, ${position.y}px)`,
 transition: 'transform 0.3s ease',
 display: 'flex',
 alignItems: 'center',
 justifyContent: 'center',
 color: 'white',
 }}
 >
 Box
</div>
);
};

function AnimationController() {
 const boxRef = useRef(null);

 return (
 <div>
 <h2>Animation Controller</h2>
 <div style={{ padding: '100px' }}>
 <AnimatedBox ref={boxRef} />
 </div>
 <div>
 <button onClick={() => boxRef.current.moveLeft()}>Left</button>
 <button onClick={() => boxRef.current.moveRight()}>Right</button>
 <button onClick={() => boxRef.current.moveUp()}>Up</button>
 <button onClick={() => boxRef.current.moveDown()}>Down</button>
 <button onClick={() => boxRef.current.reset()}>Reset</button>
 <button
 onClick={() => {
 const pos = boxRef.current.getPosition();
 alert(`Current position: x=${pos.x}, y=${pos.y}`);
 }}
 >
 Get Position
 </button>
 </div>
 </div>
);
}
```

## useId (Generating Unique IDs):

```
import React, { useId } from 'react';

// Basic example with useId
function LabeledInput() {
 const id = useId(); // Generate a unique ID

 return (
 <div>
 <label htmlFor={id}>Label:</label>
 <input id={id} />
 </div>
);
}

// Multiple IDs from a single useId call
function AccessibleForm() {
 // Generate a base ID
 const id = useId();

 // Derive multiple IDs from the base
 const nameId = `${id}-name`;
 const emailId = `${id}-email`;
 const passwordId = `${id}-password`;

 return (
 <form>
 <div>
 <label htmlFor={nameId}>Name:</label>
 <input id={nameId} type="text" />
 </div>
 <div>
 <label htmlFor={emailId}>Email:</label>
 <input id={emailId} type="email" />
 </div>
 <div>
 <label htmlFor={passwordId}>Password:</label>
 <input id={passwordId} type="password" />
 </div>
 </form>
);
}

// Using useId for ARIA attributes
function Accordion() {
 const id = useId();
 const headingId = `${id}-heading`;
 const panelId = `${id}-panel`;
 const [isExpanded, setIsExpanded] = useState(false);

 return (
 <div>
```

```
<div>
 <h3 id={headingId}>
 <button
 onClick={() => setIsExpanded(!isExpanded)}
 aria-expanded={isExpanded}
 aria-controls={panelId}
 >
 Section Title {isExpanded ? '▲' : '▼'}
 </button>
 </h3>
 <div
 id={panelId}
 role="region"
 aria-labelledby={headingId}
 hidden={!isExpanded}
 >
 <p>Panel content goes here...</p>
 </div>
</div>
);

}

// Component with multiple instances
function Counter() {
 const id = useId();
 const [count, setCount] = useState(0);

 return (
 <div>
 <label htmlFor={`${id}-count`} >Counter value:</label>
 {count}
 <button onClick={() => setCount((c) => c + 1)} >Increment</button>
 </div>
);
}

// Using multiple instances in a list
function CounterList() {
 return (
 <div>
 <h2>Multiple Counter Instances</h2>
 <Counter />
 <Counter />
 <Counter />
 </div>
);
}

// Complex form with radio buttons and checkboxes
function SurveyForm() {
 const id = useId();

 return (
 <form>
```

```
<fieldset>
 <legend>What is your preferred programming language?</legend>
 <div>
 <input
 type="radio"
 id={`${id}-lang-js`}
 name={`${id}-lang`}
 value="javascript"
 />
 <label htmlFor={`${id}-lang-js`}>JavaScript</label>
 </div>
 <div>
 <input
 type="radio"
 id={`${id}-lang-py`}
 name={`${id}-lang`}
 value="python"
 />
 <label htmlFor={`${id}-lang-py`}>Python</label>
 </div>
 <div>
 <input
 type="radio"
 id={`${id}-lang-java`}
 name={`${id}-lang`}
 value="java"
 />
 <label htmlFor={`${id}-lang-java`}>Java</label>
 </div>
</fieldset>

<fieldset>
 <legend>Select your interests:</legend>
 <div>
 <input
 type="checkbox"
 id={`${id}-int-frontend`}
 name="interests"
 value="frontend"
 />
 <label htmlFor={`${id}-int-frontend`}>Frontend</label>
 </div>
 <div>
 <input
 type="checkbox"
 id={`${id}-int-backend`}
 name="interests"
 value="backend"
 />
 <label htmlFor={`${id}-int-backend`}>Backend</label>
 </div>
 <div>
 <input
 type="checkbox"
 />

```

```
 id={`${id}-int-devops`}
 name="interests"
 value="devops"
 />
 <label htmlFor={`${id}-int-devops`}>DevOps</label>
 </div>
 </fieldset>

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

## Custom Hooks in Depth

```
import { useState, useEffect, useRef, useCallback, useMemo } from 'react';

// Custom hook for fetching data with loading, error, and retry functionality
function useFetchWithRetry(url, options = {}, maxRetries = 3) {
 const [data, setData] = useState(null);
 const [loading, setLoading] = useState(true);
 const [error, setError] = useState(null);
 const [retryCount, setRetryCount] = useState(0);

 // Use a ref to track if the component is mounted
 const mountedRef = useRef(true);

 // Function to retry the fetch
 const retry = useCallback(() => {
 if (retryCount < maxRetries) {
 setRetryCount((prevCount) => prevCount + 1);
 setLoading(true);
 setError(null);
 }
 }, [retryCount, maxRetries]);

 // Function to reset the fetch
 const reset = useCallback(() => {
 setRetryCount(0);
 setLoading(true);
 setError(null);
 setData(null);
 }, []);

 // Main fetch logic
 useEffect(() => {
 let timeoutId;

 const fetchData = async () => {
 try {
 // Add exponential backoff when retrying
 // ...
 } catch (error) {
 setError(error);
 }
 };

 if (!mountedRef.current) {
 return;
 }

 fetchData();
 }, [reset]);
}

export default useFetchWithRetry;
```

```
if (retryCount > 0) {
 const backoffTime = Math.min(1000 * 2 ** retryCount, 10000);
 await new Promise((resolve) => {
 timeoutId = setTimeout(resolve, backoffTime);
 });
}

const response = await fetch(url, options);

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

const result = await response.json();

// Only update state if component is still mounted
if (mountedRef.current) {
 setData(result);
 setLoading(false);
 setError(null);
}
} catch (err) {
 // Only update state if component is still mounted
 if (mountedRef.current) {
 setError(err.message);
 setLoading(false);

 // Auto-retry if not reached max retries
 if (retryCount < maxRetries - 1) {
 retry();
 }
 }
}
};

fetchData();

// Cleanup function
return () => {
 mountedRef.current = false;
 clearTimeout(timeoutId);
};
}, [url, options, retryCount, retry, maxRetries]);

return { data, loading, error, retry, reset, retryCount };
}

// Custom hook for handling forms with validation
function useForm(initialValues = {}, validate) {
 const [values, setValues] = useState(initialValues);
 const [errors, setErrors] = useState({});
 const [touched, setTouched] = useState({});
 const [isSubmitting, setIsSubmitting] = useState(false);
 const [isValid, setIsValid] = useState(false);
```

```
// Validate when values or touched fields change
useEffect(() => {
 if (validate) {
 const validationErrors = validate(values);
 setErrors(validationErrors);
 setIsValid(Object.keys(validationErrors).length === 0);
 }
}, [values, validate]);

// Update a single field
const handleChange = useCallback((e) => {
 const { name, value, type, checked } = e.target;

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

// Handle blur event to mark field as touched
const handleBlur = useCallback((e) => {
 const { name } = e.target;

 setTouched((prevTouched) => ({
 ...prevTouched,
 [name]: true,
 }));
}, []);

// Set multiple values at once
const setMultipleValues = useCallback((newValues) => {
 setValues((prevValues) => ({
 ...prevValues,
 ...newValues,
 }));
}, []);

// Reset the form
const resetForm = useCallback(() => {
 setValues(initialValues);
 setErrors({});
 setTouched({});
 setIsSubmitting(false);
}, [initialValues]);

// Handle form submission
const handleSubmit = useCallback(
 (onSubmit) => {
 return async (e) => {
 e.preventDefault();

 // Mark all fields as touched
 const allTouched = Object.keys(values).reduce((acc, key) => {
```

```
 acc[key] = true;
 return acc;
 }, {});
}

setTouched(allTouched);

// Validate form
if (validate) {
 const validationErrors = validate(values);
 setErrors(validationErrors);

 // Don't submit if there are errors
 if (Object.keys(validationErrors).length > 0) {
 return;
 }
}

setIsSubmitting(true);

try {
 await onSubmit(values);
 resetForm();
} catch (error) {
 // Handle submission error
 setErrors((prev) => ({
 ...prev,
 submit: error.message,
 }));
} finally {
 setIsSubmitting(false);
}
};

},
[values, validate, resetForm]
);

return {
 values,
 errors,
 touched,
 isSubmitting,
 isValid,
 handleChange,
 handleBlur,
 handleSubmit,
 setValues,
 setMultipleValues,
 resetForm,
 setErrors,
};
}

// Custom hook for handling infinite scrolling
function useInfiniteScroll(callback, options = {}) {
```

```
const { threshold = 100, initialPage = 1 } = options;
const [loading, setLoading] = useState(false);
const [error, setError] = useState(null);
const [hasMore, setHasMore] = useState(true);
const [page, setPage] = useState(initialPage);

// Track loading state with ref to avoid race conditions
const loadingRef = useRef(false);

// Reference to the loader element
const loaderRef = useRef(null);

// Load more data
const loadMore = useCallback(async () => {
 // Prevent multiple concurrent loads
 if (loadingRef.current || !hasMore) return;

 loadingRef.current = true;
 setLoading(true);
 setError(null);

 try {
 const result = await callback(page);

 // Check if there's more data to load
 if (!result || (Array.isArray(result) && result.length === 0)) {
 setHasMore(false);
 } else {
 setPage((prevPage) => prevPage + 1);
 }
 } catch (err) {
 setError(err.message);
 } finally {
 setLoading(false);
 loadingRef.current = false;
 }
}, [callback, page, hasMore]);

// Set up intersection observer
useEffect(() => {
 if (!loaderRef.current) return;

 const observer = new IntersectionObserver(
 (entries) => {
 const [entry] = entries;
 if (entry.isIntersecting && hasMore && !loadingRef.current) {
 loadMore();
 }
 },
 {
 root: null,
 rootMargin: `0px 0px ${threshold}px 0px`,
 threshold: 0.1,
 }
);
}, [threshold, hasMore, loadingRef, setLoading, setError, setHasMore, setPage, callback]);
```

```
);

const currentLoader = loaderRef.current;
observer.observe(currentLoader);

return () => {
 if (currentLoader) {
 observer.unobserve(currentLoader);
 }
};

}, [loadMore, hasMore, threshold]);

// Reset the infinite scroll
const reset = useCallback(() => {
 setPage(initialPage);
 setHasMore(true);
 setError(null);
 setLoading(false);
 loadingRef.current = false;
}, [initialPage]);

return {
 loading,
 error,
 hasMore,
 page,
 loaderRef,
 loadMore,
 reset,
};
}

// Custom hook for managing window viewport size
function useWindowSize() {
 const [windowSize, setWindowSize] = useState({
 width: undefined,
 height: undefined,
 });

 useEffect(() => {
 // Handler to call on window resize
 function handleResize() {
 setWindowSize({
 width: window.innerWidth,
 height: window.innerHeight,
 });
 }

 // Add event listener
 window.addEventListener('resize', handleResize);

 // Call handler right away to get initial size
 handleResize();
 });
}
```

```
// Remove event listener on cleanup
return () => window.removeEventListener('resize', handleResize);
}, []);

return windowSize;
}

// Custom hook for persistent state with localStorage
function useLocalStorage(key, initialValue) {
 // State to store our value
 const [storedValue, setStoredValue] = useState(() => {
 if (typeof window === 'undefined') {
 return initialValue;
 }

 try {
 // Get from local storage by key
 const item = window.localStorage.getItem(key);
 // Parse stored json or return initialValue
 return item ? JSON.parse(item) : initialValue;
 } catch (error) {
 console.error(`Error reading localStorage key "${key}":`, error);
 return initialValue;
 }
 });
}

// Return a wrapped version of useState's setter function
const setValue = useCallback(
 (value) => {
 try {
 // Allow value to be a function
 const valueToStore =
 value instanceof Function ? value(storedValue) : value;

 // Save state
 setStoredValue(valueToStore);

 // Save to local storage
 if (typeof window !== 'undefined') {
 window.localStorage.setItem(key, JSON.stringify(valueToStore));
 }
 } catch (error) {
 console.error(`Error setting localStorage key "${key}":`, error);
 }
 },
 [key, storedValue]
);

// Function to remove item from localStorage
const removeValue = useCallback(() => {
 try {
 // Remove from local storage
 if (typeof window !== 'undefined') {
 window.localStorage.removeItem(key);
 }
 } catch (error) {
 console.error(`Error removing localStorage key "${key}":`, error);
 }
});
```

```
}

// Reset state
setStoredValue(initialValue);
} catch (error) {
 console.error(`Error removing localStorage key "${key}"`, error);
}
}, [key, initialValue]);

return [storedValue, setValue, removeValue];
}

// Custom hook for handling drag and drop
function useDragAndDrop(options = {}) {
 const { onDragStart, onDragEnd, onDragOver, onDrop } = options;

 const [dragging, setDragging] = useState(false);
 const [dragOver, setDragOver] = useState(false);
 const [dragData, setDragData] = useState(null);

 // Handle drag start
 const handleDragStart = useCallback(
 (e, data) => {
 setDragging(true);
 setDragData(data);

 if (e.dataTransfer) {
 e.dataTransfer.setData('text/plain', JSON.stringify(data));
 e.dataTransfer.effectAllowed = 'move';
 }

 if (onDragStart) {
 onDragStart(e, data);
 }
 },
 [onDragStart]
);

 // Handle drag end
 const handleDragEnd = useCallback(
 (e) => {
 setDragging(false);
 setDragOver(false);

 if (onDragEnd) {
 onDragEnd(e, dragData);
 }
 },
 [onDragEnd, dragData]
);

 // Handle drag over
 const handleDragOver = useCallback(
 (e) => {
```

```
e.preventDefault();
e.dataTransfer.dropEffect = 'move';
setDragOver(true);

if (onDragOver) {
 onDragOver(e);
}
},
[onDragOver]
);

// Handle drag leave
const handleDragLeave = useCallback((e) => {
 setDragOver(false);
}, []);

// Handle drop
const handleDrop = useCallback(
(e) => {
 e.preventDefault();
 setDragOver(false);

 let data;
 try {
 data = JSON.parse(e.dataTransfer.getData('text/plain'));
 } catch (err) {
 data = dragData;
 }

 if (onDrop) {
 onDrop(e, data);
 }
},
[onDrop, dragData]
);

// Create props for draggable elements
const getDraggableProps = useCallback(
(data) => ({
 draggable: true,
 onDragStart: (e) => handleDragStart(e, data),
 onDragEnd: handleDragEnd,
}),
[handleDragStart, handleDragEnd]
);

// Create props for drop targets
const getDropTargetProps = useCallback(
() => ({
 onDragOver: handleDragOver,
 onDragLeave: handleDragLeave,
 onDrop: handleDrop,
}),
[handleDragOver, handleDragLeave, handleDrop]
```

```
);

return {
 dragging,
 dragOver,
 dragData,
 getDraggableProps,
 getDropTargetProps,
};

}

// Custom hook for history/undo functionality
function useHistory(initialState) {
 const [state, setState] = useState(initialState);
 const [history, setHistory] = useState([initialState]);
 const [pointer, setPointer] = useState(0);

 // Update state and history
 const updateState = useCallback(
 (newState) => {
 const updatedState =
 typeof newState === 'function' ? newState(state) : newState;

 // Add new state to history, removing any forward history
 const newHistory = [...history.slice(0, pointer + 1), updatedState];

 setState(updatedState);
 setHistory(newHistory);
 setPointer(newHistory.length - 1);
 },
 [state, history, pointer]
);

 // Go back in history
 const undo = useCallback(() => {
 if (pointer > 0) {
 setPointer((p) => p - 1);
 setState(history[pointer - 1]);
 }
 }, [history, pointer]);

 // Go forward in history
 const redo = useCallback(() => {
 if (pointer < history.length - 1) {
 setPointer((p) => p + 1);
 setState(history[pointer + 1]);
 }
 }, [history, pointer]);

 // Reset history
 const reset = useCallback(() => {
 setState(initialState);
 setHistory([initialState]);
 setPointer(0);
 }, []);
}
```

```
}, [initialState]);

// Return state, updater, and history controls
return {
 state,
 setState: updateState,
 undo,
 redo,
 reset,
 canUndo: pointer > 0,
 canRedo: pointer < history.length - 1,
 historyLength: history.length,
 currentPosition: pointer,
};

}

// Custom hook for handling keyboard shortcuts
function useKeyboardShortcut(keys, callback, options = {}) {
 const {
 ctrl = false,
 alt = false,
 shift = false,
 meta = false,
 preventDefault = true,
 enabled = true,
 } = options;

 // Normalize key input to array
 const targetKeys = Array.isArray(keys) ? keys : [keys];

 useEffect(() => {
 if (!enabled) return;

 const handleKeyDown = (event) => {
 // Check if modifier keys match
 if (
 ctrl !== event.ctrlKey ||
 alt !== event.altKey ||
 shift !== event.shiftKey ||
 meta !== event.metaKey
) {
 return;
 }

 // Check if the pressed key matches any target key
 const pressedKey = event.key.toLowerCase();
 if (targetKeys.some((k) => k.toLowerCase() === pressedKey)) {
 if (preventDefault) {
 event.preventDefault();
 }
 callback(event);
 }
 };
 });
}
```

```
window.addEventListener('keydown', handleKeyDown);

return () => {
 window.removeEventListener('keydown', handleKeyDown);
};

}, [targetKeys, callback, ctrl, alt, shift, meta, preventDefault, enabled]);
}

// Custom hook for managing pagination
function usePagination(totalItems, options = {}) {
 const { initialPage = 1, initialPageSize = 10, maxPageButtons = 5 } = options;

 const [currentPage, setCurrentPage] = useState(initialPage);
 const [pageSize, setPageSize] = useState(initialPageSize);

 // Calculate total pages
 const totalPages = Math.max(1, Math.ceil(totalItems / pageSize));

 // Ensure current page is within valid range
 useEffect(() => {
 if (currentPage > totalPages) {
 setCurrentPage(totalPages);
 }
 }, [currentPage, totalPages]);

 // Go to a specific page
 const goToPage = useCallback(
 (page) => {
 const pageNumber = Math.min(Math.max(1, page), totalPages);
 setCurrentPage(pageNumber);
 },
 [totalPages]
);

 // Go to next page
 const nextPage = useCallback(() => {
 if (currentPage < totalPages) {
 setCurrentPage((p) => p + 1);
 }
 }, [currentPage, totalPages]);

 // Go to previous page
 const prevPage = useCallback(() => {
 if (currentPage > 1) {
 setCurrentPage((p) => p - 1);
 }
 }, [currentPage]);

 // Calculate page slice
 const startIndex = (currentPage - 1) * pageSize;
 const endIndex = Math.min(startIndex + pageSize, totalItems);

 // Generate array of page numbers to display
 const pageButtons = useMemo(() => {
```

```
const halfButtons = Math.floor(maxPageButtons / 2);
let startPage = Math.max(1, currentPage - halfButtons);
let endPage = Math.min(totalPages, startPage + maxPageButtons - 1);

// Adjust if we can't fit all buttons
if (endPage - startPage + 1 < maxPageButtons) {
 startPage = Math.max(1, endPage - maxPageButtons + 1);
}

return Array.from(
 { length: endPage - startPage + 1 },
 (_, i) => startPage + i
);
}, [currentPage, totalPages, maxPageButtons]);

return {
 currentPage,
 pageSize,
 totalPages,
 startIndex,
 endIndex,
 pageButtons,
 setPageSize,
 goToPage,
 nextPage,
 prevPage,
 canPrevPage: currentPage > 1,
 canNextPage: currentPage < totalPages,
};
}

// Custom hook for creating a debounced value
function useDebounce(value, delay) {
 const [debouncedValue, setDebouncedValue] = useState(value);

 useEffect(() => {
 // Set up timeout to update debounced value
 const timer = setTimeout(() => {
 setDebouncedValue(value);
 }, delay);

 // Clean up timer
 return () => {
 clearTimeout(timer);
 };
 }, [value, delay]);

 return debouncedValue;
}

// Custom hook for creating a throttled function
function useThrottle(callback, delay) {
 const lastCall = useRef(0);
 const timeoutRef = useRef(null);
```

```
// Clear timeout on unmount
useEffect(() => {
 return () => {
 if (timeoutRef.current) {
 clearTimeout(timeoutRef.current);
 }
 };
}, []);

// Create memoized throttled function
return useCallback(
 (...args) => {
 const now = Date.now();
 const timeSinceLastCall = now - lastCall.current;

 // If enough time has passed, call the function immediately
 if (timeSinceLastCall >= delay) {
 lastCall.current = now;
 callback(...args);
 return;
 }

 // Otherwise, schedule a call after the delay
 if (timeoutRef.current) {
 clearTimeout(timeoutRef.current);
 }

 timeoutRef.current = setTimeout(() => {
 lastCall.current = Date.now();
 callback(...args);
 }, delay - timeSinceLastCall);
 },
 [callback, delay]
);
}

// Custom hook for dark mode
function useDarkMode() {
 // Check if user has saved preference or use system preference
 const getInitialMode = () => {
 // Check localStorage
 const savedMode = localStorage.getItem('dark-mode');
 if (savedMode !== null) {
 return savedMode === 'true';
 }

 // Check system preference
 if (window.matchMedia) {
 return window.matchMedia('(prefers-color-scheme: dark)').matches;
 }

 // Default to light mode
 return false;
}
```

```
};

const [isDarkMode, setIsDarkMode] = useState(getInitialMode);

// Apply dark mode class to body
useEffect(() => {
 document.body.classList.toggle('dark-mode', isDarkMode);
 localStorage.setItem('dark-mode', isDarkMode);
}, [isDarkMode]);

// Listen for changes in system preference
useEffect(() => {
 if (!window.matchMedia) return;

 const mediaQuery = window.matchMedia('(prefers-color-scheme: dark)');
 const handleChange = () => {
 const savedMode = localStorage.getItem('dark-mode');
 if (savedMode === null) {
 setIsDarkMode(mediaQuery.matches);
 }
 };
 mediaQuery.addEventListener('change', handleChange);

 return () => {
 mediaQuery.removeEventListener('change', handleChange);
 };
}, []);

return [isDarkMode, setIsDarkMode];
}

// Custom hook for managing online/offline status
function useOnlineStatus() {
 const [isOnline, setIsOnline] = useState(
 typeof navigator !== 'undefined' ? navigator.onLine : true
);

 useEffect(() => {
 const handleOnline = () => setIsOnline(true);
 const handleOffline = () => setIsOnline(false);

 window.addEventListener('online', handleOnline);
 window.addEventListener('offline', handleOffline);

 return () => {
 window.removeEventListener('online', handleOnline);
 window.removeEventListener('offline', handleOffline);
 };
 }, []);

 return isOnline;
}
```

```
// Custom hook for measuring element size
function useElementSize() {
 const [ref, setRef] = useState(null);
 const [size, setSize] = useState({
 width: 0,
 height: 0,
 });

 const handleSize = useCallback(() => {
 if (ref) {
 const { offsetWidth, offsetHeight } = ref;
 setSize({
 width: offsetWidth,
 height: offsetHeight,
 });
 }
 }, [ref]);

 useLayoutEffect(() => {
 if (!ref) return;

 handleSize();
 });

 // Set up resize observer
 const resizeObserver = new ResizeObserver(handleSize);
 resizeObserver.observe(ref);

 return () => {
 resizeObserver.disconnect();
 };
}, [ref, handleSize]);

return [setRef, size];
}

// Custom hook for image loading state
function useImageLoader(src) {
 const [loading, setLoading] = useState(true);
 const [error, setError] = useState(null);
 const [image, setImage] = useState(null);

 useEffect(() => {
 if (!src) {
 setLoading(false);
 setError(new Error('No image source provided'));
 return;
 }

 setLoading(true);
 setError(null);

 const img = new Image();

 img.onload = () => {
```

```
 setImage(img);
 setLoading(false);
 };

 img.onerror = () => {
 setError(new Error('Failed to load image'));
 setLoading(false);
 };
}

img.src = src;

return () => {
 img.onload = null;
 img.onerror = null;
};

}, [src]);

return { loading, error, image };
}

// Custom hook for handling form file uploads
function useFileUpload(options = {}) {
 const {
 maxFiles = Infinity,
 maxSize = Infinity,
 accept = '*/*',
 onError,
 } = options;

 const [files, setFiles] = useState([]);
 const [loading, setLoading] = useState(false);
 const [error, setError] = useState(null);
 const fileInputRef = useRef(null);

 const handleError = useCallback(
 (message) => {
 const error = new Error(message);
 setError(error);

 if (onError) {
 onError(error);
 }
 },
 [onError]
);

 const validateFiles = useCallback(
 (fileList) => {
 // Check number of files
 if (fileList.length > maxFiles) {
 throw handleError(`Maximum of ${maxFiles} files allowed`);
 }
 }
);
}
```

```
// Convert FileList to array and validate each file
return Array.from(fileList).map((file) => {
 // Check file size
 if (file.size > maxSize) {
 throw handleError(
 `File ${file.name} exceeds maximum size of ${maxSize} bytes`
);
 }

 // Check file type if accept is specified
 if (accept !== '*/*') {
 const acceptTypes = accept.split(',').map((type) => type.trim());
 const fileType = file.type;

 const isAccepted = acceptTypes.some((type) => {
 // Check for exact match, wildcard match, or extension match
 if (type === fileType) return true;
 if (
 type.endsWith('/*') &&
 fileType.startsWith(type.replace('*', '/'))
)
 return true;

 // Check file extension
 const extension = `.${file.name.split('.').pop()}`;
 return type.startsWith('.') && type === extension;
 });

 if (!isAccepted) {
 throw handleError(`File ${file.name} has an invalid type`);
 }
 }

 return file;
}),
[maxFiles, maxSize, accept, handleError]
);

const handleChange = useCallback(
 (event) => {
 const fileList = event.target.files;

 if (!fileList.length) return;

 setError(null);

 try {
 const validatedFiles = validateFiles(fileList);
 setFiles((prevFiles) => [...prevFiles, ...validatedFiles]);
 } catch (error) {
 // Error is already handled in validateFiles
 if (fileInputRef.current) {
```

```
 fileInputRef.current.value = '';
 }
},
},
[validateFiles]
);

// Method to open file dialog
const openFileDialog = useCallback(() => {
 if (fileInputRef.current) {
 fileInputRef.current.click();
 }
}, []);

// Method to handle drag and drop
const handleDrop = useCallback(
 (event) => {
 event.preventDefault();
 setError(null);

 try {
 const droppedFiles = validateFiles(event.dataTransfer.files);
 setFiles((prevFiles) => [...prevFiles, ...droppedFiles]);
 } catch (error) {
 // Error is already handled in validateFiles
 }
},
[validateFiles]
);

// Method to remove a file
const removeFile = useCallback((fileToRemove) => {
 setFiles((prevFiles) => prevFiles.filter((file) => file !== fileToRemove));
}, []);

// Method to remove all files
const clearFiles = useCallback(() => {
 setFiles([]);
 setError(null);

 if (fileInputRef.current) {
 fileInputRef.current.value = '';
 }
}, []);

const uploadFiles = useCallback(
 async (url, options = {}) => {
 if (!files.length) {
 return handleError('No files to upload');
 }

 setLoading(true);
 setError(null);
 }
);
```

```
try {
 const formData = new FormData();

 files.forEach((file, index) => {
 formData.append(options.fieldName || `file-${index}`, file);
 });

 // Add any additional form data
 if (options.data) {
 Object.entries(options.data).forEach(([key, value]) => {
 formData.append(key, value);
 });
 }
}

const response = await fetch(url, {
 method: 'POST',
 ...options,
 body: formData,
});

if (!response.ok) {
 throw new Error(`Upload failed with status ${response.status}`);
}

const result = await response.json();

if (options.clearAfterUpload) {
 clearFiles();
}

 setLoading(false);
return result;
} catch (error) {
 setLoading(false);
 return handleError(`Upload failed: ${error.message}`);
},
[files, clearFiles, handleError]
);

return {
 files,
 loading,
 error,
 fileInputRef,
 handleChange,
 openFileDialog,
 handleDrop,
 removeFile,
 clearFiles,
 uploadFiles,
};
}
}
```

```
// Example of putting custom hooks together for a comprehensive application
feature
function SearchableUserList() {
 const [searchTerm, setSearchTerm] = useState('');
 const debouncedSearchTerm = useDebounce(searchTerm, 500);

 const {
 data: users,
 loading,
 error,
 refetch,
 } = useFetchWithRetry(`/api/users?search=${debouncedSearchTerm}`);
 const isOnline = useOnlineStatus();

 const {
 currentPage,
 pageSize,
 startIndex,
 endIndex,
 pageButtons,
 goToPage,
 nextPage,
 prevPage,
 canPrevPage,
 canNextPage,
 } = usePagination(users?.length || 0, { initialPageSize: 5 });

 // Get current page of users
 const currentUsers = users?.slice(startIndex, endIndex) || [];

 // Dark mode toggle
 const [isDarkMode, setIsDarkMode] = useDarkMode();

 // Handle search input change
 const handleSearchChange = (e) => {
 setSearchTerm(e.target.value);
 };

 return (
 <div className={isDarkMode ? 'dark-theme' : 'light-theme'}>
 <div className="header">
 <h2>User Directory</h2>
 <button onClick={() => setIsDarkMode(!isDarkMode)}>
 {isDarkMode ? '💡 Light Mode' : '🌙 Dark Mode'}
 </button>
 </div>

 {!isOnline && (
 <div className="offline-warning">
 You are currently offline. Some features may be unavailable.
 </div>
)}
 </div>
);
}
```

```
<div className="search-bar">
 <input
 type="text"
 placeholder="Search users..."
 value={searchTerm}
 onChange={handleSearchChange}
 />
 {debouncedSearchTerm !== searchTerm && (
 Searching...
)}
</div>

{loading ? (
 <div className="loading">Loading users...</div>
) : error ? (
 <div className="error">
 Error: {error}
 <button onClick={refetch}>Retry</button>
 </div>
) : users?.length === 0 ? (
 <div className="no-results">No users found</div>
) : (
 <>
 <ul className="user-list">
 {currentUsers.map((user) => (
 <li key={user.id} className="user-item">
 <div className="user-info">
 <h3>{user.name}</h3>
 <p>{user.email}</p>
 </div>
 <div className="user-actions">
 <button>View Profile</button>
 </div>

)))

 <div className="pagination">
 <button onClick={prevPage} disabled={!canPrevPage}>
 Previous
 </button>

 {pageButtons.map((page) => (
 <button
 key={page}
 onClick={() => goToPage(page)}
 className={currentPage === page ? 'active' : ''}
 >
 {page}
 </button>
))}
 </div>
 </>
 <button onClick={nextPage} disabled={!canNextPage}>
 Next
 </button>
)
```

```
 </button>
 </div>
 </>
)}
```

```
</div>
```

```
);
```

```
}
```

## Routing with React Router

### Conceptual Foundations

Routing in React applications allows you to show different components based on the URL path. React Router is the most popular library for implementing routing in React applications. It enables navigation between views, browser history management, and dynamic route matching.

**Mental Model:** Think of routes as "traffic directors" in your application. Just as road signs direct vehicles to different destinations, routes direct users to different components or pages based on the URL they're visiting. React Router serves as the map that connects URLs to the corresponding views.

**Historical Context:** Early React applications often used server-side routing or simple conditional rendering based on state. React Router emerged to provide a more robust, declarative routing solution that matches modern single-page application (SPA) architecture. It has gone through several major versions, with v6 being the current standard as of 2023.

### Common Misconceptions:

- React Router doesn't reload the entire page when navigating (unlike traditional server navigation)
- Routes don't automatically fetch data (you need to handle data fetching within components)
- React Router is not part of the core React library (it's a separate package that needs to be installed)

### Setting Up React Router

```
// Install using npm or yarn:
// npm install react-router-dom

// Basic setup in a React application
import React from 'react';
import ReactDOM from 'react-dom';
import { BrowserRouter, Routes, Route } from 'react-router-dom';

// Import your components
import App from './App';
import Home from './pages/Home';
import About from './pages/About';
import Contact from './pages/Contact';
import NotFound from './pages/NotFound';

ReactDOM.render(
 <BrowserRouter>
```

```
<Routes>
 <Route path="/" element={<App />}>
 <Route index element={<Home />} />
 <Route path="about" element={<About />} />
 <Route path="contact" element={<Contact />} />
 <Route path="*" element={<NotFound />} />
 </Route>
</Routes>
</BrowserRouter>,
document.getElementById('root')
);

// App.js - Parent layout component
import { Outlet } from 'react-router-dom';
import Navbar from './components/Navbar';
import Footer from './components/Footer';

function App() {
 return (
 <div className="app">
 <Navbar />
 <main>
 <Outlet /> {/* Child routes will render here */}
 </main>
 <Footer />
 </div>
);
}

export default App;
```

## Navigation Components

```
// Navbar.js - Navigation with Links
import { Link } from 'react-router-dom';

function Navbar() {
 return (
 <nav>

 <Link to="/">Home</Link>

 <Link to="/about">About</Link>

 <Link to="/contact">Contact</Link>

 </nav>
)
```

```
);

}

// Using NavLink (adds active class when route matches)
import { NavLink } from 'react-router-dom';

function Navbar() {
 return (
 <nav>

 <NavLink
 to="/"
 activeClassName={({ isActive }) => (isActive ? 'active' : '')}
 end
 >
 Home
 </NavLink>

 <NavLink
 to="/about"
 activeClassName={({ isActive }) => (isActive ? 'active' : '')}
 >
 About
 </NavLink>

 <NavLink
 to="/contact"
 activeClassName={({ isActive }) => (isActive ? 'active' : '')}
 >
 Contact
 </NavLink>

 </nav>
);
}

// Style object version
function Navbar() {
 return (
 <nav>

 <NavLink
 to="/"
 style={({ isActive }) => ({
 fontWeight: isActive ? 'bold' : 'normal',
 color: isActive ? '#0077cc' : '#333',
 })}
 end
 >

```

```

 Home
 </NavLink>

 {/* other links... */}

</nav>
);
}

// Programmatic navigation
import { useNavigate } from 'react-router-dom';

function LoginButton() {
 const navigate = useNavigate();

 const handleLogin = async (formData) => {
 try {
 await loginUser(formData);
 navigate('/dashboard'); // Navigate after successful login
 } catch (error) {
 // Handle login error
 }
 };
}

return <button onClick={handleLogin}>Log In</button>;
}

// Navigate with options
function ProfileButton() {
 const navigate = useNavigate();

 const goToProfile = () => {
 navigate('/profile', {
 replace: true, // Replace current history entry instead of adding new one
 state: { from: 'dashboard' }, // Pass data to the destination route
 });
 };

 return <button onClick={goToProfile}>View Profile</button>;
}

```

## Route Parameters and Query Strings

```

// Setting up routes with parameters
<Routes>
 <Route path="/" element={<App />}>
 <Route index element={<Home />} />
 <Route path="users" element={<UserList />} />
 <Route path="users/:userId" element={<UserProfile />} />
 <Route path="products/:category/:productId" element={<ProductDetail />} />
 </Route>

```

```
</Routes>

// Accessing route parameters
import { useParams } from 'react-router-dom';

function UserProfile() {
 const { userId } = useParams();

 return (
 <div>
 <h2>User Profile</h2>
 <p>User ID: {userId}</p>
 {/* Fetch and display user data based on userId */}
 </div>
);
}

// Accessing multiple parameters
function ProductDetail() {
 const { category, productId } = useParams();

 return (
 <div>
 <h2>Product Detail</h2>
 <p>Category: {category}</p>
 <p>Product ID: {productId}</p>
 {/* Fetch and display product data */}
 </div>
);
}

// Working with query strings
import { useSearchParams } from 'react-router-dom';

function ProductList() {
 const [searchParams, setSearchParams] = useSearchParams();

 // Read query parameters
 const page = searchParams.get('page') || '1';
 const sort = searchParams.get('sort') || 'newest';
 const category = searchParams.get('category');

 // Update query parameters
 const handleSortChange = (e) => {
 const newSort = e.target.value;
 setSearchParams((prev) => {
 prev.set('sort', newSort);
 return prev;
 });
 };

 // Set multiple parameters at once
 const applyFilters = (filters) => {
 setSearchParams({
 ...
```

```
...Object.fromEntries(searchParams),
...filters,
});
};

return (
<div>
 <h2>Product List</h2>
 <div>
 <label>
 Sort by:
 <select value={sort} onChange={handleSortChange}>
 <option value="newest">Newest</option>
 <option value="price-low">Price: Low to High</option>
 <option value="price-high">Price: High to Low</option>
 </select>
 </label>

 <button onClick={() => applyFilters({ category: 'electronics' })}>
 Electronics
 </button>
 <button onClick={() => applyFilters({ category: 'clothing' })}>
 Clothing
 </button>

 <button onClick={() => setSearchParams({})}>Clear Filters</button>
 </div>

 <p>
 Page: {page}, Sort: {sort}
 </p>
 {category && <p>Category: {category}</p>}

 /* Display products */
 <div className="products">{/* Product list */}</div>

 <div className="pagination">
 <button
 onClick={() =>
 setSearchParams({
 ...Object.fromEntries(searchParams),
 page: String(Number(page) - 1),
 })
 }
 disabled={page === '1'}
 >
 Previous Page
 </button>
 Page {page}
 <button
 onClick={() =>
 setSearchParams({
 ...Object.fromEntries(searchParams),
 page: String(Number(page) + 1),
 })
 }
 >
 Next Page
 </button>
 </div>
)
```

```
 })
 }
 >
 Next Page
 </button>
</div>
</div>
);
}

// Using location hook to access all URL information
import { useLocation } from 'react-router-dom';

function RouteInfo() {
 const location = useLocation();

 return (
 <div>
 <h3>Current Route Information</h3>
 <p>Pathname: {location.pathname}</p>
 <p>Search: {location.search}</p>
 <p>Hash: {location.hash}</p>
 {location.state && <p>State: {JSON.stringify(location.state)}</p>}
 </div>
);
}
}
```

## Nested Routes and Layouts

```
// Nested routes in the router configuration
<Routes>
 <Route path="/" element={<Layout />}>
 <Route index element={<Home />} />
 <Route path="dashboard" element={<Dashboard />}>
 <Route index element={<DashboardOverview />} />
 <Route path="stats" element={<DashboardStats />} />
 <Route path="settings" element={<DashboardSettings />} />
 </Route>
 <Route path="profile" element={<Profile />} />
 </Route>
</Routes>

// Layout component with Outlet for nested routes
function Layout() {
 return (
 <div className="app">
 <header>
 <h1>My App</h1>
 <MainNav />
 </header>
 <main>
```

```
<Outlet /> {/* Child routes render here */}
</main>
<footer>
 <p>© 2023 My App</p>
</footer>
</div>
)
}

// Dashboard with its own nested routes
function Dashboard() {
 return (
 <div className="dashboard">
 <h2>Dashboard</h2>
 <nav className="dashboard-nav">
 <NavLink to="/dashboard" end>
 Overview
 </NavLink>
 <NavLink to="/dashboard/stats">Stats</NavLink>
 <NavLink to="/dashboard/settings">Settings</NavLink>
 </nav>
 <div className="dashboard-content">
 <Outlet /> {/* Dashboard's nested routes render here */}
 </div>
 </div>
);
}
```

## Protected Routes and Authentication

```
// AuthContext for managing authentication state
import { createContext, useContext, useState, useEffect } from 'react';

const AuthContext = createContext();

export function AuthProvider({ children }) {
 const [user, setUser] = useState(null);
 const [loading, setLoading] = useState(true);

 // Check if user is already logged in on component mount
 useEffect(() => {
 const storedUser = localStorage.getItem('user');
 if (storedUser) {
 setUser(JSON.parse(storedUser));
 }
 setLoading(false);
 }, []);

 // Login function
 const login = async (credentials) => {
 try {
 // Login logic here
 } catch (error) {
 // Error handling
 }
 };

 // Logout function
 const logout = () => {
 // Logout logic here
 };

 // User context provider
 return (
 <AuthContext.Provider value={{ user, setUser, loading, setLoading, login, logout }}>
 {children}
 </AuthContext.Provider>
);
}

// Hook for useAuthContext
function useAuthContext() {
 const context = useContext(AuthContext);
 if (!context) {
 throw new Error('useAuthContext must be used within an AuthProvider');
 }
 return context;
}
```

```
// Make API call to login endpoint
const response = await fetch('/api/login', {
 method: 'POST',
 headers: { 'Content-Type': 'application/json' },
 body: JSON.stringify(credentials),
});

if (!response.ok) {
 throw new Error('Login failed');
}

const userData = await response.json();

// Store user data
setUser(userData);
localStorage.setItem('user', JSON.stringify(userData));

return userData;
} catch (error) {
 throw error;
}
};

// Logout function
const logout = () => {
 setUser(null);
 localStorage.removeItem('user');
};

const value = {
 user,
 loading,
 login,
 logout,
 isAuthenticated: !!user,
};

return <AuthContext.Provider value={value}>{children}</AuthContext.Provider>;
}

// Custom hook to use the auth context
export function useAuth() {
 const context = useContext(AuthContext);
 if (!context) {
 throw new Error('useAuth must be used within an AuthProvider');
 }
 return context;
}

// ProtectedRoute component
import { Navigate, useLocation } from 'react-router-dom';
import { useAuth } from '../contexts/auth';

function ProtectedRoute({ element }) {
```

```
const { user, loading } = useAuth();
const location = useLocation();

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

if (!user) {
 // Redirect to login, but save the location they were trying to access
 return <Navigate to="/login" state={{ from: location.pathname }} replace />;
}

return element;
}

// Using the ProtectedRoute in the router
<Routes>
 <Route path="/" element={<Layout />}>
 <Route index element={<Home />} />
 <Route path="login" element={<Login />} />
 <Route path="register" element={<Register />} />

 {/* Protected routes */}
 <Route
 path="dashboard"
 element={<ProtectedRoute element={<Dashboard />} />}
 />
 <Route path="profile" element={<ProtectedRoute element={<Profile />} />} />
 <Route
 path="settings"
 element={<ProtectedRoute element={<Settings />} />}
 />
 </Route>
</Routes>

// Login component that redirects after successful login
import { useNavigate, useLocation } from 'react-router-dom';
import { useAuth } from '../contexts/auth';

function Login() {
 const navigate = useNavigate();
 const location = useLocation();
 const { login } = useAuth();

 const [email, setEmail] = useState('');
 const [password, setPassword] = useState('');
 const [error, setError] = useState('');

 // Get the page they were trying to access
 const from = location.state?.from || '/dashboard';

 const handleSubmit = async (e) => {
 e.preventDefault();
 setError('');
 }
}
```

```
try {
 await login({ email, password });

 // Redirect to the page they were trying to access
 navigate(from, { replace: true });
} catch (err) {
 setError('Invalid email or password');
}

};

return (
 <div className="login-page">
 <h2>Log In</h2>
 {error && <div className="error">{error}</div>}

 <form onSubmit={handleSubmit}>
 <div>
 <label htmlFor="email">Email</label>
 <input
 id="email"
 type="email"
 value={email}
 onChange={(e) => setEmail(e.target.value)}
 required
 />
 </div>

 <div>
 <label htmlFor="password">Password</label>
 <input
 id="password"
 type="password"
 value={password}
 onChange={(e) => setPassword(e.target.value)}
 required
 />
 </div>

 <button type="submit">Log In</button>
 </form>
 </div>
);
}

// Role-based access control
function ProtectedRouteWithRole({ element, requiredRole }) {
 const { user, loading } = useAuth();
 const location = useLocation();

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

 return element;
}
```

```
if (!user) {
 return <Navigate to="/login" state={{ from: location.pathname }} replace />;
}

// Check for the required role
if (requiredRole && !user.roles.includes(requiredRole)) {
 return <Navigate to="/unauthorized" replace />;
}

return element;
}

// Using role-based routes
<Route
 path="admin"
 element={
 <ProtectedRouteWithRole element={<AdminDashboard />} requiredRole="admin" />
 }
/>;
```

## Handling Navigation Events and Route Transitions

```
// Using useNavigationType to determine how navigation occurred
import { useNavigationType } from 'react-router-dom';

function NavigationInfo() {
 const navigationType = useNavigationType();

 return (
 <div>
 <p>
 Navigation type: {navigationType}
 {navigationType === 'POP' && ' (user clicked back/forward button)'}
 {navigationType === 'PUSH' && ' (programmatic navigation)'}
 {navigationType === 'REPLACE' && ' (location was replaced)'}
 </p>
 </div>
);
}

// Using the Prompt component for navigation confirmation (custom implementation)
function usePrompt(message, when = true) {
 const navigate = useNavigate();
 const location = useLocation();
 const [showPrompt, setShowPrompt] = useState(false);
 const [blockedLocation, setBlockedLocation] = useState(null);

 // Save the current location to compare with next
 const savedLocation = useRef(location);

 // Listen for location changes
```

```
useEffect(() => {
 if (!when) return;

 const handleBeforeUnload = (e) => {
 e.preventDefault();
 e.returnValue = message;
 return message;
 };

 window.addEventListener('beforeunload', handleBeforeUnload);

 return () => {
 window.removeEventListener('beforeunload', handleBeforeUnload);
 };
}, [message, when]);

// Create a custom history listen to catch all navigation attempts
useEffect(() => {
 if (!when) return;

 // If location changed and prompt is active, show confirmation dialog
 if (
 savedLocation.current.pathname !== location.pathname ||
 savedLocation.current.search !== location.search
) {
 setShowPrompt(true);
 setBlockedLocation(location);
 navigate(savedLocation.current); // Prevent navigation
 } else {
 savedLocation.current = location;
 }
}, [location, navigate, when]);

// Confirmation dialog component
const ConfirmNavigation = () => {
 if (!showPrompt || !blockedLocation) return null;

 const handleConfirm = () => {
 setShowPrompt(false);
 navigate(blockedLocation);
 };

 const handleCancel = () => {
 setShowPrompt(false);
 setBlockedLocation(null);
 };

 return (
 <div className="navigation-prompt">
 <p>{message}</p>
 <div>
 <button onClick={handleConfirm}>Confirm</button>
 <button onClick={handleCancel}>Cancel</button>
 </div>
 </div>
);
}
```

```
 </div>
);
};

return ConfirmNavigation;
}

// Using the custom prompt in a form
function EditProfileForm() {
 const [formData, setFormData] = useState({
 name: 'John Doe',
 email: 'john@example.com',
 });
 const [isDirty, setIsDirty] = useState(false);

 const handleChange = (e) => {
 setFormData({ ...formData, [e.target.name]: e.target.value });
 setIsDirty(true);
 };

 const handleSubmit = (e) => {
 e.preventDefault();
 // Save data
 setIsDirty(false);
 };
}

// Show prompt only if form has unsaved changes
const ConfirmNavigation = usePrompt(
 'You have unsaved changes. Are you sure you want to leave?',
 isDirty
);

return (
 <form onSubmit={handleSubmit}>
 <ConfirmNavigation />

 <div>
 <label htmlFor="name">Name</label>
 <input
 id="name"
 name="name"
 value={formData.name}
 onChange={handleChange}
 />
 </div>

 <div>
 <label htmlFor="email">Email</label>
 <input
 id="email"
 name="email"
 type="email"
 value={formData.email}
 onChange={handleChange}
 />
 </div>
)
```

```
 />
 </div>

 <button type="submit">Save Changes</button>
 </form>
);
}
```

## Advanced Routing Techniques

```
// Lazy loading routes for code splitting
import React, { Suspense, lazy } from 'react';
import { BrowserRouter, Routes, Route } from 'react-router-dom';

// Lazy load components
const Home = lazy(() => import('./pages/Home'));
const About = lazy(() => import('./pages/About'));
const Dashboard = lazy(() => import('./pages/Dashboard'));
const Profile = lazy(() => import('./pages/Profile'));
const NotFound = lazy(() => import('./pages/NotFound'));

function App() {
 return (
 <BrowserRouter>
 <Suspense fallback={<div>Loading...</div>}>
 <Routes>
 <Route path="/" element={<Home />} />
 <Route path="/about" element={<About />} />
 <Route path="/dashboard" element={<Dashboard />} />
 <Route path="/profile" element={<Profile />} />
 <Route path="/" element={<NotFound />} />
 </Routes>
 </Suspense>
 </BrowserRouter>
);
}

// Custom route transition with framer-motion
import { AnimatePresence, motion } from 'framer-motion';
import { useLocation } from 'react-router-dom';

function AnimatedRoutes() {
 const location = useLocation();

 return (
 <AnimatePresence mode="wait">
 <motion.div
 key={location.pathname}
 initial={{ opacity: 0, y: 20 }}
 animate={{ opacity: 1, y: 0 }}
 exit={{ opacity: 0, y: -20 }}
 >
```

```
 transition={{ duration: 0.3 }}
 >
 <Routes location={location}>{/* Your routes here */}</Routes>
</motion.div>
</AnimatePresence>
);
}

// Different animations for different routes
function AnimatedRoutes() {
 const location = useLocation();

 // Define different animations based on route
 const getAnimationProps = (pathname) => {
 if (pathname.startsWith('/dashboard')) {
 return {
 initial: { opacity: 0, scale: 0.9 },
 animate: { opacity: 1, scale: 1 },
 exit: { opacity: 0, scale: 1.1 },
 transition: { duration: 0.3 },
 };
 }

 if (pathname.startsWith('/profile')) {
 return {
 initial: { opacity: 0, x: 100 },
 animate: { opacity: 1, x: 0 },
 exit: { opacity: 0, x: -100 },
 transition: { duration: 0.4 },
 };
 }

 // Default animation
 return {
 initial: { opacity: 0 },
 animate: { opacity: 1 },
 exit: { opacity: 0 },
 transition: { duration: 0.2 },
 };
 };

 const animationProps = getAnimationProps(location.pathname);

 return (
 <AnimatePresence mode="wait">
 <motion.div key={location.pathname} {...animationProps}>
 <Routes location={location}>{/* Your routes here */}</Routes>
 </motion.div>
 </AnimatePresence>
);
}

// Scroll restoration
import { useEffect } from 'react';
```

```
import { useLocation } from 'react-router-dom';

function ScrollToTop() {
 const { pathname } = useLocation();

 useEffect(() => {
 window.scrollTo(0, 0);
 }, [pathname]);

 return null;
}

// Usage in your app
function App() {
 return (
 <BrowserRouter>
 <ScrollToTop />
 <Routes>{/* Routes here */}</Routes>
 </BrowserRouter>
);
}

// More advanced scroll handling
function ScrollManager() {
 const location = useLocation();

 // Save scroll position on location change
 useEffect(() => {
 // Save current scroll position for this location
 const saveScrollPosition = () => {
 const positions = JSON.parse(
 sessionStorage.getItem('scrollPositions') || '{}'
);
 positions[location.key] = window.scrollY;
 sessionStorage.setItem('scrollPositions', JSON.stringify(positions));
 };
 saveScrollPosition();
 });

 // Save position when user navigates away
 return () => saveScrollPosition();
}, [location]);

// Restore scroll position or scroll to top
useEffect(() => {
 const positions = JSON.parse(
 sessionStorage.getItem('scrollPositions') || '{}'
);
 const scrollY = positions[location.key] || 0;

 // If returning to a position, restore it, otherwise scroll to top
 if (scrollY) {
 window.scrollTo(0, scrollY);
 } else {
 window.scrollTo(0, 0);
 }
}
```

```
}, [location]);

return null;
}
```

## API Integration and Data Fetching

### Conceptual Foundations

API integration involves connecting your React application to external services or backends to fetch, send, and manipulate data. Modern React applications typically use APIs for operations like user authentication, retrieving content, and persisting user actions.

**Mental Model:** Think of APIs as messengers between your frontend React application and other services. You send requests with instructions or information, and the API sends back responses with the data you need or confirmation of your actions.

### Common Approaches:

1. **Native Fetch API:** Built into modern browsers, provides a clean way to make HTTP requests
2. **Axios:** Popular HTTP client with more features than Fetch
3. **React Query/SWR:** Libraries that add caching, background updates, and other features on top of fetch logic
4. **GraphQL Clients:** For consuming GraphQL APIs (e.g., Apollo Client, Relay)

### Basic Data Fetching with Fetch API

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

// Component that fetches data on mount
function UserList() {
 const [users, setUsers] = useState([]);
 const [loading, setLoading] = useState(true);
 const [error, setError] = useState(null);

 useEffect(() => {
 // Define async function inside useEffect
 async function fetchUsers() {
 try {
 setLoading(true);
 const response = await fetch('https://api.example.com/users');

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

 const data = await response.json();
 setUsers(data);
 setError(null);
 } catch (error) {
 setError(error.message);
 }
 }
 fetchUsers();
 }, []);

 return (
 <div>
 {loading ?
 <p>Loading...</p>
 :
 users.map(user =>
 <div>
 {user.name}
 </div>
)}
 </div>
);
}

export default UserList;
```

```
 } catch (err) {
 setError(err.message);
 setUsers([]);
 } finally {
 setLoading(false);
 }
 }

 // Call the function
 fetchUsers();
 }, []); // Empty dependency array means this runs once on mount

 if (loading) {
 return <div>Loading users...</div>;
 }

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

 return (
 <div>
 <h2>User List</h2>

 {users.map((user) => (
 <li key={user.id}>{user.name}
)));

 </div>
);
 }
}

// POST request example
function CreateUserForm() {
 const [formData, setFormData] = useState({
 name: '',
 email: '',
 role: 'user',
 });
 const [status, setStatus] = useState('idle'); // 'idle', 'loading', 'success', 'error'
 const [error, setError] = useState(null);

 const handleChange = (e) => {
 const { name, value } = e.target;
 setFormData((prev) => ({
 ...prev,
 [name]: value,
 }));
 };

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

```
try {
 setStatus('loading');

 const response = await fetch('https://api.example.com/users', {
 method: 'POST',
 headers: {
 'Content-Type': 'application/json',
 },
 body: JSON.stringify(formData),
 });

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

 const data = await response.json();
 setStatus('success');

 // Reset form after successful submission
 setFormData({
 name: '',
 email: '',
 role: 'user',
 });

 // You might want to do something with the response data
 console.log('User created:', data);
} catch (err) {
 setStatus('error');
 setError(err.message);
}
};

return (
 <div>
 <h2>Create New User</h2>

 {status === 'success' && (
 <div className="success-message">User created successfully!</div>
)}

 {status === 'error' && (
 <div className="error-message">Error: {error}</div>
)}

 <form onSubmit={handleSubmit}>
 <div>
 <label htmlFor="name">Name</label>
 <input
 id="name"
 name="name"
 value={formData.name}
 onChange={handleChange}
 required
 >
 </div>
 </form>
 </div>
)
```

```
 />
 </div>

 <div>
 <label htmlFor="email">Email</label>
 <input
 id="email"
 name="email"
 type="email"
 value={formData.email}
 onChange={handleChange}
 required
 />
 </div>

 <div>
 <label htmlFor="role">Role</label>
 <select
 id="role"
 name="role"
 value={formData.role}
 onChange={handleChange}
 >
 <option value="user">User</option>
 <option value="admin">Admin</option>
 <option value="editor">Editor</option>
 </select>
 </div>

 <button type="submit" disabled={status === 'loading'}>
 {status === 'loading' ? 'Creating...' : 'Create User'}
 </button>
</form>
</div>
);

}

// PUT/PATCH request example
function EditUserForm({ userId, initialValue, onSuccess }) {
 const [formData, setFormData] = useState(
 initialValue || {
 name: '',
 email: '',
 role: 'user',
 }
);
 const [status, setStatus] = useState('idle');
 const [error, setError] = useState(null);

 // If initialValue changes (e.g., different user selected), update formData
 useEffect(() => {
 if (initialValue) {
 setFormData(initialValue);
 }
 }, [initialValue]);
}
```

```
}, [initialData]);

const handleChange = (e) => {
 const { name, value } = e.target;
 setFormData((prev) => ({
 ...prev,
 [name]: value,
 }));
};

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

 try {
 setStatus('loading');

 const response = await fetch(`https://api.example.com/users/${userId}`, {
 method: 'PUT', // or 'PATCH' for partial updates
 headers: {
 'Content-Type': 'application/json',
 },
 body: JSON.stringify(formData),
 });

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

 const data = await response.json();
 setStatus('success');

 // Call the success callback with updated data
 if (onSuccess) {
 onSuccess(data);
 }
 } catch (err) {
 setStatus('error');
 setError(err.message);
 }
};

return (
 <div>
 <h2>Edit User</h2>

 {status === 'success' && (
 <div className="success-message">User updated successfully!</div>
)}

 {status === 'error' && (
 <div className="error-message">Error: {error}</div>
)}

 <form onSubmit={handleSubmit}>
```

```
/* Form fields similar to CreateUserForm */}

<button type="submit" disabled={status === 'loading'}>
 {status === 'loading' ? 'Saving...' : 'Save Changes'}
</button>
</form>
</div>
);

}

// DELETE request example
function DeleteUserButton({ userId, onSuccess }) {
 const [status, setStatus] = useState('idle');

 const handleDelete = async () => {
 // Confirm before deleting
 const confirmed = window.confirm(
 'Are you sure you want to delete this user? This action cannot be undone.'
);

 if (!confirmed) return;

 try {
 setStatus('loading');

 const response = await fetch(`https://api.example.com/users/${userId}`, {
 method: 'DELETE',
 });

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

 setStatus('success');

 // Call the success callback
 if (onSuccess) {
 onSuccess();
 }
 } catch (err) {
 setStatus('error');
 alert(`Failed to delete user: ${err.message}`);
 }
 };
}

return (
 <button
 onClick={handleDelete}
 disabled={status === 'loading'}
 className="delete-button"
 >
 {status === 'loading' ? 'Deleting...' : 'Delete User'}
 </button>
)
```

```
);
 }
```

## Custom Hooks for Data Fetching

```
// Custom hook for data fetching
function useFetch(url, options = {}) {
 const [data, setData] = useState(null);
 const [loading, setLoading] = useState(true);
 const [error, setError] = useState(null);

 useEffect(() => {
 let isMounted = true;
 const controller = new AbortController();
 const signal = controller.signal;

 async function fetchData() {
 try {
 setLoading(true);

 const response = await fetch(url, {
 ...options,
 signal,
 });

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

 const result = await response.json();

 if (isMounted) {
 setData(result);
 setError(null);
 }
 } catch (err) {
 if (err.name !== 'AbortError' && isMounted) {
 setError(err.message);
 setData(null);
 }
 } finally {
 if (isMounted) {
 setLoading(false);
 }
 }
 }

 fetchData();

 // Cleanup function
 return () => {
```

```
 isMounted = false;
 controller.abort();
 };
}, [url, JSON.stringify(options)]); // Stringify options to compare objects

return { data, loading, error };
}

// Using the useFetch hook
function UserList() {
 const {
 data: users,
 loading,
 error,
 } = useFetch('https://api.example.com/users');

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

 return (
 <div>
 <h2>User List</h2>

 {users?.map((user) => (
 <li key={user.id}>{user.name}
)))

 </div>
);
}

// More comprehensive fetch hook with CRUD operations
function useApi(baseUrl) {
 const [state, setState] = useState({
 data: null,
 loading: false,
 error: null,
 });

 // GET request
 const get = useCallback(
 async (endpoint, options = {}) => {
 const url = `${baseUrl}${endpoint}`;

 setState((prev) => ({ ...prev, loading: true, error: null }));

 try {
 const response = await fetch(url, {
 method: 'GET',
 headers: {
 'Content-Type': 'application/json',
 ...options.headers,
 },
 ...options,
 });
 }
 }
);
}
```

```
});

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

const data = await response.json();
setState({ data, loading: false, error: null });
return data;
} catch (error) {
 setState({ data: null, loading: false, error: error.message });
 throw error;
}
},
[baseUrl]
);

// POST request
const post = useCallback(
 async (endpoint, payload, options = {}) => {
 const url = `${baseUrl}${endpoint}`;

 setState((prev) => ({ ...prev, loading: true, error: null }));

 try {
 const response = await fetch(url, {
 method: 'POST',
 headers: {
 'Content-Type': 'application/json',
 ...options.headers,
 },
 body: JSON.stringify(payload),
 ...options,
 });

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

 const data = await response.json();
 setState({ data, loading: false, error: null });
 return data;
 } catch (error) {
 setState({ data: null, loading: false, error: error.message });
 throw error;
 }
},
[baseUrl]
);

// PUT request
const put = useCallback(
 async (endpoint, payload, options = {}) => {
 const url = `${baseUrl}${endpoint}`;
 }
);
```

```
setState((prev) => ({ ...prev, loading: true, error: null }));

try {
 const response = await fetch(url, {
 method: 'PUT',
 headers: {
 'Content-Type': 'application/json',
 ...options.headers,
 },
 body: JSON.stringify(payload),
 ...options,
 });

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

 const data = await response.json();
 setState({ data, loading: false, error: null });
 return data;
} catch (error) {
 setState({ data: null, loading: false, error: error.message });
 throw error;
}
},
[baseUrl]
);

// PATCH request
const patch = useCallback(
 async (endpoint, payload, options = {}) => {
 const url = `${baseUrl}${endpoint}`;

 setState((prev) => ({ ...prev, loading: true, error: null }));

 try {
 const response = await fetch(url, {
 method: 'PATCH',
 headers: {
 'Content-Type': 'application/json',
 ...options.headers,
 },
 body: JSON.stringify(payload),
 ...options,
 });

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

 const data = await response.json();
 setState({ data, loading: false, error: null });
 return data;
 }
 }
);
```

```
 } catch (error) {
 setState({ data: null, loading: false, error: error.message });
 throw error;
 }
 },
 [baseUrl]
);

// DELETE request
const remove = useCallback(
 async (endpoint, options = {}) => {
 const url = `${baseUrl}${endpoint}`;

 setState((prev) => ({ ...prev, loading: true, error: null }));

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

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

 const data = await response.json();
 setState({ data, loading: false, error: null });
 return data;
 } catch (error) {
 setState({ data: null, loading: false, error: error.message });
 throw error;
 }
 },
 [baseUrl]
);

return {
 ...state,
 get,
 post,
 put,
 patch,
 remove,
};

}

// Using the comprehensive API hook
function UserPanel() {
 const api = useApi('https://api.example.com');
 const [users, setUsers] = useState([]);
}
```

```
useEffect(() => {
 // Load users on component mount
 fetchUsers();
}, []);

const fetchUsers = async () => {
 try {
 const data = await api.get('/users');
 setUsers(data);
 } catch (error) {
 console.error('Failed to fetch users:', error);
 }
};

const createUser = async (userData) => {
 try {
 const newUser = await api.post('/users', userData);
 setUsers((prevUsers) => [...prevUsers, newUser]);
 return newUser;
 } catch (error) {
 console.error('Failed to create user:', error);
 throw error;
 }
};

const updateUser = async (userId, userData) => {
 try {
 const updatedUser = await api.put(`/users/${userId}`, userData);
 setUsers((prevUsers) =>
 prevUsers.map((user) => (user.id === userId ? updatedUser : user))
);
 return updatedUser;
 } catch (error) {
 console.error('Failed to update user:', error);
 throw error;
 }
};

const deleteUser = async (userId) => {
 try {
 await api.remove(`/users/${userId}`);
 setUsers((prevUsers) => prevUsers.filter((user) => user.id !== userId));
 } catch (error) {
 console.error('Failed to delete user:', error);
 throw error;
 }
};

return (
 <div>
 <h2>User Management</h2>
 {api.loading && <div>Loading...</div>}
 {api.error && <div>Error: {api.error}</div>}
 </div>
)
```

```
{/* User creation form */}
<CreateUserForm onSubmit={createUser} />

{/* User list */}

 {users.map((user) => (
 <li key={user.id}>
 {user.name}
 <button
 onClick={() => updateUser(user.id, { ...user, verified: true })}>
 Verify
 </button>
 <button onClick={() => deleteUser(user.id)}>Delete</button>

))}

</div>
)
}
```

## Data Fetching with Axios

```
// Install Axios:
// npm install axios

import axios from 'axios';
import { useState, useEffect } from 'react';

// Basic Axios GET request
function ProductList() {
 const [products, setProducts] = useState([]);
 const [loading, setLoading] = useState(true);
 const [error, setError] = useState(null);

 useEffect(() => {
 // Create a cancel token source
 const source = axios.CancelToken.source();

 async function fetchProducts() {
 try {
 setLoading(true);

 const response = await axios.get('https://api.example.com/products', {
 cancelToken: source.token,
 });

 setProducts(response.data);
 setError(null);
 } catch (err) {
 setError(err);
 }
 }
 fetchProducts();
 }, []);

 return (
 <div>
 {loading ? <div>Loading...</div> : products.map((product) => (
 <div>
 {product.name}
 <button>
 View Details
 </button>
 </div>
))}
 </div>
);
}
export default ProductList;
```

```
 if (axios.isCancel(err)) {
 console.log('Request cancelled:', err.message);
 } else {
 setError(err.message);
 setProducts([]);
 }
 }

 fetchProducts();

 // Clean up function
 return () => {
 source.cancel('Component unmounted');
 };
}, []);

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

return (
 <div>
 <h2>Products</h2>

 {products.map((product) => (
 <li key={product.id}>
 {product.name} - ${product.price}

))}

 </div>
);
}

// Creating an Axios instance with default configuration
const apiClient = axios.create({
 baseURL: 'https://api.example.com',
 timeout: 10000,
 headers: {
 'Content-Type': 'application/json',
 Accept: 'application/json',
 },
});

// Add request interceptor (e.g., for adding auth tokens)
apiClient.interceptors.request.use(
 (config) => {
 // Get token from localStorage
 const token = localStorage.getItem('auth_token');

 // If token exists, add it to request headers
 if (token) {
```

```
 config.headers.Authorization = `Bearer ${token}`;
}

return config;
},
(error) => {
 return Promise.reject(error);
}
);

// Add response interceptor (e.g., for handling errors globally)
apiClient.interceptors.response.use(
 (response) => {
 // Any status code between 200 and 299 triggers this function
 return response;
},
(error) => {
 // Any status codes outside the range of 2xx trigger this function

 // Handle 401 Unauthorized
 if (error.response && error.response.status === 401) {
 // Clear user session
 localStorage.removeItem('auth_token');
 // Redirect to login
 window.location.href = '/login';
 }

 // Handle 404 Not Found
 if (error.response && error.response.status === 404) {
 console.error('Resource not found');
 }

 // Handle network errors
 if (error.request && !error.response) {
 console.error('Network error - no response received');
 }
}

return Promise.reject(error);
}
);

// Custom hook for Axios data fetching
function useAxios(url, options = {}) {
 const [data, setData] = useState(null);
 const [loading, setLoading] = useState(true);
 const [error, setError] = useState(null);

 useEffect(() => {
 const source = axios.CancelToken.source();

 const fetchData = async () => {
 try {
 setLoading(true);
 const response = await axios.get(url, options);
 setData(response.data);
 setError(null);
 } catch (err) {
 if (err.message === 'Cancelled') {
 setError(null);
 } else {
 setError(err);
 }
 }
 };

 fetchData();
 }, [url, options]);
}

export default useAxios;
```

```
const response = await apiClient({
 url,
 cancelToken: source.token,
 ...options,
});

setData(response.data);
setError(null);
} catch (err) {
 if (!axios.isCancel(err)) {
 setError(err.message);
 setData(null);
 }
} finally {
 setLoading(false);
}
};

fetchData();

return () => {
 source.cancel('Component unmounted');
};
}, [url, JSON.stringify(options)]);

return { data, loading, error };
}

// Using the custom Axios hook
function UserProfile({ userId }) {
 const { data, loading, error } = useAxios(`/users/${userId}`);

 if (loading) return <div>Loading user profile...</div>;
 if (error) return <div>Error loading profile: {error}</div>;
 if (!user) return <div>User not found</div>;

 return (
 <div className="user-profile">
 <h2>{user.name}</h2>
 <p>Email: {user.email}</p>
 <p>Role: {user.role}</p>
 <p>Member since: {new Date(user.createdAt).toLocaleDateString()}</p>
 </div>
);
}

// POST request with Axios
function CreatePostForm() {
 const [title, setTitle] = useState('');
 const [content, setContent] = useState('');
 const [loading, setLoading] = useState(false);
 const [error, setError] = useState(null);

 const handleSubmit = async (e) => {
```

```
e.preventDefault();

if (!title || !content) {
 setError('Please enter both title and content');
 return;
}

try {
 setLoading(true);
 setError(null);

 await apiClient.post('/posts', {
 title,
 content,
 createdAt: new Date().toISOString(),
 });

 // Reset form on success
 setTitle('');
 setContent('');

 alert('Post created successfully!');
} catch (err) {
 setError(err.response?.data?.message || err.message);
} finally {
 setLoading(false);
}
};

return (
 <form onSubmit={handleSubmit}>
 <h2>Create New Post</h2>

 {error && <div className="error">{error}</div>}

 <div>
 <label htmlFor="title">Title</label>
 <input
 id="title"
 value={title}
 onChange={(e) => setTitle(e.target.value)}
 required
 />
 </div>

 <div>
 <label htmlFor="content">Content</label>
 <textarea
 id="content"
 value={content}
 onChange={(e) => setContent(e.target.value)}
 rows="5"
 required
 />
 </div>
 </form>
);
```

```
</div>

<button type="submit" disabled={loading}>
 {loading ? 'Creating...' : 'Create Post'}
</button>
</form>
);
}
```

## React Query for Advanced Data Fetching

```
// Install React Query:
// npm install react-query

import {
 QueryClient,
 QueryClientProvider,
 useQuery,
 useMutation,
 useQueryClient,
} from 'react-query';
import axios from 'axios';

// Create a QueryClient instance
const queryClient = new QueryClient({
 defaultOptions: {
 queries: {
 refetchOnWindowFocus: false,
 retry: 1,
 staleTime: 30000, // 30 seconds
 },
 },
});

// Wrap your app with the QueryClientProvider
function App() {
 return (
 <QueryClientProvider client={queryClient}>
 <div className="app">
 <header>
 <h1>React Query Demo</h1>
 </header>
 <main>
 <UserList />
 <AddUserForm />
 </main>
 </div>
 </QueryClientProvider>
);
}
```

```
// API functions
const api = {
 getUsers: async () => {
 const { data } = await axios.get('https://api.example.com/users');
 return data;
 },
 getUserById: async (id) => {
 const { data } = await axios.get(`https://api.example.com/users/${id}`);
 return data;
 },
 createUser: async (userData) => {
 const { data } = await axios.post(
 'https://api.example.com/users',
 userData
);
 return data;
 },
 updateUser: async ({ id, ...userData }) => {
 const { data } = await axios.put(
 `https://api.example.com/users/${id}`,
 userData
);
 return data;
 },
 deleteUser: async (id) => {
 const { data } = await axios.delete(`https://api.example.com/users/${id}`);
 return data;
 },
};

// Component using useQuery for data fetching
function UserList() {
 const {
 data: users,
 isLoading,
 isError,
 error,
 refetch,
 } = useQuery('users', api.getUsers, {
 onError: (err) => {
 console.error('Error fetching users:', err);
 },
 });

 if (isLoading) return <div>Loading users...</div>

 if (isError) {
 return (
 <div className="error">
 Error: {error.message}
 <button onClick={refetch}>Retry</button>
 </div>
);
 }
}
```

```
return (
 <div>
 <h2>User List</h2>
 <button onClick={refetch}>Refresh</button>

 {users.map((user) => (
 <li key={user.id}>
 <UserItem user={user} />

))}

 </div>
);
}

// Using useMutation for data mutations
function AddUserForm() {
 const queryClient = useQueryClient();
 const [formData, setFormData] = useState({
 name: '',
 email: '',
 role: 'user',
 });

 // Create user mutation
 const mutation = useMutation(api.createUser, {
 onSuccess: () => {
 // Invalidate and refetch the users query
 queryClient.invalidateQueries('users');

 // Reset form
 setFormData({
 name: '',
 email: '',
 role: 'user',
 });

 alert('User created successfully!');
 },
 });

 const handleChange = (e) => {
 const { name, value } = e.target;
 setFormData((prev) => ({
 ...prev,
 [name]: value,
 }));
 };

 const handleSubmit = (e) => {
 e.preventDefault();
 mutation.mutate(formData);
 };
}
```

```
return (
 <div>
 <h2>Add New User</h2>

 {mutation.isError && (
 <div className="error">Error: {mutation.error.message}</div>
)}

 <form onSubmit={handleSubmit}>
 <div>
 <label htmlFor="name">Name</label>
 <input
 id="name"
 name="name"
 value={formData.name}
 onChange={handleChange}
 required
 />
 </div>

 <div>
 <label htmlFor="email">Email</label>
 <input
 id="email"
 name="email"
 type="email"
 value={formData.email}
 onChange={handleChange}
 required
 />
 </div>

 <div>
 <label htmlFor="role">Role</label>
 <select
 id="role"
 name="role"
 value={formData.role}
 onChange={handleChange}
 >
 <option value="user">User</option>
 <option value="admin">Admin</option>
 <option value="editor">Editor</option>
 </select>
 </div>

 <button type="submit" disabled={mutation.isLoading}>
 {mutation.isLoading ? 'Creating...' : 'Create User'}
 </button>
 </form>
 </div>
);

}
```

```
// Component with delete mutation
function UserItem({ user }) {
 const queryClient = useQueryClient();

 // Delete mutation
 const deleteMutation = useMutation(() => api.deleteUser(user.id), {
 onSuccess: () => {
 // Invalidate and refetch
 queryClient.invalidateQueries('users');
 },
 });

 const handleDelete = () => {
 if (window.confirm(`Are you sure you want to delete ${user.name}?`)) {
 deleteMutation.mutate();
 }
 };
}

return (
 <div className="user-item">
 <div>
 {user.name}
 <p>{user.email}</p>
 {user.role}
 </div>

 <div className="actions">
 <button
 onClick={handleDelete}
 disabled={deleteMutation.isLoading}
 className="delete-button"
 >
 {deleteMutation.isLoading ? 'Deleting...' : 'Delete'}
 </button>
 </div>
 </div>
);

}

// Query with dependent queries
function UserProfile({ userId }) {
 // Query for user details
 const userQuery = useQuery(['user', userId], () => api.getUserById(userId), {
 enabled: !!userId, // Only run if userId exists
 });

 // Dependent query - only run if userQuery succeeds
 const postsQuery = useQuery(
 ['userPosts', userId],
 () => api.getUserPosts(userId),
 {
 enabled: !!userQuery.data, // Only run if user data exists
 }
}
```

```
);

if (userQuery.isLoading) return <div>Loading user...</div>;
if (userQuery.isError) return <div>Error: {userQuery.error.message}</div>;

const user = userQuery.data;

return (
 <div className="user-profile">
 <h2>{user.name}</h2>
 <p>Email: {user.email}</p>
 <p>Role: {user.role}</p>

 <h3>User Posts</h3>
 {postsQuery.isLoading ? (
 <p>Loading posts...</p>
) : postsQuery.isError ? (
 <p>Error loading posts: {postsQuery.error.message}</p>
) : postsQuery.data.length === 0 ? (
 <p>No posts found</p>
) : (

 {postsQuery.data.map((post) => (
 <li key={post.id}>{post.title}
)))

)}
 </div>
);
}

// Infinite query for pagination
function InfinitePostList() {
 const {
 data,
 fetchNextPage,
 hasNextPage,
 isFetchingNextPage,
 isLoading,
 isError,
 error,
 } = useInfiniteQuery(
 'posts',
 async ({ pageParam = 1 }) => {
 const response = await axios.get(
 `https://api.example.com/posts?page=${pageParam}&limit=10`
);
 return response.data;
 },
 {
 getNextPageParam: (lastPage, allPages) => {
 // Return undefined when there are no more pages
 return lastPage.length === 10 ? allPages.length + 1 : undefined;
 },
 }
);
}
```

```
 }

);

// Handle intersection observer for infinite scroll
const observerRef = useRef();
const lastPostRef = useCallback(
 (node) => {
 if (isFetchingNextPage) return;

 if (observerRef.current) {
 observerRef.current.disconnect();
 }

 observerRef.current = new IntersectionObserver((entries) => {
 if (entries[0].isIntersecting && hasNextPage) {
 fetchNextPage();
 }
 });
 }
);

if (node) {
 observerRef.current.observe(node);
}

},
[isFetchingNextPage, fetchNextPage, hasNextPage]
);

if (isLoading) return <div>Loading posts...</div>;
if (isError) return <div>Error: {error.message}</div>;

return (
 <div className="post-list">
 <h2>All Posts</h2>

 {data.pages.map((group, i) => (
 <React.Fragment key={i}>
 {group.map((post, index) => {
 // Add ref to last item in the last group
 const isLastPost =
 i === data.pages.length - 1 && index === group.length - 1;

 return (
 <div
 key={post.id}
 ref={isLastPost ? lastPostRef : null}
 className="post-item"
 >
 <h3>{post.title}</h3>
 <p>{post.excerpt}</p>
 </div>
);
 })}
 </React.Fragment>
)));
}
```

```
{isFetchingNextPage && <div>Loading more...</div>}

{!hasNextPage && <div>No more posts to load</div>}
</div>
);
}
```

## Error Handling and Loading States

```
// Common patterns for handling loading states
function DataDisplay({ resourceName, fetchFn }) {
 const [state, setState] = useState({
 data: null,
 loading: true,
 error: null,
 });

 useEffect(() => {
 let isMounted = true;

 const fetchData = async () => {
 setState((prev) => ({ ...prev, loading: true, error: null }));

 try {
 const data = await fetchFn();

 if (isMounted) {
 setState({ data, loading: false, error: null });
 }
 } catch (error) {
 if (isMounted) {
 setState({ data: null, loading: false, error });
 }
 }
 };
 });

 fetchData();

 return () => {
 isMounted = false;
 };
}, [fetchFn]);

const { data, loading, error } = state;

if (loading) {
 return <LoadingIndicator />;
}

if (error) {
 return <ErrorDisplay error={error} resourceName={resourceName} />;
}
```

```
}

if (!data) {
 return <EmptyState resourceName={resourceName} />;
}

// Render actual content with data
return (
 <div className="data-display">
 {/* Children will use the data */}
 {React.Children.map(children, (child) =>
 React.cloneElement(child, { data })
)}
 </div>
);
}

// Reusable loading indicator
function LoadingIndicator({ size = 'medium', message = 'Loading...' }) {
 return (
 <div className={`loading-indicator loading-${size}`}>
 <div className="spinner"></div>
 <p>{message}</p>
 </div>
);
}

// Reusable error display
function ErrorDisplay({ error, resourceName = 'data', onRetry }) {
 // Determine friendly error message based on error
 const getErrorMessage = () => {
 if (error.message.includes('network')) {
 return 'Network error. Please check your connection and try again.';
 }

 if (error.response?.status === 404) {
 return `The ${resourceName} you're looking for could not be found.`;
 }

 if (error.response?.status === 403) {
 return 'You do not have permission to access this resource.';
 }

 if (error.response?.status === 401) {
 return 'Please log in to access this resource.';
 }

 return `Error loading ${resourceName}: ${error.message}`;
 };

 return (
 <div className="error-display">
 <div className="error-icon">⚠</div>
 <h3>Something went wrong</h3>
 </div>
);
}
```

```
<p>{getErrorMessage()}</p>

{onRetry && (
 <button onClick={onRetry} className="retry-button">
 Try Again
 </button>
)
</div>
);
}

// Empty state display
function EmptyState({ resourceName = 'items', message, actionBar }) {
 const defaultMessage = `No ${resourceName} found.`;

 return (
 <div className="empty-state">
 <div className="empty-icon">∅</div>
 <h3>{message || defaultMessage}</h3>
 {actionBar}
 </div>
);
}

// Skeleton loading example
function SkeletonLoader({ type = 'list', count = 3 }) {
 const renderSkeletonItem = () => {
 switch (type) {
 case 'userCard':
 return (
 <div className="skeleton-user-card">
 <div className="skeleton-circle"></div>
 <div className="skeleton-line skeleton-name"></div>
 <div className="skeleton-line skeleton-email"></div>
 <div className="skeleton-line skeleton-short"></div>
 </div>
);
 case 'article':
 return (
 <div className="skeleton-article">
 <div className="skeleton-line skeleton-title"></div>
 <div className="skeleton-line"></div>
 <div className="skeleton-line"></div>
 <div className="skeleton-line skeleton-short"></div>
 </div>
);
 case 'list':
 default:
 return (
 <div className="skeleton-list-item">
 <div className="skeleton-line"></div>
 </div>
);
 }
 };
}
```

```
);
 }
};

return (
 <div className={`skeleton-loader skeleton-${type}`}>
 {Array(count)
 .fill()
 .map((_, index) => (
 <div key={index} className="skeleton-item">
 {renderSkeletonItem()}
 </div>
)))
 </div>
);
}

// Progressive loading example
function ArticleList() {
 const [articles, setArticles] = useState([]);
 const [loadingMore, setLoadingMore] = useState(false);
 const [hasMore, setHasMore] = useState(true);
 const [page, setPage] = useState(1);

 const loadArticles = async (pageNum) => {
 try {
 setLoadingMore(true);

 const response = await fetch(`api/articles?page=${pageNum}&limit=10`);
 const data = await response.json();

 if (data.length === 0) {
 setHasMore(false);
 } else {
 setArticles((prev) => [...prev, ...data]);
 setPage(pageNum);
 }
 } catch (error) {
 console.error('Error loading articles:', error);
 } finally {
 setLoadingMore(false);
 }
 };

 useEffect(() => {
 loadArticles(1);
 }, []);

 const loadMore = () => {
 if (!loadingMore && hasMore) {
 loadArticles(page + 1);
 }
 };
}
```

```
return (
 <div className="article-list">
 <h2>Articles</h2>

 {articles.length === 0 && !loadingMore ? (
 <EmptyState
 resourceName="articles"
 actionButton={
 <button onClick={() => loadArticles(1)}>Refresh</button>
 }
 />
) : (
 <>
 <div className="articles">
 {articles.map((article) => (
 <ArticleCard key={article.id} article={article} />
)))
 </div>

 {loadingMore && <SkeletonLoader type="article" count={2} />}

 {hasMore && !loadingMore && (
 <button onClick={loadMore} className="load-more">
 Load More
 </button>
)}
 </>
)}
 </div>
);
}
```

## React Testing

### Conceptual Foundations

Testing in React applications ensures that your components work as expected and helps prevent bugs from being introduced as the codebase evolves. A comprehensive testing strategy includes different types of tests: unit tests, integration tests, and end-to-end tests.

**Mental Model:** Think of testing like a safety net for your code. Just as acrobats use safety nets to catch them if they fall, tests catch bugs before they reach production. Different types of tests serve different purposes, from testing small isolated pieces (unit tests) to testing how everything works together (integration and end-to-end tests).

### Common Misconceptions:

- 100% test coverage guarantees bug-free code (it doesn't)

- Testing slows down development (in the long run, it speeds it up)
- You need to test everything (focus on critical paths and edge cases)
- You can only write tests after implementing features (test-driven development writes tests first)

## Testing Tools and Setup

```
// Common testing tools for React:
// - Jest: JavaScript testing framework
// - React Testing Library: For testing React components
// - Enzyme: Alternative to React Testing Library (less recommended now)
// - Cypress: End-to-end testing framework

// Installation:
// npm install --save-dev jest @testing-library/react @testing-library/jest-dom
@testing-library/user-event

// Setting up Jest configuration in package.json
/*
"jest": {
 "testEnvironment": "jsdom",
 "setupFilesAfterEnv": [
 "<rootDir>/src/setupTests.js"
,
 "moduleNameMapper": {
 "\\\.(css|less|scss|sass)": "identity-obj-proxy",
 "\\\.
(jpg|jpeg|png|gif|eot|otf|webp|svg|ttf|woff|woff2|mp4|webm|wav|mp3|m4a|aac|oga)":
"<rootDir>/__mocks__/fileMock.js"
 }
}
*/

// Setup file (setupTests.js)
import '@testing-library/jest-dom';
// This adds custom jest matchers from jest-dom like toBeInTheDocument()

// Basic test structure
// Component.test.js or Component.spec.js
import React from 'react';
import { render, screen } from '@testing-library/react';
import userEvent from '@testing-library/user-event';
import Component from './Component';

describe('Component', () => {
 test('renders correctly', () => {
 render(<Component />);
 // Assertions...
 });

 test('handles click event', async () => {
 render(<Component />);
 // User interactions and assertions...
 });
});
```

```
});
});
```

## Unit Testing Components

```
// Testing a simple component
// Button.js
import React from 'react';

function Button({ onClick, disabled, children }) {
 return (
 <button onClick={onClick} disabled={disabled} className="button">
 {children}
 </button>
);
}

export default Button;

// Button.test.js
import React from 'react';
import { render, screen } from '@testing-library/react';
import userEvent from '@testing-library/user-event';
import Button from './Button';

describe('Button component', () => {
 test('renders with the correct text', () => {
 render(<Button>Click me</Button>);
 expect(screen.getByRole('button')).toHaveTextContent('Click me');
 });

 test('calls onClick handler when clicked', async () => {
 // Create a mock function
 const handleClick = jest.fn();

 render(<Button onClick={handleClick}>Click me</Button>);

 // Click the button
 await userEvent.click(screen.getByRole('button'));

 // Check if mock function was called
 expect(handleClick).toHaveBeenCalledTimes(1);
 });

 test('is disabled when disabled prop is true', () => {
 render(<Button disabled={true}>Click me</Button>);
 expect(screen.getByRole('button')).toBeDisabled();
 });
});

// Testing component with state
```

```
// Counter.js
import React, { useState } from 'react';

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

 const increment = () => setCount((prev) => prev + 1);
 const decrement = () => setCount((prev) => prev - 1);

 return (
 <div>
 <h2>Count: {count}</h2>
 <button onClick={increment}>Increment</button>
 <button onClick={decrement}>Decrement</button>
 </div>
);
}

// Counter.test.js
import React from 'react';
import { render, screen } from '@testing-library/react';
import userEvent from '@testing-library/user-event';
import Counter from './Counter';

describe('Counter component', () => {
 test('renders initial count of 0', () => {
 render(<Counter />);
 expect(screen.getByText('Count: 0')).toBeInTheDocument();
 });

 test('increments the count when increment button is clicked', async () => {
 render(<Counter />);

 await userEvent.click(screen.getByText('Increment'));

 expect(screen.getByText('Count: 1')).toBeInTheDocument();
 });

 test('decrements the count when decrement button is clicked', async () => {
 render(<Counter />);

 await userEvent.click(screen.getByText('Decrement'));

 expect(screen.getByText('Count: -1')).toBeInTheDocument();
 });

 test('handles multiple clicks correctly', async () => {
 render(<Counter />);

 const incrementBtn = screen.getByText('Increment');

 // Click increment three times
 await userEvent.click(incrementBtn);
 await userEvent.click(incrementBtn);
 });
}
```

```
await userEvent.click(incrementBtn);

expect(screen.getByText('Count: 3')).toBeInTheDocument();
});

});

// Testing component with props
// UserProfile.js
import React from 'react';

function UserProfile({ user, onEdit }) {
 if (!user) {
 return <p>No user data available</p>;
 }

 return (
 <div className="user-profile">
 <h2>{user.name}</h2>
 <p>Email: {user.email}</p>
 <p>Role: {user.role}</p>
 {onEdit && <button onClick={onEdit}>Edit Profile</button>}
 </div>
);
}

// UserProfile.test.js
import React from 'react';
import { render, screen } from '@testing-library/react';
import userEvent from '@testing-library/user-event';
import UserProfile from './UserProfile';

describe('UserProfile component', () => {
 const mockUser = {
 name: 'John Doe',
 email: 'john@example.com',
 role: 'Admin',
 };

 test('displays message when no user is provided', () => {
 render(<UserProfile />);
 expect(screen.getByText('No user data available')).toBeInTheDocument();
 });

 test('renders user information correctly', () => {
 render(<UserProfile user={mockUser} />);

 expect(screen.getByText('John Doe')).toBeInTheDocument();
 expect(screen.getByText('Email: john@example.com')).toBeInTheDocument();
 expect(screen.getByText('Role: Admin')).toBeInTheDocument();
 });

 test('calls onEdit when edit button is clicked', async () => {
 const handleEdit = jest.fn();
 render(<UserProfile user={mockUser} onEdit={handleEdit} />);

 await userEvent.click(screen.getByText('Edit Profile'));
 expect(handleEdit).toHaveBeenCalled();
 });
});
```

```
await userEvent.click(screen.getByText('Edit Profile'));

expect(handleEdit).toHaveBeenCalledTimes(1);
});

test('does not show edit button when onEdit is not provided', () => {
 render(<UserProfile user={mockUser} />);

 expect(screen.queryByText('Edit Profile')).not.toBeInTheDocument();
});
});
```

## Testing Forms and User Interactions

```
// LoginForm.js
import React, { useState } from 'react';

function LoginForm({ onSubmit }) {
 const [email, setEmail] = useState('');
 const [password, setPassword] = useState('');
 const [error, setError] = useState('');

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

 if (!email || !password) {
 setError('Please enter both email and password');
 return;
 }

 onSubmit({ email, password });
 };

 return (
 <form onSubmit={handleSubmit}>
 {error && <div className="error">{error}</div>}

 <div>
 <label htmlFor="email">Email</label>
 <input
 id="email"
 type="email"
 value={email}
 onChange={(e) => setEmail(e.target.value)}
 placeholder="Enter your email"
 />
 </div>

 <div>
```

```
<label htmlFor="password">Password</label>
<input
 id="password"
 type="password"
 value={password}
 onChange={(e) => setPassword(e.target.value)}
 placeholder="Enter your password"
/>
</div>

<button type="submit">Log In</button>
</form>
);
}

// LoginForm.test.js
import React from 'react';
import { render, screen, waitFor } from '@testing-library/react';
import userEvent from '@testing-library/user-event';
import LoginForm from './LoginForm';

describe('LoginForm component', () => {
 test('renders form elements correctly', () => {
 render(<LoginForm onSubmit={() => {}} />);

 expect(screen.getByLabelText(/email/i)).toBeInTheDocument();
 expect(screen.getByLabelText(/password/i)).toBeInTheDocument();
 expect(screen.getByRole('button', { name: /log in/i })).toBeInTheDocument();
 });

 test('displays error message when submitting empty form', async () => {
 render(<LoginForm onSubmit={() => {}} />);

 await userEvent.click(screen.getByRole('button', { name: /log in/i }));

 expect(
 screen.getText('Please enter both email and password')
).toBeInTheDocument();
 });

 test('submits the form with user input', async () => {
 const handleSubmit = jest.fn();
 render(<LoginForm onSubmit={handleSubmit} />);

 await userEvent.type(screen.getByLabelText(/email/i), 'test@example.com');
 await userEvent.type(screen.getByLabelText(/password/i), 'password123');

 await userEvent.click(screen.getByRole('button', { name: /log in/i }));

 expect(handleSubmit).toHaveBeenCalledWith({
 email: 'test@example.com',
 password: 'password123',
 });
 });
});
```

```
test('clears error message after entering values', async () => {
 render(<LoginForm onSubmit={() => {}} />);

 // Submit empty form to trigger error
 await userEvent.click(screen.getByRole('button', { name: /log in/i }));
 expect(
 screen.getText('Please enter both email and password')
).toBeInTheDocument();

 // Enter values and submit again
 await userEvent.type(screen.getByLabelText(/email/i), 'test@example.com');
 await userEvent.type(screen.getByLabelText(/password/i), 'password123');

 await userEvent.click(screen.getByRole('button', { name: /log in/i }));

 expect(
 screen.queryByText('Please enter both email and password')
).not.toBeInTheDocument();
});

// Testing complex form with validation
// RegistrationForm.js (partial example)
function RegistrationForm({ onSubmit }) {
 const [formData, setFormData] = useState({
 username: '',
 email: '',
 password: '',
 confirmPassword: '',
 });

 const [errors, setErrors] = useState({});
 const [isSubmitting, setIsSubmitting] = useState(false);

 const validate = () => {
 const newErrors = {};

 if (!formData.username.trim()) {
 newErrors.username = 'Username is required';
 }

 if (!formData.email.trim()) {
 newErrors.email = 'Email is required';
 } else if (!/\S+@\S+\.\S+/.test(formData.email)) {
 newErrors.email = 'Email is invalid';
 }

 if (!formData.password) {
 newErrors.password = 'Password is required';
 } else if (formData.password.length < 6) {
 newErrors.password = 'Password must be at least 6 characters';
 }
 }
}
```

```
if (formData.password !== formData.confirmPassword) {
 newErrors.confirmPassword = 'Passwords do not match';
}

setErrors(newErrors);
return Object.keys(newErrors).length === 0;
};

const handleChange = (e) => {
 const { name, value } = e.target;
 setFormData({ ...formData, [name]: value });
};

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

 const isValid = validate();
 if (!isValid) return;

 setIsSubmitting(true);

 try {
 await onSubmit(formData);
 // Reset form after successful submission
 setFormData({
 username: '',
 email: '',
 password: '',
 confirmPassword: '',
 });
 setErrors({});
 } catch (error) {
 setErrors({ submit: error.message });
 } finally {
 setIsSubmitting(false);
 }
};

return (
 <form onSubmit={handleSubmit}>
 {/* Form fields with validation errors */}
 {/* Submit button */}
 </form>
);
}

// RegistrationForm.test.js (partial example)
test('validates email format', async () => {
 render(<RegistrationForm onSubmit={() => {}} />);

 // Fill out form with invalid email
 await userEvent.type(screen.getByLabelText(/username/i), 'testuser');
 await userEvent.type(screen.getByLabelText(/email/i), 'invalid-email');
 await userEvent.type(screen.getByLabelText(/^password$/i), 'password123');
```

```
await userEvent.type(
 screen.getByLabelText(/confirm password/i),
 'password123'
);

await userEvent.click(screen.getByRole('button', { name: /register/i }));

expect(screen.getText('Email is invalid')).toBeInTheDocument();
});

// Testing file input
test('handles file upload', async () => {
 render(<FileUploadForm onUpload={mockUploadFn} />);

 const file = new File(['file content'], 'test.png', { type: 'image/png' });
 const fileInput = screen.getByLabelText(/upload file/i);

 // Mock file selection
 await userEvent.upload(fileInput, file);

 expect(fileInput.files[0]).toBe(file);
 expect(fileInput.files.length).toBe(1);

 await userEvent.click(screen.getByRole('button', { name: /upload/i }));

 expect(mockUploadFn).toHaveBeenCalledWith(file);
});
```

## Mocking API Calls and Dependencies

```
// UserList.js - component with API calls
import React, { useState, useEffect } from 'react';
import { fetchUsers, deleteUser } from './api';

function UserList() {
 const [users, setUsers] = useState([]);
 const [loading, setLoading] = useState(true);
 const [error, setError] = useState(null);

 useEffect(() => {
 const loadUsers = async () => {
 try {
 setLoading(true);
 const data = await fetchUsers();
 setUsers(data);
 setError(null);
 } catch (err) {
 setError('Failed to load users');
 setUsers([]);
 } finally {
 setLoading(false);
 }
 };
 loadUsers();
 }, []);
}

export default UserList;
```

```
 }
 });

 loadUsers();
}, []));

const handleDelete = async (userId) => {
 try {
 await deleteUser(userId);
 setUsers(users.filter((user) => user.id !== userId));
 } catch (err) {
 setError('Failed to delete user');
 }
};

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

return (
 <div>
 <h2>User List</h2>

 {users.map((user) => (
 <li key={user.id}>
 {user.name}
 <button onClick={() => handleDelete(user.id)}>Delete</button>

)))

 </div>
);
}

// api.js
export const fetchUsers = async () => {
 const response = await fetch('https://api.example.com/users');
 if (!response.ok) throw new Error('Failed to fetch users');
 return response.json();
};

export const deleteUser = async (userId) => {
 const response = await fetch(`https://api.example.com/users/${userId}`, {
 method: 'DELETE',
 });
 if (!response.ok) throw new Error('Failed to delete user');
 return true;
};

// UserList.test.js
import React from 'react';
import { render, screen, waitFor } from '@testing-library/react';
import userEvent from '@testing-library/user-event';
import UserList from './UserList';
import { fetchUsers, deleteUser } from './api';
```

```
// Mock the API module
jest.mock('./api');

describe('UserList Component', () => {
 const mockUsers = [
 { id: 1, name: 'John Doe' },
 { id: 2, name: 'Jane Smith' },
];

 beforeEach(() => {
 // Clear all mocks before each test
 jest.clearAllMocks();
 });

 test('renders loading state initially', () => {
 // Setup the mock to return a pending promise that never resolves
 fetchUsers.mockImplementation(() => new Promise(() => {}));

 render(<UserList />);

 expect(screen.getByText('Loading users...')).toBeInTheDocument();
 });

 test('renders users after successful fetch', async () => {
 // Setup mock to return resolved promise with data
 fetchUsers.mockResolvedValue(mockUsers);

 render(<UserList />);

 // Wait for loading to finish
 await waitFor(() => {
 expect(screen.queryByText('Loading users...')).not.toBeInTheDocument();
 });

 // Check if users are rendered
 expect(screen.getByText('John Doe')).toBeInTheDocument();
 expect(screen.getByText('Jane Smith')).toBeInTheDocument();
 });

 test('renders error message when fetch fails', async () => {
 // Setup mock to return rejected promise
 fetchUsers.mockRejectedValue(new Error('API error'));

 render(<UserList />);

 // Wait for loading to finish
 await waitFor(() => {
 expect(screen.queryByText('Loading users...')).not.toBeInTheDocument();
 });

 // Check if error message is rendered
 expect(screen.getByText('Error: Failed to load users')).toBeInTheDocument();
 });
});
```

```
test('removes user from list when delete is successful', async () => {
 // Setup mocks
 fetchUsers.mockResolvedValue(mockUsers);
 deleteUser.mockResolvedValue(true);

 render(<UserList />);

 // Wait for users to load
 await waitFor(() => {
 expect(screen.getByText('John Doe')).toBeInTheDocument();
 });

 // Find delete buttons
 const deleteButtons = screen.getAllByText('Delete');

 // Click the first delete button (for John Doe)
 await userEvent.click(deleteButtons[0]);

 // Check that deleteUser was called with the correct ID
 expect(deleteUser).toHaveBeenCalledWith(1);

 // Check that the user is removed from the list
 await waitFor(() => {
 expect(screen.queryByText('John Doe')).not.toBeInTheDocument();
 });

 // Jane Smith should still be in the document
 expect(screen.getByText('Jane Smith')).toBeInTheDocument();
});

test('shows error when delete fails', async () => {
 // Setup mocks
 fetchUsers.mockResolvedValue(mockUsers);
 deleteUser.mockRejectedValue(new Error('Delete failed'));

 render(<UserList />);

 // Wait for users to load
 await waitFor(() => {
 expect(screen.getByText('John Doe')).toBeInTheDocument();
 });

 // Find delete buttons
 const deleteButtons = screen.getAllByText('Delete');

 // Click the first delete button
 await userEvent.click(deleteButtons[0]);

 // Check that error message is shown
 expect(
 screen.getByText('Error: Failed to delete user')
).toBeInTheDocument();
```

```
// Both users should still be in the document
expect(screen.getByText('John Doe')).toBeInTheDocument();
expect(screen.getByText('Jane Smith')).toBeInTheDocument();
});

// Mocking fetch directly instead of module
test('renders users when fetch succeeds', async () => {
 // Mock global fetch
 global.fetch = jest.fn(() =>
 Promise.resolve({
 ok: true,
 json: () => Promise.resolve(mockUsers),
 })
);
 render(<UserList />);

 // Test assertions...

 // Restore original fetch
 global.fetch.mockRestore();
});

// Mocking third-party modules
import axios from 'axios';
jest.mock('axios');

test('fetches and displays data with axios', async () => {
 axios.get.mockResolvedValue({ data: mockUsers });

 render(<ComponentUsingAxios />);

 // Assertions...
});
```

## Testing Context and Custom Hooks

```
// AuthContext.js
import React, { createContext, useContext, useState } from 'react';

const AuthContext = createContext();

export const AuthProvider = ({ children }) => {
 const [user, setUser] = useState(null);

 const login = (userData) => {
 setUser(userData);
 };

 const logout = () => {
```

```
 setUser(null);
};

return (
 <AuthContext.Provider value={{ user, login, logout }}>
 {children}
 </AuthContext.Provider>
);
};

export const useAuth = () => {
 const context = useContext(AuthContext);
 if (!context) {
 throw new Error('useAuth must be used within anAuthProvider');
 }
 return context;
};

// ProfilePage.js
import React from 'react';
import { useAuth } from './AuthContext';

function ProfilePage() {
 const { user, logout } = useAuth();

 if (!user) {
 return <div>Please log in to view your profile</div>;
 }

 return (
 <div>
 <h1>Profile</h1>
 <p>Name: {user.name}</p>
 <p>Email: {user.email}</p>
 <button onClick={logout}>Log Out</button>
 </div>
);
}

// Testing components with context
// ProfilePage.test.js
import React from 'react';
import { render, screen } from '@testing-library/react';
import userEvent from '@testing-library/user-event';
import ProfilePage from './ProfilePage';
import { AuthProvider, useAuth } from './AuthContext';

// Mock the useAuth hook
jest.mock('./AuthContext', () => {
 const originalModule = jest.requireActual('./AuthContext');

 return {
 ...originalModule,
 useAuth: jest.fn(),
 };
});
```

```
};

});

describe('ProfilePage', () => {
 test('shows login message when user is not authenticated', () => {
 // Setup the mock to return null user
 useAuth.mockReturnValue({
 user: null,
 logout: jest.fn(),
 });

 render(<ProfilePage />);

 expect(
 screen.getByText('Please log in to view your profile')
).toBeInTheDocument();
 });

 test('displays user profile when authenticated', () => {
 // Setup the mock to return a user
 const mockLogout = jest.fn();
 useAuth.mockReturnValue({
 user: { name: 'John Doe', email: 'john@example.com' },
 logout: mockLogout,
 });

 render(<ProfilePage />);

 expect(screen.getByText('Profile')).toBeInTheDocument();
 expect(screen.getByText('Name: John Doe')).toBeInTheDocument();
 expect(screen.getByText('Email: john@example.com')).toBeInTheDocument();
 });

 test('logout button calls logout function', async () => {
 // Setup the mock
 const mockLogout = jest.fn();
 useAuth.mockReturnValue({
 user: { name: 'John Doe', email: 'john@example.com' },
 logout: mockLogout,
 });

 render(<ProfilePage />);

 // Click the logout button
 await userEvent.click(screen.getByText('Log Out'));

 // Check if logout was called
 expect(mockLogout).toHaveBeenCalledTimes(1);
 });
});

// Alternative: Test with wrapped component
// This approach doesn't mock the hook, but provides the context
describe('ProfilePage with AuthProvider', () => {
```

```
test('displays user profile when authenticated', () => {
 // Create a wrapper component that provides the context
 function Wrapper({ children }) {
 return (
 <AuthProvider
 initialValue={{
 user: { name: 'John Doe', email: 'john@example.com' },
 }}
 >
 {children}
 </AuthProvider>
);
 }

 // Render with custom wrapper
 render(<ProfilePage />, { wrapper: Wrapper });

 expect(screen.getByText('Name: John Doe')).toBeInTheDocument();
});
});

// Testing a custom hook
// useCounter.js
import { useState, useCallback } from 'react';

export function useCounter(initialValue = 0) {
 const [count, setCount] = useState(initialValue);

 const increment = useCallback(() => {
 setCount((prev) => prev + 1);
 }, []);

 const decrement = useCallback(() => {
 setCount((prev) => prev - 1);
 }, []);

 const reset = useCallback(() => {
 setCount(initialValue);
 }, [initialValue]);

 return { count, increment, decrement, reset };
}

// useCounter.test.js
import { renderHook, act } from '@testing-library/react-hooks';
import { useCounter } from './useCounter';

describe('useCounter hook', () => {
 test('should initialize with default value', () => {
 const { result } = renderHook(() => useCounter());

 expect(result.current.count).toBe(0);
 });
}
```

```
test('should initialize with custom value', () => {
 const { result } = renderHook(() => useCounter(10));

 expect(result.current.count).toBe(10);
});

test('should increment counter', () => {
 const { result } = renderHook(() => useCounter());

 act(() => {
 result.current.increment();
 });

 expect(result.current.count).toBe(1);
});

test('should decrement counter', () => {
 const { result } = renderHook(() => useCounter());

 act(() => {
 result.current.decrement();
 });

 expect(result.current.count).toBe(-1);
});

test('should reset counter to initial value', () => {
 const { result } = renderHook(() => useCounter(5));

 act(() => {
 result.current.increment();
 result.current.increment();
 result.current.reset();
 });

 expect(result.current.count).toBe(5);
});

test('should update when initialValue changes', () => {
 const { result, rerender } = renderHook(
 ({ initialValue }) => useCounter(initialValue),
 {
 initialProps: { initialValue: 0 },
 }
);

 // Rerender with new props
 rerender({ initialValue: 10 });

 act(() => {
 result.current.reset();
 });

 expect(result.current.count).toBe(10);
});
```

```
});
});
```

## Snapshot Testing

```
// Snapshot testing captures a component's output and compares it to a saved
version
// This helps detect unexpected changes in the UI

// Install jest-snapshot
// npm install --save-dev jest-snapshot

// Card.js
import React from 'react';

function Card({ title, description, imageUrl }) {
 return (
 <div className="card">
 {imageUrl && }
 <div className="card-content">
 <h3 className="card-title">{title}</h3>
 <p className="card-description">{description}</p>
 </div>
 </div>
);
}

// Card.test.js
import React from 'react';
import { render } from '@testing-library/react';
import Card from './Card';

describe('Card component', () => {
 it('renders correctly with all props', () => {
 const { container } = render(
 <Card
 title="Example Card"
 description="This is an example card description."
 imageUrl="https://example.com/image.jpg"
 />
);

 // This creates/updates a snapshot file
 expect(container).toMatchSnapshot();
 });

 it('renders correctly without image', () => {
 const { container } = render(
 <Card
 title="Example Card"
 description="This is an example card description."
 />
);
 });
});
```

```
 />
);

expect(container).toMatchSnapshot();
});

});

// Running tests with snapshots:
// - First run creates the snapshots
// - Subsequent runs compare with saved snapshots
// - Update snapshots with: jest --updateSnapshot or jest -u

// Inline snapshots
it('renders correctly with inline snapshot', () => {
 const { container } = render(
 <Card title="Test Card" description="Test description" />
);

 // Creates an inline snapshot in the test file itself
 expect(container).toMatchSnapshot();
});
```

## Integration Testing

```
// Integration tests verify that multiple components work together correctly

// TodoApp.js - Multiple components working together
import React, { useState } from 'react';

function TodoForm({ addTodo }) {
 const [text, setText] = useState('');

 const handleSubmit = (e) => {
 e.preventDefault();
 if (!text.trim()) return;

 addTodo(text);
 setText('');
 };

 return (
 <form onSubmit={handleSubmit} data-testid="todo-form">
 <input
 type="text"
 value={text}
 onChange={(e) => setText(e.target.value)}
 placeholder="Add a new todo"
 data-testid="todo-input"
 />
 <button type="submit" data-testid="add-button">
 Add

```

```
 </button>
 </form>
);
}

function TodoList({ todos, toggleTodo, deleteTodo }) {
 return (
 <ul data-testid="todo-list">
 {todos.map((todo) => (
 <li key={todo.id} data-testid={`todo-item-${todo.id}`}>
 <input
 type="checkbox"
 checked={todo.completed}
 onChange={() => toggleTodo(todo.id)}
 data-testid={`todo-checkbox-${todo.id}`}
 />
 <span
 style={{ textDecoration: todo.completed ? 'line-through' : 'none' }}
 data-testid={`todo-text-${todo.id}`}
 >
 {todo.text}

 <button
 onClick={() => deleteTodo(todo.id)}
 data-testid={`todo-delete-${todo.id}`}
 >
 Delete
 </button>

)));

);
}

function TodoApp() {
 const [todos, setTodos] = useState([]);

 const addTodo = (text) => {
 const newTodo = {
 id: Date.now(),
 text,
 completed: false,
 };
 setTodos([...todos, newTodo]);
 };

 const toggleTodo = (id) => {
 setTodos(
 todos.map((todo) =>
 todo.id === id ? { ...todo, completed: !todo.completed } : todo
)
);
 };
}
```

```
const deleteTodo = (id) => {
 setTodos(todos.filter((todo) => todo.id !== id));
};

return (
 <div className="todo-app">
 <h1>Todo App</h1>
 <TodoForm addTodo={addTodo} />
 <TodoList todos={todos} toggleTodo={toggleTodo} deleteTodo={deleteTodo} />
 </div>
);
}

// TodoApp.test.js - Integration test
import React from 'react';
import { render, screen, waitFor } from '@testing-library/react';
import userEvent from '@testing-library/user-event';
import TodoApp from './TodoApp';

describe('TodoApp Integration', () => {
 test('user can add a todo', async () => {
 render(<TodoApp />);

 // Find input and add button
 const input = screen.getByTestId('todo-input');
 const addButton = screen.getByTestId('add-button');

 // Add a new todo
 await userEvent.type(input, 'Buy groceries');
 await userEvent.click(addButton);

 // Check if todo was added to the list
 const todoList = screen.getByTestId('todo-list');
 expect(todoList).toHaveTextContent('Buy groceries');

 // Get the added todo item (we can't know the exact ID)
 const todoItem = screen.getText('Buy groceries').closest('li');
 expect(todoItem).toBeInTheDocument();
 });

 test('user can complete and delete a todo', async () => {
 render(<TodoApp />);

 // Add a todo first
 const input = screen.getByTestId('todo-input');
 const addButton = screen.getByTestId('add-button');

 await userEvent.type(input, 'Clean house');
 await userEvent.click(addButton);

 // Find the new todo item
 const todoText = screen.getText('Clean house');
 const todoItem = todoText.closest('li');
 const todoId = todoItem.dataset-testid.replace('todo-item-', '');
 });
});
```

```
// Find the checkbox and check it
const checkbox = screen.getByTestId(`todo-checkbox-${todoId}`);
await userEvent.click(checkbox);

// Check if todo text is struck through
expect(todoText).toHaveStyle('text-decoration: line-through');

// Now delete the todo
const deleteButton = screen.getByTestId(`todo-delete-${todoId}`);
await userEvent.click(deleteButton);

// Check if todo was removed
expect(todoText).not.toBeInTheDocument();
});

test('full workflow - add, complete, and delete todos', async () => {
 render(<TodoApp />);

 // Initially, the list should be empty
 const todoList = screen.getByTestId('todo-list');
 expect(todoList.children.length).toBe(0);

 // Add first todo
 await userEvent.type(screen.getByTestId('todo-input'), 'Task 1');
 await userEvent.click(screen.getByTestId('add-button'));

 // Add second todo
 await userEvent.type(screen.getByTestId('todo-input'), 'Task 2');
 await userEvent.click(screen.getByTestId('add-button'));

 // Check that both todos are in the list
 expect(todoList.children.length).toBe(2);
 expect(screen.getByText('Task 1')).toBeInTheDocument();
 expect(screen.getByText('Task 2')).toBeInTheDocument();

 // Mark the first todo as completed
 const firstTodoText = screen.getByText('Task 1');
 const firstTodoItem = firstTodoText.closest('li');
 const firstTodoId = firstTodoItem.dataset.testid.replace('todo-item-', '');

 await userEvent.click(screen.getByTestId(`todo-checkbox-${firstTodoId}`));

 // Verify it's completed (has line-through)
 expect(firstTodoText).toHaveStyle('text-decoration: line-through');

 // Delete the second todo
 const secondTodoText = screen.getByText('Task 2');
 const secondTodoItem = secondTodoText.closest('li');
 const secondTodoId = secondTodoItem.dataset.testid.replace(
 'todo-item-',
 ''
);
});
```

```

 await userEvent.click(screen.getByTestId(`todo-delete-${secondTodoId}`));

 // Verify only first todo remains
 expect(todoList.children.length).toBe(1);
 expect(screen.getByText('Task 1')).toBeInTheDocument();
 expect(screen.queryByText('Task 2')).not.toBeInTheDocument();
 });
});

// Testing a component with context and routing
import { MemoryRouter, Routes, Route } from 'react-router-dom';

function renderWithRouter(ui, { route = '/', ...renderOptions } = {}) {
 return render(
 <MemoryRouter initialEntries={[route]}>{ui}</MemoryRouter>,
 renderOptions
);
}

test('navigates to product page when product is clicked', async () => {
 renderWithRouter(
 <Routes>
 <Route path="/" element={<ProductList />} />
 <Route path="/products/:id" element={<ProductDetail />} />
 </Routes>
);

 // Find a product and click it
 await userEvent.click(screen.getByText('Product 1'));

 // Check that we navigated to the product detail page
 expect(screen.getByText('Product Details')).toBeInTheDocument();
});

```

## Test Coverage and Best Practices

```

// Running tests with coverage
// package.json script:
// "test:coverage": "jest --coverage"

// Best practices:
// 1. Test behavior, not implementation
// Bad: Testing implementation details
test('calls fetch API correctly', () => {
 const spy = jest.spyOn(global, 'fetch');
 render(<MyComponent />);
 expect(spy).toHaveBeenCalledWith('/api/data');
});

// Good: Testing behavior/output
test('displays loading and then data', async () => {

```

```
render(<MyComponent />);
expect(screen.getByText('Loading...')).toBeInTheDocument();

await waitFor(() => {
 expect(screen.getByText('Data loaded')).toBeInTheDocument();
});
});

// 2. Use screen queries appropriately
// getBy* - element must be in the DOM or test fails
// queryBy* - returns null if element not found (good for testing absence)
// findBy* - returns a promise, good for async rendering

// 3. Prefer user-centric queries
// Good - how users would find elements
screen.getByRole('button', { name: /submit/i });
screen.getByLabelText('Email');
screen.getByPlaceholderText('Enter your name');
screen.getText('Welcome');

// Avoid - too implementation-specific
screen.getByTestId('submit-button');

// 4. Use userEvent over fireEvent
// userEvent simulates real user interactions better
// fireEvent only fires the specified event

// Good:
await userEvent.type(input, 'hello');

// Avoid:
fireEvent.change(input, { target: { value: 'hello' } });

// 5. Test common edge cases
test('handles empty data', () => {
 mockFn.mockResolvedValue([]);
 render(<MyList />);

 expect(screen.getByText('No items found')).toBeInTheDocument();
});

test('handles error state', async () => {
 mockFn.mockRejectedValue(new Error('Failed'));
 render(<MyList />);

 await waitFor(() => {
 expect(screen.getByText(/error/i)).toBeInTheDocument();
 });
});

// 6. Prefer small, focused tests
// Instead of one large test, write multiple small tests

// 7. Test accessibility
```

```
test('form is accessible', () => {
 render(<LoginForm />);

 // Check that inputs have associated labels
 expect(screen.getByLabelText('Email')).toBeInTheDocument();
 expect(screen.getByLabelText('Password')).toBeInTheDocument();

 // Check ARIA attributes
 const submitButton = screen.getByRole('button', { name: /login/i });
 expect(submitButton).toHaveAttribute('aria-busy', 'false');
});

// 8. Set up global test utilities
// setupTests.js - extend matchers
import '@testing-library/jest-dom';

// Add custom matcher
expect.extend({
 toBeWithinRange(received, floor, ceiling) {
 const pass = received >= floor && received <= ceiling;
 if (pass) {
 return {
 message: () =>
 `expected ${received} not to be within range ${floor} - ${ceiling}`,
 pass: true,
 };
 } else {
 return {
 message: () =>
 `expected ${received} to be within range ${floor} - ${ceiling}`,
 pass: false,
 };
 }
 },
});

// Create test utils file
// testUtils.js
function renderWithProviders(ui, options = {}) {
 const {
 initialState = {},
 store = configureStore({ reducer, initialState }),
 ...renderOptions
 } = options;

 function Wrapper({ children }) {
 return (
 <Provider store={store}>
 <ThemeProvider theme="light">
 <MemoryRouter>{children}</MemoryRouter>
 </ThemeProvider>
 </Provider>
);
 }
}
```

```
 return render(ui, { wrapper: Wrapper, ...renderOptions });
}

// Use in tests
test('component with providers', () => {
 renderWithProviders(<MyComponent />);
 // Test assertions...
});
```

## Performance Optimization in React

### Conceptual Foundations

Performance optimization in React involves minimizing unnecessary renders, reducing bundle size, and optimizing resource usage to create a fast, responsive user experience. React provides several tools and patterns to help identify and fix performance bottlenecks.

**Mental Model:** Think of performance optimization like fine-tuning a race car. The car might work without tuning, but optimizations make it faster and more efficient. Similarly, React apps work without optimization, but strategic improvements can significantly enhance the user experience.

### Common Misconceptions:

- Premature optimization is good (it often complicates code without measurable benefits)
- Every component needs to be optimized (focus on the ones causing actual problems)
- Optimization is all-or-nothing (even small, targeted improvements help)
- Virtual DOM is always fast (it still has costs, especially with complex state)

### Identifying Performance Issues

```
// Using React Developer Tools Profiler
// Install React Developer Tools extension for Chrome or Firefox
// Then use the Profiler tab to record and analyze component renders

// Using the performance API to measure
function MeasuredComponent() {
 useEffect(() => {
 // Start measuring
 performance.mark('component-start');

 // Do something expensive
 const expensiveCalculation = () => {
 let result = 0;
 for (let i = 0; i < 1000000; i++) {
 result += i;
 }
 return result;
 };
 });
}
```

```
expensiveCalculation();

// End measuring
performance.mark('component-end');
performance.measure(
 'Component Render Time',
 'component-start',
 'component-end'
);

const measurements = performance.getEntriesByType('measure');
console.log('Render time:', measurements[0].duration, 'ms');

// Clean up marks and measures
performance.clearMarks();
performance.clearMeasures();
}, []);

return <div>Measured Component</div>;
}

// Using why-did-you-render library
// npm install @welldone-software/why-did-you-render
// In src/wdjr.js:
import React from 'react';

if (process.env.NODE_ENV === 'development') {
 const whyDidYouRender = require('@welldone-software/why-did-you-render');
 whyDidYouRender(React, {
 trackAllPureComponents: true,
 });
}

// In your component file:
function ExampleComponent(props) {
 // Component implementation
}

ExampleComponent.whyDidYouRender = true;

// Using React.Profiler component
function ProfilingExample() {
 const onRender = (
 id, // the "id" prop of the Profiler tree
 phase, // "mount" (first render) or "update" (re-render)
 actualDuration, // time spent rendering the committed update
 baseDuration, // estimated time to render the entire subtree without
 memoization
 startTime, // when React began rendering this update
 commitTime // when React committed this update
) => {
 console.log({
 id,
 phase,
```

```

 actualDuration,
 baseDuration,
 startTime,
 commitTime,
 });
};

return (
 <React.Profiler id="MyComponent" onRender={onRender}>
 <MyComponent />
 </React.Profiler>
);
}

// Custom hook for measuring render time
function useRenderTime(componentName) {
 const renderCount = useRef(0);

 useEffect(() => {
 renderCount.current += 1;

 const startTime = performance.now();

 return () => {
 const endTime = performance.now();
 const renderTime = endTime - startTime;
 console.log(
 `${componentName} rendered ${renderCount.current} times. ` +
 `Last render took ${renderTime.toFixed(2)}ms`
);
 };
 });
}

function ComponentWithRenderTimeLogging() {
 useRenderTime('ComponentWithRenderTimeLogging');

 // Component implementation

 return <div>Component content</div>;
}

```

## Component Memoization

```

// Using React.memo to prevent unnecessary renders
import React, { memo } from 'react';

// Regular component - will re-render whenever parent re-renders
function RegularComponent({ name }) {
 console.log('RegularComponent rendered');
 return <div>Hello, {name}</div>;
}

```

```
}

// Memoized component - only re-renders if props change
const MemoizedComponent = memo(function MemoizedComponent({ name }) {
 console.log('MemoizedComponent rendered');
 return <div>Hello, {name}</div>;
});

// Parent component to demonstrate
function ParentComponent() {
 const [count, setCount] = useState(0);
 const [name, setName] = useState('World');

 return (
 <div>
 <button onClick={() => setCount((c) => c + 1)}>
 Increment count: {count}
 </button>
 <button onClick={() => setName(name === 'World' ? 'React' : 'World')}>
 Toggle name
 </button>

 <RegularComponent name={name} />
 <MemoizedComponent name={name} />
 </div>
);
}

// With custom comparison function
const MemoizedWithCustomComparison = memo(
 function MemoizedWithCustomComparison({ user }) {
 console.log('MemoizedWithCustomComparison rendered');
 return <div>Hello, {user.name}</div>;
 },
 (prevProps, nextProps) => {
 // Return true if the props are "equal" (no re-render needed)
 // Return false if they're not equal (re-render needed)
 return prevProps.user.id === nextProps.user.id;
 }
);

// Common issues with memo
function BadMemoExample({ onClick }) {
 return (
 <MemoizedComponent
 // This inline object is a NEW object on every render
 user={{ name: 'John', age: 30 }}
 // This inline function is a NEW function on every render
 onAction={() => console.log('Action triggered')}
 // This prop is fine - primitive value
 count={5}
 // This is a prop passed from parent - may cause issues if parent
 // creates a new function on each render
 onClick={onClick}
 >
);
}
```

```
 />
);
}

// Solution: useMemo and useCallback
function GoodMemoExample() {
 // Memoize the object
 const user = useMemo(() => ({ name: 'John', age: 30 }), []);

 // Memoize the function
 const handleAction = useCallback(() => {
 console.log('Action triggered');
 }, []);

 return <MemoizedComponent user={user} onAction={handleAction} count={5} />;
}

// Optimizing components with children
const MemoizedContainer = memo(function MemoizedContainer({ children }) {
 console.log('Container rendered');
 return <div className="container">{children}</div>;
});

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

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

 {/* Will re-render when count changes because children is recreated */}
 <MemoizedContainer>
 <p>This content causes re-renders: {count}</p>
 </MemoizedContainer>

 {/* Solution: memoize the children too */}
 <MemoizedContainer>
 {useMemo(
 () => (
 <p>This content is stable despite parent changes</p>
),
 []
)}
 </MemoizedContainer>
 </div>
);
}
```

## useMemo and useCallback

```
import React, { useState, useMemo, useCallback } from 'react';

// Using useMemo for expensive calculations
function ExpensiveCalculation({ numbers }) {
 // This will only recalculate when numbers changes
 const sum = useMemo(() => {
 console.log('Calculating sum...');
 return numbers.reduce((acc, num) => acc + num, 0);
 }, [numbers]);

 return <div>Sum: {sum}</div>;
}

// Using useMemo for derived values
function UsersList({ users, searchTerm }) {
 // Memoize the filtered list to prevent recalculation on unrelated state changes
 const filteredUsers = useMemo(() => {
 console.log('Filtering users...');
 return users.filter((user) =>
 user.name.toLowerCase().includes(searchTerm.toLowerCase())
);
 }, [users, searchTerm]);

 return (

 {filteredUsers.map((user) => (
 <li key={user.id}>{user.name}
))}

);
}

// Using useCallback for event handlers
function SearchForm({ onSearch }) {
 const [query, setQuery] = useState('');

 // Without useCallback - creates a new function on every render
 const handleSearchWithoutCallback = () => {
 onSearch(query);
 };

 // With useCallback - stable function reference unless dependencies change
 const handleSearchWithCallback = useCallback(() => {
 onSearch(query);
 }, [onSearch, query]);

 return (
 <div>
 <input
 type="text"
 value={query}
 onChange={(e) => setQuery(e.target.value)}
 />
);
}
```

```
 <button onClick={handleSearchWithCallback}>Search</button>
 </div>
);
}

// Using useCallback with custom hooks
function useDebounce(value, delay) {
 const [debouncedValue, setDebouncedValue] = useState(value);

 useEffect(() => {
 const handler = setTimeout(() => {
 setDebouncedValue(value);
 }, delay);
 });

 return () => {
 clearTimeout(handler);
 };
}, [value, delay]);

return debouncedValue;
}

function SearchComponent({ onSearch }) {
 const [query, setQuery] = useState('');

 // Debounce the search value
 const debouncedQuery = useDebounce(query, 500);

 // Only re-create this function when debouncedQuery changes
 const search = useCallback(() => {
 onSearch(debouncedQuery);
 }, [onSearch, debouncedQuery]);

 // Run search when debounced query changes
 useEffect(() => {
 if (debouncedQuery) {
 search();
 }
 }, [debouncedQuery, search]);

 return (
 <input
 type="text"
 value={query}
 onChange={(e) => setQuery(e.target.value)}
 placeholder="Search..." />
);
}

// Common pitfalls and solutions
function UserProfile({ userId, onSave }) {
 const [user, setUser] = useState(null);
```

```
// Pitfall: unnecessary useCallback with no dependencies
// This doesn't actually help performance
const badlyMemoizedFunction = useCallback(() => {
 console.log('This still gets a new reference on every render');
}, []); // No dependencies

// Pitfall: forgetting dependencies
const forgottenDependency = useCallback(() => {
 // This uses userId but it's not in the dependency array!
 console.log(`User ${userId} details`);
}, []); // userId is missing!

// Correct usage with all dependencies
const properlyMemoizedFunction = useCallback(() => {
 console.log(`User ${userId} details`);
}, [userId]);

// Pitfall: memoizing everything unnecessarily
// Don't do this for simple values
const simpleValue = useMemo(() => 42, []); // Overkill for a constant

// Correct: only memoize expensive calculations
const expensiveCalculation = useMemo(() => {
 return user ? calculateUserScore(user) : 0;
}, [user]);

return <div>{/* Component content */}</div>;
}

// Using memo, useMemo, and useCallback together
const MemoizedChildComponent = memo(function ChildComponent({
 data,
 onAction,
}) {
 console.log('Child rendered');
 return (
 <div>
 <h3>{data.title}</h3>
 <button onClick={onAction}>Perform Action</button>
 </div>
);
});

function ParentComponent() {
 const [count, setCount] = useState(0);
 const [title, setTitle] = useState('Hello World');

 // Stable reference for object prop
 const data = useMemo(
 () => ({
 title,
 description: 'This is a description',
 }),
 [title]
}
```

```
);

// Stable reference for function prop
const handleAction = useCallback(() => {
 console.log('Action performed');
}, []);

return (
 <div>
 <button onClick={() => setCount((c) => c + 1)}>Increment: {count}</button>
 <button
 onClick={() =>
 setTitle((t) => (t === 'Hello World' ? 'Hi there' : 'Hello World'))
 }
 >
 Toggle Title
 </button>

 <MemoizedChildComponent data={data} onAction={handleAction} />
 </div>
);
}
```

## State Management Optimization

```
// Splitting state to prevent unnecessary renders
function UserProfileWithBadState() {
 // Problem: Any change to any property re-renders entire component
 const [userData, setUserData] = useState({
 name: 'John',
 email: 'john@example.com',
 preferences: {
 darkMode: false,
 notifications: true,
 language: 'en',
 },
 posts: [],
 friends: [],
 });

 // Updating one field updates the entire state
 const toggleDarkMode = () => {
 setUserData({
 ...userData,
 preferences: {
 ...userData.preferences,
 darkMode: !userData.preferences.darkMode,
 },
 });
 };
}
```

```
return (
 <div>
 <h1>{userData.name}</h1>
 <Preferences
 preferences={userData.preferences}
 onChange={toggleDarkMode}
 />
 <Posts posts={userData.posts} />
 <Friends friends={userData.friends} />
 </div>
);
}

// Better: Split state by concern
function UserProfileWithSplitState() {
 // Split state into logical pieces
 const [user, setUser] = useState({ name: 'John', email: 'john@example.com' });
 const [preferences, setPreferences] = useState({
 darkMode: false,
 notifications: true,
 language: 'en',
 });
 const [posts, setPosts] = useState([]);
 const [friends, setFriends] = useState([]);

 // Only updates the relevant piece of state
 const toggleDarkMode = () => {
 setPreferences((prev) => ({
 ...prev,
 darkMode: !prev.darkMode,
 }));
 };

 return (
 <div>
 <h1>{user.name}</h1>
 <Preferences preferences={preferences} onChange={toggleDarkMode} />
 <Posts posts={posts} />
 <Friends friends={friends} />
 </div>
);
}

// Using React context efficiently
// Problem: All consumers re-render when any part of context changes
const BadUserContext = createContext();

function BadUserProvider({ children }) {
 const [user, setUser] = useState({ name: 'John', isLoggedIn: true });
 const [preferences, setPreferences] = useState({ darkMode: false });

 // Everything is in one context value
 const value = {
 user,
```

```
preferences,
setUser,
setPreferences,
};

return (
<BadUserContext.Provider value={value}>{children}</BadUserContext.Provider>
);
}

// Better: Split context by concern
const UserContext = createContext();
const PreferencesContext = createContext();

function OptimizedProvider({ children }) {
 const [user, setUser] = useState({ name: 'John', isLoggedIn: true });
 const [preferences, setPreferences] = useState({ darkMode: false });

 const userValue = useMemo(() => ({ user, setUser }), [user]);
 const preferencesValue = useMemo(
 () => ({ preferences, setPreferences }),
 [preferences]
);

 return (
 <UserContext.Provider value={userValue}>
 <PreferencesContext.Provider value={preferencesValue}>
 {children}
 </PreferencesContext.Provider>
 </UserContext.Provider>
);
}

// Components only re-render when relevant context changes
function Profile() {
 const { user } = useContext(UserContext);
 return <h1>Hello, {user.name}</h1>;
}

function ThemeToggle() {
 const { preferences, setPreferences } = useContext(PreferencesContext);

 return (
 <button
 onClick={() =>
 setPreferences((p) => ({
 ...p,
 darkMode: !p.darkMode,
 }))
 }
 >
 {preferences.darkMode ? 'Light Mode' : 'Dark Mode'}
 </button>
);
}
```

```
}

// Using useReducer for complex state
function complexReducer(state, action) {
 switch (action.type) {
 case 'UPDATE_USER':
 return {
 ...state,
 user: {
 ...state.user,
 ...action.payload,
 },
 };
 case 'TOGGLE_THEME':
 return {
 ...state,
 preferences: {
 ...state.preferences,
 darkMode: !state.preferences.darkMode,
 },
 };
 // Other actions...
 default:
 return state;
 }
}

function AppWithReducer() {
 const [state, dispatch] = useReducer(complexReducer, {
 user: { name: 'John', isLoggedIn: true },
 preferences: { darkMode: false },
 posts: [],
 friends: [],
 });

 // Memoize child components or use React.memo
 const userSection = useMemo(
 () => (
 <UserSection
 user={state.user}
 onUpdate={(userData) =>
 dispatch({
 type: 'UPDATE_USER',
 payload: userData,
 })
 }
 />
),
 [state.user]
);

 const themeSection = useMemo(
 () => (
 <ThemeSection

```

```
 darkMode={state.preferences.darkMode}
 onToggle={() => dispatch({ type: 'TOGGLE_THEME' })}}
 />
),
 [state.preferences.darkMode]
);

return (
 <div>
 {userSection}
 {themeSection}
 {/* Other sections */}
 </div>
);
}

// Using custom hooks to encapsulate and optimize state logic
function useUser() {
 const [user, setUser] = useState({ name: 'John', isLoggedIn: true });

 const updateUser = useCallback((updates) => {
 setUser((prev) => ({ ...prev, ...updates }));
 }, []);

 const logout = useCallback(() => {
 setUser((prev) => ({ ...prev, isLoggedIn: false }));
 }, []);

 return {
 user,
 updateUser,
 logout,
 };
}

function usePreferences() {
 const [preferences, setPreferences] = useState({ darkMode: false });

 const toggleDarkMode = useCallback(() => {
 setPreferences((prev) => ({
 ...prev,
 darkMode: !prev.darkMode,
 }));
 }, []);

 return {
 preferences,
 toggleDarkMode,
 };
}

function OptimizedApp() {
 const { user, updateUser, logout } = useUser();
 const { preferences, toggleDarkMode } = usePreferences();
```

```
return (
 <div>
 <UserProfile user={user} onUpdate={updateUser} onLogout={logout} />
 <ThemeToggle darkMode={preferences.darkMode} onToggle={toggleDarkMode} />
 </div>
);
}
```

## List and Large Data Set Optimization

```
// Virtualizing long lists
// npm install react-window

import { FixedSizeList } from 'react-window';

function VirtualizedList({ items }) {
 // Render only the visible items for better performance
 const Row = ({ index, style }) => (
 <div style={style}>Item {items[index].name}</div>
);

 return (
 <FixedSizeList
 height={400}
 width={300}
 itemCount={items.length}
 itemSize={35}
 >
 {Row}
 </FixedSizeList>
);
}

// With infinite loading
// npm install react-window-infinite-loader

import { FixedSizeList } from 'react-window';
import InfiniteLoader from 'react-window-infinite-loader';

function InfiniteList({
 items,
 loadMoreItems,
 hasNextPage,
 isNextPageLoading,
}) {
 const itemCount = hasNextPage ? items.length + 1 : items.length;

 const loadMoreItems = isNextPageLoading ? () => {} : loadMoreItems;

 const isItemLoaded = (index) => !hasNextPage || index < items.length;
```

```
const Item = ({ index, style }) => {
 if (!isItemLoaded(index)) {
 return <div style={style}>Loading...</div>;
 }

 return <div style={style}>{items[index].name}</div>;
};

return (
 <InfiniteLoader
 isItemLoaded={isItemLoaded}
 itemCount={itemCount}
 loadMoreItems={loadMoreItems}
 >
 {({ onItemsRendered, ref }) => (
 <FixedSizeList
 ref={ref}
 height={400}
 width={300}
 itemCount={itemCount}
 itemSize={35}
 onItemsRendered={onItemsRendered}
 >
 {Item}
 </FixedSizeList>
)}
 </InfiniteLoader>
);
}

// Efficient list rendering with key prop
function EfficientList({ items }) {
 return (

 {items.map((item) => (
 // Using a stable unique key helps React identify which items
 // have changed, been added, or been removed
 <li key={item.id}>{item.name}
))}

);
}

// Bad key usage (can cause performance issues)
function InefficientList({ items }) {
 return (

 {items.map((item, index) => (
 // Using index as key can cause issues when list items can
 // reorder or when items can be added/removed in the middle
 <li key={index}>{item.name}
))}

);
}
```

```
);

}

// Optimizing list item components
const MemoizedListItem = memo(function ListItem({ item, onAction }) {
 return (

 {item.name}
 <button onClick={() => onAction(item.id)}>View Details</button>

);
});

function OptimizedList({ items }) {
 // Create a stable reference for the event handler
 const handleAction = useCallback((id) => {
 console.log('Action on item', id);
 }, []);

 return (

 {items.map((item) => (
 <MemoizedListItem key={item.id} item={item} onAction={handleAction} />
))}

);
}

// Using a pagination approach instead of rendering all items
function PaginatedList({ items, itemsPerPage = 10 }) {
 const [currentPage, setCurrentPage] = useState(1);

 const totalPages = Math.ceil(items.length / itemsPerPage);
 const startIndex = (currentPage - 1) * itemsPerPage;
 const endIndex = Math.min(startIndex + itemsPerPage, items.length);

 const currentItems = items.slice(startIndex, endIndex);

 return (
 <div>

 {currentItems.map((item) => (
 <li key={item.id}>{item.name}
)));

 <div className="pagination">
 <button
 onClick={() => setCurrentPage((p) => Math.max(p - 1, 1))}
 disabled={currentPage === 1}
 >
 Previous
 </button>
 </div>
 </div>
);
}
```

```

 Page {currentPage} of {totalPages}

<button
 onClick={() => setCurrentPage((p) => Math.min(p + 1, totalPages))}
 disabled={currentPage === totalPages}
>
 Next
</button>
</div>
</div>
);
}

// Optimizing rendering of large datasets using chunking
function ChunkedRendering({ items }) {
 const [renderedItems, setRenderedItems] = useState([]);
 const [isRendering, setIsRendering] = useState(false);
 const chunkSize = 100;
 const timeoutRef = useRef(null);

 // Render items in chunks
 useEffect(() => {
 setRenderedItems([]);

 if (items.length === 0) return;

 const renderChunk = (startIndex) => {
 const endIndex = Math.min(startIndex + chunkSize, items.length);

 setIsRendering(true);
 setRenderedItems((prev) => [
 ...prev,
 ...items.slice(startIndex, endIndex),
]);

 if (endIndex < items.length) {
 // Schedule the next chunk
 timeoutRef.current = setTimeout(() => {
 renderChunk(endIndex);
 }, 10); // Small delay to avoid blocking the main thread
 } else {
 setIsRendering(false);
 }
 };

 renderChunk(0);
 });

 return () => {
 if (timeoutRef.current) {
 clearTimeout(timeoutRef.current);
 }
 };
}
```

```
}, [items]);

 return (
 <div>
 {isRendering && (
 <div>
 Rendering {renderedItems.length} of {items.length} items...
 </div>
)}

 {renderedItems.map((item) => (
 <li key={item.id}>{item.name}
))}

 </div>
);
}
```

## Code Splitting and Lazy Loading

```
// React.lazy and Suspense for component code splitting
import React, { Suspense, lazy } from 'react';

// Instead of:
// import ExpensiveComponent from './ExpensiveComponent';

// Use:
const ExpensiveComponent = lazy(() => import('./ExpensiveComponent'));

function MyApp() {
 return (
 <div>
 <h1>My App</h1>
 <Suspense fallback={<div>Loading...</div>}>
 <ExpensiveComponent />
 </Suspense>
 </div>
);
}

// Code splitting based on routes
import { BrowserRouter, Routes, Route } from 'react-router-dom';

const Home = lazy(() => import('./pages/Home'));
const Dashboard = lazy(() => import('./pages/Dashboard'));
const Profile = lazy(() => import('./pages/Profile'));
const Settings = lazy(() => import('./pages/Settings'));

function App() {
 return (
```

```
<BrowserRouter>
 <div className="app">
 <header>{/* Navigation */}</header>
 <main>
 <Suspense fallback={<div>Loading page...</div>}>
 <Routes>
 <Route path="/" element={<Home />} />
 <Route path="/dashboard" element={<Dashboard />} />
 <Route path="/profile" element={<Profile />} />
 <Route path="/settings" element={<Settings />} />
 </Routes>
 </Suspense>
 </main>
 </div>
</BrowserRouter>
);
}

// Conditional code splitting with lazy loading
function FeatureFlaggedComponent({ featureEnabled }) {
 // Load different components based on feature flag
 const FeatureComponent = lazy(() =>
 featureEnabled
 ? import('./NewFeatureComponent')
 : import('./LegacyComponent')
);
 return (
 <Suspense fallback={<div>Loading feature...</div>}>
 <FeatureComponent />
 </Suspense>
);
}

// Code splitting based on user interaction
function ImageViewer({ image }) {
 const [isEditing, setIsEditing] = useState(false);

 // Only load the editor when the user needs it
 const ImageEditor = lazy(() => import('./ImageEditor'));

 return (
 <div>

 <button onClick={() => setIsEditing(true)}>Edit Image</button>

 {isEditing && (
 <Suspense fallback={<div>Loading editor...</div>}>
 <ImageEditor image={image} onClose={() => setIsEditing(false)} />
 </Suspense>
)}
 </div>
);
}
```

```
}

// Preloading modules for better UX
const DashboardComponent = lazy(() => import('./Dashboard'));

function PreloadExample() {
 const [showDashboard, setShowDashboard] = useState(false);

 // Preload the dashboard when user hovers the button
 const handleMouseEnter = () => {
 // This starts loading the component before it's rendered
 const preload = import('./Dashboard');
 };

 return (
 <div>
 <button
 onMouseEnter={handleMouseEnter}
 onClick={() => setShowDashboard(true)}
 >
 Show Dashboard
 </button>

 {showDashboard && (
 <Suspense fallback={<div>Loading...</div>}>
 <DashboardComponent />
 </Suspense>
)}
 </div>
);
}

// Code splitting with named exports
// Component with named exports:
// export const ComponentA = () => <div>Component A</div>;
// export const ComponentB = () => <div>Component B</div>;

// Importing with lazy:
const ModuleComponents = lazy(() => import('./ModuleWithComponents'));

function App() {
 return (
 <Suspense fallback={<div>Loading...</div>}>
 <ModuleComponents.ComponentA />
 <ModuleComponents.ComponentB />
 </Suspense>
);
}

// Alternative solution with default export for each component:
const ComponentA = lazy(() =>
 import('./ModuleWithComponents').then((module) => ({
 default: module.ComponentA,
 }))
);
```

```
);

// Using Suspense for data fetching (experimental)
// This approach is evolving in React
function SuspenseForData() {
 return (
 <Suspense fallback={<div>Loading user data...</div>}>
 <UserData />
 </Suspense>
);
}

// Create a resource using a suspense-compatible data fetching library
const userResource = createResource(fetchUserData);

function UserData() {
 // This will suspend if data isn't ready
 const user = userResource.read();

 return <div>User: {user.name}</div>;
}
```

## Bundle Optimization

```
// Using dynamic imports to split bundles
function DynamicImportExample() {
 const [chart, setChart] = useState(null);

 const loadChartLibrary = async () => {
 try {
 // Only load the large chart library when needed
 const ChartModule = await import('chart.js');
 const ctx = document.getElementById('chart').getContext('2d');

 setChart(new ChartModule.Chart(ctx, {
 type: 'bar',
 data: { /* chart data */ }
 }));
 } catch (error) {
 console.error('Failed to load chart library', error);
 }
 };

 return (
 <div>
 <button onClick={loadChartLibrary}>
 Show Chart
 </button>
 <canvas id="chart" width="400" height="300"></canvas>
 </div>
);
}
```

```
}

// Tree shaking - importing only what you need
// Bad: Imports the entire library
import * as _ from 'lodash';

// Good: Import only the needed functions
import map from 'lodash/map';
import filter from 'lodash/filter';

// Better: Use a library that supports tree shaking
import { map, filter } from 'lodash-es';

// Using the React production build
// In webpack.config.js:
const TerserPlugin = require('terser-webpack-plugin');

module.exports = {
 mode: 'production',
 optimization: {
 minimizer: [
 new TerserPlugin({
 terserOptions: {
 compress: {
 drop_console: true, // Remove console logs
 },
 },
 }),
],
 },
 // Other webpack config...
};

// Remove PropTypes in production
// Install babel-plugin-transform-react-remove-prop-types
// .babelrc
{
 "plugins": [
 ["transform-react-remove-prop-types", {
 "removeImport": true
 }]
]
}

// Split vendor and application code
// webpack.config.js
module.exports = {
 // Other config...
 optimization: {
 splitChunks: {
 chunks: 'all',
 cacheGroups: {
 vendor: {
 test: /[\\/]node_modules[\\/]/,
 }
 }
 }
 }
}
```

```
 name: 'vendors',
 chunks: 'all',
 },
 },
 },
},
};

// Using environment variables to strip out development-only code
// .env.production
NODE_ENV=production
REACT_APP_API_URL=https://api.production.com

// In component:
if (process.env.NODE_ENV === 'development') {
 // Development-only code
 console.log('Debug info:', data);
}

// Using performance budget with webpack
// webpack.config.js
module.exports = {
 // Other config...
 performance: {
 hints: 'warning', // or 'error' to fail the build
 maxAssetSize: 200000, // bytes
 maxEntrypointSize: 400000, // bytes
 },
};

// Analyzing bundle size
// npm install --save-dev webpack-bundle-analyzer
// webpack.config.js
const BundleAnalyzerPlugin = require('webpack-bundle-analyzer').BundleAnalyzerPlugin;

module.exports = {
 // Other config...
 plugins: [
 new BundleAnalyzerPlugin()
]
};

// Compression
// webpack.config.js
const CompressionPlugin = require('compression-webpack-plugin');

module.exports = {
 // Other config...
 plugins: [
 new CompressionPlugin({
 algorithm: 'gzip',
 test: /\.js$|\.css$|\.html$/,
 threshold: 10240, // Only compress assets > 10kb
 })
],
};
```

```
 minRatio: 0.8
 })
]
};
```

## Common Performance Patterns

```
// Event pooling to prevent excessive renders
function SearchInput({ onSearch }) {
 const [query, setQuery] = useState('');
 const timeoutRef = useRef(null);

 // Debounce the search to prevent too many state updates
 const handleChange = (e) => {
 const value = e.target.value;
 setQuery(value);

 clearTimeout(timeoutRef.current);

 timeoutRef.current = setTimeout(() => {
 onSearch(value);
 }, 300);
 };

 useEffect(() => {
 // Cleanup on unmount
 return () => {
 if (timeoutRef.current) {
 clearTimeout(timeoutRef.current);
 }
 };
 }, []);

 return (
 <input
 type="text"
 value={query}
 onChange={handleChange}
 placeholder="Search..." />
);
}

// Optimizing scroll/resize event handlers with throttling
function ScrollPositionTracker() {
 const [scrollY, setScrollY] = useState(0);

 useEffect(() => {
 // Throttle scroll event handler
 const handleScroll = throttle(() => {
 setScrollY(window.scrollY);
 }, 100);
 window.addEventListener('scroll', handleScroll);
 return () => {
 window.removeEventListener('scroll', handleScroll);
 };
 }, []);
}
```

```
}, 100);

window.addEventListener('scroll', handleScroll);

return () => {
 window.removeEventListener('scroll', handleScroll);
};

}, []));

return (
 <div className="scroll-tracker">
 <p>Scroll position: {scrollY}px</p>
 </div>
);
}

// Throttle implementation
function throttle(fn, delay) {
 let lastCall = 0;

 return function (...args) {
 const now = Date.now();

 if (now - lastCall < delay) {
 return;
 }

 lastCall = now;
 return fn(...args);
 };
}

// Using IntersectionObserver for lazy loading images
function LazyImage({ src, alt, placeholder }) {
 const [isLoaded, setIsLoaded] = useState(false);
 const [isInView, setIsInView] = useState(false);
 const imgRef = useRef();

 useEffect(() => {
 const observer = new IntersectionObserver(
 ([entry]) => {
 if (entry.isIntersecting) {
 setIsInView(true);
 observer.disconnect();
 }
 },
 { threshold: 0.1 }
);

 if (imgRef.current) {
 observer.observe(imgRef.current);
 }

 return () => {

```

```
 observer.disconnect();
 };
}, []);

return (
 <div className="lazy-image-container" ref={imgRef}>
 {isInView ? (
 <img
 src={src}
 alt={alt}
 onLoad={() => setIsLoaded(true)}
 style={{
 opacity: isLoading ? 1 : 0,
 transition: 'opacity 0.3s',
 }}
 />
) : (

)}
 </div>
);
}

// Windowing for large forms and inputs
function LargeForm() {
 // Instead of rendering all form fields at once, render them in focused groups
 const [activeSection, setActiveSection] = useState('personal');

 return (
 <form>
 <nav>
 <button
 type="button"
 onClick={() => setActiveSection('personal')}
 className={activeSection === 'personal' ? 'active' : ''}
 >
 Personal Info
 </button>
 <button
 type="button"
 onClick={() => setActiveSection('address')}
 className={activeSection === 'address' ? 'active' : ''}
 >
 Address
 </button>
 <button
 type="button"
 onClick={() => setActiveSection('payment')}
 className={activeSection === 'payment' ? 'active' : ''}
 >
 Payment
 </button>
 </nav>

```

```
{activeSection === 'personal' && (
 <section>
 <h2>Personal Information</h2>
 {/* Personal info fields */}
 </section>
)};

{activeSection === 'address' && (
 <section>
 <h2>Address</h2>
 {/* Address fields */}
 </section>
)};

{activeSection === 'payment' && (
 <section>
 <h2>Payment Details</h2>
 {/* Payment fields */}
 </section>
)
</form>
);
}

// Using Web Workers for CPU-intensive operations
function DataProcessor() {
 const [data, setData] = useState([]);
 const [results, setResults] = useState(null);
 const [isProcessing, setIsProcessing] = useState(false);

 const processData = () => {
 setIsProcessing(true);

 // Create a web worker
 const worker = new Worker('/processDataWorker.js');

 // Send data to worker
 worker.postMessage(data);

 // Handle response
 worker.onmessage = (e) => {
 setResults(e.data);
 setIsProcessing(false);
 worker.terminate();
 };

 // Handle errors
 worker.onerror = (error) => {
 console.error('Worker error:', error);
 setIsProcessing(false);
 worker.terminate();
 };
 };
}
```

```
return (
 <div>
 <textarea
 onChange={(e) => setData(e.target.value.split('\n'))}
 placeholder="Enter data, one item per line"
 rows={10}
 />

 <button onClick={processData} disabled={isProcessing || !data.length}>
 {isProcessing ? 'Processing...' : 'Process Data'}
 </button>

 {results && (
 <div>
 <h3>Results:</h3>
 <pre>{JSON.stringify(results, null, 2)}</pre>
 </div>
)}
 </div>
);

// worker.js
/*
self.onmessage = function(e) {
 const data = e.data;

 // Perform CPU-intensive calculation
 const results = data.map(item => {
 // Complex processing...
 return processItem(item);
 });

 // Send results back to main thread
 self.postMessage(results);
};

function processItem(item) {
 // Simulate complex calculation
 let result = 0;
 for (let i = 0; i < 1000000; i++) {
 result += Math.sqrt(i) * parseInt(item, 10);
 }
 return result;
}
*/

```

## Modern React Patterns and Best Practices

### Conceptual Foundations

Modern React patterns are established solutions to common design challenges in React applications. These patterns help create more maintainable, reusable, and testable components by leveraging React's features in idiomatic ways.

**Mental Model:** Think of these patterns as established architectural blueprints. Just as architects don't design every building from scratch but use proven blueprints modified for specific requirements, React developers use these patterns as starting points for solving common problems.

## Component Design Patterns

```
// Compound Component Pattern
// Allows components to work together to form a cohesive API
import React, { createContext, useContext, useState } from 'react';

// Step 1: Create a context
const TabsContext = createContext();

// Step 2: Create the parent component
function Tabs({ children, defaultIndex = 0 }) {
 const [activeIndex, setActiveIndex] = useState(defaultIndex);

 // Value to be shared with child components
 const value = { activeIndex, setActiveIndex };

 return (
 <TabsContext.Provider value={value}>
 <div className="tabs">{children}</div>
 </TabsContext.Provider>
);
}

// Step 3: Create child components
function TabList({ children }) {
 return <div className="tab-list">{children}</div>;
}

function Tab({ children, index }) {
 const { activeIndex, setActiveIndex } = useContext(TabsContext);
 const isActive = activeIndex === index;

 return (
 <button
 className={`tab ${isActive ? 'active' : ''}`}
 onClick={() => setActiveIndex(index)}
 >
 {children}
 </button>
);
}

function TabPanels({ children }) {
 const { activeIndex } = useContext(TabsContext);
```

```
const activeChild = React.Children.toArray(children)[activeIndex];

return <div className="tab-panels">{activeChild}</div>;
}

function TabPanel({ children }) {
 return <div className="tab-panel">{children}</div>;
}

// Step 4: Add the child components as properties of the parent
Tabs.TabList = TabList;
Tabs.Tab = Tab;
Tabs.TabPanels = TabPanels;
Tabs.TabPanel = TabPanel;

// Usage
function App() {
 return (
 <Tabs>
 <Tabs.TabList>
 <Tabs.Tab index={0}>Tab 1</Tabs.Tab>
 <Tabs.Tab index={1}>Tab 2</Tabs.Tab>
 <Tabs.Tab index={2}>Tab 3</Tabs.Tab>
 </Tabs.TabList>
 <Tabs.TabPanels>
 <Tabs.TabPanel>Content for Tab 1</Tabs.TabPanel>
 <Tabs.TabPanel>Content for Tab 2</Tabs.TabPanel>
 <Tabs.TabPanel>Content for Tab 3</Tabs.TabPanel>
 </Tabs.TabPanels>
 </Tabs>
);
}

// Render Props Pattern
// Uses a prop (typically called render or children) to share code between
// components
function DataFetcher({ url, render }) {
 const [data, setData] = useState(null);
 const [loading, setLoading] = useState(true);
 const [error, setError] = useState(null);

 useEffect(() => {
 setLoading(true);

 fetch(url)
 .then((response) => {
 if (!response.ok) throw new Error('Network response was not ok');
 return response.json();
 })
 .then((data) => {
 setData(data);
 setError(null);
 })
 .catch((error) => {
```

```
 setError(error.message);
 setData(null);
 })
 .finally(() => {
 setLoading(false);
 });
}, [url]);
```

```
 return render({ data, loading, error });
}
```

```
// Usage of render props
function App() {
 return (
 <DataFetcher
 url="https://api.example.com/users"
 render={({ data, loading, error }) => {
 if (loading) return <div>Loading...</div>;
 if (error) return <div>Error: {error}</div>;
 if (!data) return <div>No data found</div>;
```

```

 return (

 {data.map((user) => (
 <li key={user.id}>{user.name}
)));

);
 }
 />
);
}
```

```
// Alternative using children as a function
function DataFetcherWithChildren({ url, children }) {
 const [data, setData] = useState(null);
 const [loading, setLoading] = useState(true);
 const [error, setError] = useState(null);

 // Same fetch logic as above...

 return children({ data, loading, error });
}

// Usage with children as a function
function App() {
 return (
 <DataFetcherWithChildren url="https://api.example.com/users">
 {({ data, loading, error }) => {
 // Render logic...
 }}
 </DataFetcherWithChildren>
);
}
```

```
// Higher-Order Component (HOC) Pattern
// Creates a new component by wrapping another component
function withAuth(WrappedComponent) {
 // Return a new component
 return function WithAuth(props) {
 const [isAuthenticated, setIsAuthenticated] = useState(false);
 const [user, setUser] = useState(null);

 // Check authentication status when component mounts
 useEffect(() => {
 const checkAuth = async () => {
 try {
 const response = await fetch('/api/auth/check');
 if (response.ok) {
 const userData = await response.json();
 setUser(userData);
 setIsAuthenticated(true);
 } else {
 setIsAuthenticated(false);
 setUser(null);
 }
 } catch (error) {
 console.error('Auth check failed:', error);
 setIsAuthenticated(false);
 setUser(null);
 }
 }
 checkAuth();
 }, []);

 // Redirect if not authenticated
 if (!isAuthenticated) {
 return (
 <div>
 <p>Please log in to access this page</p>
 <button>Log In</button>
 </div>
);
 }

 // Render the wrapped component with additional props
 return <WrappedComponent {...props} user={user} />;
 };
}

// Using the HOC
function Dashboard({ user }) {
 return (
 <div>
 <h1>Welcome, {user.name}</h1>
 <p>Email: {user.email}</p>
 {/* Dashboard content */}
 </div>
);
}
```

```
</div>
);
}

const DashboardWithAuth = withAuth(Dashboard);

// Hook Pattern
// Custom hooks encapsulate and reuse stateful logic
function useLocalStorage(key, initialValue) {
 // State to store our value
 const [storedValue, setStoredValue] = useState(() => {
 try {
 // Get from local storage by key
 const item = window.localStorage.getItem(key);
 // Parse stored json or return initialValue
 return item ? JSON.parse(item) : initialValue;
 } catch (error) {
 console.error(error);
 return initialValue;
 }
 });
}

// Return a wrapped version of useState's setter function
const setValue = (value) => {
 try {
 // Allow value to be a function
 const valueToStore =
 value instanceof Function ? value(storedValue) : value;
 // Save state
 setStoredValue(valueToStore);
 // Save to local storage
 window.localStorage.setItem(key, JSON.stringify(valueToStore));
 } catch (error) {
 console.error(error);
 }
};

return [storedValue, setValue];
}

// Using the custom hook
function App() {
 const [name, setName] = useLocalStorage('name', 'Bob');

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

```
}

// Stateless UI Components Pattern
// Separating UI from logic for better reusability
// 1. Presentational component (just UI rendering)
function Button({ onClick, disabled, children, variant = 'primary' }) {
 const className = `button button-${variant}`;

 return (
 <button className={className} onClick={onClick} disabled={disabled}>
 {children}
 </button>
);
}

// 2. Container component (handles logic)
function SubmitButtonContainer({ onSubmit, isSubmitting, children }) {
 const [submitted, setSubmitted] = useState(false);

 const handleClick = async () => {
 setSubmitted(true);
 try {
 await onSubmit();
 } catch (error) {
 console.error('Submission failed:', error);
 } finally {
 setSubmitted(false);
 }
 };

 // Use the presentational Button component for UI
 return (
 <Button
 onClick={handleClick}
 disabled={submitted || isSubmitting}
 variant="primary"
 >
 {children || (isSubmitting ? 'Submitting...' : 'Submit')}
 </Button>
);
}

// Usage
function FormExample() {
 const [isSubmitting, setIsSubmitting] = useState(false);

 const handleSubmit = async () => {
 setIsSubmitting(true);
 // Simulate API call
 await new Promise((resolve) => setTimeout(resolve, 1000));
 setIsSubmitting(false);
 };

 return (

```

```
<form>
 {/* Form fields */}
 <SubmitButtonContainer
 onSubmit={handleSubmit}
 isSubmitting={isSubmitting}
 />
</form>
);

}

// Props Getter Pattern
// Allows component users to apply their own props while preserving required
behavior
function useToggle(initialState = false) {
 const [on, setOn] = useState(initialState);

 const toggle = () => setOn((prevOn) => !prevOn);
 const reset = () => setOn(initialState);

 // Props getter for the toggle element
 const getTogglerProps = (props = {}) => {
 return {
 'aria-pressed': on,
 onClick: callAll(toggle, props.onClick),
 ...props,
 };
 };
}

// Helper to call multiple functions
function callAll(...fns) {
 return (...args) => {
 fns.forEach((fn) => fn && fn(...args));
 };
}

return {
 on,
 toggle,
 reset,
 getTogglerProps,
};
}

// Usage
function ToggleComponent() {
 const { on, getTogglerProps } = useToggle();

 return (
 <div>
 <button
 {...getTogglerProps({
 className: 'toggle-btn',
 // Adding our own onClick that will run after the toggle
 onClick: () => console.log('toggled!'),
 })}
 </button>
 </div>
);
}
```

```
 })}
 >
 {on ? 'ON' : 'OFF'}
 </button>
 <div>The button is {on ? 'on' : 'off'}</div>
 </div>
);
}

// State Reducer Pattern
// Gives consumers more control over the internal state changes
function useToggleWithReducer(
 initialState = false,
 reducer = (state, action) => {
 switch (action.type) {
 case 'toggle':
 return !state;
 case 'reset':
 return initialState;
 default:
 return state;
 }
}
) {
 const [on, dispatch] = useReducer(reducer, initialState);

 const toggle = () => dispatch({ type: 'toggle' });
 const reset = () => dispatch({ type: 'reset' });

 const getTogglerProps = (props = {}) => {
 return {
 'aria-pressed': on,
 onClick: callAll(toggle, props.onClick),
 ...props,
 };
 };

 function callAll(...fns) {
 return (...args) => {
 fns.forEach((fn) => fn && fn(...args));
 };
 }

 return {
 on,
 toggle,
 reset,
 getTogglerProps,
 };
}

// Usage with custom reducer
function CustomToggle() {
 function customReducer(state, action) {
```

```
switch (action.type) {
 case 'toggle':
 // Prevent toggling to the "on" state if it's the weekend
 if (!state && isWeekend()) {
 console.log('No toggling on the weekend!');
 return state;
 }
 return !state;
 case 'reset':
 return false;
 default:
 return state;
}

const { on, getTogglerProps } = useToggleWithReducer(false, customReducer);

function isWeekend() {
 const day = new Date().getDay();
 return day === 0 || day === 6; // 0 is Sunday, 6 is Saturday
}

return (
 <div>
 <button {...getTogglerProps()}>{on ? 'ON' : 'OFF'}</button>
 <div>
 {isWeekend() && (Toggling to ON is disabled on weekends)}
 </div>
 </div>
);
}

// Controlled vs Uncontrolled Components
// 1. Uncontrolled component (internal state)
function UncontrolledInput() {
 const inputRef = useRef();

 const handleSubmit = (e) => {
 e.preventDefault();
 console.log('Input value:', inputRef.current.value);
 };

 return (
 <form onSubmit={handleSubmit}>
 <label>
 Uncontrolled input:
 <input type="text" defaultValue="default text" ref={inputRef} />
 </label>
 <button type="submit">Submit</button>
 </form>
);
}

// 2. Controlled component (external state)
```

```
function ControlledInput() {
 const [value, setValue] = useState('');

 const handleChange = (e) => {
 setValue(e.target.value);
 };

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

 return (
 <form onSubmit={handleSubmit}>
 <label>
 Controlled input:
 <input type="text" value={value} onChange={handleChange} />
 </label>
 <button type="submit">Submit</button>
 </form>
);
}

// 3. Component that can be either controlled or uncontrolled
function FlexibleInput({ value: controlledValue, onChange, defaultValue }) {
 const [internalValue, setInternalValue] = useState(defaultValue || '');
 const isControlled = controlledValue !== undefined;
 const displayValue = isControlled ? controlledValue : internalValue;

 const handleChange = (e) => {
 const newValue = e.target.value;

 // Update internal state if uncontrolled
 if (!isControlled) {
 setInternalValue(newValue);
 }

 // Call onChange if provided
 if (onChange) {
 onChange(e);
 }
 };
}

return <input type="text" value={displayValue} onChange={handleChange} />;
}

// Usage
function FormWithBothInputs() {
 const [controlledValue, setControlledValue] = useState('controlled value');

 return (
 <div>
 <div>
 <label>Uncontrolled:</label>
 </div>
 </div>
);
}
```

```
 <FlexibleInput defaultValue="default value" />
 </div>
 <div>
 <label>Controlled:</label>
 <FlexibleInput
 value={controlledValue}
 onChange={(e) => setControlledValue(e.target.value)}
 />
 </div>
</div>
);
}
```

## Context Usage Patterns

```
// 1. Basic Context Creation
const ThemeContext = createContext();

function ThemeProvider({ children, initialTheme = 'light' }) {
 const [theme, setTheme] = useState(initialTheme);

 const toggleTheme = () => {
 setTheme((prevTheme) => (prevTheme === 'light' ? 'dark' : 'light'));
 };

 const value = {
 theme,
 toggleTheme,
 };

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

// Custom hook for consuming the context
function useTheme() {
 const context = useContext(ThemeContext);
 if (context === undefined) {
 throw new Error('useTheme must be used within a ThemeProvider');
 }
 return context;
}

// 2. Context with Multiple Providers
// User context
const UserContext = createContext();

function UserProvider({ children }) {
 const [user, setUser] = useState(null);
```

```
const login = (userData) => setUser(userData);
const logout = () => setUser(null);

const value = { user, login, logout };

return <UserContext.Provider value={value}>{children}</UserContext.Provider>;
}

function useUser() {
 const context = useContext(UserContext);
 if (context === undefined) {
 throw new Error('useUser must be used within a UserProvider');
 }
 return context;
}

// Notifications context
const NotificationsContext = createContext();

function NotificationsProvider({ children }) {
 const [notifications, setNotifications] = useState([]);

 const addNotification = (notification) => {
 setNotifications((prev) => [...prev, notification]);
 };

 const removeNotification = (id) => {
 setNotifications((prev) => prev.filter((notif) => notif.id !== id));
 };

 const value = {
 notifications,
 addNotification,
 removeNotification,
 };

 return (
 <NotificationsContext.Provider value={value}>
 {children}
 </NotificationsContext.Provider>
);
}

function useNotifications() {
 const context = useContext(NotificationsContext);
 if (context === undefined) {
 throw new Error(
 'useNotifications must be used within a NotificationsProvider'
);
 }
 return context;
}

// Combined providers
```

```
function AppProviders({ children }) {
 return (
 <ThemeProvider>
 <UserProvider>
 <NotificationsProvider>{children}</NotificationsProvider>
 </UserProvider>
 </ThemeProvider>
);
}

// Usage
function App() {
 return (
 <AppProviders>
 <Main />
 </AppProviders>
);
}

// 3. Context with a Reducer
const CounterContext = createContext();

// Reducer function
function counterReducer(state, action) {
 switch (action.type) {
 case 'increment':
 return { count: state.count + (action.payload || 1) };
 case 'decrement':
 return { count: state.count - (action.payload || 1) };
 case 'reset':
 return { count: 0 };
 default:
 return state;
 }
}

function CounterProvider({ children }) {
 const [state, dispatch] = useReducer(counterReducer, { count: 0 });

 // Create action creators
 const increment = (amount) =>
 dispatch({ type: 'increment', payload: amount });
 const decrement = (amount) =>
 dispatch({ type: 'decrement', payload: amount });
 const reset = () => dispatch({ type: 'reset' });

 const value = {
 count: state.count,
 increment,
 decrement,
 reset,
 };

 return (
 <CounterContext.Provider value={value}>{children}</CounterContext.Provider>
);
}
```

```
<CounterContext.Provider value={value}>{children}</CounterContext.Provider>
);
}

function useCounter() {
 const context = useContext(CounterContext);
 if (context === undefined) {
 throw new Error('useCounter must be used within a CounterProvider');
 }
 return context;
}

// 4. Context with Async Actions
const TodosContext = createContext();

function todosReducer(state, action) {
 switch (action.type) {
 case 'FETCH_INIT':
 return { ...state, isLoading: true, error: null };
 case 'FETCH_SUCCESS':
 return {
 ...state,
 isLoading: false,
 error: null,
 todos: action.payload,
 };
 case 'FETCH_FAILURE':
 return {
 ...state,
 isLoading: false,
 error: action.payload,
 };
 case 'ADD_TODO':
 return {
 ...state,
 todos: [...state.todos, action.payload],
 };
 case 'UPDATE_TODO':
 return {
 ...state,
 todos: state.todos.map((todo) =>
 todo.id === action.payload.id ? action.payload : todo
),
 };
 case 'DELETE_TODO':
 return {
 ...state,
 todos: state.todos.filter((todo) => todo.id !== action.payload),
 };
 default:
 return state;
 }
}
```

```
function TodosProvider({ children }) {
 const [state, dispatch] = useReducer(todosReducer, {
 todos: [],
 isLoading: false,
 error: null,
 });

 // Async action to fetch todos
 const fetchTodos = async () => {
 dispatch({ type: 'FETCH_INIT' });
 try {
 const response = await fetch('https://api.example.com/todos');
 if (!response.ok) throw new Error('Failed to fetch');
 const data = await response.json();
 dispatch({ type: 'FETCH_SUCCESS', payload: data });
 } catch (error) {
 dispatch({ type: 'FETCH_FAILURE', payload: error.message });
 }
 };

 // Async action to add a todo
 const addTodo = async (todo) => {
 try {
 const response = await fetch('https://api.example.com/todos', {
 method: 'POST',
 headers: { 'Content-Type': 'application/json' },
 body: JSON.stringify(todo),
 });
 if (!response.ok) throw new Error('Failed to add todo');
 const newTodo = await response.json();
 dispatch({ type: 'ADD_TODO', payload: newTodo });
 return newTodo;
 } catch (error) {
 dispatch({ type: 'FETCH_FAILURE', payload: error.message });
 throw error;
 }
 };
}

// Provide the state and actions
const value = {
 todos: state.todos,
 isLoading: state.isLoading,
 error: state.error,
 fetchTodos,
 addTodo,
 // Add other actions as needed
};

return (
 <TodosContext.Provider value={value}>{children}</TodosContext.Provider>
);
}

function useTodos() {
```

```
const context = useContext(TodosContext);
if (context === undefined) {
 throw new Error('useTodos must be used within a TodosProvider');
}
return context;
}
```

## Advanced Component Composition

```
// 1. Using children for flexible compositions
function Card({ title, children }) {
 return (
 <div className="card">
 <div className="card-header">
 <h2>{title}</h2>
 </div>
 <div className="card-body">{children}</div>
 </div>
);
}

// Usage
function UserCard({ user }) {
 return (
 <Card title={user.name}>
 <p>Email: {user.email}</p>
 <p>Role: {user.role}</p>
 <button>View Profile</button>
 </Card>
);
}

// 2. Composition with layout components
function TwoColumnLayout({ left, right }) {
 return (
 <div className="two-column-layout">
 <div className="column left">{left}</div>
 <div className="column right">{right}</div>
 </div>
);
}

// Usage
function Dashboard() {
 return (
 <TwoColumnLayout
 left={<UserList users={users} />}
 right={<UserDetails selectedUser={selectedUser} />}
 />
);
}
```

```
// 3. Prop Collection and Getter Pattern for composition
function useFormInput(initialValue) {
 const [value, setValue] = useState(initialValue);
 const [touched, setTouched] = useState(false);
 const [error, setError] = useState(null);

 const handleChange = (e) => {
 setValue(e.target.value);
 };

 const handleBlur = () => {
 setTouched(true);
 };

 const validate = (validator) => {
 if (!validator) return true;

 const validationError = validator(value);
 setError(validationError);
 return !validationError;
 };
}

// Prop collection pattern
const inputProps = {
 value,
 onChange: handleChange,
 onBlur: handleBlur,
};

// Prop getter pattern (more flexible)
const getInputProps = (props = {}) => ({
 ...inputProps,
 ...props,
 // Override onChange to merge behaviors
 onChange: (e) => {
 inputProps.onChange(e);
 props.onChange && props.onChange(e);
 },
 // Override onBlur to merge behaviors
 onBlur: (e) => {
 inputProps.onBlur(e);
 props.onBlur && props.onBlur(e);
 },
});

return {
 value,
 touched,
 error,
 setValue,
 validate,
 inputProps, // For the collection pattern
 getInputProps, // For the getter pattern
}
```

```
};

}

// Usage with prop collection pattern
function LoginForm() {
 const email = useFormInput('');
 const password = useFormInput('');

 const handleSubmit = (e) => {
 e.preventDefault();
 const isEmailValid = email.validate((value) => {
 return !/\S+@\S+\.\S+/.test(value) ? 'Invalid email format' : null;
 });

 const isPasswordValid = password.validate((value) => {
 return value.length < 6 ? 'Password must be at least 6 characters' : null;
 });

 if (isEmailValid && isPasswordValid) {
 console.log('Form submitted', {
 email: email.value,
 password: password.value,
 });
 }
 };
}

return (
 <form onSubmit={handleSubmit}>
 <div>
 <label>Email:</label>
 <input type="email" {...email.inputProps} />
 {email.touched && email.error && (
 <div className="error">{email.error}</div>
)}
 </div>

 <div>
 <label>Password:</label>
 <input type="password" {...password.inputProps} />
 {password.touched && password.error && (
 <div className="error">{password.error}</div>
)}
 </div>

 <button type="submit">Login</button>
 </form>
);
}

// 4. Function as Child Component (FaC) / Render Prop
function Counter({ children }) {
 const [count, setCount] = useState(0);

 const increment = () => setCount(count + 1);
```

```
const decrement = () => setCount(count - 1);

// The component expects children to be a function
return children({ count, increment, decrement });
}

// Usage of function as children
function CounterDisplay() {
 return (
 <Counter>
 {({ count, increment, decrement }) => (
 <div>
 <button onClick={decrement}>-</button>
 {count}
 <button onClick={increment}>+</button>
 </div>
)}
 </Counter>
);
}

// 5. Polymorphic Components (components that can render as different elements)
function Box({ as: Component = 'div', children, ...props }) {
 return <Component {...props}>{children}</Component>;
}

// Usage
function Example() {
 return (
 <>
 <Box className="box">I'm a div (default)</Box>
 <Box as="section" className="box">
 I'm a section
 </Box>
 <Box as="button" onClick={() => alert('Clicked!')}>
 I'm a button
 </Box>
 <Box as={Link} to="/some-path">
 I'm a React Router Link
 </Box>
 </>
);
}

// 6. Component slots pattern (named placeholders)
function PageLayout({ header, sidebar, footer, children }) {
 return (
 <div className="page-layout">
 <header className="header">{header}</header>
 <div className="content-area">
 <aside className="sidebar">{sidebar}</aside>
 <main className="main-content">{children}</main>
 </div>
 <footer className="footer">{footer}</footer>
 </div>
);
}
```

```
 </div>
);
}

// Usage
function MyPage() {
 return (
 <PageLayout
 header={<Navigation />}
 sidebar={<SideMenu />}
 footer={<Copyright />}
 >
 <h1>Page Content</h1>
 <p>This is the main content of the page.</p>
 </PageLayout>
);
}
```

## State Management Patterns

```
// 1. Local component state
function Counter() {
 const [count, setCount] = useState(0);

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

// 2. Lifting state up
function ParentComponent() {
 const [count, setCount] = useState(0);

 const increment = () => setCount(count + 1);

 return (
 <div>
 <p>Count: {count}</p>
 <ChildComponent count={count} increment={increment} />
 </div>
);
}

function ChildComponent({ count, increment }) {
 return (
 <div>
 <p>Child showing count: {count}</p>
 <button onClick={increment}>Increment from child</button>
 </div>
);
}
```

```
 </div>
);
}

// 3. State with useReducer
function reducer(state, action) {
 switch (action.type) {
 case 'increment':
 return { count: state.count + 1 };
 case 'decrement':
 return { count: state.count - 1 };
 case 'reset':
 return { count: 0 };
 default:
 throw new Error(`Unsupported action type: ${action.type}`);
 }
}

function CounterWithReducer() {
 const [state, dispatch] = useReducer(reducer, { count: 0 });

 return (
 <div>
 <p>Count: {state.count}</p>
 <button onClick={() => dispatch({ type: 'increment' })}>+</button>
 <button onClick={() => dispatch({ type: 'decrement' })}>-</button>
 <button onClick={() => dispatch({ type: 'reset' })}>Reset</button>
 </div>
);
}

// 4. Context with reducer for global state
const CounterContext = createContext();

function CounterProvider({ children }) {
 const [state, dispatch] = useReducer(reducer, { count: 0 });

 return (
 <CounterContext.Provider value={{ state, dispatch }}>
 {children}
 </CounterContext.Provider>
);
}

function useCounterContext() {
 const context = useContext(CounterContext);
 if (!context) {
 throw new Error('useCounterContext must be used within a CounterProvider');
 }
 return context;
}

// Usage in multiple components
function CounterDisplay() {
```

```
const { state } = useCounterContext();
return <div>Count: {state.count}</div>;
}

function CounterButtons() {
 const { dispatch } = useCounterContext();
 return (
 <div>
 <button onClick={() => dispatch({ type: 'increment' })}>+</button>
 <button onClick={() => dispatch({ type: 'decrement' })}>-</button>
 <button onClick={() => dispatch({ type: 'reset' })}>Reset</button>
 </div>
);
}

function App() {
 return (
 <CounterProvider>
 <CounterDisplay />
 <CounterButtons />
 </CounterProvider>
);
}

// 5. Custom store with Context API - a simple Redux-like store
function createStore(reducer, initialState) {
 const StoreContext = createContext();

 function StoreProvider({ children }) {
 const [state, dispatch] = useReducer(reducer, initialState);

 const store = {
 state,
 dispatch,
 };

 return (
 <StoreContext.Provider value={store}>{children}</StoreContext.Provider>
);
 }

 function useStore() {
 const context = useContext(StoreContext);
 if (!context) {
 throw new Error('useStore must be used within a StoreProvider');
 }
 return context;
 }

 return {
 StoreProvider,
 useStore,
 };
}
```

```
// Usage with a todo application
const initialTodosState = {
 todos: [],
 filter: 'all',
};

function todosReducer(state, action) {
 switch (action.type) {
 case 'add_todo':
 return {
 ...state,
 todos: [
 ...state.todos,
 {
 id: Date.now(),
 text: action.payload,
 completed: false,
 },
],
 };
 case 'toggle_todo':
 return {
 ...state,
 todos: state.todos.map((todo) =>
 todo.id === action.payload
 ? { ...todo, completed: !todo.completed }
 : todo
),
 };
 case 'set_filter':
 return {
 ...state,
 filter: action.payload,
 };
 default:
 return state;
 }
}

const { StoreProvider, useStore } = createStore(
 todosReducer,
 initialTodosState
);

// Components
function TodoApp() {
 return (
 <StoreProvider>
 <h1>Todo App</h1>
 <AddTodo />
 <FilterButtons />
 <TodoList />
 </StoreProvider>
);
}
```

```
);

}

function AddTodo() {
 const { dispatch } = useStore();
 const [text, setText] = useState('');

 const handleSubmit = (e) => {
 e.preventDefault();
 if (!text.trim()) return;
 dispatch({ type: 'add_todo', payload: text });
 setText('');
 };

 return (
 <form onSubmit={handleSubmit}>
 <input
 value={text}
 onChange={(e) => setText(e.target.value)}
 placeholder="Add a todo"
 />
 <button type="submit">Add</button>
 </form>
);
}

function FilterButtons() {
 const { state, dispatch } = useStore();

 return (
 <div>
 <button
 className={state.filter === 'all' ? 'active' : ''}
 onClick={() => dispatch({ type: 'set_filter', payload: 'all' })}
 >
 All
 </button>
 <button
 className={state.filter === 'active' ? 'active' : ''}
 onClick={() => dispatch({ type: 'set_filter', payload: 'active' })}
 >
 Active
 </button>
 <button
 className={state.filter === 'completed' ? 'active' : ''}
 onClick={() => dispatch({ type: 'set_filter', payload: 'completed' })}
 >
 Completed
 </button>
 </div>
);
}

function TodoList() {
```

```

const { state, dispatch } = useStore();

const filteredTodos = state.todos.filter((todo) => {
 if (state.filter === 'active') return !todo.completed;
 if (state.filter === 'completed') return todo.completed;
 return true; // 'all' filter
});

return (

 {filteredTodos.map((todo) => (
 <li
 key={todo.id}
 style={{ textDecoration: todo.completed ? 'line-through' : 'none' }}
 onClick={() => dispatch({ type: 'toggle_todo', payload: todo.id })}
 >
 {todo.text}

))}

);
}

```

## Error Handling Patterns

```

// 1. Error boundaries (class component required)
class ErrorBoundary extends React.Component {
 constructor(props) {
 super(props);
 this.state = { hasError: false, error: null, errorInfo: null };
 }

 static getDerivedStateFromError(error) {
 // Update state so the next render will show the fallback UI
 return { hasError: true };
 }

 componentDidCatch(error, errorInfo) {
 // Log the error to an error reporting service
 console.error('Error caught by boundary:', error, errorInfo);
 this.setState({ error, errorInfo });

 // You could also send to a service like Sentry
 // logErrorToService(error, errorInfo);
 }

 render() {
 if (this.state.hasError) {
 // You can render any custom fallback UI
 return (
 this.props.fallback || (

```

```
<div className="error-ui">
 <h2>Something went wrong.</h2>
 {this.props.showDetails && (
 <details style={{ whiteSpace: 'pre-wrap' }}>
 {this.state.error && this.state.error.toString()}

 {this.state.errorInfo && this.state.errorInfo.componentStack}
 </details>
)}
 {this.props.onReset && (
 <button onClick={this.props.onReset}>Try again</button>
)}
</div>
)
);
}

return this.props.children;
}
}

// Usage
function BuggyComponent() {
 const [shouldError, setShouldError] = useState(false);

 if (shouldError) {
 throw new Error('Simulated error');
 }

 return (
 <div>
 <h2>Component with a potential error</h2>
 <button onClick={() => setShouldError(true)}>Trigger Error</button>
 </div>
);
}

function App() {
 return (
 <div>
 <h1>Error Boundary Example</h1>

 <ErrorBoundary
 fallback=<div>Something went wrong in the buggy component.</div>
 showDetails={process.env.NODE_ENV !== 'production'}
 onReset={() => window.location.reload()}
 >
 <BuggyComponent />
 </ErrorBoundary>

 <div>This part of the app will continue working</div>
 </div>
);
}
```

```
// 2. Try-catch for async operations
function UserProfile({ userId }) {
 const [user, setUser] = useState(null);
 const [error, setError] = useState(null);
 const [loading, setLoading] = useState(true);

 useEffect(() => {
 async function fetchUser() {
 try {
 setLoading(true);
 setError(null);

 const response = await fetch(`/api/users/${userId}`);

 if (!response.ok) {
 throw new Error(`Failed to fetch user: ${response.statusText}`);
 }

 const userData = await response.json();
 setUser(userData);
 } catch (err) {
 setError(err.message);
 setUser(null);
 } finally {
 setLoading(false);
 }
 }

 fetchUser();
 }, [userId]);

 if (loading) return <div>Loading user...</div>;
 if (error) return <div>Error loading user: {error}</div>;
 if (!user) return <div>No user found</div>;

 return (
 <div>
 <h2>{user.name}</h2>
 <p>Email: {user.email}</p>
 {/* Additional user information */}
 </div>
);
}

// 3. Custom hook for error handling in forms
function useFormWithValidation(initialValues) {
 const [values, setValues] = useState(initialValues);
 const [errors, setErrors] = useState({});
 const [touched, setTouched] = useState({});
 const [isSubmitting, setIsSubmitting] = useState(false);

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

```
 setValues((prev) => ({ ...prev, [name]: value }));
};

const handleBlur = (e) => {
 const { name } = e.target;
 setTouched((prev) => ({ ...prev, [name]: true }));
};

const validateField = (name, value, validators) => {
 if (!validators || !validators[name]) return null;

 const fieldValidators = validators[name];
 for (const validator of fieldValidators) {
 const error = validator(value, values);
 if (error) return error;
 }

 return null;
};

const validateForm = (validators) => {
 const newErrors = {};
 let isValid = true;

 Object.keys(values).forEach((name) => {
 const error = validateField(name, values[name], validators);
 if (error) {
 newErrors[name] = error;
 isValid = false;
 }
 });
}

setErrors(newErrors);
return isValid;
};

const handleSubmit = (onSubmit, validators) => {
 return async (e) => {
 e.preventDefault();

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

 // Validate form
 const isValid = validateForm(validators);

 if (isValid) {
 setIsSubmitting(true);
 try {
 await onSubmit(values);

```

```
 } catch (error) {
 // Handle submit error - can set a general form error
 setErrors(({prev}) => ({...prev, _general: error.message}));
 } finally {
 setIsSubmitting(false);
 }
 }
 };
 };

 return {
 values,
 errors,
 touched,
 isSubmitting,
 handleChange,
 handleBlur,
 handleSubmit,
 validateField,
 validateForm,
 setValues,
 setErrors,
 };
}
}

// Usage of the form validation hook
function ContactForm() {
 const {
 values,
 errors,
 touched,
 isSubmitting,
 handleChange,
 handleBlur,
 handleSubmit,
 } = useFormWithValidation({
 name: '',
 email: '',
 message: '',
 });

 const validators = {
 name: [({value}) => (!value.trim() ? 'Name is required' : null)],
 email: [
 ({value}) => (!value.trim() ? 'Email is required' : null),
 (value) => (!/\S+@\S+\.\S+/.test(value) ? 'Email is invalid' : null),
],
 message: [
 ({value}) => (!value.trim() ? 'Message is required' : null),
 (value) => (value.trim().length < 10 ? 'Message is too short' : null),
],
 };

 const submitForm = async (formValues) => {
```

```
// API call simulation
await new Promise((resolve) => setTimeout(resolve, 1000));
console.log('Form submitted:', formValues);
// In a real app, you'd make an API call here
};

return (
 <form onSubmit={handleSubmit(submitForm, validators)}>
 {errors._general && (
 <div className="error-message">{errors._general}</div>
)}

 <div>
 <label htmlFor="name">Name</label>
 <input
 id="name"
 name="name"
 value={values.name}
 onChange={handleChange}
 onBlur={handleBlur}
 />
 {touched.name && errors.name && (
 <div className="error-message">{errors.name}</div>
)}
 </div>

 <div>
 <label htmlFor="email">Email</label>
 <input
 id="email"
 name="email"
 type="email"
 value={values.email}
 onChange={handleChange}
 onBlur={handleBlur}
 />
 {touched.email && errors.email && (
 <div className="error-message">{errors.email}</div>
)}
 </div>

 <div>
 <label htmlFor="message">Message</label>
 <textarea
 id="message"
 name="message"
 value={values.message}
 onChange={handleChange}
 onBlur={handleBlur}
 />
 {touched.message && errors.message && (
 <div className="error-message">{errors.message}</div>
)}
 </div>
)
```

```
 <button type="submit" disabled={isSubmitting}>
 {isSubmitting ? 'Submitting...' : 'Submit'}
 </button>
 </form>
);
}

// 4. Centralized error handling with context
const ErrorContext = createContext();

function ErrorProvider({ children }) {
 const [errors, setErrors] = useState([]);

 const addError = (error) => {
 const id = Date.now();
 setErrors((prev) => [...prev, { id, message: error }]);

 // Auto remove after some time
 setTimeout(() => {
 removeError(id);
 }, 5000);
 };

 const removeError = (id) => {
 setErrors((prev) => prev.filter((error) => error.id !== id));
 };

 return (
 <ErrorContext.Provider value={{ errors, addError, removeError }}>
 {children}
 </ErrorContext.Provider>
);
}

function useError() {
 return useContext(ErrorContext);
}

// Error display component
function ErrorDisplay() {
 const { errors, removeError } = useError();

 if (errors.length === 0) return null;

 return (
 <div className="error-container">
 {errors.map((error) => (
 <div key={error.id} className="error-notification">
 {error.message}
 <button onClick={() => removeError(error.id)}>x</button>
 </div>
)));
 </div>
);
}
```

```
);
}

// Using the error context
function App() {
 return (
 <ErrorProvider>
 <div className="app">
 <ErrorDisplay />
 <MainContent />
 </div>
 </ErrorProvider>
);
}

function MainContent() {
 const { addError } = useError();

 const causeProblem = () => {
 // Simulate an error
 try {
 throw new Error('Something went wrong!');
 } catch (error) {
 addError(error.message);
 }
 };

 return (
 <div>
 <h1>Main Content</h1>
 <button onClick={causeProblem}>Trigger Error</button>
 </div>
);
}
```

## Project Structure and Organization

### Conceptual Foundations

Proper project structure in React applications helps with maintainability, scalability, and developer productivity. There is no one-size-fits-all approach, but certain patterns have emerged as effective ways to organize React codebases.

**Mental Model:** Think of your application structure like a well-organized library. Books (components) are categorized by subject (features), with clear labeling and cross-referencing systems. Just as a library evolves its organization to accommodate new acquisitions and changing needs, your application structure should evolve with the codebase while maintaining clear organization principles.

### Feature-Based Structure

In a feature-based structure, you organize your code by feature or domain, rather than by type of file. This approach helps keep related files together, making it easier to navigate and maintain the codebase as it grows.

```
src/
 └── features/
 ├── auth/
 │ ├── components/
 │ │ ├── LoginForm.js
 │ │ ├── RegistrationForm.js
 │ │ └── PasswordReset.js
 │ ├── hooks/
 │ │ └── useAuth.js
 │ ├── services/
 │ │ └── authService.js
 │ ├── utils/
 │ │ └── authUtils.js
 │ ├── contexts/
 │ │ └── AuthContext.js
 │ ├── types/
 │ │ └── auth.types.js
 │ └── AuthRoutes.js
 └── index.js // Exports public API for the feature

 ├── users/
 │ ├── components/
 │ ├── hooks/
 │ ├── services/
 │ ├── utils/
 │ ├── contexts/
 │ ├── types/
 │ └── index.js

 └── dashboard/
 ├── components/
 ├── hooks/
 └── ...

 └── shared/ // Shared across features
 ├── components/
 │ ├── Button/
 │ │ ├── Button.js
 │ │ ├── Button.css
 │ │ └── Button.test.js
 │ ├── Card/
 │ └── ...
 ├── hooks/
 │ └── useLocalStorage.js
 └── utils/
```

```
 └── formatters.js
 └── validators.js
 └── ...

 └── contexts/
 └── ThemeContext.js
 └── ...

 └── services/
 └── api.js
 └── ...

└── config/
 └── routes.js
 └── constants.js
 └── ...

└── App.js
└── index.js
```

## Atomic Design Structure

Atomic Design is a methodology for creating design systems, breaking down interfaces into five distinct levels: atoms, molecules, organisms, templates, and pages. This approach can also be applied to organizing React components.

```
src/
└── components/
 ├── atoms/ // Basic building blocks
 │ ├── Button/
 │ ├── Input/
 │ ├── Typography/
 │ └── ...
 ├── molecules/ // Simple combinations of atoms
 │ ├── FormField/
 │ ├── SearchBar/
 │ ├── Notification/
 │ └── ...
 ├── organisms/ // Complex UI components
 │ ├── Header/
 │ ├── Footer/
 │ ├── Sidebar/
 │ ├── UserCard/
 │ └── ...
 └── templates/ // Page layouts
 ├── DashboardTemplate/
 ├── AuthTemplate/
 └── ...
```

```
|- pages/ // Complete pages
 |- Dashboard/
 |- Login/
 |- UserProfile/
 |- ...
|- hooks/
|- contexts/
|- services/
|- utils/
|- App.js
|- index.js
```

## Hybrid Structure

A hybrid approach combines elements of different organizational structures, adapting to the specific needs of your project.

```
src/
 |- features/ // Domain-specific features
 |- auth/
 |- users/
 |- dashboard/
 |- components/ // UI components (can follow Atomic Design)
 |- common/ // Generic, reusable components
 |- Button/
 |- Card/
 |- ...
 |- domain/ // Business domain-specific components
 |- UserProfile/
 |- ProductCard/
 |- ...
 |- hooks/ // Custom hooks
 |- common/ // Generic hooks
 |- useLocalStorage.js
 |- useFetch.js
 |- ...
 |- domain/ // Domain-specific hooks
 |- useCart.js
 |- useAuth.js
 |- ...
 |- services/ // API and service interactions
 |- api.js // Base API setup
 |- authService.js
 |- userService.js
```

```
└ ...
 └ utils/ // Utility functions
 └ formatters.js
 └ validators.js
 └ ...
 └ ...
 └ contexts/ // Context providers
 └ AuthContext.js
 └ ThemeContext.js
 └ ...
 └ ...
 └ pages/ // Page components
 └ Home.js
 └ Login.js
 └ Dashboard.js
 └ ...
 └ ...
 └ routes/ // Routing configuration
 └ PrivateRoute.js
 └ routes.js
 └ ...
 └ ...
 └ styles/ // Global styles
 └ theme.js
 └ global.css
 └ ...
 └ ...
 └ config/ // Application configuration
 └ constants.js
 └ environment.js
 └ ...
 └ ...
 └ assets/ // Static assets
 └ images/
 └ icons/
 └ ...
 └ ...
 └ types/ // TypeScript types/interfaces
 └ user.types.ts
 └ product.types.ts
 └ ...
 └ ...
 └ App.js
 └ index.js
```

## Best Practices for Project Structure

1. **Be consistent:** Once you choose a structure, stick with it throughout the project. Consistency helps both existing team members and new developers navigate the codebase.

2. **Document your structure:** Provide a README that explains your project's organizational approach, especially for non-standard structures.
3. **Keep related files together:** Group files that work together closely, regardless of their type (a component, its tests, and its styles, for example).
4. **Create clear boundaries:** Features should have well-defined interfaces through their index.js files, exposing only what's necessary.
5. **Avoid too much nesting:** Deep nesting of folders makes navigation more difficult. Try to keep your structure flat where possible.
6. **Evolve gradually:** Start with a simpler structure and evolve it as the project grows. Don't overengineer from the beginning.
7. **Group by domain first, type second:** For large applications, grouping by feature/domain first provides better isolation and makes it easier to find related code.
8. **Export deliberate public APIs:** Each module/feature should have a clear public API, typically exported from an index.js file.
9. **Colocate tests with the code they test:** Keep test files next to the implementation files or in parallel test folders with matching structure.
10. **Use barrel files for cleaner imports:** Create index.js files in directories to export their contents, allowing for cleaner imports from parent directories.

## Component Organization

Regardless of the overall project structure, individual components can be organized in several ways:

### 1. Single file components:

```
// Button.js
import React from 'react';
import './Button.css';

function Button({ label, onClick }) {
 return (
 <button className="button" onClick={onClick}>
 {label}
 </button>
);
}

export default Button;
```

### 2. Component with dedicated folder:

```
Button/
├── Button.js // Component logic
├── Button.css // Component styles
├── Button.test.js // Component tests
└── ButtonTypes.js // TypeScript types/interfaces
└── index.js // Exports the component
```

```
// Button/index.js
export { default } from './Button';
export * from './Button';
export * from './ButtonTypes';

// Button/Button.js
import React from 'react';
import './Button.css';
import { ButtonProps } from './ButtonTypes';

function Button({ label, onClick }: ButtonProps) {
 // Component implementation
}

export default Button;
```

### 3. Component folder with sub-components:

```
Dropdown/
├── Dropdown.js // Main component
├── DropdownItem.js // Sub-component
├── DropdownMenu.js // Sub-component
├── DropdownToggle.js // Sub-component
├── Dropdown.css // Styles for all components
└── Dropdown.test.js // Tests for all components
└── index.js // Exports all components
```

```
// Dropdown/index.js
export { default } from './Dropdown';
export { default as DropdownItem } from './DropdownItem';
export { default as DropdownMenu } from './DropdownMenu';
export { default as DropdownToggle } from './DropdownToggle';

// Usage
import Dropdown, { DropdownItem, DropdownMenu } from './components/Dropdown';
```

## File Naming Conventions

Consistent file naming helps maintain order and clarity in your project. Choose one convention and stick to it:

1. **PascalCase**: Used for component files

```
Button.js
UserProfile.js
```

2. **camelCase**: Often used for utility functions and hooks

```
useAuth.js
formatDate.js
```

3. **kebab-case**: Sometimes used for CSS files or other assets

```
button-styles.css
user-avatar.png
```

4. **Consistent suffixes**: Add suffixes to indicate file type

```
Button.component.js
auth.service.js
theme.context.js
useLocalStorage.hook.js
```

## Import Organization

For better maintainability, organize your imports consistently:

```
// 1. External libraries
import React, { useState, useEffect } from 'react';
import PropTypes from 'prop-types';
import { useHistory } from 'react-router-dom';

// 2. Internal modules (absolute imports)
import { useAuth } from 'hooks/useAuth';
import { formatDate } from 'utils/formatters';

// 3. Components
import Button from 'components/Button';
import Card from 'components/Card';

// 4. Assets and styles
import './UserProfile.css';
import userIcon from 'assets/icons/user.svg';
```

```
// 5. Types (for TypeScript)
import { User } from 'types/user.types';

function UserProfile({ userId }) {
 // Component implementation
}

export default UserProfile;
```

## Build and Deployment Workflows

### Conceptual Foundations

The build and deployment process transforms your React source code into optimized production-ready assets, then makes them available to users. A well-designed workflow ensures reliable deliveries, maintains quality, and provides a smooth development experience.

**Mental Model:** Think of build and deployment like preparing and delivering a product. The build phase is manufacturing: raw materials (source code) are processed into a finished product (optimized bundle). Deployment is distribution: getting the product to consumers efficiently and reliably, with quality checks at each stage.

### Development Environment

```
Create a new React application
npx create-react-app my-app
or with TypeScript
npx create-react-app my-app --template typescript

Using Vite for faster development
npm create vite@latest my-app -- --template react
or with TypeScript
npm create vite@latest my-app -- --template react-ts

Start development server
npm start # For Create React App
or
npm run dev # For Vite

Environment variables in development
Create .env file in project root
REACT_APP_API_URL=https://dev-api.example.com # For Create React App
VITE_API_URL=https://dev-api.example.com # For Vite
```

Environment variables configuration:

```
// Using environment variables in your code
// Create React App
const apiUrl = process.env.REACT_APP_API_URL;

// Vite
const apiUrl = import.meta.env.VITE_API_URL;

// Access in components
function ApiService() {
 useEffect(() => {
 fetch(apiUrl)
 .then((response) => response.json())
 .then((data) => console.log(data));
 }, []);
}

return <div>API Service Component</div>;
}
```

## Build Process

```
Build for production
npm run build

Analyze bundle size (Create React App)
npm run build -- --stats
npx source-map-explorer build/static/js/main.*.js

Bundle analysis with Webpack Bundle Analyzer
npm install --save-dev webpack-bundle-analyzer

Configuration in webpack.config.js
const BundleAnalyzerPlugin = require('webpack-bundle-analyzer').BundleAnalyzerPlugin;

module.exports = {
 // other webpack config
 plugins: [
 new BundleAnalyzerPlugin()
]
};
```

## Customizing the build process in Create React App:

```
// Use react-app-rewired to customize without ejecting
npm install --save-dev react-app-rewired customize-cra

// Create config-overrides.js in project root
const { override, addBabelPlugin, adjustStyleLoaders } = require('customize-cra');
```

```
module.exports = override(
 // Add Babel plugins
 addBabelPlugin([
 'babel-plugin-transform-imports',
 {
 '@material-ui/core': {
 transform: '@material-ui/core/${member}',
 preventFullImport: true
 }
 }
]),
 // Adjust style loaders
 adjustStyleLoaders(({ use: [css, postcss, resolve, processor] }) => {
 css.options.modules = {
 localIdentName: '[name]__[local]___[hash:base64:5]'
 };
 })
);

// Update package.json scripts
"scripts": {
 "start": "react-app-rewired start",
 "build": "react-app-rewired build",
 "test": "react-app-rewired test"
}
```

## Continuous Integration (CI)

GitHub Actions example:

```
.github/workflows/ci.yml
name: React App CI

on:
 push:
 branches: [main, develop]
 pull_request:
 branches: [main, develop]

jobs:
 build-and-test:
 runs-on: ubuntu-latest

 steps:
 - uses: actions/checkout@v2

 - name: Set up Node.js
 uses: actions/setup-node@v2
 with:
```

```
node-version: '16'
cache: 'npm'

- name: Install dependencies
 run: npm ci

- name: Run linter
 run: npm run lint

- name: Run tests
 run: npm test -- --coverage

- name: Build
 run: npm run build

- name: Archive production artifacts
 uses: actions/upload-artifact@v2
 with:
 name: build
 path: build
```

## Deployment Options

### 1. Static hosting (Netlify, Vercel, GitHub Pages):

```
Install Netlify CLI
npm install netlify-cli -g

Deploy to Netlify
netlify deploy

For Vercel
npm install -g vercel
vercel

For GitHub Pages (in package.json)
"scripts": {
 "predeploy": "npm run build",
 "deploy": "gh-pages -d build"
}
```

netlify.toml configuration:

```
netlify.toml
[build]
 command = "npm run build"
 publish = "build"

Handle client-side routing
```

```
[[redirects]]
from = "/"
to = "/index.html"
status = 200
```

## 2. Docker deployment:

```
Dockerfile
Build stage
FROM node:16-alpine as build
WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .
RUN npm run build

Production stage
FROM nginx:stable-alpine
COPY --from=build /app/build /usr/share/nginx/html
COPY nginx.conf /etc/nginx/conf.d/default.conf
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

nginx.conf for client-side routing:

```
server {
 listen 80;

 location / {
 root /usr/share/nginx/html;
 index index.html index.htm;
 try_files $uri /index.html;
 }
}
```

Build and run Docker container:

```
Build the Docker image
docker build -t my-react-app .

Run the container
docker run -p 8080:80 my-react-app
```

## 3. AWS Deployment (S3 + CloudFront):

```
Install AWS CLI
pip install awscli

Configure AWS credentials
aws configure

Build the app
npm run build

Deploy to S3
aws s3 sync build/ s3://my-app-bucket --delete

Create CloudFront invalidation
aws cloudfront create-invalidation --distribution-id DISTRIBUTION_ID --paths "/*"
```

## Environment-Specific Builds

```
Create environment-specific .env files
.env # Default values
.env.development # Development overrides
.env.test # Test environment
.env.production # Production settings

Environment-specific builds
NODE_ENV=production npm run build
```

For more complex setups using different APIs per environment:

```
// config.js
const configs = {
 development: {
 apiUrl: 'https://dev-api.example.com',
 featureFlags: {
 newFeature: true,
 },
 },
 staging: {
 apiUrl: 'https://staging-api.example.com',
 featureFlags: {
 newFeature: true,
 },
 },
 production: {
 apiUrl: 'https://api.example.com',
 featureFlags: {
 newFeature: false,
 },
 },
};
```

```
};

const env = process.env.REACT_APP_ENV || 'development';
export default configs[env];

// Usage in components
import config from './config';

function ApiService() {
 useEffect(() => {
 fetch(config.apiUrl).then((response) => response.json());
 }, []);
}

return <div>{config.featureFlags.newFeature && <NewFeatureComponent />}</div>;
}
```

## Continuous Deployment (CD)

GitHub Actions example for CD to Netlify:

```
.github/workflows/deploy.yml
name: Deploy to Netlify

on:
 push:
 branches: [main]

jobs:
 deploy:
 runs-on: ubuntu-latest

 steps:
 - uses: actions/checkout@v2

 - name: Set up Node.js
 uses: actions/setup-node@v2
 with:
 node-version: '16'
 cache: 'npm'

 - name: Install dependencies
 run: npm ci

 - name: Run tests
 run: npm test

 - name: Build
 run: npm run build
 env:
 REACT_APP_API_URL: ${{ secrets.REACT_APP_API_URL }}
```

```
- name: Deploy to Netlify
 uses: netlify/actions/cli@master
 env:
 NETLIFY_AUTH_TOKEN: ${{ secrets.NETLIFY_AUTH_TOKEN }}
 NETLIFY_SITE_ID: ${{ secrets.NETLIFY_SITE_ID }}
 with:
 args: deploy --dir=build --prod
```

## Progressive Web Apps (PWA)

Create React App PWA configuration:

```
// Enable PWA in Create React App
// Set to true in src/index.js
serviceWorkerRegistration.register();

// Customize the service worker in public/manifest.json
{
 "short_name": "My App",
 "name": "My React Application",
 "icons": [
 {
 "src": "favicon.ico",
 "sizes": "64x64 32x32 24x24 16x16",
 "type": "image/x-icon"
 },
 {
 "src": "logo192.png",
 "type": "image/png",
 "sizes": "192x192"
 },
 {
 "src": "logo512.png",
 "type": "image/png",
 "sizes": "512x512"
 }
],
 "start_url": ".",
 "display": "standalone",
 "theme_color": "#000000",
 "background_color": "#ffffff"
}

// Add custom service worker logic
// src/service-worker.js in Create React App
self.addEventListener('fetch', (event) => {
 if (event.request.url.includes('/api/')) {
 // Custom caching strategy for API calls
 event.respondWith(
 caches.open('api-cache').then((cache) => {
 return fetch(event.request)
```

```
.then((response) => {
 cache.put(event.request, response.clone());
 return response;
})
.catch(() => {
 return cache.match(event.request);
});
})
);
}
});
```

## Multiple Environments with Feature Flags

```
// Feature flag service
const featureFlagService = {
 flags: {},

 async init() {
 try {
 // Fetch flags from backend
 const response = await fetch('/api/feature-flags');
 this.flags = await response.json();
 } catch (error) {
 console.error('Failed to fetch feature flags:', error);
 // Fallback to default flags
 this.flags = {
 newDesign: false,
 betaFeatures: false,
 };
 }
 },
 isEnabled(flagName) {
 return this.flags[flagName] === true;
 },
};

// Feature flag context
const FeatureFlagContext = createContext({
 isEnabled: () => false,
});

function FeatureFlagProvider({ children }) {
 const [isLoading, setIsLoading] = useState(true);
 const [service, setService] = useState({
 isEnabled: () => false,
 });
 useEffect(() => {
 const loadFlags = async () => {
```

```
 await featureFlagService.init();
 setService(featureFlagService);
 setIsLoading(false);
}

loadFlags();
}, []));

if (isLoading) {
 return <div>Loading application...</div>;
}

return (
 <FeatureFlagContext.Provider value={service}>
 {children}
 </FeatureFlagContext.Provider>
);
}

// Using feature flags in components
function MyComponent() {
 const featureFlags = useContext(FeatureFlagContext);

 return (
 <div>
 {featureFlags.isEnabled('newDesign') ? (
 <NewDesignComponent />
) : (
 <LegacyDesignComponent />
)}

 {featureFlags.isEnabled('betaFeatures') && <BetaFeatureSection />}
 </div>
);
}
}
```

## Modern Development Environment Setup

### Essential Development Tools

#### 1. Node.js and npm/yarn

```
Check Node.js version
node -v

Install specific Node.js version with NVM (Node Version Manager)
nvm install 16
nvm use 16

Or use Volta for Node.js version management
volta install node@16
```

```
volta install npm@8

Initialize a new project
npm init -y

Use yarn instead of npm (optional)
npm install -g yarn
yarn init -y
```

## 2. Code Editor: VS Code Configuration

Essential VS Code extensions for React development:

```
// VS Code extensions.json
{
 "recommendations": [
 "dbaeumer.vscode-eslint",
 "esbenp.prettier-vscode",
 "dsznajder.es7-react-js-snippets",
 "formulahendry.auto-rename-tag",
 "christian-kohler.path-intellisense",
 "streetsidesoftware.code-spell-checker",
 "aaron-bond.better-comments",
 "wix.vscode-import-cost",
 "eamodio.gitlens"
]
}
```

VS Code workspace settings:

```
// .vscode/settings.json
{
 "editor.formatOnSave": true,
 "editor.defaultFormatter": "esbenp.prettier-vscode",
 "editor.codeActionsOnSave": {
 "source.fixAll.eslint": true
 },
 "javascript.updateImportsOnFileMove.enabled": "always",
 "typescript.updateImportsOnFileMove.enabled": "always",
 "emmet.includeLanguages": {
 "javascript": "javascriptreact",
 "typescript": "typescriptreact"
 }
}
```

## 3. Linting and Formatting

ESLint configuration:

```
// .eslintrc.js
module.exports = {
 env: {
 browser: true,
 es2021: true,
 node: true,
 jest: true,
 },
 extends: [
 'eslint:recommended',
 'plugin:react/recommended',
 'plugin:react-hooks/recommended',
 'plugin:jsx-a11y/recommended',
 'plugin:import/errors',
 'plugin:import/warnings',
 'prettier',
],
 parserOptions: {
 ecmaFeatures: {
 jsx: true,
 },
 ecmaVersion: 12,
 sourceType: 'module',
 },
 plugins: ['react', 'react-hooks', 'jsx-a11y', 'import'],
 rules: {
 'react/react-in-jsx-scope': 'off', // Not needed in React 17+
 'react/prop-types': 'warn',
 'no-console': ['warn', { allow: ['warn', 'error'] }],
 'no-unused-vars': 'warn',
 'import/order': [
 'error',
 {
 groups: [
 'builtin',
 'external',
 'internal',
 'parent',
 'sibling',
 'index',
],
 'newlines-between': 'always',
 alphabetize: { order: 'asc', caseInsensitive: true },
 },
],
 },
 settings: {
 react: {
 version: 'detect',
 },
 'import/resolver': {
 node: {
 extensions: ['.js', '.jsx', '.ts', '.tsx'],
 }
 }
 }
}
```

```
 moduleDirectory: ['node_modules', 'src'],
 },
},
},
};
```

Prettier configuration:

```
// .prettierrc
{
 "singleQuote": true,
 "trailingComma": "es5",
 "tabWidth": 2,
 "semi": true,
 "printWidth": 80,
 "bracketSpacing": true,
 "arrowParens": "avoid",
 "jsxBracketSameLine": false
}
```

Integrating ESLint and Prettier:

```
Install ESLint and Prettier
npm install --save-dev eslint prettier

Install ESLint plugins and configs
npm install --save-dev eslint-plugin-react eslint-plugin-react-hooks eslint-
plugin-javascript-a11y eslint-plugin-import

Install Prettier integration with ESLint
npm install --save-dev eslint-config-prettier eslint-plugin-prettier

Add scripts to package.json
"scripts": {
 "lint": "eslint --ext .js,.jsx,.ts,.tsx src/",
 "lint:fix": "eslint --ext .js,.jsx,.ts,.tsx src/ --fix",
 "format": "prettier --write \"src/**/*.{js,jsx,ts,tsx,json,css,scss,md}\""
}
```

#### 4. Git Configuration

Gitignore file:

```
.gitignore
dependencies
/node_modules
/.pnp
```

```
.pnp.js

testing
/coverage

production
/build
/dist

misc
.DS_Store
.env.local
.env.development.local
.env.test.local
.env.production.local

npm-debug.log*
yarn-debug.log*
yarn-error.log*

Editor directories and files
.idea
.vscode/*
!.vscode/extensions.json
!.vscode/settings.json
*.suo
.ntvs
*.njsproj
*.sln
*.sw?
```

Git hooks with Husky:

```
Install Husky and lint-staged
npm install --save-dev husky lint-staged

Set up husky
npx husky install
npx husky add .husky/pre-commit "npx lint-staged"
```

Configuration for lint-staged:

```
// package.json
{
 "lint-staged": {
 "*.{js,jsx,ts,tsx)": ["eslint --fix", "prettier --write"],
 "*.{json,css,scss,md)": ["prettier --write"]
 }
}
```

Commitlint for enforcing commit message conventions:

```
Install commitlint
npm install --save-dev @commitlint/cli @commitlint/config-conventional

Create config file
echo "module.exports = {extends: ['@commitlint/config-conventional']}" >
commitlint.config.js

Add commit-msg hook
npx husky add .husky/commit-msg 'npx --no-interactive commitlint --edit "$1"'
```

## Project Templates and Bootstrapping

### 1. Create React App

```
Basic CRA
npx create-react-app my-app

With TypeScript
npx create-react-app my-app --template typescript

With Redux
npx create-react-app my-app --template redux

With Redux + TypeScript
npx create-react-app my-app --template redux-typescript
```

### 2. Vite

```
Create Vite project
npx create vite@latest my-app -- --template react

With TypeScript
npx create vite@latest my-app -- --template react-ts

Install dependencies
cd my-app
npm install
```

### 3. Next.js

```
Create Next.js app
npx create-next-app my-app
```

```
With TypeScript
npx create-next-app@latest --ts my-app

Start development server
npm run dev
```

## Setting Up a Comprehensive React Project

```
Create a project with Vite
npm create vite@latest my-app -- --template react-ts
cd my-app

Install base dependencies
npm install

Install UI library
npm install @chakra-ui/react @emotion/react @emotion/styled framer-motion

Install routing
npm install react-router-dom

Install state management
npm install zustand

Install form handling
npm install react-hook-form zod @hookform/resolvers

Install testing tools
npm install --save-dev vitest @testing-library/react @testing-library/jest-dom
@testing-library/user-event jsdom

Install utilities
npm install axios date-fns lodash

Install dev tools
npm install --save-dev eslint prettier eslint-config-prettier eslint-plugin-prettier
eslint-plugin-react eslint-plugin-react-hooks eslint-plugin-jsx-a11y
eslint-plugin-import husky lint-staged
```

## Setup testing configuration for Vite:

```
// vitest.config.js
import { defineConfig } from 'vitest/config';
import react from '@vitejs/plugin-react';

export default defineConfig({
 plugins: [react()],
 test: {
 globals: true,
```

```
 environment: 'jsdom',
 setupFiles: './src/test/setup.js',
 },
});

// src/test/setup.js
import '@testing-library/jest-dom';
```

Add test script to package.json:

```
"scripts": {
 "dev": "vite",
 "build": "tsc && vite build",
 "preview": "vite preview",
 "test": "vitest",
 "test:coverage": "vitest run --coverage",
 "lint": "eslint --ext .js,.jsx,.ts,.tsx src/",
 "lint:fix": "eslint --ext .js,.jsx,.ts,.tsx src/ --fix",
 "format": "prettier --write \"src/**/*.{js,jsx,ts,tsx,json,css,scss,md}\""
}
```

## Developer Workflow Optimization

### 1. Debugging Tools

React Developer Tools browser extension:

```
// Debugging with React Developer Tools
// In your React component, add:
function MyComponent() {
 // Add a debugger statement where you want to pause
 debugger;

 return <div>Component Content</div>;
}
```

Using React DevTools profiler:

1. Open the React Developer Tools extension
  2. Click on the Profiler tab
  3. Click the Record button
  4. Interact with your app
  5. Stop recording to analyze render times
- 6. Browser Extensions for React Development**

- React Developer Tools (Chrome/Firefox)
- Redux DevTools (for Redux)
- Apollo Client DevTools (for GraphQL)
- Axe DevTools (for accessibility testing)

### 3. Virtual Environment Consistency

```
Set up Docker development environment
Create Dockerfile
FROM node:16-alpine

WORKDIR /app

COPY package*.json ./
RUN npm install

COPY . .

EXPOSE 3000

CMD ["npm", "start"]

Create docker-compose.yml
version: '3'
services:
 app:
 build: .
 ports:
 - "3000:3000"
 volumes:
 - .:/app
 - /app/node_modules
```

### 4. Automated Testing Setup

```
// Component testing example with React Testing Library
// Button.test.jsx
import { render, screen, fireEvent } from '@testing-library/react';
import Button from './Button';

describe('Button', () => {
 test('renders correctly', () => {
 render(<Button label="Click me" />);
 expect(screen.getByText('Click me')).toBeInTheDocument();
 });

 test('calls onClick handler when clicked', () => {
 const handleClick = vi.fn();
 render(<Button label="Click me" onClick={handleClick} />);
 });
});
```

```
fireEvent.click(screen.getByText('Click me'));

expect(handleClick).toHaveBeenCalledTimes(1);
});

});

// Integration testing with mock service worker
// src/mocks/handlers.js
import { rest } from 'msw';

export const handlers = [
 rest.get('https://api.example.com/users', (req, res, ctx) => {
 return res(
 ctx.status(200),
 ctx.json([
 { id: 1, name: 'John Doe' },
 { id: 2, name: 'Jane Smith' },
])
);
 }),
];
];
```

## 5. VS Code Snippets for React

```
// .vscode/react.code-snippets
{
 "React Functional Component": {
 "prefix": "rfc",
 "body": [
 "import React from 'react';",
 "",
 "function ${1:ComponentName}() {",
 " return (",
 " <div>",
 " $0",
 " </div>",
 ");",
 "}",
 "",
 "export default ${1:ComponentName};"
],
 "description": "React Functional Component"
 },
 "React Functional Component with Props": {
 "prefix": "rfcp",
 "body": [
 "import React from 'react';",
 "",
 "function ${1:ComponentName}(${2:props}) {",
 " return (",
 " <div>",
 " $0",
 " </div>",
 ");"
],
 "description": "React Functional Component with Props"
 }
};
```

```
" </div>",
);
}",
"",
"export default ${1:ComponentName};"
],
"description": "React Functional Component with Props"
},
"useState Hook": {
"prefix": "usestate",
"body": [
"const [${1:state}, useState] = useState(${2:initialState});"
],
"description": "React useState Hook"
},
"useEffect Hook": {
"prefix": "useeffect",
"body": ["useEffect(() => {", " $0", "}, [${1:dependencies}]);"],
"description": "React useEffect Hook"
}
}
```

## Important Tooling in the React Ecosystem

### State Management Libraries

#### 1. Redux

```
Install Redux
npm install redux react-redux redux-thunk @reduxjs/toolkit
```

Basic Redux Toolkit setup:

```
// store/slices/counterSlice.js
import { createSlice } from '@reduxjs/toolkit';

const counterSlice = createSlice({
 name: 'counter',
 initialState: {
 value: 0,
 },
 reducers: {
 increment: (state) => {
 state.value += 1;
 },
 decrement: (state) => {
 state.value -= 1;
 },
 }
});
```

```
incrementByAmount: (state, action) => {
 state.value += action.payload;
},
},
});

export const { increment, decrement, incrementByAmount } = counterSlice.actions;
export default counterSlice.reducer;

// store/index.js
import { configureStore } from '@reduxjs/toolkit';
import counterReducer from './slices/counterSlice';

export const store = configureStore({
 reducer: {
 counter: counterReducer,
 },
});

// App.js
import { Provider } from 'react-redux';
import { store } from './store';
import Counter from './components/Counter';

function App() {
 return (
 <Provider store={store}>
 <div className="App">
 <Counter />
 </div>
 </Provider>
);
}

// Counter.js
import { useSelector, useDispatch } from 'react-redux';
import { increment, decrement } from '../store/slices/counterSlice';

function Counter() {
 const count = useSelector((state) => state.counter.value);
 const dispatch = useDispatch();

 return (
 <div>
 <h2>Count: {count}</h2>
 <button onClick={() => dispatch(increment())}>+</button>
 <button onClick={() => dispatch(decrement())}>-</button>
 </div>
);
}
```

## 2. Zustand

```
Install Zustand
npm install zustand
```

Basic Zustand setup:

```
// store/useStore.js
import create from 'zustand';

const useStore = create((set) => ({
 count: 0,
 increment: () => set((state) => ({ count: state.count + 1 })),
 decrement: () => set((state) => ({ count: state.count - 1 })),
 reset: () => set({ count: 0 }),
}));

export default useStore;

// Counter.js
import useStore from '../store/useStore';

function Counter() {
 const { count, increment, decrement, reset } = useStore();

 return (
 <div>
 <h2>Count: {count}</h2>
 <button onClick={increment}>+</button>
 <button onClick={decrement}>-</button>
 <button onClick={reset}>Reset</button>
 </div>
);
}

}
```

### 3. Recoil

```
Install Recoil
npm install recoil
```

Basic Recoil setup:

```
// store/atoms.js
import { atom, selector } from 'recoil';

export const countState = atom({
 key: 'countState',
 default: 0,
```

```
});

export const countDoubleState = selector({
 key: 'countDoubleState',
 get: ({ get }) => {
 const count = get(countState);
 return count * 2;
 },
});

// App.js
import { RecoilRoot } from 'recoil';
import Counter from './components/Counter';

function App() {
 return (
 <RecoilRoot>
 <div className="App">
 <Counter />
 </div>
 </RecoilRoot>
);
}

// Counter.js
import { useRecoilState, useRecoilValue } from 'recoil';
import { countState, countDoubleState } from '../store/atoms';

function Counter() {
 const [count, setCount] = useRecoilState(countState);
 const doubleCount = useRecoilValue(countDoubleState);

 return (
 <div>
 <h2>Count: {count}</h2>
 <h3>Double Count: {doubleCount}</h3>
 <button onClick={() => setCount(count + 1)}>+</button>
 <button onClick={() => setCount(count - 1)}>-</button>
 <button onClick={() => setCount(0)}>Reset</button>
 </div>
);
}
```

## UI Component Libraries

### 1. Material UI

```
Install Material UI
npm install @mui/material @mui/icons-material @emotion/react @emotion/styled
```

## Basic Material UI usage:

```
// App.js
import { ThemeProvider, createTheme } from '@mui/material/styles';
import CssBaseline from '@mui/material/CssBaseline';
import Button from '@mui/material/Button';
import { deepPurple, amber } from '@mui/material/colors';

// Custom theme
const theme = createTheme({
 palette: {
 primary: deepPurple,
 secondary: amber,
 mode: 'light', // 'dark' for dark mode
 },
});

function App() {
 return (
 <ThemeProvider theme={theme}>
 <CssBaseline /> {/* Normalizes styles */}
 <div className="App">
 <Button variant="contained" color="primary">
 Primary Button
 </Button>
 <Button variant="outlined" color="secondary">
 Secondary Button
 </Button>
 </div>
 </ThemeProvider>
);
}

}
```

## 2. Chakra UI

```
Install Chakra UI
npm install @chakra-ui/react @emotion/react @emotion/styled framer-motion
```

## Basic Chakra UI usage:

```
// App.js
import { ChakraProvider, extendTheme } from '@chakra-ui/react';
import { Button, Box, Heading, Stack } from '@chakra-ui/react';

// Custom theme
const theme = extendTheme({
 colors: {
 brand: {

```

```
 100: '#f7fafc',
 900: '#1a202c',
 },
},
});

function App() {
 return (
 <ChakraProvider theme={theme}>
 <Box p={4}>
 <Heading mb={4}>My Chakra App</Heading>
 <Stack direction="row" spacing={4}>
 <Button colorScheme="blue">Button</Button>
 <Button colorScheme="green" variant="outline">
 Button
 </Button>
 </Stack>
 </Box>
 </ChakraProvider>
);
}
```

### 3. Tailwind CSS

```
Install Tailwind CSS
npm install -D tailwindcss postcss autoprefixer
npx tailwindcss init -p
```

Tailwind configuration:

```
// tailwind.config.js
module.exports = {
 content: [
 "./src/**/*.{js,jsx,ts,tsx}",
],
 theme: {
 extend: {
 colors: {
 primary: '#3490dc',
 secondary: '#ffed4a',
 danger: '#e3342f',
 },
 },
 },
 plugins: [],
}

// src/index.css
@tailwind base;
```

```
@tailwind components;
@tailwind utilities;

// Custom component styles
@layer components {
 .btn-primary {
 @apply bg-primary text-white py-2 px-4 rounded hover:bg-primary/80;
 }
}
```

Basic Tailwind usage:

```
function App() {
 return (
 <div className="container mx-auto p-4">
 <h1 className="text-3xl font-bold mb-4 text-primary">My Tailwind App</h1>
 <div className="flex space-x-4">
 <button className="btn-primary">Primary Button</button>
 <button className="bg-secondary text-gray-800 py-2 px-4 rounded hover:bg-secondary/80">
 Secondary Button
 </button>
 </div>
 </div>
);
}
```

## Form Handling Libraries

### 1. React Hook Form

```
Install React Hook Form
npm install react-hook-form
```

Basic React Hook Form usage:

```
import { useForm } from 'react-hook-form';

function LoginForm() {
 const {
 register,
 handleSubmit,
 formState: { errors },
 } = useForm();

 const onSubmit = (data) => {
 console.log(data);
}
```

```
// Submit to server...
};

return (
 <form onSubmit={handleSubmit(onSubmit)}>
 <div>
 <label htmlFor="email">Email</label>
 <input
 id="email"
 {...register('email', {
 required: 'Email is required',
 pattern: {
 value: /\S+@\S+\.\S+/,
 message: 'Invalid email format',
 },
))}
 />
 {errors.email && <p className="error">{errors.email.message}</p>}
 </div>

 <div>
 <label htmlFor="password">Password</label>
 <input
 id="password"
 type="password"
 {...register('password', {
 required: 'Password is required',
 minLength: {
 value: 6,
 message: 'Password must be at least 6 characters',
 },
))}
 />
 {errors.password && <p className="error">{errors.password.message}</p>}
 </div>

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

## 2. Formik with Yup

```
Install Formik and Yup
npm install formik yup
```

Basic Formik usage:

```
import { Formik, Form, Field, ErrorMessage } from 'formik';
import * as Yup from 'yup';

// Validation schema
const LoginSchema = Yup.object().shape({
 email: Yup.string().email('Invalid email').required('Email is required'),
 password: Yup.string()
 .min(6, 'Password must be at least 6 characters')
 .required('Password is required'),
});

function LoginForm() {
 return (
 <Formik
 initialValues={{ email: '', password: '' }}
 validationSchema={LoginSchema}
 onSubmit={(values, { setSubmitting }) => {
 setTimeout(() => {
 console.log(values);
 // Submit to server...
 setSubmitting(false);
 }, 400);
 }}
 >
 {({ isSubmitting }) => (
 <Form>
 <div>
 <label htmlFor="email">Email</label>
 <Field id="email" name="email" type="email" />
 <ErrorMessage name="email" component="p" className="error" />
 </div>

 <div>
 <label htmlFor="password">Password</label>
 <Field id="password" name="password" type="password" />
 <ErrorMessage name="password" component="p" className="error" />
 </div>

 <button type="submit" disabled={isSubmitting}>
 {isSubmitting ? 'Submitting...' : 'Login'}
 </button>
 </Form>
)}
 </Formik>
);
}
```

### 3. Integrating React Hook Form with Zod

```
Install React Hook Form and Zod
npm install react-hook-form zod @hookform/resolvers
```

React Hook Form with Zod schema validation:

```
import { useForm } from 'react-hook-form';
import { zodResolver } from '@hookform/resolvers/zod';
import { z } from 'zod';

// Define schema with Zod
const loginSchema = z.object({
 email: z.string()
 .email('Invalid email format')
 .min(1, 'Email is required'),
 password: z.string()
 .min(6, 'Password must be at least 6 characters')
 .min(1, 'Password is required'),
});

// Infer TypeScript type from schema
type LoginFormData = z.infer<typeof loginSchema>;

function LoginForm() {
 const { register, handleSubmit, formState: { errors } } = useForm<LoginFormData>(
{
 resolver: zodResolver(loginSchema)
});

 const onSubmit = (data: LoginFormData) => {
 console.log(data);
 // Submit to server...
 };

 return (
 <form onSubmit={handleSubmit(onSubmit)}>
 <div>
 <label htmlFor="email">Email</label>
 <input id="email" {...register('email')} />
 {errors.email && <p className="error">{errors.email.message}</p>}
 </div>

 <div>
 <label htmlFor="password">Password</label>
 <input
 id="password"
 type="password"
 {...register('password')}
 />
 {errors.password && <p className="error">{errors.password.message}</p>}
 </div>

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

```
);
 }
```

## Data Fetching Libraries

### 1. React Query (TanStack Query)

```
Install React Query
npm install @tanstack/react-query
```

Basic React Query setup:

```
// App.js
import { QueryClient, QueryClientProvider } from '@tanstack/react-query';
import { ReactQueryDevtools } from '@tanstack/react-query-devtools';
import UserList from './components/UserList';

// Create a client
const queryClient = new QueryClient({
 defaultOptions: {
 queries: {
 refetchOnWindowFocus: false,
 retry: 1,
 },
 },
});

function App() {
 return (
 <QueryClientProvider client={queryClient}>
 <div className="App">
 <h1>React Query Example</h1>
 <UserList />
 </div>
 <ReactQueryDevtools initialIsOpen={false} />
 </QueryClientProvider>
);
}

// UserList.js
import { useQuery } from '@tanstack/react-query';

// API functions
const fetchUsers = async () => {
 const response = await fetch('https://api.example.com/users');
 if (!response.ok) {
 throw new Error('Network response was not ok');
 }
 return response.json();
```

```

};

function UserList() {
 const {
 data: users,
 isLoading,
 isError,
 error,
 refetch
 } = useQuery({
 queryKey: ['users'],
 queryFn: fetchUsers
 });

 if (isLoading) return <div>Loading users...</div>;
 if (isError) return <div>Error: {error.message}</div>

 return (
 <div>
 <button onClick={() => refetch()}>Refresh Users</button>

 {users.map(user => (
 <li key={user.id}>{user.name}
))}

 </div>
);
}

```

Mutations with React Query:

```

import { useMutation, useQueryClient } from '@tanstack/react-query';

// API function
const addUser = async (newUser) => {
 const response = await fetch('https://api.example.com/users', {
 method: 'POST',
 headers: {
 'Content-Type': 'application/json',
 },
 body: JSON.stringify(newUser),
 });

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

 return response.json();
};

function AddUserForm() {
 const queryClient = useQueryClient();

```

```

const [name, setName] = useState('');

const mutation = useMutation({
 mutationFn: addUser,
 onSuccess: () => {
 // Invalidate and refetch
 queryClient.invalidateQueries({ queryKey: ['users'] });
 setName(''); // Reset form
 },
});

const handleSubmit = (e) => {
 e.preventDefault();
 mutation.mutate({ name });
};

return (
 <form onSubmit={handleSubmit}>
 <input
 type="text"
 value={name}
 onChange={e => setName(e.target.value)}
 placeholder="User name"
 />
 <button
 type="submit"
 disabled={mutation.isPending}
 >
 {mutation.isPending ? 'Adding...' : 'Add User'}
 </button>
 {mutation.isError && (
 <div>Error: {mutation.error.message}</div>
)}
 </form>
);
}

```

## 2. SWR (Stale-While-Revalidate)

```

Install SWR
npm install swr

```

Basic SWR usage:

```

import useSWR from 'swr';

// Fetcher function
const fetcher = async (url) => {
 const response = await fetch(url);

```

```

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

function UserProfile({ userId }) {
 const { data, error, isLoading, mutate } = useSWR(
 `https://api.example.com/users/${userId}`,
 fetcher
);

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

 return (
 <div>
 <h2>{data.name}</h2>
 <p>Email: {data.email}</p>
 <button onClick={() => mutate()}>Refresh</button>
 </div>
);
}

```

Global SWR configuration:

```

// App.js
import { SWRConfig } from 'swr';

function App() {
 return (
 <SWRConfig
 value={{
 fetcher: (url) => fetch(url).then(res => res.json()),
 revalidateOnFocus: false,
 revalidateOnReconnect: true,
 refreshInterval: 0,
 shouldRetryOnError: false
 }}
 >
 <div className="App">
 <UserProfile userId="123" />
 </div>
 </SWRConfig>
);
}

```

### 3. Apollo Client (for GraphQL)

```
Install Apollo Client
npm install @apollo/client graphql
```

Basic Apollo Client setup:

```
// App.js
import { ApolloClient, InMemoryCache, ApolloProvider, gql } from '@apollo/client';

// Create Apollo client
const client = new ApolloClient({
 uri: 'https://api.example.com/graphql',
 cache: new InMemoryCache()
});

function App() {
 return (
 <ApolloProvider client={client}>
 <div className="App">
 <h1>Apollo Client Example</h1>
 <UserList />
 </div>
 </ApolloProvider>
);
}

// UserList.js
import { useQuery, gql } from '@apollo/client';

// GraphQL query
const GET_USERS = gql`
query GetUsers {
 users {
 id
 name
 email
 }
}
`;

function UserList() {
 const { loading, error, data } = useQuery(GET_USERS);

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

 return (

 {data.users.map(user => (
 <li key={user.id}>{user.name} ({user.email})
))}

);
}
```

```
);
 }
```

## GraphQL mutations with Apollo:

```
import { useMutation, gql } from '@apollo/client';

const ADD_USER = gql`
mutation AddUser($name: String!, $email: String!) {
 addUser(name: $name, email: $email) {
 id
 name
 email
 }
}
`;

function AddUserForm() {
 const [name, setName] = useState('');
 const [email, setEmail] = useState('');

 const [addUser, { loading, error }] = useMutation(ADD_USER, {
 update(cache, { data: { addUser } }) {
 cache.modify({
 fields: {
 users(existingUsers = []) {
 const newUserRef = cache.writeFragment({
 data: addUser,
 fragment: gql`
 fragment NewUser on User {
 id
 name
 email
 }
 `
 });
 return [...existingUsers, newUserRef];
 }
 }
 });
 }
 });
}

const handleSubmit = (e) => {
 e.preventDefault();
 addUser({ variables: { name, email } });
 setName('');
 setEmail('');
};

return (
 <form onSubmit={handleSubmit}>
```

```
<div>
 <input
 placeholder="Name"
 value={name}
 onChange={e => setName(e.target.value)}
 />
</div>
<div>
 <input
 placeholder="Email"
 value={email}
 onChange={e => setEmail(e.target.value)}
 />
</div>
<button type="submit" disabled={loading}>
 {loading ? 'Adding...' : 'Add User'}
</button>
{error && <div>Error: {error.message}</div>}
</form>
);
}
```

## Animation Libraries

### 1. Framer Motion

```
Install Framer Motion
npm install framer-motion
```

Basic animations with Framer Motion:

```
import { motion } from 'framer-motion';

function AnimatedBox() {
 return (
 <motion.div
 initial={{ opacity: 0, scale: 0.5 }}
 animate={{ opacity: 1, scale: 1 }}
 transition={{ duration: 0.5 }}
 style={{
 width: 100,
 height: 100,
 backgroundColor: 'blue',
 borderRadius: 10
 }}
 />
);
}
```

```
// With hover and tap animations
function InteractiveButton() {
 return (
 <motion.button
 initial={{ opacity: 0 }}
 animate={{ opacity: 1 }}
 whileHover={{
 scale: 1.1,
 backgroundColor: '#ff0055',
 transition: { duration: 0.2 }
 }}
 whileTap={{ scale: 0.95 }}
 style={{
 padding: '10px 20px',
 backgroundColor: '#0099ff',
 color: 'white',
 border: 'none',
 borderRadius: 5
 }}
 >
 Click Me
 </motion.button>
);
}
```

Animating a list of items:

```
import { motion, AnimatePresence } from 'framer-motion';

function TodoList({ items }) {
 return (

 <AnimatePresence>
 {items.map(item => (
 <motion.li
 key={item.id}
 initial={{ opacity: 0, height: 0 }}
 animate={{ opacity: 1, height: 'auto' }}
 exit={{ opacity: 0, height: 0 }}
 transition={{ duration: 0.3 }}
 >
 {item.text}
 </motion.li>
))}
 </AnimatePresence>

);
}

// Page transitions
function PageTransition({ children }) {
 return (
```

```
<motion.div
 initial={{ opacity: 0, x: -200 }}
 animate={{ opacity: 1, x: 0 }}
 exit={{ opacity: 0, x: 200 }}
 transition={{ duration: 0.5 }}
>
 {children}
</motion.div>
);
}
```

## 2. React Spring

```
Install React Spring
npm install @react-spring/web
```

Basic React Spring usage:

```
import { useSpring, animated } from '@react-spring/web';

function AnimatedComponent() {
 const [springs, api] = useSpring(() => ({
 from: { opacity: 0, y: 100 },
 to: { opacity: 1, y: 0 },
 config: { mass: 1, tension: 280, friction: 60 }
 }));

 return (
 <animated.div
 style={{
 ...springs,
 backgroundColor: 'blue',
 width: 100,
 height: 100,
 borderRadius: 8
 }}
 />
);
}

// Interactive animation
function InteractiveCard() {
 const [flipped, setFlipped] = useState(false);

 const { transform, opacity } = useSpring({
 opacity: flipped ? 1 : 0,
 transform: `perspective(600px) rotateX(${flipped ? 180 : 0}deg)`,
 config: { mass: 5, tension: 500, friction: 80 }
 });
}
```

```

return (
 <div onClick={() => setFlipped(!flipped)}>
 <animated.div
 style={{
 opacity: opacity.to(o => 1 - o),
 transform,
 backgroundColor: 'coral',
 position: 'absolute',
 width: 200,
 height: 200,
 padding: 20,
 borderRadius: 8
 }}
 >
 Click to flip (Front)
 </animated.div>
 <animated.div
 style={{
 opacity,
 transform: transform.to(t => `${t} rotateX(180deg)`),
 backgroundColor: 'lightblue',
 position: 'absolute',
 width: 200,
 height: 200,
 padding: 20,
 borderRadius: 8
 }}
 >
 Click to flip (Back)
 </animated.div>
 </div>
);
}

```

## Testing Tools

### 1. Jest with React Testing Library

```

With Create React App, these are included
For Vite projects, install:
npm install --save-dev vitest @testing-library/react @testing-library/user-event
@testing-library/jest-dom jsdom

```

Basic component testing:

```

// Button.test.jsx
import { render, screen, fireEvent } from '@testing-library/react';
import Button from './Button';

```

```

describe('Button', () => {
 test('renders correctly', () => {
 render(<Button>Click me</Button>);
 expect(screen.getByRole('button')).toHaveTextContent('Click me');
 });

 test('handles click events', () => {
 const handleClick = jest.fn();

 render(<Button onClick={handleClick}>Click me</Button>);

 fireEvent.click(screen.getByRole('button'));

 expect(handleClick).toHaveBeenCalledTimes(1);
 });

 test('applies custom className', () => {
 render(<Button className="custom-button">Click me</Button>);
 expect(screen.getByRole('button')).toHaveClass('custom-button');
 });
});

```

Setup with Vitest for Vite projects:

```

// vitest.config.js
import { defineConfig } from 'vitest/config';
import react from '@vitejs/plugin-react';

export default defineConfig({
 plugins: [react()],
 test: {
 globals: true,
 environment: 'jsdom',
 setupFiles: './src/test/setup.js',
 },
});

// src/test/setup.js
import '@testing-library/jest-dom';

```

Testing hooks with React Testing Library:

```

// useCounter.test.js
import { renderHook, act } from '@testing-library/react';
import { useCounter } from './useCounter';

describe('useCounter', () => {
 test('should initialize with default value', () => {

```

```
const { result } = renderHook(() => useCounter());
expect(result.current.count).toBe(0);
});

test('should initialize with provided value', () => {
 const { result } = renderHook(() => useCounter(10));
 expect(result.current.count).toBe(10);
});

test('should increment counter', () => {
 const { result } = renderHook(() => useCounter());

 act(() => {
 result.current.increment();
 });

 expect(result.current.count).toBe(1);
});

test('should decrement counter', () => {
 const { result } = renderHook(() => useCounter());

 act(() => {
 result.current.decrement();
 });

 expect(result.current.count).toBe(-1);
});
```

## 2. Cypress for End-to-End Testing

```
Install Cypress
npm install --save-dev cypress
```

Add script to package.json:

```
{
 "scripts": {
 "cypress:open": "cypress open",
 "cypress:run": "cypress run"
 }
}
```

Basic Cypress test:

```
// cypress/e2e/login.cy.js
describe('Login Flow', () => {
 beforeEach(() => {
 cy.visit('/login');
 });

 it('should display login form', () => {
 cy.get('form').should('be.visible');
 cy.get('input[name="email"]').should('be.visible');
 cy.get('input[name="password"]').should('be.visible');
 cy.get('button[type="submit"]').should('be.visible');
 });

 it('should show error on invalid login', () => {
 cy.get('input[name="email"]').type('invalid@example.com');
 cy.get('input[name="password"]').type('wrongpassword');
 cy.get('button[type="submit"]').click();

 cy.get('.error-message').should('be.visible');
 cy.get('.error-message').should('contain', 'Invalid credentials');
 });

 it('should redirect to dashboard on successful login', () => {
 cy.get('input[name="email"]').type('user@example.com');
 cy.get('input[name="password"]').type('password123');
 cy.get('button[type="submit"]').click();

 // Verify redirect to dashboard
 cy.url().should('include', '/dashboard');
 cy.get('.welcome-message').should('contain', 'Welcome, User');
 });
});
```

Custom Cypress commands:

```
// cypress/support/commands.js
Cypress.Commands.add('login', (email, password) => {
 cy.visit('/login');
 cy.get('input[name="email"]').type(email);
 cy.get('input[name="password"]').type(password);
 cy.get('button[type="submit"]').click();
});

// Usage in tests
it('should display user profile after login', () => {
 cy.login('user@example.com', 'password123');
 cy.visit('/profile');
 cy.get('.profile-name').should('contain', 'User Name');
});
```

### 3. Testing with Mock Service Worker (MSW)

```
Install MSW for API mocking
npm install --save-dev msw
```

Setting up MSW:

```
// src/mocks/handlers.js
import { rest } from 'msw';

export const handlers = [
 // Mock user login endpoint
 rest.post('/api/login', (req, res, ctx) => {
 const { email, password } = req.body;

 if (email === 'user@example.com' && password === 'password123') {
 return res(
 ctx.status(200),
 ctx.json({
 id: '123',
 name: 'Test User',
 email: 'user@example.com',
 token: 'fake-jwt-token'
 })
);
 }

 return res(
 ctx.status(401),
 ctx.json({ message: 'Invalid credentials' })
);
 }),

 // Mock users endpoint
 rest.get('/api/users', (req, res, ctx) => {
 return res(
 ctx.status(200),
 ctx.json([
 { id: '1', name: 'John Doe', email: 'john@example.com' },
 { id: '2', name: 'Jane Smith', email: 'jane@example.com' }
])
);
 })
];

// src/mocks/server.js
import { setupServer } from 'msw/node';
import { handlers } from './handlers';

export const server = setupServer(...handlers);
```

```
// src/setupTests.js
import { server } from './mocks/server';

// Start server before all tests
beforeAll(() => server.listen());

// Reset handlers after each test
afterEach(() => server.resetHandlers());

// Close server after all tests
afterAll(() => server.close());
```

Testing with MSW:

```
// LoginForm.test.jsx
import { render, screen, fireEvent, waitFor } from '@testing-library/react';
import LoginForm from './LoginForm';

test('shows success message on successful login', async () => {
 render(<LoginForm />);

 fireEvent.change(screen.getByLabelText(/email/i), {
 target: { value: 'user@example.com' }
 });

 fireEvent.change(screen.getByLabelText(/password/i), {
 target: { value: 'password123' }
 });

 fireEvent.click(screen.getByRole('button', { name: /log in/i }));

 await waitFor(() => {
 expect(screen.getByText(/login successful/i)).toBeInTheDocument();
 });
});

test('shows error message on failed login', async () => {
 render(<LoginForm />);

 fireEvent.change(screen.getByLabelText(/email/i), {
 target: { value: 'wrong@example.com' }
 });

 fireEvent.change(screen.getByLabelText(/password/i), {
 target: { value: 'wrongpassword' }
 });

 fireEvent.click(screen.getByRole('button', { name: /log in/i }));

 await waitFor(() => {
 expect(screen.getByText(/invalid credentials/i)).toBeInTheDocument();
 });
});
```

```
});
});
```

## Utility Libraries

### 1. Lodash

```
Install Lodash
npm install lodash
```

Using Lodash in React:

```
import { debounce, throttle } from 'lodash';
import { groupBy, sortBy, uniqBy } from 'lodash';

// Debounced search input
function SearchInput({ onSearch }) {
 const debouncedSearch = useCallback(
 debounce((value) => {
 onSearch(value);
 }, 500),
 [onSearch]
);

 return (
 <input
 type="text"
 onChange={(e) => debouncedSearch(e.target.value)}
 placeholder="Search..."
 />
);
}

// Data manipulation
function UserList({ users }) {
 // Group users by role
 const groupedUsers = groupBy(users, 'role');

 // Sort users by name
 const sortedUsers = sortBy(users, ['name']);

 // Remove duplicates
 const uniqueUsers = uniqBy(users, 'email');

 return (
 <div>
 {/* Render the processed user data */}
 </div>
);
}
```

```
);
 }
```

## 2. date-fns

```
Install date-fns
npm install date-fns
```

Using date-fns in React:

```
import { format, formatDistance, parseISO, isValid, addDays } from 'date-fns';

function DateDisplay({ dateString }) {
 // Parse ISO string to Date object
 const date = parseISO(dateString);

 // Check if date is valid
 if (!isValid(date)) {
 return Invalid date;
 }

 return (
 <div>
 <p>Formatted: {format(date, 'MMMM do, yyyy')}</p>
 <p>Relative: {formatDistance(date, new Date(), { addSuffix: true })}</p>
 <p>Tomorrow: {format(addDays(date, 1), 'yyyy-MM-dd')}</p>
 </div>
);
}
```

## 3. Zod (Type Validation)

```
Install Zod
npm install zod
```

Using Zod for data validation:

```
import { z } from 'zod';

// Define schema
const UserSchema = z.object({
 id: z.string().uuid(),
 name: z.string().min(2).max(100),
 email: z.string().email(),
 age: z.number().int().positive().optional(),
```

```
role: z.enum(['admin', 'user', 'editor']),
createdAt: z.string().datetime(),
tags: z.array(z.string()).default([]),
settings: z.object({
 newsletter: z.boolean().default(false),
 theme: z.enum(['light', 'dark']).default('light')
}).optional()
});

// Type inference
type User = z.infer<typeof UserSchema>;

function UserProcessor({ userData }) {
 const processUserData = () => {
 try {
 // Validate and parse incoming data
 const validatedUser = UserSchema.parse(userData);
 return validatedUser;
 } catch (error) {
 if (error instanceof z.ZodError) {
 console.error('Validation error:', error.errors);
 return null;
 }
 throw error;
 }
 };

 const user = processUserData();

 if (!user) {
 return <div>Invalid user data</div>;
 }

 return (
 <div>
 <h2>{user.name}</h2>
 <p>Email: {user.email}</p>
 <p>Role: {user.role}</p>
 </div>
);
}
```

## Real-World Project Examples

### 1. Authentication System

```
// src/contexts/AuthContext.js
import { createContext, useContext, useState, useEffect } from 'react';

const AuthContext = createContext(null);
```

```
export function AuthProvider({ children }) {
 const [user, setUser] = useState(null);
 const [loading, setLoading] = useState(true);

 useEffect(() => {
 // Check for stored authentication on load
 const storedUser = localStorage.getItem('user');
 if (storedUser) {
 setUser(JSON.parse(storedUser));
 }
 setLoading(false);
 }, []);

 const login = async (email, password) => {
 try {
 // In a real app, call an authentication API
 const response = await fetch('/api/login', {
 method: 'POST',
 headers: { 'Content-Type': 'application/json' },
 body: JSON.stringify({ email, password })
 });

 if (!response.ok) {
 const userData = await response.json();
 throw new Error(userData.message || 'Login failed');
 }

 const userData = await response.json();

 // Store user data and token
 localStorage.setItem('user', JSON.stringify(userData));
 localStorage.setItem('token', userData.token);

 setUser(userData);
 return userData;
 } catch (error) {
 console.error('Login error:', error);
 throw error;
 }
 };

 const register = async (name, email, password) => {
 try {
 const response = await fetch('/api/register', {
 method: 'POST',
 headers: { 'Content-Type': 'application/json' },
 body: JSON.stringify({ name, email, password })
 });

 if (!response.ok) {
 const userData = await response.json();
 throw new Error(userData.message || 'Registration failed');
 }
 }
 };
}
```

```

 const userData = await response.json();
 return userData;
} catch (error) {
 console.error('Registration error:', error);
 throw error;
}
};

const logout = () => {
 localStorage.removeItem('user');
 localStorage.removeItem('token');
 setUser(null);
};

return (
 <AuthContext.Provider
 value={{
 user,
 loading,
 login,
 register,
 logout,
 isAuthenticated: !!user
 }}
 >
 {children}
 </AuthContext.Provider>
);
}

export function useAuth() {
 const context = useContext(AuthContext);
 if (context === null) {
 throw new Error('useAuth must be used within an AuthProvider');
 }
 return context;
}

```

Protected route implementation:

```

// src/components/ProtectedRoute.js
import { Navigate, Outlet } from 'react-router-dom';
import { useAuth } from '../contexts/AuthContext';

function ProtectedRoute() {
 const { isAuthenticated, loading } = useAuth();

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

 return isAuthenticated ? <Outlet /> : <Navigate to="/login" />;
}

```

```

}

// App.js with route protection
import { BrowserRouter, Routes, Route } from 'react-router-dom';
import { AuthProvider } from './contexts/AuthContext';
import ProtectedRoute from './components/ProtectedRoute';
import Login from './pages/Login';
import Register from './pages/Register';
import Dashboard from './pages/Dashboard';
import Profile from './pages/Profile';

function App() {
 return (
 <AuthProvider>
 <BrowserRouter>
 <Routes>
 <Route path="/login" element={<Login />} />
 <Route path="/register" element={<Register />} />

 {/* Protected routes */}
 <Route element={<ProtectedRoute />}>
 <Route path="/dashboard" element={<Dashboard />} />
 <Route path="/profile" element={<Profile />} />
 </Route>

 <Route path="*" element={<Navigate to="/login" />} />
 </Routes>
 </BrowserRouter>
 </AuthProvider>
);
}

```

Login page implementation:

```

// src/pages/Login.js
import { useState } from 'react';
import { useNavigate, Link } from 'react-router-dom';
import { useAuth } from '../contexts/AuthContext';
import { useForm } from 'react-hook-form';
import { z } from 'zod';
import { zodResolver } from '@hookform/resolvers/zod';

// Form validation schema
const loginSchema = z.object({
 email: z.string().email('Please enter a valid email'),
 password: z.string().min(6, 'Password must be at least 6 characters')
});

function Login() {
 const { login } = useAuth();
 const navigate = useNavigate();
 const [error, setError] = useState('');

```

```
const [isSubmitting, setIsSubmitting] = useState(false);

const { register, handleSubmit, formState: { errors } } = useForm({
 resolver: zodResolver(loginSchema)
});

const onSubmit = async (data) => {
 try {
 setError('');
 setIsSubmitting(true);

 await login(data.email, data.password);
 navigate('/dashboard');
 } catch (err) {
 setError(err.message || 'Failed to log in');
 } finally {
 setIsSubmitting(false);
 }
};

return (
 <div className="login-page">
 <div className="login-container">
 <h1>Log In</h1>

 {error && <div className="error-message">{error}</div>}

 <form onSubmit={handleSubmit(onSubmit)}>
 <div className="form-group">
 <label htmlFor="email">Email</label>
 <input
 id="email"
 type="email"
 {...register('email')}
 />
 {errors.email && (
 <p className="error">{errors.email.message}</p>
)}
 </div>

 <div className="form-group">
 <label htmlFor="password">Password</label>
 <input
 id="password"
 type="password"
 {...register('password')}
 />
 {errors.password && (
 <p className="error">{errors.password.message}</p>
)}
 </div>

 <button
 type="submit"
 >

```

```

 disabled={isSubmitting}
 className="btn btn-primary"
 >
 {isSubmitting ? 'Logging in...' : 'Log In'}
 </button>
</form>

<p className="register-link">
 Don't have an account? <Link to="/register">Register here</Link>
</p>
</div>
</div>
);
}

```

## 2. Data Dashboard with Filters and Visualization

```

import { useState, useEffect } from 'react';
import {
 LineChart, Line, BarChart, Bar,
 XAxis, YAxis, CartesianGrid, Tooltip, Legend,
 ResponsiveContainer
} from 'recharts';
import { format, subDays, eachDayOfInterval } from 'date-fns';

function Dashboard() {
 const [data, setData] = useState([]);
 const [loading, setLoading] = useState(true);
 const [error, setError] = useState(null);
 const [filter, setFilter] = useState('30days');
 const [chartType, setChartType] = useState('line');

 useEffect(() => {
 const fetchData = async () => {
 try {
 setLoading(true);

 // In a real app, this would be an API call with the filter applied
 // For demo, we'll generate sample data
 const endDate = new Date();
 let startDate;

 switch (filter) {
 case '7days':
 startDate = subDays(endDate, 7);
 break;
 case '30days':
 startDate = subDays(endDate, 30);
 break;
 case '90days':
 startDate = subDays(endDate, 90);
 break;
 }

 const data = eachDayOfInterval(
 { start: startDate, end: endDate },
 { step: 1 }
).map((date) => ({
 date: date.toISOString().split('T')[0],
 value: Math.floor(Math.random() * 100)
 }));
 setData(data);
 } catch (err) {
 setError(err);
 } finally {
 setLoading(false);
 }
 };
 fetchData();
 }, [filter]);
}

export default Dashboard;

```

```
 break;
 default:
 startDate = subDays(endDate, 30);
 }

const dateRange = eachDayOfInterval({ start: startDate, end: endDate });

const sampleData = dateRange.map(date => {
 const formattedDate = format(date, 'MMM dd');

 return {
 date: formattedDate,
 sales: Math.floor(Math.random() * 1000) + 500,
 visitors: Math.floor(Math.random() * 500) + 200,
 };
});

setData(sampleData);
} catch (err) {
 setError('Failed to load dashboard data');
 console.error(err);
} finally {
 setLoading(false);
}
};

fetchData();
}, [filter]);

// Calculate totals for summary boxes
const totalSales = data.reduce((sum, item) => sum + item.sales, 0);
const totalVisitors = data.reduce((sum, item) => sum + item.visitors, 0);
const averageSalesPerDay = data.length > 0 ? totalSales / data.length : 0;

if (loading) return <div className="loading">Loading dashboard data...</div>;
if (error) return <div className="error-message">{error}</div>;

return (
<div className="dashboard">
 <h1>Sales Dashboard</h1>

 {/* Filter controls */}
 <div className="controls">
 <div className="time-filter">
 <label>Time Period:</label>
 <select
 value={filter}
 onChange={(e) => setFilter(e.target.value)}
 >
 <option value="7days">Last 7 Days</option>
 <option value="30days">Last 30 Days</option>
 <option value="90days">Last 90 Days</option>
 </select>
 </div>
 </div>
)
```

```
<div className="chart-type">
 <label>Chart Type:</label>
 <select
 value={chartType}
 onChange={(e) => setChartType(e.target.value)}
 >
 <option value="line">Line Chart</option>
 <option value="bar">Bar Chart</option>
 </select>
</div>
</div>

{/* Summary boxes */}
<div className="summary-boxes">
 <div className="summary-box">
 <h3>Total Sales</h3>
 <p className="amount">${totalSales.toLocaleString()}</p>
 </div>

 <div className="summary-box">
 <h3>Total Visitors</h3>
 <p className="amount">{totalVisitors.toLocaleString()}</p>
 </div>

 <div className="summary-box">
 <h3>Avg. Daily Sales</h3>
 <p className="amount">${averageSalesPerDay.toFixed(2)}</p>
 </div>
</div>

{/* Chart */}
<div className="chart-container">
 <h2>Sales & Visitors Trend</h2>

 <ResponsiveContainer width="100%" height={400}>
 {chartType === 'line' ? (
 <LineChart data={data}>
 <CartesianGrid strokeDasharray="3 3" />
 <XAxis dataKey="date" />
 <YAxis yAxisId="left" />
 <YAxis yAxisId="right" orientation="right" />
 <Tooltip />
 <Legend />
 <Line
 yAxisId="left"
 type="monotone"
 dataKey="sales"
 stroke="#8884d8"
 name="Sales ($)"
 />
 <Line
 yAxisId="right"
 type="monotone"
 />
 </LineChart>
) : (
 <BarChart data={data}>
 <CartesianGrid strokeDasharray="3 3" />
 <XAxis dataKey="date" />
 <YAxis />
 <Bar
 dataKey="sales"
 fill="#8884d8"
 name="Sales ($)"
 />
 <Bar
 dataKey="visitors"
 fill="#f9a86a"
 name="Visitors"
 />
 </BarChart>
)}
 </ResponsiveContainer>
</div>
```

```

 dataKey="visitors"
 stroke="#82ca9d"
 name="Visitors"
 />
 </LineChart>
) : (
 <BarChart data={data}>
 <CartesianGrid strokeDasharray="3 3" />
 <XAxis dataKey="date" />
 <YAxis />
 <Tooltip />
 <Legend />
 <Bar dataKey="sales" fill="#8884d8" name="Sales ($)" />
 <Bar dataKey="visitors" fill="#82ca9d" name="Visitors" />
 </BarChart>
){}
</ResponsiveContainer>
</div>

/* Data table */
<div className="data-table">
 <h2>Detailed Data</h2>

 <table>
 <thead>
 <tr>
 <th>Date</th>
 <th>Sales ($)</th>
 <th>Visitors</th>
 </tr>
 </thead>
 <tbody>
 {data.map((item, index) => (
 <tr key={index}>
 <td>{item.date}</td>
 <td>${item.sales.toLocaleString()}</td>
 <td>{item.visitors.toLocaleString()}</td>
 </tr>
))}
 </tbody>
 </table>
</div>
</div>
);
}

```

### 3. E-commerce Product Catalog with Filtering

```

import { useState, useEffect } from 'react';
import { useSearchParams } from 'react-router-dom';

```

```
function ProductCatalog() {
 const [products, setProducts] = useState([]);
 const [categories, setCategories] = useState([]);
 const [loading, setLoading] = useState(true);
 const [error, setError] = useState(null);

 // Use URL search params for filtering
 const [searchParams, setSearchParams] = useSearchParams();
 const categoryFilter = searchParams.get('category') || 'all';
 const priceFilter = searchParams.get('price') || 'all';
 const sortBy = searchParams.get('sort') || 'newest';
 const searchQuery = searchParams.get('search') || '';

 // Fetch products and categories
 useEffect(() => {
 const fetchData = async () => {
 try {
 setLoading(true);

 // In a real app, these would be API calls
 // Simulating fetch with timeout
 await new Promise(resolve => setTimeout(resolve, 800));

 // Mocked product data
 const mockProducts = [
 {
 id: 1,
 name: 'Wireless Headphones',
 category: 'electronics',
 price: 99.99,
 rating: 4.5,
 image: 'headphones.jpg',
 inStock: true,
 createdAt: '2023-02-15'
 },
 {
 id: 2,
 name: 'Cotton T-Shirt',
 category: 'clothing',
 price: 19.99,
 rating: 4.2,
 image: 'tshirt.jpg',
 inStock: true,
 createdAt: '2023-03-20'
 },
 {
 id: 3,
 name: 'Running Shoes',
 category: 'footwear',
 price: 89.99,
 rating: 4.7,
 image: 'shoes.jpg',
 inStock: true,
 createdAt: '2023-01-10'
 }
];
 setProducts(mockProducts);
 } catch (err) {
 setError(err.message);
 } finally {
 setLoading(false);
 }
 };
 });
}
```

```
 },
 {
 id: 4,
 name: 'Smartphone',
 category: 'electronics',
 price: 699.99,
 rating: 4.8,
 image: 'smartphone.jpg',
 inStock: false,
 createdAt: '2023-04-05'
 },
 {
 id: 5,
 name: 'Coffee Mug',
 category: 'home',
 price: 12.99,
 rating: 4.0,
 image: 'mug.jpg',
 inStock: true,
 createdAt: '2023-03-01'
 }
];
 }

 // Extract unique categories
 const uniqueCategories = [...new Set(mockProducts.map(p => p.category))];

 setProducts(mockProducts);
 setCategories(uniqueCategories);
 } catch (err) {
 setError('Failed to load products');
 console.error(err);
 } finally {
 setLoading(false);
 }
};

fetchData();
}, []);

// Apply filters and sorting
const filteredProducts = products
 .filter(product => {
 // Category filter
 if (categoryFilter !== 'all' && product.category !== categoryFilter) {
 return false;
 }

 // Price filter
 if (priceFilter === 'under25' && product.price >= 25) return false;
 if (priceFilter === '25to100' && (product.price < 25 || product.price > 100)) return false;
 if (priceFilter === 'over100' && product.price <= 100) return false;

 // Search query
 })
 .sort((a, b) => a.name.localeCompare(b.name));

```

```
if (searchQuery &&
!product.name.toLowerCase().includes(searchQuery.toLowerCase())) {
 return false;
}

return true;
})
.sort((a, b) => {
// Sorting
if (sortBy === 'newest') {
 return new Date(b.createdAt) - new Date(a.createdAt);
}
if (sortBy === 'priceAsc') {
 return a.price - b.price;
}
if (sortBy === 'priceDesc') {
 return b.price - a.price;
}
if (sortBy === 'rating') {
 return b.rating - a.rating;
}
return 0;
});

// Update filters
const updateFilters = (key, value) => {
 const newParams = new URLSearchParams(searchParams);
 newParams.set(key, value);
 setSearchParams(newParams);
};

if (loading) return <div className="loading">Loading products...</div>;
if (error) return <div className="error-message">{error}</div>;

return (
<div className="product-catalog">
<h1>Product Catalog</h1>

{/* Search bar */}
<div className="search-bar">
<input
 type="text"
 placeholder="Search products..."
 value={searchQuery}
 onChange={(e) => updateFilters('search', e.target.value)}
/>
</div>

<div className="catalog-container">
{/* Filters sidebar */}
<aside className="filters">
<div className="filter-section">
<h3>Categories</h3>

```

```

 <button
 className={categoryFilter === 'all' ? 'active' : ''}
 onClick={() => updateFilters('category', 'all')}
 >
 All Categories
 </button>

{categories.map(category => (
 <li key={category}>
 <button
 className={categoryFilter === category ? 'active' : ''}
 onClick={() => updateFilters('category', category)}
 >
 {category.charAt(0).toUpperCase() + category.slice(1)}
 </button>

))}

</div>

<div className="filter-section">
 <h3>Price Range</h3>

 <button
 className={priceFilter === 'all' ? 'active' : ''}
 onClick={() => updateFilters('price', 'all')}
 >
 All Prices
 </button>

 <button
 className={priceFilter === 'under25' ? 'active' : ''}
 onClick={() => updateFilters('price', 'under25')}
 >
 Under $25
 </button>

 <button
 className={priceFilter === '25to100' ? 'active' : ''}
 onClick={() => updateFilters('price', '25to100')}
 >
 $25 to $100
 </button>

 <button
 className={priceFilter === 'over100' ? 'active' : ''}
 onClick={() => updateFilters('price', 'over100')}
 >
 Over $100
 </button>

</div>
```

```
 </button>

 </div>
</aside>

/* Products grid */

<div className="products-header">
 <div className="results-count">
 {filteredProducts.length} {filteredProducts.length === 1 ? 'product' : 'products'} found
 </div>

 <div className="sort-options">
 <label>Sort by: </label>
 <select
 value={sortBy}
 onChange={(e) => updateFilters('sort', e.target.value)}
 >
 <option value="newest">Newest</option>
 <option value="priceAsc">Price: Low to High</option>
 <option value="priceDesc">Price: High to Low</option>
 <option value="rating">Rating</option>
 </select>
 </div>
 </div>


```

{filteredProducts.length === 0 ? (

```
 <div className="no-results">
 No products match your filters. Try changing your search criteria.
 </div>
) : (
 <div className="products-grid">
 {filteredProducts.map(product => (
 <div key={product.id} className="product-card">
 <div className="product-image">

 {!product.inStock && Out of
Stock}
 </div>

 <div className="product-info">
 <h3>{product.name}</h3>
 <div className="product-category">{product.category}</div>
 <div className="product-price">${product.price.toFixed(2)}</div>
 </div>
 <div className="product-rating">
 ★ {product.rating} ({Math.floor(Math.random() * 100) + 10})
 </div>
 <button
 className="add-to-cart"
 disabled={!product.inStock}
 >Add to Cart</button>
 </div>
 </div>
)
```

```
>
 {product.inStock ? 'Add to Cart' : 'Out of Stock'}
 </button>
</div>
</div>
))}
</div>
)
</div>
</div>
</div>
);
}
```

## Common Interview Questions and Challenges

### React Fundamentals

#### 1. What is the difference between state and props?

State:

- Managed within the component
- Can be updated by the component using `setState` or `useState`
- Changes in state trigger a re-render
- Private to the component that defines it

Props:

- Passed down from parent components
- Read-only within the receiving component
- Changes in props from parent trigger a re-render
- Used for communication between components

#### 2. Explain the Virtual DOM and its advantages.

The Virtual DOM is a lightweight JavaScript representation of the actual DOM. React uses it to:

- Minimize direct DOM manipulation (which is slow)
- Batch updates for better performance
- Compare previous and current states to determine the minimal set of changes needed (diffing)
- Enable cross-platform development since the Virtual DOM is not tied to the browser DOM

#### 3. What is JSX and why does React use it?

JSX is a syntax extension for JavaScript that looks similar to HTML. React uses JSX because:

- It makes the code more readable and intuitive
- It allows you to write markup and logic in the same file
- It helps prevent injection attacks by escaping embedded values
- It's transformed into regular JavaScript functions (`React.createElement`) at build time

#### 4. Explain React component lifecycle methods.

In class components:

- **Mounting:** constructor → getDerivedStateFromProps → render → componentDidMount
- **Updating:** getDerivedStateFromProps → shouldComponentUpdate → render → getSnapshotBeforeUpdate → componentDidUpdate
- **Unmounting:** componentWillUnmount

In functional components with hooks:

- **Mounting:** Function execution → useEffect(() => {}, []) (empty dependency array)
- **Updating:** Function execution → useEffect with relevant dependencies
- **Unmounting:** useEffect cleanup function

## React Hooks

#### 5. What are React Hooks and why were they introduced?

React Hooks are functions that let you "hook into" React state and lifecycle features from functional components. They were introduced to:

- Allow using state and other React features without writing classes
- Make it easier to reuse stateful logic between components
- Organize related logic in one place instead of splitting it across lifecycle methods
- Avoid complex patterns like render props and higher-order components

#### 6. Explain the rules of hooks.

- Only call hooks at the top level of your function component
- Don't call hooks inside loops, conditions, or nested functions
- Only call hooks from React function components or custom hooks
- The name of custom hooks should start with "use"

#### 7. What is the difference between useEffect and useLayoutEffect?

Both hooks run side effects, but with different timing:

useEffect:

- Runs asynchronously after the DOM has been updated
- Doesn't block browser painting
- Preferred for most side effects

useLayoutEffect:

- Runs synchronously before the browser paints
- Blocks visual updates until the callback is finished
- Use when you need to make DOM measurements or DOM mutations that should be visible immediately

## State Management

## 8. When would you use Context API vs. Redux?

Context API:

- Simpler applications with less complex state
- When state changes are infrequent
- When you have specific states that only certain components need
- For theme management, user authentication, language preferences

Redux:

- Complex applications with lots of state interactions
- When you need advanced features like middleware, time-travel debugging
- For better performance with many updates (since Context triggers all consumers to re-render)
- When state structure is complex and deeply nested
- When you need a predictable state container with strict update patterns

## 9. Explain the concept of "lifting state up" in React.

"Lifting state up" means moving state from child components to a common ancestor (parent) component. This is done when:

- Multiple components need to reflect the same changing data
- Components need to be in sync
- You want to maintain a single "source of truth" for a particular state

The parent component then manages the state and passes it down via props to child components, along with any functions needed to update the state.

## Performance Optimization

### 10. How would you optimize a React application for performance?

- Use React.memo() for functional components to prevent unnecessary renders
- Use shouldComponentUpdate or PureComponent for class components
- Implement virtualization for long lists with react-window or react-virtualized
- Code splitting with React.lazy() and Suspense to reduce bundle size
- Memoize expensive calculations with useMemo
- Use useCallback to maintain function reference stability
- Implement proper dependency arrays in useEffect to avoid unnecessary effect runs
- Debounce or throttle event handlers for performance-intensive operations
- Profile and analyze your app with React DevTools to identify performance bottlenecks

### 11. What is the purpose of the "key" prop when rendering a list of components?

The "key" prop helps React identify which items have changed, been added, or been removed. It's used to give elements in a list a stable identity. This allows React to reuse existing DOM elements when possible, rather than rebuilding them, which improves performance. Keys should be:

- Unique among siblings
- Stable across renders

- Typically from your data (like IDs)

Using indexes as keys can lead to bugs when items are reordered or when you add/remove items in the middle of the list.

## Common Coding Challenges

### 12. Implement a simple counter with React hooks.

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

 const increment = () => setCount(prevCount => prevCount + 1);
 const decrement = () => setCount(prevCount => prevCount - 1);
 const reset = () => setCount(0);

 return (
 <div>
 <h2>Count: {count}</h2>
 <button onClick={decrement}>-</button>
 <button onClick={reset}>Reset</button>
 <button onClick={increment}>+</button>
 </div>
);
}
```

### 13. Create a custom hook for managing form state.

```
function useForm(initialValues = {}) {
 const [values, setValues] = useState(initialValues);
 const [errors, setErrors] = useState({});
 const [touched, setTouched] = useState({});

 const handleChange = (e) => {
 const { name, value } = e.target;
 setValues(prev => ({ ...prev, [name]: value }));
 };

 const handleBlur = (e) => {
 const { name } = e.target;
 setTouched(prev => ({ ...prev, [name]: true }));
 };

 const resetForm = () => {
 setValues(initialValues);
 setErrors({});
 setTouched({});
 };

 const validateField = (name, value, validationRules) => {
```

```
if (!validationRules) return null;

if (validationRules.required && !value) {
 return 'This field is required';
}

if (validationRules.minLength && value.length < validationRules.minLength) {
 return `Must be at least ${validationRules.minLength} characters`;
}

if (validationRules.pattern && !validationRules.pattern.test(value)) {
 return validationRules.message || 'Invalid format';
}

return null;
};

const validateForm = (validationRules) => {
 const newErrors = {};
 let isValid = true;

 Object.keys(values).forEach(name => {
 const fieldRules = validationRules?[name];
 const error = validateField(name, values[name], fieldRules);

 if (error) {
 newErrors[name] = error;
 isValid = false;
 }
 });

 setErrors(newErrors);
 return isValid;
};

return {
 values,
 errors,
 touched,
 handleChange,
 handleBlur,
 resetForm,
 validateForm,
 setValues,
 setErrors
};
}

// Usage example
function SignupForm() {
 const {
 values,
 errors,
 touched,
```

```
handleChange,
handleBlur,
validateForm,
resetForm
} = useForm({
 name: '',
 email: '',
 password: ''
});

const validationRules = {
 name: { required: true },
 email: {
 required: true,
 pattern: /^[^s@]+@[^\s@]+\.[^\s@]+$/,
 message: 'Invalid email format'
 },
 password: { required: true, minLength: 8 }
};

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

 if (validateForm(validationRules)) {
 console.log('Form submitted:', values);
 resetForm();
 }
};

return (
 <form onSubmit={handleSubmit}>
 <div>
 <label>Name</label>
 <input
 name="name"
 value={values.name}
 onChange={handleChange}
 onBlur={handleBlur}
 />
 {touched.name && errors.name && (
 <div className="error">{errors.name}</div>
)}
 </div>

 <div>
 <label>Email</label>
 <input
 name="email"
 type="email"
 value={values.email}
 onChange={handleChange}
 onBlur={handleBlur}
 />
 {touched.email && errors.email && (

```

```
 <div className="error">{errors.email}</div>
)}
</div>

<div>
 <label>Password</label>
 <input
 name="password"
 type="password"
 value={values.password}
 onChange={handleChange}
 onBlur={handleBlur}
 />
 {touched.password && errors.password && (
 <div className="error">{errors.password}</div>
)}
</div>

 <button type="submit">Sign Up</button>
</form>
);
}
```

#### 14. Implement a simple search filter for a list of items.

```
function SearchableList({ items }) {
 const [searchTerm, setSearchTerm] = useState('');

 // Filter items based on search term
 const filteredItems = useMemo(() => {
 return items.filter(item =>
 item.name.toLowerCase().includes(searchTerm.toLowerCase())
);
 }, [items, searchTerm]);

 return (
 <div>
 <input
 type="text"
 placeholder="Search items..."
 value={searchTerm}
 onChange={(e) => setSearchTerm(e.target.value)}
 />

 {filteredItems.length === 0 ? (
 <p>No items match your search.</p>
) : (

 {filteredItems.map(item => (
 <li key={item.id}>{item.name}
)))

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

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

## 15. Build a simple accordion component.

```
function Accordion({ items }) {
 const [activeIndex, setActiveIndex] = useState(null);

 const toggleItem = (index) => {
 setActiveIndex(activeIndex === index ? null : index);
 };

 return (
 <div className="accordion">
 {items.map((item, index) => (
 <div key={index} className="accordion-item">
 <button
 className={`accordion-header ${activeIndex === index ? 'active' : ''}`}
 onClick={() => toggleItem(index)}
 >
 {item.title}

 {activeIndex === index ? '-' : '+'}

 </button>

 {activeIndex === index && (
 <div className="accordion-content">
 {item.content}
 </div>
)}
 </div>
))}
 </div>
);
}

// Usage
function App() {
 const accordionItems = [
 {
 title: 'Section 1',
 content: 'Content for section 1...'
 },
 {
 title: 'Section 2',
 content: 'Content for section 2...'
 },
 {
 title: 'Section 3',
 content: 'Content for section 3...'
 }
]
}
```

```
 title: 'Section 3',
 content: 'Content for section 3...'
 }
];

return <Accordion items={accordionItems} />;
}
```

## Resources for Continued Learning

### Official Documentation

- [React Documentation](#): The official React documentation, completely rewritten in 2023.
- [Create React App](#): Official documentation for Create React App.
- [Vite Documentation](#): Documentation for Vite, a faster and leaner development experience.
- [React Router](#): Documentation for React Router.
- [Redux Toolkit](#): Official documentation for Redux Toolkit.

### Books

- "**React: Up and Running**" by Stoyan Stefanov
- "**Learning React**" by Alex Banks and Eve Porcello
- "**Pro React 16**" by Adam Freeman
- "**React Design Patterns and Best Practices**" by Michele Bertoli
- "**Testing React Applications with Jest**" by Jeff Palmer

### Online Courses

- [Epic React](#) by Kent C. Dodds: Comprehensive React training
- [React - The Complete Guide](#) on Udemy by Maximilian Schwarzmüller
- [Frontend Masters React Path](#)
- [Wes Bos React Courses](#)
- [React for Beginners](#) by Wes Bos

### Blogs and Websites

- [Overreacted](#) by Dan Abramov: Deep dives into React concepts
- [Kent C. Dodds Blog](#): Articles on React and testing
- [Robin Wieruch's Blog](#): In-depth React tutorials
- [Josh W. Comeau's Blog](#): Interactive articles on React and front-end development
- [DEV Community](#): Community articles about React

### YouTube Channels

- [Academind](#)
- [Ben Awad](#)
- [Codevolution](#)
- [Web Dev Simplified](#)
- [Jack Herrington](#)

## Advanced Topics to Explore

- **React Server Components:** Learn about the new React architecture
- **Concurrent Mode:** Understanding how React 18 improves rendering
- **Suspense for Data Fetching:** Advanced data loading patterns
- **React Native:** Applying your React knowledge to mobile development
- **GraphQL with Apollo:** Advanced data fetching paradigms
- **Monorepos with NX or Turborepo:** Managing multiple React applications
- **Micro-frontends:** Building large-scale applications with multiple teams

## Community and Networking

- [React Subreddit](#)
- [React Discord](#)
- [Stack Overflow - React](#)
- Local React meetups in your area
- React conferences: React Conf, React Europe, React Summit

## Staying Up-to-Date

- Follow the [React Blog](#)
- Subscribe to newsletters:
  - [React Status](#)
  - [JavaScript Weekly](#)
  - [This Week In React](#)
- Follow React team members on Twitter/X
- Regularly check the React GitHub repository for upcoming features and changes

## Practice Projects

- Build a personal portfolio site with React
- Create a blog application with React and a headless CMS
- Build a real-time chat application
- Create a task management application with drag-and-drop functionality
- Build an e-commerce store with cart and checkout functionality
- Create a weather application that consumes a weather API
- Build a social media dashboard with charts and data visualization
- Create a photo gallery with search and filtering capabilities

By continuously expanding your knowledge and building real-world projects, you'll steadily master React and modern JavaScript development.