React Exercises for BCU students

Author: MSc. Tran Vinh Khiem

Class: BCU2025

Section 5: Advanced Component Patterns

As React applications grow in complexity, developers often encounter scenarios where they need to reuse component logic or share UI concerns across multiple components. Advanced component patterns like Higher-Order Components (HOCs) and Render Props provide powerful solutions for achieving reusability and flexibility beyond simple component composition.

5.1 Higher-Order Components (HOCs)

A **Higher-Order Component (HOC)** is an advanced technique in React for reusing component logic. HOCs are functions that take a component as an argument and return a new component with enhanced capabilities. They are a pattern derived from React's compositional nature.

Think of HOCs as pure functions: they take a component and return a new component. They don't modify the input component, nor do they use inheritance to copy its behavior. Instead, HOCs compose the original component by wrapping it in another component.

Structure of an HOC

```
function withSomething(WrappedComponent) {

  return function EnhancedComponent(props) {

    // ... add some logic here ...

    return <WrappedComponent {...props} />;

  };

}

// Usage:

const MyEnhancedComponent = withSomething(MyComponent);
```

Common Use Cases for HOCs:

- **Code Reusability:** Share common logic (e.g., data fetching, authentication, logging) across multiple components without duplicating code.
- **Props Manipulation:** Inject additional props into the wrapped component.
- **State Abstraction:** Abstract away stateful logic from presentational components.

- **Conditional Rendering:** Control whether a component is rendered based on certain conditions (e.g., authentication status).

5.2 Render Props

A **render prop** is a technique for sharing code between React components using a prop whose value is a function. This function returns a React element and allows the consuming component to decide what to render. The term “render prop” refers to the fact that a component with a render prop takes a function that returns a React element and calls it instead of implementing its own render logic.

Structure of Render Props

```
class DataSource extends React.Component {

  constructor(props) {

    super(props);

    this.state = {

      data: null,

      loading: true,

    };

  }

  componentDidMount() {

    // Simulate data fetching

    setTimeout(() => {

      this.setState({

        data: { message: "Data from render prop!" },

        loading: false,

      });

    }, 1000);

  }
  render() {

    return this.props.render(this.state);

  }

}

// Usage:

<DataSource render={({ data, loading }) => (

  <div>

    {loading ? <p>Loading...</p> : <p>{data.message}</p>}

  </div>

)} />
```

Common Use Cases for Render Props:

- **Flexible UI Composition:** Allows the parent component to control the rendering of the child component, providing greater flexibility.
- **Cross-Cutting Concerns:** Share non-visual logic (e.g., mouse position tracking, window size) with multiple components.
- **Decoupling Logic from UI:** Separates the logic of how data is fetched or managed from how it is displayed.

HOCs vs. Render Props

Both HOCs and Render Props are patterns for code reuse and sharing logic. The choice between them often depends on the specific use case and personal preference:

Feature	**Higher-Order Components (HOCs)**	**Render Props**
Mechanism	Function that takes a component and returns a new component.	Component that takes a function as a prop to render content.
Composition	Wraps the component, implicitly passing props.	Explicitly passes data/functions to the render prop function.

Feature	**Higher-Order Components (HOCs)**	**Render Props**
Flexibility	Less flexible in terms of rendering logic.	More flexible, allows full control over rendering.

prop drilling if not carefully managed. | Can lead to nested JSX, sometimes called "callback hell" if not managed well. | | **Debugging** | Can be harder to debug due to multiple layers of abstraction. | Easier to debug as the data flow is more explicit. | | **Naming** | Can lead to naming collisions if not careful. | Less prone to naming collisions. |

In modern React, with the advent of Hooks, many use cases for HOCs and Render Props can now be achieved more simply and directly using custom Hooks. However, understanding these patterns is still valuable for working with older codebases or for specific scenarios where they might be more appropriate.

Section 6: Advanced Hooks and Performance Optimization

React Hooks have revolutionized how we write functional components, allowing us to manage state and side effects with greater flexibility and less boilerplate. Beyond `useState` and `useEffect`, several other hooks, along with memoization techniques, are crucial for optimizing the performance of complex React applications.

6.1 `useReducer` Hook

The `useReducer` Hook is an alternative to `useState` for managing more complex state logic, especially when the state transitions depend on the previous state or involve multiple sub-values. It is often preferred over `useState` when you have complex state logic that involves multiple sub-values or when the next state depends on the previous one. `useReducer` is also useful for optimizing performance of components that trigger deep updates because you can pass `dispatch` down instead of callbacks.

It takes a reducer function and an initial state, and returns the current state and a `dispatch` function. The `dispatch` function is used to trigger state updates by passing an action object to the reducer.

import React, { useReducer } from 'react';

// Reducer function: takes current state and an action, returns new state

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:
      throw new Error();
  }
}
function CounterWithReducer() {
  // useReducer returns the current state and a dispatch function
  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>
  );
}
```

6.2 `useCallback` Hook

The `useCallback` Hook returns a memoized callback function. This is useful when passing callbacks to optimized child components that rely on reference equality to prevent unnecessary renders. If a function is recreated on every render, a child component wrapped in `React.memo` (or a pure component) will still re-render because the prop it receives (the function) is considered new.

`useCallback` will only return a memoized version of the callback if one of the dependencies has changed. This can prevent unnecessary re-renders of child components.

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

// A child component that only re-renders if its props change

const Button = React.memo(({ onClick, children }) => {

  console.log(`Rendering button: ${children}`);

  return <button onClick={onClick}>{children}</button>;

});

function ParentComponent() {

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

  const [name, setName] = useState('Alice');

  // Memoized callback: only recreated if `count` changes

  const handleClick = useCallback(() => {

    setCount(prevCount => prevCount + 1);

  }, []); // Empty dependency array means it's created once

  // This callback will be recreated on every render because `name` changes

  const handleNameChange = () => {

    setName(name === 'Alice' ? 'Bob' : 'Alice');

  };
  return (
    <div>
      <p>Count: {count}</p>
      <Button onClick={handleClick}>Increment Count</Button>
      <p>Name: {name}</p>
      <button onClick={handleNameChange}>Change Name</button>
    </div>
  );
}
```

6.3 `useMemo` Hook

The `useMemo` Hook returns a memoized value. It is used to optimize expensive calculations by caching the result and only re-calculating it when one of its dependencies changes. This prevents unnecessary re-computations on every render, which can be a performance bottleneck for complex calculations.

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

function calculateExpensiveValue(num) {
  console.log('Calculating expensive value...');
  // Simulate a heavy computation
  let result = 0;
  for (let i = 0; i < 100000000; i++) {
    result += num;
  }
  return result;
}
```

```
function MemoizedComponent() {

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

  const [anotherState, setAnotherState] = useState(0);

  // The expensive calculation will only re-run when `count` changes

  const memoizedValue = useMemo(() => calculateExpensiveValue(count), [count]);

  return (

    <div>

      <p>Count: {count}</p>

      <p>Expensive Value: {memoizedValue}</p>

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

      <p>Another State: {anotherState}</p>

      <button onClick={() => setAnotherState(anotherState + 1)}>Increment Another
State</button>

    </div>

  );

}
```

6.4 `React.memo`

`React.memo` is a higher-order component (HOC) that memoizes a functional component. It works similarly to `PureComponent` for class components. If your component renders the same result given the same props, you can wrap it in `React.memo` for a performance boost in some cases. This prevents re-rendering of the component if its props have not changed.

```
import React from 'react';

const MyPureComponent = React.memo(function MyComponent(props) {

  /* render using props */
  console.log('MyPureComponent re-rendered');

  return <div>{props.data}</div>;

});

// In a parent component:

function Parent() {

  const [count, setCount] = React.useState(0);

  const [text, setText] = React.useState('Hello');

  return (

    <div>

      <MyPureComponent data={text} />

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

      <button onClick={() => setText('World')}>Change Text</button>

    </div>

  );

}
```

`React.memo` performs a shallow comparison of props. If you need a deep comparison, you can provide a custom comparison function as the second argument to `React.memo`.

Performance Optimization Strategies

Beyond these hooks, several other strategies contribute to optimizing React application performance:

- **Code Splitting (Lazy Loading):** Divide your code into smaller chunks that are loaded on demand, reducing the initial load time of your application. This can be achieved using `React.lazy` and `Suspense`.

- **Virtualization (Windowing):** For long lists of data, render only the items that are currently visible within the viewport. Libraries like `react-window` or `react-virtualized` can help with this.
- **Avoiding Unnecessary Re-renders:** Understand when and why components re-render. Use `React.memo`, `useCallback`, and `useMemo` effectively. Avoid passing new object/array literals as props if their content is the same.
- **Server-Side Rendering (SSR) / Static Site Generation (SSG):** For content-heavy applications, SSR or SSG can improve initial load times and SEO by rendering React components to HTML on the server.
- **Image Optimization:** Optimize image sizes and formats, and use lazy loading for images that are not immediately visible.
- **Throttling and Debouncing:** Limit the rate at which a function can fire, especially for event handlers like `onScroll` or `onInputChange`.

By combining these advanced hooks and optimization techniques, developers can build highly performant and responsive React applications that deliver a smooth user experience.

Section 7: Error Handling and Advanced Rendering

Robust applications need to handle errors gracefully to provide a good user experience. React provides a mechanism called Error Boundaries to catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI instead of crashing the entire application. Additionally, advanced rendering techniques like Portals allow for flexible UI placement outside the normal DOM hierarchy.

7.1 Error Boundaries

Error Boundaries are React components that catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI instead of the component tree crashing. Error boundaries catch errors during rendering, in lifecycle methods, and in constructors of the whole tree below them.

Limitations of Error Boundaries

Error boundaries **do not** catch errors for:

- Event handlers (use `try-catch` blocks inside them)
- Asynchronous code (e.g., `setTimeout` or `requestAnimationFrame` callbacks)
- Server-side rendering
- Errors thrown in the error boundary itself

Implementing an Error Boundary

An error boundary is a class component that implements at least one of `static getDerivedStateFromError()` or `componentDidCatch()` lifecycle methods.

- `static getDerivedStateFromError(error)`: This method is called after an error has been thrown by a descendant component. It receives the error that was thrown as a parameter and should return a value to update state. This allows the error boundary to render a fallback UI.

- `componentDidCatch(error, info)`: This method is called after an error has been thrown by a descendant component. It receives two parameters: `error` (the error that was thrown) and `info` (an object with a `componentStack` key containing information about which component threw the error). This method is used for side effects like logging errors.

```
import React from 'react';

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, error: error };

  }

  componentDidCatch(error, errorInfo) {

    // You can also log the error to an error reporting service

    console.error("Uncaught error:", error, errorInfo);

    this.setState({

      errorInfo: errorInfo

    });

  }

  render() {
```

```
if (this.state.hasError) {
  // You can render any custom fallback UI
  return (
    <div style={{
      border: '2px solid red',
      padding: '20px',
      margin: '20px',
      backgroundColor: '#ffe6e6',
      color: '#cc0000',
      borderRadius: '8px'
    }}>
      <h2>Something went wrong.</h2>
      <details style={{ whiteSpace: 'pre-wrap' }}>
        {this.state.error && this.state.error.toString()}
        <br />
        {this.state.errorInfo && this.state.errorInfo.componentStack}
      </details>
      <p>Please refresh the page or try again later.</p>
    </div>
  );
}
return this.props.children;
```

```
  }
}
// Example usage:
function BuggyCounter() {
  const [counter, setCounter] = React.useState(0);
  const handleClick = () => {
    setCounter(prevCounter => prevCounter + 1);
  };
  if (counter === 5) {
    // Simulate an error
    throw new Error('I crashed!');
  }
  return <h1 onClick={handleClick}>{counter}</h1>;
}
function App() {
  return (
    <div>
      <p>Click the numbers to increase the counters.</p>
      <p>The counter will crash when it reaches 5.</p>
      <hr />
      <ErrorBoundary>
        <BuggyCounter />
      </ErrorBoundary>

      <ErrorBoundary>

        <BuggyCounter />

      </ErrorBoundary>

      <hr />

      <p>These two counters are independent.</p>

    </div>

  );

}
```

7.2 Portals

Portals provide a way to render children into a DOM node that exists outside the DOM hierarchy of the parent component. This is particularly useful for components like modals, tooltips, and popovers that need to be rendered at the top level of the DOM to avoid z-index issues or overflow problems from parent components.

`ReactDOM.createPortal(child, container)`

- `child`: Any renderable React child, such as an element, string, or fragment.
- `container`: A DOM element. This is the location where the portal's content will be mounted.

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

import ReactDOM from 'react-dom';

const modalRoot = document.getElementById('modal-root');

function Modal({ children, isOpen, onClose }) {

 const [el] = useState(() => document.createElement('div'));

 useEffect(() => {

  if (isOpen) {
      modalRoot.appendChild(el);
    } else {
      // Clean up on close or unmount
      if (modalRoot.contains(el)) {
        modalRoot.removeChild(el);
      }
    }
    return () => {
      if (modalRoot.contains(el)) {
        modalRoot.removeChild(el);
      }
    };
  }, [isOpen, el]);
  if (!isOpen) {
    return null;
  }
  const modalOverlayStyles = {
    position: 'fixed',
    top: 0,
    left: 0,
    right: 0,
    bottom: 0,
    backgroundColor: 'rgba(0, 0, 0, 0.7)',
    display: 'flex',
    justifyContent: 'center',
    alignItems: 'center',
    zIndex: 1000
  };
  const modalContentStyles = {
    backgroundColor: 'white',
    padding: '30px',
    borderRadius: '10px',
    boxShadow: '0 5px 15px rgba(0, 0, 0, 0.3)',
    maxWidth: '500px',
    width: '90%',
    position: 'relative'
  };
  const closeButtonStyles = {
    position: 'absolute',
    top: '10px',
    right: '10px',
    background: 'none',
    border: 'none',
    fontSize: '24px',
    cursor: 'pointer',
    color: '#666'
  };
  return ReactDOM.createPortal(
    <div style={modalOverlayStyles} onClick={onClose}>
      <div style={modalContentStyles} onClick={e => e.stopPropagation()}>
        <button style={closeButtonStyles} onClick={onClose}>&times;</button>
        {children}
      </div>
    </div>,
    el
  );
}
// Example usage:
function App() {
  const [isModalOpen, setIsModalOpen] = useState(false);
  const openModal = () => setIsModalOpen(true);
  const closeModal = () => setIsModalOpen(false);
  return (
    <div style={{
      textAlign: 'center',
      padding: '50px',
      fontFamily: 'Arial, sans-serif'
    }}>
      <h1>Advanced Rendering with Portals</h1>
      <p>Click the button below to open a modal that renders outside the normal DOM flow.</p>
      <button
        onClick={openModal}
        style={{
          padding: '12px 24px',
          fontSize: '18px',
          backgroundColor: '#007bff',
          color: 'white',
          border: 'none',
          borderRadius: '5px',
          cursor: 'pointer'
        }}
      >
        Open Modal
      </button>
      <Modal isOpen={isModalOpen} onClose={closeModal}>
        <h2>Welcome to the Portal Modal!</h2>
        <p>This content is rendered using a React Portal, meaning it lives in a different part of the
DOM tree than its parent component.</p>
        <p>This is useful for things like modals, tooltips, and popovers to avoid layout issues.</p>
        <button
          onClick={closeModal}
          style={{
            padding: '10px 20px',
            fontSize: '16px',
            backgroundColor: '#6c757d',
            color: 'white',
            border: 'none',
            borderRadius: '5px',
            cursor: 'pointer',
            marginTop: '20px'
          }}
        >
          Close Modal
        </button>
      </Modal>
      <div id="modal-root"></div> {/* This is where the modal will be mounted */}
    </div>
  );
}
```

Key Considerations for Portals

- **Event Bubbling:** Events fired from inside a portal will still propagate up to the parent components in the React tree, even if the portal's DOM node is not a direct child of the parent's DOM node. This means you can still use React's event system as usual.
- **Accessibility:** When using modals or other elements rendered with portals, ensure you manage focus and keyboard navigation for accessibility. Libraries like `react-modal` often handle these concerns for you.

Error Boundaries and Portals are powerful tools for building more resilient and flexible React applications, allowing developers to handle unexpected errors gracefully and manage complex UI layouts effectively.

Advanced Practical Exercises

These exercises are designed to challenge your understanding of advanced React concepts and patterns. They will help you build more robust, performant, and maintainable applications.

Exercise 13: Render Props for Data Fetching

Objective: Implement a component using the Render Props pattern to fetch data and pass it to its children.

Instructions:

1. Create a functional component named `DataLoader`.
2. `DataLoader` should accept a `render` prop, which is a function.
3. Inside `DataLoader`, use `useState` and `useEffect` to simulate fetching data (e.g., from `https://jsonplaceholder.typicode.com/posts/1`). Manage `data`, `loading`, and `error` states.
4. Call the `render` prop function, passing the `data`, `loading`, and `error` states as arguments.
5. In your `App` component, use `DataLoader` and its `render` prop to display the fetched data or a loading/error message.

Expected Output: A component that fetches data and displays it, with loading and error states handled by the consumer of the `DataLoader` component.

Hints:

- The `render` prop will be a function that takes an object as an argument.
- Remember to handle the `loading` and `error` states within the `render` prop function.

Exercise 14: Complex State Management with `useReducer`

Objective: Rebuild the Todo List application (Exercise 5) using the `useReducer` Hook for state management.

Instructions:

1. Create a reducer function for your todo list that handles actions like `ADD_TODO`, `TOGGLE_TODO`, and `REMOVE_TODO`.
2. Modify the `TodoList` component to use `useReducer` instead of `useState` for managing the todo items.
3. Ensure all existing functionalities (adding, displaying) work correctly.
4. Add a new feature: a checkbox next to each todo item to mark it as completed/uncompleted. Update the reducer to handle the `TOGGLE_TODO` action.
5. Add a button to remove a todo item. Update the reducer to handle the `REMOVE_TODO` action.

Expected Output: A todo list with add, toggle completion, and remove functionalities, all managed by a `useReducer` hook.

Hints:

- The reducer function takes `state` and `action` as arguments.
- The `action` object should have a `type` property and potentially a `payload`.
- `useReducer` returns the current state and a `dispatch` function.

Exercise 15: Optimizing a List with `React.memo` and `useCallback`

Objective: Demonstrate performance optimization techniques (`React.memo` and `useCallback`) in a component with a list of items.

Instructions:

1. Create a `ListItem` functional component that displays an item and a button to perform an action (e.g., "Delete"). This component should be wrapped with `React.memo`.
2. In a parent component (`OptimizedList`), render a list of `ListItem` components.
3. Pass a callback function (e.g., `handleDeleteItem`) from the parent to each `ListItem`.
4. Use `useCallback` to memoize the `handleDeleteItem` function in the parent component, ensuring it's not recreated on every render unless its dependencies change.

5. Add a state in the parent component that, when updated, causes the parent to re-render but should *not* cause the `ListItem` components to re-render unnecessarily (unless their own props change).
6. Use `console.log` inside `ListItem` to observe when it re-renders.

Expected Output: A list of items where individual `ListItem` components only re-render when their specific props change, demonstrating the effectiveness of `React.memo` and `useCallback`.

Hints:

- `React.memo` performs a shallow comparison of props by default.
- `useCallback` requires a dependency array; an empty array means the function is created once.

Exercise 16: Implementing an Error Boundary

Objective: Create a reusable `ErrorBoundary` component to gracefully handle JavaScript errors in child components and display a fallback UI.

Instructions:

1. Create a class component named `ErrorBoundary`.
2. Implement `static getDerivedStateFromError(error)` to update the component's state (e.g., `hasError: true`).
3. Implement `componentDidCatch(error, info)` to log the error (e.g., to `console.error`).
4. In the `render` method, if `hasError` is true, display a fallback UI (e.g., "Something went wrong."). Otherwise, render `this.props.children`.
5. Create a `BuggyComponent` that intentionally throws an error under certain conditions (e.g., when a button is clicked a specific number of times).
6. Wrap the `BuggyComponent` with your `ErrorBoundary` in your `App` component.

Expected Output: When the `BuggyComponent` throws an error, the `ErrorBoundary` should catch it and display the fallback UI instead of crashing the entire application.

Hints:

- Error boundaries only catch errors in their child component tree, not within themselves.

- Errors in event handlers are not caught by error boundaries.

Exercise 17: Modal with Portals

Objective: Build a reusable modal component that renders outside the normal DOM hierarchy using `ReactDOM.createPortal`.

Instructions:

1. In your `public/index.html` file, add a new `div` element with an `id` (e.g., `modal-root`) outside the main `root` div.
2. Create a functional component named `Modal` that accepts `isOpen`, `onClose`, and `children` props.
3. Inside the `Modal` component, use `ReactDOM.createPortal` to render its `children` into the `modal-root` DOM node.
4. Implement basic modal styling (overlay, content box, close button).
5. Add an `onClick` handler to the overlay to close the modal when clicked outside the content.
6. In your `App` component, use the `Modal` component to display some content.

Expected Output: A modal window that appears on top of the main application content, and can be closed by clicking outside or on a close button.

Hints:

- Import `ReactDOM` from `react-dom`.
- Ensure the `modal-root` element exists in your `index.html`.
- Prevent event propagation from the modal content to the overlay to avoid accidental closing.

Exercise 18: Testing a Simple Component (Introduction to React Testing Library)

Objective: Write basic unit tests for a simple React component using React Testing Library.

Instructions:

1. Install necessary testing libraries: `npm install --save-dev @testing-library/react @testing-library/jest-dom jest` (if not already installed by `create-react-app`).

2. Create a test file (e.g., `Counter.test.js`) for your `Counter` component (from Exercise 3).
3. Write tests to:
    - Verify that the initial count is displayed correctly.
    - Simulate a click on the "Increment" button and assert that the count increases.
    - Simulate a click on the "Decrement" button and assert that the count decreases.
    - Simulate a click on the "Reset" button and assert that the count resets to 0.
4. Use `render`, `screen.getByText`, `fireEvent.click`, and `expect` from React Testing Library.

Expected Output: All tests for the `Counter` component pass successfully, demonstrating how to test component rendering and user interactions.

Hints:

- Focus on testing user behavior, not internal implementation details.
- Use `screen.getByRole` or `screen.getByText` to query elements.
- `fireEvent` simulates user interactions.

Exercise 19: Testing a Form Component

Objective: Write tests for a form component, covering input changes, validation, and submission.

Instructions:

1. Create a test file (e.g., `LoginForm.test.js`) for your `LoginForm` component (from Exercise 10).
2. Write tests to:
    - Verify that the form renders correctly with initial empty inputs.
    - Simulate typing into the email and password fields.
    - Assert that validation error messages appear when inputs are invalid.
    - Assert that the submit button is disabled when the form is invalid.
    - Simulate a successful form submission and assert that a success message is displayed.
3. Use `render`, `screen.getByLabelText`, `screen.getByRole`, `fireEvent.change`, `fireEvent.click`, and `expect`.

Expected Output: All tests for the `LoginForm` component pass, ensuring its functionality and validation logic work as expected.

Hints:

- Use `fireEvent.change` to simulate user input into form fields.
- Test both valid and invalid input scenarios.
- Assert the presence or absence of error messages.

This expanded set of exercises provides a deeper dive into advanced React concepts, preparing students for more complex real-world applications and professional development roles. Each exercise is designed to be hands-on, encouraging active learning and problem-solving.

Homework 1: Simple Express API with MongoDB

Objective: Create a basic Node.js/Express.js server that connects to a MongoDB database and implements CRUD operations for a simple resource.

Instructions:

1. Set up a new Node.js project with Express.js and Mongoose.
2. Create a MongoDB connection using Mongoose.
3. Define a `Task` schema with fields: `title` (required), `description`, `completed` (boolean, default false), `createdAt` (date).
4. Implement the following API endpoints:
    - `GET /api/tasks` - Retrieve all tasks
    - `POST /api/tasks` - Create a new task
    - `GET /api/tasks/:id` - Retrieve a specific task
    - `PUT /api/tasks/:id` - Update a task
    - `DELETE /api/tasks/:id` - Delete a task
5. Add proper error handling and validation.
6. Test all endpoints using a tool like Postman or Thunder Client.

Expected Output: A fully functional REST API that can perform CRUD operations on tasks stored in MongoDB.

Hints:

- Use environment variables for the MongoDB connection string.
- Implement proper HTTP status codes for different scenarios.
- Add middleware for parsing JSON and handling CORS.

Homework 2: API Authentication with JWT

Objective: Extend the previous API to include user registration and login with JSON Web Token (JWT) authentication.

Instructions:

1. Create a `User` schema with fields: `username` (unique), `email` (unique), `password` (hashed).
2. Implement password hashing using bcryptjs.
3. Create authentication endpoints:
   - `POST /api/auth/register` - User registration
   - `POST /api/auth/login` - User login
   - `GET /api/auth/profile` - Get user profile (protected)
4. Implement JWT token generation and verification middleware.
5. Protect the task endpoints so that users can only access their own tasks.
6. Add a `userId` field to the Task schema to associate tasks with users.

Expected Output: An authenticated API where users must register/login to access their personal tasks.

Hints:

- Never store plain text passwords in the database.
- Use middleware to verify JWT tokens on protected routes.
- Include user information in the JWT payload.

Homework 3: User Authentication Frontend

Objective: Implement user authentication in the React frontend with protected routes and persistent login state.

Instructions:

1. Create authentication components:
   - Login form
   - Registration form
   - User profile display
2. Implement authentication context using React Context API.
3. Create protected routes that require authentication.
4. Store JWT tokens securely and implement automatic logout on token expiration.
5. Add navigation that changes based on authentication status.
6. Implement form validation for registration and login.

Expected Output: A complete authentication system where users can register, login, and access protected content.

Hints:

- Use React Router for navigation and protected routes.
- Store tokens in localStorage but validate them on app startup.
- Implement automatic token refresh if your backend supports it.

Homework 4: Real-time Chat with Socket.io

Objective: Add real-time functionality to a MERN application using Socket.io for instant messaging.

Instructions:

1. **Backend Setup:**
    - Install and configure Socket.io with Express.js.
    - Create a `Message` schema with fields: `content`, `sender` (user reference), `room`, `timestamp`.
    - Implement socket event handlers for joining rooms, sending messages, and user presence.
2. **Real-time Features:**
    - Implement user authentication for socket connections.
    - Create chat rooms or direct messaging functionality.
    - Add typing indicators and online user status.
    - Store message history in MongoDB.
3. **Frontend Implementation:**
    - Install Socket.io client and create connection management.
    - Build chat interface with message display and input.
    - Implement real-time message updates and notifications.
    - Add emoji support and message formatting.

Expected Output: A real-time chat application integrated into the MERN stack with persistent message storage.

Hints:

- Authenticate socket connections using JWT tokens.
- Implement rate limiting to prevent spam.
- Consider implementing message encryption for security.

Homework 5: MERN E-commerce Platform

Objective: Build a comprehensive e-commerce platform with product management, shopping cart, and order processing.

Instructions:

1. **Database Design:**
    - Create schemas for Product, Category, Cart, Order, and OrderItem.
    - Implement product variants (size, color) and inventory management.
    - Design user address and payment information storage.
2. **Backend API:**
    - Implement product catalog with search, filtering, and pagination.
    - Create shopping cart functionality with session management.
    - Build order processing workflow with status tracking.
    - Add admin endpoints for product and order management.
3. **Frontend Development:**
    - Create product listing and detail pages with image galleries.
    - Implement shopping cart with quantity management.
    - Build checkout process with form validation.
    - Create user dashboard for order history and profile management.
    - Develop admin panel for product and order management.

Expected Output: A fully functional e-commerce platform with complete shopping and administrative functionality.

Hints:

- Implement proper inventory management to prevent overselling.
- Use transactions for order processing to ensure data consistency.
- Consider implementing payment gateway integration (Stripe, PayPal).