Conditional Rendering And useEffect

Software Development Bootcamp

Topic

Conditional Rendering

What Is Conditional Rendering?

Conditional rendering is the process of displaying different content based on certain conditions. In React, this means:

- Showing or hiding components
- Rendering different components based on state
- Handling loading and error states
- Managing user permissions and access

Why Use Conditional Rendering?

- Create dynamic user interfaces
- Handle different application states
- Control access to features
- Improve user experience
- Manage loading and error states

Basic Conditional Rendering

There are several ways to conditionally render content:

- If statements
- Ternary operators
- Logical && operator
- Switch statements

Why Ternary Operators Are Preferred

- Can be used directly inside JSX
- More concise and readable
- Maintains single return statement
- Works inside expressions
- Enables inline conditions
- Better fits React's declarative nature

Ternary Operator Review

A ternary operation has three parts:

```
condition ? valueIfTrue : valueIfFalse
```

Ternary Example

- If `isLoggedIn` is true, render `<UserProfile />`
- If `isLoggedIn` is false, render `<LoginForm />`
- `<Footer />` always renders
- All logic stays within a single return statement
- Helps maintain clean component structure

```
return (
   <div>
      {isLoggedIn ? <UserProfile /> :
<LoginForm />}
      <Footer /> {/* Other components can
follow */}
   </div>
);
```

Key Benefits

1. Inline Evaluation
    a. Conditions evaluate right where they're needed
    b. No need for separate logic blocks
2. Composition
    a. Can easily combine multiple conditions
    b. Works well with other React patterns
3. Readability
    a. Clear visual structure
    b. Easy to follow the logic flow
    c. Keeps related code together
4. Flexibility
    a. Works for components, props, and text
    b. Can be as simple or complex as needed
    c. Easy to modify and maintain

Common Use Cases For Conditional Rendering

- Authentication states
- Loading states
- Error handling
- Feature flags
- User permissions
- Responsive design

Topic

useEffect

What Is useEffect?

useEffect is a React Hook that lets you synchronize your component with external systems and handle side effects.

What Are Side Effects?

- Any operation that reaches outside the component's scope
- Actions that can't be done during rendering
- Operations that affect something outside the normal render flow

useEffect: Key Characteristics

1. **Timing**
    a. Runs after render
    b. Can run on every render
    c. Can run conditionally
    d. Can clean up after itself
2. **Dependencies**
    a. Controls when effect runs
    b. Can be empty array, no array, or with values
    c. React watches for changes

useEffect: Mental Model

- Think of effects as "synchronization"
- Component needs to sync with something external
- Effect runs when synchronization might be needed
- Cleanup handles "unsyncing"

useEffect: Important Considerations

1. **Effects should be focused**
    a. One responsibility per effect
    b. Easier to maintain and debug
    c. Clearer dependency arrays
2. **Dependencies**
    a. Include all values used from props/state
    b. Consider what changes should trigger effect
    c. Use dependency linting rules
3. **Performance**
    a. Don't overuse effects
    b. Batch related state updates
    c. Use cleanup functions properly

useEffect Data Fetching: State Setup

- `products` stores our fetched data
  - Starts empty `([])` to avoid null errors
  - Will hold an array of products once data arrives.
  - Using an empty array lets us map safely before data arrives
- `loading` tracks if we're fetching data
  - Starts true because we fetch when component starts
  - Changes to false when fetch completes
- `error` Stores any error messages
  - Starts as null
  - Gets set if fetch fails
  - Shows error messages to users

```
const [products, setProducts] = useState([]);
 const [loading, setLoading] = useState(true);
 const [error, setError] = useState(null);
```

useEffect Data Fetching: Effect Setup

- `useEffect` Manages our side effect (data fetching)
  - Runs after component first appears
  - Empty dependency array ([]) means run once
  - Perfect for initial data fetching
- `async function` Handles the fetch operation
  - Created inside useEffect to keep it contained
  - Uses try/catch to handle errors
  - Updates our three states as needed

```
useEffect(() => {
  // Function to fetch our products
  async function fetchProducts() {
    try {
      const response = await
fetch('https://api.example.com/products');
      const data = await response.json();
      setProducts(data);
    } catch (err) {
      setError('Could not fetch products');
    } finally {
      setLoading(false);
    }
  }
  // Call the function
  fetchProducts();
}, []); // Empty array means run once when
component starts
```

useEffect Data Fetching: Displaying Data

- Only runs if loading and error checks pass
- Maps over our products array
- Each product needs a unique key
- Shows the actual data to users

```
return (
  <div className="product-list">
    <h2>Products</h2>
    {products.map(product => (
      <div key={product.id}
className="product">
        <h3>{product.name}</h3>
        <p>${product.price}</p>
      </div>
    ))}
  </div>
);
```

Exercise

Post Selector