Lists & Keys in React

What is List Rendering?

List rendering in React means dynamically creating UI elements by iterating over a collection of data, typically an array. This is most commonly achieved using the JavaScript `Array.prototype.map()` method directly within your JSX. Each item in the array is transformed into a React element.

Explanation: When rendering lists, React needs a way to efficiently update and reconcile the UI when the list changes (items are added, removed, or reordered). This is where **keys** come in.

- **Keys help React identify which items in a list have changed, been added, or been removed.** They provide a stable identity to each list item.
- **Keys must be unique** among sibling elements within the same list.
- **Keys should be stable and persistent** across re-renders. This means the key for a specific item should not change if that item remains in the list.

Example: Rendering a List of Users

```
const users = [
  { id: 1, name: "Ali" },
  { id: 2, name: "Sara" },
  { id: 3, name: "Farhan" }
];

function UserList() {
  return (
    <ul>
      {users.map(user => (
        // The 'key' prop is crucial here
        <li key={user.id}>{user.name}</li>
      ))}
    </ul>
  );
}
```

Important: The `key` prop is not directly accessible within the component itself (e.g., you can't access `props.key`). It's a special prop used internally by React.

Why use `key` in lists?

- **Performance:** Keys allow React to efficiently re-render lists. When a list changes, React uses the keys to figure out exactly which items need to be updated, instead of re-rendering the entire list.
- **Correctness:** Without stable keys, React might re-use component instances for different data items, leading to unexpected behavior, incorrect state, or bugs, especially in complex components with internal state.
- **Identity:** Keys provide a unique and stable identity to each list item, even if their data or position changes.

Do NOT use `index` as `key` if the list can change order, be filtered, or have items added/removed in the middle.

- While using `index` from `map((item, index) => ...)` might seem convenient, it's problematic if the order of items in the array can change, or if items can be added/removed from the middle. If the order changes, React might incorrectly optimize by thinking existing components simply moved, when in fact they represent different data. This can lead to subtle and hard-to-debug UI issues, especially with interactive list items (e.g., input fields losing focus, wrong checkbox being checked).
- **Best Practice:** Always try to use a stable, unique ID from your data (like `user.id` from a database) as the key. If you genuinely have no stable IDs and the list order never changes, using the index can be a fallback, but be aware of the pitfalls.

Conditional Rendering

What is Conditional Rendering?

Conditional rendering in React means rendering different components or elements based on certain conditions. This allows you to create dynamic UIs that respond to changes in state, props, user input, or data.

Explanation: You will frequently want to render different content based on various conditions, such as:

- Whether a user is logged in.
- If data has finished loading from an API.

- If an error occurred.
- Toggling visibility of UI elements.

React's declarative nature makes conditional rendering straightforward using standard JavaScript operators.

Example: Show/Hide Message (using `&&` logical AND)

JavaScript

```
import { useState } from 'react'; // Assuming useState is imported from
previous hour

function MessageToggle() {
  const [isVisible, setIsVisible] = useState(true);

  return (
    <div>
      <button onClick={() => setIsVisible(!isVisible)}>
        {isVisible ? "Hide" : "Show"} Message
      </button>
      {/* Short-circuiting: If isVisible is true, <p>Hello! I am visible.</p>
is rendered.
          If isVisible is false, the expression short-circuits, and nothing
is rendered. */}
      {isVisible && <p>Hello! I am visible.</p>}
    </div>
  );
}
```

Common Conditional Rendering Techniques:

1. **Ternary Operator (`condition ? expressionIfTrue : expressionIfFalse`)**: Excellent for choosing between two distinct outcomes.

   ```
   function AuthStatus({ isLoggedIn }) {
     return (
       <div>
         {isLoggedIn ? <Dashboard /> : <LoginForm />}
       </div>
     );
   }
   ```

2. **Logical AND (`&&`) Short-Circuiting**: Useful for rendering something *only if* a condition is true, otherwise rendering nothing.

   JavaScript

```
function CartSummary({ cart }) {
  return (
    <div>
      <h3>Your Cart</h3>
      {/* Renders CartSummary component only if cart.length is greater
than 0 */}
      {cart.length > 0 && <p>You have {cart.length} items in your
cart.</p>}
      {/* More complex CartSummary component can go here */}
    </div>
  );
}
```

3. **`if/else` Statements (within functions, before `return`):** You can use standard JavaScript `if`/`else` or `switch` statements inside your component functions (before the `return` statement) to return different JSX based on conditions.

```
function Greeting({ user }) {
  if (user) {
    return <h1>Welcome back, {user.name}!</h1>;
  }
  return <h1>Please log in.</h1>;
}
```

`useEffect` Hook (Side Effects)

What is `useEffect`?

The `useEffect` Hook allows you to perform "side effects" in functional components. Side effects are operations that interact with the outside world or have observable effects beyond rendering the component itself.

Explanation:

- **Runs After Render:** `useEffect` runs *after* every render of your component (unless you specify dependencies). This means React guarantees the DOM is updated before your effect runs.
- **Accepts a Callback Function:** The first argument to `useEffect` is a function where you put your side effect logic.
- **Dependency Array:** The optional second argument is a dependency array. This array controls *when* the effect re-runs.

- **Common Side Effects:**
    - **Data Fetching:** Making API calls to load data.
    - **Subscriptions:** Setting up event listeners (e.g., for real-time updates).
    - **DOM Manipulation:** Directly interacting with the browser's DOM (e.g., changing a document title).
    - **Timers:** Setting up `setInterval` or `setTimeout`.
    - Logging (though `console.log` can be done anywhere, `useEffect` is good for observing render cycles).

Basic Example: Timer (with Cleanup)

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

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

  useEffect(() => {
    // This effect runs once on mount, and sets up an interval
    const interval = setInterval(() => {
      setCount(prevCount => prevCount + 1); // Use functional update for
setCount
    }, 1000);

    // Cleanup Function: This function is returned from the effect.
    // It runs when the component unmounts OR before the effect re-runs (if
dependencies change).
    // Essential for preventing memory leaks (e.g., clearing timers,
unsubscribing).
    return () => clearInterval(interval);
  }, []); // Empty dependency array means this effect runs ONLY once on mount
and cleans up on unmount.

  return <p>Timer: {count}</p>;
}
```

Cleanup Functions: Returning a function from `useEffect` is crucial for "cleaning up" side effects. This prevents memory leaks and ensures your application behaves correctly when components unmount or dependencies change. Common cleanups include:

- Clearing timers (`clearInterval`, `clearTimeout`).
- Unsubscribing from event listeners or web sockets.
- Canceling pending network requests.

Dependency Array Behavior (Crucial for Control)

The dependency array (`[]`) is the second argument to `useEffect` and dictates when your effect function will re-run.

1. **No Dependency Array:**

```
useEffect(() => {
  console.log("Runs after EVERY render (initial and updates)");
  // Be cautious! Can lead to infinite loops if state is updated inside
});
```

   - The effect runs after *every* render, regardless of what changed. Rarely desired for side effects like API calls.

2. **Empty Dependency Array (`[]`):**

```
useEffect(() => {
  console.log("Runs ONLY once on initial mount, and cleans up on
unmount");
}, []);
```

   - This is typically used for effects that should only run once when the component first appears on the screen (e.g., initial data fetching, setting up global event listeners). The cleanup function (if any) runs when the component unmounts.

3. **Dependency Array with Values (`[dep1, dep2]`):**

```
useEffect(() => {
  console.log("Runs on initial mount AND whenever 'count' changes");
  // This effect logic will execute if 'count' changes between renders
}, [count]); // 'count' is a dependency
```

   - The effect runs once on initial mount, and then again **only if any of the values in the dependency array change** between renders.
   - **Rule of Thumb:** If your effect uses any variables (props, state, or functions) defined *outside* the effect's scope, they should usually be included in the

dependency array. If you omit them, your effect might use "stale" (outdated) values.

API Calls with `useEffect` (Important Real-World Scenario)

A very common use case for `useEffect` is fetching data from an API when a component loads.

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

function DataFetcher() {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    async function fetchData() {
      try {
        setLoading(true); // Indicate loading has started
        const response = await
fetch('https://jsonplaceholder.typicode.com/todos/1');
        if (!response.ok) { // Check for HTTP errors (e.g., 404, 500)
          throw new Error(`HTTP error! status: ${response.status}`);
        }
        const json = await response.json();
        setData(json);
      } catch (e) {
        console.error("Fetching error: ", e);
        setError(e.message); // Set error state
      } finally {
        setLoading(false); // Indicate loading has finished
      }
    }

    fetchData(); // Call the async function
  }, []); // Empty dependency array: fetch data only once on mount

  if (loading) return <p>Loading data...</p>;
  if (error) return <p>Error: {error}</p>;
  if (!data) return <p>No data found.</p>; // Should not happen if
loading/error are handled

  return (
    <div>
      <h3>Fetched Todo</h3>
      <p>ID: {data.id}</p>
      <p>Title: {data.title}</p>
      <p>Completed: {data.completed ? "Yes" : "No"}</p>
    </div>
  );
}
```

Key points in the API Call example:

- Using an `async` function inside `useEffect` for cleaner `await` syntax.
- Handling **loading states**: Shows "Loading..." to the user.
- Handling **error states**: Catches network errors or bad responses and displays an error message. This is critical for robust applications.
- `finally` block to ensure `setLoading(false)` runs whether successful or not.
- Empty dependency array `[]` ensures the data is fetched only once when the component mounts.

Forms & Input Handling

Controlled Components in React

In React, "controlled components" are the recommended way to handle form inputs. In a controlled component, the form elements (like `<input>`, `<textarea>`, `<select>`) are directly linked to and controlled by the component's state. The component's state becomes the "single source of truth" for the input's value.

Explanation:

- **Value is Controlled by State:** The `value` attribute of the input element is set by a piece of state.
- **Changes Handled by `onChange`:** Whenever the input's value changes (e.g., user types), the `onChange` event handler fires. This handler updates the component's state with the new value.
- **Predictable Behavior:** This pattern ensures that the React state always reflects what's in the input field, making it easy to validate, manipulate, and submit form data.

Example: Simple Form (Functional Component)

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

function ContactForm() {
  // Declare state variables for each input field
  const [name, setName] = useState("");
  const [email, setEmail] = useState("");

  function handleSubmit(e) {
    e.preventDefault(); // Prevents the default browser form submission (page
reload)
    alert(`Submitted:\nName: ${name}\nEmail: ${email}`);
    // Clear the form after submission
    setName("");
    setEmail("");
  }

  return (
    <form onSubmit={handleSubmit}>
      {/* Name input: value linked to 'name' state, onChange updates 'name'
state */}
      <label htmlFor="nameInput">Name:</label>
      <input
        id="nameInput"
        type="text"
        value={name}
        onChange={e => setName(e.target.value)}
        placeholder="Your Name"
        required // Basic HTML5 validation
      />
      <br />
      {/* Email input: value linked to 'email' state, onChange updates
'email' state */}
      <label htmlFor="emailInput">Email:</label>
      <input
        id="emailInput"
        type="email"
        value={email}
        onChange={e => setEmail(e.target.value)}
        placeholder="Your Email"
        required
      />
      <br />
      <button type="submit">Submit</button>
    </form>
  );
}
```

Class Component Form Example:

Handling forms in class components follows a similar principle, but uses `this.state` and `this.setState()`.

```
import React from 'react';

class LoginForm extends React.Component {
  constructor(props) { // Constructor accepts props and calls super(props)
    super(props);
    this.state = { email: '', password: '' }; // Initialize state object
  }

  // Arrow function for handleChange ensures 'this' context is bound
correctly
  handleChange = (e) => {
    // Uses computed property names ([e.target.name]) to update state
dynamically
    // based on the input's 'name' attribute.
    this.setState({ [e.target.name]: e.target.value });
  }

  handleSubmit = (e) => {
    e.preventDefault(); // Prevents page reload
    alert(`Logging in with:\nEmail: ${this.state.email}\nPassword:
${this.state.password}`);
    // You would typically send this data to an API here
    // Optionally clear form: this.setState({ email: '', password: '' });
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label htmlFor="loginEmail">Email:</label>
        <input
          id="loginEmail"
          type="email"
          name="email" // 'name' attribute is crucial for dynamic
handleChange
          value={this.state.email}
          onChange={this.handleChange}
          required
        />
        <br />
        <label htmlFor="loginPassword">Password:</label>
        <input
          id="loginPassword"
          type="password"
          name="password" // 'name' attribute
```

```
          value={this.state.password}
          onChange={this.handleChange}
          required
        />
        <br />
        <button type="submit">Login</button>
      </form>
    );
  }
}
```

Uncontrolled Components (Brief Mention)

While controlled components are the standard, React also supports "uncontrolled components" where the form data is handled by the DOM itself, often using `refs` to get their values directly. These are generally less preferred for most use cases due to less immediate control and harder validation, but they exist. (No need for detailed demo here unless specifically requested).

Summary Checklist

- List rendering with `map()`
- Proper use and importance of `key` in lists (unique, stable, avoid index)
- Conditional rendering techniques (ternary, `&&` short-circuiting, `if/else`)
- `useEffect` for side effects and component lifecycles (mount, update, unmount)
- Understanding the `useEffect` dependency array (empty, with dependencies, no array)
- Essential `useEffect` cleanup functions to prevent memory leaks
- Practical example of data fetching with `useEffect`, including loading and error handling
- Controlled forms using `useState` (functional) and `this.state` (class)
- Form handling in both component types
- `e.preventDefault()` for form submission control