Question 1: How did you manage to fetch the list and what tool did you use?

Ans:

Steps Taken:

1. **Used the `fetch` API and used it Reuable hook called useFetchPokemon**:
   - I created an asynchronous function to make a GET request to the Pokémon API.
   - Parsed the response using `.json()` to extract the data.
2. **Managed State**:
   - The fetched list was stored in a local state using **React's `useState` hook**.
   - The component re-rendered dynamically to display the Pokémon list once the data was fetched.
3. **Handled Side Effects**:
   - Leveraged **React's `useEffect` hook** to trigger the data fetch when the component mounted.

Question 2: What steps would you take to future improve this?

Ans:

1. Use Redux Toolkit for State Management

- Introduce **Redux Toolkit** to manage the Pokémon list globally. This would make the state easily accessible across multiple components, reducing the need to pass props.

2. Pagination or Infinite Scrolling

- Instead of fetching all Pokémon at once, implement pagination or infinite scrolling to load data in smaller batches, improving performance and user experience.

Question 4: What makes the createSlice in redux-toolkit difference then A Reducer in redux?

Ans:

- createSlice automatically generates action creators and action types, which reduces boilerplate code.
- Built-in Immutability with Immer in createSlice.
  Uses **Immer.js** internally to allow writing "mutating" code (like state.list.push) while producing an immutable state.
- Integration with Redux DevTools
  • Traditional Reducer: Requires explicit naming conventions to make actions identifiable in DevTools.
  • createSlice: Automatically names actions in the format sliceName/actionName (e.g., pokemon/removePokemon), improving traceability in Redux DevTools.

Question 5: Describe the benefits of immutable code.

Ans:

Thread Safety

- Immutable objects are safe to use when multiple processes or threads work at the same time. Since the data doesn't change, there's no risk of one thread messing up another's work.

Better Security

- Immutable objects are safer because they can't be altered after creation. For example, storing passwords or cryptographic keys in immutable objects ensures they won't accidentally change or be tampered with.

Easier to Test

- Since immutable objects never change, you know exactly how they behave. This makes it much simpler to write and run tests without worrying about changes affecting your results.

Fewer Errors

- When something can't be changed, it's less likely to cause mistakes. Immutable objects stay the same, reducing the risk of accidental bugs.

More Readable and Efficient

- Using immutable objects like strings makes your code clearer and often faster to run since the system doesn't need to worry about tracking changes.

Fewer Bugs

- With immutable data, you don’t need to worry about unexpected changes causing issues. This makes it easier to pinpoint and fix problems.

Consistent and Reliable

- Immutable systems stay in a stable, predictable state because you replace parts instead of modifying them. This is like swapping in a new piece rather than trying to fix an old one.

Stronger Security for Systems

- Immutable systems, like servers, are more secure because they aren't changed after they're set up. This makes it harder for attackers to exploit them.

6: How can you verify the action has been dispatched?

Ans

1. Use Redux DevTools

2. Log Actions in Middleware

3. Write Unit Tests for Dispatch

Question 7: explain the use of `useEffect` hook in React Question

Ans:

Key Uses:

1. **Data Fetching:** Fetch data when a component loads.
2. **Subscriptions:** Add/remove event listeners or subscriptions.
3. **DOM Updates:** Update page titles or manipulate the DOM.
4. **Reacting to Changes:** Re-run logic when specific values (dependencies) change.

9: What use cases would a HOC be usefull? Question

Ans:

Key Use Cases:

1. **Code Reuse**: Share logic (e.g., authentication) across multiple components.
2. **Conditional Rendering**: Render components based on certain conditions (e.g., user role).
3. **Enhancing Props**: Modify or add props to a component.
4. **Global State Access**: Provide context or global state to components.
5. **Logging/Analytics**: Track component rendering or user interactions.
6. **Permission Management**: Restrict access based on user roles.
7. **Error Boundaries**: Catch errors in components for graceful fallback.
8. **Lazy Loading**: Load components dynamically to improve performance.
9. **Component Composition**: Combine smaller components with shared functionality.

When to Use HOCs:

- To **reuse logic** across multiple components.
- To **enhance components** without modifying them directly.

10: What does it indicate when a component is prefixed with `use` and `with` Question

Ans:

The `use` prefix signals that the component or function is a **custom hook** in React.
Custom hooks are used for **reusable logic** in functional components and must follow the **rules of hooks**.

11: What is a Generic type in typescript? Question

Ans:

A **generic type** in TypeScript allows you to define a function, class, or interface that can work with any data type while still maintaining type safety. It acts as a placeholder for a specific type, which is provided later when the function, class, or interface is used.

12: Whats the difference between a controlled and uncontrolled input in React?

Ans:

Controlled Input:

- The value of the input is controlled by React state.
- The input's value is updated through `useState` or component state.
- Every change triggers a re-render.
-

Uncontrolled Input:

- The value is managed by the DOM, not React.
- You access the value using a `ref`.
- No re-renders on value changes.