Assignment `09 - Optimizing our App`

Owner |  Pankaj Kumar

1: When and why do we need lazy()?

- The `lazy()` function is a feature introduced in React 16.6 that allows for the lazy loading of components.
- The lazy() function is typically used in scenarios where you have large or less frequently used components that you want to load `asynchronously`.

▼ A few situations when you might need to use lazy() are:

1. Large component bundles: If your application has large components or dependencies, loading them synchronously during the initial render can cause significant delays. By using lazy() along with code splitting, you can split these components into separate chunks and load them only when necessary, improving the overall performance of your application.

2. Infrequently accessed routes: If you have certain routes or pages in your application that are rarely accessed, it might be inefficient to load their associated components upfront. Using lazy() allows you to lazily load these components when the specific route is visited, reducing the initial bundle size and improving the application's initial load time.

3. Enhancing performance: By employing lazy() and code splitting, you can ensure that only the necessary components are loaded upfront, while the rest are loaded on-demand. This approach helps reduce the initial bundle size and improves performance by minimizing the amount of JavaScript that needs to be downloaded and executed during the initial page load.

2: What is suspense?

- Suspense is a component that helps manage the loading state of dynamic imports, such as lazily loaded components, and provides a fallback UI to display while the requested content is being loaded. It enables a better user experience by showing a loading indicator or placeholder content until the desired component or data is ready to be rendered..
- Suspense component allows us to show some fallback content (such as a loading indicator/ Shimmer component) while we're waiting for the lazy component to load or the component is not yet rendered. It is similar to `catch` block. If a component suspends, the closest `Suspense` component above the suspending component `catches` it

```
import React, { Suspense } from 'react';

const About = React.lazy(() => import('./About'));

function MyComponent() {
return (
  <div>
    <Suspense fallback={<div>Loading...</div>}>
      <About />
    </Suspense>
  </div>
);
}
```

- The `fallback` prop accepts any `React elements` that you want to render while waiting for the component to load. You can place the Suspense component anywhere above the lazy component. You can even wrap `multiple lazy components` with a `single` Suspense component.

3: Why we got this error : A component suspended while responding to synchronous input. This will cause the UI to be replaced with a loading indicator. To fix, updates that suspend should be wrapped with startTransition? How does suspense fix this error?

This error is thrown as Exception by React when the promise to dynamically import the lazy component is not yet resolved and the Component is expected to render in the meantime. If only the dynamic import is done and there is no `<Suspense />` component then this error is shown. React expects a Suspense boundary to be in place for showing a fallback prop until the promise is getting resolved. If showing the shimmer (loading indicator) is not desirable in some situations, then `startTransistion` API can used to show the old UI while new UI is being prepared. React do this without having to delete or remove the Suspense component or its props from your code.

4: Advantages and disadvantages of using this code splitting pattern?

Advantages	Disadvantages
1. Improved performance: Code splitting allows you to load only the necessary parts of your application when they are needed, reducing the initial bundle size. This results in faster load times and improved overall performance, particularly for larger applications.	**1. Increased complexity:** Code splitting adds complexity to your application's architecture, requiring you to manage and coordinate the loading of different code chunks. This can make the development process more challenging and may require additional tooling and configuration.
2. Faster initial load: By deferring the loading of non-essential code until it is actually required, code splitting can significantly reduce the time it takes for the initial page to load. This can greatly enhance the user experience, especially for users with slower internet connections or on mobile devices.	**2. Potential for suboptimal user experience:** If code splitting is not implemented carefully, it can lead to suboptimal user experiences. Poorly optimized code splitting may result in visible loading delays or multiple small network requests, which can negatively impact the perceived performance of your application.
3. Better resource utilization: With code splitting, you can optimize resource utilization by loading code chunks only when they are needed. This can reduce memory consumption and improve the efficiency of your application.	**3. Additional network requests:** Code splitting can lead to an increase in the number of network requests made by your application. While these requests may be smaller in size, the total number of requests may still impact the overall load time, especially in scenarios with slower network connections.

Advantages	Disadvantages
4. Enhanced caching: Code splitting enables better caching and reusability of code. Once a code chunk is loaded, it can be cached by the browser, allowing subsequent visits to your application to benefit from faster load times.	**4. Compatibility concerns:** Code splitting relies on newer web standards and features, such as dynamic imports and ES modules. While these features are widely supported in modern browsers, older browsers may not fully support them, potentially leading to compatibility issues.
5. Smoother user interaction: By loading code asynchronously and showing loading indicators or fallback content during the loading process, code splitting provides a more seamless user experience. It prevents the entire application from freezing or becoming unresponsive while waiting for large chunks of code to load.	**5. Build and deployment complexity:** Implementing code splitting may require additional build and deployment configurations and tools. This can introduce complexity to your development workflow and may require learning and integrating new tools into your existing build process.

5: When do we and why do we need suspense?

- **Lazy loading components:** When you want to load components lazily or on-demand, Suspense allows you to specify fallback content that will be displayed while the component is being loaded. This helps improve the user experience by showing a loading indicator or placeholder content until the component is ready to render.
- **Data fetching:** If your application needs to fetch data from an API or perform asynchronous operations, Suspense can be used to handle the loading state and display fallback content until the data is available. This simplifies the management of loading states and ensures a smooth transition between loading and displaying the data.
- **Code splitting:** Code splitting involves breaking your application's code into smaller chunks that can be loaded separately. When using dynamic imports and code splitting techniques, Suspense can be used to wrap the components that are being lazily loaded. It allows you to provide fallback content while the code chunks are being fetched and loaded, improving the perceived performance of your application.
- **Concurrent mode (experimental):** React's concurrent mode, introduces new features like rendering fallback content for slow components and prioritizing

updates. Suspense plays a crucial role in managing concurrent rendering and helps coordinate the rendering and fallback states of components.