React 11 - React Perf and Rendering

- React Perf
    - Code splitting
    - dynamic imports
    - Lazy Loading
    - React.Lazy and
    - react suspense
- Memoization
    - React.memo
    - useMemo
    - useCallBack

title: Code Splitting and Lazy Loading

Scenario Without Code Splitting and Lazy Loading

In a standard setup, all of your JavaScript code, including all the components, gets bundled into one large file during the build process.

When a user accesses your application, this entire bundle is loaded and executed in the browser.

Suppose we have this app with three compnents HomePage,AboutPage,ContactPage and a Navbar component to navigate between them

Below is an example of how you can create HomePage, AboutPage, and ContactPage components and implement routing for these components using react-router-dom.

First, you need to install react-router-dom if you haven't already:

npm install react-router-dom

Then, you can create your components and implement the routing as follows:

Create the HomePage Component:

```
const HomePage = () => {
   return (
     <div>
       <h1>Home Page</h1>
       <p>Welcome to the Home Page!</p>
     </div>
   );
 };
  export default HomePage;
```

Create the AboutPage Component:

```
// AboutPage.jsx
import React from 'react';
```

```
const AboutPage = () => {
 return (
   <div>
     <h1>About Page</h1>
     <p>Learn more about us on this page.</p>
   </div>
 );
};

export default AboutPage;
```

Create the ContactPage Component:

```
// ContactPage.jsx
import React from 'react';

const ContactPage = () => {
 return (
   <div>
     <h1>Contact Page</h1>
     <p>Get in touch with us through this page.</p>
   </div>
 );
};

export default ContactPage;
```

Create the Navbar Component:

```
// Navbar.jsx
import React from 'react';
import { Link } from 'react-router-dom';

const Navbar = () => {
 return (
   <nav>
     <ul>
       <li>
         <Link to="/">Home</Link>
       </li>
       <li>
         <Link to="/about">About</Link>
       </li>
       <li>
         <Link to="/contact">Contact</Link>
       </li>
     </ul>
   </nav>
 );
};

export default Navbar;
```

Set Up Routing in the App Component:

```
// App.jsx
import React from 'react';
import { BrowserRouter as Router, Route, Routes } from 'react-router-dom';
import HomePage from './components/HomePage';
import AboutPage from './components/About';
import ContactPage from './components/Contact';
import Navbar from './components/Navbar';

const App = () => {
 return (
   <Router>
     <div>
       <Navbar />
       <Routes>
         <Route path="/" element={<HomePage />} />
         <Route path="/about" element={<AboutPage />} />
         <Route path="/contact" element={<ContactPage />} />
       </Routes>
     </div>
   </Router>
 );
};

export default App;
```

This setup creates a simple React application with three pages: Home, About, and Contact, and a navbar to navigate between them using react-router-dom. When you click on the links in the navbar, the corresponding page component will be rendered.

In this example, all the components (HomePage, AboutPage, and ContactPage) are loaded at once, regardless of whether the user needs them immediately or not.

typically we do not need to create a build if we are running our application in development mode when using Vite.

In development mode, Vite uses a different approach for serving modules compared to a production build. Vite serves each module separately, which means you might not see a single large bundle like in a production build.

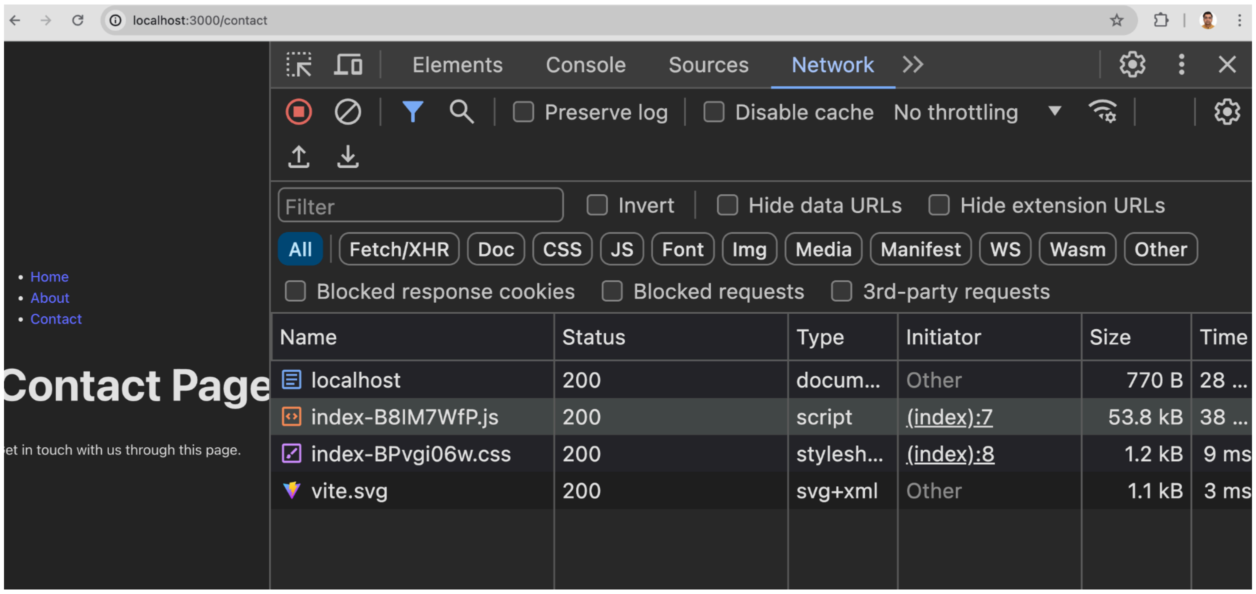
To simulate how it looks in production, we create a build first

npm run build

Then we serve this build (which serves the static bundled files) serve is a static file server that you can use to serve your production build locally. It helps you see how your application will perform in a production environment without deploying it to a remote server.

sudo npm install -g serve

serve -s dist - Here, -s stands for "single-page application" mode, which ensures that all routes are served correctly.



Name	Status	Type	Initiator	Size	Time
localhost	200	docum...	Other	770 B	28 ...
index-B8IM7WfP.js	200	script	(index):7	53.8 kB	38 ...
index-BPvgi06w.css	200	stylesh...	(index):8	1.2 kB	9 ms
vite.svg	200	svg+xml	Other	1.1 kB	3 ms

Here the complete bundle has been loaded which has loaded all the components regardless of if need them or not

Problems That Can Arise in this scenario

1. Long Initial Load Time: The initial load time can be significantly high as the browser needs to download a large JavaScript bundle before rendering any content.
2. Poor Performance: Large bundle sizes can lead to performance issues, especially on slower networks or less powerful devices.
3. Unnecessary Resource Usage: Users might download code for components and features they never interact with, leading to wasted resources.

title: Dynamic Imports

Dynamic import is a powerful feature in JavaScript that allows you to load modules asynchronously. This means that instead of loading all your JavaScript code upfront, you can load parts of it on demand. This can be particularly useful in large applications where you want to optimize performance and reduce initial load times.

In React, dynamic import can be used to load components only when they are needed. This helps in splitting your code and loading it in smaller chunks, which is often referred to as "code splitting".

Let's take a same example and implement dynamic import using functional components and hooks.

What is Dynamic Import?

Dynamic import is a feature that allows you to import JavaScript modules (including React components) dynamically and asynchronously. Instead of loading all components at once, you can load them on demand, which can significantly reduce the initial load time of your application. This process is known as code splitting.

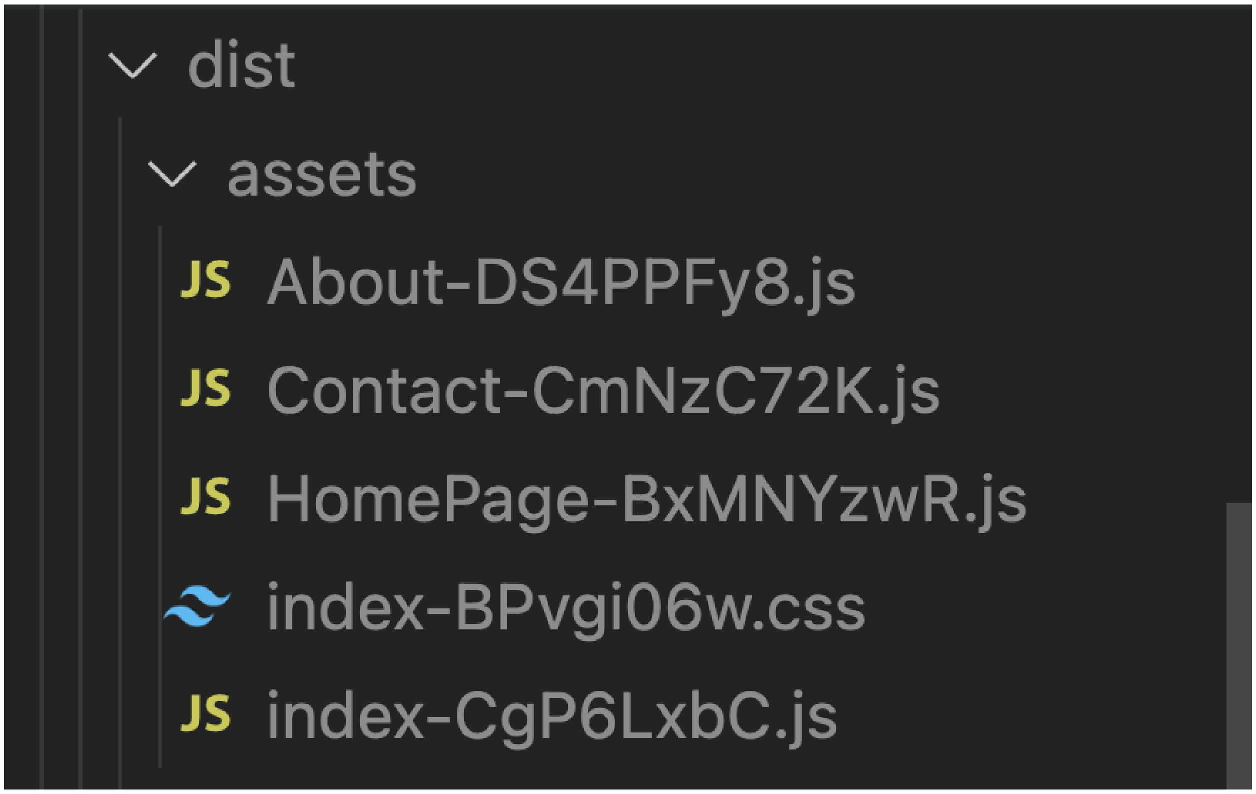
How to do it

1. Instead of importing the components at the top of the file, we use the import() function to load them dynamically within the routes.
2. Modify App Component to Use Dynamic Imports:

```
// App.jsx
import React, { useState, useEffect } from "react";
import { BrowserRouter as Router, Route, Routes } from "react-router-dom";
import Navbar from "./components/Navbar";

const App = () => {
 const [HomePage, setHomePage] = useState(null);
 const [AboutPage, setAboutPage] = useState(null);
 const [ContactPage, setContactPage] = useState(null);

 useEffect(() => {
   // Preload HomePage component
   import("./components/HomePage").then((module) => setHomePage(() =>
module.default));
 }, []);

 const loadHomePage = () => {
   import("./components/HomePage").then((module) => setHomePage(() =>
module.default));
 };

 const loadAboutPage = () => {
   import("./components/About").then((module) => setAboutPage(() =>
module.default));
 };

 const loadContactPage = () => {
   import("./components/Contact").then((module) =>
     setContactPage(() => module.default)
   );
 };


 return (
   <Router>
     <div>
       <nav>
         <ul>
           <li>
             <Link to="/" onClick={loadHomePage}>
               Home
             </Link>
           </li>
           <li>
             <Link to="/about" onClick={loadAboutPage}>
               About
             </Link>
           </li>
           <li>
             <Link to="/contact" onClick={loadContactPage}>
               Contact
             </Link>
           </li>
         </ul>
       </nav>{" "}
       <Routes>
         <Route
           path="/"
           element={HomePage ? <HomePage /> : <div>Loading...</div>}
         />
         <Route
           path="/about"
           element={AboutPage ? <AboutPage /> : <div>Loading...</div>}
         />
```

```
          <Route
            path="/contact"
            element={ContactPage ? <ContactPage /> : <div>Loading...</div>}
          />
        </Routes>
      </div>
    </Router>
 );
};

export default App;
```

Notice the different chunks that gets created when we build this again



```
dist
  assets
    About-DS4PPFy8.js
    Contact-CmNzC72K.js
    HomePage-BxMNYzwR.js
    index-BPvgi06w.css
    index-CgP6LxbC.js
```

How Dynamic Import Optimizes the Code

Code Splitting:

Dynamic imports create separate chunks for each component. When you run npm run build, Vite will create separate files for HomePage, AboutPage, and ContactPage. These files are only loaded when needed.

On-Demand Loading:

Instead of loading all components at once, components are loaded only when a user navigates to the respective route. This reduces the initial load time, making the app faster to start.
Improved Performance:

By splitting the code and loading components on-demand, you reduce the size of the initial JavaScript bundle. This can lead to faster page loads and improved performance, especially for users with slower network connections.

Approach Analysis

In this implementation:

We maintain state using the useState hook to keep track of whether each component has been loaded.

We use the import() function to dynamically load the components when a button is clicked.

The import() function returns a promise that resolves to the module object, from which we can access the default export (the component itself).

Once the component is loaded, we update the state to render the component.

Now you will see whenever you click on a button only the component associated with that button will get loaded and chunks will be created for each respective component separately

While dynamic import as shown in the example above can be a powerful tool, there are a few reasons why it might not be the best approach for every situation.

1. Manual State Management: In the given example, we manually manage the state for each dynamically imported component. This can become cumbersome and error-prone as the number of components grows, leading to more boilerplate code and potential bugs.
2. Lack of Built-in Fallback UI: The example does not provide a built-in way to show a fallback UI while the component is

loading. While we can add this manually, it requires additional code and effort. React.Suspense offers a straightforward way to handle this. We will see this next

3. Complexity: The code for manually importing and managing state can become complex, especially in larger applications. This complexity can make the code harder to maintain and understand.
4. Best Practices: Using React.lazy and Suspense is a React-recommended approach for code splitting and lazy loading. It leverages React's built-in mechanisms, ensuring better integration and reliability.

title: Lazy and Suspense

Implementing Code Splitting with lazy and suspense

React.lazy() allows you to define a component that is loaded dynamically. React.Suspense provides a way to handle the loading state while the component is being loaded.

1. Setup the Project:
    a. Get basic project setup with react-router-dom installed.
2. Update Component Imports to Use React.lazy():

   a. Modify the imports in App.jsx to use React.lazy() for dynamic imports.

3. Wrap Components with React.Suspense:
   a. Use React.Suspense to handle the loading state with a fallback component.

```
// App.jsx
// import React, { useState, useEffect } from "react";
import React, { Suspense, lazy } from "react";
import { BrowserRouter as Router, Route, Routes, Link } from "react-router-dom";
// import HomePage from './components/HomePage';
// import AboutPage from './components/About';
// import ContactPage from './components/Contact';
// import Navbar from './components/Navbar';
// Lazy load the components
const HomePage = lazy(() => import("./HomePage"));
const AboutPage = lazy(() => import("./AboutPage"));
const ContactPage = lazy(() => import("./ContactPage"));

const App = () => {
 // const [HomePage, setHomePage] = useState(null);
 // const [AboutPage, setAboutPage] = useState(null);
 // const [ContactPage, setContactPage] = useState(null);

 // useEffect(() => {
 //   // Preload HomePage component
 //   import("./components/HomePage").then((module) =>
 //     setHomePage(() => module.default)
 //   );
 // }, []);

 // const loadHomePage = () => {
 //   import("./components/HomePage").then((module) =>
 //     setHomePage(() => module.default)
 //   );
 // };
 // const loadAboutPage = () => {
 //   import("./components/About").then((module) =>
 //     setAboutPage(() => module.default)
 //   );
 // };

 // const loadContactPage = () => {
 //   import("./components/Contact").then((module) =>
 //     setContactPage(() => module.default)
 //   );
 // };

 return (
   <Router>
     <div>
       <nav>
         <ul>
           <li>
             <Link to="/">Home</Link>
           </li>
           <li>
             <Link to="/about">About</Link>
           </li>
           <li>
             <Link to="/contact">Contact</Link>
           </li>
         </ul>
       </nav>{" "}
       <Suspense fallback={<div>Loading...</div>}>
         <Routes>
           <Route path="/" element={<HomePage />} />
            <Route path="/about" element={<AboutPage />} />
            <Route path="/contact" element={<ContactPage />} />
         </Routes>
       </Suspense>
     </div>
   </Router>
 );
};

export default App;
```

React.lazy()

React.lazy() allows you to dynamically import components, which means the component will only be loaded when it is needed. This helps in reducing the initial load time of the application by splitting the code into smaller chunks.

React.Suspense
It is a component that can wrap lazy-loaded components and handle the loading state. It takes a fallback prop, which is a React element that will be displayed while the component is being loaded.

The fallback prop is used to display a loading indicator while the lazy-loaded component is being fetched. This provides a better user experience as the user is informed that something is happening in the background.

React.memo

In React, functional components re-render whenever their parent component re-renders. If the props haven't changed, this can be inefficient.

React.memo helps improve performance by memoizing the component, thus skipping re-renders when the props remain the same.

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

// A simple Counter component
const Counter = ({ count }) => {
 console.log('Counter component re-rendered');
 return <div>Counter: {count}</div>;
};

const Memo = () => {
 const [count, setCount] = useState(0);
 const [otherState, setOtherState] = useState(false);

 const incrementCount = () => {
   setCount(count + 1);
 };

 const toggleOtherState = () => {
   setOtherState(!otherState);
 };

 return (
   <div>
     <h1>React.memo Example</h1>
     <Counter count={count} />
     <button onClick={incrementCount}>Increment Counter</button>
     <button onClick={toggleOtherState}>Toggle Other State</button>
   </div>
 );
};

export default Memo;
```

Now the point to notice is that the Counter component re-renders when either of the state changes

So now we use React.memo to memoize the Counter component

```
const Counter = React.memo(({ count }) => {
 console.log('Counter component re-rendered');
 return <div>Counter: {count}</div>;
});
```

Counter Component: The Counter component is memoized using React.memo. This means it will only re-render if its props change. The count prop is passed to it, and it displays the current counter value.

When you click the “Toggle Other State” button, the otherState in the App component updates, but the Counter component does not re-render. This is because its count prop hasn’t changed, thanks to React.memo.

Should we use React.memo for every component ?

there are scenarios where it might not be appropriate or beneficial to use React.memo. Here are some reasons why you might choose not to use it:

1.Simple Components: If a component is simple and doesn’t involve complex rendering logic, the overhead of memoization might outweigh the benefits. Memoization itself incurs a performance cost due to the shallow comparison of props.

2.Dynamic Content: If a component frequently receives new props that change on every render, memoization won't be effective. The component will still re-render due to the constant prop changes.

3.Complex Props: If a component receives complex props (e.g., objects, arrays, functions), React.memo uses shallow comparison by default. This means it only checks if the references to these objects have changed, not their content. If the props are deeply nested or change frequently, you might need to implement a custom comparison function, which can add complexity.

4.Early Optimization: Premature optimization can lead to unnecessary complexity. It's often better to profile and identify performance bottlenecks before applying optimizations like memoization.

When to use

1. Pure Components:
   a. Use React.memo for functional components that render the same output given the same props. These components are often referred to as pure components.
   b. Example: A component that only displays props without any side effects.
2. Performance Optimization:
   a. When you notice that a component is re-rendering frequently without any changes to its props, and this is impacting performance.

b. Example: A large list component that re-renders whenever the parent component re-renders, even if the list data hasn't changed.

3. Expensive Rendering:
    a. For components that perform expensive calculations or renderings. Memoizing these components can prevent unnecessary calculations and improve performance.
    b. Example: A chart component that involves complex calculations.

title: useMemo Hook

We will start with a component that performs a costly computation every time it renders, and then we will optimize it using useMemo.

Initial Scenario without useMemo

Here, we have a simple React component that calculates the sum of a large array of numbers whenever it renders. This can be a costly computation and can lead to performance issues if the component re-renders frequently

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

const generateLargeArray = () => {
   console.time('generateLargeArray');
 const largeArray = [];
 for (let i = 0; i < 1000000; i++) {
   largeArray.push(i);
 }
   console.timeEnd('generateLargeArray');
 return largeArray;
};

const sumArray = (arr) => {
 console.log('Calculating sum...');
 return arr.reduce((acc, curr) => acc + curr, 0);
};

const LargeArraySum = () => {
 const [count, setCount] = useState(0);
 const largeArray = generateLargeArray();
 const sum = sumArray(largeArray);

 return (
   <div>
     <h1>Sum: {sum}</h1>
     <button onClick={() => setCount(count + 1)}>Increment</button>
     <p>Count: {count}</p>
   </div>
 );
};

export default LargeArraySum;
```

generateLargeArray: Generates a large array with 1,000,000 elements.
sumArray: Calculates the sum of all elements in the array and logs when it's being called.

LargeArraySum: A component that displays the sum of the array and has a button to increment a counter.

Performance Issue:

Every time the component re-renders (e.g., when the button is clicked), the sumArray function is called again, recalculating the sum of the array even though the array has not changed. This is inefficient and can lead to poor performance.

Optimized Scenario with useMemo

We can use the useMemo hook to memoize the result of the sumArray function so that it's only recalculated when the array changes.

```
const LargeArraySum = () => {
 const [count, setCount] = useState(0);
 const largeArray = useMemo(() => generateLargeArray(), []);

 const sum = useMemo(() => sumArray(largeArray), [largeArray]);
```

Explanation:
useMemo(() => generateLargeArray(), []): Memoizes the large array, so generateLargeArray is only called once when the component mounts.

useMemo(() => sumArray(largeArray), [largeArray]): Memoizes the result of sumArray, so it's only recalculated when largeArray changes. Since largeArray doesn't change in this example, sumArray is only called once.

Performance Improvement:

By using useMemo, the costly sumArray function is not re-executed on every render unless the dependency (largeArray) changes. This prevents unnecessary recalculations and significantly improves performance.

Some of the scenarios where useMemo is useful

1. Complex calculation
2. Filtering or Sorting Large Lists
    a. When you need to filter or sort large lists, useMemo can help optimize performance by ensuring the operation is only performed when necessary.

    ```
    const filteredItems = useMemo(() => {
      console.log('Filtering items...');
      return items.filter(item => item.includes(filter));
     }, [items, filter]);
    ```
3. Complex Derived State
    a. When you have state that is derived from other state values and the computation is non-trivial, useMemo ensures that this derived state is only recomputed when necessary.
    b. We did the derived state case in Watchlist

title: useCallback Hook

Scenario without useCallback

Let's consider a simple React component that renders a list of items. There's a button for each item that, when clicked, removes the item from the list.

Without useCallback

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

const ItemList = () => {
 const [items, setItems] = useState(['Item 1', 'Item 2', 'Item 3']);

 const removeItem = (itemToRemove) => {
   setItems((prevItems) => prevItems.filter((item) => item !== itemToRemove));
 };

 return (
   <div>
     {items.map((item) => (
       <div key={item}>
         {item}
         <button onClick={() => removeItem(item)}>Remove</button>
       </div>
     ))}
   </div>
 );
};

export default ItemList;
```

Explanation

In this example:

The removeItem function is recreated every time the ItemList component re-renders.

Since the removeItem function is passed as a prop to each button's onClick event, each button receives a new function reference on each render.

This can lead to performance issues, especially if the list of items is large, because the component and its children are unnecessarily re-rendered.

Optimizing with useCallback

To optimize this scenario, we can use the useCallback hook to memoize the removeItem function, ensuring that it is only recreated when the dependencies change.

```
const removeItem = useCallback((itemToRemove) => {
  setItems((prevItems) => prevItems.filter((item) => item !== itemToRemove));
}, []);
```

useCallback: The useCallback hook returns a memoized version of the callback function that only changes if one of the dependencies has changed. In this case, we have no dependencies, so removeItem will be memoized and will not change across renders.

Optimization: Since removeItem is now memoized, it maintains the same reference between renders unless the state changes. This prevents unnecessary re-renders of the child components, thus improving performance.