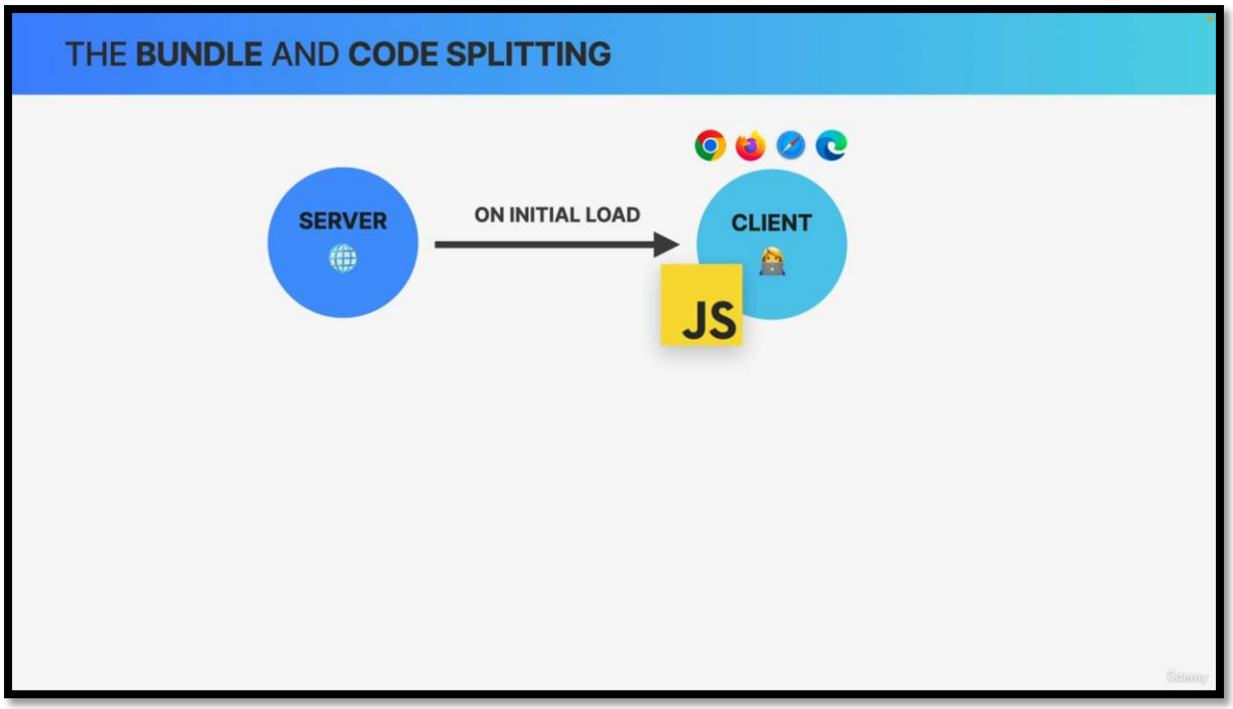
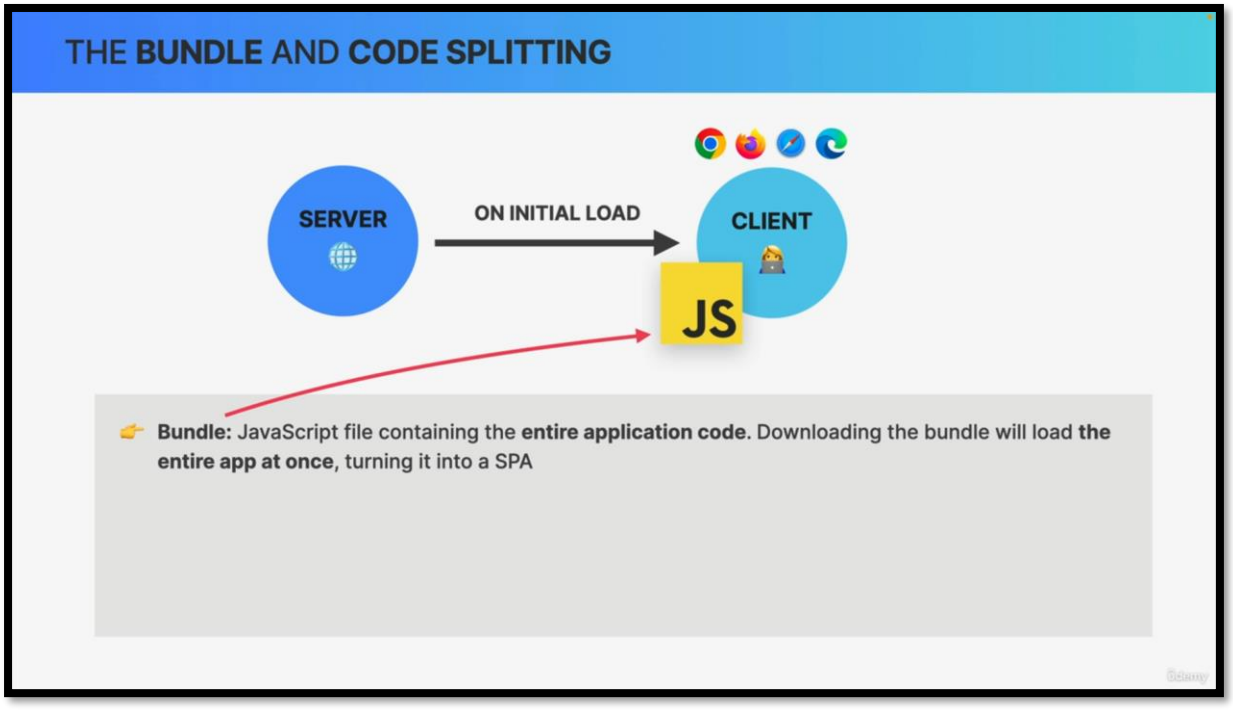
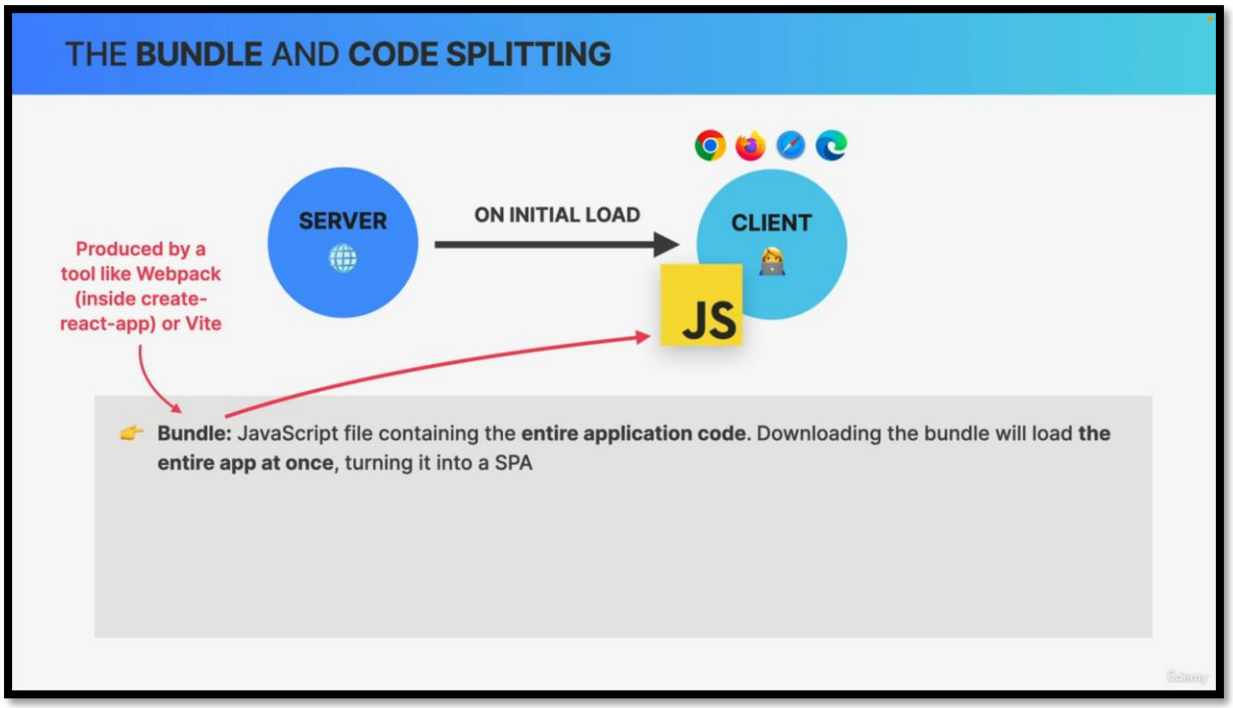



Whenever some user navigates to our application, they're basically visiting a website that is hosted on some server. So that's just how every single website and web application work.

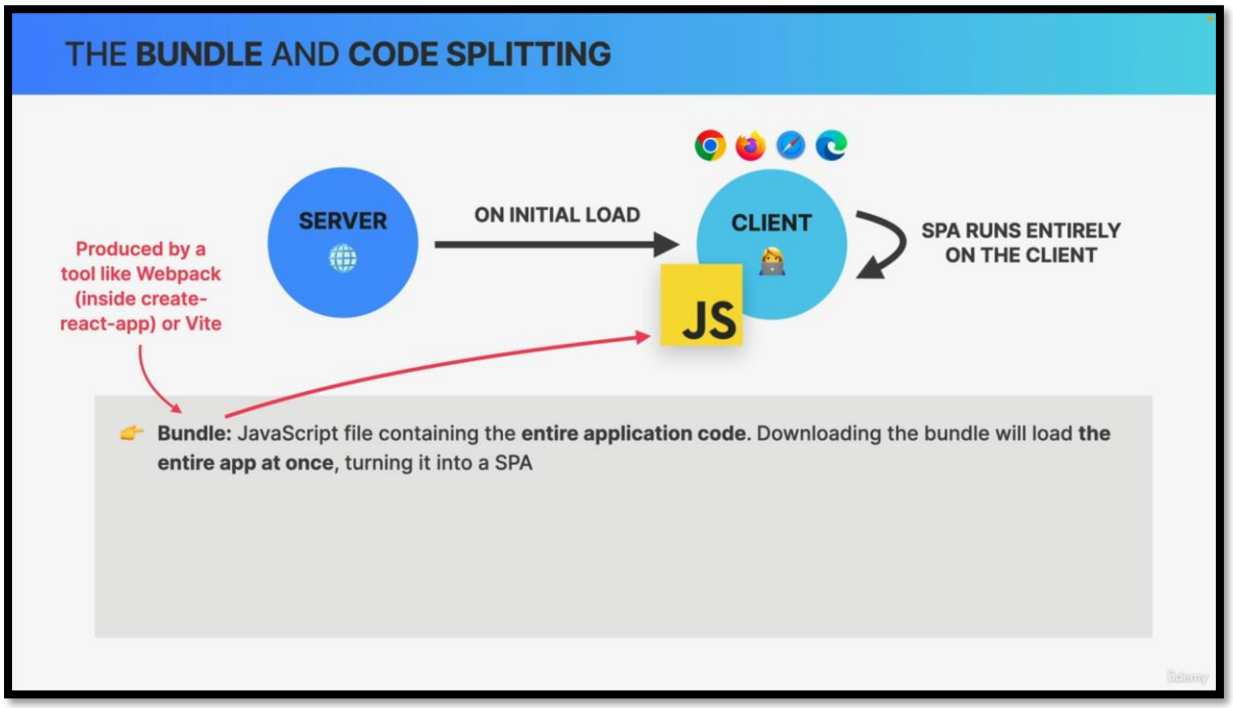
Now, once the user actually navigates to the app, the server will send back a huge JavaScript file to the client that is requesting it and this file is the bundle.



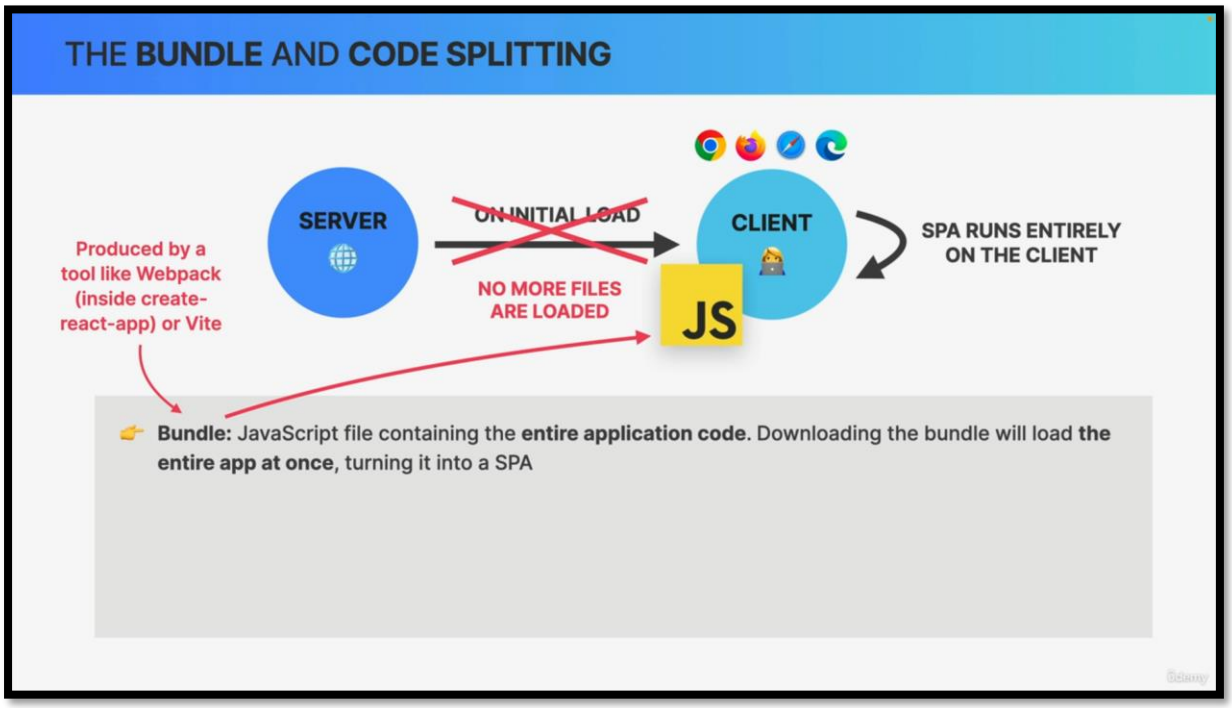
So, the bundle is simply a JavaScript file that contains the entire code of the application.



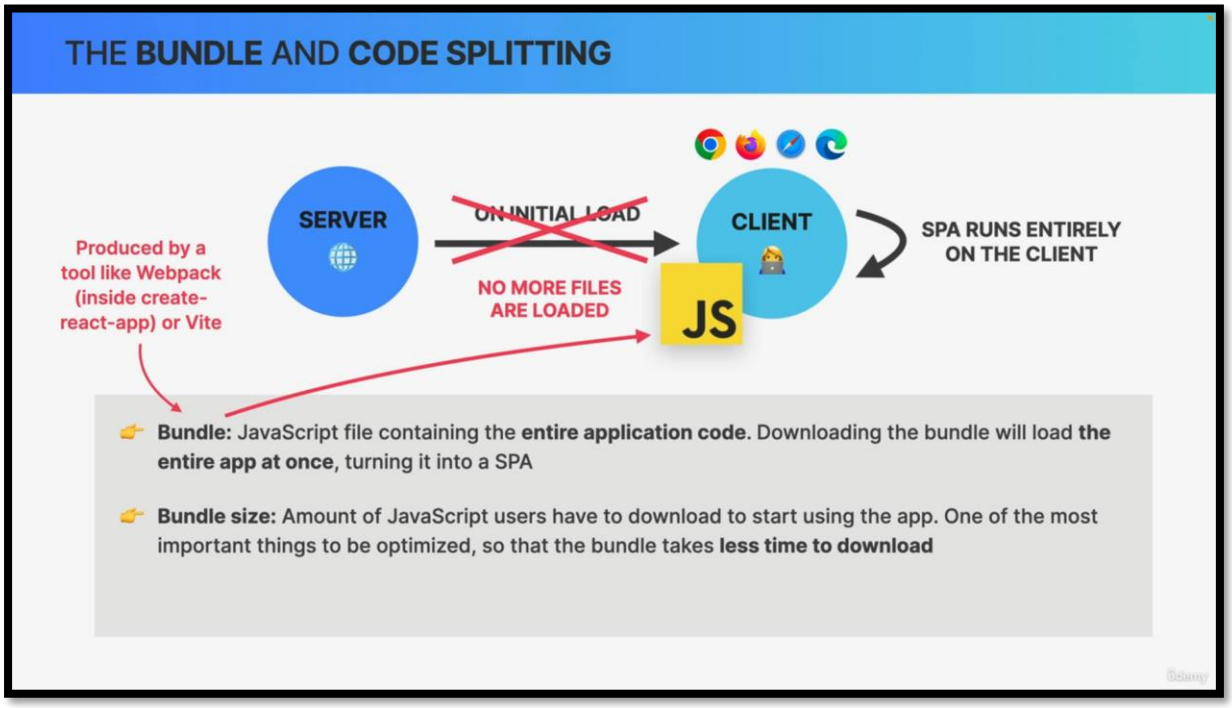

It is called a bundle because a tool like Vite or Webpack has bundled all our development files into one huge bundle which, again, contains all the application code.




This means that once the code has been downloaded, the entire React application is loaded at once, which essentially turns it into a single-page application that is running entirely on the client.




So, whenever the URL changes in the app, the client just renders a new React component but without loading any new files from the server because all the JavaScript code is already there in the bundle.

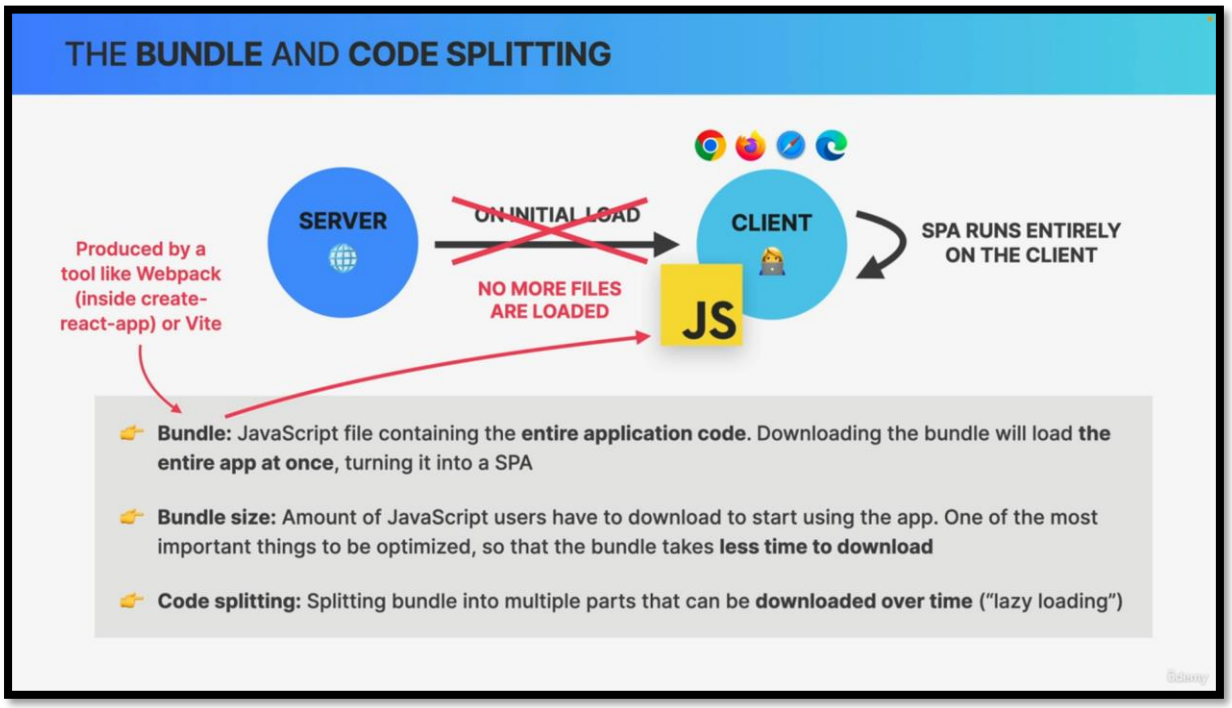



The bundle size is of course the amount of JavaScript code that every single user needs to download in order to start using the application.

So, if the bundle has, for example, 500 kilobytes, then all those bytes need to be downloaded before the app even starts working.

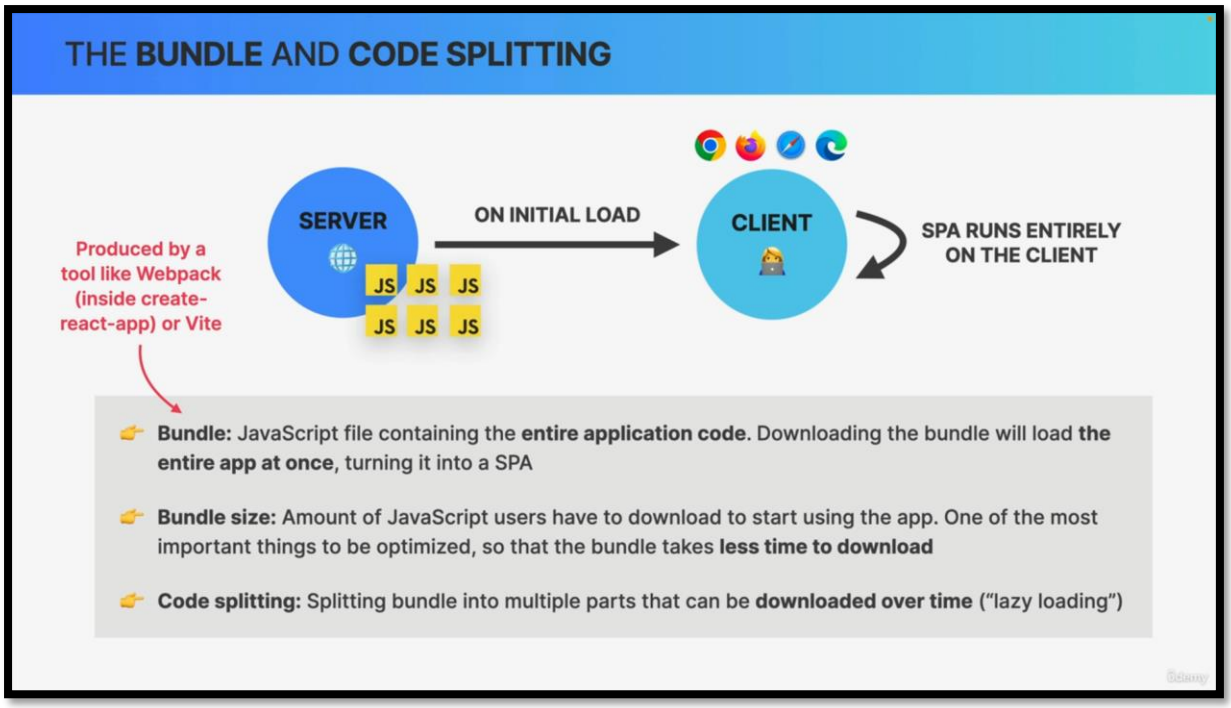
The bigger the bundle, the longer it's gonna take to download which can become a huge problem.

Therefore, bundle size is probably the most important thing that we need to optimize.




Thankfully for us and for our users, it's not very hard to do.

So, we can just use a technique called code splitting and code splitting basically takes the bundle and, as the name says, splits it into multiple parts.



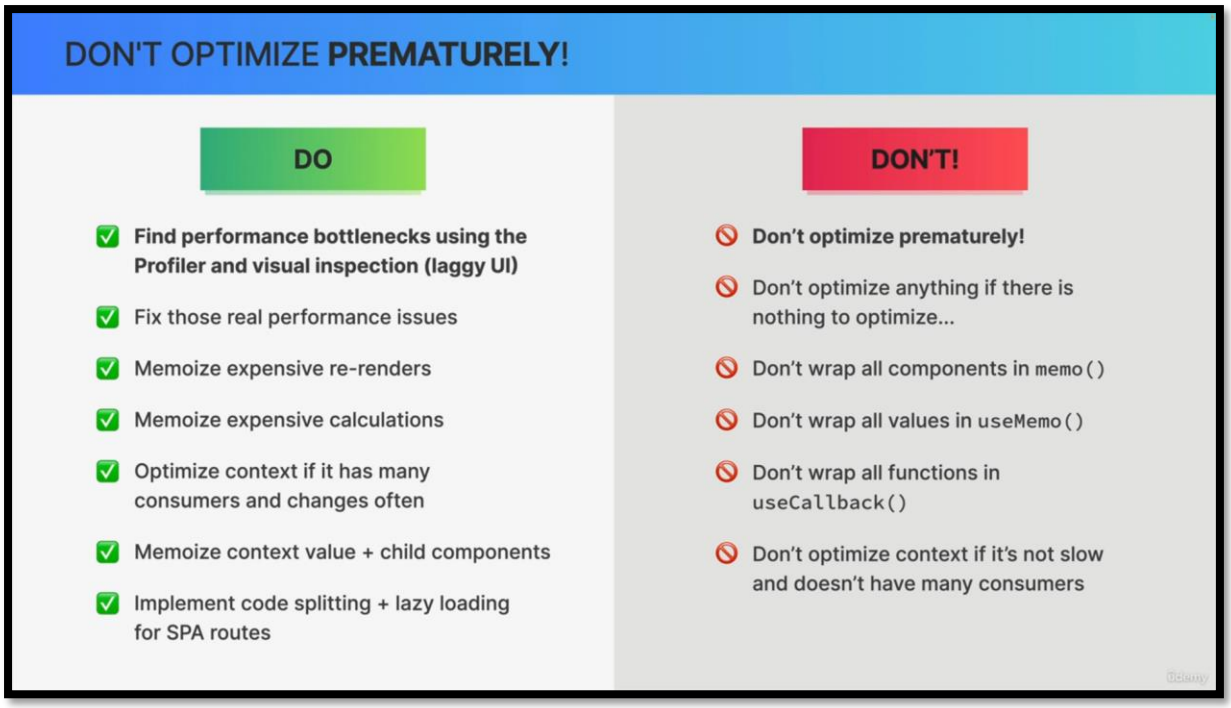

So instead of just having one huge JavaScript file, we will have multiple smaller files which can then be downloaded over time as they become necessary for the application.

This process of loading code sequentially is called lazy loading.

This really is one of the biggest performance gains that you can achieve for your users.

Now, there are many ways in which we can split the bundle, so in which we can lazily load our components.

1. The most common one is to split the bundle at the route level or, in other words, at the page level.




### Memoize Context Values

If your context provider is providing a complex object or array, memoize it using useMemo in your provider component. This can prevent unnecessary re-renders of consumers when other parts of the context change.

### Split Contexts

If your context provider is providing multiple values, consider splitting them into separate contexts. This allows consumers to subscribe only to the parts of the context they need, reducing unnecessary re-renders.

### Avoid Deep Nesting

Avoid deeply nesting context providers and consumers. Deeply nested consumers can trigger re-renders when a higher-level context changes. Try to minimize the nesting of context providers to reduce the scope of re-renders.

### Use useMemo with Context Values

If your context values are objects or arrays, use the useMemo hook to memoize them. This can prevent unnecessary re-renders of consumers that depend on those values.

### Optimize Context Consumers

Ensure that your context consumers are optimized. Use memo to prevent re-renders of consumer components when their props haven't changed.

### Provider Value Composition

Avoid creating complex objects as the value prop for the context provider. Compose the value prop using useMemo or other memoization techniques to ensure that the provider's value prop remains stable.

### Limit Context Updates

Be mindful of how frequently you update context values. If a context value changes too frequently, it can lead to many unnecessary re-renders. Try to batch context updates when possible.

### Avoid Excessive Dependencies

Keep the dependency arrays of hooks like useEffect and useMemo minimal. Avoid adding context values to the dependency array of a hook if the hook doesn't need to react to changes in that specific context value.

### Use Function Memoization

If you have functions in your context, consider memoizing them using useCallback. This ensures that function references remain stable between re-renders.

By following these strategies, you can optimize context re-renders and improve the performance of your React application, especially in scenarios where context is used to share state or functions across components.