Using React.lazy and Suspense

In this lesson, we'll discuss how using React.lazy and Suspense make things easy.

WE'LL COVER THE FOLLOWING

Easy Dynamic Imports

`React.lazy` and `Suspense` make using dynamic imports in a React application so easy.

For example, consider the demo code for the Benny application below:

```
import React from 'react'
import Benny from './Benny'
import Scene from './Scene'
import GameInstructions from './GameInstructions'

class Game extends Component {
  state = {
    startGame: false
  }
  render () {
    return !this.state.startGame ?
        <GameInstructions /> :
                <Scene />
  }
}
export default Game;
```

Based on the state property `startGame`, either the `GameInstructions` or `Scene` component is rendered when the user clicks the “Start Game” button.

`GameInstructions` represents the home page of the game and `Scene` represents the entire scene of the game itself.

Importing `Scene` Using `React.lazy`

In this implementation, `GameInstructions` and `Scene` will be bundled together in the same Javascript resource.

Even when the user hasn't shown intent to start playing the game, we would have loaded the complex `Scene` component which contains all the logic for the game scene and sent it to the user.

So, what do we do?

Let's defer the loading of the `Scene` component.

Here's how easy `React.lazy` makes that.

```
// before
import Scene from './Scene'
// now
const Scene = React.lazy(() => import('./Scene'))
```

`React.lazy` takes a function that must call a dynamic import. In this case, the dynamic import call is `import('./Scene')`.

Note that `React.lazy` expects the dynamically loaded module to have a `default` export containing a React component.

With the `Scene` component now dynamically loaded, when the application is bundled for production, a separate module or JavaScript file will be created for the `Scene` dynamic import.

When the app loads, this JavaScript file won't be sent to the user. However, if they click the "Start Game" button and show intent to load the `Scene` component, a network request would be made to fetch the resource from the remote server.

Fetching from the server introduces some latency. To handle this, wrap the `Scene` component in a `Suspense` component to show a fallback for when the resource is being fetched.

Here's what I mean:

```
import { Suspense } from 'react'
```

```
const Scene = React.lazy(() => import('./Scene'))

class Game extends Component {

  state = {
    startGame: false
  }
  render () {
    return !this.state.startGame ?
        <GameInstructions /> :
                // look here
                <Suspense fallback="<div>loading ...</div>">
                  <Scene />
                </Suspense>
  }
}
export default Game;
```

When the network request is initiated to fetch the `Scene` resource, we'll show a "loading..." fallback courtesy of the `Suspense` component.

`Suspense` takes a `fallback` prop which can be a markup as shown here, or a full-blown React component.

With `React.lazy` and `Suspense` you can suspend the fetching of a component until much later and show a fallback for when the resource is being fetched.

How convenient!

Also, you can place the `Suspense` component anywhere above the lazy-loaded component, in this case, the `Scene` component.

If you also had multiple lazy-loaded components, you could wrap them in single or multiple `Suspense` components depending on your specific use case.

In the next lesson, we'll learn how to handle errors when things don't go according to plan.