Code-Splitting

Bundling

Most React apps will have their files “bundled” using tools like Webpack, Rollup or Browserify. Bundling is the process of following imported files and merging them into a single file: a “bundle”. This bundle can then be included on a webpage to load an entire app at once.

Example

App:

```
// app.js
import { add } from './math.js';

console.log(add(16, 26)); // 42
```

```
// math.js
export function add(a, b) {
  return a + b;
}
```

Bundle:

```
function add(a, b) {
  return a + b;
}

console.log(add(16, 26)); // 42
```

Note:

> Your bundles will end up looking a lot different than this.

If you're using Create React App, Next.js, Gatsby, or a similar tool, you will have a Webpack setup out of the box to bundle your app.

If you aren't, you'll need to set up bundling yourself. For example, see the Installation and Getting Started guides on the Webpack docs.

Code Splitting

Bundling is great, but as your app grows, your bundle will grow too. Especially if you are including large third-party libraries. You need to keep an eye on the code you are including in your bundle so that you don't accidentally make it so large that your app takes a long time to load.

To avoid winding up with a large bundle, it's good to get ahead of the problem and start "splitting" your bundle. Code-Splitting is a feature supported by bundlers like Webpack, Rollup and Browserify (via factor-bundle) which can create multiple bundles that can be dynamically loaded at runtime.

Code-splitting your app can help you "lazy-load" just the things that are currently needed by the user, which can dramatically improve the performance of your app. While you haven't reduced the overall amount of code in your app, you've avoided loading code that the user may never need, and reduced the amount of code needed during the initial load.

`import()`

The best way to introduce code-splitting into your app is through the dynamic `import()` syntax.

Before:

```
import { add } from './math';
```

```
console.log(add(16, 26));
```

After:

```
import("./math").then(math => {
  console.log(math.add(16, 26));
});
```

When Webpack comes across this syntax, it automatically starts code-splitting your app. If you're using Create React App, this is already configured for you and you can start using it immediately. It's also supported out of the box in Next.js.

If you're setting up Webpack yourself, you'll probably want to read Webpack's guide on code splitting. Your Webpack config should look vaguely like this.

When using Babel, you'll need to make sure that Babel can parse the dynamic import syntax but is not transforming it. For that you will need @babel/plugin-syntax-dynamic-import.

`React.lazy`

> **Note:**
> `React.lazy` and Suspense are not yet available for server-side rendering. If you want to do code-splitting in a server rendered app, we recommend Loadable Components. It has a nice guide for bundle splitting with server-side rendering.

The `React.lazy` function lets you render a dynamic import as a regular component.

Before:

```
import OtherComponent from './OtherComponent';
```

After:

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

This will automatically load the bundle containing the `OtherComponent` when this component is first rendered.

`React.lazy` takes a function that must call a dynamic `import()`. This must return a `Promise` which resolves to a module with a `default` export containing a React component.

The lazy component should then be rendered inside a `Suspense` component, which allows us to show some fallback content (such as a loading indicator) while we're waiting for the lazy component to load.

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

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

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

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

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

function MyComponent() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <section>
          <OtherComponent />
          <AnotherComponent />
        </section>
      </Suspense>
    </div>
```

```
    );
  }
```

Error boundaries

If the other module fails to load (for example, due to network failure), it will trigger an error. You can handle these errors to show a nice user experience and manage recovery with Error Boundaries. Once you've created your Error Boundary, you can use it anywhere above your lazy components to display an error state when there's a network error.

```
import React, { Suspense } from 'react';
import MyErrorBoundary from './MyErrorBoundary';

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

const MyComponent = () => (
  <div>
    <MyErrorBoundary>
      <Suspense fallback={<div>Loading...</div>}>
        <section>
          <OtherComponent />
          <AnotherComponent />
        </section>
      </Suspense>
    </MyErrorBoundary>
  </div>
);
```

Route-based code splitting

Deciding where in your app to introduce code splitting can be a bit tricky. You want to make sure you choose places that will split bundles evenly, but won't disrupt the user experience.

A good place to start is with routes. Most people on the web are used to page transitions taking some amount of time to load. You also tend to be re-rendering the entire page at once so your users are unlikely to be interacting with other elements on the page at the same time

so your users are unlikely to be interacting with other elements on the page at the same time.

Here's an example of how to setup route-based code splitting into your app using libraries like React Router with `React.lazy`.

```
import React, { Suspense, lazy } from 'react';
import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';

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

const App = () => (
  <Router>
    <Suspense fallback={<div>Loading...</div>}>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
      </Routes>
    </Suspense>
  </Router>
);
```

Named Exports

`React.lazy` currently only supports default exports. If the module you want to import uses named exports, you can create an intermediate module that reexports it as the default. This ensures that tree shaking keeps working and that you don't pull in unused components.

```
// ManyComponents.js
export const MyComponent = /* ... */;
export const MyUnusedComponent = /* ... */;
```

```
// MyComponent.js
export { MyComponent as default } from "./ManyComponents.js";
```

```
// MyApp.js
import React, { lazy } from 'react';
const MyComponent = lazy(() => import("./MyComponent.js"));
```