What are dynamic imports in JavaScript?

Written by Seth Corker on Benevolent Bytes

What is it?

ECMAScript modules are becoming the standard across the web, and one cool benefit of that is built-in syntax for dynamic imports. Your usual import might look something like:

```
import { format, formatDistance, formatRelative, subDays } from 'date-fns'
```

This means you've got access to the functions you've imported all throughout your file, but it also means that everything is loaded all at once. If you're using a bundler like Webpack, this might result in everything being smooshed together into the same file. There's another way, that's where dynamic imports come into play, they look like this:

```
import('date-fns').then(({ format }) => {/* ... */});
```

An import with a promise? Yep, this allows you to well... import a module when you need it, so maybe the end-user on your website doesn't need to load it at all!

A Juice and Coffee example

Have a look at the CodeSandbox and/or follow along with me.

<iframe src="https://codesandbox.io/p/github/Darth-Knoppix/experiment-js-dynamic-import/main?embed=1&file=%2Fmain
style={{width: '100%', height: '500px'}} title="Darth-Knoppix/experiment-js-dynamic-import/main" allow="accelerometer; ambient-light-sensor; camera; encrypted-media; geolocation; gyroscope; hid; microphone; midi; payment; usb; vr; xr-spatial-tracking" sandbox="allow-forms allow-modals allow-popups allow-presentation allow-same-origin allow-scripts">

Setting the scene Let's have a look at a scenario. We're creating an ordering system for a small food court, it's minimal currently, there's only a juice stand and a coffee stall. We're tasked with offering customers an easy way to buy the specialty from each shop. With that in mind, let's have a look at some code!

Here are our two "shops" they are modules that export an `orderSpecialty` function.

```
// coffee.js
export function orderSpecialty() {
  return "1x flat white and a croissant";
}
```

```
// juice.js
export function orderSpecialty() {
  return "1x medium sugar cane juice with ginger and mint";
}
```

So we've got our two shops, what do we do now? Let's show it on a web page. We'll add a couple of buttons we can reference shortly and a `data-shop` attribute so we can see which one the customer chose.

```
<div id="app">
  <h1>Choose your drink</h1>
  <button data-shop="juice">Order specialty juice</button>
  <button data-shop="coffee">Order specialty coffee</button>
</div>
<script type="module" src="/main.js"></script>
```

Finally, let's take a look at how we might do our dynamic import for our specialty order from each shop.

```
// main.js
async function chooseShop(shopType) {
  if (shopType === "juice") {
    return await import("./juice.js");
  } else if (shopType === "coffee") {
    return await import("./coffee.js");
  }
}
```

```
// Make an order based on the shop and alert the order
export async function order(e) {
  const shop = e.target.dataset.shop;
  const result = (await chooseShop(shop)).orderSpecialty();
  alert(result);
}

// Setup the event listeners on the buttons
document
  .querySelectorAll("button")
  .forEach((el) => el.addEventListener("click", order));
```

Our function `chooseShop` is the key focus here, we accept a `shopType` and depending on which one we get, we'll do a different import. Either importing `juice.js` or `coffee.js`. In this example, we get the shop, wait for the module to load and then show the order to the user. It's a little more work than just declaring our imports at the top, so why would we ever use this? ## Where is it useful? In our food court example, it might not be super useful with only two shops, but what about 100s or 1000s! If we have a customer wanting to order from 1 of 1000 shops, we'd have to import all of those shops so we can order the specialty item. The more complex features the shops would like to offer, like menus, contact details etc. would make each module larger and take longer to load.

Dynamic imports allow us to only load what we need. We can choose parts of our codebase we want to segment off and load as a user action. In our case, we only load the shop module that the customer is ordering from, but you could pick other ways to do it. Some common ways of splitting up loading into modules is based on routing, user intent and user interaction.

What's the alternative?

There are a few options. 1. We could just import everything all at once with the trade off that it's going to be larger and the end-user might not need all the other imports. 2. We could avoid JavaScript for our shops entirely. Maybe we create another API that we can fetch from so that we only load each shop over the network similarly we do currently. The trade-off here is that we're more limited in how dynamic we can be, and it requires some rearchitecting of our solution. 3. We could try splitting imports by browser path. If a user visits a particular page, let's say our coffee shop, we could just load what we need there. The tradeoff is that we need to rearchitect our app and ensure that each page only loads the files it needs. We may also need to redirect a user on the initial interaction.

There's even more options depending on what bundler you're using, how you want to split things up and if your app is large enough for it to make sense in the first place. Dynamic imports are a nice primitive that we can use to choose how we would like to deliver our app to users. This can mean downloading less JavaScript on the initial load, making our app quicker and only requesting what we need when we need it, which is especially useful on mobile connections. It's another tool to add to your JavaScript tool belt.

Resources

- Dynamic import on MDN
- experiment-js-dynamic-import Repository on GitHub