Dynamic import()

By Mathias Bynens
V8 JavaScript whisperer

Note: Dynamic `import()` is available in Chrome 63 and Safari Technology Preview 24.

Dynamic `import()` introduces a new function-like form of `import` that unlocks new capabilities compared to static `import`. This article compares the two and gives an overview of what's new.

Static `import` (recap)

Chrome 61 shipped with support for the ES2015 `import` statement within modules.

Consider the following module, located at `./utils.mjs`:

```
// Default export
export default () => {
  console.log('Hi from the default export!');
};

// Named export `doStuff`
export const doStuff = () => {
  console.log('Doing stuff…');
};
```

Here's how to statically import and use the `./utils.mjs` module:

```
<script type="module">
  import * as module from './utils.mjs';
  module.default();
  // → logs 'Hi from the default export!'
  module.doStuff();
  // → logs 'Doing stuff…'
</script>
```

Note: The previous example uses the `.mjs` extension to signal that it's a module rather than a regular script. On the web, file extensions don't really matter, as long as the files are served with the correct MIME type (e.g. `text/javascript` for JavaScript files) in the `Content-Type` HTTP header. The `.mjs` extension is especially useful on other platforms such as Node.js, where there's no concept of MIME types or other hooks such as `type="module"` to determine whether something is a module or a regular script. We're using the same extension here for consistency across platforms and to clearly make the distinction between modules and regular scripts.

This syntactic form for importing modules is a **static** declaration: it only accepts a string literal as the module specifier, and introduces bindings into the local scope via a pre-runtime "linking" process. The static `import` syntax can only be used at the top-level of the file. Static `import` enables important use cases such as static analysis, bundling tools, and tree-shaking.

In some cases, it's useful to:

- import a module on-demand (or conditionally)
- compute the module specifier at runtime
- import a module from within a regular script (as opposed to a module)

None of those are possible with static `import`.

Dynamic `import()` 🔥

Dynamic `import()` introduces a new function-like form of `import` that caters to those use cases. `import(moduleSpecifier)` returns a promise for the module namespace object of the requested module, which is created after fetching, instantiating, and evaluating all of the module's dependencies, as well as the module itself.

Here's how to dynamically import and use the `./utils.mjs` module:

```
<script type="module">
  const moduleSpecifier = './utils.mjs';
  import(moduleSpecifier)
    .then((module) => {
      module.default();
      // → logs 'Hi from the default export!'
      module.doStuff();
      // → logs 'Doing stuff…'
    });
</script>
```

Since **`import()`** returns a promise, it's possible to use **`async`**/**`await`** instead of the **`then`**-based callback style:

```
<script type="module">
  (async () => {
    const moduleSpecifier = './utils.mjs';
    const module = await import(moduleSpecifier)
    module.default();
    // → logs 'Hi from the default export!'
    module.doStuff();
    // → logs 'Doing stuff…'
  })();
</script>
```

Note: Although **`import()`** *looks* like a function call, it is specified as *syntax* that just happens to use parentheses (similar to **`super()`**). That means that **`import`** doesn't inherit from **`Function.prototype`** so you cannot **`call`** or **`apply`** it, and things like **`const importAlias = import`** don't work — heck, **`import`** is not even an object! This doesn't really matter in practice, though.

Here's an example of how dynamic **`import()`** enables lazy-loading modules upon navigation in a small single-page application:

```
<!DOCTYPE html>
<meta charset="utf-8">
<title>My library</title>
<nav>
  <a href="books.html" data-entry-module="books">Books</a>
  <a href="movies.html" data-entry-module="movies">Movies</a>
  <a href="video-games.html" data-entry-module="video-games">Video Games</a>
</nav>
<main>This is a placeholder for the content that will be loaded on-demand.</main>
<script>
  const main = document.querySelector('main');
  const links = document.querySelectorAll('nav > a');
  for (const link of links) {
    link.addEventListener('click', async (event) => {
      event.preventDefault();
      try {
        const module = await import(`/${link.dataset.entryModule}.mjs`);
        // The module exports a function named `loadPageInto`.
        module.loadPageInto(main);
      } catch (error) {
        main.textContent = error.message;
      }
```

```
    });
  }
</script>
```

The lazy-loading capabilities enabled by dynamic **`import()`** can be quite powerful when applied correctly. For demonstration purposes, Addy modified an example Hacker News PWA that statically imported all its dependencies, including comments, on first load. The updated version uses dynamic **`import()`** to lazily load the comments, avoiding the load, parse, and compile cost until the user really needs them.

Recommendations

Static **`import`** and dynamic **`import()`** are both useful. Each have their own, very distinct, use cases. Use static **`import`**s for initial paint dependencies, especially for above-the-fold content. In other cases, consider loading dependencies on-demand with dynamic **`import()`**.

Last updated July 2, 2018.