# Import maps and why they matter

Import Maps are a way to map the names of modules to the actual files that contain them.

Using a script of type `importmap` we can specify the location of script we want to use. For example we can use this script to set up the mapping for the Lodash library.

```
<script type="importmap">
{
  "imports": {
    "lodash": "/node_modules/lodash-es/lodash.js"
  }
}
</script>
```

We can then use the mapping in our scripts. We no longer need to provide the full path to the library in our import statements

```
import lodash from 'lodash';
```

Another reason why import maps are important is because they allow more flexibility in the way that we import modules.

Libraries like Lodash have many modules that you can import individually or you can import the full library if you sho choose.

This import map will allow us to import the full library and to import modules inside the library.

```
<script type="importmap">
  {
    "imports": {
      "lodash": "/node_modules/lodash/lodash.js",
      "lodash/": "/node_modules/lodash/"
    }
  }
</script>
```

By specifying a separate module specifier name as lodash/ and mirroring the same thing in the address /node_modules/lodash/, you are allowing for specific modules in the package to be imported with ease which will look something like this:

```
// You can directly import lodash
import _lodash from "lodash";

"lodash"// or import a specific moodule
import _shuffle from "lodash/shuffle.js";
```

## Polyfilling import maps

The es-module-shims package can polyfill import maps (and other module-related specifications) for older browsers.

It requires some changes to our code. First, we need to load es-module-shims using a script tag with the `async` attribute so it won't block rendering.

```
<script
  async
  src="https://ga.jspm.io/npm:es-module-shims@1.5.8/dist/es-module-shims.j
</script>
```

The value for the type attribute for the import map changes to `importmap-shim` so it can use the shim script we loaded.

```
<script type="importmap-shim">
{
  "imports": {
    "lodash": "/node_modules/lodash-es/lodash.js"
  }
}
</script>
```

Likewise, the type attribute for module scripts changes to `module-shim` so it can use the import map we defined.

```
<script type="module-shim">
  import { partition } from "lodash";
  // ...
  partition(users, 'active');
'active'</script>
```

The order of the scripts does matter. In any other order, the scripts will not work and you will get an error.

## ESM Module availablity

The second issue is that not all packages in NPM are available as ESModules.

Tools like Snowpack can help deal with this. In its simplest use Snowpack ***"re-installs your dependencies as single JS files to a new web_modules/ directory"*** to be used with import commands.

As of April of 2022, Snowpack is no longer maintained and the developers don't recommend using it for new projects. The reasoning behind the change is documented in 6 More Things I Learned Building Snowpack to 20,000 Stars (Part 2)

Until we can find a better alternative, we'll continue to work with Snowpack.

## Running Snowpack

You can either run Snopwack manually in your directory:

```
npx snowpack dev
```

or run it as an NPM script:

```
"scripts": {
  "prepare": "snowpack dev"
}
```

Snowpack will generate a `web_modules` directories that you've specified as dependencies in your package.json and an import map file that you can use to import the modules.

If you need to use packages that are not defined as dependencies in your project's package.json you can define them as part of your package.json definition

```
"snowpack": {
  "dependencies": [
    "file1",
    "file2",
    "cor"file1"    "module1",
    "module2",
    "moodule3"
  ]
}
```

It is still not an ideal solution, but it works and provides an interim solution until all modules are available as ESModules.

## If not Snowpack then what?