Import maps: What they are and why they are important

Most current browsers support ESModules. This is the way the ECMAScript standard defines to import modules. However, because it's come late to the party after UMD and CommonJS modules, they are not widely used on client-side code. One of the problems is how to reference the modules we want to import.

Import maps are a way to map the names of imported packages to the full path on the filesystem allowing developers to use bare module import.

At their simplest, import maps are a list of tuples in the form of `<package-name>: <path-to-package>`

The following example gives Lodash the identified lodash, like so:

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

Then you can do the following in your module scripts:

```
<script type="module">
import { partition } from "lodash";

partition(users, 'active');
</script>
```

Then start a server in the root of your project where your package.json and node_modules folder live. And that's `python3 -m http.server` or whichever you

prefer.

Now you can import your module scripts with the bare module syntax directly in the browser without intermediate build steps.

The devil is in the details

There are two issues that make this technique more difficult to use as it should be right now.

Browser support

The first one is browser support. Import Maps are currently only supported in Chromium browsers (Chrome, Edge, Opera among others). They are not supported in Firefox and Safari.

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 defer attribute so it won't block rendering.

```
<script
        defer
        src="es-module-shims.js">
</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 isssue 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.

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