Maximilian Fellner
Posted on 21. Juli

Dynamic import with HTTP URLs in Node.js

#node #deno

Is it possible to import code in Node.js from HTTP(S) URLs just like in the browser or in Deno? After all, Node.js has had stable support for ECMAScript modules since version 14, released in April 2020. So what happens if we just write something like

Maximilian Fellner

I am a passionate software engineer who loves science and technology.

LOCATION
Berlin

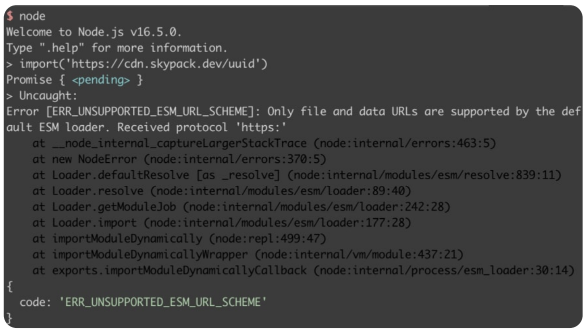
WORK
Tech Lead at N26

JOINED
8. Nov. 2019

More from Maximilian Fellner

Node.js compatibility: Using npm packages in Deno
#node #deno #npm #typescript

`import('https://cdn.skypack.dev/uuid')`?



```
$ node
Welcome to Node.js v16.5.0.
Type ".help" for more information.
> import('https://cdn.skypack.dev/uuid')
Promise { <pending> }
> Uncaught:
Error [ERR_UNSUPPORTED_ESM_URL_SCHEME]: Only file and data URLs are supported by the def
ault ESM loader. Received protocol 'https:'
    at __node_internal_captureLargerStackTrace (node:internal/errors:463:5)
    at new NodeError (node:internal/errors:370:5)
    at Loader.defaultResolve [as _resolve] (node:internal/modules/esm/resolve:839:11)
    at Loader.resolve (node:internal/modules/esm/loader:89:40)
    at Loader.getModuleJob (node:internal/modules/esm/loader:242:28)
    at Loader.import (node:internal/modules/esm/loader:177:28)
    at importModuleDynamically (node:repl:499:47)
    at importModuleDynamicallyWrapper (node:internal/vm/module:437:21)
    at exports.importModuleDynamicallyCallback (node:internal/process/esm_loader:30:14)
{
  code: 'ERR_UNSUPPORTED_ESM_URL_SCHEME'
}
```

Unfortunately it's neither possible import code from HTTP URLs statically nor dynamically because the URL scheme is not supported.

Loaders and VM

An experimental feature of Node.js are custom loaders. A loader is basically a set of "hook" functions to resolve and load source code. There is even an example of an HTTP loader.

Such a loader would be passed to Node.js as a command line argument:

```
node --experimental-loa
```

A downside to this approach is that a loader's influence is quite limited. For instance, the execution context of the downloaded code cannot be modified. The team working on loaders is still modifying their API, so this could still be subject to change.

Another Node.js API that offers more low-level control is vm. It enables the execution of raw JavaScript code within the V8 virtual machine.

In this blog post, we're going to use it to create our own dynamic import implementation!

Downloading code

Let's start with downloading the remotely hosted code. A very simple and naive solution is to just use "node-fetch" or a similar library:

```
import fetch from 'node

async function fetchCod
```

```
    const response = awai
    if (response.ok) {
      return response.tex
    } else {
      throw new Error(
        `Error fetching $
      );
  }
```

We can use this function to download any ECMAScript module from a remote server. In this example we are going to use the lodash-es module from Skypack[1], the CDN and package repository of the Snowpack build tool.

```
  const url = 'import cdn
  const source = await fe
```

Obviously important security and performance aspects have been neglected here. A more fully-featured solution would handle request headers, timeouts, and caching amongst other things.

Evaluating code

For the longest time, Node.js has provided the vm.Script class to compile and execute raw source code. It's a bit like `eval` but more sophisticated. However, this API only works with the classic CommonJS modules.

For ECMAScript modules, the new vm.Module API must be used and it is still experimental. To enable it, Node.js must be run with the `--experimental-vm-modules` flag.

To use `vm.Module` we are going to implement the 3 distinct steps creation/parsing, linking, and evaluation:

Creation/parsing

First, we need to create an execution context. This is going to be the global context in which the code will be executed. The context can be just an empty object but some code may require certain global variables, like those defined by Node.js itself.

```
import vm from 'vm';

const context = vm.crea
```

Next, we create an instance of `vm.SourceTextModule` which is a subclass of `vm.Module` specifically for raw source code strings.

```
return new vm.SourceTex
  identifier: url,
  context,
});
```

The `identifier` is the name of the module. We set it to the original HTTP URL because we are going to need it for resolving additional imports in the next step.

Linking

In order to resolve additional static `import` statements in the code, we must implement a custom `link` function. This function should return a new `vm.SourceTextModule` instance for the two arguments it receives:

- The **specifier** of the imported dependency. In ECMAScript modules this can either be an absolute or a relative URL to another file, or a "bare specifier" like `"lodash-es"`.
- The **referencing module** which is an instance of `vm.Module` and the "parent" module of the imported dependency.

In this example we are only going to deal with URL imports for now:

```
async function link(spe
  // Create a new absol
  // module's URL (spec
  // URL (referencingMo
  const url = new URL(
    specifier,
    referencingModule.i
  ).toString();
  // Download the raw s
  const source = await
  // Instantiate a new
  return new vm.SourceT
```

```
      identifier: url,
      context: referencin
    });
  }

  await mod.link(link); /
```

Evaluation

After the `link` step, the original module instance is fully initialised and any exports could already be extracted from its namespace. However, if there are any imperative statements in the code that should be executed, this additional step is necessary.

```
  await mod.evaluate(); /
```

Getting the exports

The very last step is to extract whatever the module exports from its namespace.

```
  // The following corres
  // import { random } fr
  const { random } = mod.
```

Providing global dependencies

Some modules may require certain global variables in their execution context. For instance, the uuid package depends on `crypto`, which is the Web Crypto API. Node.js provides an implementation of this API since version 15 and we can inject it into the context as a global variable.

```
import { webcrypto } fr
import vm from 'vm';

const context = vm.crea
```

By default, no additional global variables are available to the executed code. It's very important to consider the security implications of giving potentially untrusted code access to additional global variables, e.g. `process`.

Bare module specifiers

The ECMAScript module specification allows for a type of import declaration that is sometimes called "bare module specifier". Basically, it's similar to how a `require` statement of CommonJS would look like when importing a module from `node_modules`.

```
import uuid from 'uuid'
```

Because ECMAScript modules were designed for the web, it's not immediately clear how a bare module specifier should be treated. Currently there is a draft proposal by the W3C community for "import maps". So far, some browsers and other runtimes have already added support for import maps, including Deno. An import map could look like this:

```
{
  "imports": {
    "uuid": "https://
  }
}
```

Using this construct, the `link` function that is used by `SourceTextModule` to resolve additional imports could be updated to look up entries in the map:

```
const { imports } = imp

const url =
  specifier in imports
    ? imports[specifier
    : new URL(specifier
```

Importing core node modules

As we have seen, some modules may depend on certain global variables while others may use bare module specifiers. But what if a module wants to import a core node module like `fs`?

We can further enhance the `link` function to detect wether an import is for a Node.js builtin module. One possibility would be to look up the specifier in the list of builtin module names.

```
import { builtinModules

// Is the specifier, e.
if (builtinModules.incl
  // Create a vm.Module
}
```

Another option would be to use the import map and the convention that every builtin module can be imported with the `node:` URL protocol. In fact, Node.js ECMAScript modules already support `node:`, `file:` and `data:` protocols for their import statements (and we just added support for `http/s:`).

```
// An import map with a
const { imports } = {
  imports: { fs: 'node:
};

const url =
  specifier in imports
    ? new URL(imports[s
    : new URL(specifier
if (
  url.protocol === 'htt
  url.protocol === 'htt
) {
  // Download code and
} else if (url.protocol
  // Create a vm.Module
} else {
  // Other possible sch
}
```

Creating a vm.Module for a Node.js builtin

So how do we create a `vm.Module` for a Node.js builtin module? If we used another SourceTextModule with an `export` statement for, e.g. `fs`, it would lead to an endlessly recursive loop of calling the `link` function over and over again.

On the other hand, if we use a SourceTextModule with the code `export default fs`, where `fs` is a global variable on the context, the exported module would be wrapped inside an object with the `default` property.

```
// This leads to an end
new vm.SourceTextModule
// This ends up as an o
new vm.SourceTextModule
  context: { fs: await
});
```

However, we can use vm.SyntheticModule. This implementation of `vm.Module` allows us to programatically construct a module without a source code string.

```
// Actually import the
const imported = await
const exportNames = Obj
// Construct a new modu
return new vm.Synthetic
  exportNames,
  function () {
    for (const name of
      this.setExport(na
    }
  },
  {
    identifier,
    context: referencin
  }
);
```

Conclusion

The (still experimental) APIs of Node.js allow us to implement a solution for dynamically importing code from HTTP URLs "in user space". While ECMAScript modules and `vm.Module` were used in this blog post, `vm.Script` could be used to implement a similar solution for CommonJS modules.

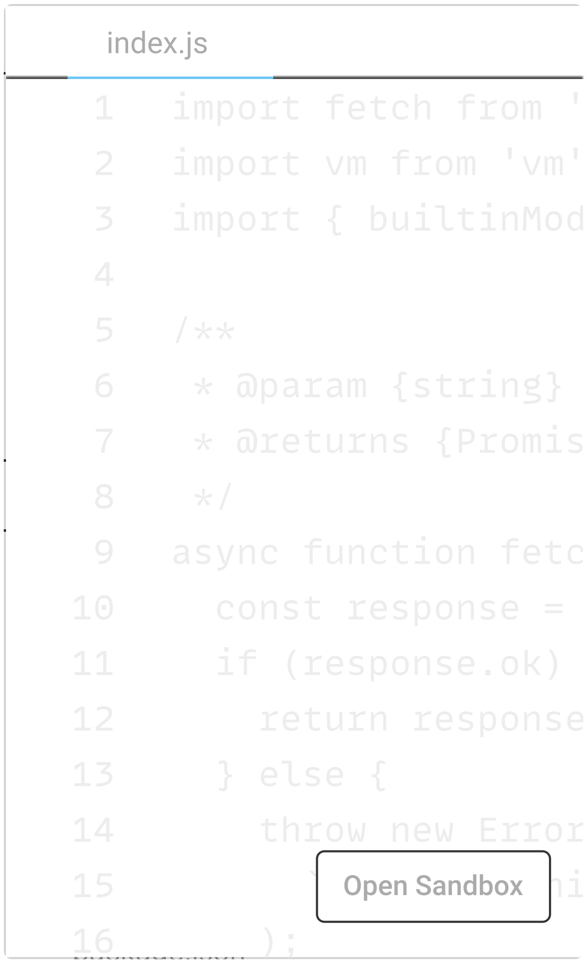
Loaders are another way to achieve some of the same goals. They provide a simpler API and enhance the behaviour of the native `import` statements. On the other hand, they are less flexible and they're possibly *even more experimental* than `vm.Module`.

There are many details and potential pitfalls to safely downloading and caching remotely hosted code that were not covered. Not to even mention the **security implications** of running arbitrary code. A more "production ready" (and potentially safer) runtime that uses HTTP imports is already available in **Deno**.

That said, it's interesting to see what can be achieved with the experimental APIs and there may be certain use cases where the risks to use them are calculable enough.

Complete example

Check out a complete working example on Code Sandbox:



```
import fetch from '
import vm from 'vm'
import { builtinMod

/**
 * @param {string}
 * @returns {Promis
 */
async function fetc
  const response =
  if (response.ok)
    return response
  } else {
    throw new Error
      `
    );
```

Or find the code in this repository:

mfellner / **react-micro-frontends**

Examples for React micro frontends

---

1. Skypack is nice because it offers ESM versions of most npm packages. ↩

Discussion (0)

Code of Conduct • Report abuse