Using modules in browser

Browsers are beginning to support es6 modules without polyfills! This means that we can take modules and use them as is without having to transpile if we're only supporting modern browsers.

We'll revisit modules: what they are and how they work. Unlike the cursory look we did when we discussed modules in the context of Rollup and Webpack, we'll take a deeper look at how do modules work in the browser and look at examples of how we can best leverage them today by using new syntax on the script tag.

ES6/ES2015 Modules

Modules allow you to package related variables and functions in a single module file. The data and functions in your modules are invisible to the outside world unless you explicitly make them available.

Browser support for ES2016 modules

Modules in browsers are mostly supported behind flags. The currect supported browsers are:

- Safari 10.1.
- Chrome Canary 60 – behind the Experimental Web Platform flag in `chrome:flags`
- Firefox 54 – behind the `dom.moduleScripts.enabled` setting in `about:config`
- Edge 15 – behind the Experimental JavaScript Features setting in `about:flags`

Why use modules

Just like ShadowDOM allows you to encapsulate HTML, CSS and Javascript, modules allow you to encapsualte your scripts. You have full control over what gets exposed to outside scripts and can keep your implementation details hidden by simply not exposing them.

Creating modules

There are two ways to create a module. External and internal modules, each of which can export and import multiple named functions from other modules.

Multiple exports and imports

Take the following `utils.js` external module that exports text maniplation functions: One to add text to a div in the body of a page and one to create an h1 element.

```
// utils.js
export function addTextToBody(text) {
  const div = document.createElement('div');
  div.textContent = text;
  document.body.appendChild(div);
}

export function createHeader(text) {
  const header = document.createElement('h1');
  header.textContent = text;
  document.body.appendChild(header);
}
```

This internal module imports `addTextToBody` and `createHeader` from `utils.js` and uses them as local functions without name spacing.

```
<script type="module">
  import {addTextToBody, createHeader} from 'utils';

  addTextToBody('Modules are pretty cool.');
</script>
```

You can rename your imports to make them easier to work with. Working with the same example, we can shorten the name of our `addTextToBody` import by using the `as` keyword and the name we want to give it. We then use the name we chose rather than the original function name.

```
<script type="module">
  import {
    addTextToBody as addText,
    createHeader} from 'utils';

  createHeader('Hellow World');
  addText('Modules are pretty cool.');
</script>
```

Importing the complete module

When we have multiple imports we can also import the complete module rather than specifying items to import. The module is written as normal.

When it comes to import and use the module, however, we use a different syntax.

```
import * as util from  'utils';

util.createHeader('wassup, doc');

util.addTextToBody('I\'m huntting wabbits');
```

Unlike when we imported specific functions we must qualify the functions we import using a wildcard. This may be useful when working with multiple modules as it may avoid name collisions.

Exporting a default function or class

We can also define a single function or class to export by adding the `default` keyworks to a class or function. In this example we export a `addTextToBody` as a default function.

```
// utils.js
export default function addTextToBody(text) {
  const div = document.createElement('div');
  div.textContent = text;
  document.body.appendChild(div);
}
```

You can also use anonymous functions declarations when working with default exports, we can make addTextToBody and anonymous functon and use it as a default export.

```
// utils.js
export default function (text) {
  const div = document.createElement('div');
  div.textContent = text;
  document.body.appendChild(div);
}
```

When it comes time to import it, we give it a name and use the same syntax we used with multiple imports. The name of the function we’re importing is less important, because we’ve identified the default function we want to import.

```
//------ main1.js ------
import addText from 'utils.js';
addText('hello');
```

We can do the same thing with classes. We declare a default export of an anonymous class.

```
// utilsClass.js
export default class { ··· } // no semicolon!
```

When it comes time to import the class we use the same syntax but we then initialize the class using a constant or variable, like shown below:

```
//------ main2.js ------
import MyClass from 'utilClass';
const inst = new MyClass();
```

Mix and match

You can also mix and match named and default exports. Doing this is perfectly legal:

```
export default function addTextToBody(text) {
  const div = document.createElement('div');
  div.textContent = text;
  document.body.appendChild(div);
}

export function createHeader(text) {
  const header = document.createElement('h1');
  header.textContent = text;
  document.body.appendChild(header);
}
```

and then use the following import statement:

```
import {default as addText, createHeader} from 'utils';

// do work with the functions
```

It is advisable to only mix the different export strategies in a single module only when you have a good reason. They will make code harder to reason

Fallbacks for older browsers

The last concern when working with native module implementations is how to handle older browsers. Most modern browsers have repurposed the `type` attribute of the `script` element: If it’s value is `module` the JS engine will treat the content as a module with different rules than those for normal scripts.

> To target older browsers use the `nomodule` attribute.

```
<script type="module" src="module.js"></script>
```

```
<script nomodule src="fallback.js"></script>
```

Differences between regular scripts and module scripts when used in the browsers (taken from exploring ES6:

	Scripts	**Modules**
HTML element	`<script>`	`<script type="module">`
Default mode	non-strict	strict
Top-level variables are	global	local to module
Value of `this` at top level	`window`	`undefined`
Executed	synchronously	asynchronously
Declarative imports (`import` statement)	no	yes
Programmatic imports (Promise-based API)	yes	yes
File extension	`.js`	`.js`

Things to consider

Imports and exports must be at the top level. They must be at the top most level of your script, otherwise they'll throw errors.

Imports are hoisted to the top of the script so it doesn't matter where they are in the script and you can use an imported function before you actually import it.

Imports are read-only views on export meaning that you can't change an imported function. If you need to change it make a local version of it and use it instead.

They only run once per page no matter how many times you load it.

Modules run on strict mode by default. There are several implications for this:

- Variables can’t be left undeclared
- Function parameters must have unique names (or are considered syntax errors)
- with is forbidden
- Errors are thrown on assignment to read-only properties
- Octal numbers like 00840 are syntax errors
- Attempts to delete undeletable properties throw an error delete prop is a syntax error, instead of assuming delete global[prop]
- eval doesn’t introduce new variables into its surrounding scope
- eval and arguments can’t be bound or assigned to
- arguments doesn’t magically track changes to method parameters
- arguments.callee throws a TypeError, no longer supported
- arguments.caller throws a TypeError, no longer supported
- Context passed as this in method invocations is not “boxed” (forced) into becoming an Object
- No longer able to use fn.caller and fn.arguments to access the JavaScript stack
- Reserved words (e.g protected, static, interface, etc) cannot be bound

Modules and script modules never block rendering. They always run as if the `defered` attribute was set in the calling script tag. The defer tag means that the script will execute after the content is downloaded but before the DOMContentLoaded event is fired.

Modules and inline (script) modules can use async attribute meaning that they can be made to load without blocking HTML rendering but it also means that you can no longer guarantee execution order. If the order your scripts run in is important rely on the defered attribute discussed earlier.

Must use a valid Javascript Mime Type or it will not execute. In this context, a valid Javascript Mime Type is one of those listed in the HTML Standard:

- application/ecmascript
- application/javascript
- application/x-ecmascript
- application/x-javascript
- text/ecmascript
- text/javascript
- text/javascript1.0
- text/javascript1.1
- text/javascript1.2

- text/javascript1.3
- text/javascript1.4
- text/javascript1.5
- text/jscript
- text/livescript
- text/x-ecmascript
- text/x-javascript

Performance may not be as good as we'd like.

Because module support in browser is new, it may not perform as well as bundled modules. Just as I was getting ready to publish this article I found a post on module performance versus bundled content.

In Browser module loading - can we stop bundling yet? Sérgio Gomes walks down how he tested performance of bundled versus unbundled modules. His results are interesting and worth trying to reproduce.

I expect things will improve as browsers fix bug and improve performance. The best solution was/is/will continue to be to use HTTP2 and preload.

Links and resources

- https://jakearchibald.com/2017/es-modules-in-browsers/
- https://medium.com/@samthor/es6-modules-in-chrome-canary-m60-ba588dfb8ab7
- http://2ality.com/2014/09/es6-modules-final.html
- https://ponyfoo.com/articles/es6-modules-in-depth
- Browser module loading - can we stop bundling yet?