**The State of JavaScript Modules** (30 minute presentation)

Web & Mobile Developers Meetup

08/22/19

**[SLIDE 1 - Introduction]**

Hi there! Thanks for coming. So today I’m going to be covering The State of JavaScript Modules, which is my attempt at explaining such a complicated subject.

We are going to learn about the history first, and then dive into some coding examples and the current state of modules.

So let’s start!

**[SLIDE 2 - JavaScript Growth]**

**[SLIDE 3 - Why Do We Need Modules?]**

*We do we need JavaScript modules? What’s the problem?*

Well, there are a few reasons. Mainly because JavaScript was moving away from the browser, and therefore no longer had access to the window object.

Since all the code would exist globally on the window object, name conflicts would occur frequently with different libraries and frameworks.

So basically, developers needed a way to contain their code.

**[SLIDE 3 - Scope of Presentation]**

JavaScript modules are essentially two parts. The module definitions which define the syntax and API you use when implementing them. And module loaders, which load your modules in the order of your dependency tree, synchronously or asynchronously.

We will not be covering module loaders for the sake of time.

**[SLIDE 4]**

The Mediocre Solution for Containing JavaScript Code

**[SLIDE 5 - THE REVEALING MODULE]**

Javascript does not have the typical 'private' and 'public' specifiers of more traditional object oriented languages like C# or Java. The Revealing Module pattern is a design pattern for Javascript applications that elegantly solves this problem.

**[SLIDE 6]**

The Unofficial Solutions for Containing JavaScript Code

**[SLIDE 7 - CommonJS]**

CommonJS was started in 2009 by Mozilla engineer Kevin Dangoor.

CommonJS is a project with the goal to establish conventions on module ecosystem for JavaScript outside of the web browser.

CommonJS is not affiliated with the ECMA group who define the ECMAScript standards.

Pros

* Simple: a developer can grasp the concept without looking at the docs.
* Dependency management is integrated: modules require other modules and get loaded in the needed order.
* require can be called anywhere: modules can be loaded programmatically.
* Circular dependencies are supported.

Cons

* Synchronous API makes it not suitable for certain uses (client-side).
* One file per module.
* Browsers require a loader library or transpiling.
* No constructor function for modules (Node supports this though).
* Hard to analyze for static code analyzers.

**[SLIDE 8 - AMD]**

AMD was born out of a group of developers that were displeased with the direction adopted by CommonJS. In fact, AMD was split from CommonJS early in its development. The main difference between AMD and CommonJS lies in its support for asynchronous module loading.

AMD modules can be loaded in any order.

Pros

* Asynchronous loading (better startup times).
* Circular dependencies are supported.
* Compatibility for require and exports.
* Dependency management fully integrated.
* Modules can be split in multiple files if necessary.
* Constructor functions are supported.
* Plugin support (custom loading steps).

Cons

* Slightly more complex syntactically.
* Loader libraries are required unless transpiled.
* Hard to analyze for static code analyzers.

**[SLIDE 9 - UMD]**

UMD is a pattern of universal module definition for JavaScript modules. These modules are capable of working everywhere, be it in the client, on the server or elsewhere.

Differences Between NodeJS and CommonJS

<https://www.quora.com/How-is-the-Node-js-module-system-different-than-CommonJS>