Table of Contents

Introduction

This is my first gitbook. The Souce:webpack.github.io

MOTIVATION

Today’s websites are evolving into web apps:

- More and more JavaScript is in a page.
- You can do more stuff in modern browsers.
- Fewer full page reloads → even more code in a page.

As a result there is a **lot** of code on the client side!

A big code base needs to be organized. Module systems offer the option to split your code base into modules.

MODULE SYSTEM STYLES

There are multiple standards for how to define dependencies and export values:

- `<script>`-tag style (without a module system)
- CommonJs
- AMD and some dialects of it
- ES6 modules
- and more…

`<script>`-tag style

This is the way you would handle a modularized code base if you didn’t use a module system.

Modules export an interface to the global object, i. e. the `window` object. Modules can access the interface of dependencies over the global object.

Common problems

- Conflicts in the global object.
- Order of loading is important.
- Developers have to resolve dependencies of modules/libraries.

- In big projects the list can get really long and difficult to manage.

CommonJs: synchronous `require`

This style uses a synchronous `require` method to load a dependency and return an exported interface. A module can specify `exports` by adding properties to the exports object or setting the value of `module.exports` .

require("module"); require("../file.js"); exports.doStuff = function() {}; module.exports = someValue; It’s used on server-side by node.js.

Pros

- Server-side modules can be reused
- There are already many modules in this style (npm)
- very simple and easy to use.

Cons

- blocking calls do not apply well on networks. Network requests are asynchronous.
- No parallel require of multiple modules

Implementations

- node.js - server-side
- browserify
- modules-webmake - compile to one bundle
- wreq - client-side

AMD: asynchronous require

`Asynchronous Module Definition`

Other module systems (for the browser) had problems with the synchronous `require` (CommonJs) and introduced an asynchronous version (and a way to define modules and exporting values):

require(["module", "../file"], function(module, file) { / ... / }); define("mymodule", ["dep1", "dep2"], function(d1, d2) { return someExportedValue; });

Pros

- Fits to the asynchronous request style in networks.
- Parallel loading of multiple modules.

Cons

- Coding overhead. More difficult to read and write.
- Seems to be some kind of workaround.

Implementations

- require.js - client-side
- curl - client-side

Read more about CommonJs and AMD.

ES6 modules

EcmaScript6 adds some language constructs to JavaScript, which form another module system.

import "jquery"; export function doStuff() {} module "localModule" {}

Pros

- Static analysis is easy
- Future-proof as ES standard

Cons

- Native browser support will take time
- Very few modules in this style

Unbiased solution

Give the developer the choice of the module style. Allow existing code to work. Make it easy to add custom module styles.

TRANSFERRING

Modules should be executed on the client, so they must be transferred from the server to the browser.

There are two extremes on how to transfer modules:

- 1 request per module
- all modules in one request

Both are used in the wild, but both are suboptimal:

- 1 request per module
- Pro: only required modules are transferred
- Con: many requests means much overhead
- Con: slow application startup, because of request latency
- all modules in one request
- Pro: less request overhead, less latency
- Con: not (yet) required modules are transferred too

Chunked transferring

A more flexible transferring would be better. A compromise between the extremes is better in most cases.

→ While compiling all modules: Split the set of modules into multiple smaller batches (chunks).

We get multiple smaller requests. Chunks with modules that are not required initially are only requested on demand. The initial request doesn't contain your complete code base and is smaller.

The "split points" are up to the developer and optional.

→ A big code base is possible!

Note: The idea is from Google's GWT.

Read more about Code Splitting.

WHY ONLY JAVASCRIPT?

Why should a module system only help the developer with JavaScript? There are many other static resources that need to be handled:

- stylesheets
- images
- webfonts
- html for templating
- etc.

And also:

- coffeescript → javascript
- less stylesheets → css stylesheets
- jade templates → javascript which generates html
- i18n files → something
- etc.

This should be as easy as:

`require("./style.css");` require("./style.less"); require("./template.jade"); require("./image.png"); Read more about Using loaders and Loaders.

STATIC ANALYSIS

When compiling all the modules a static analysis tries to find dependencies.

Traditionally this could only find simple stuff without expression, but i.e. require("./template/" + templateName + ".jade") is a common construct.

Many libraries are written in different styles. Some of them are very weird…

Strategy

A clever parser would allow most existing code to run. If the developer does something weird it would try to find the most compatible solution.