

# Dependency Management

## Modules and Packages with JavaScript

Qafoo GmbH  
June 11, 2013

What comes next?

---

# Welcome

# About Me

---

## Jakob Westhoff

- ▶ More than 12 years of professional PHP experience
- ▶ More than 9 years of professional JavaScript experience
- ▶ Open source enthusiast
- ▶ Regular speaker at (inter)national conferences
- ▶ Consultant, Trainer and Author

Working with



**Helping people to create  
high-quality Applications**

<http://qafoo.com>





What comes next?

---

# Motivation

# Dependency Management - Motivation

---

- ▶ Webapplications (Rich internet applications) codebases are growing
- ▶ Projects consist of multiple files
- ▶ Usage of external libraries gets more important every day
- ▶ Different builds are required for different platforms (Desktop, Mobile, Legacy, ...)

# Dependency Management - Idea

---

- ▶ Code is structured into smaller units
  - ▶ Packages
  - ▶ Modules
  - ▶ Prototypes ("Classes")
- ▶ Dependencies are defined by each unit
- ▶ Automated buildsystems integrate all needed units into one application
  - ▶ Dynamically (inside the browser)
  - ▶ Statically (during a build step)
  - ▶ Mixture of both



What comes next?

---

# CommonJS Modules

- ▶ CommonJS
  - ▶ Started in 2009
  - ▶ Goal: Specifying APIs for a JavaScript Eco System for more than only the browser
  - ▶ Open proposal process (Mailinglist)
  - ▶ Specifications only considered final after several implementations

# CommonJS specifications

---

- ▶ Current specifications
  - ▶ Modules (`require`)
  - ▶ Packages (`package.json`)
  - ▶ System (`stdin`, `stdout`, ...)

# CommonJS - require

---

- ▶ Specification for interoperable modules
- ▶ Isolated private scope for each module
- ▶ Possibility to import other modules using a custom name
- ▶ Possibility to export certain objects/functions for use in other modules
- ▶ Flexible implementation of look up logic to locate modules

# CommonJS - require

---

- ▶ Free variable exports and require available in any module

```
1 function add(lhs, rhs) {  
2     return lhs + rhs;  
3 }  
4  
5 function increment(lhs) {  
6     return add(lhs, 1);  
7 }  
8  
9 exports.inc = increment
```

```
1 var incMod =  
2     require('increment');  
3  
4 incMod.inc(41) // 42  
5  
6 // or  
7  
8 var inc =  
9     require('increment').inc;  
10  
11 inc(41) // 42
```

# CommonJS - require

---

- ▶ Free variable `module` can be used to override exports completely (Not actually specified)

```
1 function add(lhs, rhs) {  
2   return lhs + rhs;  
3 }  
4  
5 function increment(lhs) {  
6   return add(lhs, 1);  
7 }  
8  
9 module.exports = increment;
```

```
1 var inc =  
2   require('increment');  
3  
4 inc(41) // 42
```

# Challenges of require

---

- ▶ Determine list of Modules to load
  - ▶ Lazy loading?
  - ▶ Recursive search
- ▶ Locate modules in the filesystem, database, \$storage
  - ▶ Absolute path
  - ▶ Relative to the requiring module
  - ▶ Include path?

## require on the server-side

---

- ▶ Files can be loaded and evaluated **synchronously**
  - ▶ Modules are identified and initialized during runtime
  - ▶ `require` can be a synchronous action without any callback.
- ▶ Modules filepath is known for relative based resolving
- ▶ Some sort of include path can be checked for module if direct resolving failed.



# require inside the browser

---

- ▶ Lazy loading
  - ▶ Modules need to be fetched **asynchronously**
  - ▶ Synchronous require calls can't be used
- ▶ Eager loading
  - ▶ Fetch every needed module before executing the application
  - ▶ May load and evaluate more JavaScript code than needed
  - ▶ Needs static dependency analysis during some build step
  - ▶ Works with synchronous require calls

What comes next?

---

# Asynchronous Module Definition API

# Asynchronous Module Definition API (AMD)

---

- ▶ Extracted from CommonJS Transport/C
- ▶ API specifying an **asynchronous** way of declaring modules
- ▶ Aimed at systems like browsers, who can't request modules synchronously
- ▶ Can be used as a transport for already available CommonJS modules
  - ▶ Needs some sort of preprocessing
- ▶ **Only a specification not a real implementation**

# AMD - define

---

- ▶ Global function define has to be made available by any implementation
- ▶ `define([id], [dependencies], factory)`

```
1 define(  
2   'increment',  
3   ['add'],  
4   function( add ) {  
5       var increment = function(a) {  
6           return add(a,1);  
7       };  
8  
9       return {  
10          'inc': increment  
11      };  
12  }  
13 );
```

# AMD in the wild

---

- ▶ Full specification available online
  - ▶ <https://github.com/amdjs/amdjs-api/wiki/AMD>
- ▶ A lot of different libraries are starting to support AMD:
  - ▶ jQuery
  - ▶ MooTools
  - ▶ Dojo
  - ▶ ...
- ▶ NodeJS and Browser implementations exist

# AMD - other cool aspects

---

- ▶ Simple CommonJS Wrapper Syntax
- ▶ Support for special LoaderObjects
  - ▶ Load in evaluate resources in a special way (templates, json, coffee-script)

# CommonJS Wrapper Syntax

---

- ▶ CommonJS Modules are using special variables/functions to handle requirements
  - ▶ `require`
  - ▶ `exports`
  - ▶ `module`
- ▶ Possibility to use CommonJS Modules inside the browser desirable

# CommonJS Wrapper Syntax

---

- ▶ Problem: Synchronous calling behaviour
  - ▶ `require` immediately returns the requested module
- ▶ Solution: Asynchronous wrapper + a little bit of magic
  - ▶ Async wrapper around `require`, `exports`, `module`
  - ▶ Static code analysis of module to isolate needed dependencies



# CommonJS Wrapper Syntax - Example

---

```
1 var otherModule = require('otherModule');
2
3 function add(a, b) {
4     return a + b;
5 }
6
7 function inc(value) {
8     return value + 1;
9 }
10
11 exports.inc = inc;
```

# CommonJS Wrapper Syntax - Example

---

```
1  define(function(require, exports, module ){
2
3      var otherModule = require('otherModule');
4
5      function add(a, b) {
6          return a + b;
7      }
8
9      function inc(value) {
10         return value + 1;
11     }
12
13     exports.inc = inc;
14
15 });
```

# Using CommonJS Modules

---

- ▶ A great amount of CommonJS modules can be used
- ▶ Wrapping process can be automated
- ▶ A lot of people always use the wrapped syntax, as they find it more readable

# Limitations of Wrapped Syntax

---

- ▶ The wrapped syntax has certain limitations
- ▶ Non static require dependencies can't be resolved
  - ▶ `var foo = require(["path", "file"].join("/"));`
- ▶ Modules can't be given an arbitrary name
  - ▶ The name is given by their filepath

What comes next?

---

# Require.js

# Require.js

---

- ▶ Implementation of the AMD specification
- ▶ Small footprint (6,1kb minified+gzipped)
- ▶ Well documented
- ▶ Quite feature complete optimizer (r.js)
- ▶ Node.js AMD bridge
- ▶ Vast amount of LoaderObjects

# Using requirejs

---

- ▶ Create an application conforming to the AMD spec
- ▶ Download the `require.js` loader from <http://requirejs.org>
- ▶ Add `require.js` to your main application html
- ▶ Require.js will take it from here

# File structure

---

For Require.js to work properly you need a certain file structure

application-directory

- ▶ index.html
- ▶ CSS
- ▶ ...
- ▶ **scripts**
  - ▶ **require.js**
  - ▶ **main.js**
  - ▶ **module1.js**
  - ▶ **subfolder**
    - ▶ **module2.js**



# Require.js - Loading the library

---

- ▶ Loading `require.js` into your application context

```
1 <html>
2   <head>
3     <script data-main="scripts/main"
4         src="scripts/require.js">
5     </script>
6   </head>
7   <body> ... </body>
8 </html>
```

- ▶ You should only provide one application entry point
- ▶ Specify the entry point using `data-main`
- ▶ This allows for easier usage of the **optimizer** later on

# Require.js - An example

---

- ▶ A project consisting of 4 files:

File: scripts/main.js

```
1 require(['a', 'b'], function(a, b) {  
2 });
```

File: scripts/a.js

```
1 require(['c'], function(c) {  
2 });
```

File: scripts/b.js

```
1 require(['c'], function(c) {  
2 });
```

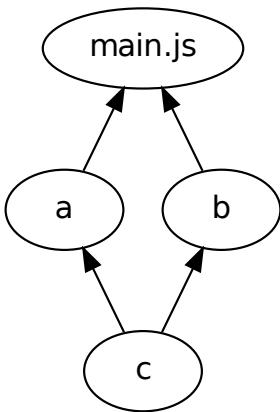
File: scripts/c.js

```
1 require([], function() {  
2 });
```














# Require.js - An example

---

- ▶ Graphical representation of example dependencies



# Require.js - An example

Name Path	Method	Status Text	Type	Initiator	Size Content	Time Latency	Timeline	9ms	13ms	17ms	21ms	26ms	30ms	34ms	38ms
 <b>Index.html</b> /Users/jakot	GET	Success	text/html	Other	(from ca...	1ms 1ms									
 <b>require.js</b> /Users/jakot	GET	Success	text/jav...	<a href="#">index.html?Z</a> Parser	(from ca...	1ms 1ms									
 <b>main.js</b> /Users/jakot	GET	Success	text/jav...	<a href="#">require.js:1884</a> Script	0B 42B	1ms 0									
 <b>ajs</b> /Users/jakot	GET	Success	text/jav...	<a href="#">require.js:1884</a> Script	0B 34B	3ms 0									
 <b>b.js</b> /Users/jakot	GET	Success	text/jav...	<a href="#">require.js:1884</a> Script	0B 34B	2ms 0									
 <b>c.js</b> /Users/jakot	GET	Success	text/jav...	<a href="#">require.js:1884</a> Script	0B 30B	2ms 0									

# What comes next?

---

r.js

## Disadvantages of AMD in production use

---

- ▶ Dynamic loading of resources is nice during development
- ▶ It's mostly catastrophic for production use
- ▶ The application should be packaged into one or multiple bigger modules
- ▶ Packages should be properly minified and/or compressed

`r.js` automatically does that for AMD modules

## r.js - The optimizer

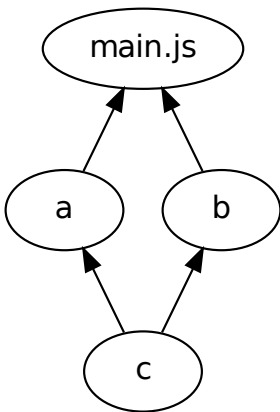
---

- ▶ `r.js` is part of `require.js`
- ▶ Compatible with all AMD conform implementations
- ▶ Allows for automatic dependency tracing, combining and minification
- ▶ Extensibly configurable to include non AMD modules as well
- ▶ Split your app into a defined set of packages and auto generate them

# Optimizing the a,b,c example

---

- ▶ Remember the example utilizing the a, b and c modules?





# Optimizing the a,b,c example

---

- ▶ Before optimization can be done a lean configuration for `r.js` is needed
- ▶ The configuration can be placed anywhere inside or outside the application tree
- ▶ Only the paths inside the configuration need match your project

# Optimizing the a,b,c example

---

- ▶ Configuration for our a, b, c example project
- ▶ Named `app.build.config` and placed at the project root

```
1 {
2   appDir: './',
3   baseUrl: 'scripts',
4   dir: 'build',
5   modules: [
6     {
7       name: 'main',
8     },
9   ]
10 }
```

# Optimizing the a,b,c example

---





- ▶ After creating a configuration simply run `r.js` with it

```
1 r.js -o app.build.config
```

```
1 Tracing dependencies for: main
2 Uglifying file: /Users/jakob/playground/requirejs/build/app.build.js
3 Uglifying file: /Users/jakob/playground/requirejs/build/scripts/a.js
4 Uglifying file: /Users/jakob/playground/requirejs/build/scripts/b.js
5 Uglifying file: /Users/jakob/playground/requirejs/build/scripts/c.js
6 Uglifying file: /Users/jakob/playground/requirejs/build/scripts/main.js
7 Uglifying file: /Users/jakob/playground/requirejs/build/scripts/require.js
8
9 scripts/main.js
10 -----
11 scripts/c.js
12 scripts/a.js
13 scripts/b.js
14 scripts/main.js
```

# Optimizing the a,b,c example

- ▶ `r.js` creates a copy of the whole project including any resource
- ▶ Including the build `main.js` with resolved and embedded dependencies
- ▶ Simply open the `index.html` from the build folder

Name Path	Meth...	Status Text	Type	Initiator	Size Conten	Time Latency	Timeline	12ms	17ms	23ms	29ms
 <code>index.html</code> /Users/jakob/playgrou	GET	Succ...	text/...	Other	(fro...	1ms 1ms					
 <code>require.js</code> /Users/jakob/playgrou	GET	Succ...	text/...	<code>index.html:7</code> Parser	0B 16.11Ki	1ms 0					
 <code>main.js</code> /Users/jakob/playgrou	GET	Succ...	text/...	<code>require.js:7</code> Script	0B 232B	1ms 0					

# Almond - Minimal Loader Shim

---





- ▶ Loading the complete `require.js` library can be a significant overhead
- ▶ After all files have been combined there is not reason for this
- ▶ The XHR and asynchronous handler code is not needed anymore
- ▶ Only a basic shim providing the necessary API is needed for the project to run

# Almond - Minimal Loader Shim

---

- ▶ Almond is such a minimal loader
- ▶ It can be directly embedded into the build source file
- ▶ Only **1.3kb** (minified+gzipped) in size
- ▶ <https://github.com/jrburke/almond>

# Almond - Minimal Loader Shim

Name Path	Meth...	Status Text	Type	Initiator	Size Conten	Time Latenc)	Timeline	6 ms	9 ms	12 ms	16 ms
 <b>index.html</b> /Users/jakob/Qafoo/d	GET	Success	text/...	Other	0 B 138 B	2 ms 0					
 <b>main-built.js</b> /Users/jakob/Qafoo/d	GET	Success	text/...	<a href="#">index.html:5</a> Parser	0 B 3.1 KB	2 ms 0					

What comes next?

---

# Alternatives



# Alternative Dependency Management tools

---

- ▶ Most sophisticated inner app dependency management:  
`require.js`, `r.js`
- ▶ Dependency management on different layers is required as well
  - ▶ External libraries
  - ▶ Ressources
  - ▶ Components
  - ▶ ...

# Different Layer Alternatives

---

- ▶ npm (Node package manager)
  - ▶ <http://npmjs.org>
  - ▶ Package management utility for nodejs applications
  - ▶ Installation of dependencies from `package.json`
  - ▶ Does provide in-browser libraries like jQuery and underscore as well
  - ▶ Specialized in JavaScript files only

# Different Layer Alternatives

---

- ▶ Jam

- ▶ <http://jamjs.org>
- ▶ Package manager for in-browser packages
- ▶ Generates needed `require.js` configuration automatically
- ▶ Depends on `require.js` as a dependency manager

# Different Layer Alternatives

---

- ▶ Bower

- ▶ <http://twitter.github.com/bower>
- ▶ Newly created dependency management system especially for the in-browser resources
- ▶ Manages JavaScript, CSS, HTML, ...
- ▶ Like npm, but for the browser libraries
- ▶ Integrates nicely with `require.js` **or any other dependency management tool**

# Different Layer Alternatives

---

- ▶ Components

- ▶ <https://github.com/component/component>
- ▶ Client-side centric package manager
- ▶ **Only** usable with CommonJS or AMD modules
- ▶ Allows for integration of different Assets (Images, CSS, Fonts, ...)
- ▶ Uses `github` as a registry

# Different Layer Alternatives

---

- ▶ Volo
  - ▶ <http://volojs.org/>
  - ▶ Package manager as well as **build system**
  - ▶ Compatible to CommonJS and AMD modules
  - ▶ Hard to integrate with other build automation tools like `grunt.js`

# Different Layer Alternatives

---

- ▶ `ender.js`
  - ▶ <http://ender.no.de/>
  - ▶ npm based dependency management tool especially for in-browser libraries
  - ▶ Does include packaging and minification
  - ▶ Hard to integrate with application-level dependency management
  - ▶ Good at integrating different micro-frameworks into one library

What comes next?

---

# Conclusion



# Conclusion

---

- ▶ Complex JavaScript applications consist of a lot of different parts
- ▶ CommonJS Modules provide a clean way of defining them
- ▶ Modules inside the browser have special requirements (asynchronous loading)
- ▶ AMD is a solution to this problem
- ▶ `require.js` and `r.js` the most sophisticated solution using AMD
- ▶ Alternatives for other levels of dependency management exist (Bower, Jam, ...)



THANK YOU

Rent a quality expert  
[qafoo.com](http://qafoo.com)