require.js Documentation

Release 1.0.1

Tomasz Grajewski

December 14, 2013

Contents

CHAPTER 1

Introduction

If you have written code in other programming languages (like Python for example), then you already should know what's a module. *require.js* allows you to write your JavaScript code in a similar manner, to divide code into separate modules and packages, clearly stating dependencies and relations between files. It's very powerful idea, that will change how you write code. Your code will become simpler, more readable and maintainable.

This solution is similar to the RequireJS, which does support asynchronous code loading. See also:

- AMD API wiki
- Why AMD? (Require.js)
- Dojo Tutorial: AMD Modules

If you are an advanced user of RequireJS, then you should know, that the solution described here is in many areas different and many 'features' were not implemented on purpose. Summarizing this up: overall idea is very similar, but API has many differences. This solution was inspired by the Python Philosophy.

1.1 What's a module?

A *module* is a single JavaScript file, that is a collection of related functions or classes or both. In case of a more complicated code a module may also contain only one class definition or only one function. Module should be a reusable piece of code, that you will want to use in many software projects.

A *package* is a group of modules (usually somewhat related).

JavaScript at the time of writing this, doesn't support any kind of modules or packages. We've borrowed this idea from other programming languages to make writing complex JavaScript applications easier.

1.2 Integrating require.js

Dependencies: None. **Supported browsers:** All (IE8 and older with an Object.create() polyfill).

To add *require.js* to your site, just include it before any of your code and other libraries, like jQuery. This is done usually at the end of the <body> tag:

```
        ...

        <script src="scripts/require.js"></script>  <!-- Add this line -->
        <script src="scripts/settings.js"></script>  <!-- This is optional, you need to create this
        <!-- Other modules in your application. Note that third-party modules probably won't be compa
        <script src="scripts/jquery.js"></script>
        <script src="scripts/my_module.js"></script>
        ...
    </body>
</html>
```

Note: *require.js* won't load your modules asynchronously. This is on purpose, to reduce confusion and increase performance of complicated applications. Ask your back end developer to merge all JavaScript modules to a single file, this is done usually by using some third-party static files compressor.

1.3 How to 'create' a module?

To define a module, just create a JavaScript file, and write something like this:

```
define('great_module', function() {

    // My module code goes here...
    function myFunction() {
        return 5;
    }

    // 'Export' any things you want other developers (or you) should use.
    return {'myFunction': myFunction};
});
```

The `function` statement at the beginning of the above example is a simple way to make all your code private and decide which functions and classes should be available to others using the `return` statement at the end of module definition.

To use `myFunction` in other modules use the global `require()` function, which is somewhat similar to Python's `import` statement (at least the whole idea is similar).

```
// 'great_module' must match with what was specified in the define() call above.
var greatModule = require('great_module');

greatModule.myFunction();
```

Below is a more practical example:

```
define('numbers', function() {

    function radians(n) {
        return n * Math.PI / 180;
    }

    function degrees(n) {
        return n * 180 / Math.PI;
    }

    return {
```

```
        'radians': radians,
        'degrees': degrees
    };
});
```

Above is a simple module example with two functions that help converting angles between radians and degrees. When you want to use these functions, again use `require()` *anywhere* in your code:

```
var numbers = require('numbers');

numbers.degrees(Math.PI);
numbers.radians(180);
```

1.4 Dependencies between modules

When you have code divided into many small modules it's very important to explicitly state, how modules depend on each other. This is usually written at the beginning of a module's source code for readability. Using *require.js* you can state dependencies in a `define()` call like this:

```
// Define a module and use some code from the 'numbers' module without calling require().
define('my_custom_module', ['numbers'], function(numbers) {

    // This is a space of an other module. Here you can use the numbers module from previous example

    function fullCircle() {
        return numbers.radians(Math.PI * 2);
    }

    return {'fullCircle': fullCircle}
});
```

In the above example the `'my_custom_module'` uses a `'numbers'` module, this is stated in the `define()` call.

`dependencies` is just an array of module names that are required, so the current module can work.

```
define(moduleName, [dependencies], function(dependency1, dependency2, ...) {
    ...
});
```

1.5 An alternate syntax

You can also use a shorter syntax when you want to create module that's a group of constants or functions.

```
define('config', {
    'DEBUG': true,
    'FPS': 60,
    ...
});


var config = require('config');

if (config.DEBUG)
    ...
```

CHAPTER 2

The 'settings' module

You can define configuration options for your application in a 'settings' module. Then, other modules will reference this settings module and treat it as a central point of a run time configuration.

```
define('settings', {
    // Used by require.js, defaults to true, used to toggle 'no conflict mode' for jQuery.
    'JQUERY_NO_CONFLICT': false,

    // Other custom, user-defined settings example:
    'DEBUG': true,
    'ANIMATIONS': true,
    'FPS': 60,
    ...
});
```

Although 'settings' in the above example is a simple JavaScript object, when using it in other modules, *require.js* gives you a getter function to ease development.

```
var settings = require('settings');

if (settings('DEBUG', false))
    ...
```

settings in this case is a function, that returns given named configuration options and in case of an unknown, not defined options, this function returns a fallback value (the second argument).

CHAPTER 3

Good practices

Below is a list of good practices, that when followed should somewhat increase code quality and readability.

1. Module names should match 1:1 to JavaScript file names (without extension). Module `'numbers'` should reside in a file named `numbers.js`. Modules that are inside sub-directories should include those directories in the module name. So a module *numbers* placed in a directory *math* should be named `'math/numbers'`.

   Usually file names are all lower case, also consider separating words with an underscore character. Actually CamelCase in module names is not supported and when such module name is used, `define()` will throw an exception.

2. Although everything in a module code is private, you should export as much as possible, so other developers won't have problems to reuse your code. When some variables or functions are considered *internal*, then you may prepend an underscore character to their name, so other developers will know that they are messing with some internals.

3. In case of many dependencies it's recommended to write them using the `require()` function. Instead:

   ```
   define('my_module', ['dep1', 'dep1', 'dep3', 'dep4', ...], function(dep1, dep2, dep3, dep4, ...)
       ...
   ```

   Write this:

   ```
   define('my_module', function() {
       var dep1 = require('dep1');
       var dep2 = require('dep2');
       var dep3 = require('dep3');
       var dep4 = require('dep4');
       ...
   ```

   The above notation is more verbose, but also more readable in case of many dependencies.

4. If your module needs to initialize itself in some way, it's better if time of this initialization can be chosen at run time. For example instead of adding event listeners to some DOM elements, you could write a pair if functions `install()/uninstall()` or `enable()/disable()`, so other developers using your module can decide when they want to initialize given libraries (probably as late as possible to improve loading time).

CHAPTER 4

Differences from other popular implementations

Simple. Just look at the source code, it's just damn 2 simple functions. Actually it's more error checking than actual code.

Compatible. Well, almost. Syntax is very near to the RequireJS, so if you won't like this solution or need more features, then you should be able to painlessly swap implementations.

No asynchronous code loading. When dividing your project into many small modules you may end up with lots of JavaScript files. Loading them into browser will be slow, so merging them to a single file is recommended. What's also recommended is to let your back end developers combine and compress JS files into single one.

Loading other files, like text, CSV files is also not possible. If you need these features then use RequireJS, but it may also mean that you're probably making your code complicated (hope you've a reason to!).

Cruft-free. No need to install any temporarily-popular servers, package managers, parsers and other useless stuff. It's just one JavaScript file.

Exceptions. `define()` and `require()` will throw meaningful exceptions that should ease you debugging in case you get lost. No error should pass silently.

Small. Around 5 KB of code when uncompressed, mostly due to custom exception classes. Less than 1 KB compressed and gzipped.

Settings. Standardizes one place to store all your app-wide configuration options, that your modules can leverage.

Bug-free. Seriously, the code is very simple, also lack of redundant or useless features helps in this regard.

jQuery support. You can toggle 'no conflict mode' in the settings module. Include jQuery *after* `require.js` and after `settings.js` (if you have one).

Written with *The Zen of Python* in mind.

CHAPTER 5

API Reference

define(*moduleName*[, *dependencies*], *moduleCode*)

Creates a module from `moduleCode` and stores it in the `define.modules` container for later retrieval with `require()`.

Arguments

- **moduleName** (*string*) – Must be a string with proper name. Allowed characters are lower case letters, digits, underscores and slashes (when module nested in sub-directories). `moduleName` should be an absolute path to a module, including the file name, but without extension. This value must be unique, no two modules with the same name are allowed.
- **dependencies** (*array*) – Optional, can be specified only if `moduleCode` is a function. `dependencies` must be an array of strings, where each string is name of some other module. Each module must be loaded prior to this point and will be passed as an argument to the function specified in `moduleCode`.
- **moduleCode** – Should be a callback function that returns module definition. It will be called immediately and will receive all modules specified in `dependencies` as arguments. `moduleCode` can be also any other object, which may prove useful in case of defining application settings or other constant values.

Throws

- **define.Error** – When `define()` is called with `dependencies` specified, but `moduleCode` is not a function.
- **define.ArgumentCountError** – When called with not enough or too many arguments.
- **define.InvalidModuleNameError** – When `moduleName` contains not allowed characters or is empty or is not a string.
- **define.InvalidModuleError** – When `moduleCode` is `undefined` or it's function that doesn't return anything.
- **define.DuplicateModuleError** – When given `moduleName` is already used by an other module.

Returns undefined

require(*moduleName*)

Retrieves module from the internal module storage (IE. `define.modules`). `moduleName` must be a string.

Throws

- **require.Error** – When given module specified in `moduleName` doesn't exist (was not defined).
- **require.ArgumentsError** – When arguments count is not one or `moduleName` is not a string.

Returns module definition, that is any object stored previously with `define()` call.

CHAPTER 6

Testing

Tests are written using Jasmine framework. Just open the `tests/SpecRunner.html` file in any browser to run tests.

CHAPTER 7

Version history

v1.0.1 Added IE8 and IE7 support when an `Object.create()` polyfill is present.

v1.0.0 Initial release.

CHAPTER 8

License

The MIT License (MIT)

Copyright © 2013 Tomasz Grajewski

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sub license, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Jasmine © 2008-2013 Pivotal Labs