REQUIREJS API

This is the RequireJS 2.0 API. If you want 1.0: Link to 1.0.

USAGE

§ 1

Load JavaScript Files

§ 1.1

RequireJS takes a different approach to script loading than traditional <script> tags. While it can also run fast and optimize well, the primary goal is to encourage modular code. As part of that, it encourages using **module IDs** instead of URLs for script tags.

RequireJS loads all code relative to a baseUrl. The baseUrl is normally set to the same directory as the script used in a data-main attribute for the top level script to load for a page. The data-main attribute is a special attribute that require.js will check to start script loading. This example will end up with a baseUrl of **scripts**:

```
<!--This sets the baseUrl to the "scripts" directory, and
    loads a script that will have a module ID of 'main'-->
<script data-main="scripts/main.js" src="scripts/require.js"></script>
```

Or, baseUrl can be set manually via the RequireJS config. If there is no explicit config and data-main is not used, then the default baseUrl is the directory that contains the HTML page running RequireJS.

RequireJS also assumes by default that all dependencies are scripts, so it does not expect to see a trailing ".js" suffix on module IDs. RequireJS will automatically add it when translating the module ID to a path. With the paths config, you can set up locations of a group of scripts. All of these capabilities allow you to use smaller strings for scripts as compared to traditional <script> tags.

There may be times when you do want to reference a script directly and not conform to the "baseUrl + paths" rules for finding it. If a module ID has one of the following characteristics, the ID will not be passed through the "baseUrl + paths" configuration, and just be treated like a regular URL that is relative to the document:

- *Ends in ".js".*
- *Starts with a "/".*
- *Contains an URL protocol, like "http:" or "https:".*

In general though, it is best to use the baseUrl and "paths" config to set paths for module IDs. By doing so, it gives you more flexibility in renaming and configuring the paths to different locations for optimization builds.

Similarly, to avoid a bunch of configuration, it is best to avoid deep folder hierarchies for scripts, and instead either keep all the scripts in baseUrl, or if you want to separate your library/vendor-supplied code from your app code, use a directory layout like this:

- *www/*
  - *index.html*
  - *js/*
    - *app/*
      - *sub.js*
    - *lib/*
      - *jquery.js*
      - *canvas.js*
    - *app.js*
    - *require.js*

in index.html:

```
<script data-main="js/app.js" src="js/require.js"></script>
```

and in app.js:

```
requirejs.config({
    //By default load any module IDs from js/lib
    baseUrl: 'js/lib',
    //except, if the module ID starts with "app",
    //load it from the js/app directory. paths
    //config is relative to the baseUrl, and
    //never includes a ".js" extension since
    //the paths config could be for a directory.
    paths: {
        app: '../app'
    }
});

// Start the main app logic.
requirejs(['jquery', 'canvas', 'app/sub'],
function   ($,        canvas,    sub) {
    //jQuery, canvas and the app/sub module are all
    //loaded and can be used here now.
});
```

Notice as part of that example, vendor libraries like jQuery did not have their version numbers in their file names. It is recommended to store that version info in a separate text file if you want to track it, or if you use a tool like volo, it will stamp the package.json with the version information but keep the file on disk as "jquery.js". This allows you to have the very minimal configuration instead of having to put an entry in the "paths" config for each library. For instance, configure "jquery" to be "jquery-1.7.2".

Ideally the scripts you load will be modules that are defined by calling define(). However, you may

need to use some traditional/legacy "browser globals" scripts that do not express their dependencies via define(). For those, you can use the shim config. To properly express their dependencies.

If you do not express the dependencies, you will likely get loading errors since RequireJS loads scripts asynchronously and out of order for speed.

data-main Entry Point

§ 1.2

The data-main attribute is a special attribute that require.js will check to start script loading:

```
<!--when require.js loads it will inject another script tag
    (with async attribute) for scripts/main.js-->
<script data-main="scripts/main" src="scripts/require.js"></script>
```

You will typically use a data-main script to set configuration options and then load the first application module. Note: the script tag require.js generates for your data-main module includes the async attribute. This means that **you cannot assume that the load and execution of your data-main script will finish prior to other scripts referenced later in the same page.**

For example, this arrangement will fail randomly when the require.config path for the 'foo' module has not been set prior to it being require()'d later:

```
<script data-main="scripts/main" src="scripts/require.js"></script>
<script src="scripts/other.js"></script>
```

```
// contents of main.js:
require.config({
    paths: {
        foo: 'libs/foo-1.1.3'
    }
});
```

```
// contents of other.js:

// This code might be called before the require.config() in main.js
// has executed. When that happens, require.js will attempt to
// load 'scripts/foo.js' instead of 'scripts/libs/foo-1.1.3.js'
require(['foo'], function(foo) {

});
```

If you want to do require() calls in the HTML page, then it is best to not use data-main. data-main is only intended for use when the page just has one main entry point, the data-main script. For pages that want to do inline require() calls, it is best to nest those inside a require() call for the configuration:

```
<script src="scripts/require.js"></script>
<script>
```

```
require(['scripts/config'], function() {
    // Configuration loaded now, safe to do other require calls
    // that depend on that config.
    require(['foo'], function(foo) {

    });
});
</script>
```

Define a Module

§ 1.3

A module is different from a traditional script file in that it defines a well-scoped object that avoids polluting the global namespace. It can explicitly list its dependencies and get a handle on those dependencies without needing to refer to global objects, but instead receive the dependencies as arguments to the function that defines the module. Modules in RequireJS are an extension of the Module Pattern, with the benefit of not needing globals to refer to other modules.

The RequireJS syntax for modules allows them to be loaded as fast as possible, even out of order, but evaluated in the correct dependency order, and since global variables are not created, it makes it possible to load multiple versions of a module in a page.

(If you are familiar with or are using CommonJS modules, then please also see CommonJS Notes for information on how the RequireJS module format maps to CommonJS modules).

There should only be **one** module definition per file on disk. The modules can be grouped into optimized bundles by the optimization tool.

Simple Name/Value Pairs

§ 1.3.1

If the module does not have any dependencies, and it is just a collection of name/value pairs, then just pass an object literal to define():

```
//Inside file my/shirt.js:
define({
    color: "black",
    size: "unisize"
});
```

Definition Functions

§ 1.3.2

If the module does not have dependencies, but needs to use a function to do some setup work, then define itself, pass a function to define():

```
//my/shirt.js now does setup work
//before returning its module definition.
```

```
define(function () {
    //Do setup work here

    return {
        color: "black",
        size: "unisize"
    }
});
```

Definition Functions with Dependencies

§ 1.3.3

If the module has dependencies, the first argument should be an array of dependency names, and the second argument should be a definition function. The function will be called to define the module once all dependencies have loaded. The function should return an object that defines the module. The dependencies will be passed to the definition function as function arguments, listed in the same order as the order in the dependency array:

```
//my/shirt.js now has some dependencies, a cart and inventory
//module in the same directory as shirt.js
define(["./cart", "./inventory"], function(cart, inventory) {
        //return an object to define the "my/shirt" module.
        return {
            color: "blue",
            size: "large",
            addToCart: function() {
                inventory.decrement(this);
                cart.add(this);
            }
        }
    }
);
```

In this example, a my/shirt module is created. It depends on my/cart and my/inventory. On disk, the files are structured like this:

- *my/cart.js*
- *my/inventory.js*
- *my/shirt.js*

The function call above specifies two arguments, "cart" and "inventory". These are the modules represented by the "./cart" and "./inventory" module names.

The function is not called until the my/cart and my/inventory modules have been loaded, and the function receives the modules as the "cart" and "inventory" arguments.

Modules that define globals are explicitly discouraged, so that multiple versions of a module can exist in a page at a time (see **Advanced Usage**). Also, the order of the function arguments should match the order of the dependencies.

The return object from the function call defines the "my/shirt" module. By defining modules in this way, "my/shirt" does not exist as a global object.

Define a Module as a Function
§ 1.3.4

Modules do not have to return objects. Any valid return value from a function is allowed. Here is a module that returns a function as its module definition:

```
//A module definition inside foo/title.js. It uses
//my/cart and my/inventory modules from before,
//but since foo/title.js is in a different directory than
//the "my" modules, it uses the "my" in the module dependency
//name to find them. The "my" part of the name can be mapped
//to any directory, but by default, it is assumed to be a
//sibling to the "foo" directory.
define(["my/cart", "my/inventory"],
    function(cart, inventory) {
        //return a function to define "foo/title".
        //It gets or sets the window title.
        return function(title) {
            return title ? (window.title = title) :
                   inventory.storeName + ' ' + cart.name;
        }
    }
);
```

Define a Module with Simplified CommonJS Wrapper
§ 1.3.5

If you wish to reuse some code that was written in the traditional CommonJS module format it may be difficult to re-work to the array of dependencies used above, and you may prefer to have direct alignment of dependency name to the local variable used for that dependency. You can use the simplified CommonJS wrapper for those cases:

```
define(function(require, exports, module) {
        var a = require('a'),
            b = require('b');

        //Return the module value
        return function () {};
    }
);
```

This wrapper relies on Function.prototype.toString() to give a useful string value of the function contents. This does not work on some devices like the PS3 and some older Opera mobile browsers. Use the optimizer to pull out the dependencies in the array format for use on those devices.

More information is available on the CommonJS page, and in the "Sugar" section in the Why AMD page.

Define a Module with a Name
§ 1.3.6

You may encounter some define() calls that include a name for the module as the first argument to define():

```
//Explicitly defines the "foo/title" module:
define("foo/title",
    ["my/cart", "my/inventory"],
    function(cart, inventory) {
        //Define foo/title object in here.
    }
);
```

These are normally generated by the optimization tool. You can explicitly name modules yourself, but it makes the modules less portable -- if you move the file to another directory you will need to change the name. It is normally best to avoid coding in a name for the module and just let the optimization tool burn in the module names. The optimization tool needs to add the names so that more than one module can be bundled in a file, to allow for faster loading in the browser.

Other Module Notes
§ 1.3.7

One module per file.: Only one module should be defined per JavaScript file, given the nature of the module name-to-file-path lookup algorithm. You shoud only use the optimization tool to group multiple modules into optimized files.

Relative module names inside define(): For require("./relative/name") calls that can happen inside a define() function call, be sure to ask for "require" as a dependency, so that the relative name is resolved correctly:

```
define(["require", "./relative/name"], function(require) {
    var mod = require("./relative/name");
});
```

Or better yet, use the shortened syntax that is available for use with translating CommonJS modules:

```
define(function(require) {
    var mod = require("./relative/name");
});
```

This form will use Function.prototype.toString() to find the require() calls, and add them to the dependency array, along with "require", so the code will work correctly with relative paths.

Relative paths are really useful if you are creating a few modules inside a directory, so that you can share the directory with other people or other projects, and you want to be able to get a handle on the sibling modules in that directory without having to know the directory's name.

Relative module names are relative to other names, not paths: The loader stores modules by their name and not by their path internally. So for relative name references, those are resolved relative to the module name making the reference, then that module name, or ID, is converted to a path if needs to be loaded. Example code for a 'compute' package that has a 'main' and 'extras' modules in it:

```
* lib/
    * compute/
        * main.js
        * extras.js
```

where the main.js module looks like this:

```
define(["./extras"], function(extras) {
    //Uses extras in here.
});
```

If this was the paths config:

```
require.config({
    baseUrl: 'lib',
    paths: {
      'compute': 'compute/main'
    }
});
```

And a require(['compute']) is done, then lib/compute/main.js will have the module name of 'compute'. When it asks for './extras', that is resolved relative to 'compute', so 'compute/./extras', which normalizes to just 'extras'. Since there is no paths config for that module name, the path generated will be for 'lib/extras.js', which is incorrect.

For this case, packages config is a better option, since it allows setting the main module up as 'compute', but internally the loader will store the module with the ID of 'compute/main' so that the relative reference for './extras' works.

Another option is to construct a module at lib/compute.js that is just define(['./compute/main'], function(m) { return m; });, then there is no need for paths or packages config.

Or, do not set that paths or packages config and do the top level require call as require(['compute/main']).

Generate URLs relative to module: You may need to generate an URL that is relative to a module. To do so, ask for "require" as a dependency and then use require.toUrl() to generate the URL:

```
define(["require"], function(require) {
    var cssUrl = require.toUrl("./style.css");
});
```

Console debugging: If you need to work with a module you already loaded via a require(["module/name"], function(){}) call in the JavaScript console, then you can use the require() form that just uses the string name of the module to fetch it:

```
require("module/name").callSomeFunction()
```

Note this only works if "module/name" was previously loaded via the async version of require: require(["module/name"]). If using a relative path, like './module/name', those only work inside define

Circular Dependencies

§ 1.3.8

If you define a circular dependency ("a" needs "b" and "b" needs "a"), then in this case when "b"'s module function is called, it will get an undefined value for "a". "b" can fetch "a" later after modules have been defined by using the require() method (be sure to specify require as a dependency so the right context is used to look up "a"):

```
//Inside b.js:
define(["require", "a"],
    function(require, a) {
        //"a" in this case will be null if "a" also asked for "b",
        //a circular dependency.
        return function(title) {
            return require("a").doSomething();
        }
    }
);
```

Normally you should not need to use require() to fetch a module, but instead rely on the module being passed in to the function as an argument. Circular dependencies are rare, and usually a sign that you might want to rethink the design. However, sometimes they are needed, and in that case, use require() as specified above.

If you are familiar with CommonJS modules, you could instead use **exports** to create an empty object for the module that is available immediately for reference by other modules. By doing this on both sides of a circular dependency, you can then safely hold on to the the other module. This only works if each module is exporting an object for the module value, not a

function:

```
//Inside b.js:
define(function(require, exports, module) {
    //If "a" has used exports, then we have a real
    //object reference here. However, we cannot use
    //any of "a"'s properties until after "b" returns a value.
    var a = require("a");

    exports.foo = function () {
        return a.bar();
    };
});
```

Or, if you are using the dependency array approach, ask for the special 'exports' dependency:

```
//Inside b.js:
define(['a', 'exports'], function(a, exports) {
    //If "a" has used exports, then we have a real
    //object reference here. However, we cannot use
    //any of "a"'s properties until after "b" returns a value.

    exports.foo = function () {
        return a.bar();
    };
});
```

Specify a JSONP Service Dependency

§ 1.3.9

JSONP is a way of calling some services in JavaScript. It works across domains and it is an established approach to calling services that just require an HTTP GET via a script tag.

To use a JSONP service in RequireJS, specify "define" as the callback parameter's value. This means you can get the value of a JSONP URL as if it was a module definition.

Here is an example that calls a JSONP API endpoint. In this example, the JSONP callback parameter is called "callback", so "callback=define" tells the API to wrap the JSON response in a "define()" wrapper:

```
require(["http://example.com/api/data.json?callback=define"],
    function (data) {
        //The data object will be the API response for the
        //JSONP data call.
        console.log(data);
    }
);
```

This use of JSONP should be limited to JSONP services for initial application setup. If the

JSONP service times out, it means other modules you define via define() may not get executed, so the error handling is not robust.

Only JSONP return values that are JSON objects are supported. A JSONP response that is an array, a string or a number will not work.

This functionality should not be used for long-polling JSONP connections -- APIs that deal with real time streaming. Those kinds of APIs should do more script cleanup after receiving each response, and RequireJS will only fetch a JSONP URL once -- subsequent uses of the same URL as a dependency in a require() or define() call will get a cached value.

Errors in loading a JSONP service are normally surfaced via timeouts for the service, since script tag loading does not give much detail into network problems. To detect errors, you can override requirejs.onError() to get errors. There is more information in the Handling Errors section.

Undefining a Module

§ 1.3.10

There is a global function, **requirejs.undef()**, that allows undefining a module. It will reset the loader's internal state to forget about the previous definition of the module.

However, it will not remove the module from other modules that are already defined and got a handle on that module as a dependency when they executed. So it is really only useful to use in error situations when no other modules have gotten a handle on a module value, or as part of any future module loading that may use that module. See the errback section for an example.

If you want to do more sophisticated dependency graph analysis for undefining work, the semi-private onResourceLoad API may be helpful.

MECHANICS

§ 2

RequireJS loads each dependency as a script tag, using head.appendChild().

RequireJS waits for all dependencies to load, figures out the right order in which to call the functions that define the modules, then calls the module definition functions once the dependencies for those functions have been called. Note that the dependencies for a given module definition function could be called in any order, due to their sub-dependency relationships and network load order.

Using RequireJS in a server-side JavaScript environment that has synchronous loading should be as

easy as redefining require.load(). The build system does this, the require.load method for that environment can be found in build/jslib/requirePatch.js.

In the future, this code may be pulled into the require/ directory as an optional module that you can load in your env to get the right load behavior based on the host environment.

CONFIGURATION OPTIONS

§ 3

When using require() in the top-level HTML page (or top-level script file that does not define a module), a configuration object can be passed as the first option:

```
<script src="scripts/require.js"></script>
<script>
  require.config({
    baseUrl: "/another/path",
    paths: {
        "some": "some/v1.0"
    },
    waitSeconds: 15
  });
  require( ["some/module", "my/module", "a.js", "b.js"],
    function(someModule,    myModule) {
        //This function will be called when all the dependencies
        //listed above are loaded. Note that this function could
        //be called before the page is loaded.
        //This callback is optional.
    }
  );
</script>
```

You may also call require.config from your data-main Entry Point, but be aware that the data-main script is loaded asynchronously. Avoid other entry point scripts which wrongly assume that data-main and its require.config will always execute prior to their script loading.

Also, you can define the config object as the global variable require **before** require.js is loaded, and have the values applied automatically. This example specifies some dependencies to load as soon as require.js defines require():

```
<script>
    var require = {
        deps: ["some/module1", "my/module2", "a.js", "b.js"],
        callback: function(module1, module2) {
            //This function will be called when all the dependencies
            //listed above in deps are loaded. Note that this
```

```
        //function could be called before the page is loaded.
        //This callback is optional.
    }
};
</script>
<script src="scripts/require.js"></script>
```

Note: It is best to use var require = {} and do not use window.require = {}, it will not behave correctly in IE.

There are some patterns for separating the config from main module loading.

Supported configuration options:

baseUrl: the root path to use for all module lookups. So in the above example, "my/module"'s script tag will have a src="/another/path/my/module.js". baseUrl is **not** used when loading plain .js files (indicated by a dependency string starting with a slash, has a protocol, or ends in .js), those strings are used as-is, so a.js and b.js will be loaded from the same directory as the HTML page that contains the above snippet.

If no baseUrl is explicitly set in the configuration, the default value will be the location of the HTML page that loads require.js. If a **data-main** attribute is used, that path will become the baseUrl.

The baseUrl can be a URL on a different domain as the page that will load require.js. RequireJS script loading works across domains. The only restriction is on text content loaded by text! plugins: those paths should be on the same domain as the page, at least during development. The optimization tool will inline text! plugin resources so after using the optimization tool, you can use resources that reference text! plugin resources from another domain.

paths: path mappings for module names not found directly under baseUrl. The path settings are assumed to be relative to baseUrl, unless the paths setting starts with a "/" or has a URL protocol in it ("like http:"). Using the above sample config, "some/module"'s script tag will be src="/another/path/some/v1.0/module.js".

The path that is used for a module name should **not** include an extension, since the path mapping could be for a directory. The path mapping code will automatically add the .js extension when mapping the module name to a path. If require.toUrl() is used, it will add the appropriate extension, if it is for something like a text template.

When run in a browser, paths fallbacks can be specified, to allow trying a load from a CDN location, but falling back to a local location if the CDN location fails to load.

bundles: Introduced in RequireJS 2.1.10: allows configuring multiple module IDs to be found in another script. Example:

```
requirejs.config({
    bundles: {
        'primary': ['main', 'util', 'text', 'text!template.html'],
        'secondary': ['text!secondary.html']
    }
```

```
});

require(['util', 'text'], function(util, text) {
    //The script for module ID 'primary' was loaded,
    //and that script included the define()'d
    //modules for 'util' and 'text'
});
```

That config states: modules 'main', 'util', 'text' and 'text!template.html' will be found by loading module ID 'primary'. Module 'text!secondary.html' can be found by loading module ID 'secondary'.

This only sets up where to find a module inside a script that has multiple define()'d modules in it. It does not automatically bind those modules to the bundle's module ID. The bundle's module ID is just used for locating the set of modules.

Something similar is possible with paths config, but it is much wordier, and the paths config route does not allow loader plugin resource IDs in its configuration, since the paths config values are path segments, not IDs.

bundles config is useful if doing a build and that build target was not an existing module ID, or if you have loader plugin resources in built JS files that should not be loaded by the loader plugin. **Note that the keys and values are module IDs**, not path segments. They are absolute module IDs, not a module ID prefix like paths config or map config. Also, bundle config is different from map config in that map config is a one-to-one module ID relationship, where bundle config is for pointing multiple module IDs to a bundle's module ID.

As of RequireJS 2.2.0, the optimizer can generate the bundles config and insert it into the top level requirejs.config() call. See the bundlesConfigOutFile build config option for more details.

shim: Configure the dependencies, exports, and custom initialization for older, traditional "browser globals" scripts that do not use define() to declare the dependencies and set a module value.

Here is an example. It requires RequireJS 2.1.0+, and assumes backbone.js, underscore.js and jquery.js have been installed in the baseUrl directory. If not, then you may need to set a paths config for them:

```
requirejs.config({
    //Remember: only use shim config for non-AMD scripts,
    //scripts that do not already call define(). The shim
    //config will not work correctly if used on AMD scripts,
    //in particular, the exports and init config will not
    //be triggered, and the deps config will be confusing
    //for those cases.
    shim: {
        'backbone': {
            //These script dependencies should be loaded before loading
            //backbone.js
            deps: ['underscore', 'jquery'],
            //Once loaded, use the global 'Backbone' as the
            //module value.
            exports: 'Backbone'
```

```
        },
        'underscore': {
            exports: '_'
        },
        'foo': {
            deps: ['bar'],
            exports: 'Foo',
            init: function (bar) {
                //Using a function allows you to call noConflict for
                //libraries that support it, and do other cleanup.
                //However, plugins for those libraries may still want
                //a global. "this" for the function will be the global
                //object. The dependencies will be passed in as
                //function arguments. If this function returns a value,
                //then that value is used as the module export value
                //instead of the object found via the 'exports' string.
                //Note: jQuery registers as an AMD module via define(),
                //so this will not work for jQuery. See notes section
                //below for an approach for jQuery.
                return this.Foo.noConflict();
            }
        }
    }
});

//Then, later in a separate file, call it 'MyModel.js', a module is
//defined, specifying 'backbone' as a dependency. RequireJS will use
//the shim config to properly load 'backbone' and give a local
//reference to this module. The global Backbone will still exist on
//the page too.
define(['backbone'], function (Backbone) {
  return Backbone.Model.extend({});
});
```

In RequireJS 2.0.*, the "exports" property in the shim config could have been a function instead of a string. In that case, it functioned the same as the "init" property as shown above. The "init" pattern is used in RequireJS 2.1.0+ so a string value for exports can be used for enforceDefine, but then allow functional work once the library is known to have loaded.

For "modules" that are just jQuery or Backbone plugins that do not need to export any module value, the shim config can just be an array of dependencies:

```
requirejs.config({
    shim: {
        'jquery.colorize': ['jquery'],
        'jquery.scroll': ['jquery'],
        'backbone.layoutmanager': ['backbone']
    }
});
```

Note however if you want to get 404 load detection in IE so that you can use paths fallbacks or

errbacks, then a string exports value should be given so the loader can check if the scripts actually loaded (a return from init is **not** used for enforceDefine checking):

```
requirejs.config({
    shim: {
        'jquery.colorize': {
            deps: ['jquery'],
            exports: 'jQuery.fn.colorize'
        },
        'jquery.scroll': {
            deps: ['jquery'],
            exports: 'jQuery.fn.scroll'
        },
        'backbone.layoutmanager': {
            deps: ['backbone']
            exports: 'Backbone.LayoutManager'
        }
    }
});
```

Important notes for "shim" config:

- *The shim config only sets up code relationships. To load modules that are part of or use shim config, a normal require/define call is needed. Setting shim by itself does not trigger code to load.*
- *Only use other "shim" modules as dependencies for shimmed scripts, or AMD libraries that have no dependencies and call define() after they also create a global (like jQuery or lodash). Otherwise, if you use an AMD module as a dependency for a shim config module, after a build, that AMD module may not be evaluated until after the shimmed code in the build executes, and an error will occur. The ultimate fix is to upgrade all the shimmed code to have optional AMD define() calls.*
- *If it is not possible to upgrade the shimmed code to use AMD define() calls, as of RequireJS 2.1.11, the optimizer has a wrapShim build option that will try to automatically wrap the shimmed code in a define() for a build. This changes the scope of shimmed dependencies, so it is not guaranteed to always work, but, for example, for shimmed dependencies that depend on an AMD version of Backbone, it can be helpful.*
- *The init function will **not** be called for AMD modules. For example, you cannot use a shim init function to call jQuery's noConflict. See Mapping Modules to use noConflict for an alternate approach to jQuery.*
- *Shim config is not supported when running AMD modules in node via RequireJS (it works for optimizer use though). Depending on the module being shimmed, it may fail in Node because Node does not have the same global environment as browsers. As of RequireJS 2.1.7, it will warn you in the console that shim config is not supported, and it may or may not work. If you wish to suppress that message, you can pass requirejs.config({ suppress: { nodeShim: true }});.*

Important optimizer notes for "shim" config:

- *You should use the mainConfigFile build option to specify the file where to find the shim config. Otherwise the optimizer will not know of the shim config. The other option is to duplicate the shim config in the build profile.*

- *Do not mix CDN loading with shim config in a build. Example scenario: you load jQuery from the CDN but use the shim config to load something like the stock version of Backbone that depends on jQuery. When you do the build, be sure to inline jQuery in the built file and do not load it from the CDN. Otherwise, Backbone will be inlined in the built file and it will execute before the CDN-loaded jQuery will load. This is because the shim config just delays loading of the files until dependencies are loaded, but does not do any auto-wrapping of define. After a build, the dependencies are already inlined, the shim config cannot delay execution of the non-define()'d code until later. define()'d modules do work with CDN loaded code after a build because they properly wrap their source in define factory function that will not execute until dependencies are loaded. So the lesson: shim config is a stop-gap measure for non-modular code, legacy code. define()'d modules are better.*
- *For local, multi-file builds, the above CDN advice also applies. For any shimmed script, its dependencies* ***must*** *be loaded before the shimmed script executes. This means either building its dependencies directly in the buid layer that includes the shimmed script, or loading its dependencies with a require([], function (){}) call, then doing a nested require([]) call for the build layer that has the shimmed script.*
- *If you are using uglifyjs to minify the code,* ***do not*** *set the uglify option toplevel to true, or if using the command line* ***do not*** *pass –mt. That option mangles the global names that shim uses to find exports.*

map: For the given module prefix, instead of loading the module with the given ID, substitute a different module ID.

This sort of capability is really important for larger projects which may have two sets of modules that need to use two different versions of 'foo', but they still need to cooperate with each other.

This is not possible with the context-backed multiversion support. In addition, the paths config is only for setting up root paths for module IDs, not for mapping one module ID to another one.

map example:

```
requirejs.config({
    map: {
        'some/newmodule': {
            'foo': 'foo1.2'
        },
        'some/oldmodule': {
            'foo': 'foo1.0'
        }
    }
});
```

If the modules are laid out on disk like this:

- *foo1.0.js*
- *foo1.2.js*
- *some/*
  - *newmodule.js*
  - *oldmodule.js*

When 'some/newmodule' does `require('foo')` it will get the foo1.2.js file, and when 'some/oldmodule' does `require('foo')` it will get the foo1.0.js file.

This feature only works well for scripts that are real AMD modules that call define() and register as anonymous modules. Also, **only use absolute module IDs** for map config. Relative IDs (like '../some/thing') do not work.

There is also support for a "*" map value which means "for all modules loaded, use this map config". If there is a more specific map config, that one will take precedence over the star config. Example:

```
requirejs.config({
    map: {
        '*': {
            'foo': 'foo1.2'
        },
        'some/oldmodule': {
            'foo': 'foo1.0'
        }
    }
});
```

Means that for any module except "some/oldmodule", when "foo" is wanted, use "foo1.2" instead. For "some/oldmodule" only, use "foo1.0" when it asks for "foo".

Note: when doing builds with map config, the map config needs to be fed to the optimizer, and the build output must still contain a requirejs config call that sets up the map config. The optimizer does not do ID renaming during the build, because some dependency references in a project could depend on runtime variable state. So the optimizer does not invalidate the need for a map config after the build.

config: There is a common need to pass configuration info to a module. That configuration info is usually known as part of the application, and there needs to be a way to pass that down to a module. In RequireJS, that is done with the **config** option for requirejs.config(). Modules can then read that info by asking for the special dependency "module" and calling **module.config()**. Example:

```
requirejs.config({
    config: {
        'bar': {
            size: 'large'
        },
        'baz': {
            color: 'blue'
        }
    }
});

//bar.js, which uses simplified CJS wrapping:
//http://requirejs.org/docs/whyamd.html#sugar
define(function (require, exports, module) {
```

```
    //Will be the value 'large'
    var size = module.config().size;
});

//baz.js which uses a dependency array,
//it asks for the special module ID, 'module':
//https://github.com/requirejs/requirejs/wiki/Differences-between-the-simplified-CommonJS-wrapp
define(['module'], function (module) {
    //Will be the value 'blue'
    var color = module.config().color;
});
```

For passing config to a package, target the main module in the package, not the package ID:

```
requirejs.config({
    //Pass an API key for use in the pixie package's
    //main module.
    config: {
        'pixie/index': {
            apiKey: 'XJKDLNS'
        }
    },
    //Set up config for the "pixie" package, whose main
    //module is the index.js file in the pixie folder.
    packages: [
        {
            name: 'pixie',
            main: 'index'
        }
    ]
});
```

packages: configures loading modules from CommonJS packages. See the packages topic for more information.

nodeIdCompat: Node treats module ID example.js and example the same. By default these are two different IDs in RequireJS. If you end up using modules installed from npm, then you may need to set this config value to true to avoid resolution issues. This option only applies to treating the ".js" suffix differently, it does not do any other node resolution and evaluation matching such as .json file handling (JSON handling needs a 'json!' loader plugin anyway). Available in 2.1.10 and greater.

waitSeconds: The number of seconds to wait before giving up on loading a script. Setting it to 0 disables the timeout. The default is 7 seconds.

context: A name to give to a loading context. This allows require.js to load multiple versions of modules in a page, as long as each top-level require call specifies a unique context string. To use it correctly, see the Multiversion Support section.

deps: An array of dependencies to load. Useful when require is defined as a config object before require.js is loaded, and you want to specify dependencies to load as soon as require() is defined.

Using deps is just like doing a require([]) call, but done as soon as the loader has processed the configuration. **It does not block** any other require() calls from starting their requests for modules, it is just a way to specify some modules to load asynchronously as part of a config block.

callback: A function to execute after **deps** have been loaded. Useful when require is defined as a config object before require.js is loaded, and you want to specify a function to require after the configuration's **deps** array has been loaded.

enforceDefine: If set to true, an error will be thrown if a script loads that does not call define() or have a shim exports string value that can be checked. See Catching load failures in IE for more information.

xhtml: If set to true, document.createElementNS() will be used to create script elements.

urlArgs: Extra query string arguments appended to URLs that RequireJS uses to fetch resources. Most useful to cache bust when the browser or server is not configured correctly. Example cache bust setting for urlArgs:

```
urlArgs: "bust=" +  (new Date()).getTime()
```

As of RequireJS 2.2.0, urlArgs can be a function. If a function, it will receive the module ID and the URL as parameters, and it should return a string that will be added to the end of the URL. Return an empty string if no args. Be sure to take care of adding the '?' or '&' depending on the existing state of the URL. Example:

```
requirejs.config({
    urlArgs: function(id, url) {
        var args = 'v=1';
        if (url.indexOf('view.html') !== -1) {
            args = 'v=2'
        }

        return (url.indexOf('?') === -1 ? '?' : '&') + args;
    }
});
```

During development it can be useful to use this, however **be sure** to remove it before deploying your code.

scriptType: Specify the value for the type="" attribute used for script tags inserted into the document by RequireJS. Default is "text/javascript". To use Firefox's JavaScript 1.8 features, use "text/javascript;version=1.8".

skipDataMain: Introduced in RequireJS 2.1.9: If set to true, skips the data-main attribute scanning done to start module loading. Useful if RequireJS is embedded in a utility library that may interact with other RequireJS library on the page, and the embedded version should not do data-main loading.

ADVANCED USAGE

§ 4

Loading Modules from Packages

§ 4.1

RequireJS supports loading modules that are in a CommonJS Packages directory structure, but some additional configuration needs to be specified for it to work. Specifically, there is support for the following CommonJS Packages features:

- *A package can be associated with a module name/prefix.*
- *The package config can specify the following properties for a specific package:*
  - ***name**: The name of the package (used for the module name/prefix mapping)*
  - ***location**: The location on disk. Locations are relative to the baseUrl configuration value, unless they contain a protocol or start with a front slash (/).*
  - ***main**: The name of the module inside the package that should be used when someone does a require for "packageName". The default value is "main", so only specify it if it differs from the default. The value is relative to the package folder.*

IMPORTANT NOTES

- *While the packages can have the CommonJS directory layout, the modules themselves should be in a module format that RequireJS can understand. Exception to the rule: if you are using the r.js Node adapter, the modules can be in the traditional CommonJS module format. You can use the CommonJS converter tool if you need to convert traditional CommonJS modules into the async module format that RequireJS uses.*
- *Only one version of a package can be used in a project context at a time. You can use RequireJS multiversion support to load two different module contexts, but if you want to use Package A and B in one context and they depend on different versions of Package C, then that will be a problem. This may change in the future.*

If you use a similar project layout as specified in the Start Guide, the start of your web project would look something like this (Node/Rhino-based projects are similar, just use the contents of the **scripts** directory as the top-level project directory):

- *project-directory/*
  - *project.html*
  - *scripts/*
    - *require.js*

Here is how the example directory layout looks with two packages, **cart** and **store**:

- *project-directory/*
  - *project.html*

- *scripts/*
  - *cart/*
    - *main.js*
  - *store/*
    - *main.js*
    - *util.js*
  - *main.js*
  - *require.js*

project.html will have a script tag like this:

```
<script data-main="scripts/main" src="scripts/require.js"></script>
```

This will instruct require.js to load scripts/main.js. **main.js** uses the "packages" config to set up packages that are relative to require.js, which in this case are the source packages "cart" and "store":

```
//main.js contents
//Pass a config object to require
require.config({
    "packages": ["cart", "store"]
});

require(["cart", "store", "store/util"],
function (cart,   store,   util) {
    //use the modules as usual.
});
```

A require of "cart" means that it will be loaded from **scripts/cart/main.js**, since "main" is the default main module setting supported by RequireJS. A require of "store/util" will be loaded from **scripts/store/util.js**.

If the "store" package did not follow the "main.js" convention, and looked more like this:

- *project-directory/*
  - *project.html*
  - *scripts/*
    - *cart/*
      - *main.js*
    - *store/*
      - *store.js*
      - *util.js*
    - *main.js*
    - *package.json*
    - *require.js*

Then the RequireJS configuration would look like so:

```
require.config({
    packages: [
        "cart",
        {
            name: "store",
            main: "store"
        }
    ]
});
```

To avoid verbosity, it is strongly suggested to always use packages that use "main" convention in their structure.

Multiversion Support

§ 4.2

As mentioned in Configuration Options, multiple versions of a module can be loaded in a page by using different "context" configuration options. require.config() returns a require function that will use the context configuration. Here is an example that loads two different versions of the alpha and beta modules (this example is taken from one of the test files):

```
<script src="../require.js"></script>
<script>
var reqOne = require.config({
  context: "version1",
  baseUrl: "version1"
});

reqOne(["require", "alpha", "beta",],
function(require,   alpha,   beta) {
  log("alpha version is: " + alpha.version); //prints 1
  log("beta version is: " + beta.version); //prints 1

  setTimeout(function() {
    require(["omega"],
      function(omega) {
        log("version1 omega loaded with version: " +
            omega.version); //prints 1
      }
    );
  }, 100);
});

var reqTwo = require.config({
      context: "version2",
      baseUrl: "version2"
    });

reqTwo(["require", "alpha", "beta"],
function(require,   alpha,   beta) {
  log("alpha version is: " + alpha.version); //prints 2
```

```
  log("beta version is: " + beta.version); //prints 2

  setTimeout(function() {
    require(["omega"],
      function(omega) {
        log("version2 omega loaded with version: " +
            omega.version); //prints 2
      }
    );
  }, 100);
});
</script>
```

Note that "require" is specified as a dependency for the module. This allows the require() function that is passed to the function callback to use the right context to load the modules correctly for multiversion support. If "require" is not specified as a dependency, then there will likely be an error.

Loading Code After Page Load

§ 4.3

The example above in the **Multiversion Support** section shows how code can later be loaded by nested require() calls.

Web Worker Support

§ 4.4

As of release 0.12, RequireJS can be run inside a Web Worker. Just use importScripts() inside a web worker to load require.js (or the JS file that contains the require() definition), then call require.

You will likely need to set the **baseUrl** configuration option to make sure require() can find the scripts to load.

You can see an example of its use by looking at one of the files used in the unit test.

Rhino Support

§ 4.5

RequireJS can be used in Rhino via the r.js adapter. See the r.js README for more information.

Nashorn Support

§ 4.6

As of RequireJS 2.1.16, RequireJS can be used in Nashorn, Java 8+'s JavaScript engine, via the r.js adapter. See the r.js README for more information.

Handling Errors

§ 4.7

The general class of errors are 404s for scripts (not found), network timeouts or errors in the scripts that are loaded. RequireJS has a few tools to deal with them: require-specific errbacks, a "paths" array config, and a global requirejs.onError.

The error object passed to errbacks and the global requirejs.onError function will usually contain two custom properties:

- ***requireType****: A string value with a general classification, like "timeout", "nodefine", "scripterror".*
- ***requireModules****: an array of module names/URLs that timed out.*

If you get an error with a requireModules, it probably means other modules that depend on the modules in that requireModules array are not defined.

Catching load failures in IE

§ 4.6.1

Internet Explorer has a set of problems that make it difficult to detect load failures for errbacks/paths fallbacks:

- *script.onerror does not work in IE 6-8. There is no way to know if loading a script generates a 404, worse, it triggers the onreadystatechange with a complete state even in a 404 case.*
- *script.onerror does work in IE 9+, but it has a bug where it does not fire script.onload event handlers right after execution of script, so it cannot support the standard method of allowing anonymous AMD modules. So script.onreadystatechange is still used. However, onreadystatechange fires with a complete state before the script.onerror function fires.*

So it is very difficult with IE to allow both anonymous AMD modules, which are a core benefit of AMD modules, and reliable detect errors.

However, if you are in a project that you know uses define() to declare all of its modules, or it uses the shim config to specify string exports for anything that does not use define(), then if you set the enforceDefine config value to true, the loader can confirm if a script load by checking for the define() call or the existence of the shim's exports global value.

So if you want to support Internet Explorer, catch load errors, and have modular code either through direct define() calls or shim config, always set **enforceDefine** to be true. See the next section for an example.

NOTE: If you do set enforceDefine: true, and you use data-main="" to load your main JS module, then that main JS module **must call define()** instead of require() to load the code it needs. The main JS module can still call require/requirejs to set config values, but for loading modules it should use define().

If you then also use almond to build your code without require.js, be sure to use the insertRequire build setting to insert a require call for the main module -- that serves the same purpose of the initial require() call that data-main does.

require([]) errbacks

§ 4.6.2

Errbacks, when used with requirejs.undef(), will allow you to detect if a module fails to load, undefine that module, reset the config to a another location, then try again.

A common use case for this is to use a CDN-hosted version of a library, but if that fails, switch to loading the file locally:

```
requirejs.config({
    enforceDefine: true,
    paths: {
        jquery: 'http://ajax.googleapis.com/ajax/libs/jquery/1.4.4/jquery.min'
    }
});

//Later
require(['jquery'], function ($) {
    //Do something with $ here
}, function (err) {
    //The errback, error callback
    //The error has a list of modules that failed
    var failedId = err.requireModules && err.requireModules[0];
    if (failedId === 'jquery') {
        //undef is function only on the global requirejs object.
        //Use it to clear internal knowledge of jQuery. Any modules
        //that were dependent on jQuery and in the middle of loading
        //will not be loaded yet, they will wait until a valid jQuery
        //does load.
        requirejs.undef(failedId);

        //Set the path to jQuery to local path
        requirejs.config({
            paths: {
                jquery: 'local/jquery'
            }
        });

        //Try again. Note that the above require callback
        //with the "Do something with $ here" comment will
        //be called if this new attempt to load jQuery succeeds.
        require(['jquery'], function () {});
    } else {
        //Some other error. Maybe show message to the user.
    }
});
```

With `requirejs.undef()`, if you later set up a different config and try to load the same module, the loader will still remember which modules needed that dependency and finish loading them when the newly configured module loads.

Note: errbacks only work with callback-style require calls, not define() calls. define() is only for declaring modules.

paths config fallbacks

§ 4.6.3

The above pattern for detecting a load failure, undef()ing a module, modifying paths and reloading is a common enough request that there is also a shorthand for it. The paths config allows array values:

```
requirejs.config({
    //To get timely, correct error triggers in IE, force a define/shim exports check.
    enforceDefine: true,
    paths: {
        jquery: [
            'http://ajax.googleapis.com/ajax/libs/jquery/1.4.4/jquery.min',
            //If the CDN location fails, load from this location
            'lib/jquery'
        ]
    }
});

//Later
require(['jquery'], function ($) {
});
```

This above code will try the CDN location, but if that fails, fall back to the local lib/jquery.js location.

Note: paths fallbacks only work for exact module ID matches. This is different from normal paths config which can apply to any part of a module ID prefix segment. Fallbacks are targeted more for unusual error recovery, not a generic path search path solution, since those are inefficient in the browser.

Global requirejs.onError function

§ 4.6.4

To detect errors that are not caught by local errbacks, you can override requirejs.onError():

```
requirejs.onError = function (err) {
    console.log(err.requireType);
    if (err.requireType === 'timeout') {
        console.log('modules: ' + err.requireModules);
    }

    throw err;
};
```

LOADER PLUGINS

§ 5

RequireJS supports loader plugins. This is a way to support dependencies that are not plain JS files, but are still important for a script to have loaded before it can do its work. The RequireJS wiki has a list of plugins. This section talks about some specific plugins that are maintained alongside RequireJS:

Specify a Text File Dependency

§ 5.1

It is nice to build HTML using regular HTML tags, instead of building up DOM structures in script. However, there is no good way to embed HTML in a JavaScript file. The best that can be done is using a string of HTML, but that can be hard to manage, particularly for multi-line HTML.

RequireJS has a plugin, text.js, that can help with this issue. It will automatically be loaded if the text! prefix is used for a dependency. See the text.js README for more information.

Page Load Event Support/DOM Ready

§ 5.2

It is possible when using RequireJS to load scripts quickly enough that they complete before the DOM is ready. Any work that tries to interact with the DOM should wait for the DOM to be ready. For modern browsers, this is done by waiting for the DOMContentLoaded event.

However, not all browsers in use support DOMContentLoaded. The domReady module implements a cross-browser method to determine when the DOM is ready. Download the module and use it in your project like so:

```
require(['domReady'], function (domReady) {
  domReady(function () {
    //This function is called once the DOM is ready.
    //It will be safe to query the DOM and manipulate
    //DOM nodes in this function.
  });
});
```

Since DOM ready is a common application need, ideally the nested functions in the API above could be avoided. The domReady module also implements the Loader Plugin API, so you can use the loader plugin syntax (notice the **!** in the domReady dependency) to force the require() callback function to wait for the DOM to be ready before executing. domReady will return the current document when used as a loader plugin:

```
require(['domReady!'], function (doc) {
    //This function is called once the DOM is ready,
    //notice the value for 'domReady!' is the current
    //document.
});
```

Note: If the document takes a while to load (maybe it is a very large document, or has HTML script tags loading large JS files that block DOM completion until they are done), using domReady as a

loader plugin may result in a RequireJS "timeout" error. If this a problem either increase the waitSeconds configuration, or just use domReady as a module and call domReady() inside the require() callback.

Define an I18N Bundle

§ 5.3

Once your web app gets to a certain size and popularity, localizing the strings in the interface and providing other locale-specific information becomes more useful. However, it can be cumbersome to work out a scheme that scales well for supporting multiple locales.

RequireJS allows you to set up a basic module that has localized information without forcing you to provide all locale-specific information up front. It can be added over time, and only strings/values that change between locales can be defined in the locale-specific file.

i18n bundle support is provided by the i18n.js plugin. It is automatically loaded when a module or dependency specifies the i18n! prefix (more info below). Download the plugin and put it in the same directory as your app's main JS file.

To define a bundle, put it in a directory called "nls" -- the i18n! plugin assumes a module name with "nls" in it indicates an i18n bundle. The "nls" marker in the name tells the i18n plugin where to expect the locale directories (they should be immediate children of the nls directory). If you wanted to provide a bundle of color names in your "my" set of modules, create the directory structure like so:

- *my/nls/colors.js*

The contents of that file should look like so:

```
//my/nls/colors.js contents:
define({
    "root": {
        "red": "red",
        "blue": "blue",
        "green": "green"
    }
});
```

An object literal with a property of "root" defines this module. That is all you have to do to set the stage for later localization work.

You can then use the above module in another module, say, in a my/lamps.js file:

```
//Contents of my/lamps.js
define(["i18n!my/nls/colors"], function(colors) {
    return {
        testMessage: "The name for red in this locale is: " + colors.red
    }
});
```

The my/lamps module has one property called "testMessage" that uses colors.red to show the localized value for the color red.

Later, when you want to add a specific translation to a file, say for the fr-fr locale, change my/nls/colors to look like so:

```
//Contents of my/nls/colors.js
define({
    "root": {
        "red": "red",
        "blue": "blue",
        "green": "green"
    },
    "fr-fr": true
});
```

Then define a file at my/nls/fr-fr/colors.js that has the following contents:

```
//Contents of my/nls/fr-fr/colors.js
define({
    "red": "rouge",
    "blue": "bleu",
    "green": "vert"
});
```

RequireJS will use the browser's navigator.languages, navigator.language or navigator.userLanguage property to determine what locale values to use for my/nls/colors, so your app does not have to change. If you prefer to set the locale, you can use the module config to pass the locale to the plugin:

```
requirejs.config({
    config: {
        //Set the config for the i18n
        //module ID
        i18n: {
            locale: 'fr-fr'
        }
    }
});
```

Note that RequireJS will always use a lowercase version of the locale, to avoid case issues, so all of the directories and files on disk for i18n bundles should use lowercase locales.

RequireJS is also smart enough to pick the right locale bundle, the one that most closely matches the ones provided by my/nls/colors. For instance, if the locale is "en-us", then the "root" bundle will be used. If the locale is "fr-fr-paris" then the "fr-fr" bundle will be used.

RequireJS also combines bundles together, so for instance, if the french bundle was defined like so (omitting a value for red):

```
//Contents of my/nls/fr-fr/colors.js
define({
    "blue": "bleu",
    "green": "vert"
});
```

Then the value for red in "root" will be used. This works for all locale pieces. If all the bundles listed below were defined, then RequireJS will use the values in the following priority order (the one at the top takes the most precedence):

- *my/nls/fr-fr-paris/colors.js*
- *my/nls/fr-fr/colors.js*
- *my/nls/fr/colors.js*
- *my/nls/colors.js*

If you prefer to not include the root bundle in the top level module, you can define it like a normal locale bundle. In that case, the top level module would look like:

```
//my/nls/colors.js contents:
define({
    "root": true,
    "fr-fr": true,
    "fr-fr-paris": true
});
```

and the root bundle would look like:

```
//Contents of my/nls/root/colors.js
define({
    "red": "red",
    "blue": "blue",
    "green": "green"
});
```