Amd

Getting Started with AMD

The **Asynchronous Module Definition** API specifies a mechanism for defining JavaScript modules such that the module and its dependencies can be asynchronously loaded. This is particularly well suited for the browser environment where synchronous loading of modules incurs performance, usability, debugging, and cross-domain access problems.

Basic concept

```
// The basic AMD API consists of nothing but two methods: `define` and `require`
// and is all about module definition and consumption:
// `define(id?, dependencies?, factory)` defines a module
// `require(dependencies, callback)` imports a set of dependencies and
// consumes them in the passed callback

// Let's start by using define to define a new named module
// that has no dependencies. We'll do so by passing a name
// and a factory function to define:
define('awesomeAMD', function(){
  var isAMDAwesome = function(){
    return true;
  };
  // The return value of a module's factory function is
  // what other modules or require calls will receive when
  // requiring our `awesomeAMD` module.
  // The exported value can be anything, (constructor) functions,
  // objects, primitives, even undefined (although that won't help too much).
  return isAMDAwesome;
});

// Now, let's define another module that depends upon our `awesomeAMD` module.
// Notice that there's an additional argument defining our
// module's dependencies now:
define('loudmouth', ['awesomeAMD'], function(awesomeAMD){
  // dependencies will be passed to the factory's arguments
  // in the order they are specified
  var tellEveryone = function(){
    if (awesomeAMD()){
      alert('This is sOoOo rad!');
    } else {
      alert('Pretty dull, isn\'t it?');
    }
  };
  return tellEveryone;
});

// As we do know how to use define now, let's use `require` to
// kick off our program. `require`'s signature is `(arrayOfDependencies, callback)`.
require(['loudmouth'], function(loudmouth){
  loudmouth();
});
```

```
// To make this tutorial run code, let's implement a very basic
// (non-asynchronous) version of AMD right here on the spot:
function define(name, deps, factory){
  // notice how modules without dependencies are handled
  define[name] = require(factory ? deps : [], factory || deps);
}

function require(deps, callback){
  var args = [];
  // first let's retrieve all the dependencies needed
  // by the require call
  for (var i = 0; i < deps.length; i++){
    args[i] = define[deps[i]];
  }
  // satisfy all the callback's dependencies
  return callback.apply(null, args);
}
// you can see this code in action here: http://jsfiddle.net/qap949pd/
```

Real-world usage with require.js

In contrast to the introductory example, `require.js` (the most popular AMD library) actually implements the **A** in **AMD**, enabling you to load modules and their dependencies asynchronously via XHR:

```
/* file: app/main.js */
require(['modules/someClass'], function(SomeClass){
  // the callback is deferred until the dependency is loaded
  var thing = new SomeClass();
});
console.log('So here we are, waiting!'); // this will run first
```

By convention, you usually store one module in one file. `require.js` can resolve module names based on file paths, so you don't have to name your modules, but can simply reference them using their location. In the example `someClass` is assumed to be in the `modules` folder, relative to your configuration's `baseUrl`:

- app/
- main.js
- modules/
  - someClass.js
  - someHelpers.js
  - ...
- daos/
  - things.js
  - ...

This means we can define `someClass` without specifying a module id:

```
/* file: app/modules/someClass.js */
define(['daos/things', 'modules/someHelpers'], function(thingsDao, helpers){
  // module definition, of course, will also happen asynchronously
  function SomeClass(){
    this.method = function(){/**/};
    // ...
  }
```

```
  return SomeClass;
});
```

To alter the default path mapping behavior use `requirejs.config(configObj)` in your `main.js`:

```
/* file: main.js */
requirejs.config({
  baseUrl : 'app',
  paths : {
    // you can also load modules from other locations
    jquery : '//ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min',
    coolLibFromBower : '../bower_components/cool-lib/coollib'
  }
});
require(['jquery', 'coolLibFromBower', 'modules/someHelpers'], function($, coolLib, helpers){
  // a `main` file needs to call require at least once,
  // otherwise no code will ever run
  coolLib.doFancyStuffWith(helpers.transform($('#foo')));
});
```

`require.js`-based apps will usually have a single entry point (`main.js`) that is passed to the `require.js` script tag as a data-attribute. It will be automatically loaded and executed on pageload:

```
<!DOCTYPE html>
<html>
<head>
  <title>A hundred script tags? Never again!</title>
</head>
<body>
  <script src="require.js" data-main="app/main"></script>
</body>
</html>
```

Optimizing a whole project using r.js

Many people prefer using AMD for sane code organization during development, but still want to ship a single script file in production instead of performing hundreds of XHRs on page load.

`require.js` comes with a script called `r.js` (that you will probably run in node.js, although Rhino is supported too) that can analyse your project's dependency graph, and build a single file containing all your modules (properly named), minified and ready for consumption.

Install it using `npm`:

```
$ npm install requirejs -g
```

Now you can feed it with a configuration file:

```
$ r.js -o app.build.js
```

For our above example the configuration might look like:

```
/* file : app.build.js */
({
  name : 'main', // name of the entry point
  out : 'main-built.js', // name of the file to write the output to
  baseUrl : 'app',
  paths : {
    // `empty:` tells r.js that this should still be loaded from the CDN, using
    // the location specified in `main.js`
```

```
    jquery : 'empty:',
    coolLibFromBower : '../bower_components/cool-lib/coollib'
  }
})
```

To use the built file in production, simply swap `data-main`:

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

An incredibly detailed overview of build options is available in the GitHub repo.

Topics not covered in this tutorial

- Loader plugins / transforms
- CommonJS style loading and exporting
- Advanced configuration
- Shim configuration (loading non-AMD modules)
- CSS loading and optimizing with require.js
- Using almond.js for builds

Further reading:

- Official Spec
- Why AMD?
- Universal Module Definition

Implementations:

- require.js
- dojo toolkit
- cujo.js
- curl.js
- lsjs
- mmd