Organize Your Code with RequireJS

By Jim Hoskins @jimrhoskins
27 September 2011 | Category: JavaScript

Writing web applications using JavaScript, HTML, and CSS can become overwhelming quickly. Between managing UI, data, interactions, and network requests, application code can become a real mess, and this isn't helped by the way the browser loads JavaScript code.

In most other programming environments, there is some way to split your code into modules or scripts, and specifically require one module from another.

In JavaScript, we can split our code into multiple files, but there is no way to declare in one script that you are depending on another. Worse yet, the place where scripts are loaded isn't even in the same language you are writing your code. You load all your JavaScript dependencies from your HTML document, not your JavaScript files. That may work for including a little Ajax here, and an animation there, but for large single page applications, this does not cut it.

I have had the opportunity to work on a large single page application for Treehouse, and tried using RequireJS, and I love it.

How does RequireJS make building an application easier?

Module Definition

All of your code is written in self contained modules. Modules are small chunks of your application that serve a specific purpose. How much code you define in a module is left up to you. You could write your whole app in one module, but that would be defeating the organizational benefit, and the ability to extract or replace portions of your app easily.

A typical simple module is defined like this:

```
define(function () {

  function method (x) {
    return x + x;
  }

  return {
    someValue: 'foobar',
    myMethod: method
  }
});
```

Here we use the `define()` function to define our function. In this example we passed in a function which will return our module. In this case our module is an object with two properties: a string `someValue` and a function `myMethod`.

This code would make up the entirety of our JavaScript file for this module. Nothing should be declared outside of a single `define` call.

If this code was saved in `my/utils.js`, this module would be defined as the module “my/utils”.

Module Dependencies.

Defining modules is great, but we are going to need to use parts of modules from other modules. When we define a module, we can pass a list of module names, and RequireJS will make sure to retrieve these modules before your module is executed, and it will pass those modules as parameters of the definition function.

Given we still have our “my/utils” module, and another module “models/Person”, which defines a single JavaScript class Person, here is how we might define a module that requires both of them.

```
define(["models/Person", "my/utils"], function (Person, utils)
  var people = [];

  people.push(new Person('Jim'));
  people.push(new Person(utils.someValue));


  return {people: people};
});
```

Here we defined a module that we will say lives in "app/people.js". It required two modules, "models/Person", and "my/utils". These modules were then passed to our module definition function, and we bound them to the parameters Person and utils. We could call the parameters anything at all, but I chose to reflect the module names.

In this example, I capitalized the Person module, since in this hypothetical module it is returning a JavaScript class, not a normal object with properties, so I used a capitalized name as a convention.

The module then defined a people array, added a couple of Persons to it, and exposed it as a property of its own module.

RequireJS looks at the dependencies of all the modules in your application, and will fetch and execute all of the modules in the proper order so that each module has exactly what it needs to run.

Running the App

Now we have our modules defined, we want to use them to actually start an application. This will usually be done in some sort of "main" file, which isn't really a module.

Instead of using `define()` to wrap our code, in our main file we use `require()`. The `require()` function is similar, in that you pass it an optional array of dependencies, and a function which will be executed when those dependencies are resolved. However this code is not stored as a named module, as its purpose is to be run immediately, much like the main() function in a C program.

So let us say our main file is in "/scripts/main.js", and let's also say that our modules were also defined under "/scripts/". So really "my/utils" lives in "/scripts/my/utils.js", and "models/Person" lives in "/scripts/models/Person.js", and so on. We will see how this is important later. But our main.js might look like this:

```
require(["app/people"], function(people) {
  alert(people.people[0]);
});
```

Here, we just required the “app/people module”, and bound it to people. The people module exposed a people property, which we alerted the first element of.

So now we need to include the JavaScript in our HTML page. We do it all with one script tag.

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

Our actual file structure looks like this

- root/
  - index.html
  - scripts/
  - require.js
  - main.js
  - app/
    - people.js
  - models/
    - Person.js
  - my/
    - utils.js

Our script tag is including via the src attribute the require.js script from the scripts directory. We also defined an attribute calldеd data-main, with the name of our main application module “scripts/main”. This means RequireJS will load up “/scripts/main.js” as the main entry point.

It also sets the root directory for modules to the “/scripts/” directory. This is why we can call a module “my/utils” when it really lives in “/scripts/my/utils.js”. You can require a module from an absolute location by giving its name using a leading “./” or “/” or a full URL. Otherwise it assumes the module lives relative to the main script.

Optimizing

While having code split into multiple files is great for organization, it’s terrible for performance. That is why RequireJS includes an optimizer which will take the modules of your app, move them into a single file, and minify them, to improve performance.

Bonus: RequireJS + Dojo = Awesome!

My application uses Dojo, and the entire UI is built programatically using custom Dojo Widgets. It turns out Dojo's loader is compatible with RequireJS, if you set your application up right. This made my application so much easier to develop and maintain.

Also, using CoffeeScript to extend Dojo Widgets is also amazing, but that's another post!

Follow @thinkvitamin on Twitter

Please check out Think Vitamin Membership

Other Posts You Might Find Interesting

- Sorry - No Related Posts Found

Comments

Like 3 people liked this.

Add New Comment

Optional: Login below.

Post as ...

Showing 3 comments

Sort by Best rating Subscribe by email Subscribe by RSS flash template 3 days ago

This article was extremely interesting, especially since I was searching for thoughts on this subject last week.

Like Reply

Mark McDonnell 3 days ago

Thanks for the article, I found this really useful as I had looked at RequireJS before and was confused as to how I could use it (actually, I think I looked at it form the perspective of using Dojo) - but your post has helped clarify for me and I'm definitely going to try it again :)

One issue though is the tabbing on your sub li elements have gone a bit funky. At the moment it looks like the /scripts/ directory has no content and that the main.js and require.js are placed outside of that directory (it was a little confusing at first and so I had to go over it a couple of times to make sure that was supposed to be like that).

Oh, and one last thing, maybe consider re-writing the sentence "Our script tag is including via the src attribute the require.js script from the scripts directory." as that too sounded a bit confusing. Maybe something like...

"Our script tag specifies that we load require.js from the /scripts/ directory."

Otherwise thanks for the sharing this info.

Like Reply

derekgreer 4 days ago

Awesome!

Like Reply