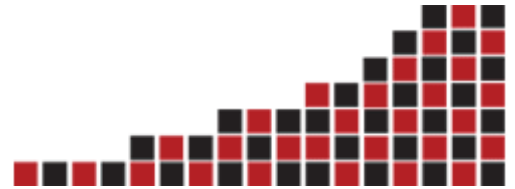


[Advertise Here](#)

DON'T BE A VICTIM  
OF YOUR OWN SITE GROWTH.



# Building Large, Maintainable, and Testable Knockout.js Applications

[Jonathan Creamer](#) on Apr 3rd 2013 with [7 Comments](#)

## Tutorial Details

- Topic: JavaScript
- Difficulty: Medium
- Estimated Completion Time: 1 hour

[Knockout.js](#) is a popular open source (MIT) [MVVM](#) JavaScript framework, created by [Steve Sandersen](#). Its website provides great information and demos on how to build simple applications, but it unfortunately doesn't do so for larger applications. Let's fill in some of those gaps!

---

## AMD and Require.js

AMD is a JavaScript module format, and one of the most popular (if not the most)

frameworks is <http://requirejs.org> by <https://twitter.com/jrburke>. It consists of two global functions called `require()` and `define()`, although `require.js` also incorporates a starting JavaScript file, such as `main.js`.

```
1 | <script src="js/require-jquery.min.js" data-main="js/main"></script
```

There are primarily two flavors of `require.js`: a vanilla `require.js` file and one that includes jQuery (`require-jquery`). Naturally, the latter is used predominately in jQuery-enabled websites. After adding one of these files to your page, you can then add the following code to your `main.js` file:

```
1 | require( [ "https://twitter.com/jrburkeapp" ], function( App ) {  
2 |     App.init();  
3 | })
```

The `require()` function is typically used in the `main.js` file, but you can use it to directly include a module anywhere. It accepts two arguments: a list of dependencies and a callback function.

The callback function executes when all dependencies finish loading, and the arguments passed to the callback function are the objects **required** in the aforementioned array.

It's important to note that the dependencies load asynchronously. Not all libraries are AMD compliant, but `require.js` provides a mechanism to shim those types of libraries so that they can be loaded.

This code requires a module called `app`, which could look like the following:

```
1 | define( [ "jquery", "ko" ], function( $, ko ) {  
2 |     var App = function(){};  
3 |  
4 |     App.prototype.init = function() {  
5 |         // INIT ALL TEH THINGS  
6 |     };  
7 |  
8 |     return new App();  
9 | });
```

The `define()` function's purpose is to define a **module**. It accepts three arguments: the name of the module (which is **typically** not included), a list of dependencies and

a callback function. The `define()` function allows you to separate an application into many modules, each having a specific function. This promotes decoupling and separation of concerns because each module has its own set of specific responsibilities.

## Using Knockout.js and Require.js Together

Knockout is AMD ready, and it defines itself as an anonymous module. You don't need to shim it; just include it in your paths. Most AMD-ready Knockout plugins list it as "knockout" rather than "ko", but you can use either value:

```
1  require.config({
2      paths: {
3          ko: "vendor/knockout-min",
4          postal: "vendor/postal",
5          underscore: "vendor/underscore-min",
6          amplify: "vendor/amplify"
7      },
8      shim: {
9          underscore: {
10             exports: "_"
11          },
12          amplify: {
13             exports: "amplify"
14          }
15      },
16      baseUrl: "/js"
17  });
```

This code goes at the top of `main.js`. The `paths` option defines a map of common modules that load with a key name as opposed to using the entire file name.

The `shim` option uses a key defined in `paths` and can have two special keys called `exports` and `deps`. The `exports` key defines what the shimmed module returns, and `deps` defines other modules that the shimmed module might depend on. For example, [jQuery Validate](#)'s shim might look like the following:

```
1  shim: {
2      // ...
3      "jquery-validate": {
4          deps: [ "jquery" ]
5      }
6  }
```

# Single- vs Multi-Page Apps

It's common to include all the necessary JavaScript in a single page application. So, you may define the configuration and the initial require of a single-page application in `main.js` like so:

```
1  require.config({
2      paths: {
3          ko: "vendor/knockout-min",
4          postal: "vendor/postal",
5          underscore: "vendor/underscore-min",
6          amplify: "vendor/amplify"
7      },
8      shim: {
9          ko: {
10             exports: "ko"
11          },
12          underscore: {
13             exports: "_"
14          },
15          amplify: {
16             exports: "amplify"
17          }
18      },
19      baseUrl: "/js"
20  });
21
22  require( [ "https://twitter.com/jrburkeapp" ], function( App ) {
23      App.init();
24  })
```

You might also need separate pages that not only have page-specific modules, but share a common set of modules. James Burke has [two repositories](#) that implement this type of behavior.

The rest of this article assumes you're building a multi-page application. I'll rename `main.js` to `common.js` and include the necessary `require.config` in the above example in the file. This is purely for semantics.

Now I'll require `common.js` in my files, like this:

```
1  <script src="js/require-jquery.js"></script>
2  <script>
3      require( [ "../js/common" ], function () {
4          //js/common sets the baseUrl to be js/ so
5          //can just ask for 'app/main1' here instead
```

```
6      //of 'js/app/main1'
7      require( [ "pages/index" ] );
8    });
9  </script>
10 </body>
11 </html>
```

The `require.config` function will execute, requiring the main file for the specific page.

The `pages/index` main file might look like the following:

```
1  require( [ "app", "postal", "ko", "viewModels/indexViewModel" ], fu
2    window.app = app;
3    window.postal = postal;
4
5    ko.applyBindings( new IndexViewModel() );
6  });
```

This `page/index` module is now responsible for loading all the necessary code for the `index.html` page. You can add other main files to the `pages` directory that are also responsible for loading their dependent modules. This allows you to break multi-page apps into smaller pieces, while avoiding unnecessary script inclusions (e.g. including the JavaScript for `index.html` in the `about.html` page).

---

## Sample Application

Let's write a sample application using this approach. It'll display a searchable list of beer brands and let us choose your favorites by clicking on their names. Here is the app's folder structure:

Let's first look at `index.html`'s HTML markup:

```
1  <section id="main">
2    <section id="container">
3      <form class="search" data-bind="submit: doSearch">
4        <input type="text" name="search" placeholder="Search"
5        <ul data-bind="foreach: beerListFiltered">
6          <li data-bind="text: name, click: $parent.addToFav
7        </ul>
8      </form>
9
10   <aside id="favorites">
```

```
11         <h3>Favorites</h3>
12         <ul data-bind="foreach: favorites">
13             <li data-bind="text: name, click: $parent.removeFr
14         </ul>
15     </aside>
16 </section>
17 </section>
18
19 <!-- import("templates/list.html") -->
20
21 <script src="js/require-jquery.js"></script>
22 <script>
23     require( [ "./js/common" ], function (common) {
24         //js/common sets the baseUrl to be js/ so
25         //can just ask for 'app/main1' here instead
26         //of 'js/app/main1'
27         require( [ "pages/index" ] );
28     });
29 </script>
```

## Pages

The structure of our application uses multiple “pages” or “mains” in a pages directory. These separate pages are responsible for initializing each page in the application.

The **ViewModels** are responsible for setting up the Knockout bindings.

## ViewModels

The `ViewModels` folder is where the main Knockout.js application logic lives. For example, the `IndexViewModel` looks like the following:

```
1 // https://github.com/jcreamer898/NetTutsKnockout/blob/master/lib/
2 define( [
3     "ko",
4     "underscore",
5     "postal",
6     "models/beer",
7     "models/baseViewModel",
8     "shared/bus" ], function ( ko, _, postal, Beer, BaseViewModel,
9
10     var IndexViewModel = function() {
11         this.beers = [];
12         this.search = "";
```

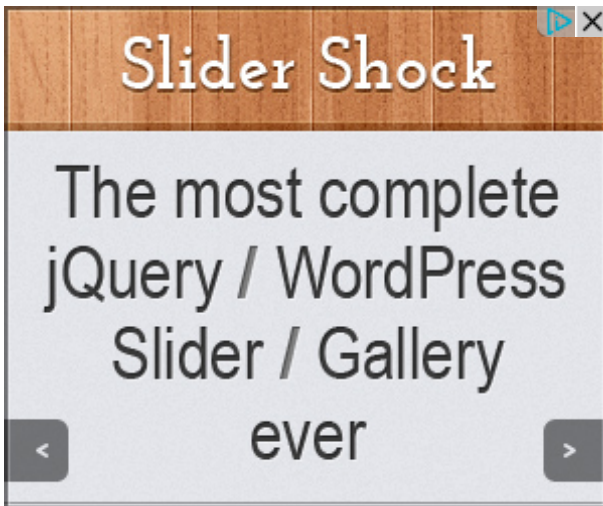
```
13
14     BaseViewModel.apply( this, arguments );
15 };
16
17 _ .extend(IndexViewModel.prototype, BaseViewModel.prototype, {
18     initialize: function() { // ... },
19
20     filterBeers: function() { /* ... */ },
21
22     parse: function( beers ) { /* ... */ },
23
24     setupSubscriptions: function() { /* ... */ },
25
26     addToFavorites: function() { /* ... */ },
27
28     removeFromFavorites: function() { /* ... */ }
29 });
30
31 return IndexViewModel;
32 });
```

The `IndexViewModel` defines a few basic dependencies at the top of the file, and it inherits `BaseViewModel` to initialize its members as knockout.js observable objects (we'll discuss that shortly).

Next, rather than defining all of the various `ViewModel` functions as instance members, underscore.js's `extend()` function extends the prototype of the `IndexViewModel` data type.

## Inheritance and a BaseModel

Inheritance is a form of code reuse, allowing you to reuse functionality between similar types of objects instead of rewriting that functionality. So, it's useful to define a base model that other models or can inherit from. In our case, our base model is `BaseViewModel`:



```

1  var BaseViewModel = function( options ) {
2      this._setup( options );
3
4      this.initialize.call( this, options );
5  };
6
7  _.extend( BaseViewModel.prototype, {
8      initialize: function() {},
9
10     _setup: function( options ) {
11         var prop;
12
13         options = options || {};
14
15         for( prop in this ) {
16             if ( this.hasOwnProperty( prop ) ) {
17                 if ( options[ prop ] ) {
18                     this[ prop ] = _.isArray( options[ prop ] ) ?
19                         ko.observableArray( options[ prop ] ) :
20                         ko.observable( options[ prop ] );
21                 }
22                 else {
23                     this[ prop ] = _.isArray( this[ prop ] ) ?
24                         ko.observableArray( this[ prop ] ) :
25                         ko.observable( this[ prop ] );
26                 }
27             }
28         }
29     }
30 });
31
32 return BaseViewModel;

```

The `BaseViewModel` type defines two methods on its prototype. The first is `initialize()`, which should be overridden in the subtypes. The second is `_setup()`, which sets up



the object for data binding.

The `_setup` method loops over the properties of the object. If the property is an array, it sets the property as an `observableArray`. Anything other than an array is made `observable`. It also checks for any of the properties' initial values, using them as default values if necessary. This is one small abstraction that eliminates having to constantly repeat the `observable` and `observableArray` functions.

## The “this” Problem

People who use Knockout tend to prefer instance members over prototype members because of the issues with maintaining the proper value of `this`. The `this` keyword is a complicated feature of JavaScript, but it's not so bad once fully grokked.

From the MDN:

“In general, the object bound to `this` in the current scope is determined by how the current function was called, it can't be set by assignment during execution, and it can be different each time the function is called.”

So, the scope changes depending on HOW a function is called. This is clearly evidenced in jQuery:

```
1 | var $el = $( "#mySuperButton" );
2 | $el.on( "click", function() {
3 |     // in here, this refers to the button
4 | });
```

This code sets up a simple `click` event handler on an element. The callback is an anonymous function, and it doesn't do anything until someone clicks on the element. When that happens, the scope of `this` inside of the function refers to the actual DOM element. Keeping that in mind, consider the following example:

```
1 | var someCallbacks = {
2 |     someVariable: "yay I was clicked",
3 |     mySuperButtonClicked: function() {
4 |         console.log( this.someVariable );
5 |     }
6 | };
```

```
7 |  
8 | var $el = $( "#mySuperButton" );  
9 | $el.on( "click", someCallbacks.mySuperButtonClicked );
```


There's an issue here. The `this.someVariable` used inside `mySuperButtonClicked()` returns undefined because `this` in the callback refers to the DOM element rather than the `someCallbacks` object.

There are two ways to avoid this problem. The first uses an anonymous function as the event handler, which in turn calls `someCallbacks.mySuperButtonClicked()`:

```
1 | $el.on( "click", function() {  
2 |     someCallbacks.mySuperButtonClicked.apply();  
3 | });
```

The second solution uses either the `Function.bind()` or `_.bind()` methods (`Function.bind()` is not available in older browsers). For example:

```
1 | $el.on( "click", _.bind( someCallbacks.mySuperButtonClicked, someCa
```

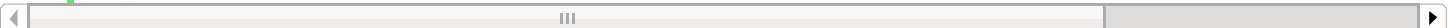


Either solution you choose will achieve the same end-result: `mySuperButtonClicked()` executes within the context of `someCallbacks`.

## “this” in Bindings and Unit Tests

In terms of Knockout, the `this` problem can show itself when working with bindings—particularly when dealing with `$root` and `$parent`. Ryan Niemeyer wrote a [delegated events plugin](#) that mostly eliminates this issue. It gives you several options for specifying functions, but you can use the `data-click` attribute, and the plugin walks up your scope chain and calls the function with the correct `this`.

```
1 | <form class="search">  
2 |   <input type="text" name="search" placeholder="Search" data-bind  
3 |     <ul data-bind="foreach: beerListFiltered">  
4 |       <li data-bind="text: name, click: $parent.addToFavorites"><  
5 |     </ul>  
6 | </form>
```



In this example, `$parent.addToFavorites` binds to the view model via a `click` binding. Since the `<li />` element resides inside a `foreach` binding, the `this` inside

`$parent.addToFavorites` refers to an instance of a the beer that was clicked on.

To get around this, the `_.bindAll` method ensures that `this` maintains its value.

Therefore, adding the following to the `initialize()` method fixes the problem:

```
1  _.extend(IndexViewModel.prototype, BaseViewModel.prototype, {
2      initialize: function() {
3          this.setupSubscriptions();
4
5          this.beerListFiltered = ko.computed( this.filterBeers, this
6
7      },
8      _._bindAll( this, "addToFavorites" );
9  });
```

The `_.bindAll()` method essentially creates an instance member called `addToFavorites()` on the `IndexViewModel` object. This new member contains the prototype version of `addToFavorites()` that is bound to the `IndexViewModel` object.

The `this` problem is why some functions, such as `ko.computed()`, accepts an optional second argument. See line five for an example. The `this` passed as the second argument ensures that `this` correctly refers to the current `IndexViewModel` object inside of `filterBeers`.

How would we test this code? Let's first look at the `addToFavorites()` function:

```
1  addToFavorites: function( beer ) {
2      if( !_._any( this.favorites(), function( b ) { return b.id() ===
3          this.favorites.push( beer );
4      }
5  }
```

If we use the [mocha](#) testing framework and [expect.js](#) for assertions, our unit test would look like the following:

```
1  it( "should add new beers to favorites", function() {
2      expect( this.viewModel.favorites().length ).to.be( 0 );
3
4      this.viewModel.addToFavorites( new Beer({
5          name: "abita amber",
6          id: 3
7      }));
8  });
```

```
9      // can't add beer with a duplicate id
10     this.viewModel.addToFavorites( new Beer({
11         name: "abita amber",
12         id: 3
13     }));
14
15     expect( this.viewModel.favorites().length ).to.be( 1 );
16 });
```

To see the full unit testing setup, [check out the repository](#).

Let's now test `filterBeers()`. First, let's look at its code:

```
1  filterBeers: function() {
2      var filter = this.search().toLowerCase();
3
4      if ( !filter ) {
5          return this.beers();
6      }
7      else {
8          return ko.utils.arrayFilter( this.beers(), function( item
9              return ~item.name().toLowerCase().indexOf( filter );
10          });
11      }
12 },
```

This function uses the `search()` method, which is databound to the value of a text `<input />` element in the DOM. Then it uses the `ko.utils.arrayFilter` utility to search through and find matches from the list of beers. The `beerListFiltered` is bound to the `<ul />` element in the markup, so the list of beers can be filtered by simply typing in the text box.

The `filterBeers` function, being such a small unit of code, can be properly unit tested:

```
1  beforeEach(function() {
2      this.viewModel = new IndexViewModel();
3
4      this.viewModel.beers.push(new Beer({
5          name: "budweiser",
6          id: 1
7      }));
8      this.viewModel.beers.push(new Beer({
9          name: "amberbock",
10         id: 2
11     }));
12 });
13
```

```
14  it( "should filter a list of beers", function() {  
15      expect( _.isFunction( this.viewModel.beerListFiltered ) ).to.b  
16  
17      this.viewModel.search( "bud" );  
18  
19      expect( this.viewModel.filterBeers().length ).to.be( 1 );  
20  
21      this.viewModel.search( "" );  
22  
23      expect( this.viewModel.filterBeers().length ).to.be( 2 );  
24  });
```

First, this test makes sure that the `beerListFiltered` is in fact a function. Then a query is made by passing the value of “bud” to `this.viewModel.search()`. This should cause the list of beers to change, filtering out every beer that does not match “bud”. Then, `search` is set to an empty string to ensure that `beerListFiltered` returns the full list.

---

## Conclusion

Knockout.js offers many great features. When building large applications, it helps to adopt many of the principles discussed in this article to help your app’s code remain manageable, testable, and maintainable. Check out the [full sample application](#), which includes a few extra topics such as messaging. It uses [postal.js](#) as a message bus to carry messages throughout the application. Using messaging in a JavaScript application can help decouple parts of the application by removing hard references to each other. Be sure and take a look!

Like

78 people like this. Be the first of your friends.



Tags: [databindingknockout](#)

### By **Jonathan Creamer**

Currently a JavaScript Engineer at [appendTo](#). Lover of all things Web, but mostly ASP.NET MVC, JavaScript, jQuery, and C#. I believe that you cannot ever stop learning which is why I stay active in the .NET world attending User Groups, blogging for [FreshBrewedCode](#), <http://jcreamerlive.com> and scouring Twitter via [@jcreamer898](#) and the interwebs for as much knowledge I can squeeze into my brain. I work at AppendTo and am having a great time developing front end applications in jQuery and JavaScript. Please feel free to contact me, I love meeting other devs who are passionate about what they do...

**Note:** Want to add some source code? Type `<pre><code>` before it and `</code></pre>` after it. [Find out more](#)