Deloitte.

Federal .NET CoP
NEWSLETTER

Fall 2013

A Tech's Side of Things

Have your own tips? Share them with us and get the chance to be included in our next edition of A Tech's Side of Things!

Are You Managing JS Dependencies like an Old Man?

by Sean Gilbert with technical contributions from
Ben Elder and **Orn Kristjansson**

Over the past decade JavaScript standards and best practices have improved but dealing with dependencies and modularity is difficult. Frameworks like **RequireJS** (**and others**) remedy these issues and more. Don't be mistaken, there are teeth and merit to this new shiny toy. Sit down, buckle up, and keep all hands and arms inside as we teleport through the JavaScript time machine!

Travel back to year 2000. JavaScript's main existence was validation. There was no notion of namespaces and all code lived in global space. After the dark ages, during the year "Revenge of the Sith" was the #1 movie and Single Page Application (SPA) was coined by Steve Yen, libraries like **jQuery**, **Mootools** and **Prototype.js** gained traction. Other best practices took root around **closures**, Immediately-Invoked Function Express (IIFE **1**, **2**), and **namespaces**. Presently, almost everyone knows best practice (**learn** & **Pluralsight**):

- ✓ No augmenting built-in native objects
- ✓ Use namespaces to prevent polluting global namespace
- ✓ Leverage IIFE
- ✓ Use **JavaScript Design Patterns** - Module, Revealing Module, MVC, & MVVM.

Using namespaces isolates code, creating a sandbox of sorts to avoid name collisions and better organized code. The drawback to namespaces is verbosity and namespace cluttering. As applications grow deep nested namespaces *(DeloitteCore.Common.ServiceProxy.doAjaxPost(method, ..))* are common, sparking an urge to put all methods into a single object. All methods on a single object are a red flag, name collisions will increase. Even using noConflict() is error prone. Race back to current day, 2013, the year of "Iron Man 3" and release of VS 2013 forget all of this mess, instead work with a module system.

A viable option to support modularity in JavaScript is to leverage namespaces and IIFE. However, the dependency problem remains and JavaScript files must be referenced in proper order which is cumbersome and error prone. A better approach, leverage **RequireJS**, it implements clean, maintainable modern modular JavaScript patterns like **AMD** and **CommonJS**. In RequireJS code is separated into reusable modules targeting a single concern. Implement **SOLID** design principles and decouple dependencies in your SPA or ASP.NET MVC JavaScript for FREE using the modularity of RequireJS. RequireJS can handle custom and third party dependencies no matter the nesting using configuration and shims. One could argue RequireJS provides basic form of **dependency injection (DI)** or **Service Location**. Say goodbye to deep nested namespaces, complex ordering of scripts and version difficulty. Instead, simply load require.js, add an entry

point using main.js, add configuration for custom and/or third party scripts then begin using the require or define to asynchronously load scripts and modules. Implementing in a SPA is straight forward as seen below. What about multi-page applications? Before you say this won't work in a traditional multi-page applications like ASP.NET MVC or even TypeScript, hold your tongue. While this is out of scope for this article please visit the following references **ASP.NET MVC 4 and RequireJS**, **Using RequireJS in an APS.NET MVC application**, **Modular JavaScript Using RequireJS** or reach out to me via **mailto:segilbert@deloitte.com**.

Load require.js inside index.html, and then tell it where to the main entry point of the application is, in this case main.js.

```
<script src="~/Scripts/require.js" data-main="App/main.js"></script>
```

Load script files based on root path (configurable); load utils.js and model.js then reference them in the function using utils and model.

```
require([
    "scripts/app/utils",
    "scripts/app/model"
], function (utils, model) {

    // the app is loaded...

});
```

Define modular dependencies for the function. RequireJS will inject system.js, logger.js, router.js, and config.js relative to root as dependencies to the function.

```
define(['durandal/system', 'services/logger', 'durandal/plugins/router', 'config'],
    function(system, logger, router, config) {
        var shell = {
            activate: activate,
            router: router
        };
        return shell;

        function activate() {
            logger.log('CodeCamper JumpStart Loaded!',
                null,
                system.getModuleId(shell),
                true);

            router.map(config.routes);
            return router.activate(config.startModule);
        }

    }
);
```

To learn more I recommend reading **Require JS Fundamentals**, **writing modular JavaScript** and reviewing **RequireJS** documentation.

Pluralsight Courses

1. **JavaScript Design Patterns**
2. **Structuring JavaScript**

Related Links

- Visit the **Federal .NET CoP Website** to access the latest community news and content.
- Visit the global **Microsoft .NET CoP.**
- Join the **Federal .NET Yammer Group** to connect with your fellow practitioners in the .NET space.

Contact Us

In the meantime, share your thoughts on the Newsletter and **take our survey**! And remember, this Newsletter will be made available to practitioners on the **Federal .NET CoP Website** and **Yammer Page**.

If you'd like to get involved with the community, request more information, or send a question or comments, please contact us at **FedNETCoP@deloitte.com**.

1633 Broadway
New York, NY 10019-6754
United States