Large Scale JavaScript on Client and Server

Module 2: Maintainable JavaScript

Shawn Wildermuth
Wilder Minds LLC
@shawnwildermuth

pluralsight
hardcore dev and IT training

Agenda

- **Maintainable JavaScript**
  - Application Frameworks
  - Avoiding the Global Scope
  - Strictness in JavaScript
  - Modular JavaScript
  - Dependency Management
  - Smart Asynchrony
  - Loose Coupling

APPLICATION FRAMEWORKS

Techniques for Maintainable JavaScript Work with Most Frameworks

- AngularJS
- Backbone
- EmberJS
- Knockout (et al.)
- Durandal
- EcmaScript 6 too

Application Frameworks can encourage more maintainable codebases, but it’s ultimately up to the developers to do the right thing.

Avoiding the Global Scope

Easy to Pollute

Too easy to create global variables

Collision

Risk overwriting existing variables

Monolithic

Encourages large blocks of code

HIDING FROM THE GLOBAL SCOPE

Use Function Scope to Avoid Global Objects

- Self-Executing Anonymous Functions (SEAF)
- Also called Self-Invoking Anonymous Functions (SIAF)
- And Immediately Invoked Function Expressions (IIFE)

SEAF in a Nutshell

```
(function () {
  // Your Code Here
})();
```

SEAF in a Nutshell

```
(function () {

  // Not leaked to the global scope
  var _cache = {};

})();
```

SEAF in a Nutshell

```
(function () {

  // Not leaked to the global scope
  var _cache = {};

  $(document).ready(function () {
    // Startup Code
    _.each(_cache, function (i) {
      // Work with Collection
    })
  });

})();
```

Closure ensures you can still use Non-global objects

SEAF in a Nutshell

```
(function ($) {

  // Not leaked to the global scope
  var _cache = {};

  $(document).ready(function () {
    // Startup Code
    _.each(_cache, function (i) {
      // Work with Collection
    })
  });

})(jQuery);
```

Just function parameters

You can pass context to the SEAF to prevent Global object lookup

STRICTNESS IN JAVASCRIPT

Enforces the Best Parts of JavaScript

- **Throws Exceptions on Bad Practices**
- **Improves Code Quality**
- **Provides Early Detection of Problem Code**
- **Not a Replacement for JSLint**

Using Strictness in JavaScript

```
(function () {

  x = 0;      // works fine
  var y = "";
  y = 123;    // works too
  // et al.

})();
```

Using Strictness in JavaScript

```
(function () {

  "use strict"; // Backwards compatible

  x = 0;      // Nope
  var y = "";
  y = 123;    // Nope
  // et al.

})();
```

CITY E
BrickFest

mod-ule

/ˈmäjo͞ol/

Noun

1. each of a set of standardized parts or independent units that can be used to construct a more complex structure.

SINGLE
UNIT OF WORK

SMALL

REUSABLE

TESTABLE

LOOSELY COUPLED

DISCRETE

Rules for Modules

No DOM Manipulation Outside a Module	No Hard Coupling to Other Modules
No Accessing Global/Native Objects	No Global Declarations

Modular JavaScript (Module Pattern)

```
var destinationsModule = (function() {
  "use strict";

  var _cache = {};

  function _fillCache(callback) {
    // ...
  }

  return {
    fillCache: _fillCache,
    cache: _cache
  };
})();
```

Modular JavaScript (JavaScript Class)

```
function Animal() {
  "use strict";

  this.cache = {};

}

Animal.prototype.walk = function () {
  // ...
}
```

Class pattern is useful for non-singleton implementations

Modular AngularJS

```
var theModule = angular.module("indexPage", []);
```

Modular AngularJS

```
var theModule = angular.module("indexPage", []);

theModule.controller("animalController", [],
  function ($scope) {
    // ...
  });
```

Modular AngularJS

```
var theModule = angular.module("indexPage", []);

theModule.controller("animalController", [],
  function ($scope) {
    // ...
  });

theModule.factory("dataFactory", [],
  function () {
    var _myData = {};
    return {
      myData: _myData
    }
  });
```

AngularJS supports multiple types of modular code

Application Framework	Types of Modularity
Plain JavaScript	Namespaces, Module Pattern, Class Pattern
AngularJS	Modules, Services, Factories, Controllers, Directives, etc.
Backbone	Namespaces and Objects. Full modules with Backbone.Marionette
EmberJS	Extend built-in objects or use ES6/Plain JavaScript Modules
Durandal	Asynchronous Module Definition
EcmaScript 6	CommonJS Compatible

de·pen·den·cy

/diˈpendənsē/

Noun

1. The degree to which each program module relies on each one of the other modules.

DEPENDENCY MANAGEMENT

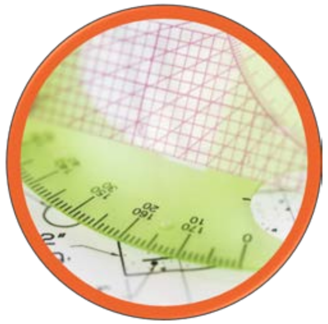
System of handling dependencies across an application

- Dependency Injection is typical pattern
  - Also called Inversion of Control
- Allows passing dependencies without global scope
- Cascading dependencies are simply handled

DEPENDENCY MANAGEMENT

REQUIREJS
(e.g. AMD)

COMMONJS



ANGULARJS

Asynchronous Module Definition (AMD)

```
// someModule.js - name of file implies module name
define([], function () {

  function _init() {

  }

  return {
    init: _init
  };

});
```

Optional other dependencies to be passed into this module

Asynchronous Module Definition (AMD)

```
// someModule.js - name of file implies module name
define("someModule", [], function () {

  function _init() {

  }

  return {
    init: _init
  };

});
```

Asynchronous Module Definition (AMD)

```
require(["someModule", "jQuery"],
  function (someModule, $) {
    // Use the dependencies
  });
```

Provides dependencies via the callback

Asynchronous Module Definition (AMD)

```
require(["./someModule/js", "jQuery"],
  function (someModule, $) {
    // Use the dependencies
  });
```

CommonJS Spec

```
var api = require('./api');
```

CommonJS Spec

```
exports.getCities = function (cb) {
  ...
};

exports.saveCities = function (cities, cb) {
  ...
};

exports.City = function (name) {
  this.name = name;
  ...
};
```

Must export the façade of the dependency

AngularJS

```
// create the module
var theModule = angular.module("indexPage", []);

theModule.factory("dataFactory", [],
  function () {
    var _myData = {};
    return {
      myData: _myData
    }
  });
```

AngularJS

```
// create the module
var theModule = angular.module("indexPage", []);

theModule.factory("dataFactory", [],
  function () {
    var _myData = {};
    return {
      myData: _myData
    }
  });

theModule.controller("controller", ["dataFactory"],
  function (dataFactory) {
    // ...
  });
```

Uses position of dependency to support minification

SMART ASYNCHRONY

Deeply nested callbacks are hard to maintain

- Should rely on existing or new patterns
  - Promises
  - Async Libraries

What’s Bad?

```
$(document).ready(function() {
  $.get("/api/destinations", function(result) {
    if (result.success) {
      if ($("#userName").length > 0) {
        $.get("/api/user/" + userid, function(result) {
          if (result.success) {
            ...
          }
        });
      }
    } else {
      alert("Failed to get destinations");
    }
  });
});
```

Promises

```
// using Q.js
someModule.makeAsyncCall()
  .then(function () { ... })
  .then(function () { ... })
  .fail(function () { ... })
  .finally(function () { ... })
  .done();
```

Async Library

```
// using Async
async.parallel([
  function(cb) {
    ...
    cb(1);
  },
  function(cb) {
    ...
    cb(2);
  }
],
function (err, results) {
  ...
  // results = [1,2]
});
```

LOOSE COUPLING

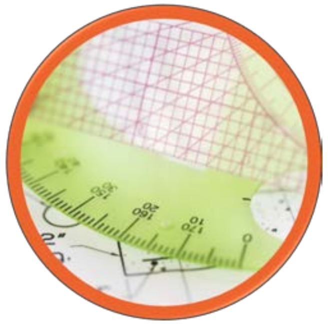
Don’t maintain hard links between Modules

- Avoid every module requiring a reference to others
  - Enables testing
- Messaging is key
  - Publish/Subscribe or Global Events

LOOSE COUPLING SOLUTIONS

JQUERY EVENTS

AMPLIFYJS



ANGULARJS

jQuery

```
// Publish Event
$.event.trigger("our.event.name", ["some", "context"]);
```

jQuery

```
// Publish Event
$.event.trigger("our.event.name", ["some", "context"]);


// Subscribe (requires DOM element)
$(document).on("our.event.name",
  function (event, some, context) {
    // ...
  });
```

AmplifyJS

```
// publish
amplify.publish("our.message.name", "some", "context");
```

AmplifyJS

```
// publish
amplify.publish("our.message.name", "some", "context");

// subscribe
amplify.subscribe("our.message.name",
  function (some, ctx) {
    // ...
  });
```

AngularJS

```
// publish
theApp.controller("bCtrl", function ($rootScope) {
  $rootScope.$broadcast("our.message.name",
    "some", "context");
});
```

AngularJS

```
// publish
theApp.controller("bCtrl", function ($rootScope) {
  $rootScope.$broadcast("our.message.name",
    "some", "context");
});

// subscribe
theApp.controller("aCtrl", function ($scope) {
  $scope.$on("our.message.name",
    function (some, ctx) { ... })
});
```

Summary

- **Maintainable JavaScript**
  - Avoiding the Global Scope means you have to worry less on the collision
  - Using strict JavaScript will highlight errors earlier
  - Structuring your code into modular units will increase stability
  - Injecting dependencies allows you to not handle the wire up of dependencies
  - Abandon nested callbacks in favor of promises or async patterns
  - Use eventing and messaging to loosely couple your modules