Web Programming

Lecture 26

Building a Single Page Application with Angular - 2

Agenda

- Using the controllerAs syntax
- Protecting the global scope using IIFEs
- Dependency injection
- Minifying and concatenating the separate files into one small application file

Controller best practice: Using the controllerAs syntax

- Angular offers a way to create a *view model* that you can bind your data to, rather than attaching everything directly to the $scope object.
- It forces you to contain your data correctly, avoiding the possibility of assigning a value directly to $scope.

Declaring controllerAs in a route definition

- The first step is to tell Angular that you want to use this controller with the controllerAs syntax
- You specify an option of controllerAs and pass it the name of the ViewModel you want to use as a string
- Typical choice of View Model name is vm

```
function config ($routeProvider) {
  $routeProvider
    .when('/', {
      templateUrl: 'home/home.view.html',
      controller: 'homeCtrl',
      controllerAs: 'vm'
    })
    .otherwise({redirectTo: '/'});
}
```

Defining the ViewModel in the controller

- Behind the scenes, when you use a controller in the application, it's generated using JavaScripts *new* method, creating a single instance
- When the *controllerAs* syntax is used, Angular uses *this* inside the function, and binds it to $scope
- Update home controller by removing $scope from function definition and update data bindings to use vm instead of $scope

```
function homeCtrl () {
  var vm = this;
  vm.pageHeader = {
    title: 'Loc8r',
    strapline: 'Find places to work with wifi near you!'
  };
  vm.sidebar = {
    content: "Looking for wifi and a seat? Etc etc…"
  };
}
```

Using the ViewModel in the view

- **Updating the home view to use vm data bindings**

```
<div id="banner" class="page-header">
  <div class="row">
    <div class="col-lg-6"></div>
      <h1>
        {{ vm.pageHeader.title }}
       <small>{{ vm.pageHeader.strapline }}</small>
      </h1>
  </div>
</div>
```

Using the ViewModel in the view

- **Updating the home view to use vm data bindings**

```
<div class="col-xs-12 list-group-item" ng-repeat="location in
 vm.data.locations | filter : textFilter">
      <h4>
        <a href="/location/{{ location._id }}">{{ location.name
    }}</a>
```

Using Services

```
function homeCtrl (loc8rData) {
  var vm = this;
  vm.pageHeader = {
    title: 'Loc8r',
    strapline: 'Find places to work with wifi near you!'
  };
  vm.sidebar = {
    content: "Looking for wifi and a seat? Etc etc…"
  };

loc8rData
.location()
.success(function(data) {
vm.data= { locations: data };
})
.error(function(e) {
console.log(e);
});
}
```

Improving browser performance

- The modular approach that we're taking to coding is great for the maintainability of the codebase, but not so great for browsers if they have to go and download each of the little files separately.
- We are going to:
  - Reduce the number of global variables
  - Reduce the number of files in the browser downloads
  - Reduce the overall size of the JavaScript

Wrap each file in an IIFE

- An IIFE is a way of encapsulating some JavaScript code in a unique scope, hiding the contents from the global scope
- If we wrap a console.log statement

```
(function(){
Console.log(“output immediately”)
})();
```

This puts the console.log inside the function scope and immediately invokes the function

Wrap each file in an IIFE

- At the moment each file is running in the global scope
- This is bad, because it clutters the global scope, increases the risk of variable name collision, and exposes the application code to potential misuse

Wrapping Angular application files in IIFEs, for example, app.js

```
(function () {
  angular.module('loc8rApp', ['ngRoute']);
  function config ($routeProvider) {
    $routeProvider
      .when('/', {
        templateUrl: 'home/home.view.html',
        controller: 'homeCtrl',
        controllerAs: 'vm'
      })
      .otherwise({redirectTo: '/'});
  }
  angular
    .module('loc8rApp')
    .config(['$routeProvider', config]);
})();
```

Wrapping Angular application files in IIFEs

- Wrap each of the java script file we have created so far
- ratingStars.directive.js
- formatDistance.filter.js
- loc8rData.service.js
- home.controller.js

Manually injecting dependencies to protect against minification

- We want to reduce the number of files and the overall file size
- This will involve minifying the scripts
- Righ now, we're injecting the names of the dependencies into controller and service functions as parameters.
- During minification these names get minified into single letters.
- The definition for homeCtrl, for example, looks like this:
  - **function homeCtrl** ($scope, loc8rData, geolocation)
- But if minified it would look something like this:
  - **function homeCtrl**(a,b,c){

Manually injecting dependencies to protect against minification

- Now the parameters arent mere parameters to be used solely in the context of the function
- They are references to other parts of the app such as services
- Our app knows what the service loc8rData is but it doesn’t know anything about a service called b
- We can protect against this by manually injecting the dependencies as strings, which wont get changes during the minify process
- Angular provides an $inject method which accepts an array of strings
- These strings are dependencies for a particular controller or service and match those being passed through as parameters

Manually injecting dependencies to protect against minification

- In the following snippet, we add the dependency injection for the homepage controller, just before the definition of the function
- homeCtrl.$inject = ['$scope', 'loc8rData', 'geolocation'];
- function homeCtrl ($scope, loc8rData, geolocation) {
- The inject method is applied to the name of the function, accepted as an array of the dependencies
- The array should contain strings, because they don’t get changed when the code is minified
- The contents of the array should be in the same order as the parameters in the function

Manually injecting dependencies to protect against minification

- Inject $http into our loc8rData service as follows
- loc8rData.$inject = ['$http'];
- function loc8rData ($http) {

Using UglifyJS to minify and concatenate scripts

- To minify and concatenate the Angular application scripts we're going to use a third-party tool called UglifyJS.
- The aim is that when the Node application starts up in Express, UglifyJS will take the source files of our Angular application and put them all into one file and compress it.
- We'll update our application to use this single output file, instead of the multiple files we're using at the moment.

Installing UglifyJS

- $ npm install uglify-js –save

Adding UglifyJS to the application

- UglifyJS needs to be required in the main /app.js
- We also reference fs module which stands for filesystem, as we need acces to the filesystem to save the new uglified file

```
var express = require('express');
var path = require('path');
var favicon = require('serve-favicon');
var logger = require('morgan');
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');
require('./app_api/models/db');
var uglifyJs = require("uglify-js");
var fs = require('fs');
```

Uglifying JavaScript files

- We will use UglifyJS to combine all our Angular application files into one and then minify them
- This process happens in memory, so once the combined file is generated we'll use the filesystem to save it
- Insert some code in the main app.js file
  - List of all the files we want to combine in an array
  - Call UglifyJS to combine and minify the file in memory
  - Save the uglified code into the public folder

Uglifying JavaScript files

```
routeapp.set('views', path.join(__dirname, 'app_server', 'views'));
app.set('view engine', 'jade');
var appClientFiles = [
  'app_client/app.js',
  'app_client/home/home.controller.js',
  'app_client/common/services/geolocation.service.js',
  'app_client/common/services/loc8rData.service.js',
  'app_client/common/filters/formatDistance.filter.js',
  'app_client/common/directives/ratingStars/ratingStars.directive.js'
];
var uglified = uglifyJs.minify(appClientFiles, { compress : false });
fs.writeFile('public/angular/loc8r.min.js', uglified.code, function (err){
  if(err) {
    console.log(err);
  } else {
    console.log('Script generated and saved: loc8r.min.js');
  }
});
```

Uglifying JavaScript files

- At this stage the minified file is 2,160 bytes, down from around 3,575 bytes for the individual files
- As well as reducing the number of browser requests from 6 to 1, we have reduced the file size by around 40%

Using the new minified file in the HTML

- Update layout.pug addding the new minified file

```
script(src='/angular/loc8r.min.js')
//- script(src='/app.js')
//- script(src='/common/services/loc8rData.service.js')
//- script(src='/common/services/geolocation.service.js')
//- script(src='/common/directives/ratingStars/ratingStars.directive.js')
//- script(src='/common/filters/formatDistance.filter.js')
//- script(src='/home/home.controller.js')
```

A full SPA: Removing reliance on server-side application

- The navigation, page framework, header and footer are all in a pug template
- To use this template we have a controller in app_server
- But to have a real SPA we want everything inside the app_client folder
- We will create host HTML page in app_client and update the express routing to point to this
- We will take the sections of the HTML page and make them into reusable components as directives

Creating in isolated HTML host page

- Convert layout.pug into HTML and save it as app_client/index.html

Routing to the static HTML file from Express

- In the main app.js file, comment out or delete the line requiring server application routes
- Add catchall app.use function to respond to any requests that make it this far by sending HTML file

```
// require('./routes')(app);
require('./app_api/routes')(app);
app.use(function(req, res) {
  res.sendfile(path.join(__dirname, 'app_client', 'index.html'));
});
```

- At this stage your homepage will run just as before but noew we have removed the reliance on server app routes to deliver base HTML page

Making reusable page framework directives

- From the HTML page, we will take the footer and navigation and turn them into directives so that we can include them in page we want
- A directive is composed of 2 main parts.
- A javascipt file to define it and a view template to display it
- In turn, each js file will have to be added to app.js so the app can use it
- Each directive will be placed as an element(or element attribute) into host views where required

Making a footer directive

- We call our footer footerGeneric and create a fodler in app_client/common/directives called footerGeneric
- Put both js and html file for the directive
- Create footerGeneric.template.html and paste in html for footer
    - <footer>
    -
    - <small>© Simon Holmes 2014</small>
    -
    - </footer>

Making a footer directive

- Create footerGeneric.directive.js
- Here define the new directive, register with the main application and assign the HTML file we have created as the view template

```
(function () {

  angular
    .module('loc8rApp')
    .directive('footerGeneric', footerGeneric);
  function footerGeneric () {
    return {
      restrict: 'EA',
      templateUrl: '/common/directives/footerGeneric/
      footerGeneric.template.html'
    };
   }

})();
```

Making a footer directive

- Add this file to appClientFiles array in app.js
- Now when we want to include footer in our Angular page we can use the new element <footer-generic></ footer-generic >

Moving the navigation into a directive

- Navigation.template.html
- Navigation.directive.js

Creating a directive for the page header

- Page header needs to display different data on different pages

```
<div id="banner" class="page-header">
  <div class="row">
    <div class="col-lg-6"></div>
      <h1>
        {{ content.title }}
       <small>{{ content.strapline }}</small>
      </h1>
  </div>
```

Creating a directive for the page header

- **Defining the page header directive: pageHeader.directive.js**

```
(function () {
  angular
    .module('loc8rApp')
    .directive('pageHeader', pageHeader);
  function pageHeader () {
    return {
      restrict: 'EA',
      scope: {
        content : '=content'
      },
      templateUrl: '/common/directives/pageHeader/pageHeader.template.html'
    };
  }
})();
```

Creating a directive for the page header

- We need to pass through the content object as an attribute of the element
- <page-header content="vm.pageHeader"></page-header>

Final homepage template

```
<navigation></navigation>

<div class="container">
  <page-header content="vm.pageHeader"></page-header>

  <div class="row">
All the html for displaying locations
</div>

  <footer-generic></footer-generic>
</div>
```

Final index.html file

```
<!DOCTYPE html>
<html ng-app="loc8rApp">
  <head>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Loc8r</title>
    <link rel="stylesheet" href="/bootstrap/css/amelia.bootstrap.css">
    <link rel="stylesheet" href="/stylesheets/style.css">
  </head>
  <body ng-view>
    <script src="/angular/angular.min.js"></script>
    <script src="/lib/angular-route.min.js"></script>
    <script src="/angular/loc8r.min.js"></script>
    <script src="//ajax.googleapis.com/ajax/libs/jquery/1.10.2/ jquery.min.js"></script>
    <script src="/bootstrap/js/bootstrap.min.js"></script>
    <script src="/javascript/validation.js"></script>
  </body>
</html>
```

References

- Getting MEAN chapter 9 Simon Holmes
- Getting MEAN chapter 10 Simon Holmes