Services & Factories

DEVELOPMENTOR

DEVELOPING PEOPLE WHO DEVELOP SOFTWARE

Objectives

- Understand how services can make your controller better
  - Use the Single Responsibility design pattern
- Learn about the most commonly used build in services
  - $scope and $rootScope
  - $cookies
  - $q
  - $window, $timeout, $interval and $log
  - Build in filters
- Understand different ways of providing values and services
  - Removes shared functionality from controllers

Scopes and RootScopes

- Each Angular application has a $rootScope service
  - Most other scopes are prototypically linked to it
- Controller and some directives create child scopes
  - Can also be done by calling $new() on another scope
- Scopes can notify Angular that data has changed
  - By calling the $watch() function
- Allow for event broadcasting and subscription
  - The $on() function subscribes an event listener
  - The $broadcast() function broadcasts an event to all child scopes
  - The $emit() function notifies parent scopes of an event

Using browser cookies

- The $cookies and $cookieStore services provide access to the browers cookies
  - Automatically (de)serialized as JSON
- Requires the ngCookies module
  - Located in angular-cookies.js

Promise and deferred results

- The $q service let us create promises
  - Based on the popular Q library by Kris Kowal
- Create a deferred object and return it's promise
  - On success resolve() the deferred
  - On failure reject() the deferred
- The caller subscribes to the then() function for notification of the result
  - First callback is for success and the second for failure

Promise and deferred results

```
app.factory("futureEven", function ($q, $timeout) {
    return {
        check: function (value) {
            var deferred = $q.defer();

            $timeout(function () {
                deferred.resolve(value % 2 === 0);
            }, 1000);

            return deferred.promise;
        }
    };
});

app.controller("DemoCtrl", function ($scope, futureEven) {
    $scope.checkEven = function () {
        $scope.isEven = undefined;
        futureEven.check($scope.value).then(function (result) {
            $scope.isEven = result;
        });
    };
});
```

Browser utility services

- The $window wraps the global window object
  - Useful for mocking test dependencies
- The $log service is a simple logging service
  - log(), error(), info() etc methods just like the console object
- There is a mock version in angular-mocks.js
  - Stores messages in an array per log level

```
app.controller("DemoCtrl", function ($scope, $window, $log) {
    $scope.displayAlert = function () {
        $log.warn("About the show an alert window");
        $window.alert("The message");
    };
});
```

Browser utility services

- The $timeout service is a wrapper for setTimeout()
  - Also caches errors and passes them to the $exceptionHandler
- Returns a promise object
  - Can be cancelled using $timeout.cancel(promise)
- Don’t use repeating $timeout() with Protractor E2E tests
  - Use the $interval() service instead

- The $interval service is a wrapper for setInterval
  - Returns a promise object
  - Can be cancelled using $interval.cancel(promise)

- Both services have mock implementations
  - Call flush() on the service to execute pending tasks

Build in filter services

- AngularJS contains a number of built-in filters
  - Most are used to format bindings
    - date
    - currency
    - Etc.
- The *filter* filter selects a subset of items from array
  - Usually used in combination with an ngRepeat directive
- Filters can also be used programmatically
  - Use the $filter service

```
Search: <input type="text" ng-model="filterText" />

<ol>
  <li ng-repeat="person in people | filter:filterText">
    {{person.firstName}} {{person.lastName}}
  </li>
</ol>
```

Creating custom Modules & Services

- Modules are groupings of related functionality
  - Also used to bootstrap the application
- Services are reusable pieces of business logic
  - Separation results in reuse and testability
- Created as singleton objects
  - Inject by AngularJS using dependency injection
- Services are created as part of a module
  - One module can take a dependency on another module

The $provide service

- Used to registering components with the $injector
  - Constants
  - Values
  - Factories
  - Services
  - Providers
  - Decorators
- Most can also be registered using a module
  - Except a decorator

Constants

- Register a simple constant value
  - Can't be changed with a decorator
- Can also be inserted into a config() function
- Often used to expose external libraries
  - Makes dependencies more explicit
  - For example underscore.js

```
var app = angular.module("myApp", []);

app.constant("myName", "Maurice");

app.controller("DemoCtrl", function ($scope, myName) {
    $scope.myName = myName; // Maurice
});
```

Values

- Register a simple value
  - Can be intercepted and changed using a decorator

```
var app = angular.module("myApp", []);

app.value("myName", "Maurice");

app.controller("DemoCtrl", function ($scope, myName) {
    $scope.myName = myName; // Maurice
});
```

Factories

- Register a factory function
  - Provides the result of the function specified to be injected
- The most common way to register services

```
var app = angular.module("myApp", []);

app.factory("nameSvc", function () {
    return {
        myName: "Maurice",
        doStuff: function() {
        }
    };
});

app.controller("DemoCtrl", function ($scope, nameSvc) {
    $scope.myName = nameSvc.myName; // Maurice
    nameSvc.doStuff();
});
```

Services

- Register a service constructor
  - Provides an instance of the constructor function provided

```
var app = angular.module("myApp", []);

app.service("nameSvc", function () {
    this.myName = "Maurice";
    this.doStuff = function () {
    };
});

app.controller("DemoCtrl", function ($scope, nameSvc) {
    $scope.myName = nameSvc.myName; // Maurice
    nameSvc.doStuff();
});
```

Providers

- The most capable way of providing an injectable
  - Take a dependency on either the named service or its provider
- The $get() function is called to create the actual service
  - The provider itself can expose additional functions as needed for configuration purposes
- Services and Factories are just convenient wrappers
  - Internally they are all implemented using a provider

A simple provider

```
var app = angular.module("myApp", []);

app.provider("nameSvc", function () {
    return {
        $get: function () {
            return {
                myName: "Maurice",
                doStuff: function () { }
            };
        }
    };
});

app.controller("DemoCtrl", function ($scope, nameSvc) {
    $scope.myName = nameSvc.myName; // Maurice
    nameSvc.doStuff();
});
```

Configuring a provider

```
var app = angular.module("myApp", []);
app.provider("nameSvc", function () {
    var theService = {
        myName: "Maurice",
        doStuff: function () { }
    };
    return {
        $get: function () {
            return theService;
        },
        changeName: function (name) {
            theService.myName = name;
        }
    };
});

app.config(function(nameSvcProvider) {
    nameSvcProvider.changeName("Jack");
});

app.controller("DemoCtrl", function ($scope, nameSvc) {
    $scope.myName = nameSvc.myName; // Jack
    nameSvc.doStuff();
});
```

Decorators

- Can modify the result of other providers
  - Either change or replace the original result
- Only constants can't be decorated
- $delegate points to the original service/value

```
var app = angular.module("myApp", []);
app.value("myName", "Maurice");

app.config(function ($provide) {
    $provide.decorator("myName", function ($delegate) {
        return "My name is " + $delegate;
    });
});

app.controller("DemoCtrl", function ($scope, myName) {
    $scope.myName = myName; // My name is Maurice
});
```

Filters

- Filters are used to manipulate expression results
- Normally used declaratively in a binding
  - Can also be inserted as a dependency and used imperatively.

```
<!DOCTYPE html>
<html>
<head>
    <title>Filter demo</title>
</head>
<body ng-app="myApp">
    <div ng-controller="DemoCtrl">
        {{now | toDate}}
        <br />
        {{now | toTime}}
    </div>

    <script src="scripts/angular.js"></script>
    <script src="App.js"></script>
</body>
</html>
```

Filters

```
var app = angular.module("myApp", []);

app.filter("toDate", function () {
    return function (date) {
        return date.toDateString();
    };
});

app.filter("toTime", function () {
    return function (date) {
        return date.toTimeString();
    };
});

app.controller("DemoCtrl", function ($scope) {
    $scope.now = new Date();
});
```

Summary

- Using service makes writing controllers easier
  - The controller should mainly call into service
  - A good place to put cross cutting concerns
- Angular provides many usable services for you
  - Scopes are the glue between the controller and view
- Expose other non Angular libraries
  - Usually as a constant
- It is easy to register your own services
  - Usually with a factory()
  - Sometimes with a provider