SERVICES

http://oricalvo.com
ori.calvo@gmail.com

Objectives

- Angular offers many built-in services
  - $compile
  - $q
  - $http
- In this chapter
  - Understand what is a service
  - Introduce the services beyond service management
  - Understand the different between $injector & $provider

http://oricalvo.com
ori.calvo@gmail.com

Services

- User defined types
- Are used to organize code (modularity)
- Share code between different application parts
- Can be injected using Angular DI feature
- Are always
  - Lazy instantiated
  - Singleton

http://oricalvo.com
ori.calvo@gmail.com

Service Recipes

- Angular supports several mechanisms for publishing services
  - Value
  - Service
  - Factory

Value

- Register a pre instantiated object
- We control the instantiation process
- No easy way to specify dependencies
- In practice could be used to publish 3rd party libraries

```
(function (window) {
    var log4JS = {
        debug: function (message) {...},
        warning: function (message) {...},
        error: function (message) {...}
    };

    window.log4JS = log4JS;
})(window);
```

```
angular.module("MyApp", [])
    .value("Logger", window.log4JS);
```

```
function HomeCtrl($scope, Logger) {
    Logger.debug("HomeCtrl created");
}
```

Service

- Allows for constructor function registration
- Can express dependencies
- Instantiated by Angular using "new" syntax

```
function Logger() {
     this.debug("Logger created");
}

Logger.prototype = {
     constructor: Logger,
     debug: function (message) {...},
     warning: function (message) {...},
     error: function (message) {...}
};

angular.module("MyApp").service("Logger", Logger);
```

```
function HomeCtrl($scope, Logger) {
     Logger.debug("HomeCtrl created");
}
```

Factory

- Register a factory function
- The factory function must return an object instance
- May specify dependencies

```
angular.module("MyApp").factory("Logger", function () {
    var counters = {
        err: 0,
        wrn: 0,
        dbg: 0,
    };
    return {
        debug: function (message) {
            console.log("DBG: " + message);
        },
        warning: function (message) {
            console.log("WRN: " + message);
        },
        error: function (message) {
            console.log("ERR: " + message);
        }
    };
});
```

```
function HomeCtrl($scope, Logger) {
    Logger.debug("HomeCtrl created");
}
```

Provider

- A service factory
- Allows the application to configure the service before the service is created
- Must conform to Angular specification
  - $get function
  - Returns the service object
- Usually is defined when implementing 3rd party Angular modules
  - Less common for application

Provider Sample


```
angular.module("MyApp", [])
     .config(function (LoggerProvider) {
         LoggerProvider.enableBuffering(10);
     });
```

```
function LoggerProvider() {
     var bufferSize = -1;
     this.$get = function () {
         return {
             debug: function (message) {...},
             warning: function (message) {...},
             error: function (message) {...}
         };
     }
     this.enableBuffering = function (bufSize) {
         bufferSize = bufSize || 5;
     }
}

angular.module("MyApp").provider("Logger", LoggerProvider);
```

```
function HomeCtrl($scope, Logger) {
     Logger.debug("HomeCtrl created");
}
```

Provider Notes

- The provider is registered with name X but is requested with the name XProvider
- Provider is only accessible during application configuration phase
  - Only config block can ask for a provider
- Provider is always instantiated
  - Even if the service is not requested by application
  - Is instantiated before config block

Constant

- Even a constant value can be injectable
- Useful for sharing constant data between different providers/services
- Can be requested by a config block

```
angular.module("MyApp", [])
     .constant("MSIE", !!document.documentMode)
     .config(function (LoggerProvider) {
         LoggerProvider.enableBuffering(10);
});
```

documentMode is an IE only attribute

```
function LoggerProvider(MSIE) {
     console.log("LoggerProvider created");
     console.log("    MSIE: " + MSIE);
     this.$get = function () {
     }
}
```

Cache

- All providers and services are singletons
  - Multiple requests for the same service return the same reference
- This implies using an internal cache for caching instantiated objects
- Internally, Angular holds two different caches
  - Provider cache
  - Service cache
- Managed by the $injector

$injector

- Is created by Angular during the bootstrapping phase
  - Loads all modules
  - Instantiates all providers (from all modules)
  - Invokes config and run blocks
  - See Bootstrapping chapter for more details
- After bootstrapping, is responsible for managing Angular Dependency Injection mechanism

$injector


```
function createInjector(modulesToLoad) {
    var providerCache = {
            $provide: {
                provider: supportObject(provider),
                factory: supportObject(factory),
                service: supportObject(service),
                value: supportObject(value),
                constant: supportObject(constant),
                decorator: decorator
            }
        },
        providerInjector = (providerCache.$injector =
            createInternalInjector(providerCache, function (serviceName, caller) {
            })),

        instanceCache = {},
        instanceInjector = (instanceCache.$injector =
            createInternalInjector(instanceCache, function (serviceName, caller) {
            }));

    return instanceInjector;

    function provider(){...}
    function factory(){...}
    function service(){...}
    function value(){...}
    function constant(){...}
    function decorator(){...}
}
```

$provide offers the same (almost) API as Angular module

Internally, two different injectors are created

Provider function

- Immediately instantiates the provider and stores it into the provider cache

```
function provider(name, provider_) {
     if (isFunction(provider_) || isArray(provider_)) {
         provider_ = providerInjector.instantiate(provider_);
     }

     if (!provider_.$get) {
         throw $injectorMinErr('pget', "...", name);
     }

     return providerCache[name + providerSuffix] = provider_;
}
```

Array ? Why ?

Provider Wrappers

- service/factory/value are just wrappers around the provider function

```
function factory(name, factoryFn, enforce) {
     return provider(name, {
         $get: factoryFn
     });
}

function service(name, constructor) {
     return factory(name, ['$injector', function ($injector) {
         return $injector.instantiate(constructor);
     }]);
}

function value(name, val) { return factory(name, valueFn(val), false); }

function valueFn(value) { return function () { return value; }; }
```

$injector API

- Usually not used directly by application
- However, you may find it useful when implementing cross application infrastructure
- Offers the following API
  - invoke
  - instantiate
  - get
  - annotate
  - has

$injector.invoke

- $injector.invoke analyzes the list of parameters and injects the requested services
- In case a service instance does not exist it is instantiated
- All parameters can be overridden using invoke's third argument (A.K.A locals)

```
angular.module("MyApp", []).controller("HomeCtrl", function HomeCtrl($injector) {
    function func($rootScope, param1) {
        console.log("%O", this);
        console.log("param1 = " + param1);
    }

    $injector.invoke(func, this, { param1: 123 });
});
```

$injector Metadata

- How does $injector retrieve the list of method's parameter ?
  - It uses Function.toString and parses the source code
- But what if we use minification tools ?
  - Like Google Closure Compiler
- Then we must manually specify the parameter list
  - Using $inject
  - Using array syntax

Manually specify Parameter List


```
function func(rs, p) {
     console.log("$rootScope: ", !!rs);
     console.log("param1: " + p);
}

func.$inject = ["$rootScope", "param1"];
```

```
var func = ["$rootScope", "param1", function (rs, p) {
     console.log("$rootScope: ", !!rs);
     console.log("param1: " + p);
}];
```

```
$injector.invoke(func, this, { param1: 123 });
```

$injector.invoke


```
function invoke(fn, self, locals) {
     var args = [],
         $inject = createInjector.$$annotate(fn, strictDi, serviceName),
         length, i,
         key;
     for (i = 0, length = $inject.length; i < length; i++) {
         key = $inject[i];
         if (typeof key !== 'string') {
             throw $injectorMinErr('itkn', ...);
         }
         args.push(
           locals && locals.hasOwnProperty(key)
           ? locals[key]
           : getService(key)
         );
     }
     if (isArray(fn)) {
         fn = fn[length];
     }

     return fn.apply(self, args);
}
```

$injector.annotate


```
function annotate(fn, strictDi, name) {
    var $inject,
        fnText,
        argDecl,
        last;
    if (typeof fn === 'function') {
        if (!($inject = fn.$inject)) {
            $inject = [];
            if (fn.length) {
                fnText = fn.toString().replace(STRIP_COMMENTS, '');
                argDecl = fnText.match(FN_ARGS);
                forEach(argDecl[1].split(FN_ARG_SPLIT), function (arg) {
                    arg.replace(FN_ARG, function (all, underscore, name) {
                        $inject.push(name);
                    });
                });
            }
            fn.$inject = $inject;
        }
    } else if (isArray(fn)) {
        last = fn.length - 1;
        $inject = fn.slice(0, last);
    }

    return $inject;
}
```

Remove Comments

What is the purpose of this code ?

Cache the result

Testability

- A typical unit test initializes a service once, and verifies it multiple times
- Usually we want the service variable to be named exactly like the service name
  - Causes variable duplication

Duplication

```
describe("Storage", function () {
    var Storage;

    beforeEach(module("MyApp"));

    beforeEach(inject(function (Storage) {
        Storage = Storage;
    }));
});
```

Underscore Stripping

- $injector removes leading and trailing underscores

```
describe("Storage", function () {
    var Storage;

    beforeEach(module("MyApp"));

    beforeEach(inject(function (_Storage_) {
        Storage = _Storage_;
    }));

    it("Does not allow deleting the root item", function () {
        expect(function () {
            Storage.deleteItem(0);
        }).toThrow();
    });
});
```

No Duplication

Challenge

- Suppose we define a JavaScript class that is instantiated multiple times
  - Therefore, we cannot register it as a service

```
function Contact($http, name, email) {
     this.$http = $http;
     this.name = name;
     this.email = email;
}

Contact.prototype.dump = function () {
     console.log(this.name + ", " + this.email);
}
```

- However, the constructor has dependencies that only Angular knows how to resolve

$injector.instantiate

- $injector can instantiate a plain JavaScript class while injecting all dependencies
- Even better, the caller can still send custom parameter (A.K.A locals)

```
angular.module("MyApp", [])
     .run(function ($injector) {
         var contact = $injector.instantiate(Contact, { name: "Ori", email: "ori@gmail.com" });
         contact.dump();
});
```

```
function Contact($http, name, email) {
     this.$http = $http;
     this.name = name;
     this.email = email;
}

Contact.prototype.dump = function () {
     console.log(this.name + ", " + this.email);
}
```

Config vs. Run Blocks

- Both config and run blocks can ask for $injector
- This implies that run block can request providers too

Error

```
angular.module("MyApp", [])
    .config(function ($injector) {
        var $provide = $injector.get("$provide");
    })
    .run(function ($injector) {
        var $provide = $injector.get("$provide");
    });
```

- Fortunately, above code generates an error
- The $injector being sent to the run block is different than the one being sent to the config block

$injector

- Internally holds two caches
  - Provider cache
  - Instance cache
- Provider cache holds only providers/constants
- Instance cache holds services/factories/values/constants
- During config block only provider cache is considered
- During run block only instance cache is considered

createInjector (again)


```
function createInjector(modulesToLoad, strictDi) {
    var loadedModules = new HashMap([], true),
        providerCache = {
            ...
        },

        providerInjector = providerCache.$injector =
            createInternalInjector(providerCache, function (serviceName, caller) {
                throw $injectorMinErr('unpr', "Unknown provider: {0}", path.join(' <- '));
            }),

        instanceCache = {},

        instanceInjector = instanceCache.$injector =
            createInternalInjector(instanceCache, function (serviceName, caller) {
                var provider = providerInjector.get(serviceName + providerSuffix, caller);
                return instanceInjector.invoke(provider.$get, provider, undefined, serviceName);
            }));

    forEach(loadModules(modulesToLoad), function (fn) { instanceInjector.invoke(fn || noop); });

    return instanceInjector;
}
```

Provider injector is not allowed to create new services

Instance injector does not consider the provider cache

Accessing $injector from the DOM

- $injector is attached to the root element using jQuery data method
- This means that 3rd party library can "reach" into Angular using the DOM
- Angular extends jQuery with the injector method

```
$(function () {
     var $injector = $("body div").injector();

     var $rootScope = $injector.get("$rootScope");

     console.log("$rootScope: " + !!$rootScope);
});
```

$provide

- Allows registration of providers/services
- Is a provider (not a service)
  - You can request it only during config block
- Offers the same API as module
  - provider, factory, service, value, constant
  - Nothing new
- However, also offers special method named decorator
  - Angular 1.4 offers a decorator method at the module level

$provide


```
function createInjector(modulesToLoad) {
    var providerCache = {
            $provide: {
                provider: supportObject(provider),
                factory: supportObject(factory),
                service: supportObject(service),
                value: supportObject(value),
                constant: supportObject(constant),
                decorator: decorator
            }
        },
        providerInjector = (providerCache.$injector =
            createInternalInjector(providerCache, function (serviceName, caller) {
            })),

        instanceCache = {},
        instanceInjector = (instanceCache.$injector =
            createInternalInjector(instanceCache, function (serviceName, caller) {
            }));

    return instanceInjector;
}
```

$provide offers the same (almost) API as Angular module

```
function Service1() {...}

function Service2() {...}

angular.module("MyApp", [])
    .config(function ($provide) {
        $provide.service({
            "Service1": Service1,
            "Service2": Service2,
        });
    });
```

Common, when implementing 3rd party library

$provide.decorator

- Allows you to "monkey patch" an existing service
- Great for enhancing existing Angular/3rd party services

```
angular.module("3rdPartyLib", []).factory("Logger", function () {
    return {
        debug: function (message) {...},
        warning: function (message) {...},
        error: function (message) {...}
    };
});
```

```
angular.module("MyApp", ["3rdPartyLib"])
    .config(function ($provide) {
        $provide.decorator("Logger", function ($delegate) {
            var originalDebug = $delegate.debug;

            $delegate.debug = function (message) {
                console.log("Before debug");
                originalDebug.call($delegate, message);
                console.log("After debug");
            }

            return $delegate;
        });
    });
```

$provide.decorator – Be Aware

- Assume you decided to decorate existing Angular service
- You may choose to return a new object from the decorator function
- However, Angular might still hold an internal reference to the original service
- Might lead to surprising results
- You should prefer “monkey patching”

Strict DI

- New to Angular 1.3
- When enabled, Angular verifies that a function being invoked by $injector supplies manually the name of the parameters to be injected

```
<!DOCTYPE html>
<html ng-app="MyApp" ng-strict-di>
 <head>
    <title>Index</title>
 </head>
 <body>
   ...
 </body>
 </html>
```

```
function HomeCtrl($scope) {
     console.log("HomeCtrl created");
}

angular.module("MyApp")
     .controller("HomeCtrl", HomeCtrl);
```

Must specify manually the parameters names

```
angular.module("MyApp")
     .controller("HomeCtrl", ["$scope", HomeCtrl]);
```

Summary

- Angular offers multiple recipes for registering services
- $injector is a core service which manages dependency injection
- $provide allows you to decorate services