Angular Services

angular services represents the model part of angular MVC architecture, the services contains the business logic of your app and can be shared across you controllers. The services are injected to controller directives and services with the help of angular Dependency Injector. Angular services are lazily instantiated meaning the DI will create the instance only when it's needed, also each service is created once with the service factory and are saved as singletons. if you want to use a service you can simply inject it in the constructor function of the controller, or service (a service can also have dependencies on other services).

In this tutorial we will cover two types of services.

Constant

A constant is a value that can't be changed. a constant can be injected everywhere.
Example let's create a constant that holds our server url: https://nztodo.herokuapp.com

open a new directory for this tutorial, and in this tutorial run:
> bower init
> bower install angular --save

create a new file called **index.html** with the following code:

```
<!DOCTYPE html>
<html ng-app="AppModule">
<head>
  <meta charset="UTF-8">
  <title>Angular Services</title>
</head>
<body>
  <div ng-controller="MainController as ctrl">
     <div ng-include="'src/templates/main.template.html'">

     </div>
  </div>

  <script src="bower_components/angular/angular.js"></script>
  <script src="src/app.module.js"></script>
  <script src="src/controllers/main.controller.js"></script>
```

```
  <script src="src/services/constants.js"></script>

</body>
</html>
```

in the index file we set the directive **ng-app** to the module we will soon create .
we also set the **ng-controller** and **ng-include** to a controller and template we will soon create.

create the folder **src** that will hold our **js** files and templates and create the file **app.module.js**

```
angular.module('AppModule', []);
```

now create in the folder **src** a folder called **services** and create the file **constants.js** in the folder with the following code:

```
angular.module('AppModule')
  .constant('SERVER_URL', 'https://nztodo.herokuapp.com/');
```

we created a **constant** and attached it to our **module**, we named our constant: **SERVER_URL** and placed the value of the URL of our server.
now we can go ahead and inject our constant to our controller.
In the **src** folder create another folder called **controllers** and create the file **main.controller.js**

```
angular.module('AppModule')
  .controller('MainController', ['SERVER_URL',function(SERVER_URL){
  var self = this;

  self.serverUrl = SERVER_URL;
}]);
```

notice that we injected the constant **SERVER_URL** and we can use it in our controller, notice also the injection syntax we are using, we are placing in an array the names of the services we want to inject then we put in the function the arguments exactly in the order we placed in the array. This syntax is for minification purpose, when we minify our source code the function argument names will change so we need to specify in a string value what the true name of the service is.
We are placing the constant in a property of the controller named **serverUrl.**

let's create the template of our controller.
in the folder **src** create a folder called **templates,** with the file: **main.template.html:**

```
<h1>
  Angular Services
</h1>
```

```
<h2>
  Constant
</h2>
<p>
  This value came from the constant we defined: <strong>{{ctrl.serverUrl}}</strong>
</p>
```

launching your app you should see the value of the constant is printed.

Factory

A factory in angular contains a recipe for creating an instance of a class. We can create the factory by calling the **module factory** method which will attach a factory to the **module.** Angular creates the factory lazily only when someone needs it, the injector will run the factory method one and cache the result. We can use the factory method to return our class or we can return a new instance of our class. To demonstrate the usage of our factory let's try and create a new file in the **services** directory called **pokemon.factory.js** to try and create our factory.

```
angular.module('AppModule')
  .factory('Pokemon', function(){

    console.log('this should run only once');

    return function(){
        var self = this;
        self.name = 'hello';
        self.sayHello = function sayHello(){
            console.log('sayHello');
        }
    }
  });
```

our factory here is returning a **Pokemon** class, notice that above the function is a message that will only be printed once at the time when the injector needs to inject this factory for the first time, after that the injector will cache the result and return the class without the top code. Notice that we are returning here a class and not a class instance, meaning that if you want to use the api of the class you will have to create a new instance of that class.

Can you guess what will happen if we use the **new** keyword ? Does it resemble a pattern you know ?

Since the injector is caching the factory method it means that if we use the **new** keyword in our function we basically are creating a single cached instance of our class which is basically a **singleton** pattern.

Don't forget to add the **pokemon.factory.js** to the **index.html** file

now to use this factory change your controller to look like this:

```
angular.module('AppModule')
  .controller('MainController', ['SERVER_URL', 'Pokemon',function(SERVER_URL,
Pokemon){
  console.log('main controller');
  var self = this;

  self.serverUrl = SERVER_URL;
  var pokemon = new Pokemon();
  pokemon.sayHello();
  console.log(pokemon.name);
  pokemon.name = 'pikachu';
  console.log(pokemon.name);
}]);
```

notice that we we are injecting the **Factory** in the array of the **controller** method of the module and also we added the **factory** argument in the controller constructor function. We can now use the **factory** class by creating a new instance of the class.

Student Task

- Create a constant holding the server url like we did in the first example
- Create a factory which declares a **Task** class which has the following private members:
    - id : number
    - title : string
    - description : string
    - group : string
    - when : date
- your factory task should have the getters and setters for the private members
- you factory should have the following public methods
    - toDict - which will return a dictionary representation of the object
    - toString - which returns a json string of the task
    - fromDict - init the members from a dictionary object

- Create another **factory** called **taskService** which is a singleton and contains the following methods - those methods should only contain signature and a console log message:
    - addTask - when we learn about server communication it will contain post request to the server to create a new task, this method accepts an instance of a Task
    - updateTask - accepts an instance of a task
    - deleteTask -accepts an instance of a task
    - getTasks - returns all the tasks
    - getTask - accepts a number return a single task
- all methods should be signature only not implemented and we will implement them when we learn about server communication