Working with server with Angular 1.5

We will often need to work with a server on our Angular app, and most of the time it will be a **REST server.** Angular provides us tools that will ease our interaction with a server, easily create requests, and get responses, and it will be especially easy to work with a rest server. In this tutorial we will try and practice Angular server communication with a rest server we created.

Our rest server - Todo server

a Rest server is a server that communicates with the clients in a certain communication convention. The rest server utilize the request to understand exactly what the user wants to do and the server uses the response to send the data and the status of the request. Usually the server is talking to a **Database** to get the data for the response.
So basically if you want to communicate with a rest server you need the api endpoint, meaning the url that you will refer and you need to send a request based on REST convention.
For our example the server url is: **https://nztodo.herokuapp.com/**
and the api that is exposed in this URL is: **/api/task/**
So with our rest server we can create a **GET** request to the following url to get all the tasks:
https://nztodo.herokuapp.com/api/task/?format=json
and if we want a single task we can send a **GET** request to the following URL:
https://nztodo.herokuapp.com/api/task/<id>/?format=json
so in a REST server to retrieve data we usually send a **GET** request.
if we want to create a new task we can send a **POST** request to the following URL:
https://nztodo.herokuapp.com/api/task/
in the post body we need to send the data for the new task we are creating, for example our post body can contain:
{"id":8499,"title":"test12","description":"hello yariv","group":"yariv-katz","when":"2016-09-01T15:10:42Z"}

so in a rest server a **POST** request is meant to create a new row in the database

put is similar to the post request only the url is:
https://nztodo.herokuapp.com/api/task/<id>/
and the body contains the new updated values for the task row with the id specified

to delete a row you simply need to send a **DELETE** request to the same URL as the PUT request and it will delete the row with the id specified in the URL.

$http

the **$http** service is a function, which takes a configuration object as single argument. the configuration object is used to create http request and return a promise. the promise function get's the response object as argument.
Let's try and use the **$http** server.

bootstrap

Create a new directory for this angular project and cd into that directory. In your terminal type:
> bower init
> bower install angular --save

create an **index.html** file in the root directory. and place the following code in that file:

```
<!DOCTYPE html>
<html ng-app="AppModule">
<head>
  <meta charset="UTF-8">
  <title>Angular Server Communication</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>
</body>
</html>
```

we will first create the **root module** of our app, we saw above that we named that module **AppModule.**
create a folder called **src** and create a file in that folder called **app.module.js** with the following code:

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

now in the **src** folder create a folder called **controllers** that will hold the controllers for our app. and in that folder create a file called **main.controller.js** with the following code:

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

      self.greeting = 'hello main controller';
  }]);
```

now create a template for your controller, in the **src** folder create a folder called **templates** and in that folder create the file: **main.template.html** with the following code:

```
<h1>
  {{ctrl.greeting}}
</h1>
```

launch you app just to check that everything is running properly, you should see the greeting message that proves that our controller controls the template we created and everything is connected.

task-service

it's not recommended to use the **$http** directly in the controller, in fact it's much better to wrap server communication in a factory. create a folder in **src** called **services** and in the folder place a file called: **task-service.factory.js** with the following code:

```
angular.module('AppModule')
  .factory('taskService', ['$http', function($http){
      return new function(){
          var self = this;
          var _url = 'https://nztodo.herokuapp.com/';

          /**
           * get all the tasks from the server
           * @returns {Promise}
           */
          self.getTasks = function getTasks(){
              return $http({
                  method: 'GET',
                  url: _url + 'api/task/?format=json'
              });
```

```
        }

        /**
         *
         * @returns {Promise}
         */
        self.createTask = function createTask(){
            return $http({
                method: 'POST',
                url: _url + 'api/task/',
                data: {
                    "title": "simple task",
                    "description": "buy groceries",
                    "group": "yariv-katz",
                    "when": "2016-09-01T15:10:42Z"
                }
            });
        }
    }
  }]);
```

notice that we created a **factory** and injected in our factory the **$http** service.
We created this factory as a singleton so that's why we are returning a **new function()...**
this will ensure us that we create a instance and the injector caching will make sure that this instance is created once, that is the reason why the name of the factory we start with a lower letter to emphasize that this factory is an instance and not a class.
We created two methods in our class, one to get all the tasks and one to create a new task. You can see the **$http** is quite simple to use, you call it as a method and place configuration object that tells which url you are sending the request what is the data and the method.
You can examine the full api of the configuration object at the following url:
https://docs.angularjs.org/api/ng/service/$http#usage
we are returning the promise from our service to be handled in the controller.

To call the methods we created above, place the following code in your **main.controller.js** file:

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

    self.greeting = 'hello main controller';

    taskService.getTasks()
        .then(function(response){
```

```
            self.tasks = response.data;
        }, function(response){
            console.error('communication error');
        });

    self.createTask = function createTask(){
        taskService.createTask()
            .then(function(){
                alert('created successfully');
            }, function(){
                alert('failed to create');
            });
    }

  }]);
```

notice that we are injecting our service at the function declaration. and when the controller runs we are calling the service **getTasks** and placing all the tasks from the response at **self.tasks**. We are also creating a controller method called **createTask** that calls the creation of a task.
In the **controller template** place the following code **main.template.html:**

```
<h1>
  Tasks
</h1>
<button ng-click="ctrl.createTask();">
  Create Task
</button>
<div>
  <ul>
    <li ng-repeat="task in ctrl.tasks">
      {{task.title}}
    </li>
  </ul>
</div>
```

we are iterating here over the **tasks** list in the controller and printing the title of every task, we also created a button that will call our controller method **createTask**.

Don’t forget to include the **task-service.factory.js** in you **index.html** file

launch your app and see the tasks are listed and also try to send a post request to the server. There are also shortcut method: **$http.post** or **$http.get** which we can use to achieve the same things with a bit shorter syntax.

Pretty cool but there is an even better way to interact with a rest server using another service called **$resource** .

$resource

$resource is a factory that is used to create objects that simplify communication with a rest server. **$resource** is an abstraction around the **$http** service which means that with **$resource** you won't have to use **$http.** let's try and change the previous usage of **$http.**
Before we start **$resource** is in an external module called **ngResource** which we have to install first.

Install ngResource

let's search for the **ngResource** module. In your terminal using bower type:
> bower search ngResource

you should see that one of the result returned is **angular-resource**, let's install this package by typing:
> bower install angular-resource --save

notice that the package was added to your **bower.json** file and also to the **bower_components** folder.

In your **index.html** file right after you include the **angular.js** script add the following line to load the **js** file of **angular-resource**:

```
<script src="bower_components/angular-resource/angular-resource.js"></script>
```

When we define our **module** we need to add a dependency on the **ngResource** package, otherwise we won't be able to inject **$service** to our module components.
Change your **app.module.js** to look like this:

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

Configuring ngResource

A module is angular has a certain lifecycle of events we can bind to, one of the primal events that we can bind to before angular loads our services and controllers is the **config** event.

Angular iterates from the root module (in our example **AppModule**) down to all the modules we include in the module dependency list (the array we list when creating our module). When angular iterates on all the modules it runs the **config** events in each module and then runs the config event in our root module. The config event runs before the DI creates a single instance of our services and it's the place where we configure the services we use.
We will use the **config** event to configure **$resource** before it's created. **$resource** automatically removes trailing slashes from the url it's getting, and since our rest server needs a trailing slash we need to disable this configuration.

In your **app.module.js** change the code to look like this:

```
angular.module('AppModule', ['ngResource'])
  .config(['$resourceProvider', function($resourceProvider){
      $resourceProvider.defaults.stripTrailingSlashes = false;
  }]);
```

using the **$resourceProvider** you can change the config options of **$resource.**

Creating instance of $resource

as we mentioned, **$resource** in a factory, which means that if we call it we will get a Class of the **$resource** object.
$resource accepts 4 arguments.

- url - the url where we are sending our request. Url can have params which we specify like so: **:param.** form example the url for our update or delete api might look like so: **https://nztodo.herokuapp.com/api/task/:id/**
- **paramDefaults** - this is a key value object that will fill the params in the url, and if no param in the url then will append it as get param. Example for the url above we can create the following object: **{id: 3}**
  it will create the url with **id** replaced with **3**. or we can use this syntax: **{id: '@id'}**
  and it will search for id in the body params and will append it to the url
- **actions -** this is an object that adds additional methods to the resource instance. The keys are the methods that will be added and the values are **configuration objects** similar to the ones used for **$http**
- **options -** object of additional options that you can provide also in the **$resourceProvider** configuration.

We won't learn about **$resource** unless we will use it. First let's create abstraction around our Task object, in the **services** folder, create another folder called **models** that will hold the model blocks of our app. In the models folder create a file called: **task.factory.js** that will look like this:

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

```
    return function(dict){
        var self = this;
        var _id, _description, _title, _group, _when;

        self.getId = function getId(){
            return _id;
        }

        self.getDescription = function getDescription(){
            return _description;
        }

        self.getTitle = function getTitle(){
            return _title;
        }

        self.getGroup = function getGroup(){
            return _group;
        }

        self.getWhen = function getWhen(){
            return _when;
        }

        self.setId = function setId(id) {
            _id = id;
        }

        self.setDescription = function setDescription(description){
            _description = description;
        }

        self.setGroup = function setGroup(group){
            _group = group;
        }

        self.setWhen = function setWhen(when){
            _when = when;
        }

        self.initFromDict = function initFromDict(dict){
            _id = dict.id;
            _title = dict.title;
```

```
            _description = dict.description;
            _group = dict.group;
            _when = dict.when;
        }

        if(dict) self.initFromDict(dict);
    }
  });
```

we are simply creating our **Task** class with the fields of a task and getters and setters for those fields, the constructor function get's a dict object which initiates our fields.

Now in our **task-service.factory.js** change the factory to look like this:

```
angular.module('AppModule')
  .factory('taskService', ['$resource', 'Task', function($resource, Task){
      return new function(){
        var self = this;
        var _url = 'https://nztodo.herokuapp.com/';

        function _handleData(data, headersGetter, status){
            data = JSON.parse(data);
            if(Array.isArray){
                var res = [];
                for (var i=0; i<data.length; i++){
                    res.push(new Task(data[i]));
                }
                return res;
            } else {
                return new Task(data);
            }
        }

        function _createResource(){
            return $resource(_url + 'api/task/', {format: 'json'}, {
                get: {
                    isArray: true,
                    method: 'GET',
                    transformResponse: _handleData
                },
                save: {
                    method: 'POST',
                    transformResponse: _handleData
```

```
            }
        });
    }

    /**
     * get all the tasks from the server
     * @returns {Promise}
     */
    self.getTasks = function getTasks(){
        var res = _createResource();
        return res.get();
    }

    /**
     *
     * @returns {Promise}
     */
    self.createTask = function createTask(){
        var res = _createResource();
        var resInst = new res();
        resInst.title = 'task with resource';
        resInst.description = 'hello world';
        resInst.group = 'yariv-katz';
        resInst.when = '2016-09-01T15:10:42Z';
        return resInst.$save();
    }
  }
}]);
```

few interesting things to note here:

- We injected the **Task** class
- we created a private method that creates our **$resource** notice that when we create our **$resource** we provide a **transformResponse** to the **$resource**, it's good practice to create abstraction for the controller to use the **models** we describe and not to handle server objects. instead we are using the **_handleData** method to return the data from the server as Task objects
- we added a method to get all the tasks and return the **$resource** instance
- we added a method to create a task, notice that to create a task we need to create the instance of the **$resource** and place all the fields we want to save in that instance.

in our controller **main.controller.js** place the following code:

```
angular.module('AppModule')
```

```
  .controller('MainController', ['taskService', function(taskService){
      var self = this;

      self.greeting = 'hello main controller';

      self.tasks = taskService.getTasks();


      self.createTask = function createTask(){
          self.createdTask = taskService.createTask()
              .then(function(){
                  alert('created successfully');
              }, function(){
                  alert('failed to create');
              });
      }

  }]);
```

notice that unlike the **$http** we are now getting the tasks without the need to chain the **then**, the nice thing to note on **$resource** that it will fill the data automatically in **ctrl.tasks** when the server returns the response. we also created abstraction around the Task object so our template need to be changed to deal with Task objects.

```
<h1>
  Tasks
</h1>
<button ng-click="ctrl.createTask();">
  Create Task
</button>
<div>
  <ul>
      <li ng-repeat="task in ctrl.tasks">
          {{task.getTitle()}}
      </li>
  </ul>
</div>
```

notice that we now have to call **getTitle** to get our title.

Student Exercise

complete the **task-service.factory.js** you created before

- make sure you have the file **task.factory.js** in the models folder
- make sure to create abstraction around **task-service.factory.js** do not return the server response rather return a task instance or task instance array.
- Fill in the method in the **task-service.factory.js**
    - addTask - returns **$resource** instance
    - getTasks - returns **$resource** instance
    - getTask - returns **$resource** instance
    - updateTask - returns **$resource** instance
    - deleteTask - returns **$resource** instance

      all $**resource** instances should contain task instance

Summary

By using angular **$http** and angular **$resource** we can simplify server client communication, easily abstract our data so our controller don't need to deal with responses and avoid chaining then around our controller. Using **ngResource** will create a clean and leaner code when you ever need to communicate with your server.