# AngularJS with SpringBoot

AngularJs is a Single page application. That means we have only index.html file, but we can change the views in single page.

Index.html

<body ng-app="">

….

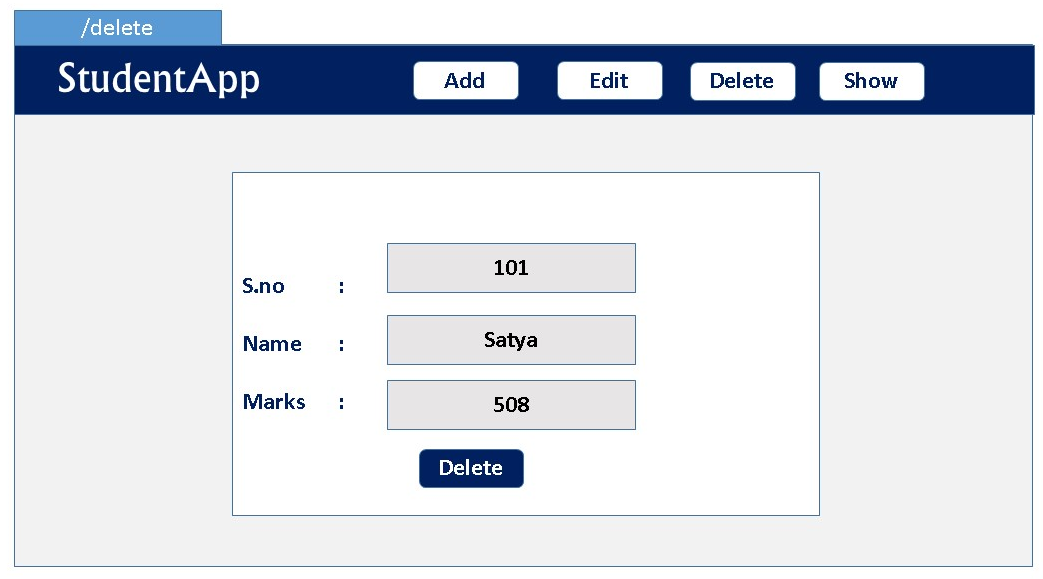
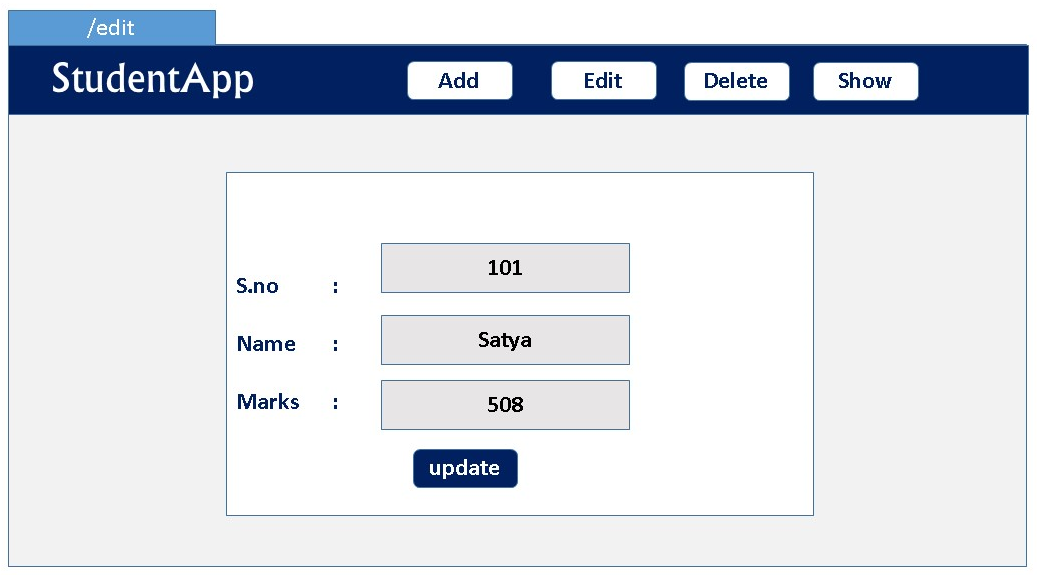
<div ng-view="">  
 // we can change the views as per controller logic display view  
 </div>

….

<body>

## SpringBoot App UI





Before going to the implementation, we need to know Basics once.

## Basics

AngularJS extends HTML attributes with **Directives**, and binds data to HTML with **Expressions**.

AngularJS is distributed as a JavaScript file, and can be added to a web page with a script tag:

<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>

* The **ng-app** directive defines an AngularJS application.
* The **ng-model** directive binds the value of HTML controls (input, select, textarea) to application data.
* The **ng-bind** directive binds application data to the HTML view
* **{{ expression }}**.  expressions bind AngularJS data to HTML the same way as the **ng-bind** directive

<div ng-app="">  
  <p>Name: <input type="text" ng-model="name"></p>  
  <p ng-bind="name"></p>

<p>{{name}} </p> //will print the same  
</div>

Understand with Example

AngularJS Example : index.html

<div ng-app="**myApp**" ng-controller="**myCtrl**">  
  
First Name: <input type="text" ng-model="firstName"><br> //binds input with model  
Last Name: <input type="text" ng-model="lastName"><br> ////binds input with model  
Full Name: {{firstName + " " + lastName}} //Prints the model values  
</div>  
  
<script>  
var app = angular.module('**myApp**', []);  
app.controller('**myCtrl**', function($scope) {  
    $scope.firstName= "John";  
    $scope.lastName= "Doe";  
});  
</script>

**1.ng-app**

* The ng-app directive tells AngularJS that this is the root element of the AngularJS application.
* All AngularJS applications must have a root element.
* You can only have one ng-app directive in your HTML document. If more than one ng-app directive appears, the first appearance will be used

In index.html. We will have to tell Angular in which part of the application it should be active.

You saw that when declaring the angular module, we named it **app**. To tell it where it should be active we add the attribute **ng-app="app"** in the tag and everything inside of it turns into an AngularJS application.

In our case, as the whole page will be an ngapp it is better to place the attribute in the <html> tag or in the <body> tag.

only one AngularJS application can be auto-bootstrapped per HTML document. The first ngApp found in the document will be used to define the root element to auto-bootstrap as an application. To run multiple applications in an HTML document you must manually bootstrap them using [angular.bootstrap](https://docs.angularjs.org/api/ng/function/angular.bootstrap) instead.

**2.ng-controller**

The ngController directive specifies a Controller class; the class contains business logic behind the application to decorate the scope with functions and values.

In above '**myCtrl**'has following business logic

app.controller('**myCtrl**', function($scope) {  
    $scope.firstName= "John";  
    $scope.lastName= "Doe";  
});

**3.angular.module**

The angular.module is a global place for creating, registering and retrieving AngularJS modules. All modules (AngularJS core or 3rd party) that should be available to an application must be registered using this mechanism.

var app = angular.module('**myApp**', []);

in this line we registred the module with variable ‘app’. Now we can access this module in whole application with variable name to perform any kind of operations.

A module is a collection of services, directives, controllers, filters, and configuration information. angular.module is used to configure the [$injector](https://docs.angularjs.org/api/auto/service/$injector).

// Create a new module

var myModule = angular.module('myModule', []);

// register a new service

myModule.value('appName', 'MyCoolApp');

// configure existing services inside initialization blocks.

myModule.config(['$locationProvider', function($locationProvider) {

// Configure existing providers

$locationProvider.hashPrefix('!');

}]);

**3.$scope**

* $scope is a Global Object which can accessed by both view and controller.
* When adding properties to the $scope object in the controller, the view (HTML) gets access to these properties
* All applications have a $rootScope which is the scope created on the HTML element that contains the ng-app directive.
* The rootScope is available in the entire application.

## AngularJs internal working

Now you will take a look at the architecture concepts of AngularJS. When an HTML document is loaded into the browser and is evaluated by the browser, the following happens:

1. The **AngularJS JavaScript** file is loaded, and the Angular global object$scope  is created. The JavaScript file that registers the controller functions is executed.
2. AngularJS scans the HTML to look for **AngularJS apps and views** and finds **a controller function corresponding to the view.**
3. AngularJS **executes the controller functions** **and updates the views** with data from the model populated by the controller.
4. AngularJS listens for browser events, such as button clicked, mouse moved, input field being changed, and so on. If any of these events happen, then AngularJS will update the view accordingly

Bootstrapping AngularJS by Adding ng-app in an HTML Page

<html lang="en" ng-app="userregistrationsystem">...</html>

This is also known as automatic initialization. So, when AngularJS finds the **ng-app** directive after analyzing the **index.html** file, it loads the associated modules and then compiles the DOM.

## Few More Examples

AngularJS Example : ng-init, **{{ expression }}** = ng-bind

<div ng-app="" ng-init="firstName='John';lastName='Doe'">  
 <p>The name is {{ firstName + " " + lastName }}</p>

<p>The name is <span ng-bind="firstName + ' ' + lastName"></span></p>  
</div>

#### AngularJS Objects

AngularJS objects are like JavaScript objects:

<div ng-app="" ng-init="person={firstName:'John',lastName:'Doe'}">  
 <p>The name is {{ person.lastName }}</p> //or

<p>The name is <span ng-bind="person.lastName"></span></p>  
</div>

#### AngularJS Arrays

AngularJS arrays are like JavaScript arrays:

<div ng-app="" ng-init="points=[1,15,19,2,40]">  
 <p>The third result is {{ points[2] }}</p>  
</div>

#### AngularJS Module

* Creating a Module

<div ng-app="myApp">...</div>  
<script>  
 var app = angular.module("myApp", []);   
</script>

* Adding a Controller

<script>  
  
var app = angular.module(**"myApp"**, []);  
  
app.controller(**"myCtrl"**, function($scope) {  
  $scope.firstName = "John";  
  $scope.lastName = "Doe";  
});  
  
</script>

#### AngularJs Validation

The ng-model directive can provide type validation for application data (number, e-mail, required):

<form ng-app="" name="myForm">  
  Email:  
  <input type="email" name="myAddress" ng-model="text">  
  <span ng-show="myForm.myAddress.$error.email">Not a valid e-mail address</span>  
</form>

#### AngularJs Controllers – with Different Data

* Inside another function

<script>  
var app = angular.module('myApp', []);  
app.controller('personCtrl', function($scope) {  
  $scope.firstName = "John";  
  $scope.lastName = "Doe";  
  **$scope.fullName = function()** {  
    return $scope.firstName + " " + $scope.lastName;  
  };  
});  
</script>

* With Array data

File : namesController.js

angular.module('myApp', []).controller('namesCtrl', function($scope) {  
  $scope.names = [  
    {name:'Jani',country:'Norway'},  
    {name:'Hege',country:'Sweden'},  
    {name:'Kai',country:'Denmark'}  
  ];  
});

<div ng-app="myApp" ng-controller="namesCtrl">  
<ul>  
  <li ng-repeat="x in names">  
    {{ x.name + ', ' + x.country }}  
  </li>  
</ul>  
</div>  
<script src="namesController.js"></script>

#### AngularJS Filters

AngularJS provides filters to transform data:

* uppercase Format a string to upper case.

<p>The name is {{ lastName | uppercase }}</p>

* lowercase Format a string to lower case.

<p>The name is {{ lastName | lowercase }}</p>

* currency Format a number to a currency format.

<h1>Price: {{ price | currency }}</h1> //output is : **Price: $58.00**

* filter Select a subset of items from an array.

This example displays only the names containing the letter "i".

<ul>  
  <li ng-repeat="x in names | filter : 'i'">  
    {{ x }}  
  </li>  
</ul>

* orderBy Orders an array by an expression.

<ul>  
  <li ng-repeat="x in names | orderBy:'country'">  
    {{ x.name + ', ' + x.country }}  
  </li>  
</ul>

* json Format an object to a JSON string.
* date Format a date to a specified format.
* number Format a number to a string.
* limitTo Limits an array/string, into a specified number of elements/characters.

#### AngularJS Services

* In AngularJS, a service is a **function, or object**, that is available for your AngularJS application.
* AngularJS has about 30 built-in services. One of them is the $location service.

1.$location

returns information about the location of the current web page:

var app = angular.module('myApp', []);  
app.controller('customersCtrl', function($scope, $location) {  
    $scope.myUrl = $location.absUrl();  
});

{{myUrl}} //prints https://www.w3schools.com/angular/tryit.asp?filename=try\_ng\_services

$2.$http

**$http** is an AngularJS service for reading data from remote servers. The AngularJS $http service makes a request to the server, and returns a response.

var app = angular.module('myApp', []);  
app.controller('myCtrl', function($scope, $http) {  
  $http.get("welcome.htm").then(function (response) {  
    $scope.myWelcome = response.data;  
  });  
});

More on Http

The example above uses the .get method of the $http service.

The .get method is a shortcut method of the $http service. There are several shortcut methods:

* .get()
* .post()
* .put()
* .delete()
* .head()
* .jsonp()
* .patch()

var app = angular.module('myApp', []);  
app.controller('myCtrl', function($scope, $http) {  
  $http({  
    method : "GET",  
      url : "welcome.htm"  
  }).then(function mySuccess(response) {  
    $scope.myWelcome = response.data;  
  }, function myError(response) {  
    $scope.myWelcome = response.statusText;  
  });  
});

Responnse Types

* .config the object used to generate the request.
* .data a string, or an object, carrying the response from the server.
* .headers a function to use to get header information.
* .status a number defining the HTTP status.
* .statusText a string defining the HTTP status.

<div ng-app="myApp" ng-controller="myCtrl">

<p>Data : {{content}}</p>

<p>Status : {{statuscode}}</p>

<p>StatusText : {{statustext}}</p>

</div>

var app = angular.module('myApp', []);  
app.controller('myCtrl', function($scope, $http) {  
  $http.get("welcome.htm")  
  .then(function(response) {  
    $scope.content = response.data;  
    $scope.statuscode = response.status;  
    $scope.statustext = response.statusText;   
  });  
});

Data : Hello AngularJS Students

Status : 200

StatusText :

$timeout

The $timeout service is AngularJS' version of the window.setTimeout function.

var app = angular.module('myApp', []);  
app.controller('myCtrl', function($scope, $timeout) {  
  $scope.myHeader = "Hello World!";  
  $timeout(function () {  
    $scope.myHeader = "How are you today?";  
  }, 2000);  
});

#### AngularJs DOM Elements

Like in JavaScript we can hide,show,disable DOM elements

<button disabled>Click Me!</button> //to Disable button

* Hide/show

<p ng-show="true">I am visible.</p>  
<p ng-show="false">I am not visible.</p>

<p ng-hide="true">I am not visible.</p>  
<p ng-hide="false">I am visible.</p>

#### AngularJS Events

You can add AngularJS event listeners to your HTML elements by using one or more of these directives:

* ng-blur
* ng-change
* ng-click
* ng-copy
* ng-cut
* ng-dblclick
* ng-focus
* ng-keydown
* ng-keypress
* ng-keyup
* ng-mousedown
* ng-mouseenter
* ng-mouseleave
* ng-mousemove
* ng-mouseover
* ng-mouseup
* ng-paste

Increase the count variable when the mouse clicked.

<div ng-app="myApp" ng-controller="myCtrl">  
 <button ng-click="count = count + 1">Click me!</button>  
 <p>{{ count }}</p>  
</div>

//At Start count is 0 once Js load  
<script>  
var app = angular.module('myApp', []);  
app.controller('myCtrl', function($scope) {  
  $scope.count = 0;  
});  
</script>

Same, replace with function call

<div ng-app="myApp" ng-controller="myCtrl">  
  
<button ng-click="myFunction()">Click me!</button>  
  
<p>{{ count }}</p>  
  
</div>  
<script>  
var app = angular.module('myApp', []);  
app.controller('myCtrl', function($scope) {  
  $scope.count = 0;  
**$scope.myFunction = function() {  
    $scope.count++;  
  }**  
});  
</script>

#### AngularJS Includes

With **ng-include** directive, you can include HTML from an external file.

<div ng-include="'myFile.htm'"></div>

#### AngularJS Routing

The ngRoute module helps your application to become a Single Page Application.

* If you want to navigate to different pages in your application, but you also want the application to be a SPA (Single Page Application), with no page reloading, you can use the ngRoute module.
* The ngRoute module routes your application to different pages without reloading the entire application.

**For doing this**

* you must include the AngularJS Route module JS

<script src="https://ajax.googleapis.com/angular-route.js"></script>

* Then you must add the ngRoute as a dependency in the application module:

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

* Now your application has access to the route module, which provides the $routeProvider. Use the $routeProvider to **configure(app.config)** different routes in your application:

app.config(function($routeProvider) {  
  $routeProvider  
  .when("/", {  
    templateUrl : "main.htm"  
  })  
  .when("/red", {  
    templateUrl : "red.htm"  
  })  
  .when("/green", {  
    templateUrl : "green.htm"  
  })  
  .when("/blue", {  
    templateUrl : "blue.htm"  
  });  
});

Example

<body ng-app="myApp">  
<a href="#/!">Main</a></p>  
<a href="#!red">Red</a>  
<a href="#!green">Green</a>  
<a href="#!blue">Blue</a>  
  
<div ng-view>

<! —HERE CONTENT CHANGES -->

</div>  
  
<script>  
var app = angular.module("myApp", ["ngRoute"]);

app.config(function($routeProvider) {  
  $routeProvider  
  .when("/", {  
    templateUrl : "main.htm"  
  })  
  .when("/red", {  
    templateUrl : "red.htm"  
  })  
  .when("/green", {  
    templateUrl : "green.htm"  
  })  
  .when("/blue", {  
    templateUrl : "blue.htm"  
  });  
});  
</script>  
</body>

1.ng-view

Your application needs a container to put the **content** provided by the routing.This container is the ng-view directive. There are three different ways to include the ng-view directive in your application:

1. <div ng-view></div>
2. <div class="ng-view"></div>
3. <ng-view></ng-view>

Applications can only have one ng-view directive, and this will be the placeholder for all views provided by the route.

2.routeProvider

With the $routeProvider you can also define a controller for each "view".

var app = angular.module("myApp", ["ngRoute"]);  
app.config(function($routeProvider) {  
  $routeProvider  
  .when("/", {  
    templateUrl : "main.htm"  
  })  
  .when("/london", {  
    templateUrl : "london.htm",  
    controller : "londonCtrl"  
  })  
  .when("/paris", {  
    templateUrl : "paris.htm",  
    controller : "parisCtrl"  
  });  
});  
**app.controller("londonCtrl", function ($scope) {  
  $scope.msg = "I love London";  
});  
app.controller("parisCtrl", function ($scope) {  
  $scope.msg = "I love Paris";  
});**

3.template

In the previous examples we have used the templateUrl property in the $routeProvider.when method. You can also use the template property, which allows you to **write HTML directly in the property value**, and not refer to a page.

var app = angular.module("myApp", ["ngRoute"]);  
app.config(function($routeProvider) {  
  $routeProvider  
  .when("/", {  
    template : "<h1>Main</h1><p>Click on the links to change this content</p>"  
  })  
  .when("/banana", {  
    template : "<h1>Banana</h1><p>Bananas contain around 75% water.</p>"  
  })  
  .when("/tomato", {  
    template : "<h1>Tomato</h1><p>Tomatoes contain around 95% water.</p>"  
  });  
});

4. otherwise

In the previous examples we have used the when method of the $routeProvider.You can also use the otherwise method, which is the default route when none of the others get a match.

If use not clicked **/banana,/tamoto** for example user clicks **/apple** then otherwise will do the job.

var app = angular.module("myApp", ["ngRoute"]);  
app.config(function($routeProvider) {  
  $routeProvider  
  .when("/banana", {  
    template : "<h1>Banana</h1><p>Bananas contain around 75% water.</p>"  
  })  
  .when("/tomato", {  
    template : "<h1>Tomato</h1><p>Tomatoes contain around 95% water.</p>"  
  })  
  .otherwise({  
    template : "<h1>None</h1><p>Nothing has been selected</p>"  
  });  
});

## AngularJS Architecture Concepts

Now you will take a look at the architecture concepts of AngularJS. When an HTML document is loaded into the browser and is evaluated by the browser, the following happens:

1. The **AngularJS JavaScript** file is loaded, and the Angular global object$scope  is created. The JavaScript file that registers the controller functions is executed.
2. AngularJS scans the HTML to look for **AngularJS apps and views** and finds **a controller function corresponding to the view.**
3. AngularJS **executes the controller functions** **and updates the views** with data from the model populated by the controller.
4. AngularJS listens for browser events, such as button clicked, mouse moved, input field being changed, and so on. If any of these events happen, then AngularJS will update the view accordingly

Bootstrapping AngularJS by Adding ng-app in an HTML Page

<html lang="en" ng-app="userregistrationsystem">...</html>

This is also known as automatic initialization. So, when AngularJS finds the **ng-app** directive after analyzing the **index.html** file, it loads the associated modules and then compiles the DOM.

The AngularJS application **UserRegistrationSystem** is defined as the AngularJS module (**angular.module**) in app.js, which is the entry point into the application.

var app = angular.module('userregistrationsystem', ['ngRoute', 'ngResource']);

As you can see, the two dependencies have been defined in **app.js** as needed by **userregistrationsystem** at startup. The two dependencies in the previous code are defined in an array in the module definition.

* **ngRoute:** The first dependency is the AngularJS **ngRoute** module, which provides routing to the application. The ngRoute module is used for deep-linking URLs to controllers and views.
* **ngResource:** The second dependency is the AngularJS **ngResource** module, which provides interaction support with RESTful services

AngularJS Routes

AngularJS routes are configured using them **$routeProvider** API. Routes are dependent on the ngRoute module, which is why its dependency is defined in an array in the module definition.

app.js. You will define four routes in your AngularJS application.

* The first is /list-all-users.
* The second one is /register-new-user
* The third one is /update-user/:id
* The fourth one is different from the other three

**var** app = angular.module('userregistrationsystem', [ 'ngRoute', 'ngResource' ]);

app.config(**function**($routeProvider) {

$routeProvider.when('/list-all-users', {

templateUrl : '/template/listuser.html',

controller : 'listUserController'

}).when('/register-new-user',{

templateUrl : '/template/userregistration.html',

controller : 'registerUserController'

}).when('/update-user/:id',{

templateUrl : '/template/userupdation.html' ,

controller : 'usersDetailsController'

}).otherwise({

redirectTo : '/home',

templateUrl : '/template/home.html',

});

});

* When the user clicks the link in the application specified at http://localhost:8080/#/list-all-users, the /list-all-users route will be followed, and the content associated with the /list-all-users URL will be displayed

in app.config , each route is mapped to a template and controller (optional).

* The controller listUserController will be called when you navigate to the URL /list-all-users
* controller registerUserController will be called when you navigate to the URL /register-new-user

AngularJS Templates

* AngularJS templates, also known as HTML partials, are HTML code that are bound to the <div ng-view> </div> tag shown in the index.html file. If you look at the code from the
* app.js file, you can see that different templateUrl values are defined for different routes, as shown in above code
* The listuser.html, userregistration.html , userupdation.html , and home.html pages are four different partials or templates, which contain HTML code and AngularJS’s built-in template language to display dynamic data in your template.

<html lang="en" ng-app="userregistrationsystem">

<head>

<title>Full Stack Development</title>

<link rel="stylesheet" href="/css/app.css">

</head>

<body>

<div **class**="page-header text-center">

<h2>User Registration System</h2>

</div>

<nav **class**="navbar navbar-default">

<div **class**="container-fluid">

<a href="#/" **class**=" navbar-btn" role="button">Home</a>

<a href="#/register-new-user" role="button"> Register New User</a>

<a href="#/list-all-users" **class**="" role="button">List All Users</a>

</div>

</nav>

<div ng-view></div>

<script src="/webjars/angularjs/1.4.9/angular.js"></script>

<script src="/webjars/angularjs/1.4.9/angular-resource.js"></script>

<script src="/webjars/angularjs/1.4.9/angular-route.js"></script>

<script src="/js/app.js"></script>

<script src="/js/controller.js"></script>

<link rel="stylesheet"

href="/webjars/bootstrap/3.3.6/css/bootstrap.css">

</body>

</html>

You have included a separate app.js, which is where you will be defining the application behavior.

Let’s create four view pages in the template folder inside the src/main/resources/static directory.

**•Home page :src/main/resources/template/home.html**

•**Register New User page :src/main/resources/template/userregistration.html**

<form ng-submit="submitUserForm()" name="myForm" **class**="form-horizontal">

<input type="text" ng-model="user.name" id="uname" />

<input type="text" ng-model="user.address" id="address"/>

<input type="email" ng-model="user.email" id="email" />

<input type="submit" value="Register User"/>

<button type="button" ng-click="resetForm()">Reset Form</button>

</form>

•List Of User page :**src/main/resources/template/listuser.html**

<table class="table table-hover table-bordered ">

<thead>

<tr>

<th>Name</th>

<th>Email</th>

<th>Address</th>

<th width="100">Edit</th>

<th width="100">Delete</th>

</tr>

</thead>

<tbody>

<tr ng-repeat="user in users">

<td>{{user.name}}</td>

<td>{{user.email}}</td>

<td>{{user.address}}</td>

<td>

<button type="button" ng-click="editUser(user.id)"> Edit</button>

</td>

<td>

<button type="button" ng-click="deleteUser(user.id)"> Delete</button></td>

</tr>

</tbody>

</table>

AngularJS Controller

controller.js file defined in the **src/main/resources/static/js** folder contains the implementation of AngularJS controllers

* -On a successful POST call, it will redirect to list-all-users
* -$scope is used to set up dynamic content for the UI elements that this controller is responsible for
* -$http: The $http service is the core feature provided by AngularJS and is used to consume the REST

app.controller('registerUserController', **function**($scope, $http, $location,

$route) {

$scope.submitUserForm = **function**() {

$http({

method : 'POST',

url : 'http://localhost:8080/api/user/',

data : $scope.user,

}).then(**function**(response) {

$location.path("/list-all-users");

$route.reload();

}, **function**(errResponse) {

$scope.errorMessage = errResponse.data.errorMessage;

});

}

$scope.resetForm = **function**() {

$scope.user = **null**;

};

});

## AngularJs + SpringBoot Example implementation

Index.html

* Create index.html & add all Anuglar related jars
* Declare body ng-app=*"stApp"*> which will be the Startig point of our AngularJs App
* Create module

var app = angular.module("myApp", []);  
if no dependencies are there [ ] will be empty

<body ng-app=*"stApp"*>

<nav class=*"navbar navbar-inverse"*>

ADD, EDIT, DELETE etc

</nav>

<div class=*"container"*>

<h1>Student Application</h1>

</div>

</body>

</html>

## References

<https://www.slideshare.net/lochnguyen/angular-js-for-java-developers-40120787>

<https://medium.com/@swhp/build-single-page-application-with-java-ee-and-angularjs-4eaacbdfcd>

<https://examples.javacodegeeks.com/desktop-java/ide/eclipse/eclipse-ide-angularjs-tutorial/>

<https://www.webcodegeeks.com/download/16411/?dlm-dp-dl-force=1&dlm-dp-dl-nonce=33f2636c9a>

# JUNIT

## 1.JUNIT Introduction

Testing is the process of checking the functionality of an application to ensure it runs as per requirements. **Unit testing** comes into picture at the developers’ level; it is the testing of single entity **(class or method).** Unit testing can be done in two ways –**manual testing & automated testing**

**1. Manual Testing:** If you execute the test cases manually without any tool support, it is known as manual testing. It is time consuming and less reliable.

**2. Automated Testing: If** you execute the test cases by tool support, it is known as automated testing. It is fast and more reliable.

**XUnit** architecture **introduced automated unit testing**. There are many unit testing frame works for different programming. Few of the unit testing frame works are:

* JAVA - **JUnit**,
* C - **CUnit**,
* C++ - **CPPUnit**,
* .NET - .**NUnit** etc..

XUnit architecture was first implemented for java.**It is known as JUnit**

JUnit

It is an open-source testing framework for java programmers. The java programmer can create test cases and test his/her own code.It is one of the unit testing framework. Current version is **junit 5**.you can download it from [**JUnit website (github)**](https://github.com/junit-team/junit4/wiki/Download-and-Install) or use below maven dependency in your **pom.xml**

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

<scope>test</scope>

</dependency>

The Junit 4.x framework is annotation based. Here’re some basic JUnit annotations

* **@Test -**Given method is the test method.= public void
* **@Test(timeout=1000)** -method will be failed if it takes more then 1000 milliseconds (1 sec).
* **@BeforeClass** - method will be invoked only once, before starting all the tests. public static void
* **@AfterClass** -method will be invoked only once, after finishing all the tests public static void
* **@Before** -method will be invoked before each test. Run before @Test, public void
* **@After** - method will be invoked after each test. Run after @Test, public void

The most important package in JUnit is **junit.framework**, which contains all the core classes. Some of the most important classes are given below

**1.Assert** - set of assert methods.

**2.TestCase** - It is the testing of single entity (class or method)

**3.TestSuite** - A test suite bundles a few unit test cases and runs them together.

**3.TestResult** - Contains methods to collect the results of executing a test case.

REMEMBER: If Junit tests not Running means

By default Maven uses the following naming conventions when looking for tests to run:

* **Test\***
* **\*Test**
* **\*TestCase**

**1.Assert**

The org.junit.Assert class provides methods to assert the program logic. Assert methods are usually used to **compare the actual value with the expected value**. All assert methods are static methods. Return type of all assert methods are void

1. **void assertEquals(boolean expected,boolean actual)**: checks that two primitives/objects are equal. It is overloaded.
2. **void assertTrue(boolean condition)**: checks that a condition is true.
3. **void assertFalse(boolean condition)**: checks that a condition is false.
4. **void assertNull(Object obj)**: checks that object is null.
5. **void assertNotNull(Object obj)**: checks that object is not null.

**2.Test Case**

It is the testing of single entity (class or method)

public class TestJunit1 {

String message = "Robert";

MessageUtil messageUtil = new MessageUtil(message);

@Test

public void testPrintMessage() {

System.out.println("Inside testPrintMessage()");

assertEquals(message, messageUtil.printMessage());

}

}

**3.Test Suites**

A test suite bundles a few unit test cases and runs them together.

import org.junit.runner.RunWith;

import org.junit.runners.Suite;

//JUnit Suite Test

@RunWith(Suite.class)

@Suite.SuiteClasses({

TestJunit1.class ,TestJunit2.class

})

public class JunitTestSuite {

}

**4.Test Runners**

Test runner is used for executing the test cases.

public class TestRunner {

public static void main(String[] args) {

Result result = JUnitCore.runClasses(TestJunit.class);

for (Failure failure : result.getFailures()) {

System.out.println(failure.toString());

}

System.out.println(result.wasSuccessful());

}

}

**TestResult(**Result**)** − Contains methods to collect the results of executing a test case.

Example 1 : Testing Annotations Working

**package** basic;

**import** org.junit.\*;

**public** **class** AnnotationsTest {

// Run once, e.g. Database connection, connection pool

@BeforeClass

**public** **static** **void** runOnceBeforeClass() {

System.***out***.println("@BeforeClass - runOnceBeforeClass");

}

// Run once, e.g close connection, cleanup

@AfterClass

**public** **static** **void** runOnceAfterClass() {

System.***out***.println("@AfterClass - runOnceAfterClass");

}

// Should rename to @BeforeTestMethod

// e.g. Creating an similar object and share for all @Test

@Before

**public** **void** runBeforeTestMethod() {

System.***out***.println("@Before - runBeforeTestMethod");

}

// Should rename to @AfterTestMethod

@After

**public** **void** runAfterTestMethod() {

System.***out***.println("@After - runAfterTestMethod");

}

@Test

**public** **void** TestMethod1() {

System.***out***.println("@Test - TestMethod1");

}

@Test

**public** **void** TestMethod2() {

System.***out***.println("@Test - TestMethod2");

}

}

@BeforeClass – runOnceBeforeClass

@Before - runBeforeTestMethod

@Test - TestMethod1

@After – runAfterTestMethod

@Before - runBeforeTestMethod

@Test - TestMethod2

@After – runAfterTestMethod

@AfterClass - runOnceAfterClass

Note: All sources of production code commonly reside in the src/main/java directory, while all test source files are kept at src/test/java

## 2. JUnit Hello World!

To write testcases we must figute out below points

1. ***Class to be tested***
2. ***Write Testcases for selected class***
3. ***Run the Test (Commandline / TestRunner class)***

1. Class to be tested

**package** junit;

**public** **class** **Calculator** {

**public** **int** square(**int** x){

**return** x\*x;

}

}

2. Write Testcases for selected class

**package** junit;

**import** **static** org.junit.Assert.\*;

**import** org.junit.Test;

**public** **class** CalculatorTest {

@Test

**public** **void** squareTest() {

Calculator calculator = **new** Calculator();

**int** sqr = calculator.square(2);

//Checking for 2

*assertEquals*("2\*2=4 Passed",4, sqr);//pass

*assertEquals*("2\*2=4 Passed",6, sqr); //Fail

}

}

3. Run the Test (Commandline / TestRunner class)

**Using command line**

**java -cp .;junit-4.XX.jar;hamcrest-core-1.3.jar org.junit.runner.JUnitCore CalculatorTest**

**Using TestRunner class**

**package** junit;

**import** org.junit.runner.\*;

**public** **class** TestRunner {

**public** **static** **void** main(String[] args) {

Result result = JUnitCore.*runClasses*(CalculatorTest.**class**);

**for** (Failure failure : result.getFailures()) {

System.***out***.println("Failure : " + failure.toString());

}

System.***out***.println("Success : " + result.wasSuccessful());

}

}

Failure : squareTest(junit.CalculatorTest): 2\*2=4 Passed expected:<6> but was:<4>

Success : false //for **assertEquals("2\*2=4 Passed",6, sqr); //Fail**

Success : true //for ***assertEquals*("2\*2=4 Passed",4, sqr);//pass**

## 4. JUnit Examples

## 4.1 Assert all Methods Example

**import** **static** org.hamcrest.CoreMatchers.*\**;

**import** **static** org.junit.Assert.*\**;

**import** java.util.Arrays;

**import** org.hamcrest.core.CombinableMatcher;

**import** org.junit.Test;

**public** **class** AssertTests {

@Test

**public** **void** testAssertArrayEquals() {

**byte**[] expected = "trial".getBytes();

**byte**[] actual = "trial".getBytes();

*assertArrayEquals*("failure - byte arrays not same", expected, actual);

}

@Test **public** **void** testAssertEquals() {

*assertEquals*("failure - strings are not equal", "text", "text");

}

@Test **public** **void** testAssertFalse() {

*assertFalse*("failure - should be false", **false**);

}

@Test **public** **void** testAssertNotNull() {

*assertNotNull*("should not be null", **new** Object());

}

@Test **public** **void** testAssertNotSame() {

*assertNotSame*("should not be same Object", **new** Object(), **new** Object());

}

@Test **public** **void** testAssertNull() {

*assertNull*("should be null", **null**);

}

@Test **public** **void** testAssertSame() {

Integer aNumber = Integer.*valueOf*(768);

*assertSame*("should be same", aNumber, aNumber);

}

// JUnit Matchers assertThat

@Test **public** **void** testAssertThatBothContainsString() {

*assertThat*("albumen", *both*(*containsString*("a")).and(*containsString*("b")));

}

@Test **public** **void** testAssertThatHasItems() {

*assertThat*(Arrays.*asList*("one", "two", "three"), *hasItems*("one", "three"));

}

@Test **public** **void** testAssertThatEveryItemContainsString() {

*assertThat*(Arrays.*asList*(**new** String[] { "fun", "ban", "net" }), *everyItem*(*containsString*("n")));

}

// Core Hamcrest Matchers with assertThat

@Test **public** **void** testAssertThatHamcrestCoreMatchers() {

*assertThat*("good", *allOf*(*equalTo*("good"), *startsWith*("good")));

*assertThat*("good", *not*(*allOf*(*equalTo*("bad"), *equalTo*("good"))));

*assertThat*("good", *anyOf*(*equalTo*("bad"), *equalTo*("good")));

*assertThat*(7, *not*(CombinableMatcher.<Integer> *either*(*equalTo*(3)).or(*equalTo*(4))));

*assertThat*(**new** Object(), *not*(*sameInstance*(**new** Object())));

}

@Test **public** **void** testAssertTrue() {

*assertTrue*("failure - should be true", **true**);

}

}

## 4.2 Test Suite

**Test suite** is used to bundle a few unit test cases and run them together. **@RunWith and @Suite** annotations are used to run the suite tests.

In below Example we are running Test1 & Test2 together using Test Suite.

1. Class to be tested

**package** testsuite;

**public** **class** Calculator {

**public** **int** square(**int** x) {

**return** x \* x;

}

**public** **int** sum(**int** x, **int** y) {

**return** x + y;

}

}

2. Write Testcases for selected class

**//1.Test1.java**

**package** testsuite;

**import** **static** org.junit.Assert.*assertEquals*;

**import** org.junit.Test;

**public** **class** Test1 {

@Test

**public** **void** squareTest() {

Calculator calculator = **new** Calculator();

**int** sqr = calculator.square(2);

*assertEquals*("2\*2=4 Passed",4, sqr);

}

}

-----------------------------------------------------------------------------------------------------

**//2.Test2.java**

**package** testsuite;

**import** **static** org.junit.Assert.*assertEquals*;

**import** org.junit.Test;

**public** **class** Test2 {

@Test

**public** **void** addTest() {

Calculator calculator = **new** Calculator();

**int** sum = calculator.sum(8,2);

*assertEquals*("8+2=10 Passed",10, sum);

}

}

3.Create Test Suite Class

**package** testsuite;

**import** org.junit.runner.RunWith;

**import** org.junit.runners.Suite;

@RunWith(Suite.**class**)

@Suite.SuiteClasses({

Test1.**class**,

Test2.**class**

})

**public** **class** CalculatorTestSuite {

}

4. Run the Test (Commandline / TestRunner class)

**Remember, TestRunner class is same for all Examples**

**package** testsuite;

**import** org.junit.runner.JUnitCore;

**import** org.junit.runner.Result;

**import** org.junit.runner.notification.Failure;

**public** **class** TestRunner {

**public** **static** **void** main(String[] args) {

Result result = JUnitCore.*runClasses*(CalculatorTestSuite.**class**);

**for** (Failure failure : result.getFailures()) {

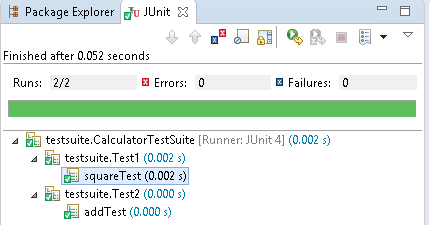
System.***out***.println("Failure : " + failure.toString());

}

System.***out***.println("Success : " + result.wasSuccessful());

}

}



## 4.3 Ignore Test

**@Ignore** annotation is used to create Ignore test. We use @ignore in two cases

**1. At Method Level** : if a method annotated with @Ignore, **that method will not be executed.**

**2. At Class Level Level** : if a class annotated with @Ignore**, all its methods will not be executed.**

1. Class to be tested

**package** ignoretest;

**public** **class** IgnoreTestClassLevel {

**private** String str1;

**private** String str2;

**private** String str3;

**public** IgnoreTestClassLevel(String str1, String str2) {

**this**.str1 = str1;

**this**.str2 = str2;

}

**public** String addStrings() {

str3 = str1 + str2;

System.***out***.println("addStrings : " + str3);

**return** str1 + str2;

}

**public** String upperCase() {

str3 = (str1 + str2).toUpperCase();

System.***out***.println("upperCase : " + str3);

**return** str1 + str2;

}

}

2. Ignore Test at Method Level

**package** ignoretest;

**import** org.junit.Test;

**import** org.junit.Ignore;

**import** **static** org.junit.Assert.*assertEquals*;

**public** **class** IgnoreTestMethodLevel {

StringUtil util = **new** StringUtil("a", "b");

String res = "";

@Ignore

@Test

**public** **void** testAddStrings() {

System.***out***.println("Inside testAddStrings()");

res = "ab";

*assertEquals*(res, util.addStrings());

}

@Test

**public** **void** testUpperCase() {

System.***out***.println("Inside testUpperCase()");

res = "AB";

*assertEquals*(res, util.upperCase());

}

}

Inside testUpperCase()

upperCase : AB

Success : true

3. Ignore Test at Class Level

**package** ignoretest;

**import** org.junit.Test;

**import** org.junit.Ignore;

**import** **static** org.junit.Assert.*assertEquals*;

@Ignore

**public** **class** IgnoreTestClassLevel {

StringUtil util = **new** StringUtil("a", "b");

String res = "";

@Test

**public** **void** testAddStrings() {

System.***out***.println("Inside testAddStrings()");

res = "ab";

*assertEquals*(res, util.addStrings());

}

@Test

**public** **void** testUpperCase() {

System.***out***.println("Inside testUpperCase()");

res = "AB";

*assertEquals*(res, util.upperCase());

}

}

Empty Output, because none of its test methods will be executed.

## 4.4 Time Test

**@Test(timeout)-** **timeout** parameter along with @Test annotation as used for Time Test.If a test case takes more time than the specified number of milliseconds, then JUnit will automatically mark it as failed.

Example: Time Test example for above StringUtil.java Class

**package** ignoretest;

**import** org.junit.Test;

**import** org.junit.Ignore;

**import** **static** org.junit.Assert.*assertEquals*;

**public** **class** StringUtilTimeTest {

StringUtil util = **new** StringUtil("a", "b");

String res = "";

**@Test(timeout = 1000)**

**public** **void** testAddStrings() {

System.***out***.println("Inside testAddStrings()");

res = "ab";

*assertEquals*(res, util.addStrings());

}

@Test

**public** **void** testUpperCase() {

System.***out***.println("Inside testUpperCase()");

res = "AB";

*assertEquals*(res, util.upperCase());

}

}

## 4.5 Exceptions Test

**@Test(expected)-** **expected** parameter along with @Test annotation as used for Exceptions Test. we can test whether our code throws an expected exception or not

Example: ExceptionsTest

**package** junit;

**public** **class** Calculator {

**public** **int** square(**int** x) {

**return** x \* x;

}

**public** **int** div(**int** a, **int** b) {

**return** a / b;

}

}

**package** junit;

**import** **static** org.junit.Assert.\*;

**import** org.junit.Test;

**public** **class** CaluculatorExceptionTest {

@Test(expected = ArithmeticException.**class**)

**public** **void** divTest() {

Calculator calculator = **new** Calculator();

**int** res = calculator.div(12,0); //success

// int res = calculator.div(12,0); //error

*assertEquals*(4, res);

}

}

## 4.6 Parameterized Test

Parameterized tests allow a developer to run the same test over and over again using different values.we use **@RunWith(Parameterized.class)** to achive this type of tests.

Example

**public** **class** EvenNumbers {

**public** Boolean checkEven(**final** Integer num) {

**for** (**int** i = 1; i <= num; i++) {

**if** (i % 2 == 0) {

**return** **true**;

}

}

**return** **false**;

}

}

**package** parameterizedtest;

**import** java.util.\*;

**import** org.junit.\*;

**import** **static** org.junit.Assert.*assertEquals*;

@RunWith(Parameterized.**class**)

**public** **class** PrimeNumberCheckerTest {

**private** Integer inum;

**private** Boolean res;

**private** EvenNumbers evenObj;

@Before

**public** **void** initialize() {

evenObj = **new** EvenNumbers();

}

**public** PrimeNumberCheckerTest(Integer inum, Boolean res) {

**this**.inum = inum;

**this**.res = res;

}

@Parameterized.Parameters

**public** **static** Collection evenNumbers() {

**return** Arrays.*asList*(**new** Object[][] {

{ 2, **true** },

{ 6, **true** },

{ 18, **true** },

{ 19, **false** },

{ 48, **true** }

});

}

@Test

**public** **void** testPrimeNumberChecker() {

System.***out***.println("Parameterized Number is : " + inum);

*assertEquals*(res, evenObj.checkEven(inum));

}

}

Parameterized Number is : 2

Parameterized Number is : 6

Parameterized Number is : 18

Parameterized Number is : 19

Parameterized Number is : 48

## 4.7 JUnit List Example

**package** other;

**import** org.junit.Test;

**import** java.util.ArrayList;

**import** java.util.Arrays;

**import** java.util.List;

**import** org.hamcrest.collection.IsEmptyCollection;

**import** **static** org.hamcrest.CoreMatchers.\*;

**import** **static** org.hamcrest.collection.IsCollectionWithSize.hasSize;

**import** **static** org.hamcrest.collection.IsIterableContainingInAnyOrder.containsInAnyOrder;

**import** **static** org.hamcrest.collection.IsIterableContainingInOrder.contains;

**import** **static** org.hamcrest.number.OrderingComparison.greaterThanOrEqualTo;

**import** **static** org.hamcrest.number.OrderingComparison.lessThan;

**import** **static** org.hamcrest.MatcherAssert.*assertThat*;

**public** **class** ListExample{

@Test

**public** **void** testAssertList() {

List<Integer> actual = Arrays.*asList*(1, 2, 3, 4, 5);

List<Integer> expected = Arrays.*asList*(1, 2, 3, 4, 5);

//All passed / true

//1. Test equal.

*assertThat*(actual, *is*(expected));

//2. Check List has this value

*assertThat*(actual, *hasItems*(2));

//3. Check List Size

*assertThat*(actual, hasSize(4));

*assertThat*(actual.size(), *is*(5));

//4. List order

// Ensure Correct order

*assertThat*(actual, contains(1, 2, 3, 4, 5));

// Can be any order

*assertThat*(actual, containsInAnyOrder(5, 4, 3, 2, 1));

//5. check empty list

*assertThat*(actual, *not*(IsEmptyCollection.empty()));

*assertThat*(**new** ArrayList<>(), IsEmptyCollection.empty());

//6. Test numeric comparisons

*assertThat*(actual, *everyItem*(greaterThanOrEqualTo(1)));

*assertThat*(actual, *everyItem*(lessThan(10)));

}

}

## 4.8 JUnit Map Example

**public** **class** MapExample {

@Test

**public** **void** testAssertMap() {

Map<String, String> map = **new** HashMap<>();

map.put("j", "java");

map.put("c", "c++");

map.put("p", "python");

map.put("n", "node");

Map<String, String> expected = **new** HashMap<>();

expected.put("n", "node");

expected.put("c", "c++");

expected.put("j", "java");

expected.put("p", "python");

//All passed / true

//1. Test equal, ignore order

assertThat(map, is(expected));

//2. Test size

assertThat(map.size(), is(4));

//3. Test map entry, best!

assertThat(map, IsMapContaining.hasEntry("n", "node"));

assertThat(map, not(IsMapContaining.hasEntry("r", "ruby")));

//4. Test map key

assertThat(map, IsMapContaining.hasKey("j"));

//5. Test map value

assertThat(map, IsMapContaining.hasValue("node"));

}

}

## 4.9 JUnit Tools

Following are the JUnit Tools used for Testing −

1. [Cactus](http://jakarta.apache.org/cactus/)
2. [JWebUnit](https://jwebunit.github.io/jwebunit/)
3. [XMLUnit](http://www.xmlunit.org/)
4. [MockObject](http://search.cpan.org/~chromatic/Test-MockObject-1.20161202/lib/Test/MockObject.pm)

## References

<http://junit.org/junit4/>

<https://www.tutorialspoint.com/junit/>

<http://www.javatpoint.com/junit-tutorial>

<http://www.mkyong.com/tutorials/junit-tutorials/>

# Mockito

Mockito facilitates creating mock objects(sample , similar Objects). Mock objects are nothing but proxy for actual implementations.

## 1.Creating Mock Objects

**1.Normal way**

StockService stockServiceMock = mock(StockService.class);

**2.Using Annotation**

@Mock

StockService stockServiceMock

## 2.Methods & Usage

All trhese methods are **static,** import provided by Mockito

**import** **static** org.mockito.Mockito.\*;

**when(...).thenReturn(...); -** adds a functionality to a mock object using the methods when(). Then()

when(...).thenReturn(...);

**// mock the behavior of stock service to return the value of various stocks**

when(stockServiceMock.getPrice(googleStock)).thenReturn(50.00);

when(stockServiceMock.getPrice(microsoftStock)).thenReturn(1000.00);

// @RunWith attaches a runner with the test class to initialize the test data

@RunWith(MockitoJUnitRunner.class)

public class MathApplicationTester {

//@InjectMocks annotation is used to create and inject the mock object

@InjectMocks //main Object

MathApplication mathApplication = new MathApplication();

//@Mock annotation is used to create the dependent mock object

@Mock //Calculator object id decalted inside MathApplication

CalculatorService calcService;

@Test

public void testAdd(){

//add the behavior of calc service to add two numbers

when(calcService.add(10.0,20.0)).thenReturn(30.00);

//test the add functionality

Assert.assertEquals(mathApplication.add(10.0, 20.0),30.0,0);

}

}

**verify()** method is ensure whether a mock method is being called with reequired arguments or not

//**verify call to calcService is made or not with same arguments** verify(calcService).add(20.0, 30.0);

**Times()** method - Suppose MathApplication should call the CalculatorService.serviceUsed() method only once

//check if add function is called three times

verify(calcService, times(3)).add(10.0, 20.0);

//verify that method was never called on a mock

verify(calcService, never()).multiply(10.0,20.0)

Mockito provides the following additional methods to vary the expected call counts

//check a minimum 1 call count

verify(calcService, atLeastOnce()).subtract(20.0, 10.0);

//check if add function is called minimum 2 times

verify(calcService, atLeast(2)).add(10.0, 20.0);

//check if add function is called maximum 3 times

verify(calcService, atMost(3)).add(10.0,20.0);

**Exception**

@Test(expected = RuntimeException.class)

public void testAdd(){

//add the behavior to throw exception

doThrow(new RuntimeException("Add operation not implemented"))

.when(calcService).add(10.0,20.0);

}

**InOrder** class - takes care of the order of method calls

//create an inOrder verifier for a single mock

InOrder inOrder = inOrder(calcService);

//following will make sure that add is first called then subtract is called.

inOrder.verify(calcService).subtract(20.0,10.0);

inOrder.verify(calcService).add(20.0,10.0);

**reset** - reset a mock so that it can be reused later

//reset the mock

reset(calcService);

[**spy()**](http://javadoc.io/page/org.mockito/mockito-core/latest/org/mockito/Mockito.html#spy-T-)**/**[**@Spy**](http://javadoc.io/page/org.mockito/mockito-core/latest/org/mockito/Spy.html)**:** partial mocking, real methods are invoked but still can be verified and stubbed

# Hamcrest Matchers

Hamcrest provides a more readable, declarative approach to asserting and matching your test results.

Hamcrest has the target to make tests as readable as possible. For example, the is method is a thin wrapper for equalTo(value).

**import** **static** org.hamcrest.MatcherAssert.assertThat;

**import** **static** org.hamcrest.Matchers.is;

**import** **static** org.hamcrest.Matchers.equalTo;

**boolean** a;

**boolean** b;

// all statements test the same

assertThat(a, equalTo(b));

assertThat(a, is(equalTo(b)));

assertThat(a, is(b));

The following snippets compare pure JUnit 4 assert statements with Hamcrest matchers.

// JUnit 4 for equals check

assertEquals(expected, actual);

// Hamcrest for equals check

assertThat(actual, is(equalTo(expected)));

// JUnit 4 for not equals check

assertNotEquals(expected, actual)

// Hamcrest for not equals check

assertThat(actual, is(not(equalTo(expected))));

The following are the most important Hamcrest matchers:

* allOf - matches if all matchers match (short circuits)
* anyOf - matches if any matchers match (short circuits)
* not - matches if the wrapped matcher doesn’t match and vice
* equalTo - test object equality using the equals method
* is - decorator for equalTo to improve readability
* hasToString - test Object.toString
* instanceOf, isCompatibleType - test type
* notNullValue, nullValue - test for null
* sameInstance - test object identity
* hasEntry, hasKey, hasValue - test a map contains an entry, key or value
* hasItem, hasItems - test a collection contains elements
* hasItemInArray - test an array contains an element
* closeTo - test floating point values are close to a given value
* greaterThan, greaterThanOrEqualTo, lessThan, lessThanOrEqualTo
* equalToIgnoringCase - test string equality ignoring case
* equalToIgnoringWhiteSpace - test string equality ignoring differences in runs of whitespace
* containsString, endsWith, startsWith - test string matching

# Integration Testing

Integration testing is all about testing all pieces of an application working together as they would in a live or production environment

To convert any JUnit test into a proper integration test, there are really two basic things that you need to do.

* The first is you need to annotate your tests with the @RunWith annotation and specify that you want to run it with the SpringJUnit4ClassRunner.class
* The second is you need to add the @SpringApplicationConfiguration  annotation and provide your main Spring Boot class for your application.

@RunWith(SpringJUnit4ClassRunner.**class**)

@SpringApplicationConfiguration(Demo.**class**)

**public** **class** UserRepoIntegrationTest {

@Autowired

**private** UserRepository userRepository;

@Test

**public** **void** testFindAll() {

List<User> users = userRepository.findAll();

assertThat(users.size(), is(greaterThanOrEqualTo(0)));

}

}

Regardless of the test result - successful or unsuccessful, open your IDE Console tab and you should notice that it looks like your application started (Spring logo, info etc). This happens because our application actually starts with integration tests

## MockMvc

**Spring MockMVC** to perform **integration testing** of spring webmvc controllers

**MockMVC** class is part of [Spring MVC](https://howtodoinjava.com/spring-mvc-tutorial/) test framework which helps in testing the controllers explicitly starting a Servlet container.

@RunWith(SpringRunner.**class**)

@WebMvcTest(StudentController.**class**)

**public** **class** StudentIntegrationTests {

@Autowired

**private** MockMvc mvc;

}

* [SpringRunner](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/test/context/junit4/SpringRunner.html) is an alias for the SpringJUnit4ClassRunner.
* [@WebMvcTest](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/test/autoconfigure/web/servlet/WebMvcTest.html) annotation is used for Spring MVC tests. It disables full auto-configuration and instead apply only configuration relevant to MVC tests.
* StudentController.**class** means initialize only this controller and provide dependent Mock object of this controller.

**MockMvcRequestBuilders -**hit the APIs & passing the path parameters and verify the status response codes

**MockMvcResultMatchers –** get the response content & matches with expected content

## References

Junit & Mockito

Tutorilaspoint.com

MockMVC

<https://howtodoinjava.com/spring-boot2/spring-boot-mockmvc-example/>

# Help.html

## StudentApp Document

### StudentApp + AngularJs+ CurdRepository+MockMVC References

**CURD + POSTMAN**  
<https://www.callicoder.com/spring-boot-rest-api-tutorial-with-mysql-jpa-hibernate/>

**AngularJS+Templates**  
<https://examples.javacodegeeks.com/enterprise-java/spring/boot/spring-boot-and-angularjs-integration-tutorial/>

**Spring+AngularJS Controllers**  
<https://java2blog.com/spring-boot-angularjs-example/>

**SpringBoot MockMVC Tutorial**

* Mokito : <https://howtodoinjava.com/spring-boot2/spring-boot-mockito-junit-example/>
* MOCK MVC : <https://howtodoinjava.com/spring-boot2/spring-boot-mockmvc-example/>

### StudentApp MongoRepository

Example App : <https://www.journaldev.com/18156/spring-boot-mongodb>

**1. Add Maven Dependency:**    spring-boot-starter-data-mongodb

2.Create Sudent Collection

>use student

>db.createCollection("student");

>db.student.insert(  
   {  
      sno: 501,  
      name: "Satya Kaveti",  
      city:"Vijayawada",  
      marks:508  
   }  
)

3.in **application.propertis** , add Mongo DB details

spring.data.mongodb.database=student  
spring.data.mongodb.port=27017  
spring.data.mongodb.host=localhost

2.Create **StudentModel**.java with **@Document & Id**annotations

3.create **StudentMongoRepository**extends **MongoRepository**

4.Create **StudentMongoController.java**

**How can Spring Boot work without driver configuration?**

Spring Boot gives you defaults on all things, the default in database is **H2**, so when you want to change this and use any other database you must define the connection attributes in the **application.properties** file.

* H2 is one of the popular in memory databases. Spring Boot has default integration for H2
* is live only during the time of execution of the application, not for real world applications
* The h2-\*.jar is just an engine (the code) of the database. It is read-only and it does not store any information. The data in H2 can be stored either in memory or on disk in a specified file. You are actually specifying one:

In our Application we are using MySQL, so we provided details with out Driver detilas

spring.datasource.url = jdbc:mysql://localhost:3306/student?useSSL=false

spring.datasource.username = root

spring.datasource.password = root

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<scope>runtime</scope>

</dependency>

* [**@EnableAutoConfiguration**](https://docs.spring.io/spring-boot/docs/1.2.1.RELEASE/api/org/springframework/boot/autoconfigure/EnableAutoConfiguration.html) : This annotation tells Spring to automatically configure your application based on the dependencies that you have added in the **pom.xml** file.
* For example, If **spring-data-jpa** or **spring-jdbc** is in the classpath, then it automatically tries to configure a **DataSource** by reading the database properties from **application.properties** file.
* So,in above we just have to add the configuration and Spring Boot will take care of the rest.
* In the above properties file, the last two properties are for hibernate. Spring Boot uses Hibernate as the default JPA implementation.
* If you remove mysql-connector-java from pom.xml, SpringBoot unable to find the MySQL related java classes in insert application.propeties values to create datasource, it will throw below error.

Failed to bind properties under '' to com.zaxxer.hikari.HikariDataSource:

Property: driverclassname

Value: com.mysql.cj.jdbc.Driver

Origin: "driverClassName" from property source "source"

**Reason: Failed to load driver class com.mysql.cj.jdbc.Driver in either of HikariConfig class loader or Thread context classloader**

Action:

Update your application's configuration

So finally, SpringBoot uses Drivers & all normal stuff though @EnableAutoConfiguration to do the job

### Spring Boot with multiple databases

In Our StudentApp we have multiple Databases.

1.if we use only one Database :MySQL we can use default spring.data.\*properties. Spring @EnableAutoConfiguration will create Datasource by reading these properties

spring.datasource.url = jdbc:mysql://localhost:3306/student?useSSL=false

spring.datasource.username = root

spring.datasource.password = root

2.if we use two databases :MySQL, Mongo we can use default spring.data.\*properties of therir respective databases. Spring @EnableAutoConfiguration will create two different Datasource by reading these properties

spring.data.mongodb.database=student

spring.data.mongodb.port=27017

spring.data.mongodb.host=localhost

3.If we use two DB’s with our own Config properties we need to Ovveride DataSource Manually

#----------- MySQL -------------

own.spring.mysql.datasource.url = jdbc:mysql://localhost:3306/student?useSSL=false

own.spring.mysql.datasource.username = root

own.spring.mysql.datasource.password = root

#----------- MongoDB -------------

own.spring.mongo.datasource.database=student

own.spring.mongo.datasource.port=27017

own.spring.mongo.datasource.host=localhost

Because we want the Spring Boot autoconfiguration to pick up those different properties (and actually instantiate two different DataSources), we need to instantiate our DataSource beans manually in a configuration class

@Configuration  
public class MultipleDataSourceConfiguration {  
   
    @Bean  
    @Primary  
    @ConfigurationProperties(prefix="own.spring.mysql.datasource")  
    public DataSource primaryDataSource() {  
        return DataSourceBuilder.create().build();  
    }  
   
    @Bean  
    @ConfigurationProperties(prefix="own.spring.mongo.datasource")  
    public DataSource secondaryDataSource() {  
        return DataSourceBuilder.create().build();  
    }  
}

<https://medium.com/@joeclever/using-multiple-datasources-with-spring-boot-and-spring-data-6430b00c02e7>

<http://www.java2novice.com/spring-boot/configure-multiple-datasources/>

<https://www.infoq.com/articles/Multiple-Databases-with-Spring-Boot>

**@Conditional & @Profilers**

While developing Spring based applications we may come across a need to register beans conditionally.

For example, you may want to register a DataSource bean pointing to the **dev** database abd different **production database** while running in production.

To address this problem, Spring 3.1 introduced the concept of **Profiles**. When you run the application you can activate the desired profiles ,and only those beans of that profiles will be registered.

@Configuration

public class AppConfig

{

@Bean

@Profile("DEV")

public DataSource devDataSource() {

...

}

@Bean

@Profile("PROD")

public DataSource prodDataSource() {

...

}

}

Then you can specify the active profile using System Property **-Dspring.profiles.active=DEV**

Now we can configure both **JdbcUserDAO**and **MongoUserDAO**beans conditionally using **@Conditional**as follows:

@Configuration

public class AppConfig

{

@Bean

@Conditional(MySQLDatabaseTypeCondition.class)

public UserDAO jdbcUserDAO(){

return new JdbcUserDAO();

}

@Bean

@Conditional(MongoDBDatabaseTypeCondition.class)

public UserDAO mongoUserDAO(){

return new MongoUserDAO();

}

}

**How EnableAutoConfiguration implemented?**

auto-configuration is implemented with standard @Configuration classes. Additional @Conditional annotations are used to constrain when the auto-configuration should apply.

### SpringBoot Security

spring-boot-starter-security: take care of all the required dependencies related to spring security.

<dependency>

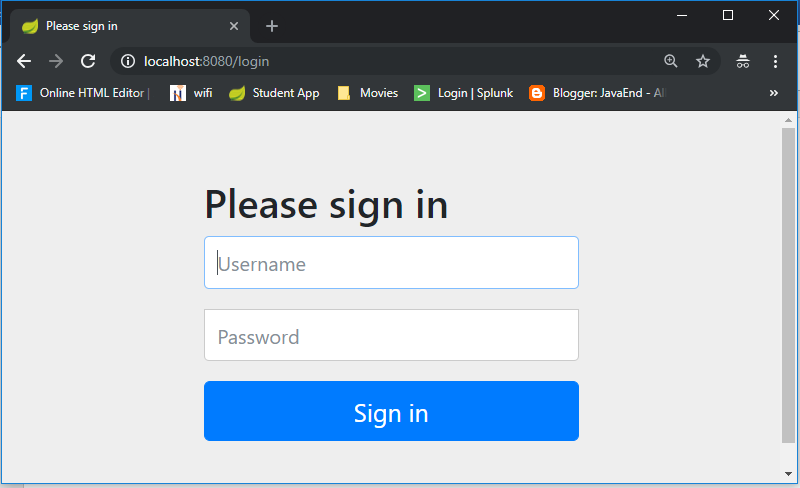
<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

This will include the *SecurityAutoConfiguration*class – containing the initial/default security configuration.

Just Run the project & see the magic



**We never created this login form, but from where it came from?**

SpringSecurity default comes with login page & you can login with generated password which is already printed in the console

Using generated security password: **8b4667a4-cc3a-47fd-b51f-b6f5e83745df**

Def.user name is : **user**

You can change the password by providing a security.user.password. This and other useful properties are externalized via [SecurityProperties](https://github.com/spring-projects/spring-boot/tree/v1.4.0.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/security/SecurityProperties.java) (properties prefix "security").

security.user.name=user

security.user.name=password

security.basic.enabled=true

To discard the security auto-configuration and add our own configuration, we need to exclude the ***SecurityAutoConfiguration*** class.

@SpringBootApplication(exclude = { SecurityAutoConfiguration.class })

public class SpringBootSecurityApplication {

    public static void main(String[] args) {

        SpringApplication.run(SpringBootSecurityApplication.class, args);

    }

}

Or by adding some configuration into the *application.properties* file:

spring.autoconfigure.exclude=org.springframework.boot.autoconfigure.security.SecurityAutoConfiguration

If we disabling security auto-configuration, we need to provide our own configuration, by extends WebSecurityConfigurerAdapter

@Configuration

@EnableWebSecurity

**public** **class** SecurityConfig **extends** WebSecurityConfigurerAdapter {

@Override

**public** **void** configure(HttpSecurity http) **throws** Exception {

// It allows configuring web based security for specific http requests

http

.authorizeRequests()

.anyRequest().authenticated()

.and()

.formLogin()

.and()

.httpBasic();

/\* ============ Custom login Page URL ==========

http

.authorizeRequests()

.antMatchers("/admin/\*\*").hasRole("ADMIN")

.antMatchers("/anonymous\*").anonymous()

.antMatchers("/login\*").permitAll()

.anyRequest().authenticated()

.and()

.formLogin()

.loginPage("/login.html")

.loginProcessingUrl("/perform\_login")

.defaultSuccessUrl("/homepage.html", true)

//.failureUrl("/login.html?error=true")

.failureHandler(authenticationFailureHandler())

.and()

.logout()

.logoutUrl("/perform\_logout")

.deleteCookies("JSESSIONID")

.logoutSuccessHandler(logoutSuccessHandler());

\*/

}

@Bean

@Override

**public** UserDetailsService userDetailsService() {

UserDetails user =

User.~~withDefaultPasswordEncoder~~()

.username("user")

.password("user")

.roles("USER")

.build();

**return** **new** InMemoryUserDetailsManager(user);

}

}

Let’s summarize what Nick did in order to add Spring Boot Security to his web app. To secure his web app,

* he added Spring Boot Security to the classpath.
* Once it was in the classpath, Spring Boot Security was enabled by default.
* He then customized the security by extending  WebSecurityConfigurerAdapter and added his own configure and userDetailsServiceimplementation.

<http://localhost:8080/login?logout>

<https://docs.spring.io/spring-security/site/docs/current/guides/html5/form-javaconfig.html>

<https://examples.javacodegeeks.com/enterprise-java/spring/boot/spring-boot-security-example/>

### SpringBoot AOP

<http://www.springboottutorial.com/spring-boot-and-aop-with-spring-boot-starter-aop>

<https://www.javainuse.com/spring/spring-boot-aop>

SpringBoot MicroServices

<https://www.dineshonjava.com/microservices-with-spring-boot/>

<https://spring.io/blog/2015/07/14/microservices-with-spring>

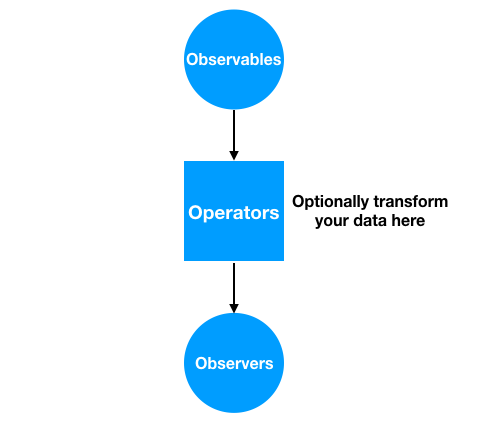
### Spring Acuator

### Reactive JavaRx

* Reactive comes from the word react, which means to **react to changes** in the state instead of actually doing the state change.
* The reactive model listens to changes in the event and runs the relevant code accordingly.
* **observer/subscriber** attached listening to the stream would receive the data.

The basic building blocks of RxJava are:

* **Observables**: That emits data streams
* **Observers and Subscribers**: That consume the data stream. The only difference between an Observer and a Subscriber is that a Subscriber class has the methods to unsubscribe/resubscribe independently without the need of the observerable methods.
* **Operators**: That transform the data stream

[](https://cdn.journaldev.com/wp-content/uploads/2018/02/rxjava-basics-flow.png)

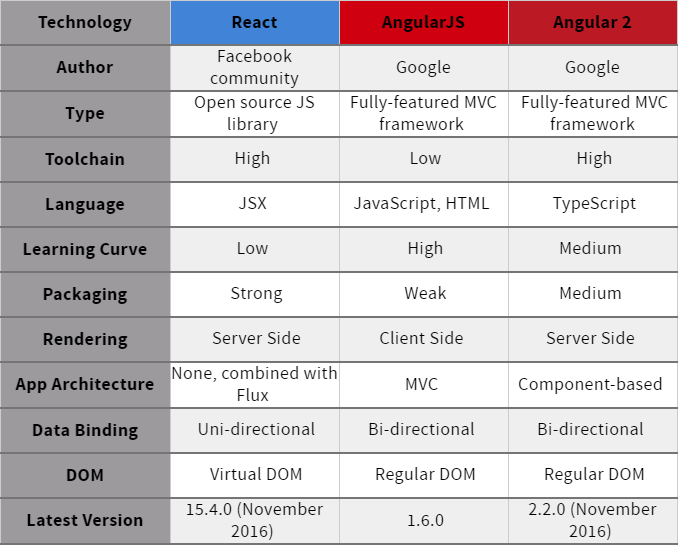
<https://cdn.journaldev.com/wp-content/uploads/2018/02/rxjava-basics-flow.png>

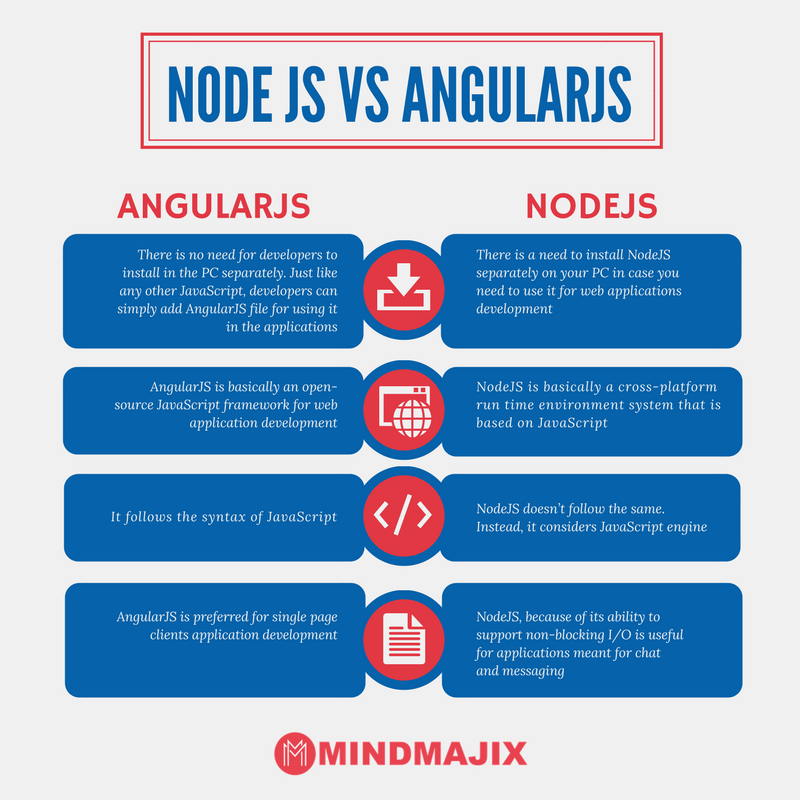
### SpringBoot – Reactive Programming

<https://www.baeldung.com/spring-webflux>

<https://www.journaldev.com/20763/spring-webflux-reactive-programming>

### AngularJs(Angular 1) vs Angular (Angular 2)





### What exactly node.js is?

Is it a web server or a programming language for server-side scripts?

* So here’s how it is, how it’s always been: a browser sends a request to a website. The site’s server receives the request, tracks down the requested file, performs any database queries as needed, and sends a response to the browser. In traditional web servers, such as Apache, each request causes the server to create a new system process to handle that request
* Now think about what that means for a traditional web server like Apache. For each and every user connected to the site, your server has to keep a connection open. Each connection requires a process, and each of those processes will spend most of its time either sitting idle (consuming memory) or waiting on a database query to complete. This means that it’s hard to scale up to high numbers of connections without grinding to a near halt and using up all your resources.
* So what’s the solution? Here’s where some of that jargon from before comes into play: specifically **non-blocking** and **event-driven**
* Think of **a non-blocking** server as a loop: it just keeps going round and round. A request comes in, the loop grabs it, passes it along to some other process (like a database query), sets up a callback, and keeps going round, ready for the next request. It doesn’t just sit there, waiting for the database to come back with the requested info.
* If the database query comes back — fine, we’ll deal with that the same way: throw a response back to the client and keep looping around. There’s theoretically no limit on how many database queries you can be waiting on, or how many clients have open requests, because you’re not spending any time waiting for them. You deal with them all in their own time
* **event-driven means**: the server only reacts when an event occurs. That could be a request, a file being loaded, or a query being executed — it really doesn’t matter.

### How to host node.js applications?

You need to Host AWS or Google Cloud or any other cloud platform because it needs Node.js to be installed.

### Npm, bower packges

Let’s understand by Example

I used the complete [**MEAN**](http://mean.io/) stack for this series

**MEAN** is a set of Open Source components that together, provide an end-to-end framework for building dynamic web applications;

* **M**ongoDB : Document database – used by your back-end application to store its data as JSON (JavaScript Object Notation) documents
* **E**xpress (sometimes referred to as Express.js): Back-end web application framework running on top of Node.js
* **A**ngular (formerly Angular.js): Front-end web app framework; runs your JavaScript code in the user’s browser, allowing your application UI to be dynamic
* **N**ode.js : JavaScript runtime environment – lets you implement your application back-end in JavaScript

1.create **package.json** to install some Node packages.(like maven, here we can see **express.js** dependency)

//package.json

{

"name": "node-rest-auth",

"main": "server.js",

"dependencies": {

"bcrypt": "^0.8.5",

"body-parser": "~1.9.2",

**"express": "~4.9.8",**

"jwt-simple": "^0.3.1",

"mongoose": "~4.2.4",

"morgan": "~1.5.0",

"passport": "^0.3.0",

"passport-jwt": "^1.2.1"

}

}

To install pacakges Run

**npm install**

This will install all our modules to **node\_modules/**.

We can also install one by one without package.json as below, it will get latest version of it

npm install mongojs

npm install express

2.create **server.js**, here we import all the needed elements and create our server with url localhost:9090

var express = require('express');

var app = express();

var bodyParser = require('body-parser');

var morgan = require('morgan');

var mongoose = require('mongoose');

var passport = require('passport');

var config = require('./config/database'); // get db config file

var User = require('./app/models/user'); // get the mongoose model

var port = process.env.PORT || 9090;

var jwt = require('jwt-simple');

// get our request parameters

app.use(bodyParser.urlencoded({ extended: false }));

app.use(bodyParser.json());

// log to console

app.use(morgan('dev'));

// Use the passport package in our application

app.use(passport.initialize());

// demo Route (GET **http://localhost:9090**)

app.get('/', function(req, res) {

res.send('Hello! The API is at http://localhost:' + port + '/api');

});

// Start the server

app.listen(port);

console.log('There will be dragons: http://localhost:' + port);

**3.config/database.js**

module.exports = {

'secret': 'devdacticIsAwesome',

'database': 'mongodb://localhost/node-rest-auth'

};

4. user model for our user authentication

**//app/models/user.js**

var mongoose = require('mongoose');

var Schema = mongoose.Schema;

var bcrypt = require('bcrypt');

// Thanks to http://blog.matoski.com/articles/jwt-express-node-mongoose/

// set up a mongoose model

var UserSchema = new Schema({

name: {

type: String,

unique: true,

required: true

},

password: {

type: String,

required: true

}

});

UserSchema.pre('save', function (next) {

var user = this;

if (this.isModified('password') || this.isNew) {

bcrypt.genSalt(10, function (err, salt) {

if (err) {

return next(err);

}

bcrypt.hash(user.password, salt, function (err, hash) {

if (err) {

return next(err);

}

user.password = hash;

next();

});

});

} else {

return next();

}

});

UserSchema.methods.comparePassword = function (passw, cb) {

bcrypt.compare(passw, this.password, function (err, isMatch) {

if (err) {

return cb(err);

}

cb(null, isMatch);

});

};

module.exports = mongoose.model('User', UserSchema);

Now the basics are set up, and you can start our server from now on just with

**node server.js**

<https://devdactic.com/restful-api-user-authentication-1/>

#### Now NPM vs Bower

Npm and Bower are both **dependency management tools**. But the main difference between both is

* **npm is used for installing Node js modules**
* **bower is used for managing front end components like html, css, js etc**

running bower install will fetch the package and put it in /vendor directory,  
running npm install it will fetch it and put it into /node\_modules directory.

**Grunt** is quite different from Npm and Bower. Grunt is a javascript task runner tool. You can do a lot of things using grunt which you had to do manually otherwise

There are grunt plugins for **compilation, uglifying your javascript, copy files/folders, minifying javascript etc.**

# References

OnlineHTML - <https://www.froala.com/online-html-editor>