# Spring Missing/ Confusing Topics

## Diffrence between @Component & @Autowire

@Compoent is just creating bean object of that class at Auto Scanning time.

@Autowire is, after creating bean object in @Componet time, it will autowires/ injecets the Dependent classes.

## Diffrence beteen @ComponentScan & @Enable AutoCOnfiguration

One of the main advantages of Spring Boot is its annotation driven versus traditional xml based configurations, **@EnableAutoConfiguration** automatically configures the Spring application based on its included jar files, it sets up defaults or helper based on dependencies in pom.xml. Auto-configuration is usually applied based on the classpath and the defined beans. Therefore, we donot need to define any of the DataSource, EntityManagerFactory, TransactionManager etc and magically based on the classpath, Spring Boot automatically creates proper beans and registers them for us. For example when there is a tomcat-embedded.jar on your classpath you likely need a TomcatEmbeddedServletContainerFactory (unless you have defined your own EmbeddedServletContainerFactory bean). @EnableAutoConfiguration has a exclude attribute to disable an auto-configuration explicitly otherwise we can simply exclude it from the pom.xml, for example if we donot want Spring to configure the tomcat then exclude spring-bootstarter-tomcat from spring-boot-starter-web.

**@ComponentScan** provides scope for spring component scan, it simply goes though *the provided base package* and picks up dependencies required by @Bean or @Autowired etc, In a typical Spring application, @ComponentScan is used in a configuration classes, the ones annotated with @Configuration. Configuration classes contains methods annotated with @Bean. These @Bean annotated methods generate beans managed by Spring container. Those beans will be auto-detected by @ComponentScan annotation. There are some annotations which make beans auto-detectable like @Repository , @Service, @Controller, @Configuration, @Component. In below code Spring starts scanning from the package including BeanA class.

# 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="">

….

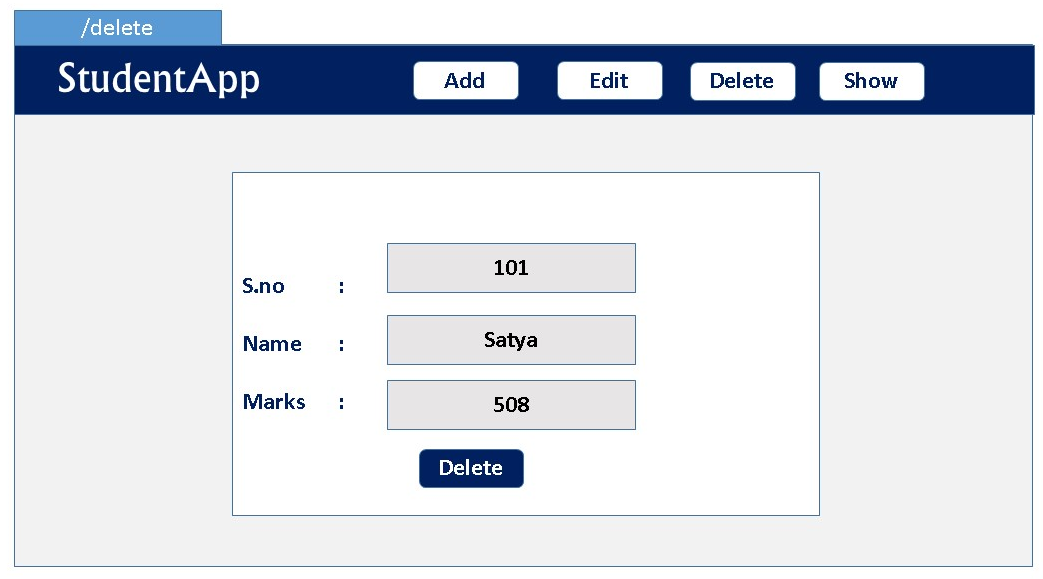
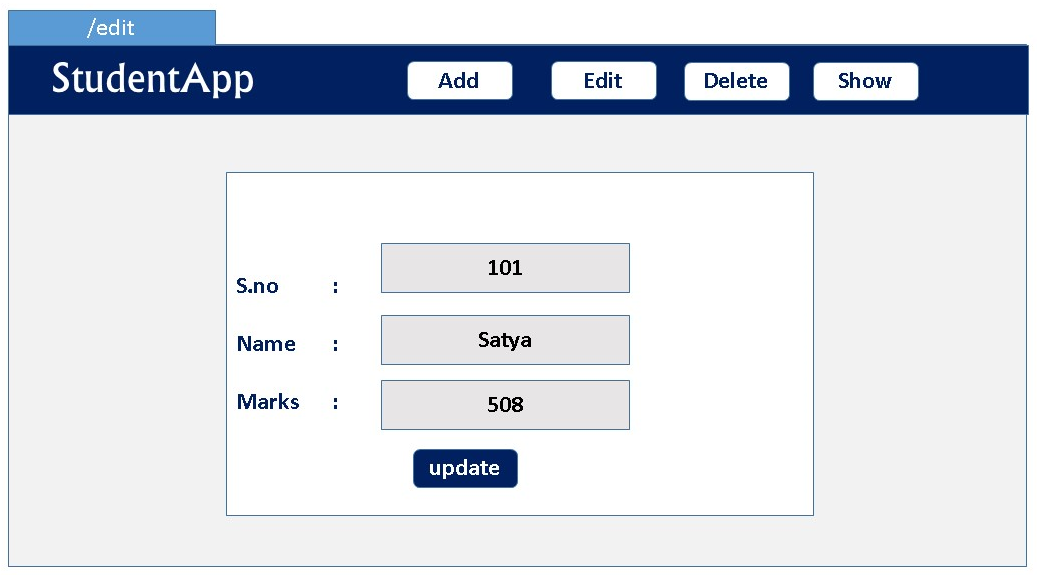
<div ng-view="">  
 // we can change the views as per controller logic display view  
 </div>

….

<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.

**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.

**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 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>

* orderBy Orders an array by an expression.

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

#### 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 :

#### 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-dblclick
* ng-focus
* ng-keydown
* ng-keypress
* ng-keyup
* ng-mousedown
* ng-mouseenter
* ng-mouseleave

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>

#### 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>

## 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

## 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>

# 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

# Types of Authentication

A servlet-based web application can choose from the following types of authentication, from least secure to most:

* Basic authentication
* Form-based authentication
* Digest authentication
* SSL and client certificate authentication

“Authentication” and “Authorization”. Authentication can be defined as the process of verifying someone’s identity by using pre-required details (Commonly username and password). Authorization is the process of allowing an authenticated user to access a specified resource (eg:-right to access a file).

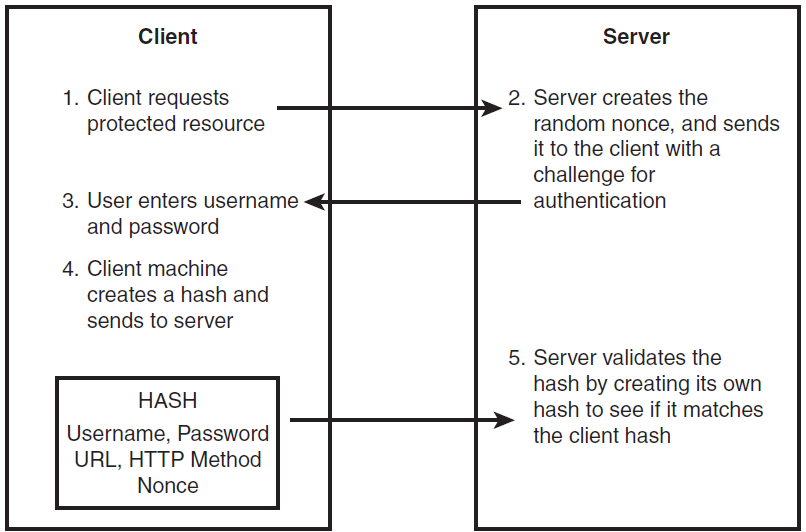
## HTTP Basic Authentication

* One solution is that of **HTTP Basic Authentication**. In this approach, an HTTP user agent simply provides a **username**and **password** to prove their authentication.
* This approach does not require cookies, session IDs, login pages, and other such specialty solutions, and because it uses the **HTTP header** itself, there’s no need to handshakes or other complex response systems.
* HTTP is not [encrypted](https://nordicapis.com/securing-your-datastream-with-p2p-encryption/) in any way. It is encapsulated in base64, and is often erroneously proclaimed as encrypted due to this

## Digest authentication

The difference between digest authentication and basic authentication is that in digest authentication, the username and password are never sent over the wire. Instead, a hash is created made up of the following pieces of information:

* The username
* The password
* The URL
* The randomly generated string (the nonce)
* The HTTP method being used

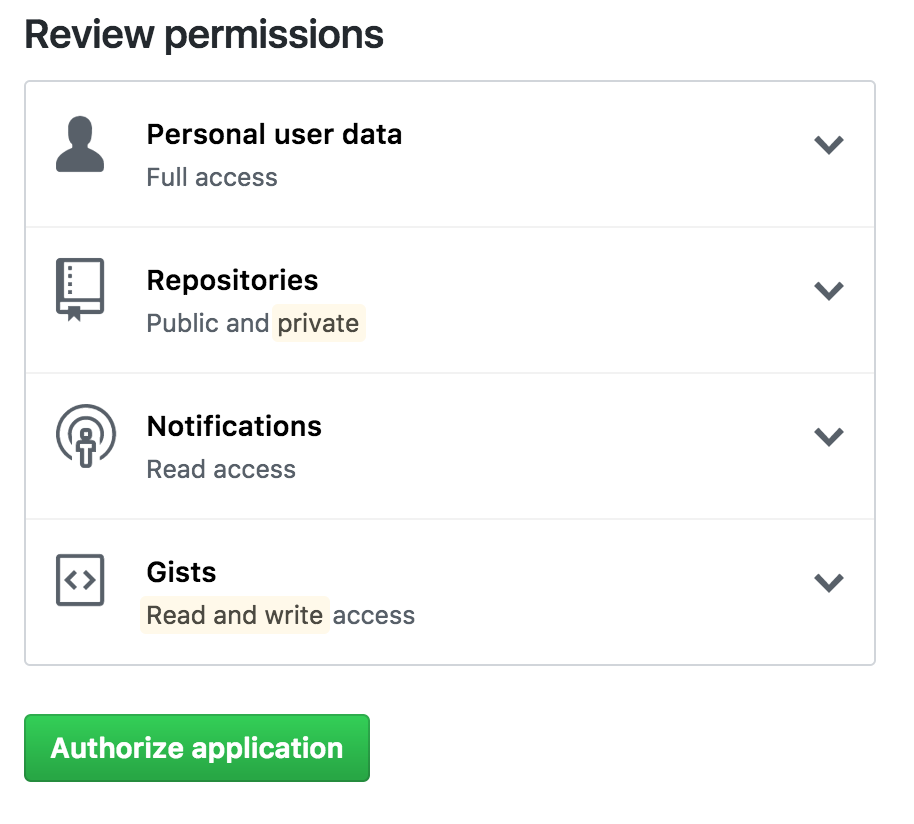


## API Keys : for Developer Quickstart

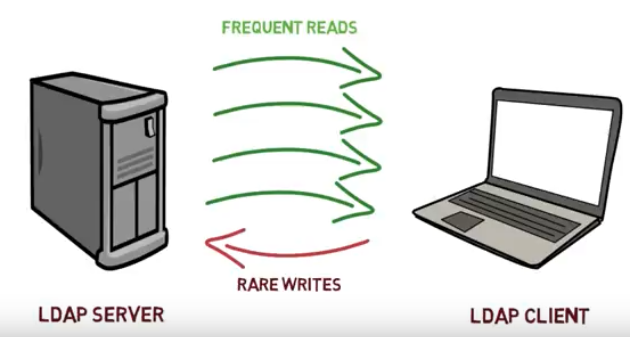
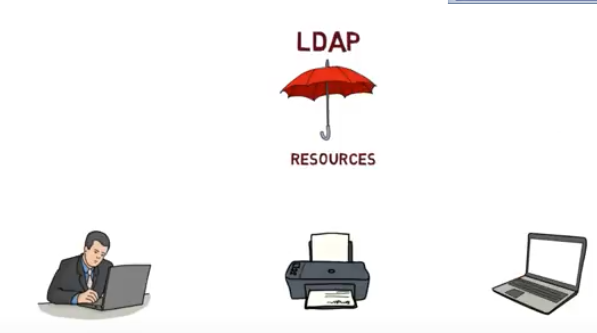
* To access Bitbucket in Hygieia, we will generate API key, and we will place that key in properties file.
* API Keys can be used as Basic HTTP Authentication credentials and provide a substitute for the account's actual username and password.
* The best thing about an API key is its simplicity. You merely log in to a service, find your API key (often in the settings screen), and copy it to use in an application, test in the browser, or use with one of these [API request tools](https://zapier.com/engineering/api-request-tools/)
* Typically, an API key gives full access to every operation an API can perform, including writing new data or deleting existing data. If you use the same API key in multiple apps, a broken app could destroy your users' data without an easy way to stop just that one app.
* Many API keys are sent in the query string as part of the URL, which makes it easier to discover for someone who should not have access to it. A better option is to put the API key in the Authorization header. In fact, that’s the [proposed standard](https://tools.ietf.org/html/rfc7235?utm_source=zapier.com&utm_medium=referral&utm_campaign=zapier):  
  Authorization: Apikey 1234567890abcdef

## OAuth Tokens: Great for Accessing User Data

* OAuth is the answer to accessing user data with APIs.
* users simply click a button to allow an application to access their accounts.



## LDAP



class Simple {

public static void main(String[] args) {

Hashtable authEnv = new Hashtable(11);

String userName = "johnlennon";

String passWord = "sushi974";

String base = "ou=People,dc=example,dc=com";

String dn = "uid=" + userName + "," + base;

String ldapURL = "ldap://ldap.example.com:389";

authEnv.put(Context.INITIAL\_CONTEXT\_FACTORY,"com.sun.jndi.ldap.LdapCtxFactory");

authEnv.put(Context.PROVIDER\_URL, ldapURL);

authEnv.put(Context.SECURITY\_AUTHENTICATION, "simple");

authEnv.put(Context.SECURITY\_PRINCIPAL, dn);

authEnv.put(Context.SECURITY\_CREDENTIALS, passWord);

try {

DirContext authContext = new InitialDirContext(authEnv);

System.out.println("Authentication Success!");

} catch (AuthenticationException authEx) {

System.out.println("Authentication failed!");

} catch (NamingException namEx) {

System.out.println("Something went wrong!");

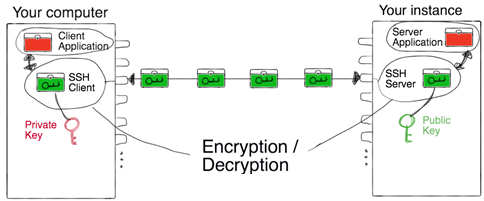
namEx.printStackTrace();

}

}

}

## SSH – Only for LINUX Server / CommadLine(git) related Access

* SSH, or secure shell, is an encrypted protocol used to administer and communicate with servers. When working with a Linux server, chances are, you will spend most of your time in a terminal session connected to your server through SSH.
* An SSH server can authenticate clients using a variety of different methods. The most basic of these is password authentication, which is easy to use, but not the most secure.
* SSH key pairs are two cryptographically secure keys that can be used to authenticate a client to an SSH server. Each key pair consists of a public key and a private key.
* The **private key** is retained by the client and should be kept absolutely secret. Any compromise of the private key will allow the attacker to log into servers that are configured with the associated public key without additional authentication. As an additional precaution, the key can be encrypted on disk with a passphrase.
* The associated public key can be shared freely without any negative consequences. The public key can be used to encrypt messages that only the private key can decrypt. This property is employed as a way of authenticating using the key pair.
* The public key is uploaded to a remote server that you want to be able to log into with SSH. The key is added to a special file within the user account you will be logging into called ~/.ssh/authorized\_keys.
* When a client attempts to authenticate using SSH keys, the server can test the client on whether they are in possession of the private key. If the client can prove that it owns the private key, a shell session is spawned or the requested command is executed.

## Base64 – not Authentication

represent [binary data](https://en.wikipedia.org/wiki/Binary_data) in an [ASCII](https://en.wikipedia.org/wiki/ASCII) string format

Each Base64 digit represents exactly 6 bits of data.

Steps:

* take three ASCII numbers 155, 162, and 233
* Convert into binary stream formate 100110111010001011101001
* groupings of six characters: 100110 111010 001011 101001.
* The binary string 100110 converts to the decimal number 38: 0\*2^01 + 1\*2^1 + 1\*2^2 + 0\*2^3 + 0\*2^4 + 1\*2^5 = 0+2+4+0+0+32.
* Base64 6-bit values 38, 58, 11 and 41.
* Using the Base64 conversion table:
  + 38 is m
  + 58 is 6
  + 11 is L
  + 41 is p

# Help.html

## StudentApp Document

**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();  
    }  
}

**@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>

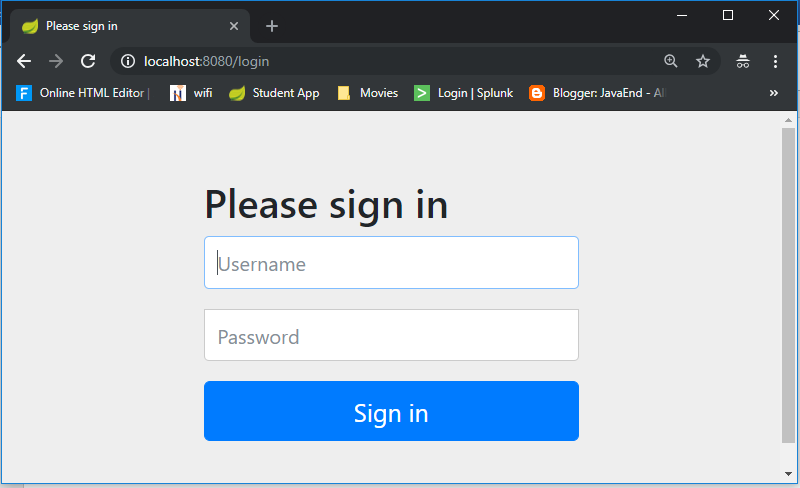
<groupId>org.springframework.boot</groupId>

<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" \t "_top) (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 we did in order to add Spring Boot Security to his web app. To secure his web app,

* we added Spring Boot Security to the classpath.
* Once it was in the classpath, Spring Boot Security was enabled by default.
* 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

Actually we don’t have Spring AOP starter in Springboot, but we can integrate using traditional Spring Framework

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aop</artifactId>

<version>5.0.1.RELEASE</version>

<scope>compile</scope>

</dependency>

<dependency>

<groupId>org.aspectj</groupId>

<artifactId>aspectjweaver</artifactId>

<version>1.8.12</version>

<scope>compile</scope>

</dependency>

Applications are generally developed with multiple layers. A typical Java application has

* Web Layer - Exposing the services to outside world using REST or a web application
* Business Layer - Business Logic
* Data Layer - Persistence Logic

While the responsibilities of each of these layers are different, there are a few common aspects that apply to all layers

* Logging
* Security

In this example we are implementing Logging – we are trying to Log the all the Application flow like, what controller is calling, what method is executing on performing certain operation.

**package** app.aop;

@Aspect

@Configuration

**public** **class** StudentAOP {

**private** Logger logger = LoggerFactory.*getLogger*(**this**.getClass());

//What kind of method calls I would intercept

//execution(\* PACKAGE.\*.\*(..))

//Weaving & Weaver

@Before("execution(\* app.repository.\*.\*(..))")

**public** **void** before(JoinPoint joinPoint){

//Advice

logger.info(" \n \n====== @Before===== \t app.repository");

logger.info(" Allowed execution for {}", joinPoint);

logger.info(" ====== @Before===== \n \n");

}

@AfterReturning(value = "execution(\* app.controller.\*.\*(..))",

returning = "result")

**public** **void** afterReturning(JoinPoint joinPoint, Object result) {

logger.info("\n \n ====== @AfterReturning ===== \t : app.controller ");

logger.info("{} returned with value {}", joinPoint, result);

logger.info(" ====== @AfterReturning ===== \n \n");

}

@After(value = "execution(\* app.security.\*.\*(..))")

**public** **void** after(JoinPoint joinPoint) {

logger.info("\n \n ====== @After ===== \t : app.security");

logger.info("after execution of {}", joinPoint);

logger.info(" ====== @After ===== \n \n");

}

}

**====== @After ===== : app.security**

2019-02-10 22:50:12.232 INFO 8152 --- [ main] dentAOP$$EnhancerBySpringCGLIB$$bb027782 : after execution of execution(UserDetailsService app.security.SecurityConfig.userDetailsService())

2019-02-10 22:50:12.236 INFO 8152 --- [ main] dentAOP$$EnhancerBySpringCGLIB$$bb027782 : **====== @After =====**

**====== @AfterReturning ===== : app.controller**

2019-02-10 22:43:58.212 INFO 2440 --- [nio-8080-exec-7] dentAOP$$EnhancerBySpringCGLIB$$45dcbfee : execution(List app.controller.StudentMongoController.getAllStudents()) returned with value [Student [sno=501, name=vinay, city=karnool, marks=545], Student [sno=502, name=VINOD, city=BNHGG, marks=456]]

2019-02-10 22:43:58.212 INFO 2440 --- [nio-8080-exec-7] dentAOP$$EnhancerBySpringCGLIB$$45dcbfee : **====== @AfterReturning =====**

Getting all users.2019-02-10 22:43:58.248 INFO 2440 --- [nio-8080-exec-9] dentAOP$$EnhancerBySpringCGLIB$$45dcbfee :

**====== @AfterReturning ===== : app.controller**

2019-02-10 22:43:58.248 INFO 2440 --- [nio-8080-exec-9] dentAOP$$EnhancerBySpringCGLIB$$45dcbfee : execution(List app.controller.StudentMongoController.getAllStudents()) returned with value [Student [sno=501, name=vinay, city=karnool, marks=545], Student [sno=502, name=VINOD, city=BNHGG, marks=456]]

2019-02-10 22:43:58.248 INFO 2440 --- [nio-8080-exec-9] dentAOP$$EnhancerBySpringCGLIB$$45dcbfee : **====== @AfterReturning =====**

**Pointcut** - Pointcut is the **expression** used to define when a call to a method should be intercepted. In the above example, "execution(\* app.repository.\*.\*(..))" is the pointcut.

**Advice** - It is the **logic that you would want to exceute** when you intercept a method. In the above example, it is the code inside the before(JoinPoint joinPoint)method.

**Aspect** – is the Class we defiend, combination of on which method **(Pointcut)** and what to do **(Advice)** is called an Aspect.

**Join Point** - When the code is executed and the condition for pointcut is met, the advice is executed. The Join Point is a specific execution instance of an advice.

**Weaver** - Weaver is the framework which implements AOP - AspectJ or Spring AOP.

### SpringBoot Acuator - Health check, Auditing, Metrics,Monitoring

Actuator brings production-ready features to our application.

**Monitoring our app, gathering metrics, understanding traffic or the state of our database becomes trivial with this dependency.**

<dependency>

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

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

</dependency>

Once above maven dependency is included in the POM file, **16 different actuator REST endpoints**, such as actuator, beans, dump, info, loggers, and metrics are exposed

Some of important and widely used actuator endpoints are given below:

|  |  |
| --- | --- |
| **ENDPOINT** | **USAGE** |
| /env | Returns list of properties in current environment |
| /health | Returns application health information. |
| /auditevents | Returns all auto-configuration candidates and the reason why they ‘were’ or ‘were not’ applied. |
| /beans | Returns a complete list of all the Spring beans in your application. |
| /trace | Returns trace logs (by default the last 100 HTTP requests). |
| /dump | It performs a thread dump. |
| /metrics | It shows metrics information like JVM memory used, system CPU usage, open files, and much more. |

You can access all avaible endpoint by this URL: [**http://localhost:8080/actuator**](http://localhost:8080/actuator)

{

"\_links": {

"self": {

"href": *"http://localhost:8080/actuator"*,

"templated": false

},

"health": {

"href": *"http://localhost:8080/actuator/health"*,

"templated": false

},

"health-component-instance": {

"href": *"http://localhost:8080/actuator/health/{component}/{instance}"*,

"templated": true

},

"health-component": {

"href": *"http://localhost:8080/actuator/health/{component}"*,

"templated": true

},

"info": {

"href": *"http://localhost:8080/actuator/info"*,

"templated": false

}

}

}

If you see we have only 2 endpoints showing (health, info) out of 16 endpoints

By default, all the actuator endpoints are exposed over **JMX** but only the health and info endpoints are exposed over **HTTP**.

Here is how you can expose actuator endpoints over HTTP and JMX using application properties -

**Exposing Actuator endpoints over HTTP**

# Use "\*" to expose all endpoints, or a comma-separated list to expose selected ones

management.endpoints.web.exposure.include=\*

management.endpoints.web.exposure.exclude=

**Exposing Actuator endpoints over JMX**

# Use "\*" to expose all endpoints, or a comma-separated list to expose selected ones

management.endpoints.jmx.exposure.include=\*

management.endpoints.jmx.exposure.exclude=

#### Securing Actuator Endpoints with Spring Security

Actuator endpoints are sensitive and must be secured from unauthorized access. you can add spring security to your application using the following dependency -

<dependency>

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

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

</dependency>

we can override the default spring security configuration and define our own access rules.

#### Creating a Custom Actuator Endpoint

To customize the endpoint and define your own endpoint, simply Create a classs annotate with @Endpoint URL :

**import** org.springframework.boot.actuate.endpoint.annotation.Endpoint;

**import** org.springframework.boot.actuate.endpoint.annotation.ReadOperation;

**import** org.springframework.stereotype.Component;

@Endpoint(id="helloEndpoint")

@Component

**public** **class** ListEndPoints {

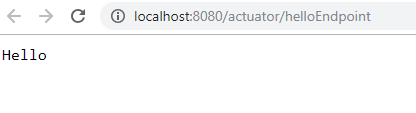
@ReadOperation

**public** String mypoint(){

**return** "Hello" ;

}

}



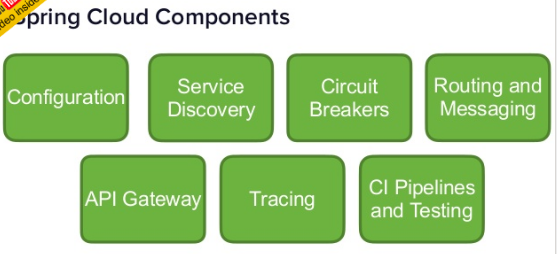
Few more Endpoints

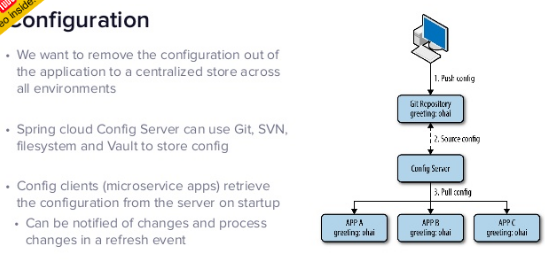


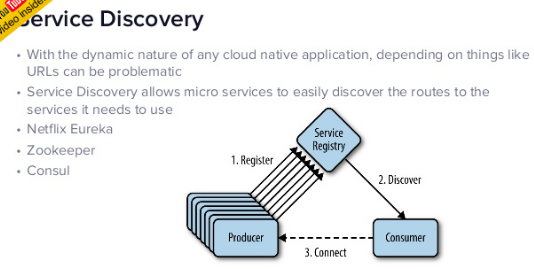
### Spring cloud

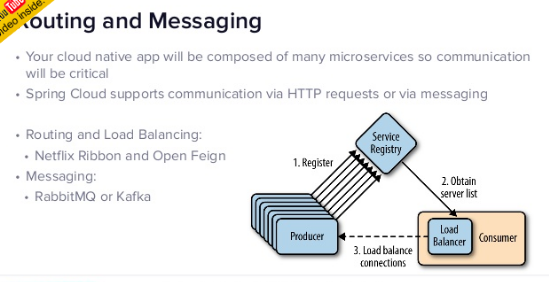
Spring Cloud focuses on providing good out of box experience for typical use cases and extensibility mechanism to cover others.

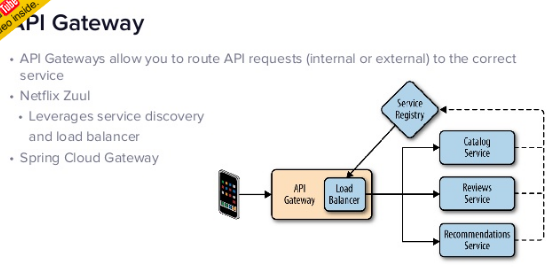
* Distributed/versioned configuration
* Service registration and discovery
* Routing
* Service-to-service calls
* Load balancing
* Circuit Breakers
* Global locks
* Leadership election and cluster state
* Distributed messaging

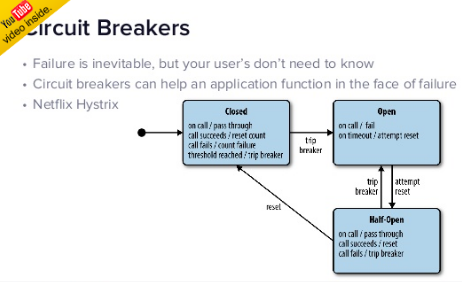


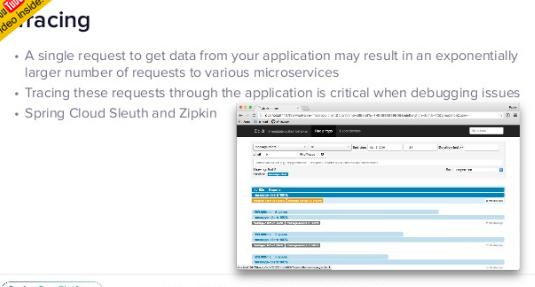


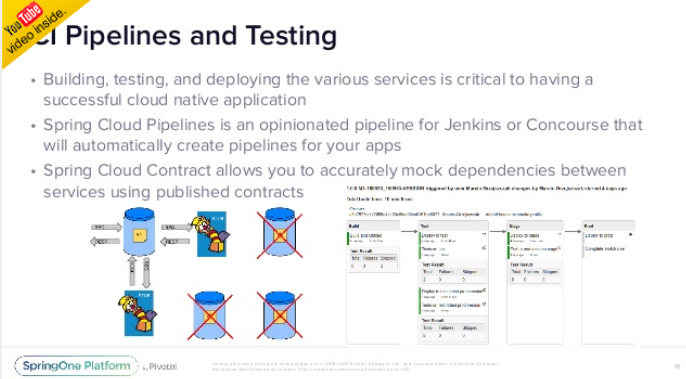


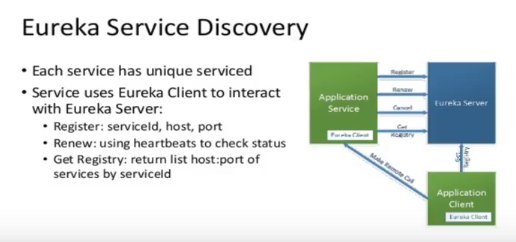


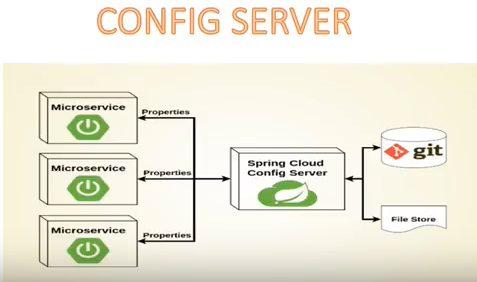








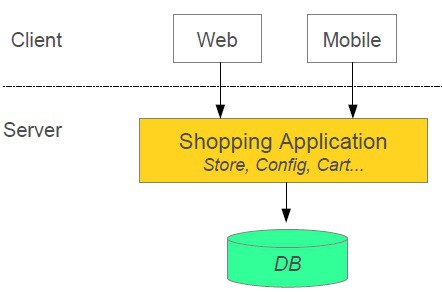




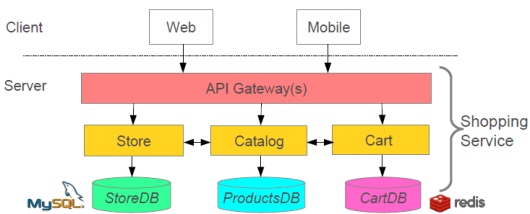
### SpringBoot MicroServices

Microservices allows us **to break our large system into the number of independent** collaborating processes.

**Shopping system without Microservices (Monolith architecture)**



**Shopping system with Microservices**

In this architecture style, the main application divided into a set of sub-applications called microservices. One large Application divided into multiple collaborating processes as below.

**Microservices Benefits**

* The smaller code base is easy to maintain.
* Easy to scale as an individual component.
* Technology diversity i.e. we can mix libraries, databases, frameworks etc.
* Fault isolation i.e. a process failure should not bring the whole system down.
* Better support for smaller and parallel team.
* Independent deployment
* Deployment time reduce

**Microservices Challenges**

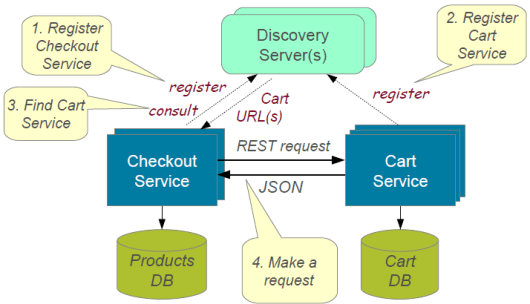
* Distributed System so hard to debug and trace the issues
* Greater need for an end to end testing
* Required cultural changes in across teams like Dev and Ops working together even in the same team.

#### Microservices Tooling Supports

**1. Use Spring for creating Microservices**

* Setup new service by using Spring Boot
* Expose resources via a RestController
* Consume remote services using RestTemplat

**2.Service Discovery**



To build a simple microservices system following steps required

1. **Creating Discovery Service (Creating Eureka Discovery Service)**
2. **Creating MicroService (the Producer),** Register itself with Discovery Service with logical service.

**3. Create Microservice Consumers** find Service registered with Discovery ServiceDiscovery client using a smart **RestTemplate**to find microservice.

#### Example:

In this Example we have 3 Micro services should communicate each other.

Each micrpservice could have seprate Server & Separate DB

**Discovery Server**

Register Service

**User**

**Server**

**Accounts**

**Server**

1**. Discovery Microservice server** – To register microservice URLS

<https://github.com/dineshonjava/discovery-microservice-server>

@EnableEurekaServer –EurekaServer Acts as a Discovery Server. To Make our class as Discovery server use @EnableEurekaServer on the top of Spring Application class

@SpringBootApplication

**@EnableEurekaServer**

public class DiscoveryMicroserviceServerApplication {

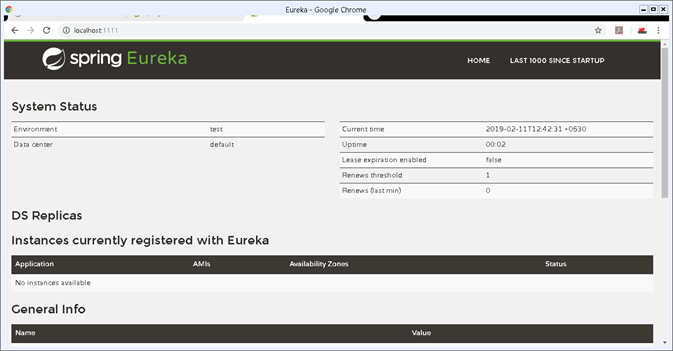
public static void main(String[] args) {

SpringApplication.run(DiscoveryMicroserviceServerApplication.class, args);

}

}

Run this Eureka Server application with right click and run as Spring Boot Application and open in browser [**http://localhost:1111/**](http://localhost:1111/)



**2. Accounts Microservice Server** – It conatins Bank Account Deatils

<https://github.com/dineshonjava/accounts-microservice-server>

**Account.java – for Account details**

public class Account implements Serializable{

private static final long serialVersionUID = 1L;

private Long amount;

private String number;

private String name;

//Setter and getters

}

**AccountController** – all these Controller urs are Registers with Discovery Service

@RestController

public class AccountController {

protected Logger logger = Logger

.getLogger(AccountController.class.getName());

@Autowired

AccountRepository accountRepository;

**@RequestMapping("/accounts")**

public Account[] all() {

logger.info("accounts-microservice all() invoked");

List<Account> accounts = accountRepository.getAllAccounts();

logger.info("accounts-microservice all() found: " + accounts.size());

return accounts.toArray(new Account[accounts.size()]);

}

**@RequestMapping("/accounts/{id}")**

public Account byId(@PathVariable("id") String id) {

logger.info("accounts-microservice byId() invoked: " + id);

Account account = accountRepository.getAccount(id);

logger.info("accounts-microservice byId() found: " + account);

return account;

}

}

To make all our controller URLS register with Discovery server, we need to annotate our main Spring Application class with ***@EnableDiscoveryClient***

@SpringBootApplication

**@EnableDiscoveryClient**

public class AccountsMicroserviceServerApplication {

public static void main(String[] args) {

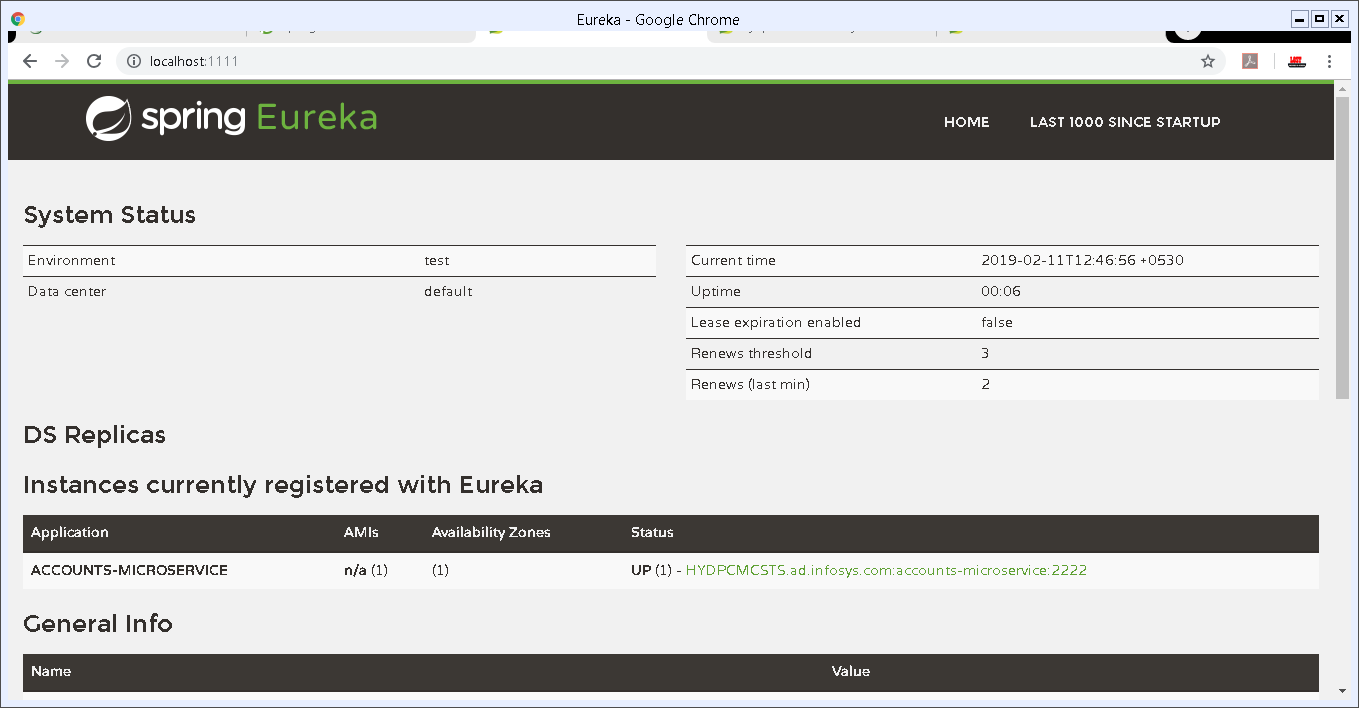
SpringApplication.run(AccountsMicroserviceServerApplication.class, args);

}

}

Now run this account service application as **Spring Boot application** and after few seconds refresh the browser to the home page of **Eureka Discovery Server**at **http://localhost:1111/** .

Now one Service registered to the Eureka registered instances with Service Name “**ACCOUNT-MICROSERVICE**” as below



**3. Webclient Mroservice server** – User will ask for Specific Account details with AccNo

<https://github.com/dineshonjava/webclient-microservice-server>

Create users to find the Producer Service registered with Discovery Service by adding ***@EnableDiscoveryClient***. This annotation also allows us to query Discovery server to find microservices

@SpringBootApplication

**@EnableDiscoveryClient**

public class WebclientMicroserviceServerApplication {

public static final String ACCOUNTS\_SERVICE\_URL = "http://ACCOUNTS-MICROSERVICE";

public static void main(String[] args) {

SpringApplication.run(WebclientMicroserviceServerApplication.class, args);

}

@Bean

@LoadBalanced

public RestTemplate restTemplate() {

return new RestTemplate();

}

@Bean

public AccountRepository accountRepository(){

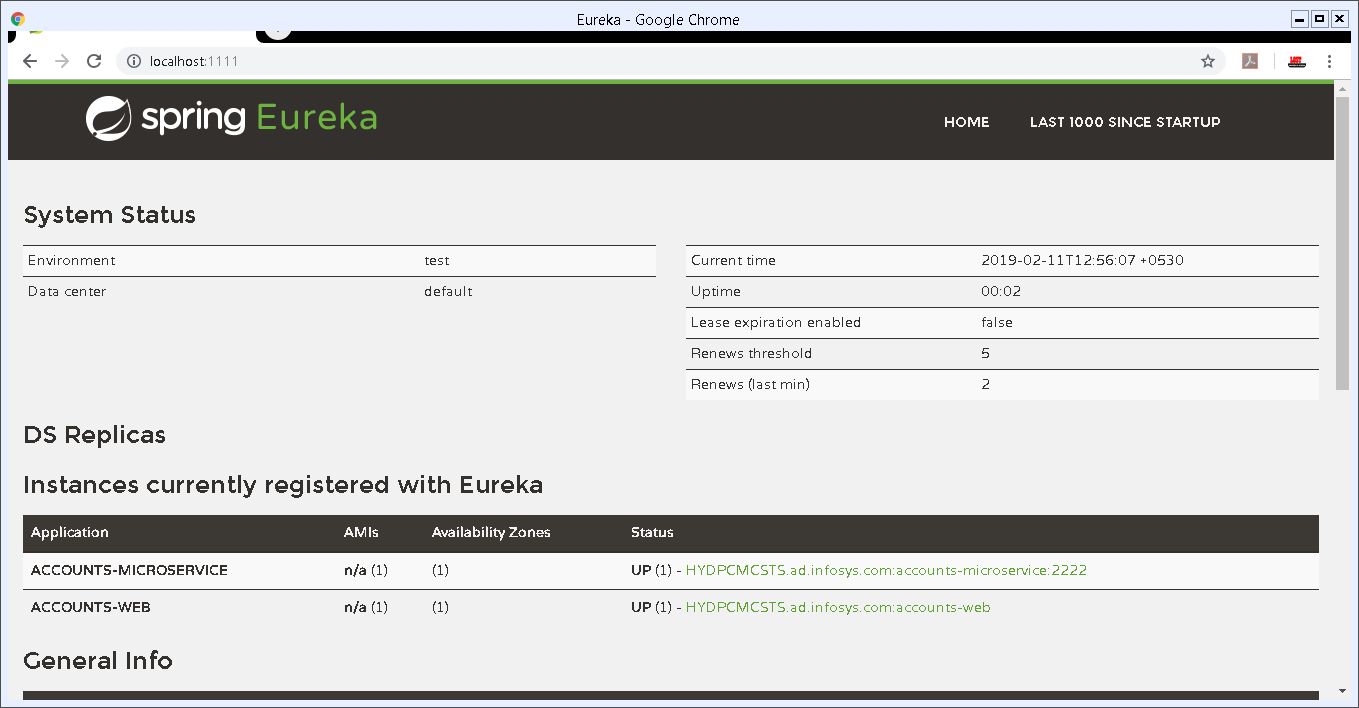
return new RemoteAccountRepository(ACCOUNTS\_SERVICE\_URL);

}

}

Now run this consumer service application as **Spring Boot application** and after few seconds refresh the browser to the home page of **Eureka Discovery Server**at [**http://localhost:1111**](http://localhost:1111)

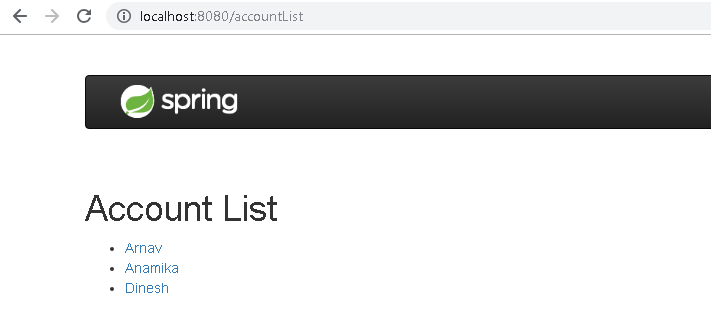
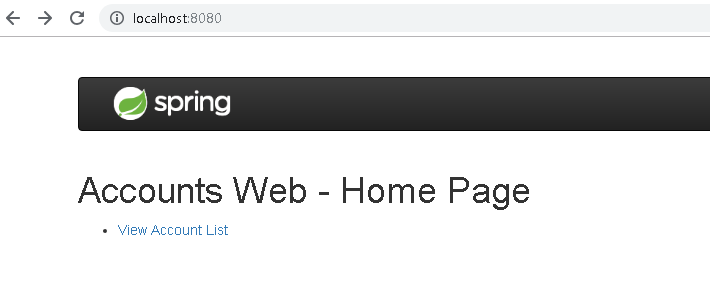
Now one more Service registered to the Eureka registered instances with Service Name “**ACCOUNTS-WEB**” as below

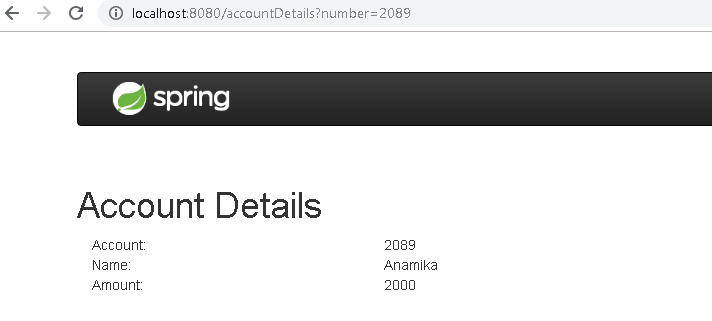


**4.Access URL Through Web Application**

We already created a Web Application in **Webclient Mroservice server** to call registred Discovery Server URLs.which is running on

**http://localhost:8080/**



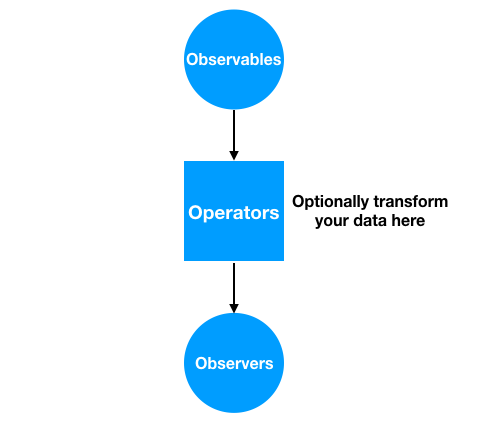


### Reactive JavaRx

* 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)

### SpringBoot – Reactive Programming

**Spring WebFlux framework is part of Spring 5 and provides reactive programming support for web applications**

**Spring WebFlux internally uses**[**Project Reactor**](http://projectreactor.io/)**and its publisher implementations –**[**Flux**](https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html)**and**[***Mono***](https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Mono.html).

<dependency>

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

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

    <version>2.0.3.RELEASE</version>

</dependency>

**We’ll now build a very simple Reactive REST *EmployeeManagement* application – using Spring WebFlux:**

* We’ll use a simple domain model – *Employee* with an *id* and a *name* field
* We’ll build REST APIs for publishing and retrieve Single as well as Collection *Employee*resources using *RestController*and *WebClient*
* And we will also be creating a secured reactive endpoint using WebFlux and Spring Security

@RestController

@RequestMapping("/employees")

public class EmployeeReactiveController {

    private final EmployeeRepository employeeRepository;

@GetMapping("/{id}")

private **Mono<Employee>** getEmployeeById(@PathVariable String id) {

    return employeeRepository.findEmployeeById(id);

}

@GetMapping

private **Flux<Employee>** getAllEmployees() {

    return employeeRepository.findAllEmployees();

}

}

**Reactive Web Client**

[*WebClient*](https://docs.spring.io/spring/docs/current/spring-framework-reference/web-reactive.html#webflux-client) introduced in Spring 5 is a non-blocking client with support for Reactive Streams.

**On the client side, we use *WebClient* to retrieve data from our endpoints created in*EmployeeController.***

public class EmployeeWebClient {

    WebClient client = WebClient.create("[http://localhost:8080](http://localhost:8080/)");

  Mono<Employee> employeeMono = client.get()

  .uri("/employees/{id}", "1")

  .retrieve()

  .bodyToMono(Employee.class);

employeeMono.subscribe(System.out::println);

Flux<Employee> employeeFlux = client.get()

  .uri("/employees")

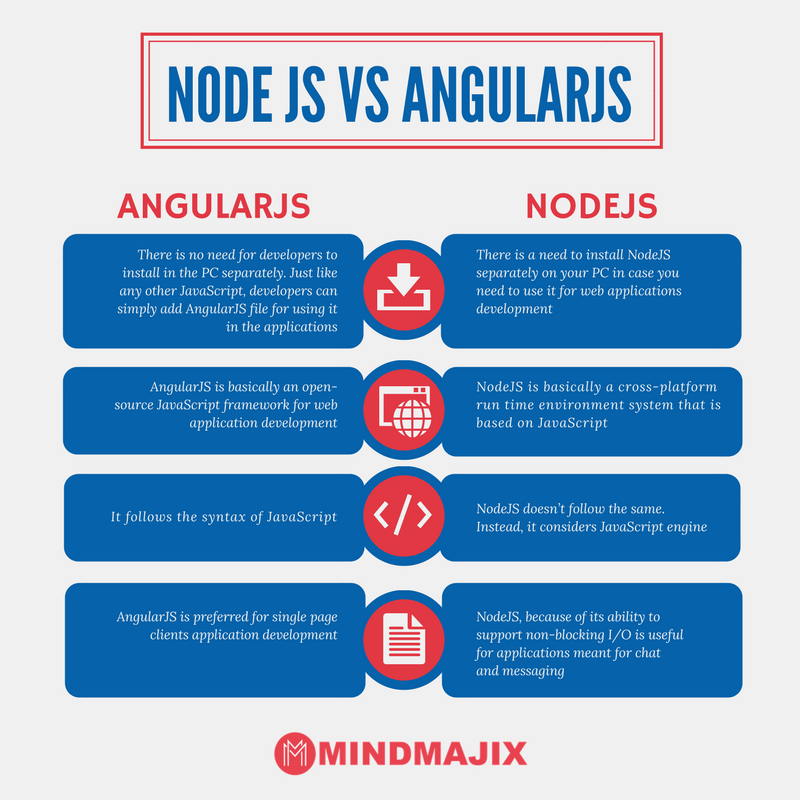
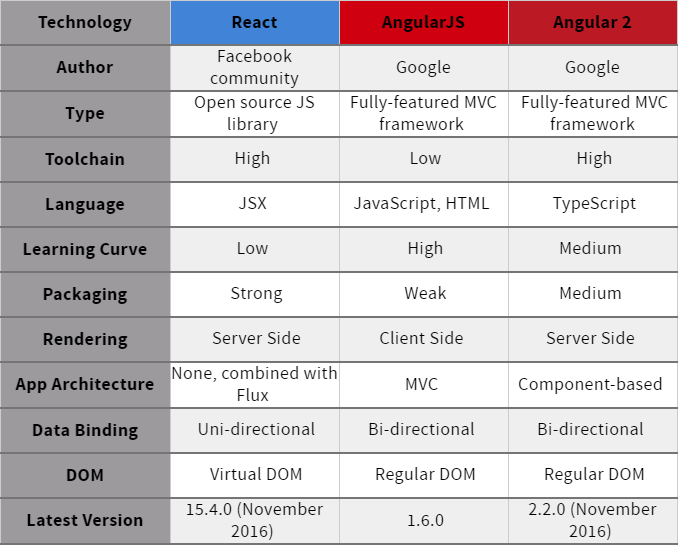
  .retrieve()

  .bodyToFlux(Employee.class);

employeeFlux.subscribe(System.out::println);

}

### 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');

// 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.**