**SPRING BOOT:**

**Spring : framework**

**Boot: bootstrap (bootstrap an appln from the scratch)**

**Problem in spring:**

It’s a huge framework

No proper best practice

Multiple config file

Multiple build and deploy

Spring Boot is

\* an opinionated

\* convention-over-configuration

\* stand-alone application

\* production-grade applications.

* Provides RAD (Rapid Application Development) features
* It highly dependent on starter template
* Spring Boot starters are templates that contain a collection of all the relevant **transitive** **dependencies** that are needed to start a particular functionality.

**Why to use starter template :**

leaves the chances of version conflict which ultimately result in more runtime exceptions.

**Setup environment:**

1. >Than Java 1.8
2. STS

Spring boot using maven:

Choose the archetype as web-app

Then add below dependency to pom file

<!-- Parent pom is mandatory to control versions of child dependencies -->

<parent>

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

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

    <version>2.0.4.RELEASE</version>

    <relativePath />

</parent>

<!-- Spring web brings all required dependencies to build web application. -->

<dependency>

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

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

</dependency>

The generated project relies on the Boot parent.

No need to specify the version.

#### @ComponentScan

This annotation provides support parallel with Spring XML’s context:component-scan element.

**@SpringBootConfiguration** (in spring boot2)

[@SpringBootConfiguration](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/SpringBootConfiguration.html) is new annotation in Spring boot 2. Previously, we have been using [@Configuration](https://docs.spring.io/spring-framework/docs/5.0.4.RELEASE/javadoc-api/org/springframework/context/annotation/Configuration.html) annotation. You can use @Configuration in place of this. Both are same thing.

@SpringBootApplication as our primary application configuration class; behind the scenes, that’s equivalent to @Configuration,

@EnableAutoConfiguration, and @ComponentScan together.

application.properties file , where we can write config details

when we use run() method is called it does

1. Sets up default config (convention)
2. Starts up application context
3. Performs class path scan
4. Starts tom cat embedded server

**NOTE**

**The auto restart of server inside eclipse is not supported before the spring boot 1.3.0 version.**

<dependency>

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

<artifactId>spring-boot-devtools</artifactId>

</dependency>

<https://howtodoinjava.com/spring-boot2/developer-tools-module-tutorial/>

spring-boot-starter-parent:

this has the information about default version to use for all libraries

**Spring Boot starters** are templates which helps to download all dependency jar needed

If still u need to change the version of jars we make use of properties tag

For eg:

spring-boot-starter-data-jpa -> downloads spring-orm, hibernate-entity-manager, spring-data-jpa

working with command line args in springboot:

<https://memorynotfound.com/spring-boot-passing-command-line-arguments-example/>

Spring configuration property :

<https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-Configuration-Binding>

<https://github.com/chuchip/servercloudconfig>

<https://www.petrikainulainen.net/programming/spring-framework/spring-from-the-trenches-injecting-property-values-into-configuration-beans/>

**Springboot Environment**

## Externalized Configuration

Spring Boot lets you externalize your configuration so that you can work with the same application code in different environments. You can use properties files, YAML files, environment variables, and command-line arguments to externalize configuration. Property values can be injected directly into your beans by using the @Valueannotation, accessed through Spring’s Environment abstraction, or be [bound to structured objects](https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-external-config.html#boot-features-external-config-typesafe-configuration-properties) through @ConfigurationProperties.

## Configuring Random Values

The RandomValuePropertySource is useful for injecting random values (for example, into secrets or test cases). It can produce integers, longs, uuids, or strings, as shown in the following example:

my.secret=${random.value}

my.number=${random.int}

my.bignumber=${random.long}

my.uuid=${random.uuid}

my.number.less.than.ten=${random.int(10)}

my.number.in.range=${random.int[1024,65536]}

**Application.properties:**

we will use these .property files for writing  environmental (server) related stuff and for even for some other reasons,

## Need for Application Configuration

Configuration for applications vary from one environment to another

* You would want to connect to a different database or queues
* You would want to connect with different services
* You would want to configure less logging in production
* You might want to have different custom configuration

<https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html>

<https://blog.indrek.io/articles/using-environment-variables-with-spring-boot/>

**Spring Data :**

Spring Data is a part of [**Spring Framework**](http://www.javaguides.net/p/spring-framework.html)

**Spring data jpa:**

Managing data between java classes or objects and the relational database is a very cumbersome and tricky task.

The DAO layer usually contains a lot of boilerplate code that should be simplified in order to reduce the number of lines of code and make the code reusable.***( reduce the amount of boilerplate code required to implement data access layers for various persistence stores.)***

Spring Data JPA is not a JPA provider, JPA provider (like Hibernate)

### When to use Spring Data JPA?

### 

### We need to understand first **What is JPA(JPA 2.2:)?**

 Java Persistence API is the Java specification for accessing, managing and persisting data between Java classes or objects and relational database.

It contains set of interfaces which need to be implemented. It is a framework that provides an extra layer of abstraction on the JPA implementation

Spring Data Commons : has following repo :

1. Repository<T, ID extends Serializable> interface
2. CrudRepository<T, ID extends Serializable> interface
3. PagingAndSortingRepository<T, ID extends Serializable> interface
4. QueryDslPredicateExecutor interface

***The generic behavior of these interface sMakes them to be typesafety and tied with one bean ie entity alone.***

### Spring Data JPA Interfaces

 enhanced support for JPA based data access layers.

Spring Data JPA project provides the following interfaces:

* JpaRepository<T, ID extends Serializable> interface
* JpaSpecificationExecutor interface

**Difference between Spring jpa and jpa:**

<https://stackoverflow.com/questions/16148188/spring-data-jpa-versus-jpa-whats-the-difference>

Links for Spring JPA data:

<https://dzone.com/articles/spring-data-jpa-1>

<https://www.javaguides.net/p/spring-data-jpa-tutorial.html>

<https://www.javaguides.net/2018/11/spring-data-jpa-tutorial-getting-started.html>

<https://dzone.com/articles/spring-boot-with-spring-data-jpa>

**create a project using 1.5.2 spring**

**latest version of mysql – 8.0**

**mysqlconnector maven –**

<dependency>

<groupId>mysql</groupId>

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

<version>8.0.16</version>

<scope>runtime</scope>

</dependency>

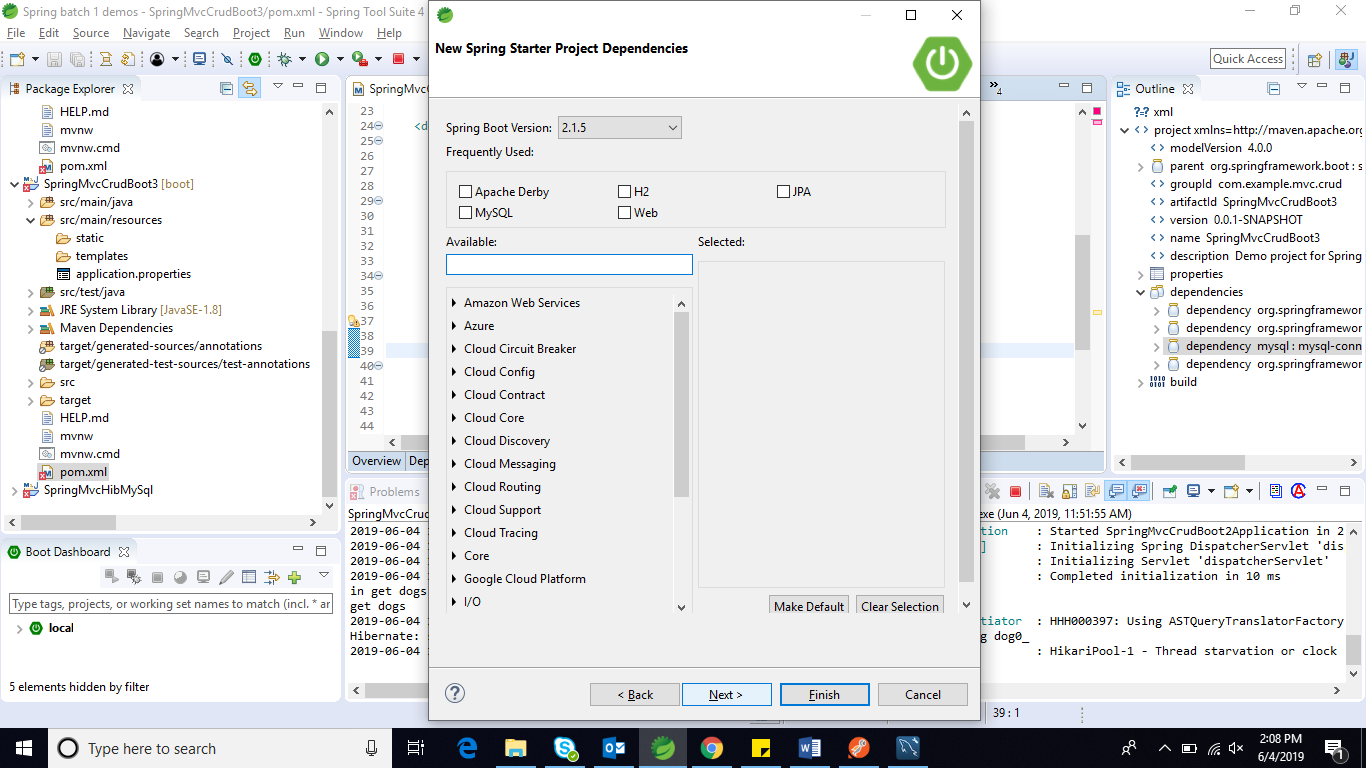
**Dialect to be used is ->** org.hibernate.dialect.MySQL5Dialect

**Create a spring starter project in STS**

**Let**

**packaging ->** jar

**java version->**1.10

select needed dependency from 

select the needed dependency, based on the project requirement.

For our data jpa , we need jpa, mysql and web

NOTE:

For this using java version as 10, we have to use below snippet in pom.xml

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<maven.compiler.source>1.8</maven.compiler.source>

<maven.compiler.target>1.8</maven.compiler.target>

</properties>

Then right click project-> maven->update

Ensure the below 2 dependency are there in pom.xml

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

And below one for database

<dependency>

<groupId>mysql</groupId>

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

<version>8.0.16</version>

<scope>runtime</scope>

</dependency>

Once this is done , we have to configure jpa info in application.properties.

Application properties -> key value paired, has configuration details for server and environment.

spring.jpa.hibernate.ddl-auto=update

spring.datasource.url=jdbc:mysql://localhost/mydb

spring.datasource.username=root

spring.datasource.password=root

spring.jpa.database-platform=org.hibernate.dialect.MySQL5Dialect

spring.jpa.show-sql=true

then create a rest controller and also inject service object

@RestController

@RequestMapping("/dogs")

**public** **class** DogController {

@Autowired DogService service;

@GetMapping

**public** List<Dog> getDogs() {

System.***out***.println("in get dogs...");

**return** service.getDogs();

}

@PostMapping

**public** **void** postDogs(@RequestBody Dog dto) {

service.add(dto);

}

@GetMapping("/{id}")

**public** Dog getById(@PathVariable(required = **true**) **int** id) {

**return** service.getDogById(id);

}

@DeleteMapping("/{id}")

**public** **void** delete(@PathVariable(required = **true**) **int** id) {

service.delete(id);

}

}

In Service Layer:

@Service

**public** **class** DogService {

**public** DogService() {

// **TODO** Auto-generated constructor stub

}

@Autowired

DogRepo repository;

**public** **void** add(Dog dto) {

repository.save(toEntity(dto));

}

**public** **void** delete(Integer id) {

repository.deleteById(id);

}

**public** List<Dog> getDogs() {

System.***out***.println("get dogs");

**return** (List<Dog>) repository.findAll();

}

**public** Dog getDogById(Integer id) {

Optional<Dog> optionalDog = repository.findById(id);

**return** optionalDog.orElseThrow(() -> **new** DogNotFoundException("Couldn't find a Dog with id: " + id));

}

**private** Dog toEntity(Dog dto) {

Dog entity = **new** Dog();

entity.setName(dto.getName());

entity.setAge(dto.getAge());

**return** entity;

}

}

In dao :

@Repository

**public** **interface** DogRepo **extends** CrudRepository<Dog, Integer> {}

also create a model class (entity)

and create a with main method and keep annotation with spring boot

@SpringBootApplication(scanBasePackages="com.example.mvc.crud")

Asume that there is an existing table, use @column annotation etc for ORM

<https://www.callicoder.com/hibernate-spring-boot-jpa-one-to-many-mapping-example/>

<https://www.callicoder.com/hibernate-spring-boot-jpa-one-to-one-mapping-example/>

spring data jpa - Query criteria

<https://javadeveloperzone.com/spring/spring-jpa-dynamic-query-example/>

<http://zetcode.com/springboot/datajpaquery/>

deploying application through command prompt:

1. open the current application path in command prompt
2. add the below jaxb dependency
3. use mvn clean install
4. then move the path to target folder (cd target)
5. type java jar <jarfilename>.jar (this you can physically find in target folder of the project)

<!-- https://mvnrepository.com/artifact/javax.xml.bind/jaxb-api -->

<dependency>

<groupId>javax.xml.bind</groupId>

<artifactId>jaxb-api</artifactId>

<version>2.2.11</version>

</dependency>

**SPRING ACTUATORS:**

* **Mechanical device for moving or controlling something**
* **Expose operational information about the running application**
* production grade tools which need not be implement by us

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

In 1.x Actuator follows a R/W model, that means we can either read from it or write to it. E.g. we can retrieve metrics or the health of our application. Alternatively, we could gracefully terminate our app or change our logging configuration.

To enable Spring Boot Actuator, we need to add this in pom.xml

<dependency>

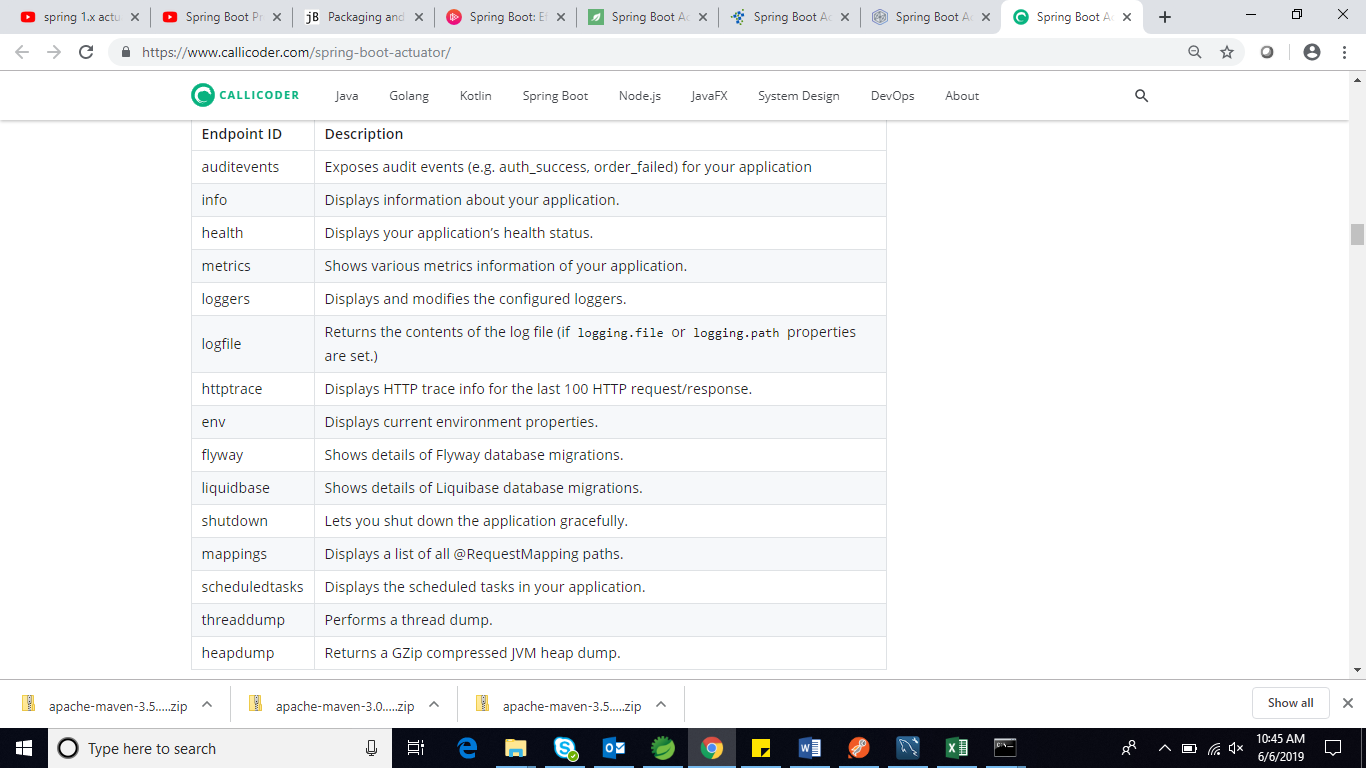
<groupId>org.springframework.boot</groupId>

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

</dependency>

In order to get it working, Actuator requires Spring MVC to expose its endpoints through HTTP.

### **Endpoints:**



If an application is healthy it will show **status : up** else **down -> /health**

actuator/info-> general information about your application obtained from build files like META-INF/build-info.properties or Git files like git.properties

**Disabling or Enabling Spring Endpoints :**

**In application.properties**

management.endpoint.shutdown.enabled=true

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

Custom endpoints :

<https://www.javadevjournal.com/spring-boot/spring-boot-actuator-custom-endpoint/>

**Spring basic authentication:**

<https://www.devglan.com/spring-security/spring-boot-security-rest-basic-authentication>

<https://o7planning.org/en/11649/secure-spring-boot-restful-service-using-basic-authentication>

**NOTE:**

https://www.infoq.com/presentations/spring-boot-actuator/ ---> spring boot 2.0

**SPRING DEV Tools**

**Live reloading** reloads or refreshes the entire app when a file changes. For example, if you were four links deep into your navigation and saved a change, live reloading would restart the app and load the app back to the initial route.

**Hot reloading** only refreshes the files that were changed without losing the state of the app. For example, if you were four links deep into your navigation and saved a change to some styling, the state would not change, but the new styles would appear on the page without having to navigate back to the page you are on because you would still be on the same page.

Adding spring-boot-devtools in a project is as simple as adding any other spring-boot module.

<dependency>

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

    <artifactId>spring-boot-devtools</artifactId>

</dependency>

**Auto-restart:**

In a typical application development environment, a developer would make some changes, build the project and deploy/start the application for new changes to take effect

Using spring-boot-devtools, this process is also automated.

(Whenever files change in the classpath, applications using spring-boot-devtools will cause the application to restart. The benefit of this feature is the time required to verify the changes made is considerably reduced)

**Live-reload:**

spring-boot-devtools module includes an embedded LiveReload server that is used to trigger a browser refresh when a resource is changed.

**SPRING Cloud Config**

<https://o7planning.org/en/11723/understanding-spring-cloud-config-server-with-example>

<https://o7planning.org/en/11727/understanding-spring-cloud-config-client-with-example>