Spring Framework and Spring Boot is completely different.

**Spring Framework:**

Spring framework is collection of libraries which we needed.

Spring Framework features:

POJO (Plain Old Java Object)

Dependency Injection

MVC (Model View Control)

REST (Representational State Transfer)

Security

Batch

Data

AOP (Aspect Oriented Programming)

……

**Spring Boot:**

Spring boot enables the developers to work with spring framework. i.e. it provides a way. By using this, we can configure the spring framework. It will setup all the needed libraries of our project.

Eg:

<dependency>

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

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

</dependency>

Spring boot have its own embedded server. We no need to setup any server separately.

(*Manual Configuration*)If we have to change the configuration. Here, no such *.xml files* are there instead here we can configure by using the **application.properties** file.

**Dependency Injection:**

We use some of the objects directly without defining the corresponding classes. But, we need that classes. So, for that some services will give all the classes needed here.

**Usage of *Autowired* Dependency Injection:**

Eg: @Component

Class Laptop class Harddrive

{ {

@Autowired

Harddrive harddrive1; }

}

**Spring Container Concept:**

Once the Spring project created and run. Spring Container gets created.

Spring Container

Alien object

Spring container is get created when we run the Application\_name.java file.

Spring container can able to store the objects of different classes. So, we can access it anywhere.

Creation of Alien’s object inside the Spring Container:

Spring Container

Alien object

package com.logesh.DependencyInjection;

import org.springframework.stereotype.Component;

// if we set a @Component annotation then ONE Object(SingleTon object) will created inside the

//...spring container(i.e. bean). We can access it anyplace without instantiating the class

@Component

public class Alien {

private int aid;

private String aname;

public int getAid() {

return aid;

}

public void setAid(int aid) {

this.aid = aid;

}

public String getAname() {

return aname;

}

public void setAname(String aname) {

this.aname = aname;

}

public void hello() {

System.out.println("Alien CLass");

}

}

Accessing the Alien object using ‘CongigurableApplicationContext.class’:

package com.logesh.DependencyInjection;

import java.util.Collections;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.ConfigurableApplicationContext;

@SpringBootApplication

public class DependencyInjectionApplication {

public static void main(String[] args) {

SpringApplication app = new SpringApplication(DependencyInjectionApplication.class);

app.setDefaultProperties(Collections.singletonMap("server.port", "8181"));

ConfigurableApplicationContext context = app.run(args);

System.out.println("Hello world");

// 'context' is used to access the SPRING CONTAINER. Inside the spring container

// object/bean is created automatically when application starts running. We can get that

// bean by using getBean() method provided by the 'ConfigurableApplicationContext.class'

**Alien alien1 = context.getBean(Alien.class);**

alien1.hello();

// after creating Laptop.java

//

}

}

Creating another object inside the SPRING CONTAINER:

Autowire

Spring Container

Laptop object

Alien object

*Laptop.java:*

package com.logesh.DependencyInjection;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Component;

// if we set a @Component annotation then ONE Object(SingleTon object) will created inside the

//...spring container(i.e. bean). We can access it anyplace without instantiating the class

@Component

public class Alien {

// here now, Alien class depends on Laptop class so. How alien class get the object

//...inside the container i.e. laptop object inside container. Autowire is the solution for that

@Autowired

private Laptop laptop;

private int aid;

private String aname;

public int getAid() {

return aid;

}

public void setAid(int aid) {

this.aid = aid;

}

public String getAname() {

return aname;

}

public void setAname(String aname) {

this.aname = aname;

}

public void hello() {

System.out.println("Alien CLass");

laptop.hai();

}

}

***We can also specify the name for the object which is saved inside the SPRING CONTAINER:***

Setting name to the object ***=*** @Component(“name of the object”)

Getting name object by using the name ***=*** @Qualifier(“name of the object”)

**@Responsebody annotation:**

@RequestMapping(‘/home’)

@Responsebody

public String createResponse() {

return “Logu”;

}

If we access the URL <http://localhost:8181/home>. Directly we get the response as ‘Logu’.

**Handling with HttpServletRequest, HttpSession:**

**URL:** <http://localhost:8181/home?name=logesh>

@RequestMapping(‘/home’)

public String getHomepage(HttpServletRequest req, HttpSession session) {

String nameStr = **req.name;**

**session.addAttribute(“name”, nameStr);**

return “index”;

}

Index.jsp:

<h1>${name}</h1>

**Model and View:**

‘ModelAndView’ is a default class that is supported in Spring Boot Application. By using this we can pass the ‘views’[i.e. jsp/thymeleaf] and ‘data’ from the Controller to ‘views’.

Syntax:

ModelAndView mv = new ModelAndView();

// adding to the variable ‘name’ DATA

String data = “Logesh”

mv.addObject(“name”, data);

//adding VIEWS

mv.setViewName(“index”);

**If we want to get MORE NUMBER of data. We can follow the following way.**

Alien.java:

Public class Alien {

private int aid;

private String aname;

getters and setters….

}

In Controller:

public void functionNAme(Alien alien) {

ModelAndView mv = new ModelAndView();

mv.addObject(alien);

}

**Querying:**

By using CRUD Repository or some other repository we can use the default methods provided by those repositories as mentioned below:

Eg: repo.findById();

And also we can fetch data by using the fields which are present in the database:

FORMAT: ‘findBy’ + ‘field\_name(with starting letter in uppercase)’

Eg: repo.findByEmployeeName();

Should include this method as ABSTRACT METHOD inside the repository:

Eg: Repository.java file:

Return\_type findByEmployeeName();

But other than that we have to run a complex query also. So, we have another way to specify our queries i.e. @Query annotation

Eg:

?1 🡺 First parameter in below method.

@Query(“from employee where employeeName=?1”)

public void findByEmployeeName(String employeeName) {

….

}

**REST API:**

@GetMapping(‘/home/{id}’)

public void getData(@Pathvariable(“id”) int id)

{

repo.findById(id);

….

}

@DeleteMapping(‘/home/{id}’)

public void deleteData(@Pathvariable(“id”) int id)

{

repo.deleteById(id);

….

}

@PostMapping(‘/home’)

public void postData(@RequestBody Alien alien)

{

repo.save(alien);

….

}

@PutMapping(‘/home’)

public void saveOrupdateData(@RequestBody Alien alien)

{

repo.save(alien);

….

}

**What is Content Negotiation?**

Server is responsible for sending the responses for the request made by the clients. Sometimes, client needs the response in a specific format i.e. JSON/ XML/ PDF format. Server should make response in all these formats.

Restricting/ appending the specific contents.

In **Spring boot**, response will send as a JSON because ‘jackson’ Maven library will take care of this. If we need XML we have to add one dependency called ‘jackson-dataformat-xml’.

We can **restrict** the data format by using the following way:

@RequestMapping(‘/home’) 🡺 @RequestMapping(path=‘/home’)

So,

@RequestMapping(path=‘/home’, produces=‘application/xml’) 🡺 this will only gives the ‘XML’ response only.

**Spring Data REST:**

If we use this, we don’t need to setup the REST API’s separately it is automatically get configured. i.e. Here, we don’t need any controller to setup the API and all other stuffs present in the controller.

Eg:

In Repo.java class:

@RepositoryRestResource(collectionResourceRel = "employee", path="employee")

public class EmployeeRepo extends JpaRepository<Employee, Integer> {

}

If we setup the above things. We can perform all the CRUD operations using the URLs. The URLs itself generated by Spring Data Rest.

Many more things to explore in **Spring Data Rest…….**