Problem with traditional spring approach

We use different modules from spring such as

Core module – to do dependency injection

MVC Module - to develop the web layer for our application, to develop restful web service Layer

DAO layer - where we use the spring JDBC, which makes our life easy to develop a data access layer for our application

ORM tools - when we are using ORM tools like Hibernate we can use spring data JPA

Typically we use XML based configuration or annotations based configuration.

In spring

* we need to make sure that each of these models is available for our application by defining

all the dependencies in the Maven pom.xml

* And at runtime we can be sure that these versions of various Modules that we use are compatible with each other
* And once we have all that in place we will build our application and will have to deploy to a external web

Spring boot will do all this for us for free and we can develop our applications rapidly and even deploy them

Spring boot features

1. Auto configuration

* If we are using mvc to develop web application or restful

Springboot will automatically configure the dispatcher servlet and do all the request mapping

We don’t have to do any xml or annotation based configuration to configure the servlet

* if you are using spring data or object relational mapping while working with tools like Hibernate

to perform database crud we no longer have to configure the data source or even the transaction manager.

Spring boot will automatically configure these for our application.

1. Spring boot starters

* Spring boot starters had taken off the problem of module availability and version compatibility
* You simply include the single dependency in your maven pom.xml and all the other libraries will be pulled accordingly.
* And also the correct versions of those libraries will be included because all that version information is available in this spring boot

1. Embedded server Container

* We don’t have to worry about deploying our application to external server
* By default TOMCAT server is used we can also change the server to jetty or undertow

1. Spring boot actuator

* spring boot gives us a lot of hea;th checks of our application for free
* through the spring boot actuators, so we can use different types of health checks that come for free
* and we can use these even on production when our application is running on production.

Spring boot In action

* Open STS
* File->new->spring starter project->
* name: hello

group: com.sau.hello

description:

package: com.sau.hello

->next->search web(below select spring web)->finish

(when we are developing the restful layer select spring web)

Create new helloWorld project

* Hellospringboot

Src/main/java

Create new class in com.sau.springboot package

* url: localhost:8080/hello

Note 1

* Types of project: 1. Maven project

2. Graddle project

* Packaging types: 1. Jar

2. War

* Java versions showing here in STS: 8, 11, 17
* Languages:

1. Java
2. Kotlin
3. Groovy

* Spring boot version using is : 2.6.3

Note 2

* when we are developing the restful layer select spring web
* for in-memory-database select H2
* MySQL database

Maven Project

Maven project structure

1. Src/main/java : source code is present
2. Src/main/resources : resource files present(application.properties)
3. Src/test/java : here all our test will go
4. pom.xml: maven project related information present

pom.xml

* spring-boot-starter-parent

it has all the version information, in effectivePOM File, search version, in properties tab all version information is present

* spring-boot-starter

all the required dependency library files will gets downloaded automatically

we can see these dependencies in Maven dependencies

* when we add spring web

spring-boot-starter-web this dependency will gets ad in the pom.xml

and all the required libraries will gets downloaded automatically

Create Spring boot project

* we are exploring the spring core dependency injection module
* name: core

group: com.sau.core

description: Exploring Spring Boot and Core

package: com.sau.core

* by default selected

type of project: maven

language: java

java version: 11

packaging: Jar

next->finish

* Src/main/java -> com.sau.core -> CoreApplication

below gives the details about the created project

**package** com.sau.core;

**import** org.springframework.boot.SpringApplication;

**import** org.springframework.boot.autoconfigure.SpringBootApplication;

//@SpringBootApplication this is the top level annotation

// this has several other annotations in it // when you put a cursor on it you can see those annotations

//@SpringBootConfiguration, @EnableAutoCongiguration, @ComponentScan() these are important annotations in this annotation

// @SpringBootApplication this annotation will do all @SpringBootConfiguration, @EnableAutoCongiguration, @ComponentScan() for our com.sau.core package

@SpringBootApplication

//CoreApplication, class name, Core is derived from projct name and appends application to it

**public** **class** CoreApplication {

// below is main method //this is where the execution of spring boot application begins

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

//SpringApplication.run // run is a static method within springBoot

//in run method, we will pass this class name CoreApplication as a parameter

// we can pass any class here which is marked with annotation @SpringBootApplication

SpringApplication.*run*(CoreApplication.**class**, args);

}

}

/\*@SpringBootConfiguration -> tells spring boot that this class here can have several beans definitions,

so that those beans will be available at the run time\*/

/\*@EnableAutoCongiguration -> tells spring boot to automatically configure the spring application based on the dependencies\*/

/\* @ComponentScan()

tells spring boot that scan this package, and figure out all the spring beans and create beans for us

it will automatically scan all the sub packages of this package com.sau.core

\*/

@SpringBootConfiguration ->

* tells spring boot that this class here can have several beans definitions, so that those beans will be available at the run time

@EnableAutoConfiguration ->

* tells spring boot to automatically configure the spring application based on the dependencies that we see on the class path
* when we add a simple starter to our maven project, automatically all the dependencies will pulled up, so we are telling spring boot to automatically add all dependencies and do any configuration that is required
* e.g spring web => if we have spring web in out dependencies then spring boot will automatically create the dispatcher servlet on all the configuration files

e.g HSQLDB => if we have this in-memory-database dependency in pom.xml

automatically spring boot will configure the data source required to connect to the database and to work with this data base

e.g MySQL => if we have this dependency in pom.xml, spring boot will configure the data source

* all the xml and java based configuration is now gone5

@ComponentScan()

* tells spring boot that scan this package, and figure out all the spring beans and create beans for us
* it will automatically scan all the sub packages of this package com.sau.core
* Src/test/java -> com.sau.core -> CoreApplicationTests

package com.sau.core;

import org.junit.jupiter.api.Test;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.boot.test.context.SpringBootTest;

import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)//by using this annotation we are using SpringRunner instead of using default J-unit runner

@SpringBootTest //tells spring boot to go and search for annotation @SpringBootApplication

// and create a spring container with the beans

class CoreApplicationTests {

@Test // each test is marked with @test in J-unit

// so when we run this test class, we can use all the beans that are available from the spring container

void contextLoads() {

}

}

Do dependency injection

* @Autowired
* // tells spring that this dependency should be automatically wired at runtime
* // spring search for the implementation class(i.e PaymentDAOImpl) of this PaymentDAO dao interface at runtime
* // it will create bean of PaymentDAOImpl because it is already marked with a stereotype annotation(i.e @Repository)
* // and inject it into this field here

In src/main/java

1. create package -> com.sau.core.dao

create interface-> PaymentDAO

create class -> PaymentDAOImpl

##PaymentDAO

**package** com.sau.core.dao;

**public** **interface** PaymentDAO {

}

## PaymentDAOImpl

**package** com.sau.core.dao;

**import** org.springframework.stereotype.Repository;

// we use PaymentDAOImpl class for the data access

@Repository // we use @Repository for the classes that handle the database

// this is a stereotype annotation

//all the stereotype annotations tells spring that a bean of this class can be created and injected

**public** **class** PaymentDAOImpl **implements** PaymentDAO {

}

1. create package -> com.sau.core.service

create interface-> PaymentService

create class -> PaymentServiceImpl

##PaymentService

**package** com.sau.core.service;

**public** **interface** PaymentService {

}

## PaymentServiceImpl

package com.sau.core.service;

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

import org.springframework.stereotype.Service;

import com.sau.core.dao.PaymentDAO;

@Service //all the stereotype annotations tells spring that a bean of this class can be created and injected

public class PaymentServiceImpl implements PaymentService {

@Autowired // tells spring that this dependency should be automatically wired at runtime

// spring search for the implementation class(i.e PaymentDAOImpl) of this PaymentDAO dao interface at runtime

// it will create bean of PaymentDAOImpl because it is already marked with a stereotype annotation(i.e @Repository)

// and inject it into this field here

private PaymentDAO dao;

public PaymentDAO getDao() {

return dao;

}

public void setDao(PaymentDAO dao) {

this.dao = dao;

}

public int calculate(int a , int b)

{

return a+b;

}}

1. Test class src/test/java

**package** com.sau.core;

1. **import** **static** org.junit.jupiter.api.Assertions.*assertNotNull*;
2. **import** org.junit.jupiter.api.Test;
3. **import** org.junit.runner.RunWith;
4. **import** org.springframework.beans.factory.annotation.Autowired;
5. **import** org.springframework.boot.autoconfigure.SpringBootApplication;
6. **import** org.springframework.boot.test.context.SpringBootTest;
7. **import** org.springframework.test.context.junit4.SpringRunner;
8. **import** com.sau.core.service.PaymentService;//inject payment service into the test class
9. **import** com.sau.core.service.PaymentServiceImpl;
10. **import** **static** org.junit.Assert.\*;
11. @RunWith(SpringRunner.**class**)//by using this annotation we are using SpringRunner instead of using default J-unit runner
12. @SpringBootTest //tells spring boot to go and search for annotation @SpringBootApplication
13. // and create a spring container with the beans
14. **class** CoreApplicationTests {
16. //inject payment service into the test class
17. @Autowired
18. PaymentServiceImpl services;
19. @Test // each test is marked with @test in J-unit
20. // so when we run this test class, we can use all the beans that are available from the spring container
21. **void** testDependencyInjection () {
22. *assertNotNull*(services.getDao());
23. }
25. @Test
26. **void** calculateSum () {
27. *assertEquals*(12,services.calculate(7, 5));
28. }
29. }

**Spring Data JPA**

**Introduction**

* **JPA** stands for java persistence API
* It is a standard from oracle, to use the tools like hibernet, eclipse link etc. to do object relational mapping
* To perform crud operations against the databases simply create repository interface per entity
* these repository interfaces extends interfaces from the spring data JPA repository

JpaRepository

ProductRepository

* the starter project to use spring data JPA is spring-boot-starter-data-jpa

step1-include spring-boot-starter-data-jpa dependency in pom.xml

step2-add the database JDBC driver in the pom.xml

<artifactId>h2</artifactId>

If we are using in-memory or embedded databases like h2, HSQL, DERBY then spring will automatically create all the data access layer related configuration

* when we use mysql or oracle data base

there we need to provide the jdbc connection url,user name and password to connect to the database in the application.properties

**Create a Project**

File->new->springStarterProject(fill names)->finish->select dependency(h2)->finish

* in pom.xml this dependency gets added

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<scope>runtime</scope>

</dependency>

* **add one more dependency in pom.xml**

<dependency>

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

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

</dependency>

* **by default hibernet is used as a ORM implementation**

**Create Model Class**

* **every Jpa entity should have at least one @ID and class should be marked with @Entity**

**package** com.sau.springdatajpa.entities;

**import** javax.persistence.Entity;

**import** javax.persistence.Id;

@Entity // to make this class a jpa entity

**public** **class** Student {

@Id // this below id field is typically a primary key in the data base // so mark with @Id annotation

// every Jpa entity should have at least one @ID and class should be marked with @Entity

**private** **long** id;

**private** String name;

**private** **int** testScore;

**public** **long** getId() {

**return** id;

}

**public** **void** setId(**long** id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **int** getTestScore() {

**return** testScore;

}

**public** **void** setTestScore(**int** testScore) {

**this**.testScore = testScore;

}

}

**Create student Repository interface**

**package** com.sau.springdatajpa.repos;

**import** org.springframework.data.jpa.repository.JpaRepository;

**import** com.sau.springdatajpa.entities.Student;

**public** **interface** StudentRepository **extends** JpaRepository<Student, Long> {

}

**Testing**

**package** com.sau.springdatajpa;

**import** org.junit.jupiter.api.Test;

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

**import** org.springframework.boot.test.context.SpringBootTest;

**import** com.sau.springdatajpa.entities.Student;

**import** com.sau.springdatajpa.repos.StudentRepository;

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

@SpringBootTest

**class** SpringdatajpaApplicationTests {

@Autowired //inject StudentRepository

**private** StudentRepository repo;

@Test

**void** testSaveStudent() {

Student student = **new** Student();

student.setId(1L);

student.setName("savita");

student.setTestScore(100);

repo.save(student); //saved to database // save is crud method that we are able to access here

Student savedStudent = repo.findById(1L).get(); // retriving from database

*assertNotNull*(savedStudent); // cheking

}

}

**Show SQL**

* **In application.properties**

spring.jpa.show-sql=true

it means the orm tool should show the sql that they are generating

**REST Introduction**

* We can able to watch video lecture because
* web browser acting as a restful client
* this website is acting as a restful provider
* REST stands for representational state transfer
* REST is a bunch of principle or architectural guidelines
* And http allows us to easily implement REST

Every web app allow the end user to perform below 4 operations

Create, read, update, delete

* Uniform Interface And Easy Access
* Uniform interface and easy access these are 2 REST principles
* http provides a uniform interface to perform the crud operations using its http methods

post🡪to create a resource

get🡪to read a resource

put🡪 to update a resource

delete🡪 to delete a resource

these methods are also known as http verbs

* http provides an easy access and unique way to identify resources through its URI’s and URL’s

e.g if we are working on employee management software we can indentify their resources using “ /”

URI’s:/employees, URI’s:/department

* URI’s and URL’s are also known as nouns because they uniquely identify the source
* **Multiple Formats**
* This is third REST principle
* Our application or restful provider should support multiple formats, and http allows us to do it very easily
* So We can support xml, json, plaintext, or any other format and the client can tell which format they want

***REST web services using spring boot***

* Create a controller class and mark it with @RestController
* controller class is the restful end place and you can define methods in the controller class as many you want
* and every method will be marked with the appropriate annotation

@GetMapping, @PostMapping, @PutMapping, @DeleteMapping

* these annotations bind your java method to the incoming http method from the client

1. @RequestMapping(“/products”) to map it to the path.

**Create the DB table**

1. Launch my sql workbench
2. Drag and drop the file to the my sql workbench or create a table of name product
3. In my sql

Select the command which you want to execute and press the execute button, then only that command will gets executed

use mydb;

create table product(

id int auto\_increment PRIMARY KEY,

name varchar(20),

description varchar(100),

price decimal(8,3)

);

select \* from product;

***Create a Project***

* Create spring boot starter project
* Select dependencies

Mysql driver

Spring data jpa

Spring web

Create the model and repository