what is spring framework ?

--->

1. Open source : Spring framework is an open-source framework for building enterprise java applications.

2. features : Spring Aims to simplify the complex enterprise Java application development process

By Offering the framework that includes technologies such as AOP, DI, POJO, and so on.

3. Lightweight : Even all this technologies spring lightweight framework, that can be used for to create scalable, rebust,

secure enterprise application.

spring framework is collection of sub framework such as spring web flow, spring mvc, spring orm and so on.

Core features of spring framework

--->

1. IOC : inversion of control Container, This container is responsible for managing the lifecycle of the to defined java object.

Significantly increasing the configurabilty of the spring base application.

IOC used Dependency Injection or Dependency lookout pattern to provide the object referece during runtime.

2. AOP: Aspect Oriented Programming

AOP aims to provide more modularity to the cross cutting concern which are functions that span across the application

such as logging, caching, transaction management authentication(in security) and so on.

3. DAF : Data Access Framework

- It simplifies the database communication process by providing direct support for popular data access framework,

in java such as jdbc, hibernate, Java persistance API also known as JPA.

- Additionally it offers features such as resource management, exception handling and resource wrapping for all the

supported frameworks further simplifying the development process

4. MVC : Spring MVC framework

- The spring MVC enables developers to create applications using the popular MVC pattern

- It is a request based framework that allows developers to create customized MVC implementation that,

exactly suits their needs the core components of spring mvc is dispatcher servlet class which handles user request

and then forward to the correct controller. This allows the controller to process the request create the model(entity)

then provide the information for the end user via specified view.

What is spring bean ?

---------->

- Beans a spring refers to an object that is managed by spring framework.

In java application the term, Bean is used in the context of the spring framework.

- The spring framework create this beans manage their life cycle and organise the dependencies with other beans.

It take care of instantiation, configuration and wiring up of objects saving developers from a lot of manual work

- Spring Bean can be configured using XML, java annotation or java code life cycle of spring bean.

Life cycle of Spring Bean ?

--->

First Let's understand the life cycle of an object so the life cycle of an object means when and how it is born, how it is behaves

throught its life and when when and how it dies.

Similarly spring bean life cycle refers to when and how the bean is instantiated, what action is performs until it lives and

when and how it is destroyed .

so the bean life cycle is managed by the spring container.

- when we run the program the first of all the spring container get started after that the container create the instance of a bean

as per the request and then dependencies are injected and finally the bean is destroyed when the spring container closed.

@Configuration Example

---->

Configuration declare class as "full" configuration class,

- class must be non-final and public

@Bean declares bean configuration inside configuration class

- method must be non-final and private (i.e public, protected or package-private)

Syntax -

@Configuration

public class AppConfig{

@Bean

public PaymentService paymentService(AccountRepository accountRepository){ //PaymentService is interface

return new PaymentServiceImpl(accountRepository); //PaymentServiceImpl is class

}

}

we can define multiple bean in configuration class

@Configuration

public class AppConfig{

@Bean

public PaymentService paymentService(){ //PaymentService is interface,payment service bean

return new PaymentServiceImpl(accountRepository); //PaymentServiceImpl is class

}

@Bean

public AccountRepository accountRepository(){

return new JdbcAccountRepository(dataSource());

}

@Bean

public DataSource dataSource(){

return .....;

}

}

Spring Component sample

------->

Spring component contains class level annotation that marks class as a spring component(@Component)

Constructor dependency injection is automatically done using @Autowired by injection the constructor parameter(s)

@Autowired on constructor injection is optional if there is only one constructor

exa;

@Component // this class consider as a bean here

public class PaymentServiceImpl{

private final AccountRepository accountRepository; //injecting another bean

@Autowired // optinal because only one dependency is here

public PaymentServiceImpl(AccountRepository accountRepository){

this.accountRepository = accountRepository;

}

}

spring components

------>

spring provides component stereotype to classify the classes as Spring Components.

--> sub -types are available as a refinement for the standard components.

- @Component are general component annotation indicating that the class should be initiazed, configured and managed by the core container.

- @Repositoty @Service @controller as meta-annotation for @Component that allows to further re-fine components.

- Own stereotype annotation can(and should) be defined support general architecture principle.

Bean naming

------>

@Configuration

public class AppConfig{

Bean

public PaymentService paymentService(AccountRepository accountRepository){ //first bean PaymentService,here it take method name

return new PaymentServiceImpl(accountRepository); //PaymentServiceImpl is class

}

@Bean

public AccountRepository accountRepository(){ // second bean accountRepository, if we don't provide name spring automatically take method name

return new JdbcAccountRepository(dataSource());

}

@Bean("ds") // here we are added bean naming for this

public DataSource dataSource(){ // third bean ds, because we mentioned ds in bean,it is imp because when we access programitically then we use bean name

return .....;

}

}

Bean Injection

--->

Bean Injection also mean dependency..

injection so the spring framework provide 4 way of Bean Injection(Dependency Injection)

So spring can configure dependencies on different injection elements

first we have a constructor injection so the constructor parameter to receive the dependencies during bean costruction so we call

this constructor injection.

field injection : this field defination to receive dependency injected with the reflection also called as field injection.

configuration methods: one or many parameters receiving dependencies through method parameters also called as method injection.

setter injection : setter injection or setter method injection so the java setter method are specialized configuration method with only one parameter

and a defined naming scheme called as Setter injection.

The injection target can be reffered using two different modes types injection an object of matching type or name injects any object by name

constructor injection.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

constructor injection exa :

@Service

public class DefaultPaymentService{

private final AccountRepository accountRepository;

public DefaultPaymentService(AccountRepository accountRepository){

this.accountRepository = accountRepository();

} // here spring will create the bean automatically for AccountRepository

}

@Qualifier

-------->

@Configuration

public class ApplicationConfig{

@Bean

@Qualifier("primary")

public AccountRepository primary(){

return new JdbcAccountRepository(..);

}

@Bean

@Qualifier("secondary")

public AccountRepository secondary(){

return new JdbcAccountRepository(..);

}

}

There is a class Applicationconfig inside two beans of the same type so two beans are of type accountrepository and

we have a primary and secondary so this means that we will create two beans the first one will be primary and second one will be secondary but this two beans are

refered by type which is accountrepository.so there we can @Qualifier second and we want to inject one of these beans we can also on the class level or

for example service level

@Service

public class DefaultPaymentService{

@Autowired

public DefaultPaymentService(@Qualifier("primary")AccountRepository accountrepository)

{

this.accountrepository=accountrepository;

} // here we are telling spring that we want to inject the bean that has the name or the qualifier called primary so this is how we use qualifer annotation to inject

// the bean also if you want to define a bean as a primary we can use @primary annotation.

}

\*\*\*\*\*\*\*\*\*\*

using @primary

------->

@Configuration

public class ApplicationConfig{

@Bean

@Primary

public AccountRepository primary(){

return new JdbcAccountRepository(..);

} // here we are priotizing the which bean should inject first, in thise case we dont use @Qualifier

@Bean

public AccountRepository secondary(){

return new JdbcAccountRepository(..);

}

}

field injection

-------->

field injection allows direct injection into field declaration without Constructor or method delegation.

but here we/you need to note that this type of injection is discouraged because it makes testing of component in isolation more complex.

therefore should only be used in test classes.

example:

@Service

public class DefaultPaymentService{

@Autowired

private AccountRepository accountRepository;

}

method injection

----->

Method injection allows setting one or many dependencies by one method.

Also it allows for initialization work if needed while receiving dependencies.

example:

@Service

public class DefaultPaymentService{

@Autowired

public void configClass(AccountRepository accountRepository,FeeCalculator feecalculator){

...........

}

setter injection

------->

Setter Injection follows Java bean naming convesion to inject dependencies.

example:

@Service

public class DefaultPaymentService{

private AccountRepository accountrepository;

@Autowired

public void setAccountRepository(AccountRepository accountRepository)

{

// ...........

}

Official recommendation (Constructor or setter based DI?)

---------->

use constructors for mandatory dependencies and setter methods or configuration methods for optinal dependencies.

the spring team generally advocates constructor injection.

Bean Scoping

--> what is bean scope ?

The Bean scope in spring framework refers to the life cycle of the spring bean.

And its availability in the contect of the application. so when a bean is instantiated or

looked up it scopes determines its life cycle and which other spring can interact with it.

- sprinng privides multiple Scopes to register and configure beans and scoping has an impact on state mangement of the component.

The default scope model is singleton. means one instance per application context. shared instance will be accessed by other components.

therefore component must be thread safe.

types of bean scope

--->

1. singleton

- The default scope model is singleton. means one instance per application context. shared instance will be accessed by other components.

- this is useful where that do nit hold state or where the same state to be shared by all users or threads.

2. prototype

- new instance is created each time . Bean is requiested from the container this is useful for beans to carry state that i specific to the other user/thread & thus can't be shared.

3. request

- This scope is valid in the context of web over spring application context for single http request.

- New Bean is created for each HTTP request.

4. session

- This scope also valid only on the contect of web where spring application context of HTTP session so it's different from the HTTP request or the request scope is available for the HTTP request.

- The session is HTTP session so this means a new Bean is created for each HTTP session by the container then

5. application

- This scope is also valid only in the context of webware spring context for the life cycle of a servlet context so this bean scoped at application level

6. websocket

- This scope is valid only in the context of webware spring application context for the life cycle of a web socket.

- The bean is scoped at the web socket level

Bean scoping is very important it you want to correctly manage your bean and as i mentioned before by default the bin scope is singleton.

Lets see how we can define the bean scope.

--->

@Configuration

public class MyConfiguration{

@Bean

@Scope("prototype")

public Bean1 bean1(){ // .... }

@Bean

@SessionScope

public Bean2 bean2(){ // ... }

}

sprinng framework special beans

--->

Environment

=========>

bean environment

So the bean environment so the bean environment is an environment abstraction.

spring provides abstraction so abstraction to decouple application code from the environment with the support for bean defination profiles that allows different sets of bean depending

the environment

for example we have local environment, dev environment, cloud environment and so on.

it helps resolving propertie for external sources for example database setting from the configuration file or reading credentials from the CLI arguments and so and so fourth.

injectable

- environment can also be injected into the code if needed

Example:

@Configuration

public class ApplicationConfig{

@Autowired

final Environment environment;

@Bean

public PaymentService paymentService(){

var profile = Profiles.of("cloud');

var isOkay = this.environment.accountProfiles(profile);

this.environment.getProperty("data.driver");

return .......

}

}

Spring Profiles

----->

The profile in spring is a named logical groupinng that may be activated programmatically or set as active through configuration.

so this feature is perticularly useful when you have beans that should be active or registered and used in certain environment and condition.

for instance you may have different for development, testing and production envionment.

and you want to make that certain beans are only used in one of this environment.

Here we used three way ---------->

// Spring component

@Service

@Profile("cloud")

public class DefaultPaymentService implements PaymentService{ }

// Configuration class

@Configuration

@Profile("cloud")

public class ApplicationCOnfig{ }

@Configuration

public class ApplicationConfig{

// bean configuration method

@Bean

@Profile("cloud")

public PaymentService paymentService() { . .............. } // this bean is perticullay special for cloud profile

}

profile activation - programmatically

--->

public static void main(String[] args){

AnnotationConfigApplicationContext applicationContext;

applicationContext = new AnnotationConfigApplicationContext();

applicationContect.getEnvironement().setActiveProfiles("cloud");

applicationContext.scan("com.sb.sample");

applicationContext.refresh(); // spring will refresh its own context.

PaymentService paymentService = applicationContext.getBean(PaymentService.class);

}

Profile activation - properties file

------->

application.yaml

spring:

profiles:

active: cloud

application.properties

spring.profiles.active=cloud

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

@Value annotation

--------->

This annotation is on field level, constructor level setter method level

this annotation is commenly used for taking values for variable in a class

whether primitive, string, complex type

this value come from property file or system property file

@Configuration

@PropertySource("classpath:database.properties") // this present in resource folder

public class ApplicationConfig{

@Value("${jdbc.url}")

private String url;

.... likewise username and password

@Bean

public DataSource dataSource(){

return ....... .;

}

@Component

public class FeeCalculator{

private String defaultLocal;

@Value("#{systemProperties['user.region']}")

public void setDefaultLocal(String defaultLocal){

this.defaultLocal = defaultLocal;

}

}

spring best practices

-------split configuration-------

avoid large configuration classes and make sure to allow split configuration classes based on architecture

classes can be import

@Configuration

public class ServiceConfig{

@Bean

public PaymentService paymentService(){ return new ........}

}

@Configuration

public class RepositoryConfig{

@Bean

public AccountRepository accountRepository(){ return new .... }

}

@Configuration

@Import({ServiceConfig.class, RepositoryConfig.class})

public class AppConfig{

@Bean

public DataSource datasource(){ .............. }

}

Spring Initializr

----------->

It provide simple webUI to configure the project to generate an endpoint that you can use via plane HTTP all you need to go start.spring.io

This service allows you to customize project to generate the build system using for example Gradle, maven

recommended to select jar file with spring boot.

So This project is generated by Spring Initializr

=================================================

Spring Boot

------->

what is spring boot?

1. Spring Based

spring boot is an approach to develop spring based applications with very less or no configuration

It leverage existing spring boot projects as well as third part projects to develop production ready application.

2. starters

It provides a set of starters pom or gradle file which can use to add required dependencies and facilitate Autoconfiguration

3.AUTO CONFIGURATION

spring boot will also come with lots of auto configuration so depending on the libraries on its class path spring boot able to automatically configure the required classes.

WHY Spring Boot ?

--------->

1. standalone applications

2. Embedded server(tomcat.jetty) No need to deploy any war files.

3. opiniated starters

4. auto configuration

5. production ready feature ( such as metric health check extranalized configuration)

6. No xml configuration ( it only need configuration whenever it's required)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

maven

java

3.1.0 (comes with some deprecation so here use

3.0.7 -- use this version

group - refers to company **com.facebook**

artifact - project or sub-company like **example**

Name - artifact

project file type : jar war(java archie, web archie)

select web dependency

**then unzip project file**

select project -> select pom.xml and open -> open as project -> trust project -> project opened

file-> project Structure -> **select sdk(java version)**

we don't need to provide version becuase springboot manage starters

**run application**

# Start The session here

change the banner -- > text google banner for spring

package com.app;  
public class MyFirstClass{  
  
 public String sayHello(){  
 return "Hello from the MyFirstClass";  
 }  
}

package com.app;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
@SpringBootApplication  
public class SpringbootAppApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(SpringbootAppApplication.class, args);  
 *// tight coupling manually created object* MyFirstClass obj = new MyFirstClass();  
 System.*out*.println(obj.sayHello()); *// Hello from the MyFirstClass* }  
}

package com.app;  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.context.annotation.Bean;  
  
@SpringBootApplication  
public class SpringbootAppApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(SpringbootAppApplication.class, args);  
 }  
  
 *// but this approach we can't follow , we follow like this* @Bean  
 public MyFirstClass myFirstClass(){  
 return new MyFirstClass();  
 }  
}

**To get the bean from the spring container**

* first way of creating th bean and getting the bean
* package com.app;  
    
  import org.springframework.boot.SpringApplication;  
  import org.springframework.boot.autoconfigure.SpringBootApplication;  
  import org.springframework.context.annotation.Bean;  
    
  @SpringBootApplication  
  public class SpringbootAppApplication {  
    
   public static void main(String[] args) {  
   var ctx = SpringApplication.*run*(SpringbootAppApplication.class, args);  
   MyFirstClass bean = ctx.getBean(MyFirstClass.class); *// here not created manual instance* System.*out*.println(bean.sayHello());  
   }  
    
   *// but this approach we can't follow , we follow like this* @Bean  
   public MyFirstClass myFirstClass(){  
   return new MyFirstClass();  
   }  
  }

second way of creating the bean, we tell the bean to spring container

*we use* ***@Component*** *here*

package com.app;  
  
import org.springframework.stereotype.Component;  
  
@Component *// or @Service, @Repository*public class MyFirstClass{  
  
 public String sayHello(){  
 return "Hello from the MyFirstClass";  
 }  
}

package com.app;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.context.annotation.Bean;  
  
@SpringBootApplication  
public class SpringbootAppApplication {  
  
 public static void main(String[] args) {  
 var ctx = SpringApplication.*run*(SpringbootAppApplication.class, args);  
 MyFirstClass bean = ctx.getBean(MyFirstClass.class);  
 System.*out*.println(bean.sayHello());  
 }  
}

spliting the configuration

--------------->

package com.app;  
  
public class MyFirstClass{  
  
 public String sayHello(){  
 return "Hello from the MyFirstClass";  
 }  
}

----

package com.app;  
  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
  
@Configuration  
public class ApplicationConfig {  
  
 @Bean  
 public MyFirstClass myFirstClass() {  
 return new MyFirstClass();  
 }  
}

// ctr+Alt+o --- remove unwanted import

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**bean naming**

package com.app;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
  
@Configuration  
public class ApplicationConfig {  
  
@Bean("myBean") *// <---- Bean naming performed*  
 public MyFirstClass myFirstClass() {  
 return new MyFirstClass();  
 }  
}

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

package com.app;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.context.annotation.Bean;  
  
@SpringBootApplication  
public class SpringbootAppApplication {  
  
 public static void main(String[] args) {  
 var ctx = SpringApplication.*run*(SpringbootAppApplication.class, args);  
 MyFirstClass bean = ctx.getBean("myBean",MyFirstClass.class);  
 *// here not created manual instance,bean naming myBean* System.*out*.println(bean.sayHello());  
  
 }  
}

=====================

package com.app;  
  
public class MyFirstClass{  
  
 private String myVar;  
  
 public MyFirstClass(String myVar) {  
 this.myVar = myVar;  
 }  
   
 public String sayHello(){  
 return "Hello from the MyFirstClass ====> myVar = "+myVar;  
 }  
}

=====================

package com.app;  
  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
  
@Configuration  
public class ApplicationConfig {  
  
 @Bean("myBean") *// <---- Bean naming performed* public MyFirstClass myFirstClass() {  
 return new MyFirstClass("first bean");  
 }  
}

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

main Springbootapplication no change

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

Understand dependency injection

-------------->

package com.app;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;  
  
@Service  
public class MyFirstService {  
 *//inject the bean* private MyFirstClass myFirstClass;  
 @Autowired *// it is optinal on constructor* public MyFirstService(MyFirstClass myFirstClass) { *//did constructor based dependency, autowire is optional becuase injecting only one bean* this.myFirstClass = myFirstClass;  
 }  
 public String tellStory() {  
 return "The Dependency injection : "+myFirstClass.sayHello();  
 }  
}

**O/p :** The Dependency injection : Hello from the MyFirstClass ====> myVar = Second bean

------

now we create two bean for Myfirstclass in config class

we can resolve this using @primary or @Qualifier

----------

package com.app;  
  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
  
@Configuration  
public class ApplicationConfig {  
  
 *@Bean* public MyFirstClass myFirstClass() {  
 return new MyFirstClass("first bean");  
 }  
  
 @Bean  
 public MyFirstClass mySecondBean() {  
 return new MyFirstClass("Second bean");  
 }  
}

-------

package com.app;  
  
public class MyFirstClass{  
  
 private String myVar;  
  
 public MyFirstClass(String myVar) {  
 this.myVar = myVar;  
 }  
  
 public String sayHello(){  
 return "Hello from the MyFirstClass ====> myVar = "+myVar;  
 }  
}

-----

========

package com.app;  
  
import org.springframework.beans.factory.annotation.Qualifier;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
  
@Configuration  
public class ApplicationConfig {  
   
 @Bean  
 @Qualifier("bean1")  
 *// here we are telling spring spring bean should inject bean1 or bean2 using @Qualifier* public MyFirstClass myFirstClass() {  
 return new MyFirstClass("first bean");  
 }  
  
 @Qualifier("bean2")  
 @Bean  
 public MyFirstClass mySecondBean() {  
 return new MyFirstClass("Second bean");  
 }  
}

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

package com.app;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.beans.factory.annotation.Qualifier;  
import org.springframework.stereotype.Service;  
  
@Service  
public class MyFirstService {  
 *//inject the bean* private MyFirstClass myFirstClass;  
  
 @Autowired *// it is optinal on constructor* public MyFirstService(@Qualifier("bean2") MyFirstClass myFirstClass) {  
 *//did constructor based dependency, autowire is optional becuase injecting only one bean* this.myFirstClass = myFirstClass;  
 }  
 public String tellStory() {  
 return "The Dependency injection : "+myFirstClass.sayHello();  
 }  
}

another option we can use here @Primary

==========> just apply on @Primary on that method which one give the highest priority

package com.app;  
  
import org.springframework.beans.factory.annotation.Qualifier;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.context.annotation.Primary;  
  
@Configuration  
public class ApplicationConfig {  
  
 @Bean  
 public MyFirstClass myFirstClass() {  
 return new MyFirstClass("first bean");  
 }  
  
 @Bean  
 public MyFirstClass mySecondBean() {  
 return new MyFirstClass("Second bean");  
 }  
  
 @Bean  
 @Primary  
 public MyFirstClass myThirdClass() {  
 return new MyFirstClass("third bean");  
 }  
  
}

-

No code changes in myFirstService no use @Qualifier

---

**dependency injection through field injection**

----------->

package com.app;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.beans.factory.annotation.Qualifier;  
import org.springframework.stereotype.Service;  
  
@Service  
public class MyFirstService {  
  
 @Autowired  
 private MyFirstClass myFirstClass;  
   
 public String tellStory() {  
 return "The Dependency injection : "+myFirstClass.sayHello();  
 }  
}

Now use , without declare @Qualifier on the config class direct add in service layer @Qualifer

---------->

package com.app;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.beans.factory.annotation.Qualifier;  
import org.springframework.stereotype.Service;  
  
@Service  
public class MyFirstService {  
  
 @Autowired  
 @Qualifier("myThirdClass")  
 private MyFirstClass myFirstClass;  
  
 public String tellStory() {  
 return "The Dependency injection : "+myFirstClass.sayHello();  
 }  
}

**Method injection**

package com.app;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.beans.factory.annotation.Qualifier;  
import org.springframework.stereotype.Service;  
  
@Service  
public class MyFirstService {  
  
  
 private MyFirstClass myFirstClass;  
  
 @Autowired  
 public void injectDependency(@Qualifier("mySecondBean")MyFirstClass myFirstClass){  
 this.myFirstClass = myFirstClass;  
 }  
  
 public String tellStory() {  
 return "The Dependency injection : "+myFirstClass.sayHello();  
 }  
}

**Setter injection**

package com.app;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.beans.factory.annotation.Qualifier;  
import org.springframework.stereotype.Service;  
  
@Service  
public class MyFirstService {  
  
  
 private MyFirstClass myFirstClass;  
  
 @Autowired  
 public void setMyFirstClass(MyFirstClass myFirstClass) {  
 *// here we use @Qualifier & give bean name* this.myFirstClass = myFirstClass;  
 *// but in spring boot 3 it will take first bean by default* }  
  
 public String tellStory() {  
 return "The Dependency injection : "+myFirstClass.sayHello();  
 }  
}

spring provide special bean environment bean

use environment bean here

-------->

package com.app;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.beans.factory.annotation.Qualifier;  
import org.springframework.core.env.Environment;  
import org.springframework.stereotype.Service;  
  
@Service  
public class MyFirstService {  
  
  
 private MyFirstClass myFirstClass;  
 private Environment environment;  
  
 @Autowired  
 public void setMyFirstClass(MyFirstClass myFirstClass) {  
 this.myFirstClass = myFirstClass; }  
  
  
 @Autowired  
 public void setEnvironment(Environment environment) {  
 this.environment = environment; }  
  
 public String getJavaVersion() {  
 return environment.getProperty("java.version"); }  
  
 public String getOsName() { return environment.getProperty("os.version"); }  
  
 public String tellStory() { return "The Dependency injection : "+myFirstClass.sayHello(); }  
  
}

----------

package com.app;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.context.annotation.Bean;  
  
@SpringBootApplication  
public class SpringbootAppApplication {  
  
 public static void main(String[] args) {  
 var ctx = SpringApplication.*run*(SpringbootAppApplication.class, args);  
 MyFirstService myFirstClass = ctx.getBean(MyFirstService.class);  
 System.*out*.println(myFirstClass.tellStory());  
 System.*out*.println(myFirstClass.getJavaVersion());  
 System.*out*.println(myFirstClass.getOsName());  
  
  
 }  
}

--- Enviornment is also use to get the property ...application.property

application.properties

application.version=v1  
my.custom.property = Hello Subhash Birajdar

-------

package com.app;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.beans.factory.annotation.Qualifier;  
import org.springframework.core.env.Environment;  
import org.springframework.stereotype.Service;  
  
@Service  
public class MyFirstService {  
  
  
 private MyFirstClass myFirstClass;  
 private Environment environment;  
  
 @Autowired  
 public void setMyFirstClass(MyFirstClass myFirstClass) {  
 this.myFirstClass = myFirstClass; }  
  
  
 @Autowired  
 public void setEnvironment(Environment environment) {  
 this.environment = environment; }  
  
 public String getJavaVersion() {  
 return environment.getProperty("java.version"); }  
  
 public String getOsName() { return environment.getProperty("os.version"); }  
  
 public String tellStory() { return "The Dependency injection : "+myFirstClass.sayHello(); }  
  
 public String getreadproperty() { return environment.getProperty("my.custom.property"); }  
  
}

--------

package com.app;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.context.annotation.Bean;  
  
@SpringBootApplication  
public class SpringbootAppApplication {  
  
 public static void main(String[] args) {  
 var ctx = SpringApplication.*run*(SpringbootAppApplication.class, args);  
 MyFirstService myFirstClass = ctx.getBean(MyFirstService.class);  
 System.*out*.println(myFirstClass.tellStory());  
 System.*out*.println(myFirstClass.getJavaVersion());  
 System.*out*.println(myFirstClass.getOsName());  
 System.*out*.println(myFirstClass.getreadproperty());  
  
  
 }  
}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*

suppose you have some custom property file then

------->

custom.properties

------>

my.prop = Hey SB

**myServiceclass**

package com.app;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.beans.factory.annotation.Qualifier;  
import org.springframework.beans.factory.annotation.Value;  
import org.springframework.stereotype.Service;  
  
@Service  
public class MyFirstService {  
  
 private final MyFirstClass myFirstClass;  
 @Value("Hello Java")  
 private String customProperty;  
 @Value("${my.prop}")  
 private String custompropertyfromAnotherField;  
 @Value("123")  
 private Integer customPropertyInt;  
  
 public String getCustompropertyfromAnotherField() {  
 return custompropertyfromAnotherField;  
 }  
  
 public String getCustomProperty() {  
 return customProperty;  
 }  
  
 public Integer getCustomPropertyInt() {  
 return customPropertyInt;  
 }  
  
 public MyFirstService(@Qualifier("mySecondBean")MyFirstClass myFirstClass) {  
 this.myFirstClass = myFirstClass;  
 }  
  
 public String tellStory() {  
 return "The Dependency injection : "+myFirstClass.sayHello();  
 }  
}

---

package com.app;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.context.annotation.Bean;  
  
@SpringBootApplication  
public class SpringbootAppApplication {  
  
 public static void main(String[] args) {  
 var ctx = SpringApplication.*run*(SpringbootAppApplication.class, args);  
 MyFirstService myFirstClass = ctx.getBean(MyFirstService.class);  
 System.*out*.println(myFirstClass.tellStory());  
 System.*out*.println(myFirstClass.getJavaVersion());  
 System.*out*.println(myFirstClass.getOsName());  
 System.*out*.println(myFirstClass.getreadproperty());  
  
  
 }  
}

---------

Caused by: java.lang.IllegalArgumentException: Could not resolve placeholder 'my.prop' in value "${my.prop}"

because it is not in application.properties , spring scan only application.properties

so we have to tell spring through @PropertySource annotation on class level to resolve this problem

--------=====>

package com.app;  
  
import org.springframework.beans.factory.annotation.Qualifier;  
import org.springframework.beans.factory.annotation.Value;  
import org.springframework.context.annotation.PropertySource;  
import org.springframework.stereotype.Service;  
  
@Service  
@PropertySource("classpath:custom.properties")  
*// this is added, so if you want to more custom properties then you can add @PropertySources({ @PropertySource..})*public class MyFirstService {  
  
 private final MyFirstClass myFirstClass;  
 @Value("Hello Java")  
 private String customProperty;  
 @Value("${my.prop}")  
 private String custompropertyfromAnotherField;  
 @Value("123")  
 private Integer customPropertyInt;  
  
 public String getCustompropertyfromAnotherField() {  
 return custompropertyfromAnotherField;  
 }  
  
 public String getCustomProperty() {  
 return customProperty;  
 }  
  
 public Integer getCustomPropertyInt() {  
 return customPropertyInt;  
 }  
  
 public MyFirstService(@Qualifier("mySecondBean")MyFirstClass myFirstClass) {  
 this.myFirstClass = myFirstClass;  
 }  
  
 public String tellStory() {  
 return "The Dependency injection : "+myFirstClass.sayHello();  
 }  
}

------

**inject multiple properties** , access multiple custom proprties file using @PropertySources

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

package com.app;  
  
import org.springframework.beans.factory.annotation.Qualifier;  
import org.springframework.beans.factory.annotation.Value;  
import org.springframework.context.annotation.PropertySource;  
import org.springframework.context.annotation.PropertySources;  
import org.springframework.stereotype.Service;  
  
@Service  
@PropertySources({  
 @PropertySource("classpath:custom.properties"),  
 @PropertySource("classpath:custom-1.properties")  
})  
public class MyFirstService {  
  
 private final MyFirstClass myFirstClass;  
 @Value("Hello Java")  
 private String customProperty;  
 @Value("${my.prop}")  
 private String custompropertyfromAnotherField;  
 @Value("123")  
 private Integer customPropertyInt;  
  
 @Value("${my.prop1}")  
 private String custompropertyfromAnotherField2;  
  
 public String getCustomProperty() {  
 return customProperty;  
 }  
  
 public String getCustompropertyfromAnotherField() {  
 return custompropertyfromAnotherField;  
 }  
  
 public Integer getCustomPropertyInt() {  
 return customPropertyInt;  
 }  
  
  
 public String getCustompropertyfromAnotherField2() {  
 return custompropertyfromAnotherField2;  
 }  
  
 public MyFirstService(@Qualifier("mySecondBean")MyFirstClass myFirstClass) {  
 this.myFirstClass = myFirstClass;  
 }  
  
 public String tellStory() {  
 return "The Dependency injection : "+myFirstClass.sayHello();  
 }  
}

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

package com.app;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.context.annotation.Bean;  
  
@SpringBootApplication  
public class SpringbootAppApplication {  
  
 public static void main(String[] args) {  
 var ctx = SpringApplication.*run*(SpringbootAppApplication.class, args);  
 MyFirstService myFirstClass = ctx.getBean(MyFirstService.class);  
 System.*out*.println(myFirstClass.tellStory());  
 System.*out*.println(myFirstClass.getCustomProperty());  
 System.*out*.println(myFirstClass.getCustompropertyfromAnotherField());  
 System.*out*.println(myFirstClass.getCustomPropertyInt());  
 System.*out*.println(myFirstClass.getCustompropertyfromAnotherField2());  
  
 }  
}

spring profiles – dev,test,staging,prod

Different beans might be registered in the development environment vs production environment

so each profile corresponds to set of configuration that define how the application should run in specific environment

so the envoronment might be development, test, staging, uat, production,

The beans that are part of a profile can be registered in the spring application context only when profile is active.

this capability can be perticularly beneficial in server schenarios

for example, in environment specific configuration you might have certain that should only active in the development environment

and different one that should be active in a production environment

for example in a development environment you might want to use that bean that clears and recreates your database with this data every time you start your application

in production environement you would certainly not want this bean to be active

also profile can be for component switching you can use profile switch out entire component or services

for exa, you may have a quick in memory database for development while in production you would fullblown database server

also profiles can be used for toggling features

prodiles can be used to enable or disable features if you are developing a new feature that's not quite ready to be deployed in production you can put this in its own profile until it's ready

---

let's we how we can work with profile

while runnning application spring override application.properties to application-dev.properties if you change profile

application-dev.properties

---------

my.custom.property = Hello Subhash Birajdar for development  
my.custom.property.int=321

**application.properties**

spring.application.name=springbootApp  
application.version=v1  
my.custom.property = Hello Subhash Birajdar  
my.prop1 = Hey SB1, from custom-1.properties in development  
*#spring.active.profile=dev -- this is wrong  
#here we are telling the order of the profile*spring.profiles.active=dev,test,custom  
*#spring.profiles.active=test,dev,custom  
# here order is for first is dev*

**profile can set programatically**

package com.app;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.context.annotation.Bean;  
  
import java.util.Collections;  
  
@SpringBootApplication  
public class SpringbootAppApplication {  
  
 public static void main(String[] args) {  
 var app = new SpringApplication(SpringbootAppApplication.class);  
 app.setDefaultProperties(Collections.*singletonMap*("spring.profiles.active","dev"));  
 var ctx=app.run(args);  
 MyFirstService bean = ctx.getBean(MyFirstService.class);  
 System.*out*.println(bean.tellStory());  
 System.*out*.println(bean.getCustomProperty());  
 System.*out*.println(bean.getCustompropertyfromAnotherField());  
 System.*out*.println(bean.getCustomPropertyInt());  
 System.*out*.println(bean.getCustompropertyfromAnotherField2());  
 }  
}

**beans which are avaialble for specific profile**

package com.app;  
  
import org.springframework.beans.factory.annotation.Qualifier;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.context.annotation.Primary;  
import org.springframework.context.annotation.Profile;  
  
@Configuration  
public class ApplicationConfig {  
  
 @Bean  
 @Profile("dev") *// this bean for peofile dev* public MyFirstClass myFirstBean() {  
 return new MyFirstClass("first bean");  
 }  
  
 @Bean  
 @Profile("test") *// this bean for peofile test* public MyFirstClass mySecondBean() {  
 return new MyFirstClass("Second bean");  
 }  
  
 @Bean  
 public MyFirstClass myThirdClass() {  
 return new MyFirstClass("third bean");  
 }  
}

--------

package com.app;  
  
import org.springframework.beans.factory.annotation.Qualifier;  
import org.springframework.beans.factory.annotation.Value;  
import org.springframework.context.annotation.PropertySource;  
import org.springframework.context.annotation.PropertySources;  
import org.springframework.stereotype.Service;  
  
@Service  
@PropertySources({  
 @PropertySource("classpath:custom.properties"),  
 @PropertySource("classpath:custom-1.properties")  
})  
public class MyFirstService {  
  
 private final MyFirstClass myFirstClass;  
 @Value("Hello Java")  
 private String customProperty;  
 @Value("${my.prop}")  
 private String custompropertyfromAnotherField;  
 @Value("123")  
 private Integer customPropertyInt;  
  
 @Value("${my.prop1}")  
 private String custompropertyfromAnotherField2;  
  
 public String getCustomProperty() {  
 return customProperty;  
 }  
  
 public String getCustompropertyfromAnotherField() {  
 return custompropertyfromAnotherField;  
 }  
  
 public Integer getCustomPropertyInt() {  
 return customPropertyInt;  
 }  
  
  
 public String getCustompropertyfromAnotherField2() {  
 return custompropertyfromAnotherField2;  
 }  
  
 public MyFirstService(@Qualifier("myFirstBean")MyFirstClass myFirstClass) {  
 this.myFirstClass = myFirstClass;  
 }  
  
 public String tellStory() {  
 return "The Dependency injection : "+myFirstClass.sayHello();  
 }  
}

\*\*\*\*\*\*\*\*\*\*\*\*\*

**change all the beans for dev profile, apply on the class level**

package com.app;  
import org.springframework.beans.factory.annotation.Qualifier;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.context.annotation.Primary;  
import org.springframework.context.annotation.Profile;  
  
@Configuration  
@Profile("dev")  
public class ApplicationConfig {  
  
 @Bean  
 public MyFirstClass myFirstBean() {  
 return new MyFirstClass("first bean");  
 }  
  
 @Bean  
 public MyFirstClass mySecondBean() {  
 return new MyFirstClass("Second bean");  
 }  
  
 @Bean  
 public MyFirstClass myThirdClass() {  
 return new MyFirstClass("third bean");  
 }  
}

---------- public static void main(String[] args) {  
 var app = new SpringApplication(SpringbootAppApplication.class);  
 app.setDefaultProperties(Collections.*singletonMap*("spring.profiles.active","dev"));

//but if you add profile for test then we get the error for configuration

}

**RESTFUL WITH SPRING IN ACTION**

SPRING REST - OVERVIEW

**1. REST - representational state transfer**

* It define software architecture style that define set of constraint to be used for creating web services and restful web services.
* This web services are often called as restful apis or restful application programming interfaces if they adhere this constraints

**2. Rest was first defined by Roy Fielding in his 2000 Doctoral dissertation.**

* The main idea behind is to tree network resources as object that can be accessed using standard HTTP methods,
* such as get, Post, Put, Delete etc
* Here are some of the fundamental elements of the rest architecture.

**3- Client - server architecture**

**4- stateless**

* each http request that happen from a client to a server should contain all the necessary information to understand and to the request in other word server should not store any data between request which keeps each request isolated and independent.

**5. Cachable**

* restful architecture allows client to cach responses must be implicitly or explictly define themselves as cachable or not prevent clients from reusing outdated or an appropriate data in response to further request.

**6. layed System**

* The architecture allows for layers within the system architecture a client can't ordinarily tell whether it is connected directly to an end server or to an intermediary along a way which can include **load balancing security measures**.

**7. code on demand(optional)**

* It is a optional constraint it allows the server to extend functionality of a client by transferring executable code.

**8. uniform interface**

* This is a fundamental to the design of any restful system it simplifies and decouples the architecture which enables part to evolve independently

**The fourth guiding principles of the uniform interface are**

1. Identification of resources,
2. Manipulation of resources through these representations
3. Self descriptive mesages
4. Hypermedia as the engine of application state(HATEOS)

**( Restful api become very popular in modern web application development due to simplicity, scalability, statelessness, & compability with the web**.**)**

They are often used to create interactive application mobile application and web services

however they are not suitabable for all type of applications & other architectures or protocols such as graphql, or grpc might be used to depending on specific needs of the application to resume

**The rest is the web architectural principle**

1. Unique Identification of resource(URI)
2. Different resources representation
3. Hypermedia / linking of resources
4. Stateless communication

standard method- get, post, put, delete and responses such as 200- ok , 404- NotFound

**We need to follow some standard for developing rest**

**1. The resource should always be plural nouns in the API endpoint and if one instance resource should be retrived, pass the id in the URl :**

for example

GET /accounts

GET /accounts/1

DELETE /accounts/2

**2. In case if nested resources(resources under a resource),the resource should be accessible as follows:**

**GET /accounts/1/payments/56**

**3. use the HTTP methods to specify what to do with this resource. With method get post put patch delete you can provide crud functionality**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Resource** | **GET** | **POST** | **PUT** | **DELETE** |
| /accounts | get all accounts | create new account | bulk update all account | delete all accounts |
| /account/1 | get account with Id 1 | **Error** | update account with ID 1 | delete account with ID 1 |
| /account/1/payment | get all payment for for account account with ID 1 | create new payment for account with ID 1 | bulk update all payment for account with ID 1 | delete all payment for account with ID 1 |

**When designing restful api we need to use HTTP methods also known as verbs**

* **GET** : This method is used to retrive data from server, it is read only operation. meaning it doesn't affect the state of resource
* **POST** : used to send data to the server to create a new resource, the data is included in the body of the request.
* **PUT** : used to update an existing resource or create it if it does not exist the update or new data is included in the body of the request.
* **DELETE** : used to delete a resource specified by a URL.
* **PATCH** : used to partial modification to a resource as opposed to the put method

put is used for full updates like post and put this data for the update is included in the body of the request.

* **OPTIONS** : used to return HTTP methods that server support for specified URL. it can be used to check the functionality of a web server.
* **HEAD** : This method is similar to get but only returns the header of the response.not actual data for example the body

This is useful when you want to check if a resource exists before trying ti download it or check if it has being modified.

**STATUS CODE**

**response status are HTTP status code**

status code nothing but 3 digit code return by the server like http response

It is indicating that the outcome of the request made by the client

* 100 (information)
* **2xx - success category**

1. 2. 200 - ok

this is standard response for the successful http request.

then you a get request for exa the server successfully processes the request and provided the request resource in the response it will return a 200.

1. 201-created

indicate that the request is successful and a new resource was created as a result this is typically the response sent after a post request

or something put if you are creating resource with that method.where new resource is created and the server based on the data send to the server.

1. 203 - No content

means **server successfully processed the requestt and there is no additional content to sent in the response payload body**

this is often the response to delete request put or post request when there is no perticular information to send back HTTP response but header is useful in other words its a way to for the server to say i did what you asked where and we are done so you don't need to go anywhere else this response may also be used when the server doesn't want to return any information like as a result of a delete request confirming the deletion is successful.

* **3xx - redirectoion category**

redirection category also represented by the three xx response codes for this category we will mainly focus on one which is the important and most used in restful apis which is the 304 not modified so the HTTP status code 304 not modified is a special type of response

**304 - NOT MODIFIED**

response that is used for **caching purposes** when a client sends a request to the server it can include a header such as if modified sense or if non match this header are used to ask the server to validate if the client's cached version of the resource is still up to date if

modified since still up to date if modified since use a timestamp to determine if the resource has been updated since the last time the client requested also we have the if non match it also works with tags or entity tags which is a token associated with a specified version the resource so if the resource has not been modified since the given date time in the case of for example if modified as or entity tag still matches so in this case if non match so the server will respond witha 304 not modified status and not send the body for the response because the client cache is still valid by utilizing a 304 response you can save bandwidth and make your web application faster by not unnecessary resending in resources that the client has already cached so this leverages one of the main principle of HTTP is being stateless but still allowing for optimizing through caching

* **4xx - client error category**

here we will check the most used response

1. **400 - bad request**

- this status code mean server was unable to understand the request due invalid syntax.

this could be mlformed request or a request with invalid arguments the client should modify the request before repeating it.

1. **401 - unauthorized**

means the reqest requires an user authentication, if the user already included authentication credentials then the forward 401

indicate that the authorization has been refused for these credentials. This could be due to a wrong username or password or user does not have the neccessary privileges. This status is similar to 403 forbidden but specifically for cases when authentication is expect but has failed or not yet bean provided.

1. **403 - forbidden**

means the **client does not have the necessary permission** for the requested resource in constract to the 401 unauthorized response autnenticating will make no difference this status code also can be used when the server not want to reveal exactly why the request has been refused or when no other responses is applicable so for all this status code are part of 4xx class of HTTP status code which indicates that server was likely an error in the request send by the clinet that prevented the server from processing it as a best practice the client should after its request or authenticate and authenticate properly based on these responses

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

@RestController

@RestController

public class PaymentRestController{

@PostMapping(value ="/payment") // this is a verb, http method

public ResponseEntity<PaymentInformation> initiatePayment(

@RequestBody PaymentRequest paymentRequest)

{

//business code

URI resultLocation = UrlComponentsBuilder

.fromPath("/payments/{id}")

.buildAndExpand(configuration.getId())

.toUri();

return ResponseEntity.created(resultLocaltion).body(configuration);

// return responsebody and status code

}

}

* To specify the response status of a controller method, annotate method with @ResponseStatus
* Spring only uses @ResponseStatus, when the marked method complete successfully(without throwing an exception)

**5XX - SERVER ERROR CATEGORY**

**500 INTERNAL SERVER ERROR**

This is generally generic error message given when an unexpected condition was encountered by the server and no more specific mesage is suitable.

In other words it means that something has gone wrong on the server itself.

it doesn't provide a clear reason because the error could be for a variety of reasons such as erver misconfiguration or exception on the server side code or server resource issue like memory duplication

1. **503 server unavailable**

which is service and available this status code indicates that the server is currently unavailable to handle the request because its temporarily overloading or down for maintenance

generally this is a temporary state.it may retry after header indicating how long to wait before making a request so both of these status codes are part of 5XX class of HTTP status codes which indicates situation where the server is aware that it has encountered an error or is otherwise incapable of performing the request these types of error are often outside of client's control and may require attention from the servers operators

// check application is uppend running or not after clean up classes and properties.file

Now first create a controller class

what is the verb and HTTP method for this endpoint

@RestController  
public class FirstController {  
  
 *// create end point* @GetMapping("/") *// this is nothing but endpoint* public String sayHello(){  
 return "Hello from my first controller"; *// httpStatus return 200 ok* } *// what is the context path '/' ?  
  
 // http method contain bunch of variable it is present in the @ResponseStatus* @GetMapping("/hello")  
 @ResponseStatus(HttpStatus.*ACCEPTED*)  
 *// this is nothing but endpoint* public String sayHello1(){ *//localhost:8080/* return "Hello1 from my first controller";  
 } *// httpStatus return 202 ACCEPTED*

// alt+ shift + enter === unused import

*// PostMapping means clinet to send some data from client to server*@PostMapping("/post")  
public String post(String message){ *// here we need to provide for example* return "Request accepted and message is :"+ message;  
}

* in crome browser localhost:8080/post --- we will get 405 method not allowed
* post request on the other hand are designed to send data to server to create or update resource so post data is sent in the body of the HTTP request not in the URL
* this is the part of the HTTP specification so now since we are not able to use the browser address bar to send post requests.
* let's see what alternative for using or as a **Postman is used for API development and testing and it simplifies each step for the API cycle** including design mocking testing documentation monitoring.

**param --- @Requestparam**

**Authorization** ---- secure backend or secure API here we can specify type of the security

**Headers** ---- by default we have few headers which hidden , these headers are automatically injected and included by postman or even the browser itself. such as Host, User-Agent, Accept, Accept-Encoding, Connection

also i can add my own custom like key - my -header value --value

**Body** ---used for post, put, patch

suppose we use post then we have different type of bodies such as

**none** ------- no body so some body have nothing

**form-data** ------just key way key-value also we pass in test or file

**raw** ----

**binary** --- we can upload file for that

**GraphQL** ---- for query and query variable

here body can be text, javaScript,JSON,HTML,XML this are supported type

**pre-request Script** ----postman also support scripting so we can before sendinig the request we can perform the script for example i can perform script or i can write a script data or to grab it to token somewhere and inject it or save it an environment variable that i can use for example in my authorization or in header an so on

**test** --- when i run or when i want to test my endpoint everything is working fine i can test script here so the script are javascript based

**collections** --- collection of API

**API ----**

**Environment** --- prod, dev, QA

**Mock server** --- use when backend is not fully implemented

**Monitoring -**

**history** -- all api of history

**Post : localhost:8080/post**

response : Request accepted and message is :"+ null // because we are not any body

@PostMapping("/posting")  
public String postMethod(@RequestBody String message){

*// here we need to provide for example* return "Request accepted and message is :"+ message;  
}

// here not able to map or to transform the message or the value of the message we

reveived from postman or our rest client and we are not able to map to the message variable in the method parameter

when we work with spring framework we need always to tell spring and to indicate for each step what is requestbody and what is the response body because spring it smart enough it is complete framework but it will not do the job for us.

so in this case we need to indicate that parameter-- String message parameter here is a request body or should represent the body of our request or of our method we have in order to fix that we need to to add one annotation @RequestBody

**@RequestBody**

this means this is the body of our request and requestbody annotation is used in the spring MVC to bind the http request body with the method parameter it's part of spring approach to build web services and is used in the controller layer where HTTP request is handled so which is part of restcontroller

when we send the request to controller to send the data or body the using @RequstBody when we annotate the method parameter with request body spring will use the HTTP message converter to convert the HTTP request body into the specified java object so here it can convert json data to the request body into the java object

**I want to post an object like data of order in customername, productname, quantity**

package com.app;  
  
import lombok.AllArgsConstructor;  
import lombok.Data;  
import lombok.NoArgsConstructor;  
  
@Data  
@NoArgsConstructor  
@AllArgsConstructor  
public class Order {  
 private String customerName;  
 private String productName;  
 private long quantity;  
*//getter,setter,toString*}

**@RestController**

**public class FirstController{**

@PostMapping("/Order\_Object")  
public String orderdisplay(@RequestBody Order order){  
 return " the order is : "+ order.toString();  
}

private variables we can access with using accessors so the accessors are getters and setters

also known as mutater methods they play crucial role in the serialization and deserialization process in the context of a framework like spring and libraries like Jackson which are commonly used for serializing and deserializing data and form format such as JSON. so this methods have specific uses so for example when an instance of an object is serialized into a format like json the getter methods are called to access the current state of the object.the value returned by these methods are then written into the serialized format.

now for setters during this deserialization setter are used to populate the field of a newly created object with the data from the serialized format.

customize the name of the json data using @JsonProperty on the field of entity like @JsonProperty("c-name')

suppose i want to send data to third party api and the names are not the same as my java object

for example:

@JsonProperty("c-name")  
private String customerName;

**Record is part of java 14**

package com.app;  
  
public record OrderRecord(  
 int id,  
 String customerName,  
 String productName  
  
) {  
}

// in controller

@PostMapping("/save/post-orderRecord")  
public String save(@RequestBody OrderRecord order){  
 return " the orderRecord : " +order.toString();  
}

when to choose old java objects like pojo , records when it comes to data transfer objects whether to to use pojo's plain old java objects or records for data transfer in java depends on the use case and the java version you are using

java records are introduced as a preview feature in java 14 and finalized in java 16 so this is new kind of type declaration in the java langauge a record class is shalllowly in mutable transpart carrier to for a fixed set of values which can be accessed with accessors method that have the same method as the fields records reduce the verbosity of java and make it make it more suitable for data Centric applications.

pojo have been used for many years in java for such tasks they are versatile and work with all java versions but require more code to write you have to declare fields and then you have to write or generate Constructors, getter, setters,equals hash code and two string method if you are using java16

or later and your dto are simple carriers of data logic records may be a better choice they provide a concise and convenient and automatic implementation of quals hash code and two string as we saw in the example

if you need mutable objects or if your dto need to include logic beyond or just storing data

pojo may be a better choice so as always the right choice depends on your specific needs and contraints now lets see how we can pass parameter to a method so let's go back to our first controller and let's see how i can pass a prameter to a specific methods so here i will just create or copy this get mapping method right here and i will put it in the bottom so this works for any type of uh of methods whether they get post delete and so on

@GetMapping("/hello/{check}")  
public String pathVar(@PathVariable("check")String str ){  
 return " the string is : "+str;  
} *// http://localhost:8080/hello/anystring*

whether i was the first one whether i pas the path variable exactly within the same name like the same naming as my variable right here so it will be just like that so spring will automatically refer this user name(check) check right here we will mention to spring that this username(check)right here this variable or parameter that we want to receive right here is the same that we have we have it here and first we can mention the same name so the same variable name or for example if i have a differenet one so to use it like this user-name(check) here this path variable also accept a parameter its' called value so by default it's empty by default it will take the variable name as the variable of as the variable of the default value so here specify it like that and let's say user-name

array = commbination of even and add in this array

use this array so display first even and then display the odd in java

input :5,7,8,9,6,11,10

output : 8,6,10,5,7,9,11

@GetMapping("/hello")  
public String requestParam(@RequestParam("check")String str ){  
 return " the string is : "+str;  
} *// http://localhost:8080/hello?check=anystring*

**what is jackson in springboot? what is the of Jackson?**

Now we should name or how we can name our methods and URLs and how spring is already interpreting that and how requests are dispatched from our rest client to back end so

now we will see the difference between path variable and requestparam so in springboot pathvariable and requestparams annotations are used for extracting values from the url HTTP request they are used in the spring mvc handler methods and here how they differ so first we start with path variable

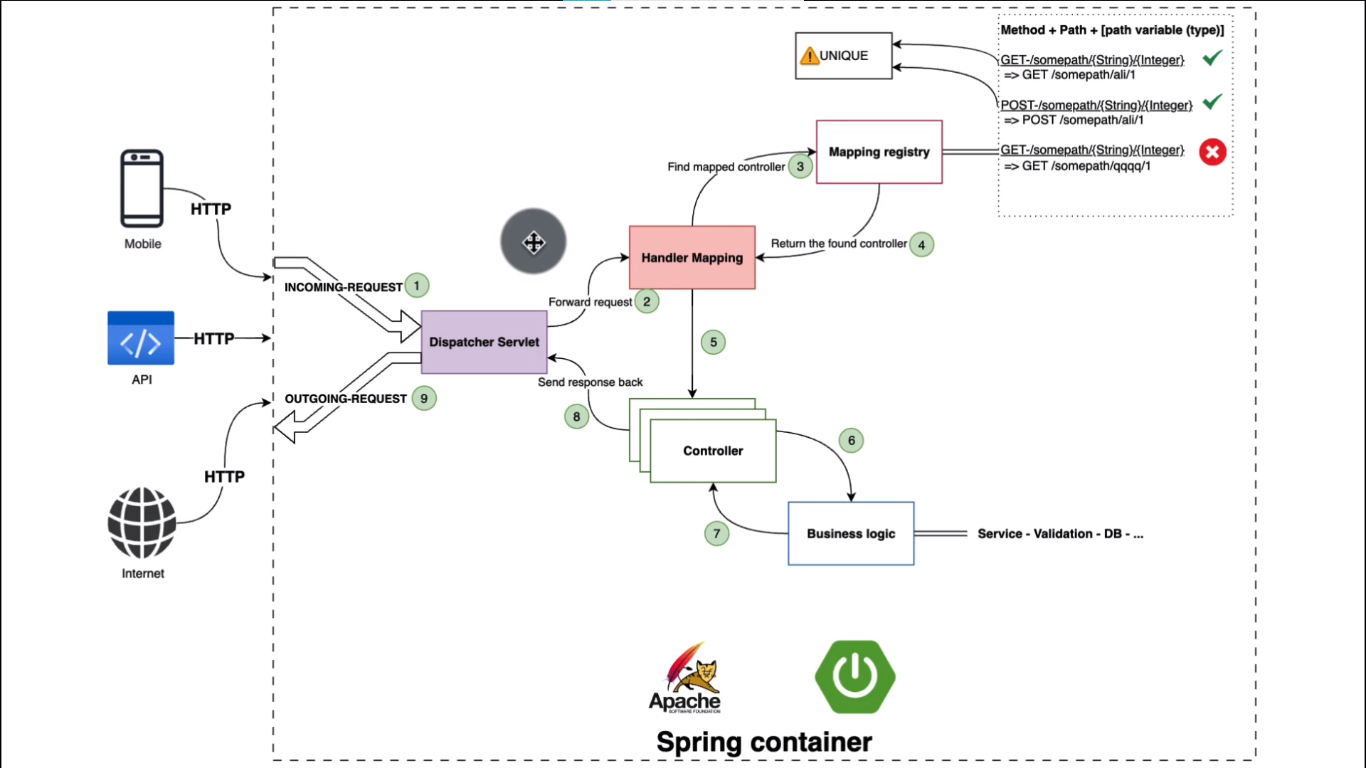
**PathVariable :**

This annotation is used to extract values from the URI path so it’s typically used in restful web services where the URL contains a value representing some sort of resources identifier so the annotated parameter and the method declaration is bound to the path variable of the same name so as we mentioned before so here if you have whether spring is capable of referencing the name that you pass right here as a path variable with the parameter name or the field name or when you use path variable also like you need to use path variable also pass the name right here to match the you have in your url.

**@Requestparam**

This annotation is used to extract **query parameters from the URL so query parameters are typically used to carry context information for the request and are separated from the URI using the exclamation mark symbol and are chained with the end symbol**

When the application start up spring creates a map in between the URLs HTTP methods and the corresponding controller methods so this is done by a class called request mapping handler mapping which scans the controller classes and builds up a registry of such mapping so when an HTTP request comes in the dispatcher servlet which is the front controller in spring MVC consults these mapping to determine which method should handle the request so then the appropriate method is then invocated and its return value is used to generate HTTP response



Here Mobile,API,Internet are client

First step is sending the request so we call this is an incoming request and first thing or the first object that will receive this request is the **front controller** which is the **dispatcher servlet**  then the dispatcher servlet will forward the request to the Handler **mapping class** (like controller/Restcontroller) or the Handler mapping object and this handler mapping will consult the **mapping registry** and it will tries to find a mapped controller and this mapping registry as I explained before so it will map or it has a registry of the method plus the path plus pathvariable and types and it will refers to a specific method of the controller

Then it finds a match it will return the controller and the method that needs to be invocated and the fifth step it will forward this request to the required controller and then we of cource have all the business logic goes along like service validation database registration consulting another API and after that we have the response so the response go back to the controller and controller will send back the response to the dispatcher servlet and then it will go back to the customer or to the client of this request and this is how it exactly works

Composition of the URL must be unique

DBeaver community version

package com.marketingapp.entity;  
import jakarta.persistence.\*;  
import lombok.AllArgsConstructor;  
import lombok.Data;  
import lombok.NoArgsConstructor;  
  
@Entity  
@Data  
@NoArgsConstructor  
@AllArgsConstructor  
@Table(name="tb\_student")  
public class Student {  
   
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private int sid; *// shift+alt select operation copy*   
   
 @Column(name="fname",length=300)  
 private String firstname;  
  
 @Column(name="lname",length=300)  
 private String lastname;  
  
 @Column(length=300,unique = true)  
 private String email;  
   
 private int age;  
}

Repository =====================

package com.marketingapp.repositoty;  
import com.marketingapp.entity.Student;  
import org.springframework.data.jpa.repository.JpaRepository;  
import java.util.List;  
import java.util.Optional;  
  
public interface StudentRepository extends JpaRepository<Student,Integer> {  
 List<Student> findAllByfirstnameContaining(String sname);  
}

=========================

StudentController ------------

package com.marketingapp.controller;  
  
import com.marketingapp.entity.Student;  
import com.marketingapp.repositoty.StudentRepository;  
import org.springframework.web.bind.annotation.\*;  
  
import java.util.List;  
import java.util.Optional;  
  
@RestController  
@RequestMapping("/api/student")  
public class StudentController {  
  
 private final StudentRepository studentRepository;  
  
 public StudentController(StudentRepository studentRepository) {  
 this.studentRepository = studentRepository;  
 }  
  
 @PostMapping("/create")  
 public Student createStudent(@RequestBody Student student){  
 return this.studentRepository.save(student);  
 } *// http://localhost:8080/api/student/create* @GetMapping("/getone/{id}")  
 public Student getStudent(@PathVariable Integer id){  
 Student student = this.studentRepository.findById(id).orElse(new Student());  
 *//Student student = new Student();* return student;  
 } *// http://localhost:8080/api/student/getone/6* @GetMapping("/student-searchname/{sname}")  
 public List<Student> getStudent(@PathVariable String sname){  
 List<Student> allByfirstname = this.studentRepository.findAllByfirstnameContaining(sname);  
 *//Student student = new Student();* return allByfirstname;  
 } *// http://localhost:8080/api/student/student-searchname/su* @GetMapping("/getall")  
 public List<Student> getStudentall(){  
 List<Student> student = this.studentRepository.findAll();  
 *//Student student = new Student();* return student;  
 } *// http://localhost:8080/api/student/getall* @PutMapping("/update/{id}")  
 public Student updateStudent(@PathVariable Integer id, @RequestBody Student student1){  
 Student student = this.studentRepository.findById(id).orElse(new Student());  
 *//Student student = new Student();* student.setFirstname(student1.getFirstname());  
 student.setLastname(student1.getLastname());  
 student.setEmail(student1.getEmail());  
 student.setAge(student1.getAge());  
 return this.studentRepository.save(student);  
 } *// http://localhost:8080/api/student/update/6* @DeleteMapping("/delete/{id}")  
 public void studentStudent(@PathVariable Integer id){  
 Student student = this.studentRepository.findById(id).orElse(new Student());  
 *//Student student = new Student();* this.studentRepository.delete(student);  
 } *// http://localhost:8080/api/student/delete/6*}

====================== Hibernate Mapping ============================

Student studing in school and student have student profile…

First create class of school and studentprofile

Student has has only one profile --- one To One

Many Student studying in one school --- many ToOne

One School has zero or Many Student----OneToMany

======== lets create mapping first school to student then student to studentprofile

School class

package com.marketingapp.entity;  
import jakarta.persistence.\*;  
import lombok.AllArgsConstructor;  
import lombok.Data;  
import lombok.NoArgsConstructor;  
import java.io.Serializable;  
import java.util.List;  
  
@Entity  
@Data  
@NoArgsConstructor  
@AllArgsConstructor  
@Table(name="tb\_school")  
public class School {  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private Integer schoolid;  
  
 @Column(name="s\_name")  
 private String schoolname;  
  
 @OneToMany(  
 mappedBy = "school"  
 )  
 private List<Student> students;  
}

StudentProfile -----------

package com.marketingapp.entity;  
import jakarta.persistence.\*;  
import lombok.AllArgsConstructor;  
import lombok.Data;  
import lombok.NoArgsConstructor;  
import java.io.Serializable;  
  
@Entity  
@Data  
@NoArgsConstructor  
@AllArgsConstructor  
@Table(name="tb\_studentProfile")  
public class StudentProfile implements Serializable {  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private Integer id;  
  
 private String bio;  
  
 @OneToOne  
 @JoinColumn(  
 name="student\_id"  
 )  
 private Student student1;  
 *//here student ref match with student class mappedBy otherwise you will get exception/error*}

Student class ---

package com.marketingapp.entity;  
import jakarta.persistence.\*;  
import lombok.AllArgsConstructor;  
import lombok.Data;  
import lombok.NoArgsConstructor;  
import java.io.Serializable;  
  
@Entity  
@Data  
@NoArgsConstructor  
@AllArgsConstructor  
@Table(name="tb\_student")  
public class Student implements Serializable {  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private int sid; *// shift+alt select operation copy past* @Column(name="fname",length=300)  
 private String firstname;  
  
 @Column(name="lname",length=300)  
 private String lastname;  
  
 @Column(length=300,unique = true)  
 private String email;  
  
 private int age;  
  
 @OneToOne(  
 mappedBy="student1",  
 cascade = CascadeType.*ALL* )  
 private StudentProfile studentProfile;  
}

update Student class

package com.marketingapp.entity;  
import jakarta.persistence.\*;  
import lombok.AllArgsConstructor;  
import lombok.Data;  
import lombok.NoArgsConstructor;  
import java.io.Serializable;  
  
@Entity  
@Data  
@NoArgsConstructor  
@AllArgsConstructor  
@Table(name="tb\_student")  
public class Student {  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private int sid; *// shift+alt select operation copy past* @Column(name="fname",length=300)  
 private String firstname;  
  
 @Column(name="lname",length=300)  
 private String lastname;  
  
 @Column(length=300,unique = true)  
 private String email;  
  
 private int age;  
  
 @OneToOne(  
 mappedBy="student1",  
 cascade = CascadeType.*ALL* )  
 private StudentProfile studentProfile;  
  
 @ManyToOne  
 @JoinColumn(name = "school\_id"  
 )  
 private School school;  
}

**SchoolController -**

package com.marketingapp.controller;  
import com.marketingapp.entity.School;  
import com.marketingapp.entity.Student;  
import com.marketingapp.repositoty.SchoolRepository;  
import com.marketingapp.repositoty.StudentRepository;  
import org.springframework.web.bind.annotation.\*;  
import java.util.List;  
  
@RestController  
public class SchoolController {  
  
 private final SchoolRepository schoolRepository;  
  
 public SchoolController(SchoolRepository schoolRepository) {  
 this.schoolRepository = schoolRepository;  
 }  
  
 @PostMapping("/create")  
 public School createStudent(@RequestBody School school){  
 return this.schoolRepository.save(school);  
 } *// http://localhost:8080/create* @GetMapping("/getone/{id}")  
 public School getStudent(@PathVariable Integer id){  
 School school = this.schoolRepository.findById(id).orElse(new School());  
 *//Student student = new Student();* return school;  
 } *// http://localhost:8080/getone/6* @GetMapping("/school-searchname/{sname}")  
 public List<School> getStudent(@PathVariable String sname){  
 List<School> allByfirstname = this.schoolRepository.findAllByschoolnameContaining(sname);  
 *//Student student = new Student();* return allByfirstname;  
 } *// http://localhost:8080/student-searchname/su* @GetMapping("/getall")  
 public List<School> getStudentall(){  
 List<School> studentList = this.schoolRepository.findAll();  
 *//Student student = new Student();* return studentList;  
 } *// http://localhost:8080/getall* @PutMapping("/update/{id}")  
 public School updateStudent(@PathVariable Integer id, @RequestBody School school1){  
 School school = this.schoolRepository.findById(id).orElse(new School());  
 *//Student school = new Student();* school.setSchoolname(school1.getSchoolname());  
 return this.schoolRepository.save(school);  
 } *// http://localhost:8080/update/6* @DeleteMapping("/delete/{id}")  
 public void studentStudent(@PathVariable Integer id){  
 School Sc = this.schoolRepository.findById(id).orElse(new School());  
 *//Student student = new Student();* this.schoolRepository.delete(Sc);  
 } *// http://localhost:8080/delete/6*}

now when we save the data that time recursion happen here

{

"firstname":"santosh",

"lastname":"biraj",

"email":"sant@gmail.com",

"age":100,

"school":{

"schoolid":1

}

}

So that’s why we change the entity in school and student

package com.marketingapp.entity;  
import com.fasterxml.jackson.annotation.JsonManagedReference;  
import jakarta.persistence.\*;  
import lombok.AllArgsConstructor;  
import lombok.Data;  
import lombok.NoArgsConstructor;  
import java.io.Serializable;  
import java.util.List;  
  
@Entity  
@Data  
@NoArgsConstructor  
@AllArgsConstructor  
@Table(name="tb\_school")  
public class School {  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private Integer schoolid;  
  
 @Column(name="s\_name")  
 private String schoolname;  
  
 @OneToMany(  
 mappedBy = "school"  
 )  
 @JsonManagedReference *// this tells jackson that the parent is in charge of serializing and child and prevents the child fromm trying to serialize the parent  
 // now we need to the json manager on parent level* private List<Student> students;  
}

====================================================

package com.marketingapp.entity;  
import com.fasterxml.jackson.annotation.JsonBackReference;  
import jakarta.persistence.\*;  
import lombok.AllArgsConstructor;  
import lombok.Data;  
import lombok.NoArgsConstructor;  
import java.io.Serializable;  
  
@Entity  
@Data  
@NoArgsConstructor  
@AllArgsConstructor  
@Table(name="tb\_student")  
public class Student {  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private int sid; *// shift+alt select operation copy past* @Column(name="fname",length=300)  
 private String firstname;  
  
 @Column(name="lname",length=300)  
 private String lastname;  
  
 @Column(length=300,unique = true)  
 private String email;  
  
 private int age;  
  
 @OneToOne(  
 mappedBy="student1",  
 cascade = CascadeType.*ALL* )  
 private StudentProfile studentProfile;  
  
 @ManyToOne  
 @JoinColumn(name = "school\_id"  
 )  
 @JsonBackReference

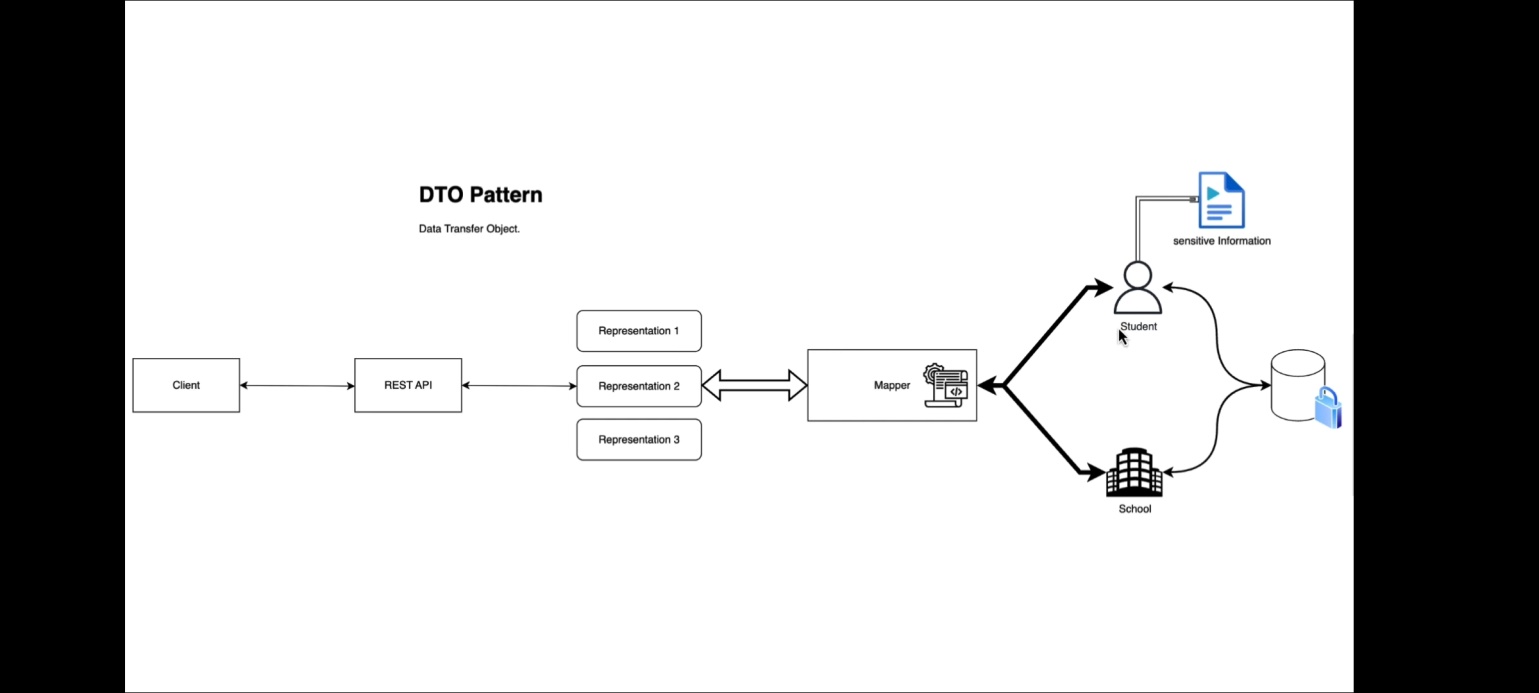
*// student object doesn't need to serialize the parent which is in our case school* private School school;  
}

// now when you save then it will not recursive

Now when you save then u will not get recursion

We will see DTO ----4.57.51s

Data Transfer Object



The main purpose of dto is to encapsulate and structure data that needs to be transferred between different parts of the system or different systems entirely

* DTO typically includes only simple data field or also called attribute and lacks the behavior of the model or entity it represents so here if we take the example of our application we have student and we have a school we have also student profile which will hold some sensitive information about the student like the address like phone number , dob so and so.
* Mainly we don’t need to expose this information to the outside world and also for example if you want to retrieve the information of the school we only need to school information and since this information are represented by entities and stored into a database system which is secured system so we need to expose only what user or the end user expecting and no more extra information so for example I mentioned for the user we for the student we don’t need to expose sensitive information such as address, dob, phone number, even in the student has a has password so we don’t need to expose that so then what we need to include a mechanism in the middle in between the entity itself and the outside world so need to add a mapper and this mapper is just a simple mechanism you can use many implementation or many tools and libraries to map objects and then if we use a mapper we can have multiple representation for the same object for example for the student we can have a representation one only to expose first name as name and we can have another representation to expose or receive data from the outside world in order to create a new student and for example only we need firstname lastname and email so this is one of the advantages is we can multiple representation for only one object or only one entity and this also applicable for school student or student profile
* The restapi should communicate with a client through these representation and we can have one representation for the read and other representation for the right example to create student we can have one representation which holds the information or like minimum required information to create a student and for the read representation we canhave a different one or a different object that will be responsible for exposing only the necessary and nonsensitive information to the outside the world
* So when building restful API DTO play a crucial role and now when it is important
* First of all we have data separation so the data transfer pattern helps to separate the internal domain model from what is exposed through the API this way we can change your internal model without affecting external representation it also helps in not exposing sensitive or unnecessary data to the API customers,
* Then abstraction DTO provide an abstraction layer they give you a clear structure of what your API will provide to the customers/client abstracting the complexity of the domain model
* The performance improvements instead of sending entire entities over the network you can send instead send a DTO which include only the necessary information these can significantly improve the performance of your API as you can control the size of your response payload.
* Then we have flexibility since DTO are separate from your domain model they allow you to tailor API response to exactly what your client need even if that does not exactly match your domain model here we are talking about multiple representations for the same object
* Finally versioning so DTO make it easier to maintain different version of your API by using different sets of DTO you can support multiple version of an API simultaneously and then next part we can use and implement in appln.

Implementation start here =======================

package com.marketingapp.payload;  
  
public record StudentDto(  
 String firstname,  
 String lastname,  
 String email,  
 int age,  
 Integer schoolid  
) {  
}

**controller**

package com.marketingapp.controller;  
  
import com.marketingapp.entity.School;  
import com.marketingapp.entity.Student;  
import com.marketingapp.payload.StudentDto;  
import com.marketingapp.repositoty.StudentRepository;  
import org.springframework.web.bind.annotation.\*;  
  
import java.util.List;  
import java.util.Optional;  
  
@RestController  
@RequestMapping("/api/student")  
public class StudentController {  
  
 private final StudentRepository studentRepository;  
  
 public StudentController(StudentRepository studentRepository) {  
 this.studentRepository = studentRepository;  
 }  
  
 @PostMapping("/create")  
 public Student createStudent(@RequestBody StudentDto studentDto){  
 Student student = this.dtoToStudent(studentDto); *// dto to student convert* return this.studentRepository.save(student);  
 } *// http://localhost:8080/api/student/create  
  
 /\*  
 {  
"firstname":"santosh",  
"lastname":"biraj",  
"email":"sant@gmail.com",  
"age":100,  
"school":{  
"schoolid":1  
}  
}  
---------------  
{  
"firstname":"sushma",  
"lastname":"bj",  
"email":"sushmaji@gmail.com",  
"schoolid":2  
}  
-----response we are getting -------------------  
{  
 "sid": 8,  
 "firstname": "sushma",  
 "lastname": "bj",  
 "email": "sushmaji@gmail.com",  
 "age": 0,  
 "studentProfile": null  
} here we dont want age,studentprofile so we create studentResponseDto  
 \*/* private Student dtoToStudent(StudentDto studentdto){ *// DTO method* var student = new Student();  
 student.setFirstname(studentdto.firstname()); *// used record here studentdto* student.setLastname(studentdto.lastname());  
 student.setEmail(studentdto.email());  
  
 var school = new School();  
 school.setSchoolid(studentdto.schoolid()); *// school id from school entity* student.setSchool(school);  
 return student;  
 }

}

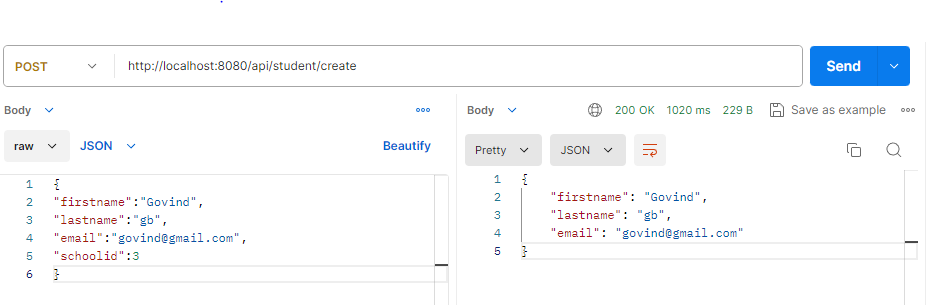
===============

package com.marketingapp.payload;  
  
public record StudentResponseDto( //when we save record of student it will return  
 String firstname,  
 String lastname,  
 String email  
) {  
}

==================

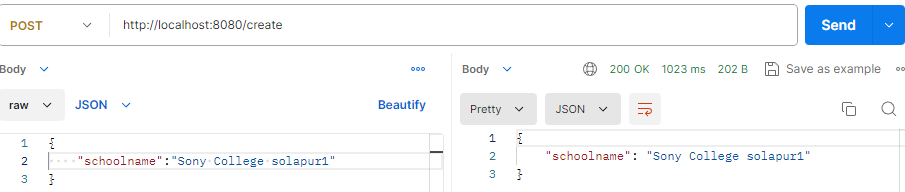
StudentController --------------------

package com.marketingapp.controller;  
  
import com.marketingapp.entity.School;  
import com.marketingapp.entity.Student;  
import com.marketingapp.payload.StudentDto;  
import com.marketingapp.payload.StudentResponseDto;  
import com.marketingapp.repositoty.StudentRepository;  
import org.springframework.web.bind.annotation.\*;  
  
import java.util.List;  
import java.util.Optional;  
  
@RestController  
@RequestMapping("/api/student")  
public class StudentController {  
  
 private final StudentRepository studentRepository;  
  
 public StudentController(StudentRepository studentRepository) {  
 this.studentRepository = studentRepository;  
 }  
  
 @PostMapping("/create")  
 public StudentResponseDto createStudent(@RequestBody StudentDto studentDto){  
 Student student = this.dtoToStudent(studentDto); *// dto to student convert* Student save = this.studentRepository.save(student); *// saved* return dtoToStudent(save); *// entity to dto* } *// http://localhost:8080/api/student/create  
  
 /\*  
 {  
"firstname":"santosh",  
"lastname":"biraj",  
"email":"sant@gmail.com",  
"age":100,  
"school":{  
"schoolid":1  
}  
}  
---------------  
{  
"firstname":"sushma",  
"lastname":"bj",  
"email":"sushmaji@gmail.com",  
"schoolid":2  
}  
-----response we are getting -------------------  
{  
 "sid": 8,  
 "firstname": "sushma",  
 "lastname": "bj",  
 "email": "sushmaji@gmail.com",  
 "age": 0,  
 "studentProfile": null  
} here we dont want age,studentprofile so we create studentResponseDto  
 \*/* private Student dtoToStudent(StudentDto studentdto){ *// DTO method* var student = new Student();  
 student.setFirstname(studentdto.firstname()); *// used record here studentdto* student.setLastname(studentdto.lastname());  
 student.setEmail(studentdto.email());  
  
 var school = new School();  
 school.setSchoolid(studentdto.schoolid()); *// school id from school entity* student.setSchool(school);  
 return student;  
 }  
   
private StudentResponseDto studentResponseDto(Student student){  
 return new StudentResponseDto(  
 student.getFirstname(),  
 student.getLastname(),  
 student.getEmail()  
 );  
}



package com.app.dto;  
  
public record SchoolDto(  
 String schoolname  
) {  
}

package com.marketingapp.controller;  
  
import com.marketingapp.entity.School;  
import com.marketingapp.entity.Student;  
import com.marketingapp.payload.SchoolDto;  
import com.marketingapp.repositoty.SchoolRepository;  
import com.marketingapp.repositoty.StudentRepository;  
import org.springframework.web.bind.annotation.\*;  
  
import java.util.List;  
  
@RestController  
public class SchoolController {  
  
 private final SchoolRepository schoolRepository;  
  
 public SchoolController(SchoolRepository schoolRepository) {  
 this.schoolRepository = schoolRepository;  
 }  
  
 @PostMapping("/create")  
 public SchoolDto createStudent(@RequestBody SchoolDto schoolDto){  
 School school = DtoToSchoolEntity(schoolDto); *//toSchool* School save = this.schoolRepository.save(school);  
 return schoolDto;  
 } *// http://localhost:8080/create  
  
// private SchoolDto schoolEntityToDto(School save) {  
// SchoolDto sb = new SchoolDto();  
// sb.schoolname(save.getSchoolname());  
// }* private School DtoToSchoolEntity(SchoolDto schoolDto) {  
 var school = new School();  
 school.setSchoolname(schoolDto.schoolname());  
 return school;  
 }



private SchoolDto toSchoolDto(School school){  
 return new SchoolDto(school.getSchoolid(),school.getSchoolname());  
}  
@GetMapping("/getall")  
public List<SchoolDto> getStudentall(){  
 *// List<School> studentList = this.schoolRepository.findAll();* return schoolRepository.findAll()  
 .stream()  
 .map(this::toSchoolDto)  
 .collect(Collectors.*toList*());  
  
 *//Student student = new Student();  
 //return studentList;*}

code reusable for separation concern and handling business logic and application specific operation, from the presentation layer is controller layer, data access layer which is repository layer so it serves as an intermediate between controller and repository encapsulating business logic and providing resusable modular and testable code base

WHY SERVICE LAYER IS IMPORTANT

* Business logic encapsulation
* Service layer is responsible for implementing business logic and performing complex operation required by the API. It encapsulate the logic that goes beyond simple data retrieve or modification such as **validation, calculation, transformation or coordination of multiple operations** so by centralizing the business logic in the service layer you promote code **reusability maintainability and readability**
* Service layer provide a separation of concerns so the service layer helps to maintain a clear separation of concerns within the API architecture
* Controllers are responsible for handling the **incoming HTTP request** providing appropriate responses.
* While repositories are responsible for handle the **data persistence** and retrieval.
* Service layer sits in between ensuring the business logic and application specific operation are decoupled from other layers so this separation improves code organization makes code easier to understand allows for independent changes and testing of each layer. and finally testing and maintainability
* Service layer provides a boundary of the **Unit testing** as you can test business logic independently for the controllers and repositories by **mocking or stabbing** the dependencies you can focus solely on testing the logic within the service method so this makes it easier to write comprehensive and targeted tested enhancing test coverage and ensuring the correctness of the business logic.
* Additionally separation of the concerns achieved by the service layer improves the maintainability of the code base as changes to the business logic and can be isolated to the service layer without affecting to the other layers

Now we will extract the code with controller and service layer

Create package of mapper inside create class **StudentMapper** ---- 5.26.10 ----

package com.marketingapp.mapper;  
  
import com.marketingapp.entity.School;  
import com.marketingapp.entity.Student;  
import com.marketingapp.payload.StudentDto;  
import com.marketingapp.payload.StudentResponseDto;  
import org.springframework.stereotype.Service;  
  
@Service  
public class StudentMapper {  
  
 public Student dtoToStudent(StudentDto studentdto){ *// DTO method* var student = new Student();  
 student.setFirstname(studentdto.firstname()); *// used record here studentdto* student.setLastname(studentdto.lastname());  
 student.setEmail(studentdto.email());  
  
 var school = new School();  
 school.setSchoolid(studentdto.schoolid()); *// school id from school entity* student.setSchool(school);  
 return student;  
 }  
  
 public StudentResponseDto dtoToStudent(Student student){  
 return new StudentResponseDto(  
 student.getFirstname(),  
 student.getLastname(),  
 student.getEmail()  
 );  
 }  
}

=====

package com.marketingapp.controller;  
import com.marketingapp.entity.Student;  
import com.marketingapp.mapper.StudentMapper;  
import com.marketingapp.payload.StudentDto;  
import com.marketingapp.payload.StudentResponseDto;  
import com.marketingapp.repositoty.StudentRepository;  
import org.springframework.web.bind.annotation.\*;  
import java.util.List;  
  
@RestController  
@RequestMapping("/api/student")  
public class StudentController {  
  
 private final StudentRepository studentRepository;  
 private final StudentMapper studentMapper;  
  
 public StudentController(StudentRepository studentRepository, StudentMapper studentMapper) {  
 this.studentRepository = studentRepository;  
 this.studentMapper = studentMapper;  
 }  
  
@PostMapping("/create")  
public StudentResponseDto createStudent(@RequestBody StudentDto studentDto){  
Student student = studentMapper.dtoToStudent(studentDto); *// dto to student convert* Student save = this.studentRepository.save(student); *// saved* return studentMapper.dtoToStudent(save); *// entity to dto* } *// http://localhost:8080/api/student/create*

again we will separate code through service layer

package com.marketingapp.controller;  
  
import com.marketingapp.entity.Student;  
import com.marketingapp.payload.StudentDto;  
import com.marketingapp.payload.StudentResponseDto;  
import com.marketingapp.service.StudentService;  
import org.springframework.web.bind.annotation.\*;  
  
import java.util.List;  
  
@RestController  
@RequestMapping("/api/student")  
public class StudentController {  
  
 private final StudentService studentService;  
  
 public StudentController(StudentService studentService) {  
 this.studentService = studentService;  
 }  
  
 @PostMapping("/create")  
 public StudentResponseDto createStudent(@RequestBody StudentDto studentDto){  
 return studentService.saveStudent(studentDto);  
 } *// http://localhost:8080/api/student/create  
  
 /\*  
 {  
"firstname":"santosh",  
"lastname":"biraj",  
"email":"sant@gmail.com",  
"age":100,  
"school":{  
"schoolid":1  
}  
}  
---------------  
{  
"firstname":"sushma",  
"lastname":"bj",  
"email":"sushmaji@gmail.com",  
"schoolid":2  
}  
-----response we are getting -------------------  
{  
 "sid": 8,  
 "firstname": "sushma",  
 "lastname": "bj",  
 "email": "sushmaji@gmail.com",  
 "age": 0,  
 "studentProfile": null  
} here we dont want age,studentprofile so we create studentResponseDto  
 \*/* @GetMapping("/getone/{id}")  
 public Student getStudent(@PathVariable Integer id){  
 return studentService.getStudent(id);  
 } *// http://localhost:8080/api/student/getone/6* @GetMapping("/student-searchname/{sname}")  
 public List<Student> getStudent(@PathVariable String sname){  
 return this.studentService.getStudentwithname(sname);  
 } *// http://localhost:8080/api/student/student-searchname/su* @GetMapping("/getall")  
 public List<Student> getStudentall(){  
 return this.studentService.getStudentall();  
 } *// http://localhost:8080/api/student/getall* @PutMapping("/update/{id}")  
 public Student updateStudent(@PathVariable Integer id, @RequestBody Student student1){  
 return this.studentService.updateStudent(id,student1);  
 } *// http://localhost:8080/api/student/update/6* @DeleteMapping("/delete/{id}")  
 public void studentStudent(@PathVariable Integer id){  
 this.studentService.deleteStudent(id);  
 } *// http://localhost:8080/api/student/delete/6*}

service

package com.marketingapp.service;  
  
import com.marketingapp.entity.Student;  
import com.marketingapp.mapper.StudentMapper;  
import com.marketingapp.payload.StudentDto;  
import com.marketingapp.payload.StudentResponseDto;  
import com.marketingapp.repositoty.StudentRepository;  
import org.springframework.stereotype.Service;  
import org.springframework.web.bind.annotation.\*;  
  
import java.util.List;  
  
@Service  
public class StudentService {  
   
 private final StudentRepository studentRepository;  
 private final StudentMapper studentMapper;  
  
 public StudentService(StudentRepository studentRepository, StudentMapper studentMapper) {  
 this.studentRepository = studentRepository;  
 this.studentMapper = studentMapper;  
 }  
  
 public StudentResponseDto saveStudent(  
 StudentDto studentDto  
 ){  
 Student student = studentMapper.dtoToStudent(studentDto); *// dto to student convert* Student save = this.studentRepository.save(student); *// saved* return studentMapper.dtoToStudent(save); *// entity to dto* }  
  
 public Student getStudent(Integer id){  
 Student student = this.studentRepository.findById(id).orElse(new Student());  
 return student;  
 }  
 public List<Student> getStudentwithname(String sname){  
 List<Student> allByfirstname = this.studentRepository.findAllByfirstnameContaining(sname);  
 return allByfirstname;  
 }  
 public List<Student> getStudentall(){  
 List<Student> student = this.studentRepository.findAll();  
 return student;  
 }  
  
 public Student updateStudent(Integer id,Student student1){  
 Student student = this.studentRepository.findById(id).orElse(new Student());  
 *//Student student = new Student();* student.setFirstname(student1.getFirstname());  
 student.setLastname(student1.getLastname());  
 student.setEmail(student1.getEmail());  
 student.setAge(student1.getAge());  
 return this.studentRepository.save(student);  
 }  
 public void deleteStudent(Integer id){  
 Student student = this.studentRepository.findById(id).orElse(new Student());  
 *//Student student = new Student();* this.studentRepository.delete(student);  
 }  
}

now we change the response in service and controller layer

controller

package com.marketingapp.controller;  
  
import com.marketingapp.entity.Student;  
import com.marketingapp.payload.StudentDto;  
import com.marketingapp.payload.StudentResponseDto;  
import com.marketingapp.service.StudentService;  
import org.springframework.web.bind.annotation.\*;  
  
import java.util.List;  
  
@RestController  
@RequestMapping("/api/student")  
public class StudentController {  
  
 private final StudentService studentService;  
  
 public StudentController(StudentService studentService) {  
 this.studentService = studentService;  
 }  
  
 @PostMapping("/create")  
 public StudentResponseDto createStudent(@RequestBody StudentDto studentDto){  
 return studentService.saveStudent(studentDto);  
 } *// http://localhost:8080/api/student/create  
  
 /\*  
 {  
"firstname":"santosh",  
"lastname":"biraj",  
"email":"sant@gmail.com",  
"age":100,  
"school":{  
"schoolid":1  
}  
}  
---------------  
{  
"firstname":"sushma",  
"lastname":"bj",  
"email":"sushmaji@gmail.com",  
"schoolid":2  
}  
-----response we are getting -------------------  
{  
 "sid": 8,  
 "firstname": "sushma",  
 "lastname": "bj",  
 "email": "sushmaji@gmail.com",  
 "age": 0,  
 "studentProfile": null  
} here we dont want age,studentprofile so we create studentResponseDto  
 \*/* @GetMapping("/getone/{id}")  
 public StudentResponseDto getStudent(@PathVariable Integer id){  
 return studentService.getStudent(id);  
 } *// http://localhost:8080/api/student/getone/6* @GetMapping("/student-searchname/{sname}")  
 public List<StudentResponseDto> getStudent(@PathVariable String sname){  
 return this.studentService.getStudentwithname(sname);  
 } *// http://localhost:8080/api/student/student-searchname/su* @GetMapping("/getall")  
 public List<StudentResponseDto> getStudentall(){  
 return this.studentService.getStudentall();  
 } *// http://localhost:8080/api/student/getall* @PutMapping("/update/{id}")  
 public Student updateStudent(@PathVariable Integer id, @RequestBody Student student1){  
 return this.studentService.updateStudent(id,student1);  
 } *// http://localhost:8080/api/student/update/6* @DeleteMapping("/delete/{id}")  
 public void studentStudent(@PathVariable Integer id){  
 this.studentService.deleteStudent(id);  
 } *// http://localhost:8080/api/student/delete/6*}

service

package com.marketingapp.service;  
  
import com.marketingapp.entity.Student;  
import com.marketingapp.mapper.StudentMapper;  
import com.marketingapp.payload.StudentDto;  
import com.marketingapp.payload.StudentResponseDto;  
import com.marketingapp.repositoty.StudentRepository;  
import org.springframework.stereotype.Service;  
import org.springframework.web.bind.annotation.\*;  
  
import java.util.List;  
import java.util.stream.Collectors;  
  
@Service  
public class StudentService {  
  
 private final StudentRepository studentRepository;  
 private final StudentMapper studentMapper;  
  
 public StudentService(StudentRepository studentRepository, StudentMapper studentMapper) {  
 this.studentRepository = studentRepository;  
 this.studentMapper = studentMapper;  
 }  
  
 public StudentResponseDto saveStudent(  
 StudentDto studentDto  
 ){  
 Student student = studentMapper.dtoToStudent(studentDto); *// dto to student convert* Student save = this.studentRepository.save(student); *// saved* return studentMapper.studentToDto(save); *// entity to dto* }  
  
 public StudentResponseDto getStudent(Integer id){  
 return this.studentRepository.findById(id)  
 .map(studentMapper::studentToDto)  
 .orElse(null);  
 }  
 public List<StudentResponseDto> getStudentwithname(String sname){  
 return this.studentRepository.findAllByfirstnameContaining(sname)  
 .stream().map(studentMapper::studentToDto).collect(Collectors.*toList*());  
 }  
 public List<StudentResponseDto> getStudentall(){  
 return this.studentRepository.findAll()  
 .stream()  
 .map(studentMapper::studentToDto)  
 .collect(Collectors.*toList*());  
 }  
  
 public Student updateStudent(Integer id,Student student1){  
 Student student = this.studentRepository.findById(id).orElse(new Student());  
 *//Student student = new Student();* student.setFirstname(student1.getFirstname());  
 student.setLastname(student1.getLastname());  
 student.setEmail(student1.getEmail());  
 student.setAge(student1.getAge());  
 return this.studentRepository.save(student);  
 }  
 public void deleteStudent(Integer id){  
 Student student = this.studentRepository.findById(id).orElse(new Student());  
 *//Student student = new Student();* this.studentRepository.delete(student);  
 }  
}

school service ----- modelmapper ---5.40.30s

perform school class on extract and maintain

extract schoolMapper and perform on service

package com.app.mapper;  
  
import com.app.dto.SchoolDto;  
import com.app.entity.School;  
  
public class SchoolMapper {  
  
 public School DtoToSchoolEntity(SchoolDto schoolDto) {  
 var school = new School();  
 school.setSchoolname(schoolDto.schoolname());  
 return school;  
 }  
  
 public SchoolDto toSchoolDto(School school){  
 return new SchoolDto(school.getSchoolid(),school.getSchoolname());  
 }  
}

===============

package com.app.service;  
  
import com.app.dto.SchoolDto;  
import com.app.entity.School;  
import com.app.mapper.SchoolMapper;  
import com.app.repository.SchoolRepository;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;  
import org.springframework.web.bind.annotation.\*;  
  
import java.util.List;  
import java.util.stream.Collectors;  
  
@Service  
public class SchoolService {  
  
 @Autowired  
 private SchoolRepository schoolRepository;  
  
 @Autowired  
 private SchoolMapper mapper;  
  
 public SchoolDto createSchool(SchoolDto schoolDto){  
 School school = mapper.DtoToSchoolEntity(schoolDto); *//toSchool* this.schoolRepository.save(school);  
 return schoolDto;  
 } *// http://localhost:8080/create* public School getSchoolById(Integer id){  
 School school = this.schoolRepository.findById(id).orElse(new School());  
 *//Student student = new Student();* return school;  
 } *// http://localhost:8080/getone/6* public List<School> getSchool(String sname){  
 List<School> allByfirstname = this.schoolRepository.findAllByschoolnameContaining(sname);  
 *//Student student = new Student();* return allByfirstname;  
 } *// http://localhost:8080/student-searchname/su* public List<SchoolDto> getSchoolall(){  
 *// List<School> studentList = this.schoolRepository.findAll();* return schoolRepository.findAll()  
 .stream()  
 .map(mapper::toSchoolDto)  
 .collect(Collectors.*toList*());  
 } *// http://localhost:8080/getall* public School updateSchool(Integer id, School school1){  
 School school = this.schoolRepository.findById(id).orElse(new School());  
 *//Student school = new Student();* school.setSchoolname(school1.getSchoolname());  
 return this.schoolRepository.save(school);  
 } *// http://localhost:8080/update/6* public void deleteSchool(Integer id){  
 School Sc = this.schoolRepository.findById(id).orElse(new School());  
 *//Student student = new Student();* this.schoolRepository.delete(Sc);  
 } *// http://localhost:8080/delete/6*}

for example if you want to search specific class or specific object to update or perform any operation on it is really bit hard to find it

so here is a simple example imagine if you are working on a bigger application for example an Ecommerce application within the same code base so this makes our life a bit complicated to find and identify the correct object to update so for this

so i want to introduce you to a topic --- which is the best way to organize your code within your application and for this one

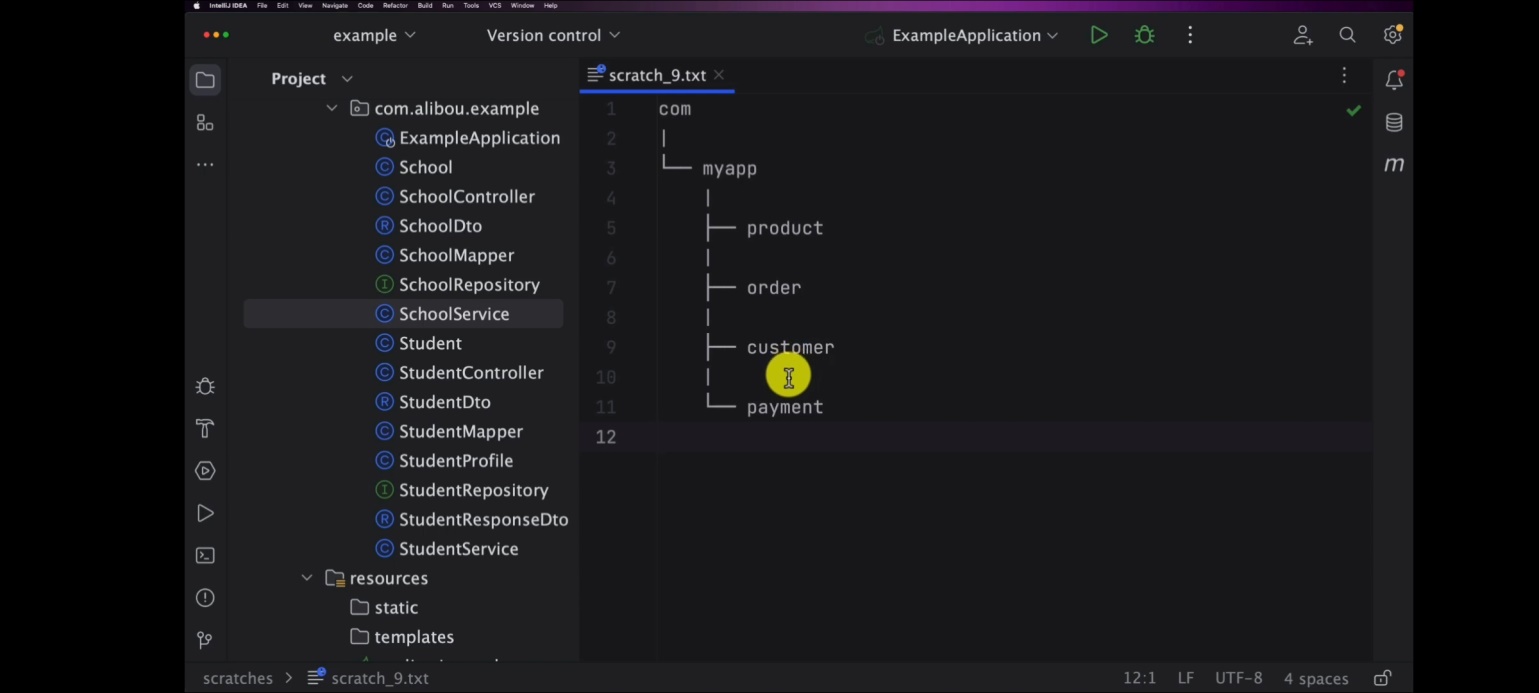
**we have saveral approaches**

I will walk you through each one of them and explain how to organize This the code using each of these approaches and then we will choose together the best approach

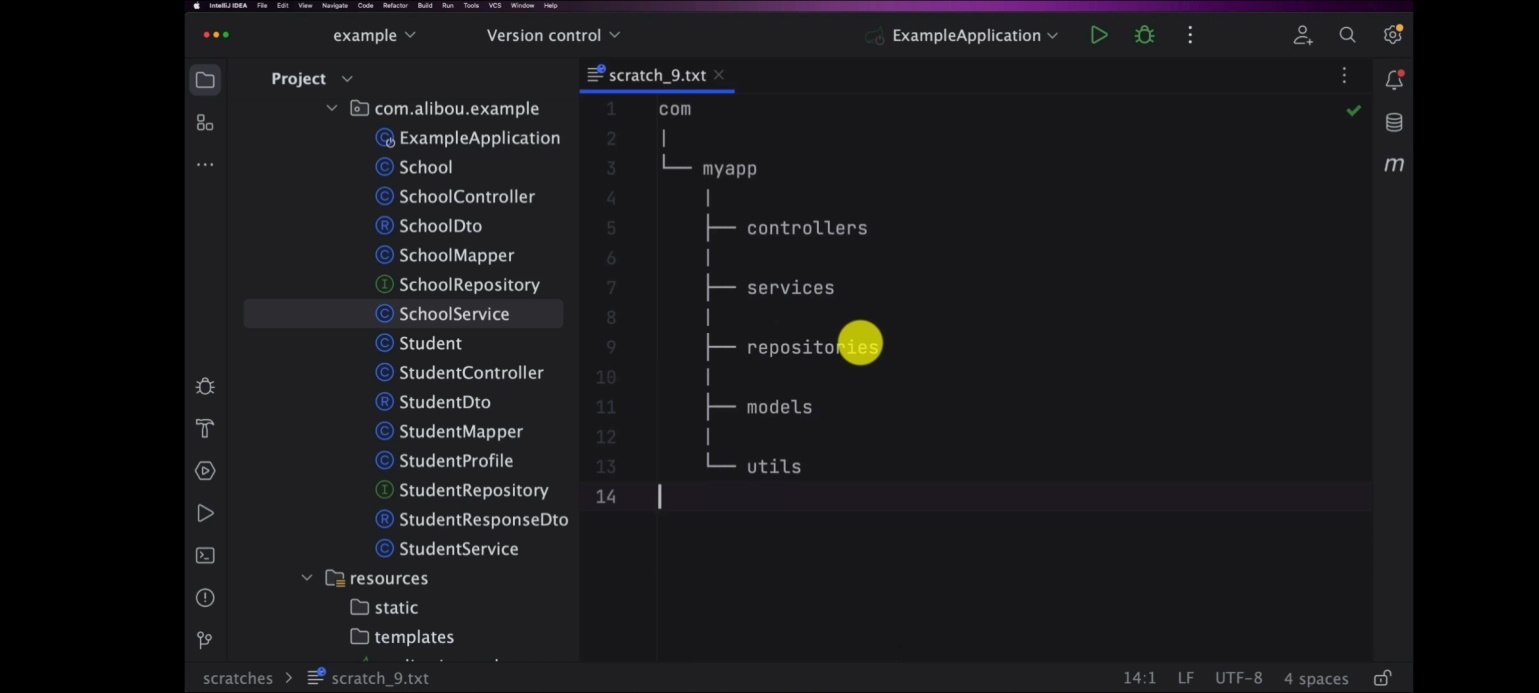
Or i will give me recommendation for the best approach to organize the code but you are always free to choose the best approach that suits you so

1. Let’s the first first approach is called bi feature in this approach the code is organized around the business capabilities of features of the applications for instance if you are developing an ecommerce application you might packages like product order customer and payment. and each of these packages would then contain all classes related to that particular feature such as controller service repository model so this organization style is beneficial for large teams and project because

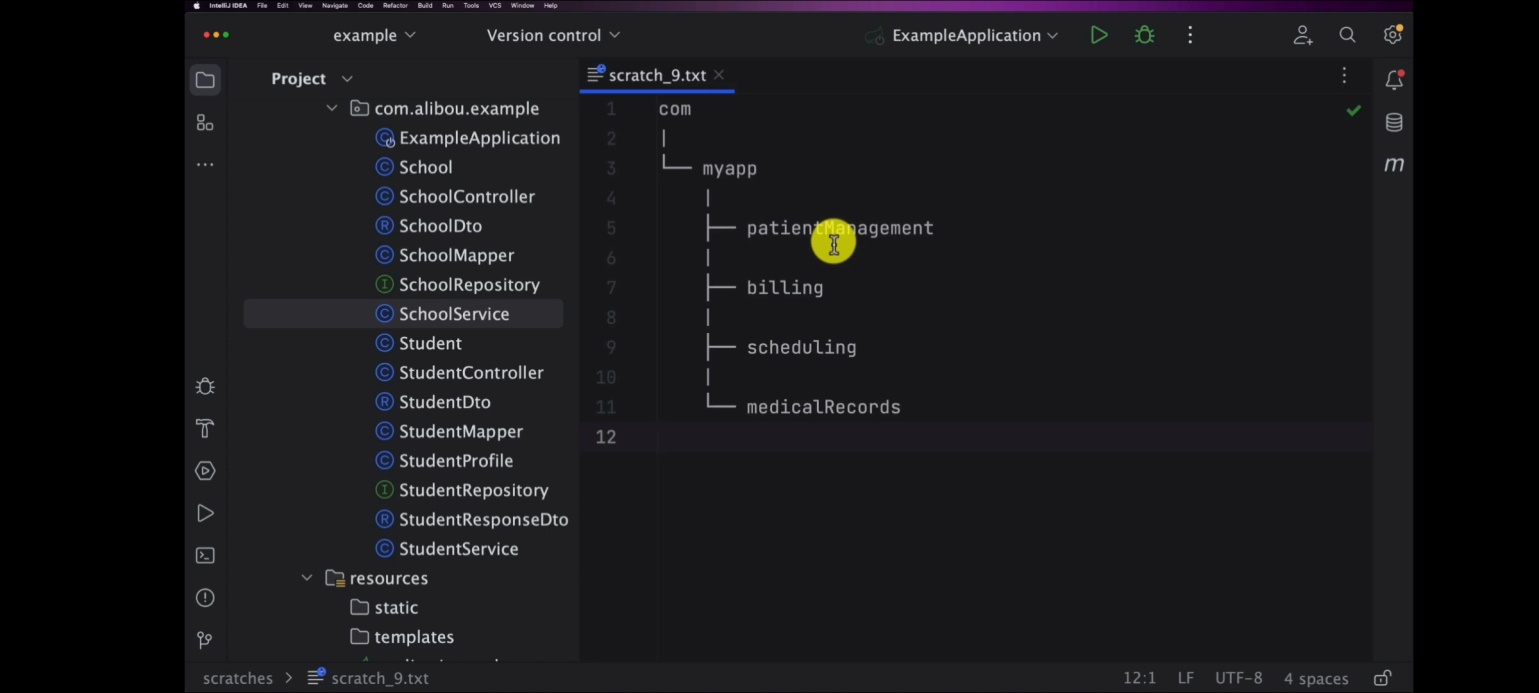
it is easy to locate the code related to specific feature.



1. for second approach is the layed approach so in this approach the code is organized around its architectural layers common layers might include controller service repository model/entity and util for example , this style is beneficial for when the focus is more on technical roads classes rather than than their business roles. however one downside is that change to a single feature could affect multiple packages.

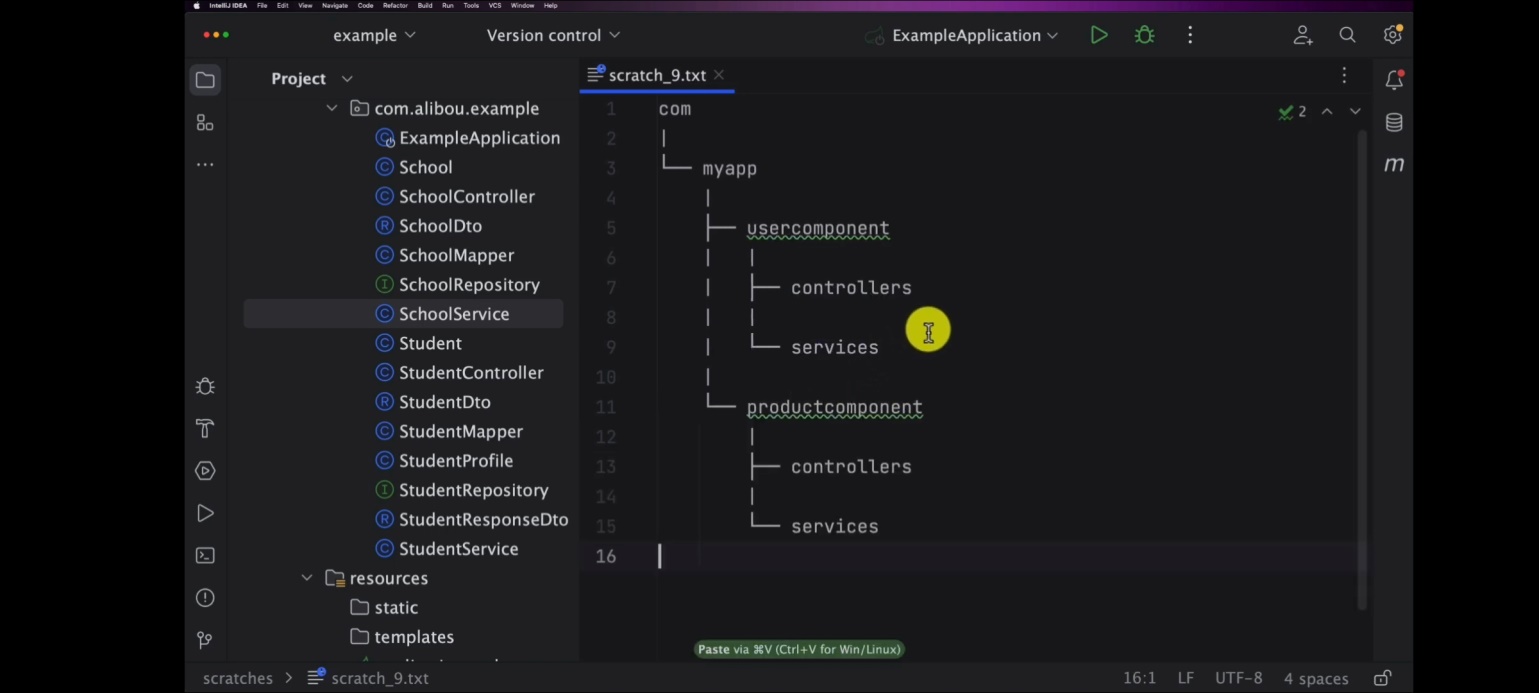


1. the third approach is, by domain so in the domain driven design or also called as DDD approach the focus is on the business domain package so packages are formed around different bounded context of subdomain within the large domain so for instance in a health care system for example here might be packages like patient management billing management scheduling and medical records



4. the last one is BU component, so in the more modularise approach you may have top level packages for each high level component or of your system within each of these packages you can further organize code by feature layer for example for example if we have user component we can have sub packages like controllers service etc

same for the product component we can have controller service model utils and so on



personally i prefer layer approach so here we can have packages for product order customer payment --

inside each of packages we will have all the related classes to the product so in this in our case we will have packages for school a package for student another package for student profile and so on we need common packages like validators or exception handling like a custom exception and so on

create a specific package for that so move on and organize our code together... school, student, StudentProfile etc

when we insert data some data will go null like email age and so on

so age is zero because it's of type int so by default the int is zero but we see also that we have email and last name they are null and this is not really appropriate

when it comes with the database and for that the data we want to persist here we can accept the null values for some fields but for other fields we can't accept null values

for example when we talk about email and student so the email can be also unique identifier important sensitive information for student so we need always to have the email and the last name for the student so this also introduces us to the next section of this course let me explain to data validation or spring validation

spring validation/ Data validation in rest API

================

the null values issues and how we can fix , The data validation is crucial in rest api for saveral resons.

1. Data integrity - data validation ensure that data being received and processed by API meets certain rules and restrictions this helps the maintain the consistency and reliability of your data for instance validating an email address format ensure that you won't have invalid email addresses in your database.

2. preventing attacks - validated data can expose your API to security vulnerability malicious users can send harmful input to your API in an attempt to perform SQL injection cross side scripting also known as xss and other types of attacks proper data validation can significantly reduce this risk

3. Error Prevention - error by validating data before it's processed you can prevent errors that might occurs during processing if you receive the number where you except the string it can cause errors in your code data validation helps detect these issues early.

4. User experience(UI) - validating input on service side and providing clearer message can greatly improve the user experience it lets user know what did wrong and how to correct it rather than leaving than guessing

5. performance - receive large amount of incorrect or poorly formatted data can affect your API performance, validation can serve as a first line of a defense against unnecessary processing such as data

6. business logic compliance - business rules determine the acceptable values or formats for certain data fields validation ensure that the data your API receives processes compiles with these business rules ... lets see in the action how we can implement this in spring application.

======================

Steps 1 : add the dependency - for validation and provides allready a starter for validating objects for in pom.xml

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-validation</artifactId>  
</dependency>

Steps 2 : validate the objects using the some annotations on the Dto layer and use the power of springboot starters to delegate almost everything

spring validation provide bunch of validation annotation

entry point of the object that we use to interact with the rest api this is on the dto layer as well in controller post and put operation.

package com.app.dto;  
import jakarta.validation.constraints.NotEmpty;  
  
public record StudentDto(  
@NotEmpty(message = "firstname should not be empty !!")  
String firstname,  
  
@NotEmpty(message = "lastname should not be empty !!")  
String lastname,

String email,  
 int age,  
 Integer schoolid  
) {  
}

steps 3: add @Valid in controller postmapping and putmapping before @RequestBody

@PostMapping("/students/create")  
public StudentResponseDto postMethod(@Valid @RequestBody StudentDto studentDto){  
 return studentService.postMethod(studentDto);  
}

@PutMapping("/student/update/{id}")  
public Student updateStudent(@Valid @PathVariable Integer id, @RequestBody Student student1){  
 return studentService.updateStudent(id,student1);  
}

add the fields with null and hit send you will get MethodArgumentNotValidException

.. failed for argument zero, field last name rejected value null exception raised 400 code

// handle the exception that is MethodArgumentNotValidException here in controller, we handle the exception or catch the exception and return proper response to end user

as we see here logs spring validation through an exception of type method argument not valid exception

@ExceptionHandler(MethodArgumentNotValidException.class)  
*//@ResponseStatus(HttpStatus.BAD\_REQUEST)*public ResponseEntity<?> handle\_method\_argument\_not\_validException(  
 MethodArgumentNotValidException e  
){  
 Map errors = new HashMap<String,String>();  
 e.getBindingResult().getFieldErrors().forEach(error -> {  
 String fieldName = error.getField();  
 String errorMessage = error.getDefaultMessage();  
 errors.put(fieldName, errorMessage);  
 });  
 *// return ResponseEntity.badRequest().body(errors);* return new ResponseEntity<>(errors, HttpStatus.*BAD\_REQUEST*);  
}

======= validation annotations ==========

* AssertFalse
* AssertTrue
* DecimalMax
* DecimalMin
* Digits
* Email
* Future
* FutureOrPresent
* Max
* Min
* Negative
* NegativeOrZero
* NotBlank
* NotEmpty
* NotNull
* Null
* Past
* PastOrPresent
* Pattern
* Positive
* PostiveOrZero
* Size

============================ \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

main important part of the **application is testing**

=========

about testing because why we need test because for example if i do a small change in one of the domain that we have what are my guarentees to say that this change

won't impact any anything else so software testing is crucial process in the realtime of software development designed to identify and rectify defects errors and

consistancies with a software application. it servers as a quality assurance mechanism ensuring that the software performs as intended and meets the requirement and

expectations of both developers and users through a systematic and structured approach.

software testing evaluates various aspect of the programs functionality usability security and perfomance.

Software testing in action so we will implements the three parts of testing so unit testing, integration testing, endtoend testing so

Importants of testing

=================== 6.22.39

1. Quality Assurance - so tests help ensure that your application functions as expected they identify bugs error and expected behaviour early in the development process allowing you to fix issues before they reach production.

2. Regression Testing - as you make changes or add new features to your application tests act as a safety net to the catch regressions they ensure that existing functionality continues to work as intended even after code changes also.

3. Documentation - test serves as the documentation for your code base they provide example of how you code is used and demonstrate it's as expected behavior.Tthis is especially helpful for other developers who may join the project later

4. Code Maintainabilty - on so also a well written test encourage good coding practices & modular design.

5. Refactoring Confidence - when you want to refactor or optimize your code tests provide a safety net they allow you to make changes with the confidence that will you immediately know if something beaks. they oftern lead to the more modular loosely coupled code that is easier to maintain

6. collaboration - also test facilitate collaboration among developers. when multiple team members are working on the same code base tests ensure that changes made by one developer.

don't negatively impact the work of others also.

7. Continious Integration/Continues Deployement(CI/CD) - also in CI/CD pipeline or continuous integration continues deployment pipeline tests are run automatically whenever new code pushed this helps catch issues early in the development process and prevents fauly code from reaching production.

8. Reduced Debugging time - also written test can take more time up front but it can be significantly reduce the time spent on debugging later in the development process. it is much easier to identify root cause of a failure when you have well defined tests as your appliction grows the complexity

9. Scalability - also increase the test provide a safety net that allows you to confidently make changes and add features without introducing unexpected issues.

10.Security - security and vulnerability tests help identify potentencial security weekness in your application this is crucial for protecting sensitive data and ensuring the overall security of your application

In Spring boot Application we can implement various types of test including unit testing integration testing even end-to-end test so each types of test focuses on specific

aspect of your application's functionality and provide a compressive safety net to ensure reliability and correctness.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

spring test - introduction

========================== 6.25.54 ========

Spring boot provides utilities and annotations for testing applications

Test supports is provided by two modules

1. Spring-Boot-test contains core items

2. spring-Boot-test autoconfigure supports auto-configuration for tests.

spring-Boot-starters-test "starter" imports

Spring-Boot Test modules

Junit,Assert, Hamcrest, and a numbers of useful libraries

Sprin test = introduction

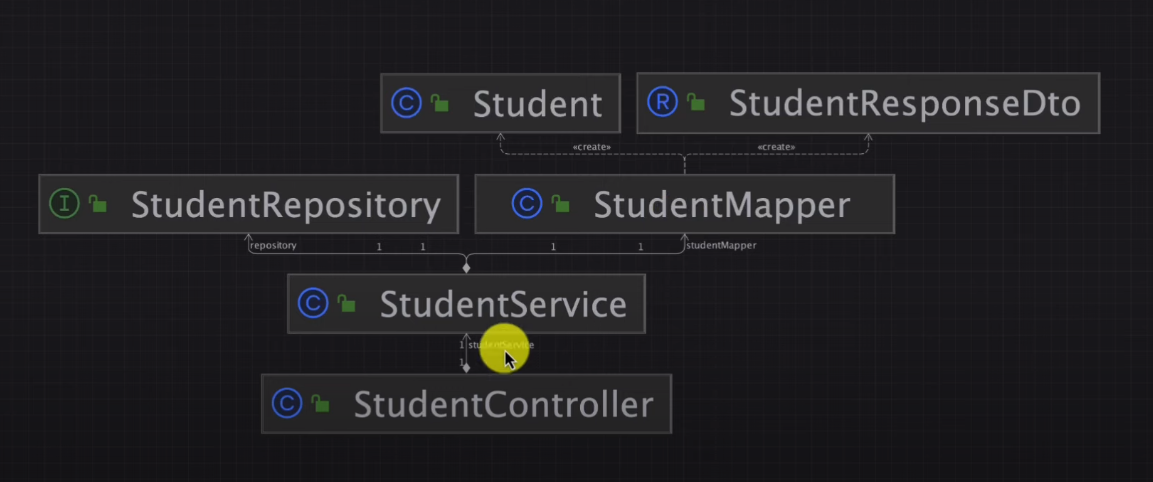
======================= 6. 26.36 ====================

A springBoot Application is a spring Application context.

Spring Boot provides @SpringBootTest annotation when you need springboot features during test.

Junit4 -> @RunWith(SpringRunner.class) to your test.

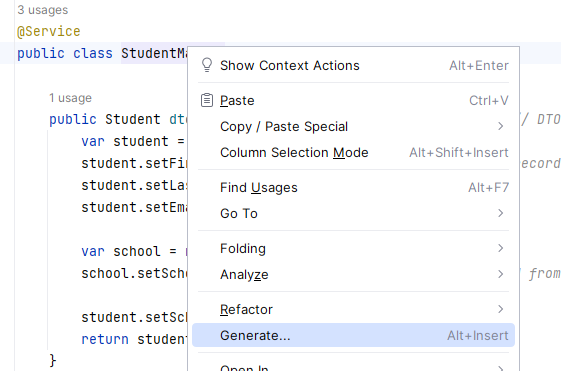
Junit5 -> no need to add equivalent @ExtendWith(SpringExtention) ---------------

* SpringBoot AutoConfiguration work well for application but sometime be a little too much tests
* It Often helps to load only the part of configuration that are required to a test a "slice" of your application.
* The springBoot autoconfiguration module include a number of annotations that can be used to automatically configure such "slices"
* Each slice restrict component scan to appropriate components and load a very restricted set of auto configuration classes

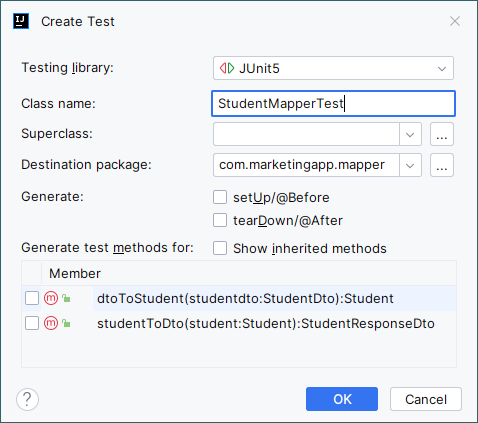
The main is studentController and Another is studentService it use StudentRepository and studentMapper and StudentMapper need Student model and StudentResponseDto

So I want to implement test for so what do you need to test which classes and which objects you need to implement tests for so here we see that

First test perform on StudentMapper on StudentMapper right click you will see this screen



When you will click generate you will see or use short ctrl + shift + t = you will get this page

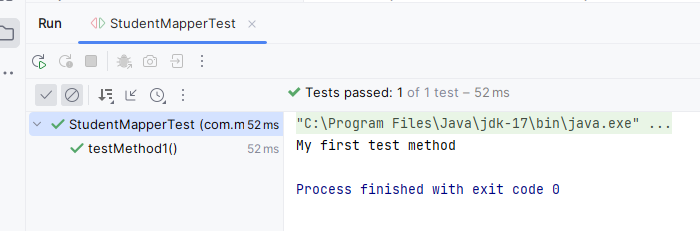


Here you can select the Testing library and class name and generate test method for testing

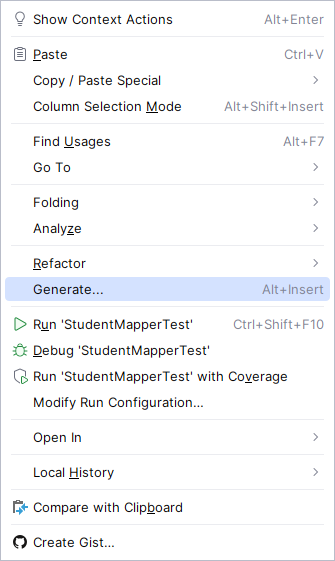
**Ctrl+shift+t =-🡪 move to testing class**

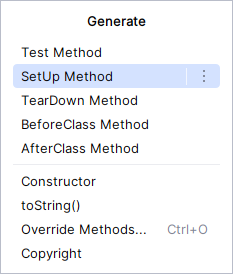
Lets generate a first test method to create a test method all you need to do is to

package com.marketingapp.mapper;  
  
import org.junit.jupiter.api.Test;  
  
import static org.junit.jupiter.api.Assertions.\*;  
  
class StudentMapperTest {  
  
 @Test  
 public void testMethod1(){  
 System.*out*.println("My first test method");  
 }  
}



We have a setup method or we can create a setup method so if you click on generate





We can generate it automatically or you can even write it your own so let’s generate it and see what it what is this method will be executed before each test that we have mainly use this setup in order to initialize few things or few parameters or anything that we want to initialize before running any test method that we have so we will test

package com.marketingapp.mapper;  
  
import org.junit.jupiter.api.BeforeEach;  
import org.junit.jupiter.api.Test;  
  
import static org.junit.jupiter.api.Assertions.\*;  
  
class StudentMapperTest {  
  
 @BeforeEach  
 void setUp() {  
 System.*out*.println("Inside the before each method");  
 }  
  
 @Test  
 public void testMethod1(){  
 System.*out*.println("My first test method");  
 }  
}

output log

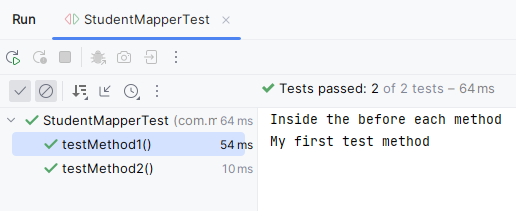
Inside the before each method

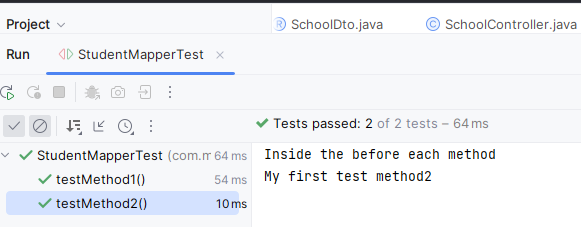
My first test method

Process finished with exit code 0

\*\*\*\*\*\*\*\*

package com.marketingapp.mapper;  
  
import org.junit.jupiter.api.BeforeEach;  
import org.junit.jupiter.api.Test;  
  
import static org.junit.jupiter.api.Assertions.\*;  
  
class StudentMapperTest {  
  
 @BeforeEach  
 void setUp() {  
 System.*out*.println("Inside the before each method");  
 }  
  
 @Test  
 public void testMethod1(){  
 System.*out*.println("My first test method");  
 }  
  
 @Test  
 public void testMethod2(){  
 System.*out*.println("My first test method2");  
 }  
  
  
  
}

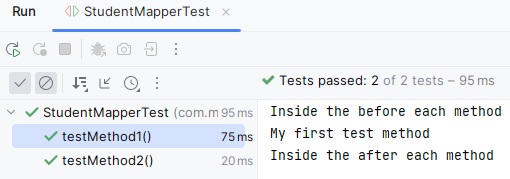


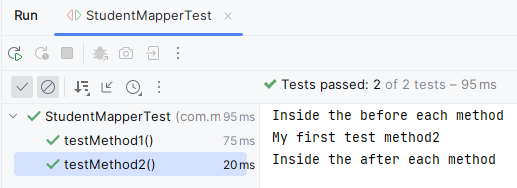


As like we have other methods and let’s go ahead and explore them one by one in order to understand the structure of a test class and what are the usages and the benefits of using these methods

Also we want to execute any script for example to reset the values of variable or classes or services or whatever we also have a method we call it tear down so the tear down is after execution of test method so let’s use the generate again and here we see that we have the tear down method perform after each method when we run the test class like closing the statement

package com.marketingapp.mapper;  
  
import org.junit.jupiter.api.AfterEach;  
import org.junit.jupiter.api.BeforeEach;  
import org.junit.jupiter.api.Test;  
  
import static org.junit.jupiter.api.Assertions.\*;  
  
class StudentMapperTest {  
  
 @BeforeEach  
 void setUp() {  
 System.*out*.println("Inside the before each method");  
 }  
  
 @Test  
 public void testMethod1(){  
 System.*out*.println("My first test method");  
 }  
  
 @AfterEach  
 void tearDown() {  
 System.*out*.println("Inside the after each method");  
 }  
  
 @Test  
 public void testMethod2(){  
 System.*out*.println("My first test method2");  
 }  
}





Running your test against database as you can just initialize and insert some data in before each, after each you can delete then or re-update the data in order to be able to use them in the next method.

\*\*\*

What if I want to execute a setup or to initialize a few attributes and variables only before the execution of the whole class so I don’t want to setup method before each but I just want it to be executedonly once so in this case we have also another so it is **beforeclass method**

class StudentMapperTest {  
  
 @BeforeAll  
 static void beforeAll() {  
 System.*out*.println("Inside the before all class each method");  
 }

@AfterAll  
 static void afterAll() {  
 System.*out*.println("Inside the after all class each method");  
 }  
}

Output;

Inside the before all class each method

Inside the before each method

My first test method2

Inside the after each method

Inside the after all class each method

Process finished with exit code 0

\*\*\*\*\*\*\*\*\*

Let’s start implementing few test cases for our student mapper class - 6.43.41

In StudentMapper test class first declare the service that you want to test so here we want to test our student mpper so let’s create an object of type mapper

The first use case or the test case that we want to implement is that we want to make sure that these two student mapper performs a correct mapping from student dto to the student object

Let’s create our first method here, make it a test method we need to add always the test annotaion

package com.marketingapp.mapper;  
  
import com.marketingapp.entity.Student;  
import com.marketingapp.payload.StudentDto;  
import org.junit.jupiter.api.\*;  
  
import static org.junit.jupiter.api.Assertions.\*;  
  
class StudentMapperTest {  
  
 StudentMapper mapper = new StudentMapper();  
  
 @BeforeEach  
 void setUp() {  
 mapper = new StudentMapper();  
 }  
  
 @Test  
 public void shouldMapStudentDtoToStudent(){  
 StudentDto dto = new StudentDto(  
 "John",  
 "sham",  
 "john@gmail.com",  
 1,  
 2  
 );  
 Student student = mapper.dtoToStudent(dto);  
 *//Assertions.assertEquals(dto.firstname(),student.getFirstname());  
 assertEquals*(dto.firstname(),student.getFirstname());  
 *assertEquals*(dto.lastname(),student.getLastname());  
 *assertEquals*(dto.email(),student.getEmail());  
 *// here we want to make sure school not be null  
 assertNotNull*(student.getSchool());  
 *assertEquals*(dto.schoolid(),student.getSchool().getSchoolid());  
  
 }  
}

Test executed successfully

Now if you are not add student in the service layer

Like I will comment that

public Student dtoToStudent(StudentDto studentdto){ *// DTO method* var student = new Student();  
 student.setFirstname(studentdto.firstname()); *// used record here studentdto* student.setLastname(studentdto.lastname());  
 student.setEmail(studentdto.email());  
  
 var school = new School();  
 school.setSchoolid(studentdto.schoolid()); *// school id from school entity  
  
 //* student.setSchool(school); // now test will fail   
 return student;  
}



Otherwise You want to make upper case as firstname also not work

@Service  
public class StudentMapper {  
  
 public Student dtoToStudent(StudentDto studentdto){ *// DTO method* var student = new Student();  
 student.setFirstname(studentdto.firstname().toUpperCase()); *// used record here studentdto* student.setLastname(studentdto.lastname());  
 student.setEmail(studentdto.email());  
  
 var school = new School();  
 school.setSchoolid(studentdto.schoolid()); *// school id from school entity* student.setSchool(school);  
 return student;  
 }

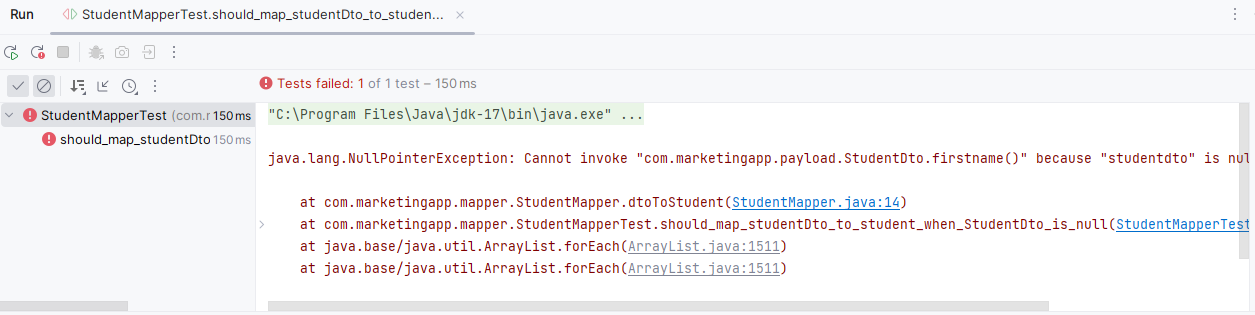


*// task is about entity to dto*

@Test  
public void shouldMapStudentToStudentResponseDto(){  
 Student student = new Student();  
 *//when* StudentResponseDto studentResponseDto = mapper.studentToDto(student);  
  
 *//then  
 assertEquals*(studentResponseDto.firstname(),student.getFirstname());  
 *assertEquals*(studentResponseDto.lastname(),student.getLastname());  
 *assertEquals*(studentResponseDto.email(),student.getEmail());  
}

//test case executed

@Test  
public void should\_map\_studentDto\_to\_student\_when\_StudentDto\_is\_null(){  
 Student student = mapper.dtoToStudent(null);  
  
 *assertEquals*("",student.getFirstname());  
 *assertEquals*("",student.getLastname());  
 *assertEquals*("",student.getEmail());  
}



Here exception is not handled so we will handle this exception here

Here when we will get exception we will throw exception or print message in the StudentMapper classs

@Service  
public class StudentMapper {  
  
 public Student dtoToStudent(StudentDto studentdto){ *// DTO method* if (studentdto == null){  
 throw new NullPointerException("The Student Dto is null");  
 }  
  
 var student = new Student();  
 student.setFirstname(studentdto.firstname());

*// used record here studentdto* student.setLastname(studentdto.lastname());  
 student.setEmail(studentdto.email());  
  
 var school = new School();  
 school.setSchoolid(studentdto.schoolid()); *// school id from school entity* student.setSchool(school);  
 return student;  
 }

Now let’s execute that test case

@Test  
public void should\_throw\_Nullpointer\_Exception\_map\_studentDto\_to\_student\_when\_StudentDto\_is\_null(){  
 *// check it will not or not  
 assertThrows*(NullPointerException.class,()->mapper.dtoToStudent(null));  
}

Now the output is

Test case executed successfully

Now I want to check the exception message

@Test  
public void should\_throw\_Nullpointer\_Exception\_map\_studentDto\_to\_student\_when\_StudentDto\_is\_null(){  
 *// check it will not or not* var exp= *assertThrows*(NullPointerException.class,()->mapper.dtoToStudent(null));  
 *assertEquals*("The student be null",exp.getMessage());  
}



The studentMapper test is complete now we will test **StudentService**

How can we test and how can we tell spring or this Student Service that we have a student repository and this one is communicate with the database and what happen in case if we don’t have real database for testing it does not necessarily need to have a database for testing that’s why we will talk about test **isolation** – means that we want to run and write tests for this Student Service in isolation its dependency

In a different way I want to execute and write test for this student service that I don’t necessarily need to have a real instance and real object of my student repository the same for student mapper so what do I need to do I need to **mock** these two objects(Student Mapper and Student Repository) so to do that we have there is a framework called **mokito**. Mokito is a popular java framework that helps to create mock objects stab behaviors when writing unit tests so its specially when you want to isolate a specific component of your code and test it in isolation from its dependency

Now we need to isolate and we need to mock.

We need to mock objects you need to mark the student repository and the student mapper to be able to run the student service test is an isolated way

Now first we go to student service and create student service test class like how we created student mapper

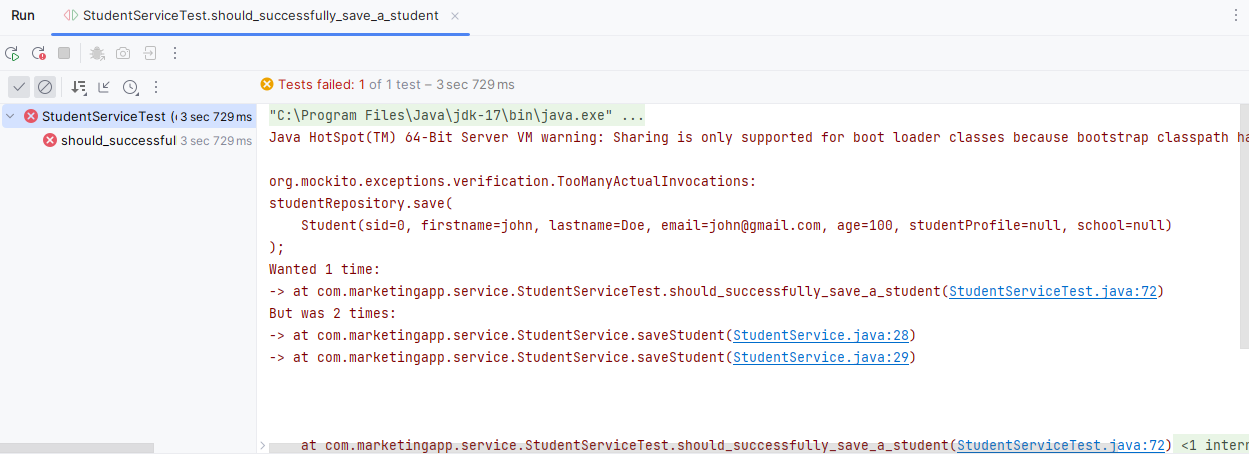
package com.marketingapp.service;  
  
import com.marketingapp.entity.Student;  
import com.marketingapp.mapper.StudentMapper;  
import com.marketingapp.payload.StudentDto;  
import com.marketingapp.payload.StudentResponseDto;  
import com.marketingapp.repositoty.StudentRepository;  
import org.junit.jupiter.api.BeforeEach;  
import org.junit.jupiter.api.Test;  
import org.mockito.InjectMocks;  
import org.mockito.Mock;  
import org.mockito.Mockito;  
import org.mockito.MockitoAnnotations;  
  
import static org.junit.jupiter.api.Assertions.\*;  
  
class StudentServiceTest {  
  
 *//which service we want to test* @InjectMocks *//org.mocks* private StudentService studentService;  
  
 *//declare the dependency* @Mock *//org.mock* private StudentRepository studentRepository;  
  
 @Mock  
 private StudentMapper studentMapper;  
  
 *//now we have to tell spring start mock for this class* @BeforeEach  
 void setUp() {  
 MockitoAnnotations.*openMocks*(this); *// here mock start* }  
*// set up the mock here  
  
 // now analyse the save method and perform* @Test  
 public void should\_successfully\_save\_a\_student(){  
 StudentDto dto = new StudentDto(  
 "john",  
 "Doe",  
 "john@gmail.com",  
 100,  
 1  
 );  
  
 *// it will come from mapping* Student student = new Student("john","Doe","john@gmail.com",100);  
  
 Student savedstudent = new Student("john","Doe","john@gmail.com",100);  
  
 savedstudent.setSid(1);  
  
 *//mock the calls, call every call that uses the dependency* Mockito.*when*(studentMapper.dtoToStudent(dto))  
 .thenReturn(student);  
 Mockito.*when*(studentRepository.save(student))  
 .thenReturn(savedstudent);  
 Mockito.*when*(studentMapper.studentToDto(savedstudent))  
 .thenReturn(new StudentResponseDto("john","Doe","john@gmail.com"));  
  
 *//when* StudentResponseDto responseDto = studentService.saveStudent(dto);  
  
 *//then  
 assertEquals*(dto.firstname(),responseDto.firstname());  
 *assertEquals*(dto.lastname(),responseDto.lastname());  
 *assertEquals*(dto.email(),responseDto.email());  
  
 }  
  
}

suppose user want to save record 2 time not possible or developer written code repository.save 2 times also not work

public StudentResponseDto saveStudent(  
 StudentDto studentDto  
){  
 Student student = studentMapper.dtoToStudent(studentDto); *// dto to student convert* Student save = this.studentRepository.save(student); *// saved* studentRepository.save(student); *// by mistakely add so for unit test* return studentMapper.studentToDto(save); *// entity to dto*}

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

package com.marketingapp.service;  
  
import com.marketingapp.entity.Student;  
import com.marketingapp.mapper.StudentMapper;  
import com.marketingapp.payload.StudentDto;  
import com.marketingapp.payload.StudentResponseDto;  
import com.marketingapp.repositoty.StudentRepository;  
import org.junit.jupiter.api.BeforeEach;  
import org.junit.jupiter.api.Test;  
import org.mockito.InjectMocks;  
import org.mockito.Mock;  
import org.mockito.Mockito;  
import org.mockito.MockitoAnnotations;  
  
import static org.junit.jupiter.api.Assertions.\*;  
  
class StudentServiceTest {  
  
 *//which service we want to test* @InjectMocks *//org.mocks* private StudentService studentService;  
  
 *//declare the dependency* @Mock *//org.mock* private StudentRepository studentRepository;  
  
 @Mock  
 private StudentMapper studentMapper;  
  
 *//now we have to tell spring start mock for this class* @BeforeEach  
 void setUp() {  
 MockitoAnnotations.*openMocks*(this); *// here mock start* }  
*// set up the mock here  
  
 // now analyse the save method and perform* @Test  
 public void should\_successfully\_save\_a\_student(){  
 StudentDto dto = new StudentDto(  
 "john",  
 "Doe",  
 "john@gmail.com",  
 100,  
 1  
 );  
  
 *// it will come from mapping* Student student = new Student("john","Doe","john@gmail.com",100);  
  
 Student savedstudent = new Student("john","Doe","john@gmail.com",100);  
  
 savedstudent.setSid(1);  
  
 *//mock the calls, call every call that uses the dependency* Mockito.*when*(studentMapper.dtoToStudent(dto))  
 .thenReturn(student);  
 Mockito.*when*(studentRepository.save(student))  
 .thenReturn(savedstudent);  
 Mockito.*when*(studentMapper.studentToDto(savedstudent))  
 .thenReturn(new StudentResponseDto("john","Doe","john@gmail.com"));  
  
 *//when* StudentResponseDto responseDto = studentService.saveStudent(dto);  
  
 *//then  
 assertEquals*(dto.firstname(),responseDto.firstname());  
 *assertEquals*(dto.lastname(),responseDto.lastname());  
 *assertEquals*(dto.email(),responseDto.email());  
  
 Mockito.*verify*(studentMapper,Mockito.*times*(1)).dtoToStudent(dto);*// dto to entity* Mockito.*verify*(studentRepository,Mockito.*times*(1)).save(student);*//saved* Mockito.*verify*(studentMapper,Mockito.*times*(1)).studentToDto(savedstudent);*//entity to repository* }  
  
}



Now test for findAll

@Test  
public void should\_all\_student(){  
 List<Student> students = new ArrayList<>();  
 Student student = new Student("john","Doe","john@gmail.com",100);  
 *//mock the calls* Mockito.*when*(studentRepository.findAll()).thenReturn(students);  
 Mockito.*when*(studentMapper.studentToDto(*any*(Student.class)))  
 .thenReturn(new StudentResponseDto(  
 "john","Doe","john@gmail.com")  
 );  
 *//when* List<StudentResponseDto> responseDtos = studentService.getStudentall();  
  
 *//Then  
 assertEquals*(students.size(),responseDtos.size());  
 *//Mockito.verify(studentRepository,Mockito.times(1));*}

Byid

@Test  
public void should\_return\_student\_by\_id(){  
 Integer studentId =1;  
 Student student = new Student("john","Doe","john@gmail.com",100);  
  
 *//mock the calls* Mockito.*when*(studentRepository.findById(studentId))  
 .thenReturn(Optional.*of*(student));  
 Mockito.*when*(studentMapper.studentToDto(*any*(Student.class)))  
 .thenReturn(new StudentResponseDto(  
 "john","Doe","john@gmail.com")  
 );  
 *//when* StudentResponseDto dto = studentService.getStudent(studentId);  
  
 *//then  
 assertEquals*(dto.firstname(),dto.firstname());  
 *assertEquals*(dto.lastname(),dto.lastname());  
 *assertEquals*(dto.email(),dto.email());  
   
}

Findbyname

@Test  
public void should\_by\_name(){  
 String studentName = "john";  
 List<Student> students = new ArrayList<>();  
 Student student = new Student(  
 "john",  
 "Doe",  
 "john@gmail.com",  
 100);  
 *//mock the calls* Mockito.*when*(studentRepository.findAllByfirstnameContaining(studentName))  
 .thenReturn(students);  
 Mockito.*when*(studentMapper.studentToDto(*any*(Student.class)))  
 .thenReturn(new StudentResponseDto(  
 "john",  
 "Doe",  
 "john@gmail.com")  
 );  
 *//when* var responseDto = studentService.getStudentwithname(studentName);  
  
 *//Then  
  
 assertEquals*(students.size(),responseDto.size());  
 *verify*(studentRepository,Mockito.*times*(1)).findAllByfirstnameContaining(studentName);  
}