Rules:

Entties should be in lowercase.

Ex:Long name;

Long projectcode

While declaring query method First letter should be capital but parameter must be same.

Ex:

Public project getByProjectcode(Long projectcode);

Observe the differences in project code usage by changing lower and uppercases).

SPRING FRAMEWORK:

Aspect oriented programming:  
Aspect-Oriented Programming (AOP) is a programming paradigm that helps developers separate cross-cutting concerns from the main business logic of their application.

**Cross-Cutting Concerns:** These are tasks or features that can affect multiple parts of your application, such as logging, security, transaction management, or error handling.

**Separation of Concerns:** AOP allows you to modularize these cross-cutting concerns into separate components called aspects.

**Aspect:** An aspect is a modular unit of cross-cutting concerns that can be applied declaratively to your code. It encapsulates behaviors that can be applied across multiple classes or modules.

**Join Point:** Points in your program where certain actions defined in aspects can be applied. Examples include method executions, exception handling, or field accesses.

**Advice:** Actual implementation of the cross-cutting behavior defined in an aspect. It specifies what action should be taken and when it should be applied at specific join points.

**Weaving:** The process of linking aspects with other application components to create an executable application. This can be done at compile-time, load-time, or runtime.

Let's say you have a banking application. Logging every method call related to transactions can be a cross-cutting concern. Instead of adding logging code to every method that involves transactions, you can create an aspect for logging. This aspect will define advice (like logging messages) and specify join points (like methods related to transactions).

ACCESSING METHOD USING OBJECT:

1. Doctor d=new Doctor();

d.assist();

DEPENDENCY INJECTION USING BEANS

2.USING BEANS  
Doctor d=app.getBean(Doctor.class);

Declare this

ApplicationContext app=new ClassPathXmlApplicationContext("spring.xml");

In pom.xml make sure you have spring-context in place of spring aop or else application context will not work.

<dependencies>  
 <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-context</artifactId>  
 <version>6.1.4</version>  
 </dependency>  
</dependencies>

In spring.xml add the class in bean

<bean class="org.example.Doctor"></bean>

Org.example is package name

Doctor is class in org.example package.

Finally it gives output.

Doctor d=app.getBean(Doctor.class);

d.assist();

create a interface staff with method assist and try to access doctor with

staff s=app.getbean(Doctor.class)

getter setter injection:

private String Qualification;

public String getQualification() {  
 return Qualification;  
}  
  
public void setQualification(String qualification) {  
 Qualification = qualification;  
}

ADD this:

<bean class="org.example.Doctor">  
 <property name="Qualification" value="MBBS"></property>  
</bean>

Access using this:

System.*out*.println(d.getQualification());

We can get value of qualification.

CONSTRUCTOR INJECTION:  
private String Qualification;

Add the construction option same as getter and setter.

<bean class="org.example.Doctor">  
 <constructor-arg value="MBBS"></constructor-arg>  
</bean>

Now apply this with doctor object d,

System.*out*.println(d.getQualification());

This is getting qualification value using constructors.

3.Annotation based calling methods:

Step 1:

ApplicationContext app=new AnnotationConfigApplicationContext(Beanconfig.class);

Without spring.xml beans are created in Beanconfig.class.

It means we are getting from that class.

Step-2:

Create bean config and add @configuration and @component-scan to it.

@Configuration  
@ComponentScan(basepackages=”org.example”)  
public class Beanconfig {

@component-scan gathers all components and creates beans in this Beansconfig class.earlier beans are in spring.xml file

Step-3:

Add @component to all classes so that component-scan knows how many beans to be created.

@Component  
public class Doctor implements Staff{

Step-4:

Access creating object using beans to get output

Doctor d=app.getBean(Doctor.class);

Finally will get the output.

Way-2(without Beanconfigclass):

<context:component-scan base-package="org.example"></context:component-scan>

Add this in spring.xml,it will create beans for all the classes which are in base package=org.example.no need to create bean id and class by our own.

Error faced:<context:component-scan> not found if below context files are not included in <beans></beans> in spring.xml

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/context

<http://www.springframework.org/schema/context/spring-context-3.0.xsd>">

with scopes:

Getters setters and Tostring() in dependency Injection:

@Component  
@Scope(scopeName = "prototype")  
public class Doctor implements Staff{  
 private String qualification;  
 private String qualification1="CA";  
 public void assist() {  
 System.*out*.println("doctor is assisting");  
 }  
  
 @Override  
 public String toString() {  
 return "Doctor{" +  
 "qualification='" + qualification + '\'' +  
 ", qualification1='" + qualification1 + '\'' +  
 '}';  
 }

Output :when we call this in main function,

d.setQualification("MBBS");  
System.*out*.println(d);  
System.*out*.println(d1);

this is output:

Doctor{qualification='MBBS', qualification1='CA'}

Doctor{qualification='null', qualification1='CA'}

(Second time is null when new d1 object created for same class because scope is prototype)

Getters setters:these helps in providing values from other classes

Tostring():to show variable when object is called tostring() must be added in that class.

\*\*\*\*\*

public class Doctor implements Staff,BeanNameAware

beannameaware ensures remaining methods are executed only after this

to use post construct add javaannatations in artifact and groupid in xml file.

It gives output after the beannameaware class.

Aspect oriented programming:

Dealing with cross cutting concerns is called aspect oriented programming.

Cross cutting concerns-logging,authorization.

These are fields which you want to use before some methods but defined saperately.

1.Create an asepect you want to use:

@Aspect  
@Component  
public class Login {  
 @Before("execution(\* org.aop.Shoppingcart.checkout())")  
 public void logger(){

1.@Aspect-it is a cross cutting one

2.@component-it is considers as bean in beanconfig.

3.@before -it tells before which class method it should execute.

Run this so that this method executed with its body before the class.

POINTCUTS:

@Aspect  
@Component  
public class Authenticationaspect {  
 @Pointcut("within(org.aop..\*)")  
 public void authenticatingpointcut(){  
  
 }  
@Pointcut("within(org.aop..\*)")  
 public void autherizingpointcut(){  
  
 }  
 @Before("authenticatingpointcut() && autherizingpointcut()")  
 public void authenticate(){  
 System.out.println("authenticate point cut request");  
 }  
  
}

CREATING SIMPLE API:

Open spring initializer

1.Add spring web

2.h2 database

Download and open in intellijid.

Later ,

1.Create controller.

2.create class hello controller.

In that write this function,

package com.example.mvcdatabase.controller;  
  
  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.RequestMethod;  
import org.springframework.web.bind.annotation.RestController;  
  
@RestController  
public class Hellocontroller {  
 @RequestMapping(value="/", method= RequestMethod.*GET*)  
 public String helloworld(){  
 return "Hello world from 2.0";  
 }  
}

@Restcontroller-

When you use @RestController, all methods in that class are implicitly annotated with @ResponseBody, which means that the return value of each method is automatically serialized into JSON or XML and sent in the HTTP response body.

@Restmapping-

@RequestMapping is a more general-purpose annotation used to map web requests to specific handler methods. It can be used on both classes and methods. It allows you to specify HTTP method types (GET, POST, PUT, DELETE, etc.), request paths, and parameters.

**Step-by-Step Process**

**1. Application Startup**

* **Spring Boot Initialization**: When you start your Spring Boot application, it initializes the Spring Application Context. This involves setting up the infrastructure needed to manage beans and handle incoming requests.
* **Component Scanning**: During initialization, Spring Boot scans the classpath for components annotated with Spring’s stereotype annotations (@Component, @Service, @RestController, etc.). This scanning is typically configured through the @SpringBootApplication annotation, which includes component scanning by default.

**2. Bean Creation and Registration**

* **Class Detection**: Spring Boot detects the Hellocontroller class because it is annotated with @RestController.
* **Bean Instantiation**: Spring Boot creates an instance of Hellocontroller. This is done automatically by the Spring container (Application Context) without requiring manual instantiation.
* **Bean Registration**: The newly created instance of Hellocontroller is registered as a bean in the Spring Application Context. This registration makes the controller available for handling web requests.

**3. Request Handling**

* **DispatcherServlet**: Spring Boot uses the DispatcherServlet to handle incoming HTTP requests. The DispatcherServlet is a central component in Spring MVC that routes requests to the appropriate handlers.
* **Request Mapping**: When an HTTP request is made to the root URL (/), the DispatcherServlet consults the Application Context to find the appropriate handler. It does this by checking the URL mappings and finding a method that matches the request.
* **Method Invocation**: The DispatcherServlet identifies that the request should be handled by the Hellocontroller class because the URL mapping (/) matches the @RequestMapping annotation on the helloworld method.
* **Response Generation**: The helloworld method is invoked on the Hellocontroller instance. Since @RestController automatically implies @ResponseBody, the return value of the method ("Hello world from 2.0") is serialized into JSON (or plain text if the default is not overridden) and sent as the HTTP response.

**Visual Representation**

Here’s a simplified visual representation of the process:

1. **Application Start-Up**

csharp

Copy code

[Spring Boot Application Start-Up]

|

V

[Component Scanning]

|

V

Detects `Hellocontroller` -> Creates and Registers Bean

1. **Request Handling**

javascript

Copy code

[Incoming HTTP GET Request to `/`]

|

V

[DispatcherServlet]

|

V

Finds `Hellocontroller` Bean1

|

V

Invokes `helloworld()` Method

|

V

Returns "Hello world from 2.0" as Response

Change setting in intellij:

1.go to advanced settings.click on awake running.

2.go to settings build,compiler,select build automatically.

Connecting to h2 database:

1.In pom.xml add springdata-jpa

2. h2-database dependencies.

In application.properties add,

spring.h2.console.enabled=true  
spring.datasource.url=jdbc:h2:mem:dcbapp  
spring.datasource.driverClassName=org.h2.Driver  
spring.datasource.username=sa  
spring.datasource.password=password  
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

after this click on reload from dependencies,

so all will be up to updated in maven.

### What Happens When You Initialize ApplicationContext?

1. **Bean Creation and Wiring:**
   * Spring scans the classes in the specified package(s) for annotations like @Component, @Service, @Repository, and @Controller, creating instances of these classes as beans.
   * Dependencies marked with @Autowired or other injection annotations are automatically wired.
2. **Configuration Application:**
   * Any classes annotated with @Configuration are processed, and the beans defined in these classes are created and managed by Spring.
3. **Application Lifecycle Management:**
   * The ApplicationContext manages the complete lifecycle of beans, including their creation, initialization, and destruction.

### Summary

* **Why Needed**: The initialization of ApplicationContext is necessary to start the Spring container, which manages your beans and configurations.
* **Role of Annotations**: While @Component and @Configuration annotations define what should be managed by Spring, the ApplicationContext is the mechanism that actually manages and provides these components.

Without initializing ApplicationContext, Spring would not know to start scanning, creating, and managing beans based on the annotations you've provided.

Flow needed to create a spring-database:

1.controller-package.

\*Create Hellocontroller class in controller package it is used to call post,get,delete mapping.

Controller is needed to call rest api using @restcontroller which has services and components.

\*\*\*\*For @postmaping and @put mapping---@jsonbody is must because we need our json data to be converted to department.\*\*\*(if you don’t it shows error I got it in @put request while passing id as path variable,but@requestbody not given for department).

\*\*you don’t ned that in @get,@delete because it was already in department form so department is returned.\*\*\*

Annatation needed:

@Restcontroller

@Autowired  
@pathvariable-to make id as part in parameter(ex:

@GetMapping("/department/{id}")

)

@get/post/delete mappings

2.entity-package

\*Create department class and create structure of database

In this structure of database is defined.

Annatation needed:@Entity

@id

Pass(department,Long)

3.Repository-package  
\*create a repositorydepartment interface and extend to jpa-repository.

here we have all save,findall methods we use in serviceimplementation.

Annatation needed:

(Here declare departmentrepository variable and use in function to return needed value because department repository only contains all needed methods here we can use it without extending or implementing use it by declaring)

@Repository

4.service package

Create service interface

Create service implementation class extend to service.

Annatation needed:

@service

@Autowired

It is important to write all our business logics.

\*\*when void method created in controller it shows problem only in interface after declaring .

it does not show error in service implementation because it returning void we should write by own.

Practical flow of MVC:

Practical example flow:

Entity-department:

Creating database table:

@Id  
@GeneratedValue(strategy = GenerationType.*AUTO*)  
private Long departmentid;  
private String departmentname;  
private String departmentaddress;  
private String departmentcode;

1.Add getter setters

2.Add constructor

3.Add null constructor

4.Add to string methods

@id-tells is primary key.

@generatedvalue-we can give values

2.repository-departmentrepository

@Repository  
public interface Departmentrepository extends JpaRepository<Department,Long> {

}

\*The @Repository annotation marks a class as a Data Access Object (DAO) that interacts with the database or other persistence mechanisms.

\*Here departmentrepository is extending all the methods from jpa repository.

3.service package-

1.interface service

package com.example.mvcdatabase.service;  
  
import com.example.mvcdatabase.entity.Department;  
  
import java.util.List;  
  
public interface Departmentservice {  
  
  
  
  
 Department savedepartment(Department department);  
  
  
 List<Department> getdata();  
}

\*All the functions we need are stored here because we can make according to requirements.when functions are in different classes and we need only some functions from many classes we use interface.

2.class servicedepartment implements service

package com.example.mvcdatabase.service;  
  
import com.example.mvcdatabase.entity.Department;  
import com.example.mvcdatabase.repository.Departmentrepository;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;  
  
import java.util.List;  
  
@Service  
public class Departmentserviceimplementation implements Departmentservice{  
 @Autowired  
 private Departmentrepository departmentrepository;  
 @Override  
 public Department savedepartment(Department department) {  
 return departmentrepository.save(department);  
 }  
  
 @Override  
 public List<Department> getdata() {  
 return departmentrepository.findAll();  
 }

@service/component-it is like component it should be given to concrete classes.(interfaces extended by classes cannot be applied to interfaces).

@autowired-

Example –

@Service

public class MyService {

@Autowired private MyComponent myComponent;

public void performAction()

{

myComponent.doSomething();

}

}

We can make a class as object with autowiring object  
and call that method in class.

ApplicationContext app=new AnnotationConfigApplicationContext(Beanconfig.class);

@autowire

@Autowired private MyComponent myComponent;

Here app is nothing but Mycomponent created because of autowiring.

@override-

@Notblank-

we can send message if one of entry in depname,code or id are missing.

Loggers:

These are helpful to show messages in the compiler.

Lombok:

@Entity  
@Data  
@AllArgsConstructor  
@NoArgsConstructor

Class department{  
}

Exception handling:

Used in getbyid to display if any given id does not exists to display messages.

@GetMapping("/department/{id}")  
public Department getbyid(@PathVariable("id") Long departmentid) throws Deparmentnotfound {  
 return departmentservice.getid(departmentid);  
}

@Override  
public Department getid(Long departmentid) throws Deparmentnotfound {  
  
 Optional<Department> department = departmentrepository.findById(departmentid);  
 if(!department.isPresent()){  
 throw new Deparmentnotfound("Department not found");  
  
 }  
 return department.get();  
}

Step-11

Application properties(to send gmail)

\*add a dependency in pom.xml

spring.mail.host=smtp.gmail.com  
spring.mail.port=587  
spring.mail.username=sandeepsunkara0007@gmail.com  
spring.mail.password=ngyklxnqvdisjepi  
spring.mail.properties.mail.smtp.auth=true  
spring.mail.properties.mail.smtp.starttls.enable=true  
spring.mail.properties.mail.debug=true

sending gmail using postmapping api :

Below we are sending parameters to,subject,message

Ere needed api to send mailis

<http://localhost:8082/department/to(mail)/subject/text> we want to send.

@PostMapping("/email")

@requestparams tells that asking us to accept that as parameters in function.  
public String sendemail(@RequestParam String to,  
 @RequestParam String subject,  
 @RequestParam String message){  
 departmentservice.sendemail(to,subject,message);  
 return "Email sent successfully!";  
  
}

Next step in service:

**JavaMailSender**: This is an interface provided by Spring that simplifies sending emails by using the JavaMail API. It handles the configuration of an SMTP server and supports composing and sending MIME-compliant email messages.

**SimpleMailMessage**: This is a class provided by the Spring Framework for constructing simple, plain text emails without attachments or rich content (i.e., no HTML or inline images). It’s a lightweight email object intended for basic email use cases.

With this we can do.

SimpleMailMessage emailMessage = new SimpleMailMessage();

emailMessage.setTo("recipient@example.com"); // Set recipient

emailMessage.setSubject("Subject of the email"); // Set subject

emailMessage.setText("This is the body of the email"); // Set email body

emailMessage.setFrom("sender@example.com"); // Set sender address

private JavaMailSender mailSender;  
@Override  
public void sendemail(String to, String subject, String message) {  
 SimpleMailMessage emailMessage = new SimpleMailMessage();  
 emailMessage.setTo(to);  
 emailMessage.setSubject(subject);  
 emailMessage.setText(message); // Use the 'message' parameter here  
 mailSender.send(emailMessage);  
 /\*message.setTo(to);  
 message.setSubject(subject);  
 message.setText(text);  
 emailsender.send(message1);\*/  
}

(adding email it in middle of a function):

sending a mail when data is added using post mapping.

public Department savedepartment(Department department) {  
 String to = "sandeepsunkara0007@gmail.com";  
 String subject = "New Department Added: " ;  
 String message = "A new department has been added with ID: ";  
  
 // Call the send email function  
 sendemail(to, subject, message);  
 return departmentrepository.save(department);  
}

step-(service layer testing).

@mockbean.

Mockito.  
package com.example.mvcdatabase.service;  
  
import com.example.mvcdatabase.entity.Department;  
import com.example.mvcdatabase.repository.Departmentrepository;  
import jakarta.persistence.Id;  
import org.junit.jupiter.api.BeforeEach;  
import org.junit.jupiter.api.Test;  
import org.mockito.Mockito;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.boot.test.context.SpringBootTest;  
import org.springframework.boot.test.mock.mockito.MockBean;  
  
import java.util.Collections;  
import java.util.List;  
  
import static org.junit.jupiter.api.Assertions.\*;  
@SpringBootTest  
class DepartmentserviceTest {  
 @Autowired  
 private Departmentservice departmentservice;  
  
 @MockBean  
 private Departmentrepository departmentrepository;  
 @BeforeEach  
 void setUp() {  
 Department department = Department.*builder*().departmentname("blr").Departmentcode("1").Departmentaddress("blr").build();  
 List<Department> departmentList = Collections.*singletonList*(department);  
 Mockito.*when*(departmentrepository.findByDepartmentname("blr")).thenReturn(departmentList);  
 }  
  
 @Test  
 public void printwhendepartnameisvalid(){  
 String departmentname="blr";  
 List<Department> found=departmentservice.getdatabyid(departmentname);  
 Department a=found.get(0);  
 *assertEquals*(departmentname,a.getDepartmentname());  
  
 }  
}

Repository layer testing:

Testentitymanager:

Persist:

package com.example.mvcdatabase.repository;  
  
import com.example.mvcdatabase.entity.Department;  
import org.junit.jupiter.api.BeforeEach;  
import org.junit.jupiter.api.Test;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;  
import org.springframework.boot.test.autoconfigure.orm.jpa.TestEntityManager;  
import org.springframework.stereotype.Repository;  
  
import java.util.List;  
  
import static org.junit.jupiter.api.Assertions.\*;  
@DataJpaTest  
class DepartmentrepositoryTest {  
 @Autowired  
 private Departmentrepository departmentrepository;  
 @Autowired  
 private TestEntityManager entityManager;  
  
 @BeforeEach  
 void setUp() {  
 Department department = Department.*builder*().departmentname("sandeep").Departmentcode("105").Departmentaddress("blr").build();  
 entityManager.persist(department);  
  
  
 }  
 @Test  
 public void whenfindbyname() {  
 List<Department> departments = departmentrepository.findByDepartmentname("sandeep");  
 Department department1 = departments.get(0);  
 *assertEquals*("sandeep", department1.getDepartmentname());  
 }  
 }

controller layer testing:

for saving department and fetchbyname.

package com.example.mvcdatabase.controller;  
  
import com.example.mvcdatabase.entity.Department;  
import com.example.mvcdatabase.service.Departmentservice;  
import org.junit.jupiter.api.BeforeEach;  
import org.junit.jupiter.api.Test;  
import org.mockito.Mockito;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;  
import org.springframework.boot.test.mock.mockito.MockBean;  
import org.springframework.http.MediaType;  
import org.springframework.test.web.servlet.MockMvc;  
import org.springframework.test.web.servlet.request.MockMvcRequestBuilders;  
import org.springframework.test.web.servlet.result.MockMvcResultMatchers;  
  
import static org.junit.jupiter.api.Assertions.\*;  
@WebMvcTest(controllers = Departmentcontroller.class)  
class DepartmentcontrollerTest {  
 @Autowired  
 private MockMvc mockMvc;  
 @MockBean  
 private Departmentservice departmentservice;  
 private Department department;  
  
 @BeforeEach  
 void setUp() {  
 Department department =Department.*builder*().departmentname("uma").Departmentaddress("blr").Departmentcode("505").build();  
  
 }  
  
 @Test  
 void savedepartment() throws Exception {  
 Department department =Department.*builder*().departmentname("uma").Departmentaddress("blr").Departmentcode("505").build();  
 Mockito.*when*(departmentservice.savedepartment(department)).thenReturn(department);  
 mockMvc.perform(MockMvcRequestBuilders.*post*("/department").contentType(MediaType.*APPLICATION\_JSON*).content("{\n" +  
 " \"departmentname\":\"pune\",\n" +  
 " \"departmentadress\":\"mumbai\",\n" +  
 " \"departmentcode\":\"mumbai\"\n" +  
 "}")).andExpect(MockMvcResultMatchers.*status*().isOk());  
 }  
  
 @Test  
 void getdatabyid() {  
  
 }  
}