**Day 11**

Prototype scope -beans cretae on req(lazy)

scope="singleton" means one instance for the entire application, while scope="prototype" means a new instance whenever requested.

* Servlet Lifecycle:
  + Java classes used for handling HTTP requests and responses.
  + Lifecycle Methodinit(), service(), and destroy().
  + Initialization: Servlet initialization init() method.
  + Destruction: Cleanup destroy() method.
  + Handling Requests: The service() handle each HTTP request.
* Spring Lifecycle:
  + Spring framework for building applications, including aspects beyond web handling, such as dependency injection, AOP, and more.
  + Lifecycle Methods: Spring beans have lifecycle methods such as init() and destroy()
  + Initialization:init()
  + Destruction: destroy()
  + Handling Requests: Spring doesn't handle HTTP requests directly; it focuses on managing beans and their dependencies. The handling of HTTP requests is typically done by controllers.

In summary, while Servlets are specifically designed for handling HTTP requests ,Spring provides framework for building enterprise applications, with a focus on managing beans and their lifecycles

In servlet we extend from httpservlet w/o that not crete

Testing become hard as need to load whloe architectur

Spring is standalone so teste easy

In servlet init comes from generic servlet we overidide init

In Spring, the init-method automatically calls after it finishes setting up or injecting dependencies into your object.

public class MyBean {

// Custom initialization method

public void anyName() throws Exception {

System.out.println("Custom initialization method called after dependency injection");

}

}

And in your Spring configuration:

<bean id="myBean" class="com.example.MyBean" init-method="anyName">

<!-- Other bean configurations and properties -->

</bean>

In Spring, you can define special methods, like anyName(), to run automatically when a bean is created or destroyed. It's a way to customize setup and cleanup tasks during the bean's lifecycle. Unlike servlets, where init and destroy are predefined, in Spring, you can name these methods as you like, offering more flexibility.

Why use spring in project

Bean factor =super i/f

Package:

* org.springframework.beans.factory

ApplicationContext Interface:

* Package: org.springframework.context
* Description: An extension of the BeanFactory interface, The ApplicationContext interface defines methods for retrieving beans, publishing events, and accessing application-specific resources.

When using XML-based configuration, one common implementation of the ApplicationContext interface is the ClassPathXmlApplicationContext class. This class allows you to load bean definitions from XML files in the classpath.

constr=ClassPathXmlApplicationContext(String configFile)

Here's a simple example using ClassPathXmlApplicationContext:

* Create an XML Configuration File (e.g., applicationContext.xml):

<!-- applicationContext.xml -->

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

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

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

<!-- Define a simple bean -->

<bean id="myBean" class="com.example.MyBean">

<property name="message" value="Hello, Spring!"/>

</bean>

</beans>

In day 11.2

Http transport

Copy same code in saop and test

Open dependant and atm i/f

i/f iple

The code uses setter-based Dependency Injection (DI) to inject the Transport implementation.

* Imports:
  + The code imports the Transport interface from the dependency package.
* ATMImpl Class:
  + This class implements the ATM interface (not provided in the code snippet).
* Fields:
  + It has a private field myTransport representing the dependency on the Transport interface.
* Constructor:
  + The constructor prints a message indicating the class name and the current value of myTransport.
* deposit and withdraw Methods:
  + These methods print messages indicating deposit and withdrawal actions and then use the myTransport object to inform the bank.
  + String to byte array conversion
  + Atmimpl depit is calling depcy(transpot layer
* setMyTransport Method:
* n the given code, Dependency Injection (DI) is achieved through the use of a setter method named setMyTransport
  + This is a setter method for Dependency Injection. It sets the myTransport field with the provided Transport implementation.
* Field Declaration:
  + The class ATMImpl has a private field named myTransport, which represents a dependency on the Transport interface.
* java
* Copy code

private Transport myTransport;

* Setter Method:
  + The class provides a setter method named setMyTransport that takes an instance of the Transport interface as a parameter and sets it to the myTransport field.

public void setMyTransport(Transport myTransport) {

this.myTransport = myTransport;

System.out.println("in setter " + this.myTransport);

}

* Dependency Injection Process:
  + When an instance of the ATMImpl class is created, the default constructor is invoked:
  + java
  + Copy code

public ATMImpl() {

System.out.println("in cnstr of " + getClass().getName() + " " + myTransport);

}

* + At this point, the myTransport field is not yet set, and its value is likely null.
  + Later in the application, the setter method is used to inject a specific implementation of the Transport interface into the ATMImpl instance:

ATMImpl atm = new ATMImpl();

atm.setMyTransport(new HttpTransport()); // Injecting an instance of HttpTransport

* + The setMyTransport method is called with an instance of HttpTransport, and this instance is assigned to the myTransport field.
  + The setter method also prints a message indicating the current value of myTransport.
* Usage of Dependency:
  + The myTransport dependency is used in the deposit and withdraw methods to inform the bank:
  + java
  + Copy code

public void deposit(double amt) {

// ...

myTransport.informBank(data);

}

public void withdraw(double amt) {

// ...

myTransport.informBank(data);

}

In summary, Dependency Injection in this code involves injecting an implementation of the Transport interface into the ATMImpl class through the setMyTransport setter method. This allows for the decoupling of the ATMImpl class from the specific implementation of the Transport interface, promoting flexibility and ease of testing.

Example usage of this class might look like the following:

java

Copy code

ATMImpl atm = new ATMImpl();

atm.setMyTransport(new HttpTransport()); // Injecting an instance of HttpTransport

atm.deposit(1000.0);

atm.withdraw(500.0);

Note: The code assumes the existence of an ATM interface and the Transport interface, as well as a specific implementation of the Transport interface called HttpTransport. Without the complete context and definitions for these interfaces and classes, it's difficult to provide a fully comprehensive explanation.

Dependency Injection (DI) in Simple Words:

Imagine you have a class called ATMImpl, and it needs to talk to a bank. Instead of the ATM figuring out how to talk to the bank by itself, someone else (like a manager) gives the ATM a way to communicate (a phone, for example).

In code:

java

Copy code

public class ATMImpl implements ATM {

private Transport myTransport; // This is like the phone

// The manager (setter method) gives the ATM a way to communicate (sets the phone)

public void setMyTransport(Transport myTransport) {

this.myTransport = myTransport;

System.out.println("in setter " + this.myTransport);

}

// The ATM uses the given phone to talk to the bank

public void deposit(double amt) {

// ...

myTransport.informBank(data);

}

// Similarly, the ATM uses the phone to talk to the bank for withdrawals

public void withdraw(double amt) {

// ...

myTransport.informBank(data);

}

}

What's Happening Here:

* ATM (ATMImpl) Needs a Way to Communicate:
  + The ATMImpl class needs a way to communicate with the bank, but it doesn't know exactly how.
* Dependency (Phone):
  + Instead of the ATM figuring out how to communicate by itself, someone else provides a way to communicate. In code, this is represented by the Transport interface.
* Setter Method is Like Receiving the Phone:
  + The setMyTransport method is like someone handing over the phone to the ATM. It sets the communication method (the myTransport field).
* Dependency Injection is Like Getting a Phone:
  + The process of giving the ATM a way to communicate (setting the phone) is called Dependency Injection.
* Flexible and Easy to Change:
  + If the ATM wants to use a different type of phone (a different implementation of Transport), it can easily do so by receiving a new phone (using the setter method). This makes the ATM flexible and easy to adapt.

In real-world terms, Dependency Injection helps keep things simple, flexible, and organized in our code. It's like providing tools to a class instead of the class trying to create its own tools.

add<bean>

<property name =”myTransport” ref =”test”/> :1 per depcy

Add init and destroy

* Singleton:
  + Idea: Like having a single instance of your favorite game on a computer.
  + Explanation: Singleton ensures that only one instance of a class exists, and everyone shares and uses that same instance.
  + Example: You have one central bank account manager, and everyone interacts with that same manager.
* Prototype:
  + Idea: Like making copies of a flyer to distribute.
  + Explanation: Prototype allows you to create multiple instances of a class, each being independent of the others.
  + Example: You have a template for a newsletter, and you can create as many copies as you want, each with its own content.

In short, Singleton is about having one shared instance, like a single game on a computer, while Prototype is about creating independent copies, like making multiple flyers from a template.

In bean-config.xml

Src resousres

Spring bean

Checked checkboc next

Click beans(next empty as we nor configured

what happens when this configuration is executed:

* <bean id="test" class="dependency.TestTransport"/>:
* o/p in cnstr of class dependency.TestTransport

As public ATMImpl() {

System.out.println("in cnstr of " +getClass().getName()+" "+myTransport);

}

* + A bean named "test" of type dependency.TestTransport is created. This bean will be a singleton by default, meaning there will be only one instance throughout the Spring container's lifecycle.
* <bean id="http" class="dependency.HttpTransport" lazy-init="true"/>:
* Skipped as singleton and lazy
  + A bean named "http" of type dependency.HttpTransport is created. This bean is configured as a singleton (lazy-init="true") which means it will be created lazily (on first request).
* <bean id="soap" class="dependency.SoapTransport" scope="prototype"/>:
  + A bean named "soap" of type dependency.SoapTransport is created. This bean is configured with a scope of "prototype," indicating that a new instance will be created every time it is requested.
* Skipped as prototype lazily loaded one per demand
* <bean id="my\_atm" class="dependent.ATMImpl" init-method="anyInit" destroy-method="anyDestory">:
  + A bean named "my\_atm" of type dependent.ATMImpl is created. This bean is a singleton by default.
  + The init-method attribute specifies a method named "anyInit" to be called after the bean is initialized.
  + The destroy-method attribute specifies a method named "anyDestory" to be called before the bean is destroyed.
  + Called as singleton and eager

o/p in cnstr of dependent.ATMImpl null

* <property name="myTransport" ref="soap" />:

Property used for setter based

* + This configures a property named "myTransport" of the "my\_atm" bean to be injected with the bean named "soap." This is an example of Setter-based Dependency Injection, where the setMyTransport method of the ATMImpl class is invoked with the "soap" bean.
* o/p in cnstr of dependency.SoapTransport
* Created dependency

o/p

in setter dependency.SoapTransport@26275bef

in initdependency.SoapTransport@26275bef

Init called for both singleton and prototype

Destory called for singleton as prototype u hav dfinened

o/p

sc is up n running

in destorydependency.SoapTransport@26275bef

Summary:

* Beans "test" and "http" are singletons (one instance per container).
* Bean "http" is lazily initialized.
* Bean "soap" is a prototype (a new instance every time it's requested).
* Bean "my\_atm" is a singleton with initialization and destruction methods specified.
* The "my\_atm" bean has its "myTransport" property injected with the "soap" bean.

This configuration demonstrates the flexibility of Spring in managing different bean scopes and lifecycle methods, as well as the ability to perform dependency injection.

Create tester

* In try with resourceThe Spring container is created and initialized with the configuration from "my-config.xml."

Run spring containers using xml instructions where it resides in classpath

* The message "sc is up n running" is printed, indicating that the Spring container is up and running.

(When the try block is exited (due to the end of the program), the Spring container is automatically closed.If there are singleton beans in the container, their destroy methods are called during container shutdown.)

In property saop to test

<property name="myTransport" ref="test" />

o/p in cnstr of class dependency.TestTransport

in cnstr of dependent.ATMImpl null

in setter dependency.TestTransport@26275bef

in initdependency.TestTransport@26275bef

sc is up n running

in destorydependency.TestTransport@26275bef

Loose coupling as call any ban ref and whole conatoner change

* What is getBean?
  + getBean is a method in Spring that retrieves an instance of a bean from the Spring container.
  + <T> T getBean(String beanId/name,Class<T> requiredType);
  + Object getBean(String name);
* How to use getBean?
  + You can use it by providing the bean's name or its class type.
  + Example:

MyBean myBean = context.getBean("myBean", MyBean.class);

* Example with Simple Bean:
  + Suppose you have a MyBean class with a message.
  + In your Spring configuration, you define this bean.
  + In your Java code, you use getBean to get an instance and then work with it.

MyBean myBean = context.getBean("myBean", MyBean.class);

myBean.displayMessage();

* This retrieves an instance of MyBean from the Spring container and calls its displayMessage method.

That's the essence of using getBean—getting instances of your beans from the Spring container for use in your application.

In tester

//withdraw 1000 from atm

//api if o.s.b.f.BeanFcatory

//<T> T getBean(String beanId/name,Class<T> requiredType);

ATMImpl ref1=ctx.getBean("my\_atm",ATMImpl.class);

ref1.withdraw(1000);

/\*@Override

public void withdraw(double amt) {

System.out.println("withdrawing "+amt);

byte[] data=("withdrawing "+amt).getBytes();

myTransport.informBank(data);

}\*/

//another demand for bean

ATMImpl ref2=ctx.getBean("my\_atm",ATMImpl.class);

System.out.println(ref1==ref2);//true as singleton

o/p

in cnstr of class dependency.TestTransport

in cnstr of dependent.ATMImpl null

in setter dependency.TestTransport@26275bef

in initdependency.TestTransport@26275bef

sc is up n running

withdrawing 1000.0

informing bank using class dependency.TestTransport layer

true

in destorydependency.TestTransport@26275bef

Now in xml

My\_atm scope ="prototype"

o/p

in cnstr of class dependency.TestTransport

sc is up n running

in cnstr of dependent.ATMImpl null

in setter dependency.TestTransport@31e5415e

in initdependency.TestTransport@31e5415e

withdrawing 1000.0

informing bank using class dependency.TestTransport layer

in cnstr of dependent.ATMImpl null

in setter dependency.TestTransport@31e5415e

in initdependency.TestTransport@31e5415e

False

On second demand container give ref1=ref2 false and

o/p line 1 as singleton and eager

o/p lin3 -6 trigger by context.getBan

Last 4 lines

False prottype’and destror method not called as protype

In xml

Cut dependency bean in top

And fref http

in cnstr of class dependency.TestTransport

sc is up n running

in cnstr of dependent.ATMImpl null

in cnstr of dependency.HttpTransport

in setter dependency.HttpTransport@31e5415e

in initdependency.HttpTransport@31e5415e

withdrawing 1000.0

informing bank using dependency.HttpTransport layer

in cnstr of dependent.ATMImpl null

in setter dependency.HttpTransport@31e5415e

in initdependency.HttpTransport@31e5415e

false

When run test spring skip protoptype as conatine not en=ven see it at startup

Then transport(singleton and eager) ->sc runnin-> in atm impl (1st demand)->not call setter as depency not created)->setter)->init-> htplayer->false as protoptype

How many beans after cntrl reach line 25

| Test  Myatm1stinstance http1instance  Myatm2 instance |
| --- |

In xml in http lazy init=false and prototype scope

in cnstr of class dependency.TestTransport

sc is up n running

in cnstr of dependent.ATMImpl null

in cnstr of dependency.HttpTransport

in setter dependency.HttpTransport@31e5415e

in initdependency.HttpTransport@31e5415e

withdrawing 1000.0

informing bank using dependency.HttpTransport layer

in cnstr of dependent.ATMImpl null

in cnstr of dependency.HttpTransport

in setter dependency.HttpTransport@550ee7e5

in initdependency.HttpTransport@550ee7e5

false

On 1 demand stack executed atm constr http ka ist instance as protyopy \_.zteer init

On 2 demand ->ref==ref2 fALSE

Each depndant has its own cpy not sharing as prototype

**Day 12**

Spring mvc template

0. Copy n Import "day14-data\day14\_help\maven based spring web MVC app\spring\_mvc\_template" , in your workspace , as existing maven project

Now we enter web basesd upto now we use core

2 type if web servlet and portlet

This is sevlet

Step 1 configuration

Copy web.xml sevlet ansd servlet mapping

In src main webapp web .xml ater welocme list

Front controller (Dispatcher servlet) start spring container

Copy servlet in web inf

As servlet name in web.xml tag is spring thats why name srping-serv;et

In index .jsp give hello link

In main/java in pkh in class

@Controller //mandatory class level

//tell sc following is req handling cotroller bean

public class HelloWorldController {

Type Hell and crtl space to add a def constr

public HelloWorldController() {

System.out.println("in ctor od"+getClass());

}

@RequestMapping is an annotation used to map a web request to a specific method in a controller. It helps define which method should handle a particular URL.

Map means it has key and value

Value in xml

* + sayHello method should handle HTTP requests with the URL pattern "/hello."
* Method Logic:
  + When a request is made to "/hello," the sayHello method is executed.
* System.out.println("in say hello");:
  + Outputs a message to the console when the sayHello method is called.
* return "/welcome";:
  + Specifies the logical view name ("/welcome") that should be returned by this handler method.
* Logical View Name (LVN):
  + The logical view name is a string that represents the name of the view that should be rendered.
* Key /hello and Value = Fully Qualified Class Name:
  + The key for this mapping is the URL pattern "/hello."
  + The value is the fully qualified class name of the controller, which in this case is the class containing the sayHello method.
* Multiple Handling Methods:
  + A controller can have multiple handler methods.
  + If there are multiple methods handling different URL patterns, the @RequestMapping annotation allows you to specify multiple mappings.
* Fully Qualified Handler Class Name + Method Name:
  + In the case of multiple handling methods, if you want to uniquely identify a method, you can use the fully qualified class name along with the method name.
  + For example: com.app.controller.HelloWorldController.sayHello
* Return Value Impact on DispatcherServlet (DS):
  + The logical view name returned by the handler method is used by the DispatcherServlet to determine which view to render.
  + In this case, it's "/welcome."

In summary, the @RequestMapping("/hello") annotation maps the "/hello" URL pattern to the sayHello method in the controller. The method returns a logical view name ("/welcome"), which is used by the DispatcherServlet to navigate to the appropriate view.

@RequestMapping("/hello")

public String sayHello() {

System.out.println("in say hello");

return "/welcome";

}

/ as parameter

Key /hello

Value =fully qualified class name

But can multiple handling methods

So value fully qualified handler class name+method name

//value com.app.controller.HelloWorldController.sayHello

return "/welcome";//lvn (logical view name)

//handler--> rets lvn -->d.s

In spring-servet

<!-- sc will scan com.app for managing spring beans -->

<context:component-scan base-package="com.app"/>

instructed to look for these special classes in the "com.app" neighborhood and bring them into the Spring application

* <bean> Element: This declares a bean in the Spring application context. id="viewResolver": Assigns the name "viewResolver" to this bean so that other parts of the application can refer to it.
* class="org.springframework.web.servlet.view.InternalResourceViewResolver": Specifies the class responsible for resolving views. In this case, it's InternalResourceViewResolver, which resolves JSP views.
* p:prefix="/WEB-INF/views": Sets the prefix for view names. When a controller returns a view name, this prefix is added to the beginning. In this example, views are expected to be in the "/WEB-INF/views" directory.
* p:suffix=".jsp": Sets the suffix for view names. This is appended to the end of view names. In this case, views are expected to end with ".jsp".
* p:viewClass="org.springframework.web.servlet.view.JstlView": Specifies the class for rendering views. JstlView is used for JSP rendering and integrates with the JavaServer Pages Standard Tag Library (JSTL).

In summary, this configuration is creating a "view resolver" bean named "viewResolver" that automatically translates view names into JSP files located in the "/WEB-INF/views" directory with a ".jsp" extension. It simplifies the process of defining and finding views in a Spring web application.

In HelloWorldController

HelloWorldController is forwarding requests to a specific JSP file named "Welcome."

Dipatcher servlet forwards (Add Attribute View Name?):

* It mentions adding an attribute related to the view name.) client to view layer in same req and call rd.forward

Rest all are spring supplied

In src –webinf-new -other -jsp file-next

Not any name but specific

Welcome

Insert header welcome

Run on server

Send / to handler mapping bean

Why /

Mapping "/" in a Spring MVC application is crucial because it sets up how the application handles requests to the root URL. It defines which controller method should run when someone accesses the main page of the website (e.g., "http://example.com/"). This helps initialize the application, display a welcome page, or perform any necessary setup. It's a way of telling Spring how to respond when users visit the home page.

But we want not null so add one more controller (@Controolet)req mapping ki vlaue / after that public Strint method name and return /index(non null value)

Why might you add another controller for handling the root URL ("/") in a Spring MVC application?

A:

Adding another controller for the root URL ("/") allows you to handle the default entry point for your application. This is often done when you want to replace default welcome pages configured in web.xml with annotated controllers. The additional controller ensures a non-null return value for the root URL, providing a clear way to initialize the application or display a welcome page.

How can you ensure that a controller method returns a non-null value for the root URL ("/") in Spring MVC?

A:

By creating a dedicated controller method for the root URL ("/") and ensuring that it returns a non-null value. For example, you can use @RequestMapping("/") and have the method return a logical view name (e.g., "/index"). This ensures that when users access the root URL, the specified view is displayed.

when a user accesses the root URL ("/"), the new controller's defaultPage method will handle the request and return the logical view name "/index". The view resolver will then resolve this to the actual JSP file located under the "views" directory.

Remove welcome list atge from weninf->web.xml . typically done when you want to handle the default request ("/") using annotated controllers instead of relying on default welcome pages.

Take index.jsp under views

Make in com.app controller (constr-reqhandling method

@Controller

public class HomePageController {

public HomePageController() {

System.out.println("in ctor of"+getClass());

}

//req handling method

@RequestMapping("/")

public String showIndex() {

System.out.println("in show index page");

return "/index";

}

Not index.jsp but /index

Now ds give to view :web-inf/views/

But this static content we wnat dynami

View only show but controller is who give result

Service k niche dao below hibernat below db

Controller call service method service call dao dao given back to controller

ModelAndView in Spring is like a messenger for both data (the model) and the logical view name.The model represents the data, and the view's logical name indicates where the data should be displayed.

that carries both a message (your data) and the address where the message should be delivered (the view). It helps controllers send information to views in a single package. The message is your model data, and the address is the logical name of the view.

think of ModelAndView like a package. It contains both the message (your data) and the address (logical view name) where the message should be delivered (the view). Here's a quick breakdown:

* Controller Action:
  + When you ask for something in a Spring web app, a controller method handles it.
  + The method sends back a ModelAndView, a package with data and the view's name.
* DispatcherServlet and ViewResolver:
  + The DispatcherServlet takes the package and asks the ViewResolver to find the actual view (like a JSP file) using the given name.
* Return to Controller:
  + Control comes back to the controller after the DispatcherServlet figures out which view to use.
* Add Data to the View:
  + Before heading to the view, the controller can add more info (model attributes) to the package.
* Forward to the View (JSP):
  + Finally, the DispatcherServlet sends the package to the view (often a JSP file).
  + The JSP can access the data using ${attr} (for example), showing it to the user.

In short, the controller decides what to display, the DispatcherServlet finds the view, and the view gets the data to show. This setup makes it simple to organize and understand the different parts of a web app.

How does the DispatcherServlet and ViewResolver work together in Spring MVC?

A:

The DispatcherServlet receives a ModelAndView from a controller. It then asks the ViewResolver to find the actual view (like a JSP file) based on the logical name provided in the ModelAndView. Once resolved, control returns to the controller, which can add more data to the view. Finally, the DispatcherServlet sends the package to the view, and the view displays the data.

What role does the DispatcherServlet play in Spring MVC?

* A. Resolves views.
* B. Handles database connections.
* C. Processes incoming requests and directs them to the appropriate controller.
* D. Initializes the Spring application context.

Answer: C

How can you ensure a controller method returns a non-null value for the root URL ("/") in Spring MVC?

* A. Using @RequestMapping("/index").
* B. Including a default welcome page in web.xml.
* C. Adding a return statement with a logical view name.
* D. Configuring a custom view resolver.

Answer: C

Handler rt model and view

Ds send to view resolver

Vr send back

Control come back to controller m

Model attribut ds check are there any model attrity if any add it under

Forward to jsp

If got req scope attr $ reqscope.attr name

ModelAndView modelAndView = new ModelAndView();

ModelAndView modelAndView = new ModelAndView("viewName");

ModelAndView modelAndView = new ModelAndView("viewName", "attributeName", attributeValue);

In index.jsp <h5>

<a href="test/test1">Test Model and view</a>

</h5>

* The URL "test/test1" suggests a two-level mapping.
* The first level is specified at the class level using @RequestMapping("/test") on the controller class.
* The second level is specified at the method level using @GetMapping("/test1") on the specific method.

the hyperlink in the index.jsp is directing the user to the URL "test/test1," and in your Spring MVC application, the TestController class is configured to handle requests with this URL pattern. The first level ("/test") is defined at the class level, and the second level ("/test1") is defined at the method level.

Until now 1 level pattern /test but now 2 level /test/tst1

1st on class level 2 on method level

Go to com.app.controll

Class test controller

@Controller

@RequestMapping("/test")//all urls with /test as url pattern will be handled by this test controller

public class TestController {

public TestController() {

System.out.println("in ctor of" +getClass());

}

//req handling method

@GetMapping("/test1") //@RequestMapping(method=get)

//key /test/test1

//value : TestController.testModelAndView

public ModelAndView testModelAndView() {

System.out.println("in test m&v");

//o.s.w.s.ModelAndView(String LVN,String modelAnd

//Object modelAtrrValue

return new ModelAndView("/test/display", "server\_ts", LocalDateTime.now());

}

This Java class is a Spring MVC controller named TestController. It handles requests with the URL pattern "/test".

@Controller: Annotation indicating that this class is a Spring MVC controller. It plays a key role in handling web requests.

* @RequestMapping("/test"): Specifies that this controller will handle requests with the base URL pattern "/test". For example, "/test/test1" will be handled by methods in this controller.
* public class TestController: Defines the class named TestController.
* public TestController(): Constructor of the TestController class. It prints a message when an instance of the controller is created.
* @GetMapping("/test1"): Specifies that the method below will handle HTTP GET requests with the URL pattern "/test/test1".
* public ModelAndView testModelAndView(): This method handles the "/test/test1" URL. It returns a ModelAndView object, which combines both model data and the logical view name.
* return new ModelAndView("/test/display", "server\_ts", LocalDateTime.now()): Creates a ModelAndView instance. The logical view name is set to "/test/display", and a model attribute named "server\_ts" is added with the current date and time (using LocalDateTime.now()).

In summary, TestController is a Spring MVC controller handling requests starting with "/test". The testModelAndView method returns a ModelAndView with a logical view name and some dynamic data.

All incoming url handle by test controller

Print server timestamp $reqscope server\_ts

Right click on views new jsp other display.jsp and write <h4>Server TimeStamp: ${requestScope.server\_ts}</h4>

* + ${requestScope.server\_ts} is a JSP expression. It's enclosed within ${} to denote that it's an expression to be evaluated by the JSP engine.
* requestScope:
  + requestScope is a JSP implicit object that refers to the HttpServletRequest object, representing the current HTTP request.
  + It allows you to access attributes stored in the request scope.
* server\_ts:
  + server\_ts is the attribute name being accessed from the request scope.
  + This attribute seems to hold the server timestamp.
* Overall Purpose:
  + The purpose of this HTML snippet is to display a heading saying "Server TimeStamp:" followed by the value of the server\_ts attribute from the request scope.
* Dynamic Content:
  + The value inside ${} is dynamically replaced with the actual value of the server\_ts attribute when the JSP page is rendered.
  + For example, if the server timestamp is "2023-01-01T12:34:56," it will be displayed as "Server TimeStamp: 2023-01-01T12:34:56" in the rendered HTML.

In summary, this JSP expression retrieves the value of the server\_ts attribute from the request scope and displays it as part of an HTML heading. It's a way to dynamically show the server timestamp on the web page.

**Day 13**

* Presentation Layer: This is the topmost layer that deals with user interfaces and displaying information to users.
* Business Logic Layer (Service Layer):Manages the application's core logic.
* Persistence Layer: deals with the storage and retrieval of data. JPA is commonly used in this layer to provide a convenient way to interact with relational databases.
* Entity Manager Layer:Manages interactions with the database.
* Session Layer (Transaction Management): This layer is responsible for managing transactions. In a JPA context, transactions are used to group multiple database operations into a single, atomic unit of work. The session layer ensures that transactions are properly started, committed, or rolled back.

Database (DB): Where data is stored.

* JDBC (Java Database Connectivity): Basic Java library for talking to databases, involves writing lots of code.
* Hibernate:An Object-Relational Mapping (ORM) framework for Java that provides a higher-level and more convenient way to interact with databases. Hibernate allows you to map Java objects to database tables, and it handles the translation between object-oriented code and relational database structures. It abstracts away much of the low-level JDBC code.
* JPA (Java Persistence API): A standardized interface for database interaction in Java,JPA is a specification that standardizes how Java applications interact with databases, providing a common set of interfaces and annotations. Hibernate is one of the implementations of JPA.

Spring Data JPA: You just need to declare an interface (DAO), and Spring Data JPA magically handles the database operations without you having to write the implementation code. It's like asking nicely, and Spring Data JPA does the work for you.

org.springframework.data.repository.Repository:

* Definition: Interface in Spring Data framework.
* Usage: Serves as a base interface for all repository interfaces.
* Purpose: Provides basic CRUD (Create, Read, Update, Delete) operations and query methods.

<T>:

* Meaning: Placeholder for the entity type.
* Example: If managing Person entities, <Person> represents the actual entity type.

<ID>:

* Meaning: Placeholder for the type of the entity's ID property.
* Example: If the entity has a Long ID, <Long> is used.

Repository<T, ID>:

* Purpose: Generic interface where T is the entity type, and ID is the type of the ID property.
* Example: PersonRepository extends Repository<Person, Long> indicates a repository for managing Person entities with a Long ID.

In essence, Repository<T, ID> is a versatile, generic interface that forms the foundation for creating repositories in Spring Data, allowing you to define and use repositories for different entity types with varying ID types.

* + Repository as a Marker Interface: interface that doesn't declare any method but is used to mark a certain capability or property. In this context, Repository is a marker interface that indicates that the interface extending it is intended to be a repository for storing and retrieving entities.
  + Extending Repository<T, ID> means saying, "I'm a repository for dealing with specific things (T) and their unique IDs (ID)." Spring Data JPA does the heavy lifting by automatically creating common database operations, saving you from writing them manually. It's like a coding assistant that understands what you need without you spelling out every detail.
  + The actual methods for CRUD operations (Create, Read, Update, Delete) are not explicitly declared in Repository. Instead, Spring Data JPA generates these methods based on naming conventions in the extending interfaces.

CRUDRepository Methods:

* + eg : CRUDRepository methods
  + long count() =Returns the number of entities available.
  + void delete(T entity)=Deletes a given entity.
  + void deleteAll()
  + Deletes all entities managed by the repository.
  + void deleteAll(Iterable<? extends T> entities)
  + Deletes the given entities.
  + void deleteById(ID id)
  + Deletes the entity with the given id.
  + boolean existsById(ID id)
  + Returns whether an entity with the given id exists.
  + Iterable<T> findAll()
  + Returns all instances of the type.
  + Iterable<T> findAllById(Iterable<ID> ids)
  + Returns all instances of the type with the given IDs.
  + Optional<T> findById(ID id)=Retrieves an entity by its id.

The findById method in CrudRepository retrieves an entity by its unique ID. It returns an Optional that either contains the entity if found or is empty if not. This helps handle the case where the entity might not exist, promoting safer code.

Advantage of Optional:

* The use of Optional encourages developers to handle cases where the entity might not exist, reducing the reliance on checking for null values directly.
  + <S extends T>

S save(S entity)

* + 3. Sub i/f
  + If the ID is null: It performs an insert query (save).
  + If the ID is not null and matches a record in the database: It performs a select and update.
  + If the ID is not null but doesn't match any record in the database: It performs a select and insert.
* The save method intelligently handles different scenarios based on the presence and matching of the ID.

save(S entity) Method:

* Function: Saves or updates a given entity.
* Conditions:
  + If ID is null: Inserts a new record.
  + If ID is not null and matches a record: Selects and updates.
  + If ID is not null but no match: Selects and inserts.

saveAll(Iterable<S> entities) Method:

* Function: Saves all given entities.
* Example:
* java
* Copy code

personRepository.saveAll(Arrays.asList(new Person("Alice"), new Person("Bob")));

* Efficiency: Handles the persistence of multiple entities efficiently.

By using CrudRepository, you get a set of common and convenient methods for working with entities in a database, reducing the need to write these CRUD methods from scratch in your DAO (Data Access Object).

* + org.springframework.data.repository.PagingAndSortingRepository: This is a sub-interface of CrudRepository and extends the basic CRUD operations with methods for sorting and pagination.
  + <T>: Represents the type of the entity (the Java class representing an object that will be stored in the database).
  + <ID>: Represents the type of the ID property of the entity. The ID property is typically used as a unique identifier for each entity.
* Methods:
  + Iterable<T> findAll(Sort sort):
    - Functionality: Retrieves all entities and returns them in the specified order.
    - Parameter (Sort): Allows you to define the sorting order (ascending or descending) based on specific entity properties.
    - Example:
    - java

Iterable<Person> sortedPeople = personRepository.findAll(Sort.by(Sort.Direction.ASC, "lastName"));

* + Page<T> findAll(Pageable pageable):
    - Functionality: Retrieves entities in a paginated manner.
    - Parameter (Pageable): Specifies the page number, the size of each page, and additional sorting information.
    - Return Type (Page<T>): Represents a page of results, including the current page's content, total number of pages, and more.
    - Example:

Page<Person> page = personRepository.findAll(PageRequest.of(0, 10, Sort.by("firstName")));

* Usage - Sorting and Pagination:
  + Sorting: When you want results in a specific order, you can use findAll(Sort sort) with a Sort object.
  + Pagination: When dealing with a large set of results, you can use findAll(Pageable pageable) to retrieve a specific page of results, specifying the page number, page size, and optional sorting.

In summary, PagingAndSortingRepository extends CrudRepository and provides additional methods for sorting and paginating the results, making it useful for scenarios where you need to manage large datasets efficiently.

Pagination

instead of showing all items on one page, you display a portion, and users can navigate through different sections.

In Spring Data repositories, methods like findAll(Pageable pageable) facilitate paginated data retrieval. For instance:

java

Copy code

Page<Person> page = personRepository.findAll(PageRequest.of(1, 10));

This retrieves the second page with ten items per page.

public interface DepartmentDao extends JpaRepository<Department,Long> {

//empty i/f

integrating Java applications with relational databases.

return deptDao.getAllDepartments();

To return deptDao.findAll();

Interface Departemnet dao comment

`@RequestParam` is a Spring framework annotation used to extract values from the query parameters of a URL or the form data submitted with an HTTP request. It is commonly used in Spring MVC controllers to handle input parameters from the client.

- \*\*Usage:\*\*

- Applied to method parameters in a controller method.

- Binds the value of a query parameter or form field to the annotated parameter.

- \*\*Example:\*\*

```java

@RestController

@RequestMapping("/api/employees")

public class EmployeeController {

@GetMapping("/search")

public List<Employee> searchEmployees(

@RequestParam(name = "name", required = false) String name,

@RequestParam(name = "department", required = false) String department) {

// Implementation to search employees based on name and/or department

}

}

```

In this example, the `searchEmployees` method can be called with URLs like `/api/employees/search?name=John&department=IT`. The `@RequestParam` annotations extract the values of the "name" and "department" query parameters.

- \*\*Attributes:\*\*

- \*\*name:\*\* Specifies the name of the request parameter to bind.

- \*\*required:\*\* Indicates whether the parameter is required. Defaults to `true`.

- \*\*Default Value:\*\*

```java

@GetMapping("/default")

public String defaultValueExample(@RequestParam(name = "param", defaultValue = "default") String param) {

// Implementation

}

```

In this example, if the "param" query parameter is not present in the URL, it defaults to "default".

- \*\*Multiple Values:\*\*

```java

@GetMapping("/multiple")

public String multipleValuesExample(@RequestParam List<String> values) {

// Implementation

}

```

This example binds multiple values for a parameter into a `List`.

Using `@RequestParam` provides a convenient way to handle input parameters in Spring MVC controllers, making it easy to work with query parameters or form data in your application.

\*\*@RequestParam:\*\*

- \*\*Purpose:\*\* Grabs data from the URL or form fields in a Spring MVC controller.

- \*\*Usage:\*\* Applied to method parameters.

- \*\*Example:\*\*

```java

@GetMapping("/search")

public List<Employee> searchEmployees(

@RequestParam(name = "name", required = false) String name,

@RequestParam(name = "department", required = false) String department) {

// Implementation to search employees based on name and/or department

}

```

- \*\*Attributes:\*\*

- \*\*name:\*\* Specifies the parameter name.

- \*\*required:\*\* Determines if the parameter is mandatory.

- \*\*Default Value Example:\*\*

```java

@GetMapping("/default")

public String defaultValueExample(@RequestParam(name = "param", defaultValue = "default") String param) {

// Implementation

}

```

- \*\*Multiple Values Example:\*\*

```java

@GetMapping("/multiple")

public String multipleValuesExample(@RequestParam List<String> values) {

// Implementation

```

Using `@RequestParam` simplifies getting data from URLs or forms in your Spring application.

Resource properties change pass

Department dao

Select e from employee e where e.dept.id =:id

\*\*Finding Employees by Last Name and First Name:\*\*

In your Spring Data JPA repository, you can create a method named `findByLastNameAndFirstName` to effortlessly retrieve employees based on their last name and first name. The method signature does the work for you, generating the appropriate query to find employees with the specified last name and first name.

```java

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

List<Employee> findByLastNameAndFirstName(String lastName, String firstName);

}

```

Now, when you use this method, it automatically fetches employees matching the given last name and first name.

```java

@Service

public class EmployeeService {

@Autowired

private EmployeeRepository employeeRepository;

public List<Employee> findEmployeesByName(String lastName, String firstName) {

return employeeRepository.findByLastNameAndFirstName(lastName, firstName);

}

}

```

This makes querying for employees based on multiple criteria simple and concise.

Pojo dao

Now we at spring data jpa

In dao

Department dao already created

Create empl dao not standalone but sub I/f extending from jpa

Add interface jparapo ctrl space only interface not impl

<Employee,Long>

Return list of emp

There is get by clause and find by clause

FindBy ctrl space

FindByDept again ctrl space

FindBy dept I'd

Arg is not @Requestparam it is in controller

Arg long deptid

Jpa implement this in a bean

This is interface

Now interface employee Service

List …

Now emplyserviceimpl

@service @transactional

@autowire

Empdao empdao

List..

Return empdao.findBy

Now service layer completed

Now above service controller

In com.app controleer employee controller

@controler@requestmapping("

Add debug print statement in constr

Add ependency autowired service

First write url [http://host:port/context](about:blank) path/emp/list

Add endpoints list makes project easy

We still write monolithic

@PostMapping

Logical view name

@requestParam long can write any name as form not created yet so wirte dept,model map

When container see this it call

Controller has to call service layer method so

Map.add attr("emplist,empService.getAllEmpl)

Add debug stmt of I'd map

Return actual view name

One option insert into

Or

Write unit test cases

Junit test cases

Com.app.dao

Employee daotest

Ye@DataJpaTest@AutoConfigure

Inject depcy autowire dao

@Test

1L as it is long

Get existing dept by its I'd

W/o dao implementation line

Runt it

In src test java com a.app dao eployeedqotesy

Hardcode 1 L

Add form in webinf dept list

Lombok reduce boiler code

Lombok 11.30

If not found installupdqte

If not found specify location(sts location)sts properties or when u lauch id

Select c..sts.exe

Reopen ide

This remove getid error

**Day 14**

Import force update

Create com .app in tha application class

Add diff pack like controlle.,pojo..

Keep perspective java or spring not java .ee

In controller pkg

Testcontroller class

Use jackson as vendor json array return

Martial java to json sent to rest client

@RestController

@RequestMapping("/test")

public class TestController {

public TestController() {

System.out.println("in ctor of" + getClass());

}

//http://host:port/test

@GetMapping

public List<Integer> testMe(){

System.out.println("in test me");

return List.of(1,2,3,4,5);

}

}

@RestController

@RequestMapping("/test")

public class TestController {

* + @RestController: Indicates that this class is a controller and that the return values of the methods should be directly written to the response body (as opposed to being resolved to a view).
  + @RequestMapping("/test"): Maps the controller to the "/test" URL path.

public TestController() {

System.out.println("in ctor of" + getClass());

}

* This is the constructor of the TestController class. It prints a message to the console when an instance of the controller is created.
* Handler Method:

@GetMapping

public List<Integer> testMe(){

System.out.println("in test me");

return List.of(1,2,3,4,5);

}

* + @GetMapping: Maps the method to handle HTTP GET requests.
  + public List<Integer> testMe(): This method returns a list of integers and is mapped to the "/test" URL path. When a GET request is made to "/test", this method is executed.
  + Inside the method, a message is printed to the console, and a list of integers (1, 2, 3, 4, 5) is returned.

In summary, when you run your Spring application, and a request is made to the "/test" URL, the testMe method of the TestController class is executed. The constructor prints a message when an instance of the controller is created. The @GetMapping annotation ensures that this method is invoked for HTTP GET requests to the specified URL path

Run as spring boot app

http://http://localhost:8080/test



Go in postman

Get paste url

Copy employee and base entity in pojos and refactor rename to entities

Rename name from first ,last join date why ? because ni likhoge tobhi hofa code reduce

In src test java

Right click on src test jva

New junitetestcase(task k niche)

New junit jupiter

Pakg com.app.dao.

name=EmployeeDaoTest

@DataJpaTest

@AutoConfigureTestDatabase(replace=Replace.NONE)

@Rollback(false)

class EmployeeDaoTest {

@Test

void test() {

fail("Not yet implemented");

}

}

The purpose of this test is to check if the saveAll method of EmployeeDao works correctly by saving a list of Employee objects to a real database.

Here's a breakdown:

* Annotations:
  + @DataJpaTest: Specifies that this is a test for a JPA repository, configuring the necessary components for testing with databases.
  + @AutoConfigureTestDatabase(replace=Replace.NONE): Instructs the test to use the actual database, not a simulated one.
  + @Rollback(false): Ensures that changes made during the test are committed to the database.
* Autowired EmployeeDao:
  + @Autowired private EmployeeDao empDao;: Injects the EmployeeDao into the test class, allowing the test to use it to interact with the database.
* Test Method:
  + The testSaveEmps method creates a list of Employee objects and saves them to the database using the saveAll method of EmployeeDao.
  + It then checks if the number of saved employees matches the expected number (4 in this case) using an assertion (assertEquals).

In essence, this test checks if the database interaction (saving multiple employees) in your Spring Boot application works as intended. The annotations help set up a testing environment, and the test method performs the actual test by creating sample data, saving it, and verifying the results.

In com.app.dao

EmployeeDao add jpaRepository

t=employee id =long

Add dependency in test

@Autowired

private EmployeeDao empDao;

Add list

List<Employee> list = List.of(

new Employee("a1", "b1", "a1@gmail.com", "1234", LocalDate.parse("2020-10-20"), 34567, "Pune", "RnD"),

new Employee("a2", "b2", "a2@gmail.com", "2234", LocalDate.parse("2020-11-20"), 44567, "Pune", "RnD"),

new Employee("a3", "b3", "a3@gmail.com", "1274", LocalDate.parse("2023-10-20"), 14567, "Pune", "RnD"),

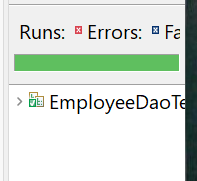
new Employee("a4", "b4", "a4@gmail.com", "2234", LocalDate.parse("2022-10-20"), 40567, "Mumbai","Finance"));

List<Employee> list2=empDao.saveAll(list);

assertEquals(4,list2.size());

Run as junit testcase

Green bar means succes

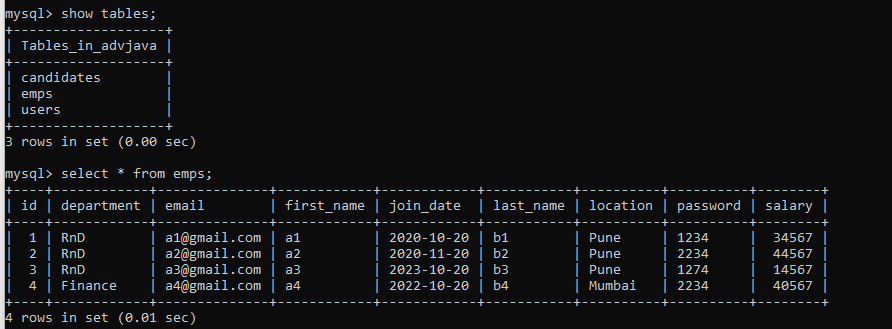


It shows debug

We see create tab;e and insert table

In mysql

Show tables



In react

Open newfolder empapp open terminal 3 dots npm installl npm start

http axios

Base url tell it connects fe to be in 8080 port

http

Service getall

Now in service layer get all

In nonthing return list emp

In com.servixe new empl service

In controller create employee controller

And /emp same as that of in react htt axios

* + @RestController: Marks the class as a controller, indicating that it handles HTTP requests and returns the response directly.
  + @RequestMapping("/employees"): Maps the controller to the "/employees" URL path.
  + @CrossOrigin(origins="http://localhost:3000"): Allows cross-origin requests from "[http://localhost:3000](http://localhost:3000/)," which is useful for web applications.
* Autowired EmployeeService:
  + @Autowired private EmployeeService empService;: Injects an instance of EmployeeService into the controller. This service likely handles business logic related to employees.
* Request Handling Methods:
  + @GetMapping: Handles HTTP GET requests to "/employees." Calls the getAllEmps method of EmployeeService to retrieve a list of all employees.
  + @PostMapping: Handles HTTP POST requests to "/employees." Expects a JSON payload (@RequestBody) representing a new employee (Employee object) and calls addEmpDetails in EmployeeService to add the new employee.
* Controller Constructor:
  + public EmployeeController() {}: An empty constructor, possibly for default initialization.

In summary, this controller provides two endpoints: one for retrieving a list of all employees (GET), and another for adding a new employee (POST). The EmployeeService is responsible for handling the underlying business logic related to employees. The @CrossOrigin annotation allows requests from a specific origin, which is useful for allowing requests from a web application running on "[http://localhost:3000](http://localhost:3000/)."

You

in easy n]and short

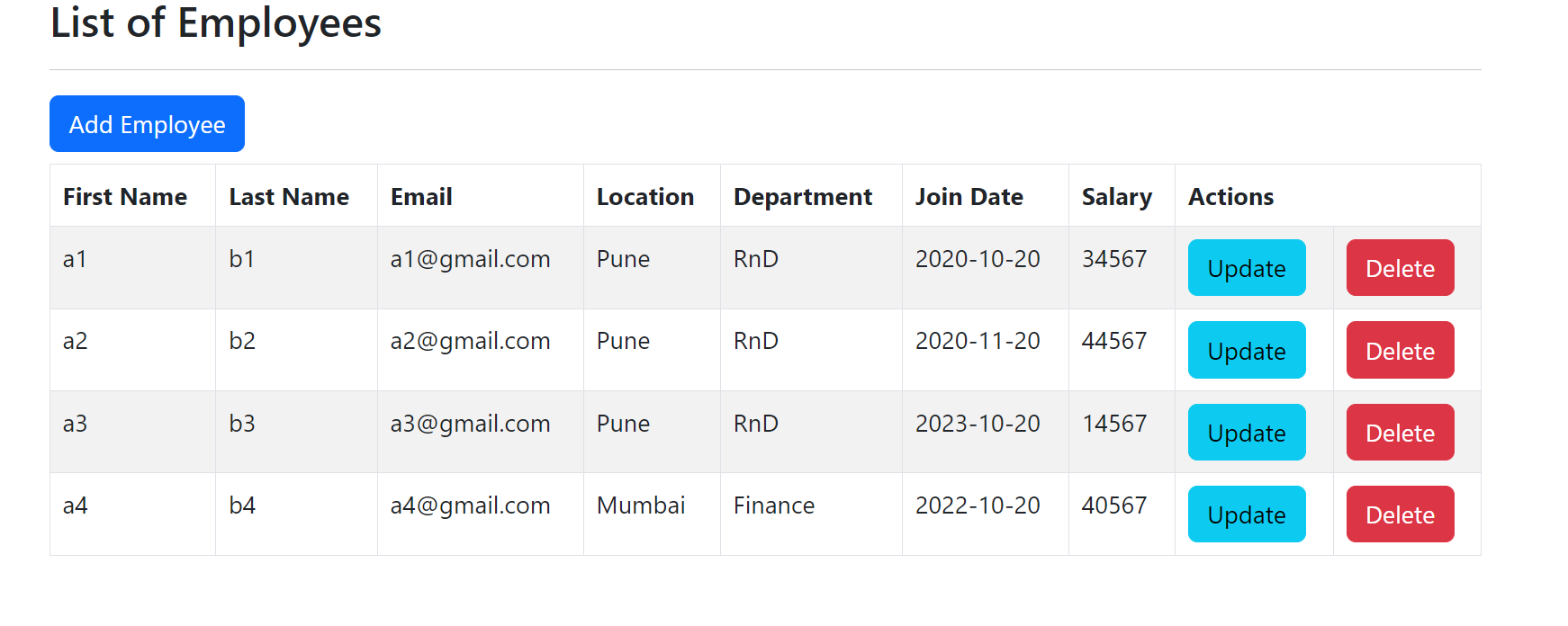
This code is for a Spring Boot controller that manages employee data:

* Purpose:
  + Manages employee-related operations.
* Endpoints:
  + GET "/employees":
    - Lists all employees.
    - Uses EmployeeService to retrieve data.
  + POST "/employees":
    - Adds a new employee.
    - Expects employee details in the request body.
* Annotations:
  + @RestController: Identifies it as a controller.
  + @RequestMapping("/employees"): Maps to "/employees" URL.
  + @CrossOrigin(origins="http://localhost:3000"): Allows requests from "[http://localhost:3000](http://localhost:3000/)."
* Dependencies:
  + Depends on EmployeeService for business logic.

In essence, it provides simple web endpoints to get a list of all employees and add a new employee. The EmployeeService handles the actual logic. Cross-origin requests are allowed from "[http://localhost:3000](http://localhost:3000/)."

Now go in react vs code and npm start

Now this displa list



In pom.xml

Line 56

<dependency>

<groupId>org.springdoc</groupId>

<artifactId>springdoc-openapi-ui</artifactId>

<version>1.7.0</version>

</dependency>

Test it with postman/swagger & then with React front end.

(Add swagger dependency here)

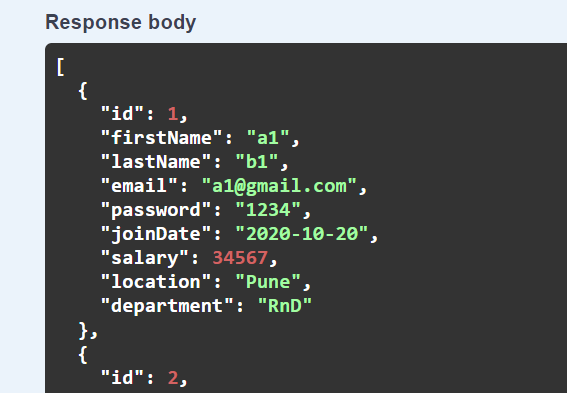
Steps

1. Add swagger dependency in pom.xml

(Already added in spring boot backend template project)

2. From web browser , access swagger endpoint

<http://localhost:8080/swagger-ui/index.html>



This is our json object

Open react frontend

Open app.js

In components in employee in line 45

In app.js

In components addemp

const create = (data) => {

return httpClient.post('', data);

};

Write in top down manner

Open controller

F

empService.deleteEmpDetails(empid);

Overloaded constructor

Initialize by super

Open emp controller

Method get

No method response but detached

Service call dao method findById

Argumenteless return throwabli instance

**Day 15**

Spring Boot's auto-configuration works like this:

1. \*\*Scan & Detect:\*\*

- Spring Boot scans your project for libraries and components.

- - It looks for specific classes, annotations, or properties that indicate the presence of certain technologies or functionalities.

When you run a Spring Boot application:

1. Spring scans for classes with special annotations.

- Spring Boot scans the classpath to identify classes annotated with `@Component`, `@Service`, `@Repository`, etc.

2. It sets up and configures beans automatically.

3. If it's a web app, an embedded server starts.

4. Properties in files like `application.properties` control settings.

5. Your main method kicks off the application. - The `main` method of the application is executed, triggering the Spring Boot application to start.

Dao run junit

Controller emp line 74

controller method for handling HTTP DELETE requests to delete employee details. It takes the employee ID (empId) from the URL, attempts to delete the corresponding employee details, and responds with a success message or an error message in a standardized format (ApiResponse). If successful, it returns an HTTP 200 OK status; otherwise, it returns a 404 NOT\_FOUND status with an error message.

method for handling HTTP DELETE requests at the specified URL (http://localhost:8080/employees/empId). Let's break down the code:

Employee fix so no {}

Empid changing so {}

- \*\*`@RequestMapping("/employees")`:\*\*

- Used at the class level to set a base URI for all methods inside the class.

- No curly braces because it's defining a base URI.

- \*\*`@DeleteMapping("/{empId}")`:\*\*

- Used at the method level to handle DELETE requests for a specific URI pattern.

- Curly braces `{}` indicate a path variable (`empId`) in the URI.

In summary, curly braces are used in `@DeleteMapping` to capture dynamic values (like an employee ID) from the URI, while `@RequestMapping` at the class level establishes a base URI without curly braces.

@DeleteMapping("/{empId}")

* this method will handle HTTP DELETE requests mapped to the specified URL pattern ("/{empId}"). The empId is a path variable extracted from the URI.

public ResponseEntity<?> deleteEmpDetails(@PathVariable Long empId)

* + This method is named deleteEmpDetails.
  + It takes a single parameter, empId, which is annotated with @PathVariable. grab specific values (like an employee ID) from the URL so that you can use them in your Java method as parametr.

System.out.println("in del emp dtls " + empId);

* This line simply prints a message to the console indicating that the method has been called with the specified empId.
* Try-Catch Block:

try {

return ResponseEntity.status(HttpStatus.OK).body(new ApiResponse(empService.deleteEmpDetails(empId)));

} catch (Exception e) {

System.out.println("err " + e);

return new ResponseEntity<>(new ApiResponse(e.getMessage()), HttpStatus.NOT\_FOUND);

}

* + The method attempts to execute the logic inside the try block.
  + If successful, it creates a ResponseEntity with an HTTP status of OK (200) and a body containing a new ApiResponse object. The ApiResponse object is constructed with the result of calling empService.deleteEmpDetails(empId).

This line of code is creating a response for a successful operation. It takes the result of deleting employee details using empService.deleteEmpDetails(empId), wraps it in a special response format called ApiResponse, and sets it as the body of the response.

* + If an exception is caught (e.g., empService.deleteEmpDetails throws an exception), it logs the error and returns a ResponseEntity with an HTTP status of NOT\_FOUND (404) and a body containing an ApiResponse with the error message.

- If the deletion of employee details is successful, it responds with a "Success" message and a status of 200 (OK).

In essence, this line of code is creating a response body using the result of deleting employee details, and it wraps this result in an `ApiResponse` object. This constructed response body will be sent back as part of a ResponseEntity, typically in the context of an HTTP response to a client.

1. \*\*Success Case:\*\*

- If everything goes well, it responds with a "Success" message and a status of 200 (OK).

- It constructs a special response object (`ApiResponse`) containing the result of deleting employee details.

2. \*\*Exception Case:\*\*

- If something unexpected happens (like an error during deletion), it catches the problem.

- It logs the error for developers to check.

- Responds with a status of 404 (Not Found) and an error message in a special response object (`ApiResponse`).

3. \*\*Diamond Operator:\*\*

- The `<>` (diamond operator) is used, allowing the compiler to figure out the exact type for the `ResponseEntity`. Diamond operator so compiler decide what type

Open deleteEmpDetails implementation bu ctrl click or in empservice

deletes employee details using the provided employee ID (empId). If the ID exists, it deletes the details and returns a success message. If the ID is invalid or the employee doesn't exist, it throws an exception with an error message.

1. \*\*`@Override` Annotation:\*\*

- Indicates that this method is intended to override a method from a superclass or interface.

2. \*\*Method Signature:\*\*

- The method is named `deleteEmpDetails`.

- It returns a `String`.

- It takes a single parameter, `empId`, representing the employee ID.

3. \*\*Method Implementation:\*\*

- Checks if an employee with the specified `empId` exists in the data source (presumably a database) using `empDao.existsById(empId)`.

- If the employee exists, it proceeds to delete the employee details with `empDao.deleteById(empId)`.

- Returns a success message, "Deleted emp details....", indicating that the deletion was successful.

4. \*\*Exception Handling:\*\*

- If the employee does not exist (invalid `empId`), it throws a custom exception (`ResourceNotFoundException`) with an error message: "Emp details can't be deleted: Invalid Emp Id!!!".

Employee service delteemp details

Api response

Controller line 51

If directly return list of entities json mapping exception because department cannot be serialized

Many to one fetching policy eager

One to may lazy so failed

If u say json ignore we cannot change pojo all time

So separation there are 2 types of pojo object entire(only for persistence layer(dao layer) and dto(to exchange data between rest client and rets server)

Serilization entity to dto deserilicati dto to entity

We want 3 paty lib so model mapper

Add jar in pom.xml

Open application class

In bean tag we can add many

Certainly, let's break it down:

1. \*\*Serialization Exception:\*\*

- When directly returning a list of entities in JSON format, there can be issues because certain properties, like a `Department` object in a many-to-one relationship, might lead to serialization problems.

- With a many-to-one relationship using eager fetching (loading related entities along with the main entity), serialization works well.

- However, in a one-to-many relationship with lazy fetching (loading related entities only when needed), it can cause serialization failures as the related entities might not be available when serializing.

3. \*\*JSON Ignore Not Ideal:\*\*

- Using `@JsonIgnore` on certain properties might solve the serialization problem, but it's not always ideal, especially if the same entity needs to be used in different contexts.

4. \*\*Separation with DTOs:\*\*

- To address this, a common practice is to separate entities (used in the persistence layer) and DTOs (Data Transfer Objects, used for exchanging data between client and server).

- Serialization of entities is then limited to the persistence layer, and DTOs are used for data exchange.

5. \*\*Serialization and Deserialization:\*\*

- The process of converting entities to DTOs during serialization and DTOs back to entities during deserialization is often needed.

6. \*\*ModelMapper:\*\*

- To simplify this process, a third-party library like ModelMapper is used.

- It allows for easy mapping between entity and DTO classes, reducing the need for manual coding.

7. \*\*Integration in Spring Boot:\*\*

- To use ModelMapper, you add its JAR dependency in the `pom.xml` file.

- In the application class (e.g., annotated with `@SpringBootApplication`), you can define a bean for ModelMapper, making it available for use throughout the application.

@Bean // equivalent to <bean id ..../> in xml file

public ModelMapper modelMapper() {

ModelMapper modelMapper = new ModelMapper();

modelMapper.getConfiguration()

.setMatchingStrategy(MatchingStrategies.STRICT) // only MATCHING prop names n data types

// between src n dest will be

// transferred , during the

// mapping

.setPropertyCondition(Conditions.isNotNull());// only non null properties will be transferred from src

// --> dest , during the mapping

return modelMapper;

This Java code, annotated with `@Bean`, configures and declares a `ModelMapper` bean for use in a Spring application. The `ModelMapper` is set to a strict matching strategy, ensuring that only properties with matching names and data types are transferred during mapping. Additionally, it's configured to transfer only non-null properties from the source to the destination during mapping. This bean can be injected into other parts of the Spring application.

This Java code snippet uses the `@Bean` annotation to declare a method named `modelMapper()` that returns a `ModelMapper` instance. This is a common usage in a Spring application to define a bean that can be managed by the Spring IoC (Inversion of Control) container. Let's break down the code:

```java

@Bean // equivalent to <bean id="modelMapper" .../> in XML file

public ModelMapper modelMapper() {

ModelMapper modelMapper = new ModelMapper();

modelMapper.getConfiguration()

.setMatchingStrategy(MatchingStrategies.STRICT) // Only matching prop names and data types between src and dest will be transferred during the mapping

.setPropertyCondition(Conditions.isNotNull()); // Only non-null properties will be transferred from src to dest during the mapping

return modelMapper;

}

```

1. \*\*`@Bean` Annotation:\*\*

- This annotation is used in a method-level to indicate that the method produces a bean to be managed by the Spring IoC container. It is equivalent to the `<bean>` element in an XML configuration file.

2. \*\*Method Signature:\*\*

- `public ModelMapper modelMapper()`: The method returns a `ModelMapper` instance.

- `ModelMapper` is a library used for mapping objects from one type to another.

3. \*\*Method Implementation:\*\*

- `ModelMapper modelMapper = new ModelMapper();`: Creates a new instance of `ModelMapper`.

- `modelMapper.getConfiguration()`: Gets the configuration of the `ModelMapper` instance.

4. \*\*Configuring ModelMapper:\*\*

- `.setMatchingStrategy(MatchingStrategies.STRICT)`: Sets the matching strategy to `STRICT`. This means only properties with matching names and data types between the source and destination objects will be transferred during the mapping.

- `.setPropertyCondition(Conditions.isNotNull())`: Sets a property condition to transfer only non-null properties from the source to the destination during mapping.

5. \*\*Return Statement:\*\*

- `return modelMapper;`: The configured `ModelMapper` instance is returned as the bean.

In summary, this code configures a `ModelMapper` bean with specific mapping strategies. The `@Bean` annotation declares this bean, making it available for dependency injection in other parts of the Spring application. The configured `ModelMapper` is particularly set to only transfer properties with matching names and data types (STRICT strategy) and to transfer only non-null properties during the mapping process.

Open controller

Certainly! This Java code represents a REST API endpoint for retrieving a list of all employees.

// REST API endpoint

// URL: http://localhost:8080/employees/

// Method: GET

// Response: List<EmployeeRespDTO>

@GetMapping

public List<EmployeeRespDTO> listAllEmps() {

System.out.println("in list all emps");

return empService.getAllEmps();

}

```

1. \*\*Mapping Annotation (`@GetMapping`):\*\*

- This annotation indicates that the method will handle HTTP GET requests.

- The absence of a specific path value means that it responds to GET requests at the base URL mapped to this controller, which is `/employees/`.

2. \*\*Method Signature:\*\*

- `public List<EmployeeRespDTO> listAllEmps()`: The method returns a list of `EmployeeRespDTO` objects, representing employee details.

- It takes no parameters.

3. \*\*Method Implementation:\*\*

- `System.out.println("in list all emps");`: This line prints a message to the console, indicating that the method is processing a request to list all employees.

- `return empService.getAllEmps();`: The method delegates the task of retrieving all employees to a service (`empService`). It then returns the list of employee details as the response.

4. \*\*Response Type:\*\*

- The response is a `List` of `EmployeeRespDTO` objects. Each `EmployeeRespDTO` likely contains information about an employee, such as their name, ID, and other relevant details.

In summary, when a GET request is made to `http://localhost:8080/employees/`, this endpoint prints a message, delegates the task of retrieving all employee details to a service (`empService`), and returns the list of employees as the response.

Get emp details ctrl click

.map(e -> mapper.map(e, EmployeeRespDTO.class)) // Stream<dto>

Done by reflection

We not use reflection but with exception of this method

Container call call getter and settre for matching

* + The method is named getAllEmps.
  + It returns a List of EmployeeRespDTO objects.
  + The @Override annotation indicates that this method overrides a method defined in a superclass or interface.
* Method Implementation:
  + empDao.findAll(): Fetches all employee entities from the data source (presumably a database) using empDao.
  + .stream(): Converts the list of entities to a stream.
  + .map(e -> mapper.map(e, EmployeeRespDTO.class)): Maps each employee entity (e) to a EmployeeRespDTO using a mapper (presumably mapper is an instance of some mapping framework or utility).
  + .collect(Collectors.toList()): Collects the mapped DTOs into a list.
* Return Statement:
  + The method returns a List of EmployeeRespDTO objects, representing the response DTOs for all employees.

In controller authenticate emp

Method post not get why

We want only email and password and not expose so ude dto

For validations add validation depcy in pom.xml

Ctr click on authreqestdto for rules

2 possibilities succes or failure

It gives a desrilizable pojo we get dto

If failure ctrl not com line 22 as sboot itself realize default validation failure but problem it gives full stack trace

This Java code is a Spring Boot controller method for handling employee authentication through an HTTP POST request at the URL `/employees/signin`. Here's a brief breakdown:

- \*\*URL and Method:\*\*

- URL: `http://localhost:8080/employees/signin`

- Method: POST

- \*\*Payload:\*\*

- Expects a JSON payload in the request body representing an employee authentication request (`AuthRequestDTO`).

- \*\*Response:\*\*

- In case of success, it returns a `ResponseEntity` with the authenticated employee details and an HTTP status of 200.

- In case of failure, it returns a `ResponseEntity` with an error message and an HTTP status of 404.

- \*\*Implementation:\*\*

- The method logs the incoming authentication request.

- Tries to authenticate the employee using a service layer method (`authenticateEmployee` in `empService`).

- If successful, returns a response with the authenticated employee details.

- If an exception occurs, logs the error and returns a response with an error message.

In essence, this endpoint allows employees to sign in by sending their credentials in the request body. The response includes either the authenticated employee details or an error message, along with the appropriate HTTP status codes.

This Java code is a Spring Boot controller method for handling HTTP POST requests at the specified URL (`http://localhost:8080/employees/signin`). Let's break it down:

```java

// URL: http://localhost:8080/employees/signin

// Method: POST

// Payload: emp req dto

// Response in case of success: ResponseEntity<emp resp dto>, SC 200

// Response in case of failure: ResponseEntity<api resp>, SC 404

@PostMapping("/signin")

public ResponseEntity<?> authenticateEmployee(@RequestBody @Valid AuthRequestDTO dto) {

System.out.println("in auth emp " + dto);

try {

// Invoke service layer method

return ResponseEntity.ok(empService.authenticateEmployee(dto));

} catch (Exception e) {

System.out.println("err " + e);

return ResponseEntity

.status(HttpStatus.NOT\_FOUND)

.body(new ApiResponse(e.getMessage()));

}

}

```

1. \*\*Mapping Annotation:\*\*

- `@PostMapping("/signin")`: Indicates that this method will handle HTTP POST requests at the specified URL ("/signin").

2. \*\*Method Signature:\*\*

- The method is named `authenticateEmployee`.

- It returns a `ResponseEntity<?>` to handle varied response types.

- It takes an `@RequestBody` annotated parameter of type `@Valid AuthRequestDTO dto`, representing the payload received in the POST request.

3. \*\*Method Implementation:\*\*

- `System.out.println("in auth emp " + dto);`: Prints a message to the console indicating that the method is processing an authentication request with the given payload.

- \*\*Try-Catch Block:\*\*

- `return ResponseEntity.ok(empService.authenticateEmployee(dto));`: If successful, invokes a service layer method (`authenticateEmployee` in `empService`) and returns a `ResponseEntity` with HTTP status 200 (OK) and the result of the service method as the body.

- `catch (Exception e) { ... }`: If an exception occurs during the authentication process, it catches the exception.

- `System.out.println("err " + e);`: Logs the error message to the console.

- `return ResponseEntity.status(HttpStatus.NOT\_FOUND).body(new ApiResponse(e.getMessage()));`: Returns a `ResponseEntity` with HTTP status 404 (NOT FOUND) and an `ApiResponse` object containing the error message as the body.

In summary, this code represents an endpoint for authenticating employees through a POST request. It takes an authentication request payload (`AuthRequestDTO`), processes it through the service layer, and returns a response with the authenticated employee details in case of success or an error message in case of failure. The response format is tailored to include the appropriate HTTP status codes (200 for success, 404 for failure).

Ctrl click authenticate employee

authenticates an employee by checking their email and password in a data source (empDao). If the authentication is successful, it maps the employee details to a response DTO (EmployeeRespDTO) and returns it. If no employee is found, it throws an exception indicating invalid credentials.

This Java code is a method annotated with `@Override`, indicating that it overrides a method from a superclass or interface. The method is part of a service layer and is responsible for authenticating an employee based on an authentication request (`AuthRequestDTO`). Here's a breakdown:

1. \*\*Method Signature:\*\*

- The method is named `authenticateEmployee`.

- It returns an `EmployeeRespDTO`, representing the response DTO for the authenticated employee.

- The `@Override` annotation indicates that this method overrides a method from a superclass or interface.

2. \*\*Method Implementation:\*\*

- `Employee emp = empDao.findByEmailAndPassword(request.getEmail(), request.getPassword())`: Invokes a method on a Data Access Object (DAO) (`empDao`). It attempts to find an employee by matching the provided email and password. If no employee is found, it throws a `ResourceNotFoundException` with an error message.

- `return mapper.map(emp, EmployeeRespDTO.class)`: If an employee is found, the method uses a mapper (presumably `mapper`) to map the employee entity to an `EmployeeRespDTO` (response DTO). The mapped DTO is then returned.

In summary, this code authenticates an employee by querying a data source (`empDao`) based on the provided email and password. If the authentication is successful, it maps the resulting employee entity to a response DTO (`EmployeeRespDTO`) using a mapper and returns it. If no employee is found, it throws a `ResourceNotFoundException` with an appropriate error message.

Run it on swagger-play with email and password

Prob prinat all satck trae

So centrailzed exception handler

Make pkg com.app.exc\_handler

This Java class, annotated with `@RestControllerAdvice`, serves as a global exception handler for a Spring application. It intercepts and handles various types of exceptions that may occur during the execution of controller methods. Here's a breakdown:

This Java class, annotated with `@RestControllerAdvice`, is a global exception handler for a Spring application. It does the following:

1. \*\*Validation Errors:\*\*

- Handles validation errors during method argument validation (`MethodArgumentNotValidException`).

- Returns a Bad Request status along with a map of field errors.

2. \*\*Resource Not Found:\*\*

- Handles resource not found exceptions (`ResourceNotFoundException`).

- Logs the error, responds with a Not Found status, and returns a custom `ApiResponse` with the error message.

3. \*\*Catch-All Exception Handling:\*\*

- Catches any other unhandled runtime exceptions.

- Logs the error, responds with an Internal Server Error status, and returns a custom `ApiResponse` with the error message.

This class centralizes error handling logic, providing consistent responses for various exception scenarios across all controllers in the application.

1. \*\*Class Annotation (`@RestControllerAdvice`):\*\*

- mandatory annotation

@RestControllerAdvice annotation marks this class as a global exception handler that works across all controllers. It combines the features of @ControllerAdvice and @ResponseBody, making it suitable for handling exceptions globally in a monolithic application.

- It combines the functionality of `@ControllerAdvice` for standalone monolithic and `@ResponseBody`. Added on exception handling method

Who is advicinh whom regarding what

This global class is telling all controller u dont write exception code i will write it’ udont call we will call u

Dont try catch block dont write sam e code 10 times

Rest control not sufficient

How to tell what is catch block

Visuilice catch block i controller

Can we have 1 try block with multiple catch bock yes

@exception handler is catch block

Its arg is exception .class

2. \*\*Method for Handling MethodArgumentNotValidException:\*\*

- `handleMethodArgumentNotValidException`: Handles exceptions of type `MethodArgumentNotValidException`, which occur during method argument validation.

- It extracts field errors, creates a map of field names to error messages, and responds with a Bad Request status along with the map of field errors.

3. \*\*Method for Handling ResourceNotFoundException:\*\*

- `handleResourceNotFoundException`: Handles exceptions of type `ResourceNotFoundException`.

- It logs the error, responds with a Not Found status, and returns a custom `ApiResponse` containing the error message.

4. \*\*Method for Handling Any Other Runtime Exceptions (Catch-All):\*\*

- `handleAnyException`: Catches any other unhandled runtime exceptions.

- It logs the error, responds with an Internal Server Error status, and returns a custom `ApiResponse` containing the error message.

In summary, this class acts as a central point to handle specific exceptions globally across all controllers in a Spring application. It provides custom responses based on the type of exception encountered, improving error handling and providing a consistent API response format.

This Java code is part of an exception handling method within a `@RestControllerAdvice` class, and it specifically handles instances of `MethodArgumentNotValidException`. Let's break it down:

1. \*\*Logging:\*\*

- `System.out.println("in method arg invalid123 " + e);`: This line prints a message to the console indicating that a `MethodArgumentNotValidException` has been caught.

2. \*\*Extracting Field Errors:\*\*

- `List<FieldError> fieldErrors = e.getFieldErrors();`: Retrieves a list of `FieldError` objects from the exception. Each `FieldError` represents a validation error on a specific field.

3. \*\*Mapping to a Map:\*\*

- `Map<String, String> map = fieldErrors.stream().collect(Collectors.toMap(FieldError::getField, FieldError::getDefaultMessage));`:

Converts the list of `FieldError` objects into a `Map` where the field name is the key, and the validation error message is the value. This mapping helps in creating a concise representation of validation errors.

Collectors to map is higher order so need key and value

Returns affected field

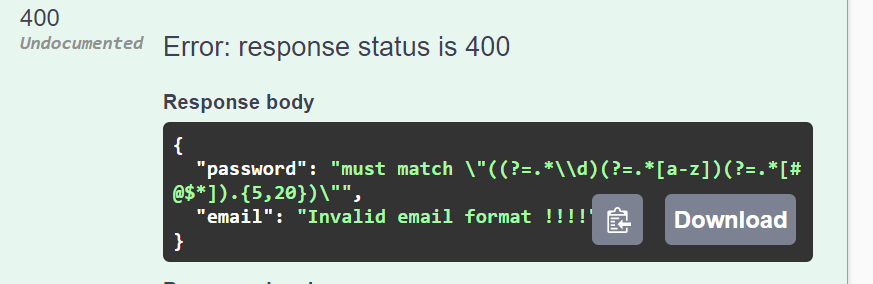
This map going to contain map of (k=filed name (email,pass) value+err msg

Set map in body

4. \*\*ResponseEntity:\*\*

- `return ResponseEntity.status(HttpStatus.BAD\_REQUEST).body(map);`: Constructs and returns a `ResponseEntity` with an HTTP status of Bad Request (400) and the created `Map` of field errors as the response body. This is a standard way of indicating a client-side error due to invalid input.

In summary, this code is part of a global exception handling mechanism for handling validation errors. It logs the exception, extracts validation errors (field errors), converts them into a map, and returns a Bad Request response with the map of field errors. This approach provides a clear and structured way to communicate validation errors in a RESTful API response.



In line 45 catch all

This Java code is an exception handling method within a `@RestControllerAdvice` class, specifically designed to handle any uncaught runtime exceptions.

This Java code is an exception handling method for catching any unhandled runtime exceptions. It logs the exception, sets the HTTP status to Internal Server Error, and returns an `ApiResponse` with the exception message. This provides a consistent way to handle unexpected server-side errors in a Spring application.

1. \*\*Annotation (`@ExceptionHandler`):\*\*

- `@ExceptionHandler(RuntimeException.class)`: This annotation indicates that this method handles exceptions of type `RuntimeException`. It serves as a catch-all for any runtime exceptions that might not be explicitly handled elsewhere.

2. \*\*HTTP Status (`@ResponseStatus`):\*\*

- `@ResponseStatus(value = HttpStatus.INTERNAL\_SERVER\_ERROR)`: Specifies that, in case of an exception being caught by this method, the HTTP status of the response should be set to Internal Server Error (500). This status is commonly used for unexpected server-side errors.

3. \*\*Exception Handling Logic:\*\*

- `public ApiResponse handleAnyException(RuntimeException e) {`: The method takes a `RuntimeException` as its parameter.

- `System.out.println("in catch-all " + e);`: Logs the exception message to the console, indicating that this is a catch-all exception handling block.

- `return new ApiResponse(e.getMessage());`: Constructs a new `ApiResponse` object with the exception message and returns it. The `ApiResponse` class is presumably a custom class for representing API responses.

In summary, this code defines a catch-all exception handling method that handles any uncaught runtime exceptions. It logs the exception and responds with an Internal Server Error status along with an `ApiResponse` containing the exception message. This approach ensures a consistent response format for unexpected server-side errors across the application.

In desc emps in db

image | longblob | YES | | NULL | |

| image\_path | varchar(255) | YES | | NULL

* @RestController:
  + This annotation combines @Controller and @ResponseBody. It indicates that the class is a controller and that each method returns the response body directly (rather than relying on a view resolver). It's typically used for building RESTful APIs.
* @RequestMapping("/employees"):
  + Specifies the base URL path ("/employees") for all the methods in this controller. For example, a method mapped to "/getDetails" in this class would be accessible at "/employees/getDetails".
* @CrossOrigin(origins = "http://localhost:3000"):
  + Allows cross-origin resource sharing (CORS) from the specified origin ("[http://localhost:3000](http://localhost:3000/)"). This annotation is commonly used when your frontend (client-side) and backend (server-side) are hosted on different domains.
* Dependencies (Autowired):
  + @Autowired is used to inject instances of EmployeeService and ImageHandlingService into the controller. These services are presumably defined elsewhere in the application and handle business logic related to employees and image handling, respectively.
  + @Qualifier("image\_db") is used to specify which bean should be injected when there are multiple candidates of the same type. In this case, it injects the bean with the name "image\_db".

This by type or by name

This find it under com.app package

Wherethere is main class

Under which it search for (is there any spring bean with id not type as by name )it service layers bean

In service image impl

Now qualifier is match with service tag

@post construct call it as applc deployment time adsingleton and eager

Certainly! This code is using the `@PostConstruct` annotation to create an initialization method named `init()` for a Spring bean. Here's a step-by-step explanation:

1. \*\*Annotation:\*\*

- `@PostConstruct`: Marks the `init()` method to be executed after the bean is constructed and its dependencies are injected.

2. \*\*Initialization Method:\*\*

- `public void init() { ... }`: The initialization method.

3. \*\*Console Output:\*\*

- `System.out.println("in init " + folderLocation);`: Prints a message to the console, indicating the start of the initialization and displaying the value of the `folderLocation` variable.

4. \*\*Check if Folder Exists:\*\*

- `File folder = new File(folderLocation);`: Creates a `File` object representing the directory specified by `folderLocation`.

- `if (folder.exists()) { ... }`: Checks if the folder already exists.

5. \*\*Folder Existence Handling:\*\*

- If the folder exists, it prints a message indicating that the folder already exists.

- If the folder does not exist:

- `folder.mkdir();`: Creates the folder.

- `System.out.println("created a folder!");`: Prints a message indicating that a folder has been created.

In summary, the `init()` method is automatically called by the Spring container after the bean is created. It checks if a specified folder exists and creates it if it doesn't, providing a simple initialization mechanism.

The `@PostConstruct` annotation is used in Java to mark a method that should be executed after a bean has been initialized. In the context of a Spring application, this method will be invoked by the Spring container immediately after the bean has been constructed and all its dependencies have been injected. Here's an explanation of the code snippet:

1. \*\*`@PostConstruct` Annotation:\*\*

- This annotation marks the `init()` method as a method that should be executed after the bean is constructed and its dependencies are injected. In the context of this code, it's used to perform initialization tasks for the bean.

2. \*\*Initialization Logic:\*\*

- `System.out.println("in init " + folderLocation);`: Prints a message to the console, indicating the initiation of the initialization process and showing the value of the `folderLocation` variable.

- `File folder = new File(folderLocation);`: Creates a `File` object representing the directory specified by `folderLocation`.

- `if (folder.exists()) { ... }`: Checks if the folder already exists.

- If the folder exists, it prints a message indicating that the folder already exists.

- If the folder does not exist, it creates the folder using `folder.mkdir()` and prints a message indicating that a folder has been created.

In summary, the `@PostConstruct`-annotated `init()` method is responsible for checking whether a specified folder exists. If the folder exists, it prints a message; if not, it creates the folder and logs a corresponding message. This method is automatically triggered by the Spring container after the bean has been initialized.

* Constructor:
  + The constructor prints a message to the console when an instance of the EmployeeController class is created. This can be useful for debugging and understanding the lifecycle of the controller.

In summary, this EmployeeController class is designed for handling HTTP requests related to employees, likely in the context of building a RESTful API. It has dependencies on EmployeeService and ImageHandlingService, and it allows cross-origin requests from "[http://localhost:3000](http://localhost:3000/)".

This code uses the `@Value` annotation in Spring to inject the value of the property named `folder.location` from the application property file into a field named `folderLocation`. This allows the configuration value to be easily accessed in the code. The `${folder.location}` syntax is part of Spring Expression Language (SpEL), providing a way to dynamically evaluate property values.

This code snippet is a part of a Spring component (likely a bean or a class managed by the Spring framework), and it uses the `@Value` annotation to inject the value of the property named `folder.location` from the application property file. The injection is done using Field Dependency Injection (Field DI) and Spring Expression Language (SpEL). Let's break it down:

```java

@Value("${folder.location}")

private String folderLocation;

```

1. \*\*`@Value("${folder.location}")` Annotation:\*\*

- `@Value`: This Spring annotation is used to inject values into fields, methods, or constructor parameters.

- `"${folder.location}"`: Specifies the property key for which the value needs to be injected. In this case, it's `folder.location`.

2. \*\*Field DI:\*\*

- `private String folderLocation;`: This private field is the target for injecting the property value. The type of the field (`String` in this case) should match the type of the property in the application property file.

3. \*\*SpEL (Spring Expression Language):\*\*

- The use of `${...}` within the `@Value` annotation is an example of SpEL. SpEL allows you to use expressions to dynamically evaluate values during the application context initialization.

4. \*\*Property Injection:\*\*

- During the Spring container initialization, the value associated with the property key `folder.location` in the application property file is injected into the `folderLocation` field.

In summary, this code uses the `@Value` annotation with SpEL to inject the value of the `folder.location` property from the application property file into the `folderLocation` field. This is a way to externalize configuration in Spring applications, allowing properties to be easily changed without modifying the source code.

In res->applprop lat 2 lines

key=folder.location

#custom property to specify location of images folder realtive to curnt project

folder.location=images/

The custom property `folder.location` is set to `images/` in the application properties file. This property specifies the location of the images folder relative to the current project, indicating that the images folder is expected to be at the root level of the project.

This custom property `folder.location` is specified in the application properties file and is set to the value `images/`. It is used to indicate the location of the images folder relative to the current project. This property can be injected into Spring components using the `@Value` annotation for further configuration and customization. The value `images/` suggests that the images folder is expected to be at the root level of the project.

This location injected in @ values in imagehandlinimpl

In swagger post image empid

Give id=1,

Any time u upload large content

o/p 201 -created

Workbench

Use select

Swagger url in browser

In controller image\_db to image\_folder

In db

Now images folder cretsed in sts

In controller

Multipart HTTP requests are a way to send files or complex data in a single HTTP request. They use `enctype="multipart/form-data"` and divide the data into parts, each with its headers and content. Parts are separated by a boundary string. This is often used in HTML forms with file uploads, where each file or form input is sent as a separate part. The server needs to parse each part individually upon receiving the request.

Multipart HTTP requests use `enctype="multipart/form-data"` to send binary or text data over HTTP. They are flexible and allow complex data, like files or form inputs, to be sent in a single request.

Key points:

1. \*\*Multiple Parts:\*\*

- Multipart requests allow multiple sections or "parts" within a single HTTP request.

2. \*\*Boundary String:\*\*

- Parts are separated by a boundary string specified in the request headers.

3. \*\*Content Structure:\*\*

- Each part has headers (content type, content disposition, metadata) followed by the actual content.

4. \*\*File Uploads:\*\*

- For file uploads, binary data is included in one of the parts.

5. \*\*Common Usage:\*\*

- Commonly used in HTML forms with file uploads, where the form's enctype is set to "multipart/form-data."

6. \*\*Server Processing:\*\*

- When the server receives a multipart request, it needs to parse each part individually.

Spring upload content and put in multipart file

And inject in method that can be use to process data

In controller line 130

1. \*\*Annotation Explanation:\*\*

- `@PostMapping(value = "/images/{empId}", consumes = "multipart/form-data")`: This method is mapped to HTTP POST requests at the URL path "/images/{empId}". It consumes data in "multipart/form-data" format, which is typical for file uploads.

2. \*\*Method Parameters:\*\*

- `@PathVariable Long empId`: Extracts the employee ID from the URL path.

- `@RequestParam MultipartFile imageFile`: Retrieves the uploaded image file from the request.

Upload file and give it to u in multi[art file

2 edy askes one is multipart file

Our job is to store i tin either server side or server side database

`MultipartFile` is a Spring interface representing a file or form item received in a multipart request. It's commonly used in Spring applications, especially in the context of file uploads. Key points:

1. \*\*Definition:\*\*

- `MultipartFile` is part of the Spring framework's `org.springframework.web.multipart` package.

2. \*\*Purpose:\*\*

- It encapsulates the uploaded file or form data received in a multipart request.

3. \*\*Usage:\*\*

- Typically used as a method parameter in controller methods handling file uploads.

4. \*\*Accessing Data:\*\*

- Provides methods to access the file's content, name, size, content type, etc.

5. \*\*Common Methods:\*\*

- Common methods include `getOriginalFilename()`, `getBytes()`, `getInputStream()`, `getSize()`, and others.

6. \*\*Integration with Spring MVC:\*\*

- Often used with Spring MVC's `@RequestParam` annotation to extract file data from a multipart request.

In summary, `MultipartFile` simplifies the handling of uploaded files in Spring applications, offering convenient methods to access various attributes of the uploaded file.

3. \*\*Method Logic:\*\*

- `System.out.println("in upload img " + empId);`: Logs a message indicating that the image upload process has started.

- `imgService.uploadImage(empId, imageFile)`: Delegates the image upload logic to the `imgService` (presumably an `ImageHandlingService`), passing the employee ID and the uploaded image file. The result is returned.

4. \*\*Response Handling:\*\*

- `return ResponseEntity.status(HttpStatus.CREATED).body(...)`: Constructs a response with a status of "Created" (201) and includes the result of the image upload operation in the response body.

In summary, this method handles the upload of an image for a specific employee. It extracts the employee ID and the image file from the request, delegates the image upload logic to a service (`imgService`), and responds with the outcome of the operation. The `consumes = "multipart/form-data"` indicates that the request is expected to contain multipart data, such as an uploaded image file.

1. \*\*Annotation Explanation:\*\*

- `@PostMapping(value = "/images/{empId}", consumes = "multipart/form-data")`: This method is mapped to HTTP POST requests at the URL path "/images/{empId}". It consumes data in "multipart/form-data" format, which is typical for file uploads.

2. \*\*Method Parameters:\*\*

- `@PathVariable Long empId`: Extracts the employee ID from the URL path.

- `@RequestParam MultipartFile imageFile`: Retrieves the uploaded image file from the request.

3. \*\*Method Logic:\*\*

- `System.out.println("in upload img " + empId);`: Logs a message indicating that the image upload process has started.

- `imgService.uploadImage(empId, imageFile)`: Delegates the image upload logic to the `imgService` (presumably an `ImageHandlingService`), passing the employee ID and the uploaded image file. The result is returned.

4. \*\*Response Handling:\*\*

- `return ResponseEntity.status(HttpStatus.CREATED).body(...)`: Constructs a response with a status of "Created" (201) and includes the result of the image upload operation in the response body.

In summary, this method handles the upload of an image for a specific employee. It extracts the employee ID and the image file from the request, delegates the image upload logic to a service (`imgService`), and responds with the outcome of the operation. The `consumes = "multipart/form-data"` indicates that the request is expected to contain multipart data, such as an uploaded image file.

In service imahe handling upload image

Certainly! This method handles the upload of an image for a specific employee:

1. \*\*Retrieve Employee:\*\*

- Fetches the employee from the repository based on the provided employee ID.

2. \*\*Store Image on Server:\*\*

- Constructs the file path for storing the image on the server side.

- Writes the byte array of the image to a file on the server using Apache Commons `FileUtils`.

3. \*\*Update Employee Information:\*\*

- Sets the image path for the employee. Alternatively, it provides an option to store the image directly in the database as a BLOB.

4. \*\*Response:\*\*

- Returns a response indicating the successful upload of the image for the specified employee.

In summary, the method ensures the image is stored either on the server side or in the database, updates the employee's information, and responds with a success message.

* Retrieve Employee:
  + Employee emp = empRepo.findById(empId).orElseThrow(() -> new ResourceNotFoundException("Invalid emp ID!!!!"));: Retrieves the employee with the given ID from the repository. Throws an exception if not found.
* Store Image on Server:
  + String path = folderLocation.concat(image.getOriginalFilename());: Constructs the path for storing the image on the server side folder. The folderLocation is likely a property specifying the location.
  + FileUtils.writeByteArrayToFile(new File(path), image.getBytes());: Writes the byte array of the image to a file on the server side using the Apache Commons FileUtils utility.
* Update Employee Information:
  + emp.setImagePath(path);: Sets the image path for the employee.
  + Alternatively (// OR block), you can store the image directly in the database as a BLOB (Binary Large Object) using emp.setImage(image.getBytes());. This depends on the preferred storage mechanism.
* Response:
  + return new ApiResponse("Image file uploaded successfully for emp id " + empId);: Creates a response indicating the successful upload of the image for the specified employee.

In summary, this method fetches the employee, saves the uploaded image to a server-side folder, updates the employee's information with the image path, and responds with a success message. It provides options for both storing the image on the server and storing it directly in the database.

Now download image

Certainly! This method is for downloading an image associated with a specific employee:

1. \*\*Retrieve Employee:\*\*

- Fetches the employee from the repository based on the provided employee ID.

2. \*\*Retrieve Image Path:\*\*

- Gets the image path from the employee entity.

3. \*\*Download Image:\*\*

- Checks if the image path is not null.

- Reads the byte array of the image from the file specified by the path using Apache Commons `FileUtils`.

- Alternatively, you can retrieve the image directly from the database.

4. \*\*Exception Handling:\*\*

- If the image path is null, it throws an exception indicating that the image is not yet assigned.

In summary, the method fetches the employee, retrieves the image, and returns the byte array of the image. If the image is not assigned, it throws an exception.

* Retrieve Employee:
  + Employee emp = empRepo.findById(empId).orElseThrow(() -> new ResourceNotFoundException("Invalid emp ID!!!!"));: Retrieves the employee with the given ID from the repository. Throws an exception if not found.
* Retrieve Image Path:
  + String path = emp.getImagePath();: Gets the image path from the employee entity.
* Download Image:
  + if (path != null) { ... }: Checks if the image path is not null.
  + return FileUtils.readFileToByteArray(new File(path));: Reads the byte array of the image from the file specified by the path using Apache Commons FileUtils.
  + Alternatively (// OR block), you can retrieve the image directly from the database using return emp.getImage();.
* Exception Handling:
  + If the image path is null, it throws an exception indicating that the image is not yet assigned.

In summary, this method fetches the employee, retrieves the image path, and returns the byte array of the image either from a file on the server or directly from the database. If the image path is not assigned, it throws an exception.

Service layer returning byte array to caller

This is for uploadig dowmlading single file

Pagination

Pagination is a technique used to break down a large set of data into smaller, manageable chunks or pages. Here's a short and easy explanation:

1. \*\*Definition:\*\*

- \*\*What is it?\*\* Pagination is a way to display and navigate through large sets of data by dividing it into smaller sections or pages.

2. \*\*How it Works:\*\*

- \*\*Division of Data:\*\* Instead of showing all data at once, it is divided into pages, each containing a limited number of items.

- \*\*Navigation:\*\* Users can navigate through pages using buttons like "Next" and "Previous" or by selecting a specific page number.

3. \*\*Benefits:\*\*

- \*\*Improved Performance:\*\* Reduces the load on servers and improves the performance of web applications by fetching and displaying only a subset of data at a time.

- \*\*User-Friendly:\*\* Enhances user experience by presenting information in manageable portions.

4. \*\*Implementation:\*\*

- \*\*Backend:\*\* In web applications, pagination is often implemented on the server-side. APIs return a specific number of items per request.

- \*\*Frontend:\*\* The frontend displays the data and provides navigation controls based on the pages received from the server.

In summary, pagination is a technique that makes it easier to handle and present large amounts of data by breaking it into smaller, more manageable pages. Users can navigate through these pages to find the information they need.

In employee controller

* Mapping:
  + @GetMapping("/paginate"): This method is mapped to handle HTTP GET requests at the path "/paginate".
* This method is a Spring MVC controller endpoint for retrieving a paginated list of employees. Key points:
* 1. \*\*Parameters:\*\*
* - `pageNumber` and `pageSize` are optional parameters in the URL, specifying the page number and size.
* 2. \*\*Logging:\*\*
* - Prints the received `pageNumber` and `pageSize` for debugging.
* 3. \*\*Service Call:\*\*
* - Invokes a service method (`getAllEmployees`) to get a paginated list based on the provided parameters.
* 4. \*\*Response Handling:\*\*
* - If the list is empty, responds with a "No Content" status.
* - If employees are found, responds with a "OK" status and includes the list in the response body.
* In summary, it facilitates paginated retrieval of employee data, and the response varies based on whether data is present or not.
* This method is a Spring MVC controller endpoint designed for retrieving a paginated list of employees. Let's break down its key components:
* 1. \*\*Method Signature:\*\*
* - `public ResponseEntity<?> getAllEmpsPaginated(...)`: The method returns a `ResponseEntity` encapsulating the response to the client. The wildcard (`?`) in the generic type allows flexibility in the response body.
* 2. \*\*Request Parameters:\*\*
* - `@RequestParam(defaultValue = "0", required = false) int pageNumber`: This annotation indicates that the `pageNumber` parameter is expected as a request parameter. The default value is set to 0, and the parameter is optional.
* - `@RequestParam(defaultValue = "3", required = false) int pageSize`: Similar to `pageNumber`, this parameter is for the page size, with a default value of 3 and is optional.
* 3. \*\*Method Logic:\*\*
* - `System.out.println("in get all emps " + pageNumber + " " + pageSize);`: Logs the values of `pageNumber` and `pageSize`. This is for debugging or logging purposes.
* - `List<EmployeeRespDTO> list = empService.getAllEmployees(pageNumber, pageSize);`: Invokes a service method (`getAllEmployees`) to retrieve a paginated list of employees based on the provided `pageNumber` and `pageSize`.
* - `if (list.isEmpty()) return ResponseEntity.status(HttpStatus.NO\_CONTENT).build();`: If the retrieved list is empty, the method responds with a status of "No Content" (204). This indicates that there is no data to return.
* - `return ResponseEntity.ok(list);`: If employees are found, the method responds with a status of "OK" (200) and includes the list of employees in the response body.
* In summary, this controller method allows clients to request a paginated list of employees by specifying the page number and page size as optional parameters. The method communicates with a service (`empService`) to retrieve the data and constructs an appropriate response using the `ResponseEntity` wrapper.
* Request Parameters:
  + @RequestParam(defaultValue = "0", required = false) int pageNumber: Defines a request parameter pageNumber with a default value of 0 and is optional.
  + @RequestParam(defaultValue = "3", required = false) int pageSize: Defines a request parameter pageSize with a default value of 3 and is optional.
* Method Logic:
  + System.out.println("in get all emps " + pageNumber + " " + pageSize);: Logs the values of pageNumber and pageSize received in the request.
  + List<EmployeeRespDTO> list = empService.getAllEmployees(pageNumber, pageSize);: Retrieves a paginated list of employees from the service based on the provided page number and page size.
  + if (list.isEmpty()) return ResponseEntity.status(HttpStatus.NO\_CONTENT).build();: If the list is empty, responds with a status of "No Content" (204).
  + return ResponseEntity.ok(list);: If employees are found, responds with a status of "OK" (200) and includes the list in the response body.

In summary, this method allows clients to retrieve a paginated list of employees by specifying the page number and page size as optional request parameters. The retrieved list is then returned in the response.

Crtl click getallemp implementation

Certainly! Let's break down the method step by step:

1. \*\*Pageable Creation:\*\*

- `Pageable pageable = PageRequest.of(pageNumber, pageSize);`: Creates a `Pageable` object, specifying the desired page number and page size.

2. \*\*Data Retrieval:\*\*

- `empDao.findAll(pageable)`: Retrieves a page of employees from the data access layer (`empDao`) based on the created `Pageable`.

- `.getContent()`: Extracts the content (list of employees) from the retrieved page.

3. \*\*DTO Mapping:\*\*

- `empList.stream().map(emp -> mapper.map(emp, EmployeeRespDTO.class)).collect(Collectors.toList());`: Maps each `Employee` in the list to a corresponding `EmployeeRespDTO` using a `ModelMapper`.

4. \*\*Return:\*\*

- Returns the list of `EmployeeRespDTO` objects.

In summary, the method creates a `Pageable` object for pagination, retrieves a page of employees from the data layer, maps them to DTOs using a `ModelMapper`, and returns the list of DTOs. This enables paginated retrieval of employee data with mapped DTOs.

* Pageable Creation:
  + Pageable pageable = PageRequest.of(pageNumber, pageSize);: Creates a Pageable object, which is an interface for pagination, specifying the desired page number and page size.
* Data Retrieval:
  + List<Employee> empList = empDao.findAll(pageable).getContent();: Retrieves a page of employees from the data access layer (empDao). The getContent() method extracts the content (list of employees) from the retrieved page.
* DTO Mapping:
  + empList.stream().map(emp -> mapper.map(emp, EmployeeRespDTO.class)).collect(Collectors.toList());: Maps each Employee in the list to a corresponding EmployeeRespDTO using a ModelMapper. The result is a List of EmployeeRespDTO objects.
* Return:
  + Returns the list of EmployeeRespDTO objects.

In summary, this method utilizes Spring Data JPA's Pageable and Page concepts to fetch a page of employees from the data layer, and then it maps the retrieved employees to a list of DTOs using a ModelMapper. The paginated list of DTOs is then returned.

Swagger

Get paginate

i/P 1,2

Certainly! Below is an example of how you might implement the described functionality using Spring MVC:

```java

@PostMapping("/employees")

public ResponseEntity<?> addEmployeeToDepartment(@RequestBody AddEmpDTO addEmpDTO) {

try {

// Server-side validation: Check if the provided department ID is valid

if (!isValidDepartment(addEmpDTO.getDeptId())) {

return ResponseEntity.status(HttpStatus.BAD\_REQUEST)

.body(new ApiResponse("Invalid department ID"));

}

// Perform other server-side validations as needed

// If validations pass, proceed to add the employee to the department

Employee newEmployee = createEmployee(addEmpDTO);

// Save the new employee to the database (assumed method)

empService.saveEmployee(newEmployee);

// Respond with success (201) and the details of the newly added employee

return ResponseEntity.status(HttpStatus.CREATED).body(newEmployee);

} catch (Exception e) {

// Respond with failure (500) and an error message

return ResponseEntity.status(HttpStatus.INTERNAL\_SERVER\_ERROR)

.body(new ApiResponse("Error adding employee to department"));

}

}

private boolean isValidDepartment(Long deptId) {

// Implement logic to check if the department ID is valid

// This could involve querying the database or checking against a predefined list

// Return true if valid, false otherwise

return true;

}

private Employee createEmployee(AddEmpDTO addEmpDTO) {

// Create a new Employee object based on the AddEmpDTO

// You might want to set other properties like name, salary, etc. as needed

Employee newEmployee = new Employee();

newEmployee.setDeptId(addEmpDTO.getDeptId());

// Perform other necessary mappings

return newEmployee;

}

```

Explanation:

Open employee controller

U already have method simplestandalone insertion addempdetail

Build in that

No change in url

Post mapping

@requestbody@valid

Dont need id as gen by hibernate

Basic details plus dept id without foreign key

Dept id property name should be deptid or id

Validation rules on dto

First name hast to be not empty

@length for varchar

Controller call service layer method

We get employee request dto

From dept id we get dept persistence id

But it in dept dai

So need to inject it

By findById orElseThroe

Call addEmpMethod

No Call empdao.save as cascading

When u make changes auto insert

Put api response controller to rest client

Status code 201

If failure email already taken

1. \*\*Endpoint:\*\*

- `@PostMapping("/employees")`: This method is mapped to handle HTTP POST requests at the path "/employees".

2. \*\*Request Payload:\*\*

- `@RequestBody AddEmpDTO addEmpDTO`: Retrieves the payload containing information to add a new employee, including the department ID.

3. \*\*Server-side Validation:\*\*

- `isValidDepartment(addEmpDTO.getDeptId())`: Performs server-side validation to check if the provided department ID is valid. You may implement this logic based on your application's requirements.

4. \*\*Error Handling:\*\*

- If validations fail or an exception occurs, the method returns a response with an appropriate status code and an error message.

5. \*\*Adding Employee:\*\*

- `Employee newEmployee = createEmployee(addEmpDTO);`: Creates a new `Employee` object based on the provided DTO.

- `empService.saveEmployee(newEmployee);`: Saves the new employee to the database (this method is assumed and may vary based on your implementation).

6. \*\*Response:\*\*

- If everything is successful, the method responds with a status of 201 (Created) and includes the details of the newly added employee.

- If there's an error, it responds with a status of 500 (Internal Server Error) and an appropriate error message.