**@SpringBootApplication**

Spring Boot @SpringBootApplication annotation is used **to mark a configuration class that declares one or more @Bean methods and also triggers auto-configuration and component scanning**. It's same as declaring a class with @Configuration, @EnableAutoConfiguration and @ComponentScan annotations.

**@Bean is just for the metadata definition to create the bean**(equivalent to tag). @Autowired is to inject the dependancy into a bean(equivalent to ref XML tag/attribute).

**@ComponentScan**

Enables component scanning of current package Also recursively scans sub-packages

If we want to scan other packages explicitly

package com.luv2code.springboot.demo.mycoolapp; … @SpringBootApplication( scanBasePackages={"com.luv2code.springboot.demo.mycoolapp", "org.acme.iot.utils", "edu.cmu.wean"}) public class MycoolappApplication { … }

SpringBootServletInitializer?

The binding is done with @GetMapping . The Application sets up the Spring Boot application. It extends from SpringBootServletInitializer so that it can be deployed as a WAR. The application can be run both by deploying the WAR on a Tomcat server and executing it as a self-executable web archive with embedded Tomcat.

**SpringApplicationBuilder**

Builder for [SpringApplication](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/SpringApplication.html" \o "class in org.springframework.boot) and [ApplicationContext](https://docs.spring.io/spring-framework/docs/6.0.2/javadoc-api/org/springframework/context/ApplicationContext.html" \o "class or interface in org.springframework.context) instances with convenient fluent API and context hierarchy support. Simple example of a context hierarchy:

**WAR vs Jar**

The first and most obvious difference is the **file extension**. JARs have the .jar extension, whereas the WAR file has the .war extension.

The second main difference is **their purpose and the way they function**. JAR files allow us to package multiple files in order to use it as a library, plugin, or any kind of application. On the other hand, WAR files are used only for web applications.

**The structure of the archives is also different.** We can create a JAR with any desired structure. In contrast, WAR has a predefined structure with WEB-INF and META-INF directories.

Finally, we can **run a JAR from the command line** if we build it as an [executable JAR](https://www.baeldung.com/executable-jar-with-maven) without using additional software. Or, we can use it as a library. In contrast, we **need a server to execute a WAR**.

@Repository

//@Repository Annotation is a specialization of @Component annotation which is used to indicate that the class provides the mechanism for storage, retrieval, update, delete and search operation on objects.

**DAO**

DAO stands for **data access object**. Usually, the DAO class is responsible for two concepts: encapsulating the details of the persistence layer and providing a CRUD interface for a single entity.

@Table

The @Table annotation **allows you to specify the details of the table that will be used to persist the entity in the database**.

@Entity

Entities in JPA are nothing but **POJOs representing data that can be persisted to the database**

**@SuppressWarnings is to suppress or ignore warnings coming from the compiler**

**@RestControllerAdvice**In addition, we can just understand it as:

@RestControler = @Controller + @ResponseBody

@RestControllerAdvice = @ControllerAdvice + @ResponseBody.

Keeping in mind that @RestControllerAdvice is more convenient annotation for handling Exception with RestfulApi.

**@RestControllerAdvice** is the combination of both**@ControllerAdvice** and **@ResponseBody**. We can use the **@ControllerAdvice** annotation for handling exceptions in the [**RESTful Services**](https://javainterviewpoint.com/spring-restful-web-services-crud-example/) but we need to add **@ResponseBody** separately.

@ExceptionHandler

The @ExceptionHandler is an annotation used to **handle the specific exceptions and sending the custom responses to the client**. Define a class that extends the RuntimeException class. You can define the @ExceptionHandler method to handle the exceptions as shown.

**@RestController**

Spring RestController annotation is **used to create RESTful web services using Spring MVC**. Spring RestController takes care of mapping request data to the defined request handler method. Once response body is generated from the handler method, it converts it to JSON or XML response.

The RestController **allows to handle all REST APIs such as GET, POST, Delete, PUT requests**.

The @Controller is a common annotation which is used to mark a class as Spring MVC Controller while the @RestController is a special controller used in RESTFul web services and the equivalent of **@Controller + @ResponseBody**

**@GetMapping**

@**GetMapping** annotation maps HTTP GET requests onto specific handler methods. It is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod. GET)

From the naming convention we can see that each annotation is meant to handle respective incoming request method type, i.e. @**GetMapping** is **used** to handle GET type of request method.

RequestMapping can be used at class level: This annotation can be used both at the class and at the method level. while GetMapping only applies to method: Annotation for mapping HTTP GET requests onto specific handler methods.

**To set port number, user name and password:**

1. Server.port=8585

spring.security.user.name=scott

spring.security.user.password=tiger

# HTTP server port server.port=7070

# Context path of the application

server.servlet.context-path=/my-silly-app

# Default HTTP session time out

server.servlet.session.timeout=15m

Default port is 8080

spring.datasource.url=jdbc:mysql://localhost:3306/ecommerce

spring.datasource.username=scott

spring.datasource.password=tiger

2. custom properties

Coach.name=Micky Mouse

3. To get actuator/info

We need to add info.app.name=App

Info.app.version=1.0

management.endpoints.web.exposure.include=health,info

management.info.env.enabled=true

/health, /info, /beans, /auditevents, /mappings

@AutoWired

/\*\* Autowiring feature of spring framework enables you to inject the object

// dependency implicitly. It internally uses setter or constructor injection.

// Autowiring can't be used to inject primitive and string values. It works with

// reference only.\*/

The @Autowired annotation in spring **automatically injects the dependent beans into the associated references of a POJO class**. This annotation will inject the dependent beans by matching the data-type (i.e. Works internally as Autowiring byType).

Entity Manager

In JPA, the EntityManager interface is used to allow applications to manage and search for entities in the relational database. The EntityManager is **an API that manages the lifecycle of entity instances**. An EntityManager object manages a set of entities that are defined by a persistence unit.

Spring session

**Spring Session JDBC** provides SessionRepository implementation backed by a relational database and configuration support.

In hibernate we use, HQL (Hibernate Query language)

**Java Persistence Query Language**

@GeneratedValue

// The @GeneratedValue annotation is to configure the way of

// increment of the specified column(field). For example when

// using Mysql, you may specify auto\_increment in the definition

// of table to make it self-incremental

@Service

//In an application, the business logic resides within the service layer so we use the

//@Service Annotation to indicate that a class belongs to that layer.

@Transactional

@Transactional annotation is used **when you want the certain method/class(=all methods inside) to be executed in a transaction**.

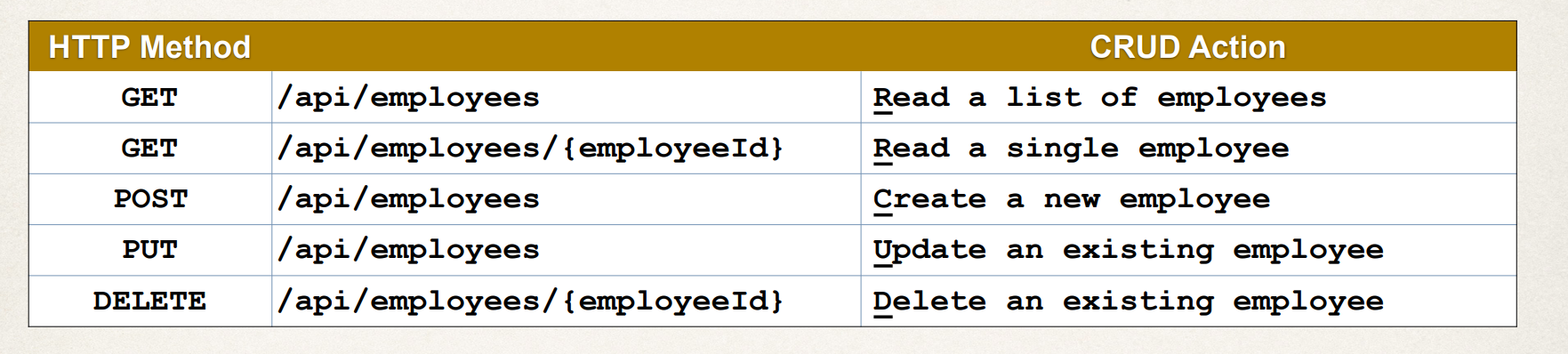
**@Transactional** is an annotation provided by Spring to manage database transactions. It ensures that a series of operations within a single method are executed in a transactional context, meaning they either all succeed or all fail (are rolled back). This is crucial for maintaining data integrity and consistency.

**Key Concepts**

1. **Transaction Management**: Ensures that a sequence of operations within a method is executed atomically.
2. **Atomicity**: All operations within a transaction either complete successfully or none of them do.
3. **Rollback**: If an exception occurs, all changes made during the transaction are rolled back to their previous state.

What is JPA?

• Java Persistence API (JPA) • Standard API for Object-to-Relational-Mapping (ORM)



Rest Controller

@GetMapping("/employees")

@GetMapping("/employees/{employeeId}")

@PostMapping("/employees")

@PutMapping("/employees")

@DeleteMapping("/employees/{employeeId}")

@Bean

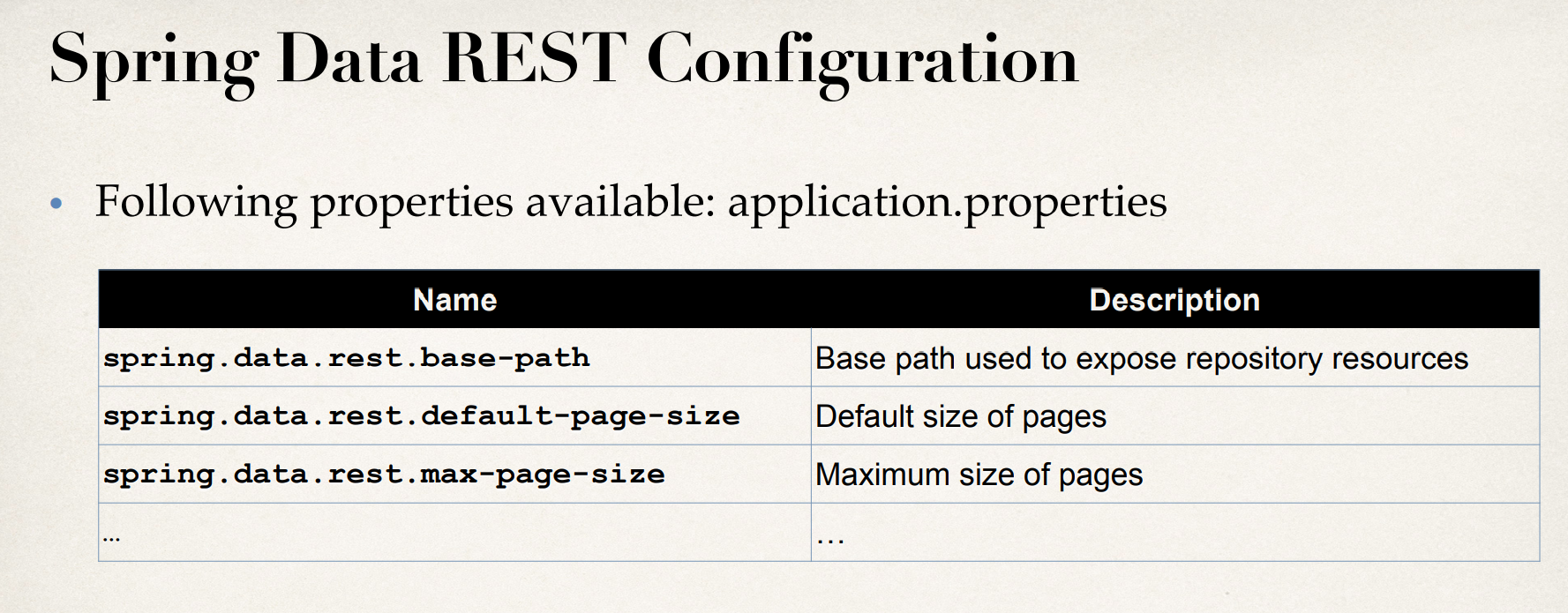
A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container. These beans are **created with the configuration metadata that you supply to the container**.

**By default scope of a bean is singleton**.

so you have to declare the scope of a been as prototype explicitly.

Pagination

• By default, Spring Data REST will return the first 20 elements • Page size = 20



http://localhost:8080/employees?page=0

<http://localhost:8080/employees?sort=lastName>

<http://localhost:8080/employees?sort=lastName,firstName,asc>

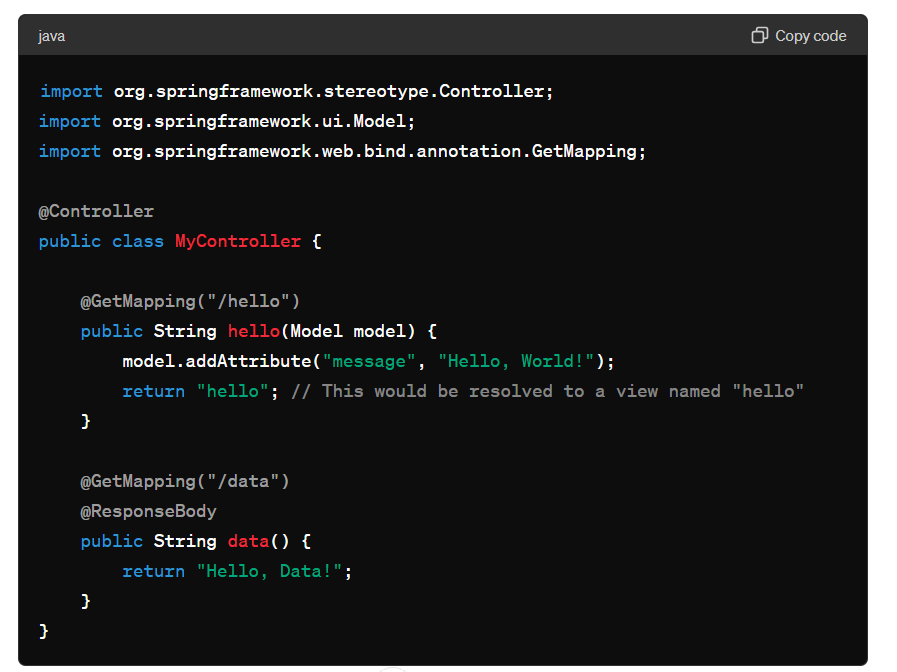
@RepositoryRestResource(path="members")

@Controller vs @RestController

In the context of a Spring MVC application, there are two primary types of controllers: **@Controller** and **@RestController**. Here’s a simple explanation of the difference between the two:

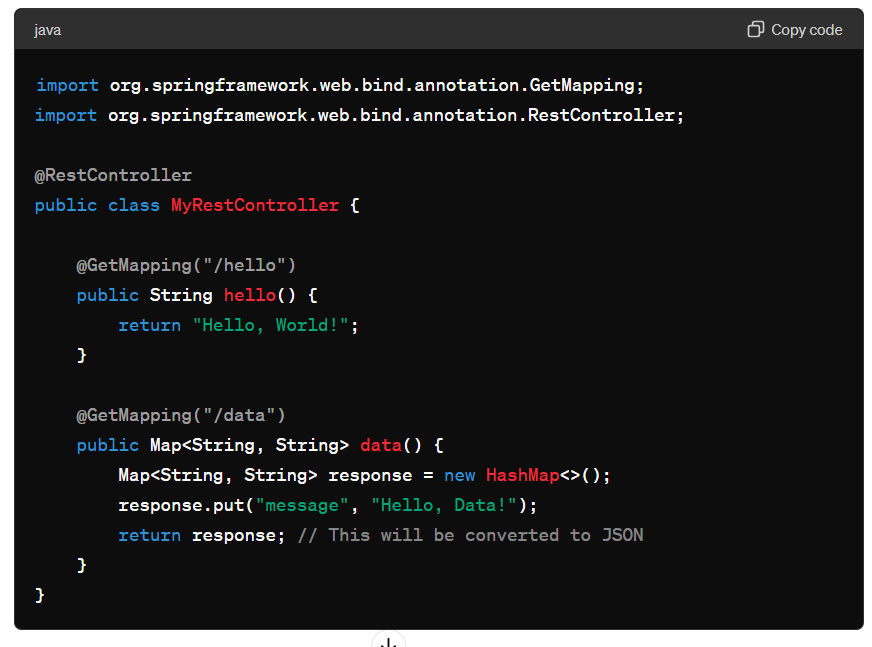
### @Controller

* **Purpose**: The **@Controller** annotation is used to define a controller in a Spring MVC application. It is typically used in web applications that serve HTML pages.
* **Return Type**: Methods in a **@Controller** usually return view names (like JSP, Thymeleaf, etc.) which are resolved by a view resolver to render HTML pages.
* **Data Handling**: To return data (like JSON or XML), you often need to use the **@ResponseBody** annotation on the method or on individual return values to indicate that the response should be written directly to the HTTP response body.
* **Example**:



### @RestController

* **Purpose**: The **@RestController** annotation is a specialized version of **@Controller** used to create RESTful web services. It is designed for APIs that primarily return data (like JSON or XML) rather than HTML views.
* **Return Type**: Methods in a **@RestController** return data directly (like JSON or XML) and the data is written directly to the HTTP response body. There is no need for the **@ResponseBody** annotation on each method because it is implied by the **@RestController** annotation.
* **Example**:

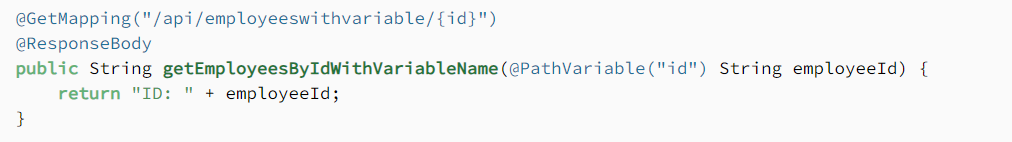


**Summary**

* **@Controller**:
  + Used for traditional web applications.
  + Typically returns views (HTML).
  + Requires **@ResponseBody** to return data directly.
* **@RestController**:
  + Used for RESTful web services.
  + Returns data directly (like JSON or XML).
  + Implicitly includes **@ResponseBody** on all methods.

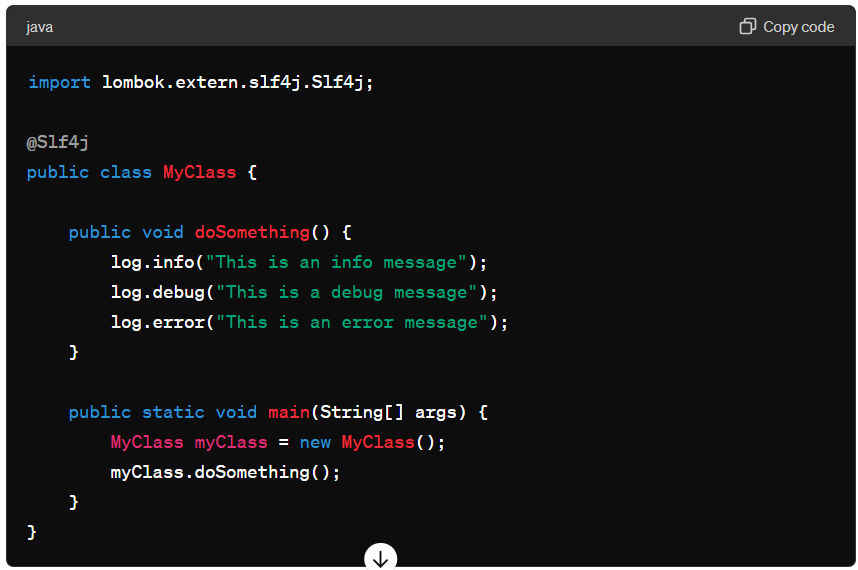
In simple terms, use **@Controller** when you are creating a web application that serves HTML pages, and use **@RestController** when you are creating a RESTful API that serves data in formats like JSON or XML.

@PathVariable :



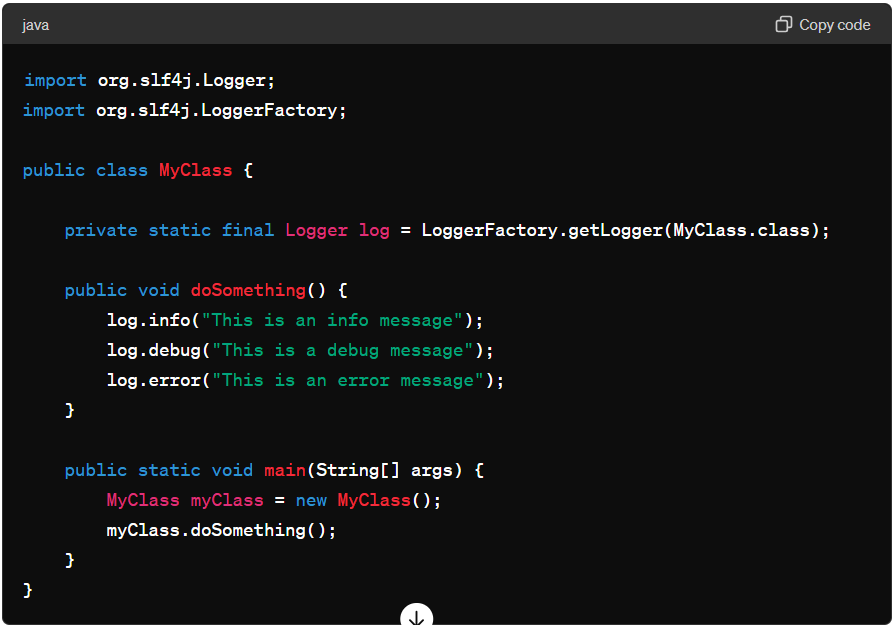
@Slf4j

**@Slf4j** is an annotation provided by the Lombok library in Java that simplifies the process of logging within your classes. When you use the **@Slf4j** annotation, Lombok automatically generates a **Logger** instance for your class, so you don't have to manually create or configure the logger.



* **Automatic Logger Generation**: When you annotate a class with **@Slf4j**, Lombok generates a **Logger** instance named **log** (or **LOG** in uppercase depending on your coding style preferences) using SLF4J (Simple Logging Facade for Java).
* **Logging Methods**: You can use various logging methods provided by the SLF4J **Logger** interface, such as **log.info()**, **log.debug()**, **log.warn()**, **log.error()**, etc.

Without Annotation :



### Conclusion

Using **@Slf4j** from Lombok simplifies the process of logging by automatically generating a **Logger** instance for your classes. It helps to reduce boilerplate code and ensures consistent logging practices across your application.