Transactional and Qualifier and Profiler  
===============================

@Transactional:

The @Transactional annotation ***in*** Spring is used to manage ***transaction*** boundaries in a declarative way. It ***ensures*** that a method is executed ***within*** a transaction context, and ***depending*** on the transaction attributes, it can roll back ***or*** commit the transaction based on the outcome. Here's how it works ***and*** its key features:

Key ***Features*** of @Transactional

Transaction Management:

When a method annotated with @Transactional is called, Spring creates a **new** transaction **if** there is none ***already*** running.

If ***there*** is an existing transaction, the method will join it.

Commit and Rollback:

By **default**, **if** the ***method*** completes successfully (without throwing a runtime exception), the transaction is committed.

If a runtime exception is thrown, the transaction is rolled back automatically.

Propagation:

@Transactional supports different propagation behaviors that specify how transactions relate to each other. Common propagation behaviors include:

REQUIRED (**default**): Joins the existing transaction or creates a **new** one **if** none exists.

REQUIRES\_NEW: Suspends the current transaction and creates a **new** one.

NESTED: Executes within a nested transaction **if** a transaction exists, or behaves like REQUIRED **if** there isn't one.

MANDATORY: Requires an existing transaction; **throws** an exception **if** there isn't one.

SUPPORTS: Participates in a transaction **if** one exists, but doesn't create a new one if none exists.

NOT\_SUPPORTED: Executes without a transaction, suspending any existing transaction.

NEVER: Executes without a transaction, throwing an exception **if** a transaction exists.

Isolation Levels:

Defines the level of isolation **for** transactions, which controls how the current transaction is isolated from other transactions. Common isolation levels include:

DEFAULT: Uses the **default** isolation level of the underlying datastore.

READ\_UNCOMMITTED: Allows dirty reads; one transaction can read changes made by another transaction before it's committed.

READ\_COMMITTED: Prevents dirty reads; a transaction cannot read changes made by another transaction before it's committed.

REPEATABLE\_READ: Prevents dirty reads and non-repeatable reads; a transaction reads the same data consistently.

SERIALIZABLE: The highest level of isolation; ensures full isolation from other transactions.

Read-Only:

The readOnly attribute can be set to **true** to optimize transaction performance **for** read-only operations, indicating that the transaction does not modify data.

Example Usage

Here's a simple example to demonstrate the use of @Transactional in a service class:

StudentService

java

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

**import** org.springframework.stereotype.Service;

**import** org.springframework.transaction.annotation.Transactional;

**import** java.util.List;

**import** java.util.Optional;

@Service

**public** **class** StudentService {

@Autowired

**private** StudentRepository studentRepository;

@Transactional // Transactional annotation applied

**public** Student saveStudent(Student student) {

**return** studentRepository.save(student);

}

@Transactional(readOnly = **true**) // Read-only transaction

**public** List<Student> getAllStudents() {

**return** studentRepository.findAll();

}

@Transactional(readOnly = **true**)

**public** Optional<Student> getStudentById(Long id) {

**return** studentRepository.findById(id);

}

@Transactional

**public** **void** deleteStudentById(Long id) {

studentRepository.deleteById(id);

}

@Transactional

**public** Student updateStudent(Long id, Student studentDetails) {

Optional<Student> studentOptional = studentRepository.findById(id);

**if** (studentOptional.isPresent()) {

Student student = studentOptional.get();

student.setName(studentDetails.getName());

student.setAddress(studentDetails.getAddress());

student.setProjects(studentDetails.getProjects());

student.setSchool(studentDetails.getSchool());

student.setCourses(studentDetails.getCourses());

**return** studentRepository.save(student);

} **else** {

**throw** **new** ResourceNotFoundException("Student not found with id " + id);

}

}

}

How It Works

Transaction Boundary: Each method annotated with @Transactional will have transaction management applied, ensuring consistent database operations.

Commit/Rollback: If any method annotated with @Transactional **throws** a runtime exception, the transaction will roll back, undoing any changes made.

Propagation: By **default**, the REQUIRED propagation behavior ensures that the methods join an existing transaction or create a **new** one **if** none exists.

Read-Only: Methods marked with readOnly = **true** optimize performance **for** read operations, indicating no changes to the database.

Using @Transactional helps ensure data integrity and consistency by automatically managing transaction boundaries, making your code more reliable and easier to maintain

Before Auto @Transactional feature we are doing manual work for these transaction management for any db operations like bellow

**import** java.sql.Connection;

**import** java.sql.DriverManager;

**import** java.sql.PreparedStatement;

**import** java.sql.SQLException;

**public** **class** StudentService {

privte statis final String ***DRIVER*** = “com.mysql.cj.jdbc.Driver”;

**private** **static** **final** String ***URL*** = "jdbc:mysql://localhost:3306/mydatabase";

**private** **static** **final** String ***USER*** = "myuser";

**private** **static** **final** String ***PASSWORD*** = "mypassword";

**public** **void** saveStudent(Student student) **throws** SQLException {

try{

Class.forName(DRIVER);

}catch(ClassNotFoundException e){ e.printStacktarce();

}

Connection connection = **null**;

**try** {

connection = DriverManager.*getConnection*(***URL***, ***USER***, ***PASSWORD***);

connection.setAutoCommit(**false**); // Disable auto-commit mode

// Save the student

String studentSql = "INSERT INTO student (name) VALUES (?)";

PreparedStatement studentStmt = connection.prepareStatement(studentSql);

studentStmt.setString(1, student.getName());

studentStmt.executeUpdate();

// Save the address

String addressSql = "INSERT INTO address (street, city, state, postal\_code) VALUES (?, ?, ?, ?)";

PreparedStatement addressStmt = connection.prepareStatement(addressSql);

Address address = student.getAddress();

addressStmt.setString(1, address.getStreet());

addressStmt.setString(2, address.getCity());

addressStmt.setString(3, address.getState());

addressStmt.setString(4, address.getPostalCode());

addressStmt.executeUpdate();

// Save the projects

String projectSql = "INSERT INTO project (title, description, student\_id) VALUES (?, ?, ?)";

PreparedStatement projectStmt = connection.prepareStatement(projectSql);

**for** (Project project : student.getProjects()) {

projectStmt.setString(1, project.getTitle());

projectStmt.setString(2, project.getDescription());

projectStmt.setLong(3, student.getId()); // Assuming student ID is generated and set

projectStmt.executeUpdate();

}

// Save the school

String schoolSql = "INSERT INTO school (name, address) VALUES (?, ?)";

PreparedStatement schoolStmt = connection.prepareStatement(schoolSql);

School school = student.getSchool();

schoolStmt.setString(1, school.getName());

schoolStmt.setString(2, school.getAddress());

schoolStmt.executeUpdate();

// Save the courses

String courseSql = "INSERT INTO course (title, description) VALUES (?, ?)";

PreparedStatement courseStmt = connection.prepareStatement(courseSql);

**for** (Course course : student.getCourses()) {

courseStmt.setString(1, course.getTitle());

courseStmt.setString(2, course.getDescription());

courseStmt.executeUpdate();

}

connection.commit(); // Commit the transaction

} **catch** (SQLException e) {

**if** (connection != **null**) {

connection.rollback(); // Rollback the transaction in case of an error

}

**throw** e;

} **finally** {

**if** (connection != **null**) {

connection.setAutoCommit(**true**); // Restore auto-commit mode

connection.close();

}

}

}

}

How to manage the Manual Tx in Hibernate:

how to manage transaction management using hibernate manually

Managing transactions manually in Hibernate involves using the Session and Transaction interfaces to control the transaction lifecycle. Here’s a step-by-step guide:

1. **Open a Session**: Create a session from the SessionFactory.
2. **Begin a Transaction**: Start a new transaction using the beginTransaction() method.
3. **Perform Database Operations**: Execute your database operations within the transaction.
4. **Commit the Transaction**: If all operations are successful, commit the transaction using the commit() method.
5. **Rollback the Transaction**: If an error occurs, rollback the transaction using the rollback() method to revert any changes.

In this example:

**import** org.hibernate.Session;

**import** org.hibernate.Transaction;

**import** org.hibernate.SessionFactory;

**import** org.hibernate.cfg.Configuration;

**public** **class** TransactionExample {

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

// Create SessionFactory

Configuration configuration = **new** Configuration().configure();

SessionFactory sessionFactory = configuration.buildSessionFactory();

// Open a Session

Session session = sessionFactory.openSession();

// Begin a Transaction

Transaction transaction = session.beginTransaction();

**try** {

// Perform Database Operations

// Example: Save a new entity

YourEntity entity = **new** YourEntity();

session.~~save~~(entity);

// Commit the Transaction

transaction.commit();

System.***out***.println("Transaction committed successfully.");

} **catch** (Exception e) {

// Rollback the Transaction if an error occurs

**if** (transaction != **null**) {

transaction.rollback();

System.***out***.println("Transaction rolled back due to an error.");

}

e.printStackTrace();

} **finally** {

// Close the Session

session.close();

}

}

}

* A session is opened from the SessionFactory.
* A transaction is started using beginTransaction().
* Database operations are performed (e.g., saving an entity).
* The transaction is committed if everything goes well.
* If an error occurs, the transaction is rolled back to undo any changes.
* The session is closed in the finally block to ensure resources are released.

**Key Differences**

1. **Declarative vs. Programmatic**:
   * **Declarative**: Using @Transactional, you simply annotate the methods that need transaction management, and Spring takes care of the rest.
   * **Programmatic**: Manually managing transactions requires explicit code to begin, commit, and rollback transactions.
2. **Error-Prone**:
   * **Declarative**: Less error-prone as the transaction management is handled by the framework.
   * **Programmatic**: More error-prone due to the need to manage transaction boundaries and error handling explicitly.
3. **Boilerplate Code**:
   * **Declarative**: Reduces boilerplate code, leading to cleaner and more maintainable code.
   * **Programmatic**: Involves more boilerplate code, making the codebase harder to maintain.
4. **Consistency**:
   * **Declarative**: Ensures consistent transaction management across the application.
   * **Programmatic**: Inconsistent management could lead to issues if not handled carefully in every method.

Using @Transactional provides a more modern, cleaner, and robust way to handle transactions in a Spring application. It abstracts away the complexities of transaction management, allowing developers to focus on the business logic

### @Qualifier

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

The @Qualifier annotation in Spring is used to resolve ambiguity when multiple beans of the same type are available in the Spring Application Context. It helps specify which bean should be injected when there are multiple candidates. This is particularly useful in scenarios where you have different implementations of an interface or multiple beans of the same type.

### How @Qualifier Works

When you have multiple beans of the same type and you want to inject a specific one, you can use the @Qualifier annotation to indicate which bean should be used. You can use @Qualifier with field injection, constructor injection, or setter injection.

### Scenario: Notification Services

Imagine we have a system with multiple notification services, such as EmailNotificationService, SMSNotificationService, and PushNotificationService. We need to dynamically select the appropriate notification service based on some criteria.

### Step 1: Define the Service Interface and Implementations without @Qualifier or @Primary

#### NotificationService Interface

public interface NotificationService {

void sendNotification(String message);

}

@Service

**public** **class** EmailNotificationService **implements** NotificationService {

@Override

**public** **void** sendNotification(String message) {

System.***out***.println("Sending email notification:"+message);

}

}

@Service

**public** **class** SMSNotificationService **implements** NotificationService {

@Override

**public** **void** sendNotification(String message) {

System.***out***.println("Sending sms notification:"+message);

}

}

@Service

**public** **class** PushNotificationService **implements** NotificationService {

@Override

**public** **void** sendNotification(String message) {

System.***out***.println("Sending push notification:"+message);

}

}

@RestController

**public** **class** NotificationController {

@Autowired

**private** NotificationService notificationService;

}

Then try to start the SpringBoot application by Running

NotificationsApplication.java

We will bellow error : why bzc there are 3 sub class for one super class ref variable so which sub class object we need to create and assign to sup class ref var is a ambiguity to spring container. If we want to avoid these ambiguity error we should use @Qualifier or @Primary

Here

@Primary : is act like a default value, this we can use it for only one class.

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

APPLICATION FAILED TO START

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

Description:

Field notificationService in com.k7it.controller.NotificationController required a single bean, but 3 were found:

- emailNotificationService: defined in file [D:\lab\202402\_JavaBatch\eclipse\_workspace\JPA\_Mappings\Notifications\target\classes\com\k7it\service\EmailNotificationService.class]

- pushNotificationService: defined in file [D:\lab\202402\_JavaBatch\eclipse\_workspace\JPA\_Mappings\Notifications\target\classes\com\k7it\service\PushNotificationService.class]

- SMSNotificationService: defined in file [D:\lab\202402\_JavaBatch\eclipse\_workspace\JPA\_Mappings\Notifications\target\classes\com\k7it\service\SMSNotificationService.class]

This may be due to missing parameter name information

Action:

Consider marking one of the beans as @Primary, updating the consumer to accept multiple beans, or using @Qualifier to identify the bean that should be consumed

@Service

@Primary

**public** **class** EmailNotificationService **implements** NotificationService

our application will start succefully and

@Autowired

**private** NotificationService notificationService;

here notificationService = new EmailNotificationService();

But@Primary can allow only one Bean. If we use @Primary for multiple sub classes what will happened?

Like

@Service

@Primary

**public** **class** SMSNotificationService **implements** NotificationService

@Service

@Primary

**public** **class** PushNotificationService **implements** NotificationService

then again restart the server : server will not start we will bellow error :

Application run failed

org.springframework.beans.factory.UnsatisfiedDependencyException: Error creating bean with name 'notificationController': Unsatisfied dependency expressed through field 'notificationService': No qualifying bean of type 'com.k7it.service.NotificationService' available: more than one 'primary' bean found among candidates: [emailNotificationService, pushNotificationService, SMSNotificationService]

if we want avoid this kind of ambiguity errors its better way to go with @Qualifier annotations .

### Scenario: Notification Services

Imagine we have a system with multiple notification services, such as EmailNotificationService, SMSNotificationService, and PushNotificationService. We need to dynamically select the appropriate notification service based on some criteria.

### Step 1: Define the Service Interface and Implementations

#### NotificationService Interface

java

public interface NotificationService {

void sendNotification(String message);

}

#### EmailNotificationService Implementation

java

@Service

@Qualifier("emailNotificationService")

**public** **class** EmailNotificationService **implements** NotificationService {

@Override

**public** **void** sendNotification(String message) {

System.***out***.println("Sending email notification:"+message);

}

}

#### SMSNotificationService Implementation

java

import org.springframework.stereotype.Service;

@Service

@Qualifier("smsNotificationService")

**public** **class** SMSNotificationService **implements** NotificationService {

@Override

**public** **void** sendNotification(String message) {

System.***out***.println("Sending sms notification:"+message);

}

}

#### PushNotificationService Implementation

java

import org.springframework.stereotype.Service;

@Service

@Qualifier("pushNotificationService")

**public** **class** PushNotificationService **implements** NotificationService {

@Override

**public** **void** sendNotification(String message) {

System.***out***.println("Sending push notification:"+message);

}

}

If we write the Controller class like bellow

@RestController

**public** **class** NotificationController {

@Autowired

@Qualifier("emailNotificationService")

**private** NotificationService notificationService;

}

Then if we start the server our Controller class we will get the object for EmailNotificationService class object into notificationService ref variable

But here one limitation is there in my NotificationController class always it will send notifications through Emial type only, if I want to send sms it not possible. Since we fixed notificationtService type as Email.

If I want to decide my notification type Dynamically my self eiterh Email or sms or push, then we need to take help of Factory design patter and we need to send one input param to specify which notification you want like “email” or “sms” or “push” . based on my input string my factory class should return corresponding service class object to my controller class.

### Step 2: Create a Factory Class

The factory class will use @Qualifier to inject all implementations and select the appropriate one dynamically.

#### NotificationServiceFactory

**package** com.k7it.factory;

**import** java.util.Map;

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

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

**import** org.springframework.stereotype.Component;

**import** com.k7it.service.NotificationService;

@Component

**public** **class** NotificationFactory {

// 1. individual Autowired into factory class

// @Autowired

// @Qualifier("emailNotificationService")

// NotificationService emailNotificationService;

//

// @Autowired

// @Qualifier("smsNotificationService")

// NotificationService smsNotificationService;

//

// @Autowired

// @Qualifier("pushNotificationService")

// NotificationService pushNotificationService;

//

//

// private Map<String, NotificationService> notificationServices = Map.of(

// "email", emailNotificationService,

// "sms", smsNotificationService,

// "push", pushNotificationService

// );

// 2. Autowired through constructor and pass which are the fields you autowired in this class while creating object for

// class from other class.

**private** Map<String, NotificationService> notificationServiceMap;

@Autowired

**public** NotificationFactory(

@Qualifier("emailNotificationService") NotificationService emailNotificationService,

@Qualifier("smsNotificationService") NotificationService smsNotificationService,

@Qualifier("pushNotificationService") NotificationService pushNotificationService){

notificationServiceMap = Map.*of*(

"email", emailNotificationService,

"sms", smsNotificationService,

"push", pushNotificationService

);

}

**public** NotificationService getNotificationService(String type) {

// 1.normal core java way of creating objects in factory pattern

// if("sms".equals(type)) {

// return new SMSNotificationService();

// }else if("email".equals(type)) {

// return new EmailNotificationService();

// }else if("push".equals(type)) {

// return new PushNotificationService();

// }

// 2. through spring based

//

// if ("email".equals(type)) {

// return emailNotificationService;

// } else if ("sms".equals(type)) {

// return smsNotificationService;

// } else if ("push".equals(type)) {

// return pushNotificationService;

// }

// 3.using map

**return** notificationServiceMap.get(type);

}

}

### Step 3: Use the Factory in the Controller

#### NotificationController

java

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

**import** org.springframework.web.bind.annotation.GetMapping;

**import** org.springframework.web.bind.annotation.RequestParam;

**import** org.springframework.web.bind.annotation.RestController;

@RestController

**public** **class** NotificationController {

**private** **final** NotificationServiceFactory notificationServiceFactory;

@Autowired

**public** NotificationController(NotificationServiceFactory notificationServiceFactory) {

**this**.notificationServiceFactory = notificationServiceFactory;

}

@GetMapping("/notify")

**public** String sendNotification(@RequestParam String message, @RequestParam String notificationType) {

NotificationService notificationService = notificationServiceFactory.getNotificationService(notificationType);

**if** (notificationService != **null**) {

notificationService.sendNotification(message);

**return** "Notification sent successfully.";

} **else** {

**return** "Invalid notification type.";

}

}

}

### Example Usage

To send an email notification:

http

GET /notify?message=Hello&notificationType=email

To send an SMS notification:

http

GET /notify?message=Hello&notificationType=sms

To send a push notification:

http

GET /notify?message=Hello&notificationType=push

### Summary

* **Define the Service Interface and Implementations**: Create multiple implementations of the NotificationService interface and annotate them with @Qualifier.
* **Create a Factory Class**: Use @Qualifier to inject all implementations into a factory class and select the appropriate one dynamically.
* **Use the Factory in the Controller**: Inject the factory into your controller and use it to select the appropriate service based on a parameter

Factory design pattern :  
================  
The Factory Design Pattern is one of the most frequently used design patterns in object-oriented programming. It falls under the category of Creational Design Patterns and is used to create objects without specifying the exact **class** of object that will be created. Here's a detailed explanation:

What is the Factory Design Pattern?

The Factory Design Pattern provides an **interface** **for** creating objects in a superclass but allows subclasses to alter the type of objects that will be created. This pattern is used when we have a **super** **class** with multiple sub-classes and based on input, we need to **return** one of the sub-classes.

Key Concepts

Factory Method: Defines an **interface** **for** creating an object, but lets subclasses alter the type of objects that will be created.

Product: The **interface** of objects that the factory method creates.

Concrete Product: The implementation of the Product **interface**.

Creator: Declares the factory method, which returns an object of type Product.

Concrete Creator: Implements the factory method to produce a concrete product.

Benefits

Encapsulation: The Factory pattern promotes encapsulation by hiding the object creation process.

Loosely Coupled Code: The client code can work with any concrete implementation of the product by referring to the **abstract** product type.

Scalability: New types of products can be added without changing existing code.

Example

Here’s a simple example to illustrate the Factory Design Pattern:

java

// Product Interface

**interface** Shape {

**void** draw();

}

// Concrete Product: Circle

**class** Circle **implements** Shape {

@Override

**public** **void** draw() {

System.out.println("Drawing a Circle");

}

}

// Concrete Product: Rectangle

**class** Rectangle **implements** Shape {

@Override

**public** **void** draw() {

System.out.println("Drawing a Rectangle");

}

}

// Factory

**class** ShapeFactory {

// Factory method to create objects

**public** Shape getShape(String shapeType) {

**if** (shapeType == **null**) {

**return** **null**;

}

**if** (shapeType.equalsIgnoreCase("CIRCLE")) {

**return** **new** Circle();

} **else** **if** (shapeType.equalsIgnoreCase("RECTANGLE")) {

**return** **new** Rectangle();

}

**return** **null**;

}

}

// Client

**public** **class** FactoryPatternDemo {

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

ShapeFactory shapeFactory = **new** ShapeFactory();

// Get an object of Circle and call its draw method

Shape shape1 = shapeFactory.getShape("CIRCLE");

shape1.draw(); // Output: Drawing a Circle

// Get an object of Rectangle and call its draw method

Shape shape2 = shapeFactory.getShape("RECTANGLE");

shape2.draw(); // Output: Drawing a Rectangle

}

}

In **this** example:

The Shape **interface** defines a common method draw() **for** all shapes.

Circle and Rectangle are concrete implementations of the Shape **interface**.

ShapeFactory is the factory **class** that creates objects of Circle and Rectangle based on the provided input.

In the client code (FactoryPatternDemo), the ShapeFactory is used to get an instance of Circle or Rectangle without exposing the instantiation logic to the client.

This approach allows you to add **new** shapes easily without modifying the factory **class**, promoting scalability and maintainability.

@Profiler  
=======

You can use @Profile in combination with different property files for different environments (profiles) like development, testing, and production. Spring Boot allows you to configure different settings for each profile by using different application.properties or application.yml files.

### How to Achieve Environment-Specific Configurations

#### Step 1: Create Environment-Specific Property Files

1. **Default** application.properties:
   * This file is used when no specific profile is active.
2. **Profile-Specific Property Files**:
   * Create application-dev.properties for development.
   * Create application-prod.properties for production.
   * Similarly, create application-test.properties for testing.

#### Example Property Files

1. application.properties (Default configuration)

properties

spring.profiles.active=dev

1. application-dev.properties (Development configuration)

properties

# Development-specific properties

spring.datasource.url=jdbc:h2:mem:devdb

spring.datasource.username=devuser

spring.datasource.password=devpass

spring.jpa.hibernate.ddl-auto=update

1. application-prod.properties (Production configuration)

properties

# Production-specific properties

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

spring.datasource.username=produser

spring.datasource.password=prodpass

spring.jpa.hibernate.ddl-auto=update

1. application-test.properties (Testing configuration)

properties

# Testing-specific properties

spring.datasource.url=jdbc:h2:mem:testdb

spring.datasource.username=testuser

spring.datasource.password=testpass

spring.jpa.hibernate.ddl-auto=update

### Step 2: Use @Value to Inject Properties in Configuration Classes

#### Development Configuration

java

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

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.Profile;

import javax.sql.DataSource;

import com.zaxxer.hikari.HikariDataSource;

@Configuration

@Profile("dev")

public class DevConfig {

@Value("${spring.datasource.url}")

private String url; // jdbc:h2:mem:devdb

@Value("${spring.datasource.username}")

private String username;//devUser

@Value("${spring.datasource.password}")

private String password;// devpass

@Bean

public DataSource dataSource() {

HikariDataSource dataSource = new HikariDataSource();

dataSource.setJdbcUrl(url);

dataSource.setUsername(username);

dataSource.setPassword(password);

return dataSource;

}

}

#### Production Configuration

java

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

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.Profile;

import javax.sql.DataSource;

import com.zaxxer.hikari.HikariDataSource;

@Configuration

@Profile("prod")

public class ProdConfig {

@Value("${spring.datasource.url}")

private String url;

@Value("${spring.datasource.username}")

private String username;

@Value("${spring.datasource.password}")

private String password;

@Bean

public DataSource dataSource() {

HikariDataSource dataSource = new HikariDataSource();

dataSource.setJdbcUrl(url);

dataSource.setUsername(username);

dataSource.setPassword(password);

return dataSource;

}

}

#### Step3: Activating Profiles

You can activate profiles in the application.properties file or set them as JVM arguments.

1. **Activating Profile in** application.properties:

properties

spring.profiles.active=dev

1. **Activating Profile as a JVM Argument**:

sh

-Dspring.profiles.active=prod

### How it Works

When you activate a specific profile (e.g., dev, prod, or test), Spring Boot automatically loads the corresponding profile-specific property file and uses the beans defined in the configuration classes annotated with @Profile.

* **Development Environment**: With spring.profiles.active=dev, Spring Boot will load application-dev.properties and use the beans defined in DevConfig.
* **Production Environment**: With spring.profiles.active=prod, Spring Boot will load application-prod.properties and use the beans defined in ProdConfig.
* **Testing Environment**: With spring.profiles.active=test, Spring Boot will load application-test.properties and use the beans defined in TestConfig.

This approach provides a clean and organized way to manage different configurations for different environments, ensuring that your application behaves correctly in each environment.

@Value Annotations:  
===================

about @Value annotaion

The @Value annotation in Spring is used to inject values into fields, method parameters, or constructor arguments. It allows you to read property values from property files, environment variables, or even system properties and inject them directly into your Spring-managed beans. This is especially useful for externalizing configuration.

**Key Features of @Value**

1. **Injecting Property Values**: You can inject values from application.properties or application.yml files.
2. **Default Values**: You can provide default values if the specified property is not found.
3. **SpEL (Spring Expression Language)**: You can use SpEL expressions to perform operations or call methods to set the value.

### Example Usage of @Value

Let's go through a few examples to understand how @Value works.

#### Example 1: Injecting Values from application.properties

Create an application.properties file with some properties:

properties

app.name=MySpringApp

app.version=1.0.0

Now, inject these values into your Spring bean:

java

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

import org.springframework.stereotype.Component;

@Component

public class AppConfig {

@Value("${app.name}")

private String appName;

@Value("${app.version}")

private String appVersion;

public void printAppDetails() {

System.out.println("App Name: " + appName);

System.out.println("App Version: " + appVersion);

}

}

#### Example 2: Using Default Values

You can provide default values if the property is not found:

java

@Value("${app.description:Default testdescription}")

private String appDescription;

In this example, if app.description is not defined in the properties file, appDescription will be set to "Default description".

#### Example 3: Using SpEL Expressions

You can use SpEL expressions to perform operations or call methods:

java

@Value("#{T(java.lang.Math).random() \* 100}")

private double randomValue;

This example injects a random value between 0 and 100.

### Full Example

Let's combine everything in a complete example:

1. application.properties:

properties

app.name=MySpringApp

app.version=1.0.0

app.description=Spring Application

1. **AppConfig Class**:

java

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

import org.springframework.stereotype.Component;

@Component

public class AppConfig {

@Value("${app.name}")

private String appName;

@Value("${app.version}")

private String appVersion;

@Value("${app.description:Default description}")

private String appDescription;

@Value("#{T(java.lang.Math).random() \* 100}")

private double randomValue;

public void printAppDetails() {

System.out.println("App Name: " + appName);

System.out.println("App Version: " + appVersion);

System.out.println("App Description: " + appDescription);

System.out.println("Random Value: " + randomValue);

}

}

1. **Main Application Class**:

java

import org.springframework.boot.CommandLineRunner;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

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

@SpringBootApplication

public class MyApp implements CommandLineRunner {

@Autowired

private AppConfig appConfig;

public static void main(String[] args) {

SpringApplication.run(MyApp.class, args);

}

@Override

public void run(String... args) throws Exception {

appConfig.printAppDetails();

}

}

When you run this application, it will read the values from application.properties and inject them into AppConfig, which then prints the app details.

### Summary

* @Value: Used to inject property values into Spring beans.
* **Sources**: Reads from application.properties, application.yml, environment variables, and system properties.
* **Default Values**: Provides default values if properties are not found.
* **SpEL**: Supports Spring Expression Language for more advanced operations.

This annotation is highly useful for externalizing configuration and making your application more flexible and easier to manage.

How to use the @Transactional annotation for MongoDB:  
==========================================

The @Transactional annotation is typically associated with relational databases and is managed by Spring's transaction management for JPA or JDBC. However, when working with MongoDB, the transaction management is a bit different due to the nature of NoSQL databases.

### Transaction Management in MongoDB

Starting with MongoDB 4.0, multi-document ACID transactions are supported, but they require special handling. Spring Data MongoDB provides support for MongoDB transactions using the @Transactional annotation. However, the setup and behavior differ slightly compared to relational databases.

### Configuring Transactions in MongoDB with Spring Data

To use transactions with MongoDB in Spring Data, you need to ensure the following:

1. **Enable Transactions**: Ensure that MongoDB is configured to support transactions. Transactions are supported on replica sets and sharded clusters.
2. **Use** @Transactional: Apply the @Transactional annotation in your service layer.

### Example Configuration and Usage

#### MongoDB Configuration

Ensure your MongoDB is running as a replica set or sharded cluster to support transactions.

#### Service Layer with @Transactional

Here's how you can use @Transactional with MongoDB in your service layer:

java

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

import org.springframework.data.mongodb.core.MongoTemplate;

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

@Service

public class StudentService {

@Autowired

private StudentMongoRepository studentMongoRepository;

@Autowired

private MongoTemplate mongoTemplate;

@Transactional("mongoTransactionManager")

public Student saveStudentToMongo(Student student) {

return studentMongoRepository.save(student);

}

@Transactional("mongoTransactionManager")

public Student updateStudentInMongo(String id, Student studentDetails) {

Student student = studentMongoRepository.findById(id).orElseThrow(() -> new ResourceNotFoundException("Student not found with id " + id));

student.setName(studentDetails.getName());

student.setAddress(studentDetails.getAddress());

student.setProjects(studentDetails.getProjects());

student.setSchool(studentDetails.getSchool());

student.setCourses(studentDetails.getCourses());

return studentMongoRepository.save(student);

}

// Other methods...

}

#### MongoDB Transaction Manager Configuration

You need to configure a transaction manager for MongoDB:

java

import com.mongodb.client.MongoClient;

import com.mongodb.client.MongoClients;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.data.mongodb.MongoDatabaseFactory;

import org.springframework.data.mongodb.MongoTransactionManager;

import org.springframework.data.mongodb.core.SimpleMongoClientDatabaseFactory;

@Configuration

public class MongoConfig {

@Bean

public MongoClient mongoClient() {

return MongoClients.create("mongodb://localhost:27017");

}

@Bean

public MongoDatabaseFactory mongoDatabaseFactory(MongoClient mongoClient) {

return new SimpleMongoClientDatabaseFactory(mongoClient, "your\_mongodb\_database");

}

@Bean

public MongoTransactionManager mongoTransactionManager(MongoDatabaseFactory dbFactory) {

return new MongoTransactionManager(dbFactory);

}

}

### Summary

* **Relational Databases**: Use @Transactional directly for transaction management.
* **MongoDB**: Transactions are supported starting with MongoDB 4.0. You need to configure MongoDB as a replica set or sharded cluster and configure a MongoTransactionManager in Spring.
* **Service Layer**: Apply @Transactional("mongoTransactionManager") to your service methods to manage transactions in MongoDB.

This setup ensures that your MongoDB operations are managed within transactions, similar to how you manage transactions in relational databases