Spring Data

Spring Data is a part of the larger Spring Framework designed to simplify data access for Java applications. With features for creating repository implementations automatically, it significantly reduces the amount of boilerplate code developers have to write.

Spring Data supports a range of data store technologies, both relational and NoSQL, including but not limited to:

* JPA (Java Persistence API)
* MongoDB
* Apache Cassandra
* Redis
* Elasticsearch
* Apache Solr
* Neo4j

For each of these data store technologies, Spring Data provides a unified programming model. It provides a common set of interfaces, namely Repository interfaces, which provide methods for CRUD (Create, Read, Update, Delete) operations, sorting, and pagination out of the box.

A unique feature of Spring Data is its ability to generate queries directly from method names in the Repository interfaces. For example, a method named **findByUsername(String username)** in a Repository would automatically be implemented as a query that searches for an entity with a specific username.

Spring Boot Data JPA is a specific project of Spring Data that integrates with JPA (Java Persistence API) for object-relational mapping with a relational database. It further simplifies the use of data access technologies, reducing the amount of boilerplate code and aiding in implementing a layer of persistence in your application.

Spring Boot builds on top of Spring Data and provides additional auto-configuration capabilities, so developers can focus more on business logic rather than setup and configuration of the persistence layer. Spring Boot provides a standalone, production-grade Spring-based application environment, simplifying the setup and execution of projects and services.

In summary, Spring Data is a powerful umbrella project in the Spring ecosystem that makes it easier and more efficient for developers to work with data stores by providing a consistent programming model across different types of databases, and Spring Boot enhances this by providing auto-configuration and runtime environment features.

Top of Form

**Advantages**

Spring Boot Data JPA has several advantages that make it an attractive choice for applications that require data persistence:

1. **Reduced Boilerplate Code:** By providing a simple interface for CRUD operations, Spring Boot Data JPA eliminates the need for boilerplate code. This reduces the amount of manual coding and makes the codebase easier to maintain.
2. **Repository Generation:** Spring Boot Data JPA can automatically create repository implementations at runtime based on interface definitions. This saves a significant amount of implementation effort.
3. **Query Methods:** You can define queries simply by declaring method signatures in the repository interface. Spring Boot Data JPA will automatically generate the implementation for these queries.
4. **Consistent Programming Model:** Whether you're working with SQL databases via JPA or with NoSQL databases, Spring Data offers a consistent programming model that abstracts the specifics of the underlying data source.
5. **Integration with Spring Ecosystem:** Being part of the Spring ecosystem, Spring Boot Data JPA integrates seamlessly with other Spring projects. This includes the transaction management capabilities of Spring, which can be applied uniformly across all data access that you perform.
6. **Flexible Mapping:** With the help of JPA, Spring Boot Data JPA supports a wide range of mappings between object-oriented domain models and relational database tables.
7. **Advanced Features:** It supports advanced features like caching, optimistic locking, and auditing out of the box.
8. **Pagination and Sorting:** The framework provides built-in support for pagination and sorting, which are essential for performance in applications with large amounts of data.
9. **Exception Translation:** It provides a persistence-technology-agnostic exception hierarchy with exceptions being translated into Spring's portable Data Access Exception hierarchy.

All these features combined make Spring Boot Data JPA an excellent choice for managing persistence in Java applications in a clean and maintainable way. It allows developers to focus more on the business logic and less on the plumbing related to data access.

Bottom of Form

Spring Boot is a module of the Spring Framework that offers Rapid Application Development, and it simplifies the bootstrapping and development of a new Spring application.

When we talk about the ORM (Object-Relational Mapping) in Spring Boot, we're typically referring to the use of Spring Data JPA, which is a part of the larger Spring Data family.

1. **Spring Data JPA:** This is an abstraction over Java Persistence API (JPA), which itself is an abstraction over JDBC. By using Spring Data JPA, developers can access and persist data between Java objects/classes and relational databases. It provides a way to reduce the amount of boilerplate code required by JPA through its repositories.
2. **Repositories:** These are interfaces provided by Spring Data JPA that we can extend to create our own interfaces. For example, by extending **CrudRepository** or **JpaRepository**, we can get basic CRUD operations (Create, Read, Update, Delete) for our entity without needing to implement these methods. We can also define our custom queries using method signatures in the repository interfaces.
3. **Entities:** These are regular POJOs (Plain Old Java Objects) that are mapped to database tables by using JPA annotations. For example, the **@Entity** annotation is used to mark a POJO as a JPA entity class, **@Id** is used to denote the primary key field, **@Column** is used for the fields that will map to the database table columns, and so on.
4. **EntityManager:** This is an interface in JPA that manages a set of entities that are defined in an application. It provides operations such as persisting, merging, removing, and finding entities. In Spring Data JPA, you don't often interact with the **EntityManager** directly as most of these operations are available through the repositories.
5. **Transactions:** Spring provides a **@Transactional** annotation to define the scope of a single database transaction. This annotation can be used at both the class level and the method level. It provides options to configure propagation, isolation, timeout, read-only, and rollback conditions.

Spring Boot also integrates well with other ORMs such as MyBatis and JOOQ. These provide different patterns for interacting with a database, but they aren't as commonly used with Spring Boot as JPA.

In summary, the combination of Spring Boot with an ORM like Spring Data JPA provides a powerful toolkit for rapidly developing database-driven applications with less boilerplate code and good transaction management support.

**Disadvantages**

While Spring Boot Data JPA offers numerous advantages, it also has some potential disadvantages or challenges, including:

1. **Learning Curve:** Spring Boot and Spring Data JPA come with a certain learning curve, particularly for developers who are new to the Spring ecosystem or those who aren't familiar with concepts like dependency injection, aspect-oriented programming, or ORM.
2. **Performance:** While JPA makes it easier to work with databases, it may not always offer the best performance. For example, it can sometimes generate inefficient SQL queries, and its use of objects for everything can increase memory consumption. Developers often have to tune the JPA provider (like Hibernate) to ensure optimal performance.
3. **Complexity:** Spring Data JPA can add a level of complexity and obscure what's happening under the hood, making debugging and performance tuning more challenging. Sometimes, it might be quicker and easier to write native SQL than to try and understand how to make Spring Data JPA do what you want, especially for complex queries.
4. **Limited Query Capabilities:** While the query generation from method names is convenient for simple cases, it's not as flexible as writing SQL queries directly. For complex queries, you'll likely have to use the **@Query** annotation and write the query using JPQL or native SQL.
5. **Not Suitable for All Applications:** If your application requires complex queries, batch updates, stored procedures, or other advanced database features, JPA and Spring Data JPA may not be the best fit. They are designed for simplicity and general use cases, so they don't always support more complex scenarios as well as some other tools.
6. **Database Vendor Lock-in:** Although JPA is designed to be independent of the specific database being used, in practice, moving from one database to another can still require significant effort. This is because different databases have different SQL dialects and feature sets, so an application using JPA may still end up using vendor-specific features, leading to potential lock-in.

In conclusion, while Spring Boot Data JPA has many advantages and can significantly increase productivity for many applications, it's not always the best tool for every situation. It's important to evaluate the specific needs and constraints of your project before deciding on the technology to use.

Setting up spring boot data jpa application

***Repository generation***

1. Repository interface (marker interface)
   1. CrudRepository(12) extends Repository
   2. PagingAndSortingRepository(2) extends Repository
   3. ListPagingAndSortingRepository(1) extends PagingAndSortingRepository
   4. ListCrudRepository(3) extends CrudRepository
   5. JpaRepository(12) extends ListCrudRepository, QueryByExampleExecutor(7),ListPagingAndSortingRepository(1)

Total 35 methods

Spring Boot configures ORM-related functionalities and deals with data sources:

**Auto-Configuration in Spring Boot**

Spring Boot's primary purpose is to simplify the configuration and initialization process for Spring applications. It does so through a process called auto-configuration. Spring Boot automatically configures your application based on the dependencies you have added to the project.

For instance, if you include the **spring-boot-starter-data-jpa** dependency in your project, Spring Boot will automatically set up the necessary configurations to integrate Spring Data JPA into your application. This includes the creation of an **EntityManagerFactory** and a default **DataSource**, among other things.

Spring Boot relies on sensible defaults based on conventions. However, if you want to override these default configurations, you can do so through the **application.properties** or **application.yml** file.

**DataSource Configuration in Spring Boot**

The **DataSource** is a fundamental concept in Spring Boot for database connectivity. It represents a database connection or a connection pool. In a Spring Boot application, you can configure multiple data sources, if needed.

Spring Boot auto-configuration attempts to automatically configure your data source. If **HikariCP**, **Tomcat pooling**, or **Commons DBCP** are on the classpath, one of them will be chosen for the **DataSource** implementation. As of Spring Boot 2.0, **HikariCP** is the default **DataSource** implementation.

Here is a basic **DataSource** configuration example in an **application.properties** file:

spring.datasource.url=jdbc:mysql://localhost/test

spring.datasource.username=dbuser

spring.datasource.password=dbpass

spring.datasource.driver-class-name=com.mysql.jdbc.Driver

You can change the default settings by specifying your own configurations. For example, to change the connection pool's maximum size when using HikariCP, you can add the following line:

spring.datasource.hikari.maximum-pool-size=5

**Integration with JPA/Hibernate**

When the **spring-boot-starter-data-jpa** starter is added to a project, Spring Boot will automatically set up Hibernate as the JPA provider and will enable the repository support.

The auto-configuration will also take care of the **DataSource** setup (as previously explained), **EntityManagerFactory** creation, **PlatformTransactionManager** setup, and Spring Data repository configuration.

Here's a basic example of how you would configure JPA/Hibernate via **application.properties**:

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect

Remember, the **spring.jpa.hibernate.ddl-auto** property should be used for development purposes only. For production, it's recommended to use database migration tools like Flyway or Liquibase.

With all these settings, Spring Boot automatically manages the complex initialization processes that were required in traditional Spring applications. You simply focus on developing the application, and Spring Boot takes care of the rest.

**Spring Data connection pool**

In the context of Spring Boot, connection pooling is a technique used to manage the database connections. A connection pool creates and maintains a pool of connections to the database. This helps to boost the performance of the applications, as it removes the overhead of initializing a new connection every time a request is made to the database.

Spring Boot supports several connection pooling solutions:

Thanks for the update. Here are the connection pool solutions supported by Spring Boot with the latest additions:

1. **HikariCP**: A lightweight, reliable, and extremely performant JDBC connection pool. As of Spring Boot 3.x, HikariCP is the default connection pool.
2. **Tomcat Connection Pool**: Offers a range of advanced features, including a connection validation and SQL statement logging and thread-local connections.
3. **Oracle UCP**: Oracle Universal Connection Pool (UCP) is a Java connection pool from Oracle that provides a rich set of connection pool features and extensions over and above standard pooling functionality.
4. **DBCP2**: Apache Commons DBCP (Database Connection Pool) software provides database connection pooling services. DBCP2 is the successor of DBCP and is currently under active development.
5. **Generic DataSource**: It's a general-purpose connection pool and can be configured for any JDBC compatible database.

To change the default connection pool from HikariCP to Tomcat connection pool, you should exclude the HikariCP dependency and include the Tomcat connection pool in your project's build configuration file (like **pom.xml** for Maven or **build.gradle** for Gradle).

In Maven, for example:

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

<exclusions>

<exclusion>

<groupId>com.zaxxer</groupId>

<artifactId>HikariCP</artifactId>

</exclusion>

</exclusions>

</dependency>

<dependency>

<groupId>org.apache.tomcat</groupId>

<artifactId>tomcat-jdbc</artifactId>

</dependency>

</dependencies>

Then, in your **application.properties** or **application.yml**, you can define your datasource settings like this:

spring.datasource.type=org.apache.tomcat.jdbc.pool.DataSource

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

spring.datasource.username=myUser

spring.datasource.password=myPassword

spring.datasource.driverClassName=com.mysql.jdbc.Driver

This way, Spring Boot will use the Tomcat connection pool instead of the default HikariCP.

**Domain-specific language (DSL)**

Spring Data offers a unique approach to implementing repositories. Not only does it provide automatic implementation for commonly used methods, but it also allows for the addition of custom methods. Intriguingly, the method signature alone provides Spring Data with the necessary details to generate an implementation for the method. This method signature essentially forms a domain-specific language (DSL) where persistence specifics are communicated.

Consider the following example:

public interface CustomerDao extends JpaRepository<Customer, Integer> {

// Finder method

public List<Customer> findByAddress(String addr);

}

Repository method names in Spring Data are constructed using a verb, an optional subject, the term "by", and a predicate. For instance, in **findByAddress()**, the verb is "find" and the predicate is "Address". The subject isn’t explicitly stated and is inferred to be a customer.

Spring Data recognizes four verbs in the method name: "get", "read", "find", and "count". The verbs "get", "read", and "find" are interchangeable, all resulting in repository methods that query for data and return objects. However, the "count" verb returns a count of matching objects rather than the objects themselves.

In the method name **findCustomerByFirstNameOrLastName(String first, String last)**, the verb is "find" and the predicate is "FirstNameOrLastName".

Predicates make up the most intriguing part of the method name. A range of comparison operators can be used to compare the property to the parameter:

**Query methods**

Query methods in the Spring Data JPA context. They allow you to define queries directly in your repository interfaces, based on method names.

The core idea is that the creation of a query can be avoided, and instead, the method name is used for creating queries.

1. **IsAfter, After, IsGreaterThan, GreaterThan:** These terms generate queries that check if a certain field is greater than the provided value.
2. **IsGreaterThanEqual, GreaterThanEqual:** These terms create queries that check if a certain field is greater than or equal to the provided value.
3. **IsBefore, Before, IsLessThan, LessThan:** These terms generate queries that check if a certain field is less than the provided value.
4. **IsLessThanEqual, LessThanEqual:** These terms generate queries that check if a certain field is less than or equal to the provided value.
5. **IsBetween, Between:** These terms generate queries that check if a certain field is between two provided values.
6. **IsNull, Null:** These terms generate queries that check if a certain field is null.
7. **IsNotNull, NotNull:** These terms generate queries that check if a certain field is not null.
8. **IsIn, In:** These terms generate queries that check if a certain field's value is in a provided set of values.
9. **IsNotIn, NotIn:** These terms generate queries that check if a certain field's value is not in a provided set of values.
10. **IsStartingWith, StartingWith, StartsWith:** These terms generate queries that check if a certain string field starts with a provided value.
11. **IsEndingWith, EndingWith, EndsWith:** These terms generate queries that check if a certain string field ends with a provided value.
12. **IsContaining, Containing, Contains:** These terms generate queries that check if a certain string field contains a provided value.
13. **IsLike, Like:** These terms generate queries that check if a certain string field matches a provided value, where the provided value can contain wildcards like **%**.
14. **IsNotLike, NotLike:** These terms generate queries that check if a certain string field does not match a provided value, where the provided value can contain wildcards like **%**.
15. **IsTrue, True:** These terms generate queries that check if a certain boolean field is true.
16. **IsFalse, False:** These terms generate queries that check if a certain boolean field is false.
17. **Is, Equals:** These terms generate queries that check if a certain field equals a provided value.
18. **IsNot, Not:** These terms generate queries that check if a certain field does not equal a provided value.

Using these keywords in your method signatures allows Spring Data JPA to understand the kind of query you want to execute and automatically generates the corresponding query.

Sample Entity Product class as follows

@Entity(name = "products")

@DynamicUpdate

public class Product {

@Id

@GeneratedValue

private int productId;

private String productTitle;

private double rating;

private String description;

private String keywords;

@OneToOne(cascade = CascadeType.ALL, fetch = FetchType.EAGER)

@JoinColumn(name = "priceId")

private Price price;

@OneToOne(cascade = CascadeType.ALL, fetch = FetchType.EAGER)

@JoinColumn(name = "stockId")

private Stock stock;

// Getters and setters, toString() method

// ...

}

Here is ProductRepository interface example for above methods

public interface **ProductRepository** extends **JpaRepository**<Product, Integer> {

// other methods...

// find by rating is after a given value

List<Product> findByRatingIsAfter(Double rating);

// find by rating is greater than a given value

List<Product> findByRatingGreaterThan(Double rating);

// find by rating is greater than or equal to a given value

List<Product> findByRatingGreaterThanEqual(Double rating);

// find by rating is before a given value

List<Product> findByRatingIsBefore(Double rating);

// find by rating is less than a given value

List<Product> findByRatingLessThan(Double rating);

// find by rating is less than or equal to a given value

List<Product> findByRatingLessThanEqual(Double rating);

// find by rating is between a given range of values

List<Product> findByRatingBetween(Double startRating, Double endRating);

// find by rating is null

List<Product> findByRatingIsNull();

// find by rating is not null

List<Product> findByRatingIsNotNull();

// find by rating is in a given set of values

List<Product> findByRatingIsIn(List<Double> ratings);

// find by rating is not in a given set of values

List<Product> findByRatingIsNotIn(List<Double> ratings);

// find by productTitle starts with a given value

List<Product> findByProductTitleStartingWith(String prefix);

// find by productTitle ends with a given value

List<Product> findByProductTitleEndingWith(String suffix);

// find by productTitle contains a given value

List<Product> findByProductTitleContaining(String keyword);

// find by productTitle is like a given pattern

List<Product> findByProductTitleLike(String pattern);

// find by productTitle is not like a given pattern

List<Product> findByProductTitleNotLike(String pattern);

// find by isActive is true

List<Product> findByIsActiveTrue();

// find by isActive is false

List<Product> findByIsActiveFalse();

// find by productTitle is equal to a given value

List<Product> findByProductTitleIs(String productTitle);

// find by productTitle is not equal to a given value

List<Product> findByProductTitleIsNot(String productTitle);

// find by productTitle contains a given value and rating is greater than a given value

List<Product> findByProductTitleContainingAndRatingGreaterThan(String keyword, Double rating);

// find by productTitle starts with a given value or description contains a given value

List<Product> findByProductTitleStartingWithOrDescriptionContaining(String prefix, String keyword);

// find by rating is greater than a given value and price is less than a given value

List<Product> findByRatingGreaterThanAndPriceLessThan(Double rating, BigDecimal price);

// find by rating is greater than a given value or isActive is true

List<Product> findByRatingGreaterThanOrIsActiveTrue(Double rating);

// get by productTitle

List<Product> getByProductTitle(String productTitle);

// read by rating greater than

List<Product> readByRatingGreaterThan(Double rating);

// count by description containing a keyword

int countByDescriptionContaining(String keyword);

// get by rating is greater than a given value

List<Product> getByRatingGreaterThan(Double rating);

// read by rating is less than a given value

List<Product> readByRatingLessThan(Double rating);

// count by rating is greater than or equal to a given value

int countByRatingGreaterThanEqual(Double rating);

// and so on for other properties and conditions

// get by productTitle and rating

List<Product> getByProductTitleAndRating(String productTitle, Double rating);

// read by productTitle or description

List<Product> readByProductTitleOrDescription(String productTitle, String description);

// count by productTitle

int countByProductTitle(String productTitle);

}

// and so on for other properties and conditions

// find price by productId

@Query("SELECT p.price FROM Product p WHERE p.productId = :productId")

Price findPriceByProductId(Integer productId);

// find stock by productId

@Query("SELECT p.stock FROM Product p WHERE p.productId = :productId")

Stock findStockByProductId(Integer productId);

}

**Query annotation**

The **@Query** annotation in Spring Data JPA allows you to define custom queries using JPQL (Java Persistence Query Language) or native SQL. It provides flexibility when the query logic cannot be expressed using the method name conventions provided by Spring Data JPA.

Here's an overview of the **@Query** annotation and its usage:

@Query("SELECT p FROM Product p WHERE p.rating > :rating")

List<Product> findProductsByRatingGreaterThan(@Param("rating") double rating);

In the above example, **@Query** is used to define a custom JPQL query. The query is specified within the double quotes. The **SELECT** statement retrieves the **Product** entities that have a rating greater than the given parameter **rating**. **:rating** is a named parameter that can be bound to the method parameter using **@Param** annotation.

The **@Param** annotation in Spring Data JPA is optional when the method parameter name matches the named parameter used in the query. If the method parameter name and the named parameter in the query have the same name, Spring Data JPA can automatically associate them without the need for the **@Param** annotation.

For example, consider the following query method without using **@Param**:

@Query("SELECT p FROM Product p WHERE p.rating > :rating")

List<Product> findProductsByRatingGreaterThan(double rating);

In this case, the **rating** method parameter is automatically bound to the named parameter **:rating** in the query because they have the same name. Spring Data JPA can infer the association based on the method signature and the query's named parameter.

However, if the method parameter name differs from the named parameter in the query, you need to use the **@Param** annotation to explicitly indicate the parameter association:

@Query("SELECT p FROM Product p WHERE p.rating > :productRating")

List<Product> findProductsByRatingGreaterThan(@Param("productRating") double rating);

Here, **@Param("productRating")** is used to specify that the method parameter **rating** should be bound to the named parameter **:productRating** in the query.

So, while the **@Param** annotation is optional when the method parameter name matches the named parameter in the query, it becomes necessary when there is a name mismatch or if you want to provide a different name for the parameter binding. Using **@Param** provides clarity and explicitness in the parameter association when needed.

**Positional Parameters in Query annotation**

Positional parameters in Spring Data JPA allow you to use a question mark (**?**) followed by an index to specify parameters in your custom queries. With positional parameters, the order of the method parameters corresponds to the order of the parameters in the query.

Here's an example that demonstrates the usage of positional parameters:

@Query("SELECT p FROM Product p WHERE p.rating > ?1 AND p.price < ?2")

List<Product> findProductsByRatingAndPrice(double minRating, BigDecimal maxPrice);

In the above example, **?1** and **?2** are positional parameters representing the first and second parameters of the method, respectively. The **?1** corresponds to **minRating**, and **?2** corresponds to **maxPrice**.

When executing the query, the values of the method parameters will be substituted in the query based on their positions.

**Which one is reamended and why?**

Named parameters are generally recommended over positional parameters in Spring Data JPA due to several reasons: t's important to note that positional parameters are indexed starting from 1, not 0. Make sure the position in the query matches the order of the method parameters.

1. Readability and Maintainability: Named parameters provide better readability, making the query more understandable. They can be directly associated with method parameters using the **@Param** annotation, improving code maintainability and reducing the chances of parameter mismatches.
2. Flexibility: Named parameters allow you to reorder or omit parameters in the method signature without affecting the query. This flexibility is crucial for complex queries with multiple parameters when you need to modify the method signature.
3. Safety and Type Safety: Named parameters offer an extra layer of safety. With named parameters, parameter values are bound to their respective named placeholders in the query based on their names, helping prevent errors caused by mismatched or missing parameters. Additionally, named parameters support type safety since they are associated with specific method parameters.
4. Query Evolution: Named parameters make it easier to evolve queries over time. You can add or modify conditions in the query without changing the method signature. This decoupling of query logic and method signature allows independent modification of the query.
5. Tool Support: Named parameters are well-supported by IDEs and tools, providing code assistance and validation. IDEs can offer suggestions for named parameters, ensuring correct parameter names are used and reducing the likelihood of errors.

**Multiple database support**

Spring Boot can easily connect to multiple databases. Let's consider an example where we will connect to two databases, **db1** and **db2**. In this example, we will use MySQL as the database.

**Step 1: Update the pom.xml**

Include the JPA and MySQL driver dependencies.

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

</dependency>

**Step 2: Configure application.properties**

Update the **application.properties** file with connection details for **db1** and **db2**.

# DB1

spring.datasource.db1.url=jdbc:mysql://localhost/db1

spring.datasource.db1.username=user1

spring.datasource.db1.password=pass1

spring.datasource.db1.driver-class-name=com.mysql.cj.jdbc.Driver

# DB2

spring.datasource.db2.url=jdbc:mysql://localhost/db2

spring.datasource.db2.username=user2

spring.datasource.db2.password=pass2

spring.datasource.db2.driver-class-name=com.mysql.cj.jdbc.Driver

**Step 3: Configure DataSources**

Create two configuration classes, each configuring a **DataSource**, **EntityManager**, **TransactionManager**, and **JpaRepository**.

@Configuration

@EnableTransactionManagement

@EnableJpaRepositories(

basePackages = "com.example.repository.db1",

entityManagerFactoryRef = "db1EntityManager",

transactionManagerRef = "db1TransactionManager"

)

public class DB1Config {

@Primary

@Bean(name = "db1DataSource")

@ConfigurationProperties(prefix = "spring.datasource.db1")

public DataSource dataSource() {

return DataSourceBuilder.create().build();

}

@Primary

@Bean(name = "db1EntityManager")

public LocalContainerEntityManagerFactoryBean entityManagerFactory(EntityManagerFactoryBuilder builder) {

return builder

.dataSource(dataSource())

.packages("com.example.model.db1")

.persistenceUnit("db1")

.build();

}

@Primary

@Bean(name = "db1TransactionManager")

public PlatformTransactionManager transactionManager() {

return new JpaTransactionManager(entityManagerFactory(builder).getObject());

}

}

**Step 4: Create Entities and Repositories**

Create your entities and repositories under the packages you specified in your configuration (**com.example.model.db1**, **com.example.repository.db1**, **com.example.model.db2**, and **com.example.repository.db2**).

Ensure that each repository extends **JpaRepository** and is placed in the correct package so that it can use the correct **EntityManagerFactory** and **PlatformTransactionManager**.

This configuration allows you to connect to multiple databases within the same application.

Assuming you have an **Employee** entity in each database and you have put these entities in the "com.example.model.db1" and "com.example.model.db2" packages respectively, the repositories would look like this:

**EmployeeRepository for DB1**

Create a new interface **EmployeeRepository** in the package **com.example.repository.db1**. Here is the code:

package com.example.repository.db1;

import com.example.model.db1.Employee;

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.stereotype.Repository;

@Repository

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

// Define your custom methods here if needed

}

**EmployeeRepository for DB2**

Similarly, create a new interface **EmployeeRepository** in the package **com.example.repository.db2**. Here is the code:

package com.example.repository.db2;

import com.example.model.db2.Employee;

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.stereotype.Repository;

@Repository

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

// Define your custom methods here if needed

}

Remember to adjust your model classes, their locations, and the identifier type based on your application. JpaRepository provides basic CRUD operations as well as some common operations such as **findAll**, **deleteAll**, **count**, etc. If you need more complex queries, you can define them as methods in your repository interface. Spring Data JPA will create the implementation automatically.

**EmployeeService** that interacts with both databases:

import com.example.model.db1.Employee as EmployeeDb1;

import com.example.model.db2.Employee as EmployeeDb2;

import com.example.repository.db1.EmployeeRepository as EmployeeRepositoryDb1;

import com.example.repository.db2.EmployeeRepository as EmployeeRepositoryDb2;

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 EmployeeService {

private final EmployeeRepositoryDb1 employeeRepositoryDb1;

private final EmployeeRepositoryDb2 employeeRepositoryDb2;

@Autowired

public EmployeeService(EmployeeRepositoryDb1 employeeRepositoryDb1, EmployeeRepositoryDb2 employeeRepositoryDb2) {

this.employeeRepositoryDb1 = employeeRepositoryDb1;

this.employeeRepositoryDb2 = employeeRepositoryDb2;

}

@Transactional("db1TransactionManager")

public List<EmployeeDb1> getAllEmployeesDb1() {

return employeeRepositoryDb1.findAll();

}

@Transactional("db2TransactionManager")

public List<EmployeeDb2> getAllEmployeesDb2() {

return employeeRepositoryDb2.findAll();

}

@Transactional("db1TransactionManager")

public Optional<EmployeeDb1> getEmployeeByIdDb1(Long id) {

return employeeRepositoryDb1.findById(id);

}

@Transactional("db2TransactionManager")

public Optional<EmployeeDb2> getEmployeeByIdDb2(Long id) {

return employeeRepositoryDb2.findById(id);

}

@Transactional("db1TransactionManager")

public EmployeeDb1 saveEmployeeDb1(EmployeeDb1 employee) {

return employeeRepositoryDb1.save(employee);

}

@Transactional("db2TransactionManager")

public EmployeeDb2 saveEmployeeDb2(EmployeeDb2 employee) {

return employeeRepositoryDb2.save(employee);

}

@Transactional("db1TransactionManager")

public void deleteEmployeeDb1(Long id) {

employeeRepositoryDb1.deleteById(id);

}

@Transactional("db2TransactionManager")

public void deleteEmployeeDb2(Long id) {

employeeRepositoryDb2.deleteById(id);

}

}

In this service, each method is annotated with **@Transactional** and a specific transaction manager for the operation. This ensures that each operation is run within its own transaction and its respective database.

This service provides basic CRUD operations (read, write, and delete) for employees in both databases. Depending on your use case, you might need to add more methods or additional logic within these methods.

The **@Transactional** annotation is not always mandatory, but it is highly recommended for methods that perform operations which require consistency, such as creating, updating, or deleting records in a database. Using **@Transactional** ensures that these operations are handled as a single unit of work, and either all changes are committed to the database, or if an error occurs, all changes are rolled back.

When using multiple databases, it becomes especially important to manage transactions carefully. In our example, specifying the transaction manager in the **@Transactional** annotation ensures that each operation uses the correct transaction manager for its database. If you don't specify the transaction manager, Spring will use the **@Primary** transaction manager by default, which may not be the correct one for the operation.

For read-only operations, such as retrieving records from the database, **@Transactional** can improve performance by enabling certain optimizations (like read-only database optimizations), and ensuring consistency (you get a consistent snapshot for your entire transaction). However, it's not always necessary if you don't need these benefits, and leaving it out can slightly reduce overhead.

In summary, while **@Transactional** is not always strictly necessary, using it properly can help ensure data consistency, prevent subtle bugs, and sometimes improve performance.