UNIT III

Spring Data JPA with Boot: Limitations of JDBC API, Why Spring Data JPA, Spring Data JPA with Spring Boot, Spring Data JPA Configuration, Pagination and Sorting, Query Approaches, Named Queries and Query, Why Spring Transaction, Spring Declarative Transaction, Update Operation in Spring Data JPA, Custom Repository Implementation, Best Practices - Spring Data JPA

Spring Data JPA with Boot

1: Limitations of JDBC API:

The JDBC API, while fundamental for interacting with relational databases, has several limitations:

1. **Boilerplate Code:** JDBC requires a significant amount of boilerplate code for tasks such as opening connections, creating statements, handling exceptions, and managing transactions.
2. **Complex Mapping:** Mapping SQL result sets to Java objects can be complex and error-prone, especially for larger and more complex data models.
3. **Manual Query Construction:** JDBC requires manual construction of SQL queries, which can lead to potential security vulnerabilities such as SQL injection if not handled carefully.
4. **Lack of Object-Relational Mapping (ORM):** JDBC does not provide built-in support for object-relational mapping, making it challenging to map database entities to Java objects

2. Why Spring Data JPA (Spring Data repository support for JPA):

Introduction to Spring Data JPA

Spring Data JPA is a part of the larger Spring Data project and provides a higher level of abstraction over JDBC, addressing its limitations. It simplifies the development of data access layers by leveraging the Java Persistence API (JPA) and providing additional features such as:

1. **Object-Relational Mapping (ORM):** Spring Data JPA simplifies the mapping of Java objects to database entities, reducing the need for manual mapping and providing a more intuitive approach to data access.

2. **Repository Abstraction:** It offers a repository abstraction that allows developers to define interfaces for data access, reducing the amount of boilerplate code required for common CRUD operations.

3. **Query Methods:** Spring Data JPA enables the creation of query methods based on method naming conventions, reducing the need for manual query construction and providing type-safe queries.

4. **Automatic Query Generation:** It can automatically generate queries based on method names, reducing the need for writing explicit SQL queries.

Comparison with Spring Data JDBC

Spring Data JPA is often compared with Spring Data JDBC, which is a persistence framework that is not as complex as Spring Data JPA. While Spring Data JDBC provides a simpler alternative to JPA, it lacks certain features such as cache, lazy loading, and write-behind, which are available in JPA. Spring Data JDBC is suitable for scenarios where a simpler persistence framework is preferred and the full capabilities of JPA are not required.

Spring Data JPA addresses the limitations of the JDBC API by providing a higher level of abstraction, simplifying data access, and offering additional features such as object-relational mapping and repository abstraction.

3. Spring Data JPA with Spring Boot:

Spring Data JPA Configuration and Custom Repository Implementation:

Spring Data JPA provides a powerful and efficient way to interact with databases in Spring Boot applications. It simplifies the data access layer by providing a higher level of abstraction over JDBC and JPA, and it also allows for the implementation of custom repositories to address specific data access requirements.

4. Spring Data JPA Configuration:

Spring Data JPA configuration involves setting up the necessary components to enable data access using JPA. Here are the key aspects of Spring Data JPA configuration:

1. **Entity Classes:** Define entity classes that represent the data model and are mapped to database tables. These classes are annotated with `@Entity` and may include additional annotations for mapping relationships and defining constraints.

2. **Repository Interfaces:** Create repository interfaces that extend the `JpaRepository` interface provided by Spring Data JPA. These interfaces define methods for common CRUD operations and can also include custom query methods.

3. **EntityManager and DataSource Configuration:** Configure the EntityManager and DataSource beans to establish the connection with the database. Spring Boot's auto-configuration capabilities simplify this process, requiring minimal configuration when using default settings.

4. **Transaction Management:** Configure transaction management to ensure the consistency and integrity of database operations. Spring Data JPA integrates seamlessly with Spring's transaction management capabilities, allowing for declarative transaction demarcation.

Configure Spring Data JPA in Spring Application with Example

Requirements: STS IDE, MySQL workbench, Java 8+
Create a spring boot project in STS. Give project name & select add required dependencies(Spring JPA, MySQL driver, Spring web) shown in attached **pom.xml**.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
```

```xml
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.6.1</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.example</groupId>
    <artifactId>ex</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>ex</name>
    <description>Demo project for Spring Boot</description>
    <properties>
        <java.version>11</java.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>
            <scope>runtime</scope>
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
```

```
            </plugin>
        </plugins>
    </build>
</project>
```

5. Pagination and Sorting in Spring Data JPA:

Pagination and sorting are crucial for presenting large datasets to users in smaller, organized chunks. In Spring Data JPA, these features are implemented to enhance the user experience and optimize data retrieval. Pagination allows the presentation of data in smaller, manageable portions, while sorting enables users to view the data in an organized manner based on specific criteria.

Implementation of Pagination and Sorting:

Paging and Sorting Repository Interface:
Spring Data JPA provides the Paging And Sorting Repository interface, which extends the Crud Repository interface. This interface offers built-in support for pagination and sorting; allowing developers to retrieve paginated and sorted subsets of data with ease.

Custom Query Methods:
Developers can define custom query methods in repository interfaces to support pagination and sorting. These methods can accept parameters for page size, page number, and sorting criteria, providing flexibility in data retrieval.

@Query Annotation:
The @Query annotation in Spring Data JPA allows the definition of custom queries using JPQL (Java Persistence Query Language) and native SQL. This enables developers to implement specific pagination and sorting logic tailored to the application's requirements.

6. Query Approaches in Spring Data JPA

Spring Data JPA supports various query approaches, including JPQL, native SQL, and dynamic queries using Specifications. These approaches offer flexibility and efficiency in retrieving data based on specific conditions and criteria.

JPQL Queries: JPQL provides a platform-independent query language for retrieving objects from the database. It allows developers to define queries based on entity attributes and relationships, providing a high level of abstraction over database-specific SQL.

Native SQL Queries: Spring Data JPA allows the execution of native SQL queries, providing direct access to the underlying database. This approach is useful for scenarios where complex or database-specific queries are required.

Specifications with Pagination: Specifications in Spring Data JPA enable the dynamic construction of queries based on specific conditions. When combined with Pageable, Specifications allow for the implementation of dynamic queries with pagination and sorting.

In Spring Data JPA, pagination, sorting, and query approaches play a vital role in efficiently retrieving and presenting data. By leveraging the built-in support for pagination and sorting, along with the flexibility of custom query methods and query approaches, developers can create robust and user-friendly data access layers in Spring Boot applications.

7. Named Queries and Query:

Defining a Named Query with JPA

Named queries are one of the core concepts in JPA. They enable you to declare a query in your persistence layer and reference it in your business code. That makes it easy to reuse an existing query. It also enables you to separate the definition of your query from your business code.

You can define a named query using a *@NamedQuery* annotation on an entity class or using a <*named-query* /> element in your XML mapping. In this article, I will show you the annotation-based mapping. It's the by far the most common approach to creating a named query.

When you define a named query, you can provide a JPQL query or a native SQL query in very similar ways. Let's take a look at both options.

Defining a Named JPL Query

The JPA specification defines its own query language. It's called JPQL, and its syntax is similar to SQL. But there are 2 essential differences between these 2:

1. You define your JPQL query based on your entity model. When you execute it, your persistence provider generates a SQL query based on your entity mappings and the provided JPQL statement. That enables you to define database-independent queries but also limits you to the features supported by your persistence provider.
2. JPQL supports only a small subset of the SQL standard and almost no database-specific features.

The definition of a named JPQL query is pretty simple. You just have to annotate one of your entity classes with *@NamedQuery* and provide 2 *String*s for the *name* and *query* attributes.

The *name* of your query has to be unique within your persistence context. You will use it in your business code or repository definition to reference the query.

The name doesn't have to follow any conventions if you want to reference the query programmatically. But if you're going to reference it in a Spring Data JPA repository, the name should start with the name of the entity class, followed by a "." and the name of the repository method.

The value of the *query* attribute has to be a String that contains a valid JPQL statement. If your query returns an entity, you can define your projection implicitly, as you can see in the *Author.findByFirstName* query. The *Author.findByFirstNameAndLastName* query contains a SELECT clause to define the projection explicitly.

JPQL, of course, supports way more features than I use in these simple examples. You can learn more about it in my Ultimate Guide to JPQL Queries with JPA and Hibernate.

```
1@Entity
2@NamedQuery(name = "Author.findByFirstName", query = "FROM Author WHERE firstName = ?1")
3@NamedQuery(name = "Author.findByFirstNameAndLastName", query = "SELECT a FROM Author a WHEF
4public class Author { ... }
```

If you want to define multiple JPQL queries and use at least JPA 2.2 or Hibernate 5.2, you can annotate your class with multiple *@NamedQuery* annotations. If you are using an older JPA or Hibernate version, you need to wrap your *@NamedQuery* annotation within a *@NamedQueries* annotation.

Defining a Named Native Query

Native SQL queries are more powerful and flexible than JPQL queries. Your persistence provider doesn't parse these queries and sends them directly to the database. That enables you to use all SQL

features supported by your database. But you also have to handle the different database dialects if you need to support multiple DBMS.

You can define a named native query in almost the same way as you specify a named JPQL query. The 3 main differences are:

1. You need to use a *@NamedNativeQuery* instead of a *@NamedQuery* annotation.
2. The value of the query attribute has to be an SQL statement instead of a JPQL statement.
3. You can define an entity class or a reference to an *@SqlResultSetMapping* that will be used to map the result of your query. Spring Data JPA can provide a set of default mappings so that you often don't need to specify it.

Here you can see the same queries as in the previous example. But this time, they are defined as native SQL queries instead of JPQL queries.

```
  @Entity
1@NamedNativeQuery(name = "Author.findByFirstName", query = "SELECT *
2FROM author WHERE first_name = ?", resultClass = Author.class)
3@NamedNativeQuery(name = "Author.findByFirstNameAndLastName",
4query = "SELECT * FROM author WHERE first_name = ? AND last_name = ?", resultClass = Author.class)
  public class Author { ... }
```

Executing a Named Query Programmatically with JPA

Using JPA's *EntityManager*, you can run named native and named JPQL queries in the same way:

1. You call the *createNamedQuery* method on the *EntityManager* with the name of the named query you want to execute. That gives you an instance of a *Query* or *TypedQuery* interface.
2. You then call the setParameter method on the returned interface for each bind parameter used in your query.
3. As a final step, you call the *getSingleResult* or *getResultSet* method on the *Query* or *TypedQuery* interface. That executes the query and returns 1 or multiple result set records.

Here you can see the required code to execute the *Author.findByFirstName* query that we defined in the 2 previous examples.

```
1       Query q = em.createNamedQuery("Author.findByFirstName");
2       q.setParameter(1, "Thorben");
3       List a = q.getResultList();
```

Before you run this code, you should activate the logging of SQL statements. You can then see the executed SQL statement and the used bind parameter values in your log file. In this example, I called the *@NamedNativeQuery* version of the previously shown *Author.findByFirstName* query.

```
1  2019-06-24 19:20:32.061 DEBUG 10596 --- [          main] org.hibernate.SQL                :
2     SELECT
3        *
4     FROM
```

```
5         author
6      WHERE
7         first_name = ?
8  2019-06-24 19:20:32.073 TRACE 10596 --- [       main] o.h.type.descriptor.sql.BasicBinder      : binding para
9  2019-06-24 19:20:32.116 TRACE 10596 --- [       main] o.h.type.descriptor.sql.BasicExtractor : extracted val
102019-06-24 19:20:32.118 TRACE 10596 --- [       main] o.h.type.descriptor.sql.BasicExtractor  : extracted valu
112019-06-24 19:20:32.119 TRACE 10596 --- [       main] o.h.type.descriptor.sql.BasicExtractor  : extracted valu
122019-06-24 19:20:32.121 TRACE 10596 --- [       main] o.h.type.descriptor.sql.BasicExtractor  : extracted valu
```

Referencing a Named Query in a Spring Data JPA repository

As you have seen in the previous example, executing a named query using JPA's *EntityManager* isn't complicated, but it requires multiple steps.

Spring Data JPA takes care of that if you reference a named query in your repository definition. Doing that is extremely simple if you follow Spring Data's naming convention. The name of your query has to start with the name of your entity class, followed by "." and the name of your repository method.

In the previous examples, I defined the named queries *Author.findByFirstName* and *Author.findByFirstNameAndLastName* as JPQL and native queries. You can reference both versions of these queries by adding the methods *findByFirstName* and *findByFirstNameAndLastName* to the *AuthorRepository*.

```
1      public interface AuthorRepository extends JpaRepository<Author, Long> {
2
3         List<Author> findByFirstName(String firstName);
4
5         List<Author> findByFirstNameAndLastName(String firstName, String lastName);
6
7      }
```

You can then inject an *AuthorRepository* instance in your business code and call the repository methods to execute the named queries.

As you can see in the following code snippet, you can use these repository methods in the same way as a repository method that executes a derived query or a declared query. Spring Data JPA handles the instantiation of the named query, sets the bind parameter values, executes the query, and maps the result.

```
1      List<Author> a = authorRepository.findByFirstName("Thorben");
```

8. Why Spring Transaction
Spring Declarative Transaction

Spring Data JPA provides a robust mechanism for managing transactions and executing update operations in Spring Boot applications.

Spring Transaction Management with Spring Data JPA
Spring Data JPA seamlessly integrates with Spring's transaction management capabilities, providing a straightforward approach to handling transactions in Spring Boot applications. The @Transactional annotation plays a pivotal role in defining transactional boundaries and ensuring the consistency and integrity of database operations.

Key Aspects of Spring Transaction Management:

@Transactional Annotation: By annotating interfaces, classes, or methods with @Transactional, developers can specify the transactional context in which the annotated code should be executed. This annotation allows for the automatic management of transactions, including transaction initiation, commit, and rollback.

Declarative Transaction Management: Spring Data JPA supports declarative transaction management, where transactional behavior is defined using annotations without the need for explicit transaction handling code. This simplifies the implementation of transactional logic and promotes a cleaner and more maintainable codebase.

Transactional Semantics: Spring Data JPA supports various transactional semantics, including read-only transactions, which optimize transaction behavior for read-only operations, and programmatic transaction management for fine-grained control over transaction boundaries.

9. Update Operation in Spring Data JPA:

Spring Data JPA facilitates the execution of update operations on entities, allowing for the modification of persistent data in the database. The update operations can be performed using custom query methods or by leveraging the built-in CRUD methods provided by Spring Data JPA repositories.

Custom Update Methods:
Developers can define custom update methods in repository interfaces using the @Modifying and @Query annotations. These custom methods enable the execution of update queries tailored to specific data modification requirements.

Automatic Flushing and Unit of Work:
In Spring Data JPA, changes made to managed entities are automatically flushed to the database at the end of the Unit of Work, typically at the end of the current transaction. This ensures that modifications to entities are propagated to the database, maintaining data consistency.

Spring Data JPA, in conjunction with Spring Boot, simplifies transaction management and update operations in Spring applications. By leveraging the @Transactional annotation and custom update methods, developers can ensure the integrity of database operations and efficiently execute data modification tasks.

10. Pagination, Sorting, and Query Approaches.

Pagination, sorting, and query approaches are essential aspects of data retrieval and presentation in Spring Data JPA. They enable the efficient handling of large datasets and provide users with organized and manageable views of the data. Here's a detailed explanation based on the provided search results:

Pagination and Sorting in Spring Data JPA

Pagination and sorting are crucial for presenting large datasets to users in smaller, organized chunks. In Spring Data JPA, these features are implemented to enhance the user experience and optimize data retrieval. Pagination allows the presentation of data in smaller, manageable portions, while sorting enables users to view the data in an organized manner based on specific criteria.

Implementation of Pagination and Sorting

1. **PagingAndSortingRepository Interface:** Spring Data JPA provides the `PagingAndSortingRepository` interface, which extends the `CrudRepository` interface. This interface offers built-in support for pagination and sorting, allowing developers to retrieve paginated and sorted subsets of data with ease.

2. **Custom Query Methods:** Developers can define custom query methods in repository interfaces to support pagination and sorting. These methods can accept parameters for page size, page number, and sorting criteria, providing flexibility in data retrieval.

3. **@Query Annotation:** The @Query annotation in Spring Data JPA allows the definition of custom queries using JPQL (Java Persistence Query Language) and native SQL. This enables developers to implement specific pagination and sorting logic tailored to the application's requirements.

Query Approaches in Spring Data JPA

Spring Data JPA supports various query approaches, including JPQL, native SQL, and dynamic queries using Specifications. These approaches offer flexibility and efficiency in retrieving data based on specific conditions and criteria.

1. **JPQL Queries:** JPQL provides a platform-independent query language for retrieving objects from the database. It allows developers to define queries based on entity attributes and relationships, providing a high level of abstraction over database-specific SQL.

2. **Native SQL Queries:** Spring Data JPA allows the execution of native SQL queries, providing direct access to the underlying database. This approach is useful for scenarios where complex or database-specific queries are required.

3. **Specifications with Pagination:** Specifications in Spring Data JPA enable the dynamic construction of queries based on specific conditions. When combined with Pageable, Specifications allow for the implementation of dynamic queries with pagination and sorting.

In Spring Data JPA, pagination, sorting, and query approaches play a vital role in efficiently retrieving and presenting data. By leveraging the built-in support for pagination and sorting, along with the flexibility of custom query methods and query approaches, developers can create robust and user-friendly data access layers in Spring Boot applications.

11. Spring Transaction and Update Operation in Spring Data JPA:

Spring Data JPA provides a robust mechanism for managing transactions and executing update operations in Spring Boot applications.

Spring Transaction Management with Spring Data JPA

Spring Data JPA seamlessly integrates with Spring's transaction management capabilities, providing a straightforward approach to handling transactions in Spring Boot applications. The `@Transactional` annotation plays a pivotal role in defining transactional boundaries and ensuring the consistency and integrity of database operations.

Key Aspects of Spring Transaction Management

1. **@Transactional Annotation:** By annotating interfaces, classes, or methods with `@Transactional`, developers can specify the transactional context in which the annotated code should be executed. This annotation allows for the automatic management of transactions, including transaction initiation, commit, and rollback.

2. **Declarative Transaction Management:** Spring Data JPA supports declarative transaction management, where transactional behavior is defined using annotations without the need for explicit transaction handling code. This simplifies the implementation of transactional logic and promotes a cleaner and more maintainable codebase.

3. **Transactional Semantics:** Spring Data JPA supports various transactional semantics, including read-only transactions, which optimize transaction behavior for read-only operations, and programmatic transaction management for fine-grained control over transaction boundaries.

Update Operation in Spring Data JPA

Spring Data JPA facilitates the execution of update operations on entities, allowing for the modification of persistent data in the database. The update operations can be performed using custom query methods or by leveraging the built-in CRUD methods provided by Spring Data JPA repositories.

Custom Update Methods

Developers can define custom update methods in repository interfaces using the `@Modifying` and `@Query` annotations. These custom methods enable the execution of update queries tailored to specific data modification requirements.

Automatic Flushing and Unit of Work

In Spring Data JPA, changes made to managed entities are automatically flushed to the database at the end of the Unit of Work, typically at the end of the current transaction. This ensures that modifications to entities are propagated to the database, maintaining data consistency.

Spring Data JPA, in conjunction with Spring Boot, simplifies transaction management and update operations in Spring applications. By leveraging the `@Transactional` annotation and custom update methods, developers can ensure the integrity of database operations and efficiently execute data modification tasks.

12. Custom Repository Implementation:

Custom Repository Implementations

Spring Data provides various options to create query methods with little coding. But when those options don't fit your needs you can also provide your own custom implementation for repository methods. This section describes how to do that.

Customizing Individual Repositories

To enrich a repository with custom functionality, you must first define a fragment interface and an implementation for the custom functionality, as follows:

Interface for custom repository functionality

```
interface CustomizedUserRepository {
```

```
  void someCustomMethod(User user);
}
```

Implementation of custom repository functionality

```
class CustomizedUserRepositoryImpl implements CustomizedUserRepository {

  public void someCustomMethod(User user) {
    // Your custom implementation
  }
}
```

The most important part of the class name that corresponds to the fragment interface is the Impl postfix.

The implementation itself does not depend on Spring Data and can be a regular Spring bean. Consequently, you can use standard dependency injection behavior to inject references to other beans (such as a JdbcTemplate), take part in aspects, and so on.

Then you can let your repository interface extend the fragment interface, as follows:

Changes to your repository interface

```
interface UserRepository extends CrudRepository<User, Long>, CustomizedUserRepository {

  // Declare query methods here
}
```

Extending the fragment interface with your repository interface combines the CRUD and custom functionality and makes it available to clients.

Spring Data repositories are implemented by using fragments that form a repository composition. Fragments are the base repository, functional aspects (such as QueryDsl), and custom interfaces along with their implementations. Each time you add an interface to your repository interface, you enhance the composition by adding a fragment. The base repository and repository aspect implementations are provided by each Spring Data module.

The following example shows custom interfaces and their implementations:

Fragments with their implementations

```
interface HumanRepository {
  void someHumanMethod(User user);
}

class HumanRepositoryImpl implements HumanRepository {

  public void someHumanMethod(User user) {
    // Your custom implementation
  }
}

interface ContactRepository {

  void someContactMethod(User user);

  User anotherContactMethod(User user);
```

```
}

class ContactRepositoryImpl implements ContactRepository {

  public void someContactMethod(User user) {
    // Your custom implementation
  }

  public User anotherContactMethod(User user) {
    // Your custom implementation
  }
}
```

The following example shows the interface for a custom repository that extends CrudRepository:
Changes to your repository interface

```
interface UserRepository extends CrudRepository<User, Long>, HumanRepository,
ContactRepository {

  // Declare query methods here
}
```

Repositories may be composed of multiple custom implementations that are imported in the order of their declaration. Custom implementations have a higher priority than the base implementation and repository aspects. This ordering lets you override base repository and aspect methods and resolves ambiguity if two fragments contribute the same method signature. Repository fragments are not limited to use in a single repository interface. Multiple repositories may use a fragment interface, letting you reuse customizations across different repositories.
The following example shows a repository fragment and its implementation:
Fragments overriding save(…)

```
interface CustomizedSave<T> {
  <S extends T> S save(S entity);
}

class CustomizedSaveImpl<T> implements CustomizedSave<T> {

  public <S extends T> S save(S entity) {
    // Your custom implementation
  }
}
```

The following example shows a repository that uses the preceding repository fragment:
Customized repository interfaces

```
interface UserRepository extends CrudRepository<User, Long>, CustomizedSave<User> {
}

interface PersonRepository extends CrudRepository<Person, Long>, CustomizedSave<Person> {
}
```

Configuration

The repository infrastructure tries to autodetect custom implementation fragments by scanning for classes below the package in which it found a repository. These classes need to follow the naming convention of appending a postfix defaulting to Impl.

The following example shows a repository that uses the default postfix and a repository that sets a custom value for the postfix:

Example 1. Configuration example

- **Java**
- **XML**

```
@EnableJpaRepositories(repositoryImplementationPostfix = "MyPostfix")
class Configuration { … }
```

The first configuration in the preceding example tries to look up a class called com.acme.repository.CustomizedUserRepositoryImpl to act as a custom repository implementation. The second example tries to look up com.acme.repository.CustomizedUserRepositoryMyPostfix.

Resolution of Ambiguity

If multiple implementations with matching class names are found in different packages, Spring Data uses the bean names to identify which one to use.

Given the following two custom implementations for the CustomizedUserRepository shown earlier, the first implementation is used. Its bean name is customizedUserRepositoryImpl, which matches that of the fragment interface (CustomizedUserRepository) plus the postfix Impl.

Example 2. Resolution of ambiguous implementations

```
package com.acme.impl.one;

class CustomizedUserRepositoryImpl implements CustomizedUserRepository {

  // Your custom implementation
}

package com.acme.impl.two;

@Component("specialCustomImpl")
class CustomizedUserRepositoryImpl implements CustomizedUserRepository {

  // Your custom implementation
}
```

If you annotate the UserRepository interface with @Component("specialCustom"), the bean name plus Impl then matches the one defined for the repository implementation in com.acme.impl.two, and it is used instead of the first one.

13. Best Practices - Spring Data JPA

1. Efficient Application Data Access Layers

Mastering Spring Data JPA can significantly enhance the ability to develop efficient application data access layers. By following best practices, such as using appropriate repository interfaces, leveraging specifications and criteria builders for dynamic queries, and optimizing queries and transactions, developers can ensure that applications perform well and scale effectively

2. Entity Class Design

Adhering to best practices for designing entity classes while utilizing JPA within a Spring Boot framework is essential. This includes creating abstract Auditable classes for auditing purposes and utilizing annotations such as `@EntityListeners` and `@MappedSuperclass` to ensure integrity, performance, and scalability of applications

3. Performance Optimization

Utilize caching mechanisms provided by Spring Data JPA, such as the `@Cacheable` annotation, to optimize performance. Additionally, consider using cache managers like `ConcurrentMapCacheManager` and `RedisCacheManager` to implement cache expiration time for improved efficiency

4. Lazy Loading and Eager Fetching

Carefully manage lazy loading and eager fetching strategies for relationships in JPA to avoid performance pitfalls. Understanding the implications of lazy loading and eager fetching can help optimize database interactions and reduce the number of unnecessary queries.

5. Control Data Access

Follow good practices to limit the loading of unnecessary objects and optimize performance when using Spring Data JPA. Utilize the tools provided by Spring Data JPA to improve control of data access and reduce the impact of data retrieval on application performance

6. Use Metamodel Classes

When working with JPAs APIs, such as the Criteria API or the Entity Graph API, prefer using JPAs Metamodel classes over String constants. This approach enhances maintainability and readability of the persistence layer.By incorporating these best practices, developers can optimize the performance, maintainability, and efficiency of Spring Data JPA applications, ensuring that they are scalable and responsive.

API & MICROSERVICES

IMPORTANT QUESTIONS - UNIT-3

1. Explain Limitations of JDBC API,
2. Why Spring Data JPA and ecplain Spring Data JPA with Spring Boot,
3. Explain Spring Data JPA Configuration,
4. Explain Pagination and Sorting,
5. Explain Query Approaches,
6. Explain Named Queries and Query,
7. Why Spring Transaction and explain Spring Declarative Transaction,
8. Explain Update Operation in Spring Data JPA,
9. Explain Custom Repository Implementation,
10. Explain Best Practices - Spring Data JPA