JPA

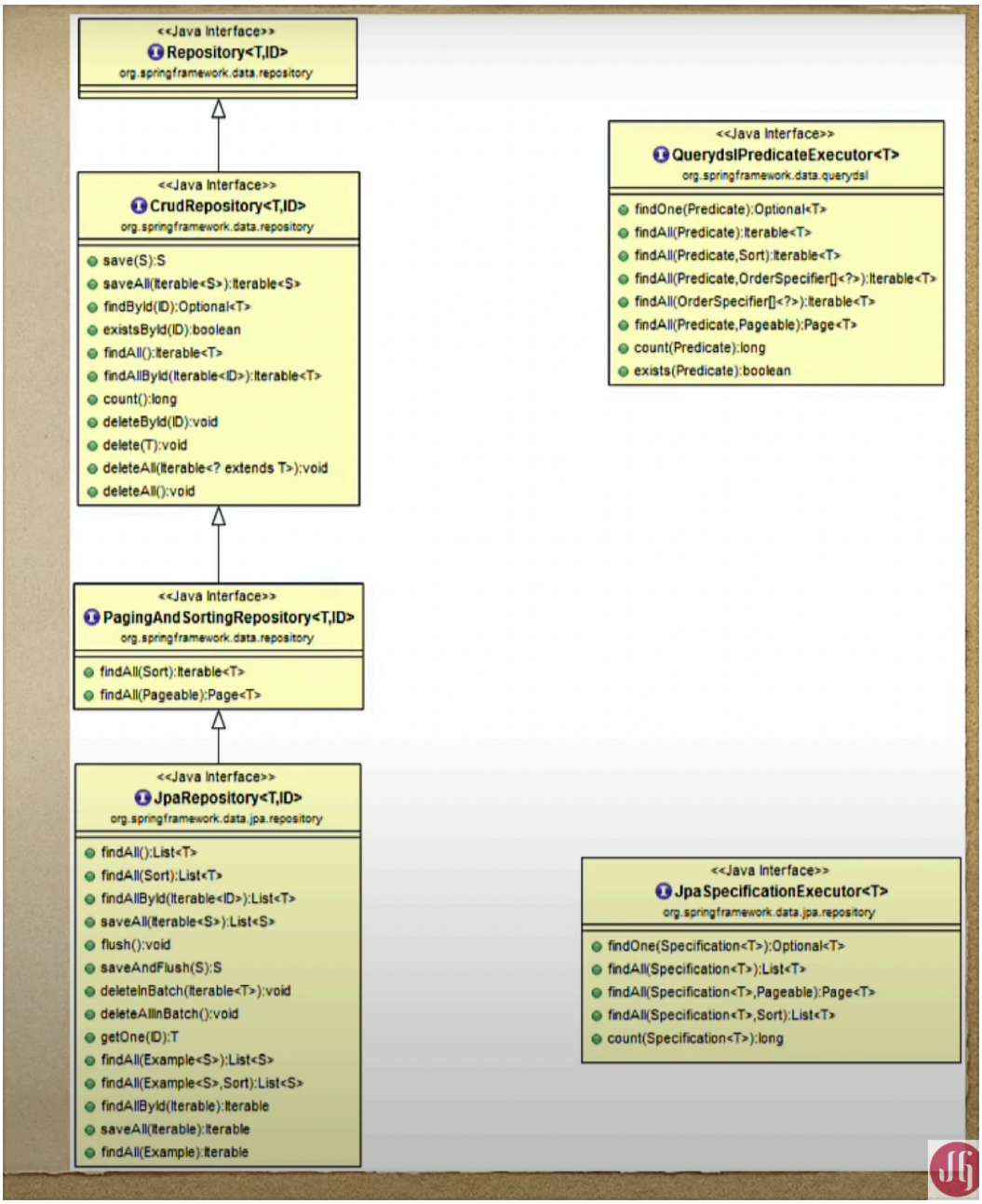
A specification describing how to manage relational data. Hibernate is an implementation of the JPA specification.

Features:

- Data conversion
- Querying
- Validation
- Scheme Generation

These are possible through Hibernate.

@JoinColumn(name = “”, referencedColumnName=“”) it is used in referencing table.
referencedColumnName replaces the use of mappedBy in referenced table.



In Hibernate and Spring Data JPA, if you define an entity without specifying a table name using annotations like `@Entity`, Hibernate will use the default behavior to generate a table name based on the entity class name.

However, if you later specify a table name using the `@Table` annotation, Hibernate will use the specified table name instead of the default generated table name. If the specified table name matches the default generated table name, it will effectively be the same table. Otherwise, Hibernate will attempt to create a new table with the specified name.

Here's an example to illustrate:

```java
@Entity
public class MyEntity {
```

```
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    // other entity attributes and methods
}
```

In this example, Hibernate will generate a table named `MyEntity` because the `@Entity` annotation doesn't specify a table name.

If you later modify the entity to specify a different table name using the `@Table` annotation:

```java
@Entity
@Table(name = "custom_table_name")
public class MyEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    // other entity attributes and methods
}
```

Hibernate will use the table name `custom_table_name` instead of the default generated table name `MyEntity`. If `custom_table_name` already exists in the database, Hibernate will use it. If not, Hibernate will attempt to create a new table with the specified name.

In summary, Hibernate will not create a duplicate table if you specify a table name using `@Table` after the entity has already been defined. Instead, it will use the specified table name if it exists or attempt to create a new table with that name.

Flushing the session:

To achieve this in Spring Data JPA with JpaRepository, you can follow these steps:

1. **Start Transaction**: Ensure that your method is annotated with `@Transactional` to start a transaction.

2. **Fetch Parent Entity**: Use the `JpaRepository` method to fetch the parent

entity from the database.

3. **Modify the Entity**: Make changes to the fetched parent entity within the transactional method.

4. **Flush the Session**: Explicitly flush the Hibernate session to synchronize the changes to the database.

5. **Verify Changes**: Check whether the changes made to the parent entity are propagated to the database.

Here's a sample code snippet illustrating these steps:

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

@Service
public class ParentService {

    @Autowired
    private ParentRepository parentRepository;

    @Transactional
    public void modifyParentEntity() {
        // Step 2: Fetch Parent Entity
        Parent parent = parentRepository.findById(1L).orElse(null);
        if (parent == null) {
            // Handle case when parent is not found
            return;
        }

        // Step 3: Modify the Entity
        parent.setName("Updated Name");

        // Step 4: Flush the Session
        parentRepository.flush(); // This will synchronize the changes to the database

        // Step 5: Verify Changes
        // You can perform additional operations to verify changes, such as fetching the entity again
        Parent updatedParent = parentRepository.findById(1L).orElse(null);
        if (updatedParent != null) {
            System.out.println("Parent Name after modification: " + updatedParent.getName());
```

```
    }
  }
}
```

In this example:

- `ParentService` class contains a method `modifyParentEntity()` annotated with `@Transactional`.
- It fetches a parent entity by its ID using the `findById()` method provided by the `ParentRepository`.
- The fetched parent entity is modified by changing its name.
- The `flush()` method is called on the repository to synchronize the changes with the database.
- Finally, the changes are verified by fetching the entity again and printing its name.

Ensure that the `Parent` entity class is correctly mapped and configured with Spring Data JPA. Additionally, the `ParentRepository` should extend the `JpaRepository<Parent, Long>` interface.

Unit of Work

Correct. In the context of software architecture and database transactions, the Unit of Work pattern is a design pattern that helps manage database transactions and ensure data consistency. Here's how it works:

1. **Tracking Changes**: The Unit of Work keeps track of all changes made to the entities (objects representing database records) within a particular business transaction. These changes can include inserts, updates, and deletes.

2. **Deferred Execution**: Instead of immediately applying these changes to the database, the Unit of Work defers their execution until the end of the transaction. This allows multiple operations to be batched together and executed atomically, ensuring data integrity and consistency.

3. **Committing Changes**: At the end of the transaction, the Unit of Work examines the tracked changes and generates the necessary SQL statements (such as INSERT, UPDATE, DELETE) to apply these changes to the database.

4. **Database Interaction**: Finally, the Unit of Work interacts with the database to execute the generated SQL statements, making the changes permanent. If any errors occur during this process, the Unit of Work can handle them and ensure that the transaction is rolled back to maintain data integrity.

By encapsulating the logic for tracking changes and coordinating database

interactions, the Unit of Work pattern helps simplify transaction management and promotes a cleaner separation of concerns in the application architecture. It is commonly used in conjunction with other patterns such as the Repository pattern to implement data access layers in object-oriented applications.

Dynamic and Named queries in JPA

In JPA (Java Persistence API), you can write named and dynamic queries using JPQL (Java Persistence Query Language) or Criteria API. Here's how you can write named and dynamic queries using both approaches:

Named Queries:

1. **JPQL Named Queries**:
   - Define named queries in your entity classes or in an XML file (`orm.xml`).
   - Annotate the entity class or package with `@NamedQuery` or `@NamedQueries`.
   - Use the named query in your code by referencing its name.

Example:

```java
@Entity
@NamedQuery(name = "Employee.findByLastName", query = "SELECT e FROM Employee e WHERE e.lastName = :lastName")
public class Employee {
    // Entity mapping and other attributes
}
```

Usage:

```java
TypedQuery<Employee> query = entityManager.createNamedQuery("Employee.findByLastName", Employee.class);
query.setParameter("lastName", "Smith");
List<Employee> employees = query.getResultList();
```

2. **Dynamic Named Queries**:
   - You can define dynamic named queries using the Criteria API.
   - Construct the query criteria dynamically based on runtime conditions.

Example:

```java
```

```
String lastName = "Smith";
CriteriaBuilder cb = entityManager.getCriteriaBuilder();
CriteriaQuery<Employee> query = cb.createQuery(Employee.class);
Root<Employee> root = query.from(Employee.class);
query.select(root).where(cb.equal(root.get("lastName"), lastName));
List<Employee> employees =
entityManager.createQuery(query).getResultList();
```

Dynamic Queries:

1. **JPQL Dynamic Queries**:
   - Construct JPQL queries as strings.
   - Use parameters to make the queries dynamic.

Example:

```java
String lastName = "Smith";
String jpql = "SELECT e FROM Employee e WHERE e.lastName = :lastName";
TypedQuery<Employee> query = entityManager.createQuery(jpql,
Employee.class);
query.setParameter("lastName", lastName);
List<Employee> employees = query.getResultList();
```

2. **Criteria API Dynamic Queries**:
   - Construct queries programmatically using the Criteria API.
   - Build query predicates dynamically based on runtime conditions.

Example:

```java
String lastName = "Smith";
CriteriaBuilder cb = entityManager.getCriteriaBuilder();
CriteriaQuery<Employee> query = cb.createQuery(Employee.class);
Root<Employee> root = query.from(Employee.class);
query.select(root).where(cb.equal(root.get("lastName"), lastName));
List<Employee> employees =
entityManager.createQuery(query).getResultList();
```

Both named and dynamic queries have their pros and cons. Named queries offer better readability, centralized management, and potential performance benefits due to caching. On the other hand, dynamic queries provide more flexibility and are suitable for scenarios where the query logic needs to be constructed dynamically at runtime. Choose the approach that best suits your

application requirements.

Example of fetching only 2 results:

In the provided code snippet, `findFirst2ByGenre(String genre)` is a method signature defined in a Spring Data JPA repository interface (`AuthorRepository`). This method retrieves the first two entities (`AuthorNameAge`) matching the specified criteria, which is querying by the `genre` property of the `Author` entity.

Here's the breakdown of what each part does:

- **`findFirst2`**: This part specifies that you want to retrieve the first two records that match the query criteria. It's a derived query method naming convention provided by Spring Data JPA, which allows you to specify the desired behavior in the method name.

- **`ByGenre`**: This part specifies the criteria for the query. In this case, it indicates that you want to retrieve entities based on the `genre` property of the `Author` entity.

- **`String genre`**: This is the parameter passed to the method, representing the value to be used in the query for filtering based on the `genre` property.

So, when you invoke `findFirst2ByGenre("someGenre")`, it will return the first two `AuthorNameAge` entities that match the specified `genre`. The `@Transactional(readOnly = true)` annotation ensures that this query is executed in a read-only transaction context, which is a good practice for read-only operations to improve performance and prevent accidental modifications to the database.

@JavaType and @JdbcType

The annotations `@JavaType` and `@JdbcType` are not standard annotations in Java or JDBC. However, I can provide insights based on common naming conventions and practices in Java and JDBC programming:

1. **@JavaType**:
   - This annotation might be used to specify the Java data type of a field or parameter in a Java class.
   - It could indicate how a database data type maps to a specific Java data type when interacting with the database.
   - For example, it might specify that a `VARCHAR` column in the database corresponds to a Java `String` field.

2. **@JdbcType**:
   - This annotation might be used to specify the JDBC data type of a field or

parameter in a Java class.
  - It could indicate how a Java data type maps to a specific JDBC data type when interacting with the database using JDBC.
  - For example, it might specify that a Java `String` field should be handled as a `VARCHAR` parameter in a JDBC prepared statement.

While the exact meanings and usages of `@JavaType` and `@JdbcType` annotations may vary depending on the specific library or framework you're using, they likely serve to provide metadata or mapping information for interactions between Java code and a database. Always refer to the documentation or source code of the library or framework you're working with for accurate information on these annotations.

Since `@JavaType` and `@JdbcType` annotations are not standard annotations in Java or JDBC, I can provide a hypothetical example demonstrating how similar annotations might be used to provide metadata or mapping information between Java types and database types:

```java
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface JavaType {
    Class<?> value();
}

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface JdbcType {
    int value();
}

public class Employee {
    @JavaType(String.class)
    @JdbcType(java.sql.Types.VARCHAR)
    private String name;

    // Other fields and methods
}
```

In this example:

- `@JavaType` annotation is used to specify the Java data type of a field (`String` in this case).

- `@JdbcType` annotation is used to specify the JDBC data type of a field (`VARCHAR` in this case, represented by `java.sql.Types.VARCHAR`).

These annotations provide metadata about how the Java data type (`String`) of the `name` field in the `Employee` class corresponds to the JDBC data type (`VARCHAR`) when interacting with a database using JDBC. This metadata can be used by frameworks or libraries to automatically handle mappings between Java objects and database columns.

It's important to note that this is a simplified example for illustrative purposes, and actual usage and semantics of annotations may vary depending on the framework or library being used. Always refer to the documentation or source code of the framework or library you're working with for accurate information on annotations and their usage.

@NaturalId

https://thorben-janssen.com/naturalid-good-way-persist-natural-ids-hibernate/

In Hibernate, the `@NaturalId` annotation is used to mark a property of an entity as a "natural identifier." A natural identifier is a unique attribute or combination of attributes that uniquely identify an entity in the domain model, similar to a primary key, but derived from the business domain rather than generated by the database.

Here's the primary use of `@NaturalId`:

1. **Declaring Natural Identifiers**:
   - By annotating a property with `@NaturalId`, you specify that this property serves as a natural identifier for the entity. It indicates to Hibernate that this property should be treated as unique and indexed, similar to a primary key, for faster lookups.

Here's an example:

```java
@Entity
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NaturalId
    @Column(nullable = false, unique = true)
```

```
    private String sku;

    // Other properties and methods
}
```

In this example, the `sku` property is marked as a natural identifier for the `Product` entity using `@NaturalId`. This indicates that the `sku` should be treated as a unique identifier for each product, and Hibernate will ensure that it remains unique and indexed for efficient querying.

Additional points to note about `@NaturalId`:

- Unlike the primary key (`@Id`), natural identifiers are not generated by the database but are typically supplied by the application or derived from the business domain.
- Natural identifiers can provide meaningful and stable identifiers for entities, especially when dealing with legacy databases or integration with external systems.
- Hibernate automatically manages the uniqueness and indexing of natural identifiers, ensuring efficient lookups without the need for additional configuration.

Using `@NaturalId` can be beneficial when working with domain models where certain attributes naturally serve as unique identifiers for entities, providing flexibility and performance benefits in certain scenarios.

Flush() vs persists()

https://medium.com/@niketl16/spring-data-jpa-persisting-entities-vs-flushing-changes-7ac13cea2f17#:~:text=setAge(30)%3B-,userRepository.,for%20the%20transaction%20to%20complete

persist() and merge()

https://vladmihalcea.com/jpa-persist-and-merge/#:~:text=While%20a%20save%20method%20might,you%20need%20to%20call%20merge%20

Hibernate State Transitions

https://vladmihalcea.com/a-beginners-guide-to-jpa-hibernate-entity-state-transitions/

Flush Strategies

https://vladmihalcea.com/a-beginners-guide-to-jpahibernate-flush-strategies/

Entity LifeCycle

https://thorben-janssen.com/entity-lifecycle-model/

Difference between different methods in JPA

https://thorben-janssen.com/persist-save-merge-saveorupdate-whats-difference-one-use

https://stackoverflow.com/questions/5862680/whats-the-difference-between-session-persist-and-session-save-in-hibernate

The behavior of `persist` and `save` methods in Hibernate when an entity with the primary key already exists depends on whether the entity is considered transient or detached.

1. **`persist` Method**:
   - The `persist` method is used to make a transient entity (i.e., an entity not yet associated with a persistence context) managed.
   - If you invoke `persist` on a transient entity and an entity with the same identifier already exists in the persistence context, Hibernate will throw a `javax.persistence.EntityExistsException`.
   - This exception indicates that an entity with the same identifier is already managed in the persistence context, and Hibernate cannot persist another entity with the same identifier.

2. **`save` Method**:
   - The `save` method is typically used to either persist a new entity or update an existing entity.
   - When you use `save` to persist a new entity, Hibernate will generate the primary key value and insert the entity into the database.
   - If you use `save` to persist an entity with a specified primary key, Hibernate will attempt to insert the entity into the database with the provided primary key value.
   - If an entity with the same primary key already exists in the database, Hibernate will throw a `org.hibernate.HibernateException`, indicating a violation of primary key constraints.
   - However, if you use `saveOrUpdate` method, it will either insert a new entity or update an existing entity depending on whether the entity with the same primary key already exists in the persistence context.

In summary, `persist` method expects the entity to be transient and throws an

exception if an entity with the same identifier already exists in the persistence context. On the other hand, `save` method will attempt to insert the entity into the database even if an entity with the same primary key already exists, leading to a constraint violation exception.

Working with own Repository

https://thorben-janssen.com/spring-data-jpa-base-repository/

https://thorben-janssen.com/composite-repositories-spring-data-jpa/
https://vladmihalcea.com/custom-spring-data-repository/

Problem with Fragment Repositories:

Sure, let's illustrate this with an example:

Suppose you have a base repository interface called `BaseRepository` and a fragment repository interface called `CustomRepository`:

```java
// Base repository interface
public interface BaseRepository<T, ID> extends JpaRepository<T, ID> {
    // Base repository methods
}

// Fragment repository interface with custom methods
public interface CustomRepository<T, ID> {
    void customMethod();
}
```

Now, the issue is that you cannot directly combine these two interfaces like this:

```java
// This won't work - you cannot directly include fragment repositories within the base repository interface definition
public interface BaseRepository<T, ID> extends JpaRepository<T, ID>, CustomRepository<T, ID> {
    // Base repository methods
}
```

Instead, you must keep them as separate interfaces and extend them in a separate combined repository interface:

```java
```

```
// Base repository interface
public interface BaseRepository<T, ID> extends JpaRepository<T, ID> {
    // Base repository methods
}

// Fragment repository interface with custom methods
public interface CustomRepository<T, ID> {
    void customMethod();
}

// Combined repository interface extending both the base and fragment
repositories
public interface CombinedRepository<T, ID> extends BaseRepository<T, ID>,
CustomRepository<T, ID> {
    // This interface extends both the base and fragment repositories
}
```

In this setup, `CombinedRepository` extends both `BaseRepository` and `CustomRepository`, allowing you to access methods from both interfaces. This separation ensures a cleaner and more modular design, where each repository interface focuses on a specific set of functionality.

Hibernate Performance tuning

https://stackify.com/find-hibernate-performance-issues/

https://thorben-janssen.com/how-to-activate-hibernate-statistics-to-analyze-performance-issues/

Hibernate: Generate_Statistics

In Spring Boot, you can configure Hibernate properties, including `hibernate.generate_statistics`, using the `application.properties` or `application.yml` configuration files.

Here's how you can set up `hibernate.generate_statistics`:

1. **Using application.properties**:

```properties
# Enable Hibernate statistics
spring.jpa.properties.hibernate.generate_statistics=true
```

2. **Using application.yml**:

```yaml
# Enable Hibernate statistics
spring:
  jpa:
    properties:
      hibernate:
        generate_statistics: true
```

With either approach, Spring Boot will configure Hibernate to enable statistics collection during the application startup, and you can access these statistics programmatically using the `SessionFactory` or `EntityManagerFactory`.

Native query using EntityManger

To write a native SQL query using EntityManager in JPA, you can use the `createNativeQuery` method. Here's how you can do it:

```java
import javax.persistence.EntityManager;
import javax.persistence.Query;
import java.util.List;

// Assuming you have an EntityManager instance injected or obtained through EntityManagerFactory
EntityManager entityManager;

// Write your native SQL query as a string
String sqlQuery = "SELECT * FROM your_table_name WHERE your_condition";

// Create a native query using EntityManager
Query query = entityManager.createNativeQuery(sqlQuery);

// If your query returns entities, you can specify the entity class
// query = entityManager.createNativeQuery(sqlQuery, YourEntityClass.class);

// Execute the query and get the result list
List<Object[]> resultList = query.getResultList();

// Process the result list as needed
for (Object[] row : resultList) {
    // Access individual columns in each row
    // Example: Object column1 = row[0];
    //          Object column2 = row[1];
```

```
}
```

In the example above:
- Replace `"SELECT * FROM your_table_name WHERE your_condition"` with your actual SQL query.
- If your query returns entities, you can specify the entity class as the second argument of `createNativeQuery`.
- After executing the query with `getResultList()`, you will get a list of Object arrays where each array represents a row from the query result. You can access individual columns in each row using array indexing. If your query returns entities, each array element will be an instance of your entity class.

Remember to be cautious when using native SQL queries as they bypass JPQL and may be less portable across different database systems. Also, be mindful of SQL injection vulnerabilities when constructing dynamic queries.

Spring Data JPA – Adding a Method in All Repositories

https://www.baeldung.com/spring-data-jpa-method-in-all-repositories

Other way of doing:

Yes, you can provide an implementation for your custom repository interface if needed. Here's how you can do it:

1. Define your custom repository interface with the custom method(s):

```java
public interface CustomRepository<T, ID> {
    void customMethod();
}
```

2. Create an implementation class for your custom repository:

```java
import org.springframework.stereotype.Repository;

@Repository
public class CustomRepositoryImpl<T, ID> implements CustomRepository<T, ID> {

    @Override
    public void customMethod() {
        // Implement your custom logic here
    }
```

```
}
```

Make sure to annotate the implementation class with `@Repository` to ensure it's picked up by Spring component scanning.

3. Extend your repository interfaces with the custom repository:

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

public interface YourRepository extends JpaRepository<YourEntity, Long>, CustomRepository<YourEntity, Long> {
    // Spring Data JPA will provide the default CRUD methods
    // You can add additional custom methods here if needed
}
```

With this setup, Spring Data JPA will automatically provide the default CRUD methods for your repository, and your custom method implementation will be available as well. You can then inject your repository (`YourRepository`) into your services and use both the default and custom methods.

Spring Data JPA Projections

https://www.baeldung.com/spring-data-jpa-projections

https://vladmihalcea.com/spring-jpa-dto-projection/

Yes, you can have extra instance variables in your projection class that are not defined in the entity class. These additional variables can be used to hold calculated values, transient data, or any other information that is not directly mapped from the entity.

Here's an example:

Let's say you have an entity class `Customer` with properties `id`, `firstName`, and `lastName`, and you want to project only `id` and `fullName`, where `fullName` is a combination of `firstName` and `lastName`. Additionally, you want to include an extra variable `age` in your projection class that is not directly mapped from the entity.

First, define your projection class with the desired constructor and parameter names, including the extra variable:

```java
public class CustomerProjection {
```

```
    private Long id;
    private String fullName;
    private int age; // Extra variable

    public CustomerProjection(Long id, String fullName, int age) {
        this.id = id;
        this.fullName = fullName;
        this.age = age;
    }

    // Getters and setters
}
```

Then, in your repository query method, calculate the value for the `age` variable and include it in the constructor call:

```java
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import org.springframework.data.jpa.repository.JpaRepository;
import javax.persistence.*;

public interface CustomerRepository extends JpaRepository<Customer, Long>
{

    @Query("SELECT c.id AS id, CONCAT(c.firstName, ' ', c.lastName) AS
fullName, YEAR(CURRENT_DATE()) - YEAR(c.dateOfBirth) AS age FROM
Customer c WHERE c.id = :customerId")
    CustomerProjection findCustomerProjectionById(@Param("customerId")
Long customerId);
}
```

In this example, `CustomerProjection` is the DTO class with the desired constructor parameter names (`id`, `fullName`, and `age`). The `age` variable is calculated using the `YEAR` function in the query to calculate the age based on the `dateOfBirth` property of the `Customer` entity.

When you call the `findCustomerProjectionById` method in your repository, Spring Data JPA will map the query result to the `CustomerProjection` DTO using the specified constructor and parameter names, including the extra `age` variable.

Projection using Jpa Tuple

https://vladmihalcea.com/why-you-should-use-the-hibernate-resulttransformer-to-customize-result-set-mappings/

https://vladmihalcea.com/hibernate-resulttransformer/
https://prateek-ashtikar512.medium.com/how-to-fetch-dto-via-javax-persistence-tuple-and-jpql-e482262caac

In Spring Boot, you can specify the location of the SQL files in your `application.properties` file using the `spring.datasource.data` property. Here's how you can do it:

```properties
spring.datasource.data=classpath:data.sql
```

In this example, `data.sql` is the name of the SQL file located in the resources directory (`src/main/resources`). By specifying `classpath:` prefix, you're indicating that the file is located on the classpath.

If you have multiple SQL files, you can specify them as a comma-separated list:

```properties
spring.datasource.data=classpath:data1.sql, classpath:data2.sql
```

This configuration tells Spring Boot to execute the SQL statements in the specified files during application startup.

Remember to adjust the filenames and paths according to your actual file structure and requirements.

Projection using transformer:

To create a hierarchical relationship between entities and transform them into a single DTO using a result transformer, you can utilize a custom `ResultTransformer`. This transformer would handle the transformation of the main entity (Post) and its associated entities (Comment) into a single DTO (PostDTO).

Here's a step-by-step guide:

1. Define your entity classes `Post` and `Comment`.

```java
@Entity
@Table(name = "posts")
public class Post {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
```

```
    private String title;

    @OneToMany(mappedBy = "post", cascade = CascadeType.ALL, fetch =
FetchType.LAZY)
    private List<Comment> comments;

    // Getters and setters
}

@Entity
@Table(name = "comments")
public class Comment {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String text;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "post_id")
    private Post post;

    // Getters and setters
}
```

2. Define your DTO class `PostDTO` representing the hierarchical structure.

```java
public class PostDTO {
    private Long id;
    private String title;
    private List<String> comments;

    // Getters and setters
}
```

3. Implement a custom `ResultTransformer` to transform the query result into the desired DTO structure.

```java
import org.hibernate.transform.ResultTransformer;

public class PostDTOTransformer implements ResultTransformer {
    @Override
```

```
    public Object transformTuple(Object[] tuple, String[] aliases) {
        PostDTO postDTO = new PostDTO();
        postDTO.setId((Long) tuple[0]);
        postDTO.setTitle((String) tuple[1]);
        postDTO.setComments((List<String>) tuple[2]);
        return postDTO;
    }

    @Override
    public List transformList(List collection) {
        return collection;
    }
}
```

4. Execute a query using Hibernate's `createQuery` method and set the custom `ResultTransformer`.

```java
import org.hibernate.query.Query;
import javax.persistence.EntityManager;
import java.util.List;

// Assume you have an EntityManager named "entityManager"

String jpqlQuery = "SELECT p.id, p.title, c.text FROM Post p LEFT JOIN p.comments c";
Query query = entityManager.createQuery(jpqlQuery).unwrap(org.hibernate.query.Query.class);

query.setResultTransformer(new PostDTOTransformer());

List<PostDTO> postDTOs = query.getResultList();
```

In this example:

- We define the `Post` and `Comment` entities representing the hierarchical relationship between posts and comments.
- We create a `PostDTO` class representing the hierarchical structure we want for the DTO.
- We implement a custom `ResultTransformer` (`PostDTOTransformer`) to handle the transformation from the query result to the `PostDTO` structure.
- We execute a query that selects post information (`id` and `title`) along with associated comment text (`text`) using a left join.
- We set the custom `ResultTransformer` to transform the query result into a

list of `PostDTO` objects.

This approach allows you to create a single DTO (`PostDTO`) representing a hierarchical relationship between entities (`Post` and `Comment`) using a custom result transformer.

Other way of doing:

Yes, you can achieve the transformation without using a custom `ResultTransformer`, especially if you're using JPQL or Criteria API queries with JPA. You can handle the transformation manually after retrieving the results from the query.

Here's how you can do it:

1. Execute a JPQL query to fetch posts along with their associated comments.

```java
import javax.persistence.EntityManager;
import javax.persistence.Query;
import java.util.List;

// Assume you have an EntityManager named "entityManager"

String jpqlQuery = "SELECT p, c FROM Post p LEFT JOIN p.comments c";
Query query = entityManager.createQuery(jpqlQuery);

List<Object[]> results = query.getResultList();
```

2. Iterate over the results and manually transform them into `PostDTO` objects.

```java
List<PostDTO> postDTOs = new ArrayList<>();

for (Object[] result : results) {
    Post post = (Post) result[0];
    Comment comment = (Comment) result[1];

    PostDTO postDTO = new PostDTO();
    postDTO.setId(post.getId());
    postDTO.setTitle(post.getTitle());
    postDTO.getComments().add(comment.getText());

    postDTOs.add(postDTO);
}
```

In this example:

- We execute a JPQL query that selects `Post` entities along with their associated `Comment` entities using a left join.
- We retrieve the results as a list of Object arrays (`List<Object[]>`), where each array contains a `Post` and its associated `Comment`.
- We iterate over the results and manually extract the `Post` and `Comment` objects.
- We create a new `PostDTO` object and populate it with data from the `Post` and `Comment` objects.
- We add the `PostDTO` object to a list of `PostDTOs`.

This approach provides manual control over the transformation process and doesn't require a custom `ResultTransformer`. It's suitable for simpler cases where the transformation logic is straightforward. However, for more complex transformations or when dealing with a large number of entities, a custom `ResultTransformer` approach might be more appropriate.

https://vladmihalcea.com/custom-spring-data-repository/

CASCADE:

In JPA, the behavior depends on how you have configured the relationship between the `Post` entity and the `Comment` entity, specifically regarding cascade operations.

If you have configured the cascade type appropriately, saving a `Post` entity that has associated `Comment` entities will automatically persist the comments to the database without the need to explicitly save them.

Here's how you can configure cascading behavior in JPA:

```java
@Entity
@Table(name = "posts")
public class Post {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String title;

    @OneToMany(mappedBy = "post", cascade = CascadeType.ALL,
orphanRemoval = true)
    private List<Comment> comments;
```

```
    // Getters and setters
}

@Entity
@Table(name = "comments")
public class Comment {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String text;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "post_id")
    private Post post;

    // Getters and setters
}
```

In this example:

- In the `Post` entity, the `comments` field is annotated with `@OneToMany` with `cascade = CascadeType.ALL`. This means that any operation (including `persist`, `merge`, `remove`, etc.) performed on the `Post` entity will be cascaded to its associated `Comment` entities.
- Additionally, `orphanRemoval = true` ensures that if a comment is removed from the `comments` collection of a `Post`, it will be deleted from the database as well.

With this configuration, when you save a `Post` entity that has associated `Comment` entities, the comments will automatically be persisted to the database along with the post. You don't need to explicitly save the comments separately. However, if the cascade type is not properly configured or is not specified, you would need to save the comments explicitly before saving the post to ensure they are persisted to the database.

If cascade options are not specified on the relationship between entities, JPA will not automatically persist, merge, or remove associated entities when you perform operations on the owning entity. Let's consider a scenario without cascade options:

```java
@Entity
@Table(name = "posts")
public class Post {
```

```
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String title;

    @OneToMany(mappedBy = "post")
    private List<Comment> comments;

    // Getters and setters
}

@Entity
@Table(name = "comments")
public class Comment {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String text;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "post_id")
    private Post post;

    // Getters and setters
}
```

In this example:

- In the `Post` entity, the `comments` field is annotated with `@OneToMany` without specifying any cascade options.
- In the `Comment` entity, the `post` field is annotated with `@ManyToOne`.

Without cascade options specified, if you create or load a `Post` entity and add `Comment` entities to its `comments` collection, those comments will not be persisted to the database when you persist the `Post` entity.

You would need to save the `Post` entity first and then separately save each `Comment` entity if you want them to be persisted:

```java
Post post = new Post();
post.setTitle("Example Post");

Comment comment1 = new Comment();
```

```
comment1.setText("First comment");
comment1.setPost(post); // Set the association

Comment comment2 = new Comment();
comment2.setText("Second comment");
comment2.setPost(post); // Set the association

post.getComments().add(comment1);
post.getComments().add(comment2);

entityManager.persist(post); // Save the Post entity
entityManager.persist(comment1); // Save the first Comment entity
entityManager.persist(comment2); // Save the second Comment entity
```

Without cascade options, you'll need to manage the persistence of associated entities manually, which can be more cumbersome and error-prone compared to using cascade options where appropriate.

CASCADE type REMOVE vs Orphan Removal:

While both orphan removal and cascade type `REMOVE` can result in the deletion of related entities, they serve different purposes and have different behaviors:

1. **Orphan Removal**:
   - Orphan removal is a feature that automatically removes child entities (orphans) when they are no longer referenced by their parent entity.
   - It's typically used with one-to-many or many-to-one associations to ensure that child entities are deleted from the database when they are removed from the collection of their parent entity.
   - Orphan removal is specified using the `orphanRemoval` attribute in the `@OneToMany` or `@OneToOne` annotation.
   - Orphan removal only applies to entities that are removed from the collection of their parent entity; it does not cascade removal operations to other entities.

2. **Cascade Type `REMOVE`**:
   - Cascade type `REMOVE` is one of the cascade options that specifies that operations on the parent entity should be cascaded to related entities, resulting in the deletion of related entities when the parent entity is deleted.
   - It's typically used to ensure that related entities are deleted when their parent entity is deleted.
   - Cascade type `REMOVE` is specified using the `cascade` attribute in the `@OneToMany`, `@ManyToOne`, `@OneToOne`, or `@ManyToMany` annotation.
   - Cascade type `REMOVE` applies to the removal of the entire entity, including its relationships and related entities.

In summary, orphan removal and cascade type `REMOVE` have different scopes and purposes:

- Orphan removal is concerned with removing child entities that are no longer referenced by their parent entity's collection.
- Cascade type `REMOVE` is concerned with cascading removal operations to related entities when the parent entity is deleted.

@EmbeddedId with @AttributeOverride

@Id cannot be used if @EmbeddedId is used. It defines a composite key containing attributes from @Embeddable class.

`@AttributeOverride` is an annotation in JPA (Java Persistence API) used to override the mapping of a single property of an embedded attribute within an entity.

When you have an embeddable class used as an embedded ID (identifier) in an entity, you may need to customize the mapping of individual attributes within that embeddable class. This is where `@AttributeOverride` comes into play.

Here's how `@AttributeOverride` works:

1. **Usage**:
   - `@AttributeOverride` is applied to an entity class to override the mapping of an attribute defined within an embeddable class used as an embedded ID.

2. **Target**:
   - `@AttributeOverride` targets an embedded attribute within an entity.

3. **Attributes**:
   - `name`: Specifies the name of the embedded attribute to be overridden.
   - `column`: Specifies the column settings to be applied for the overridden attribute.

4. **Example**:
   ```java
   @Entity
   public class Employee {
       @EmbeddedId
       @AttributeOverride(name = "startDate", column = @Column(name = "start_date_override"))
       private EmployeeId id;

       // Other properties and methods
   }
   ```

```
@Embeddable
public class EmployeeId {
    private Long employeeId;
    private LocalDate startDate;

    // Getters and setters
}
```

In this example:
- `Employee` entity uses an embeddable class `EmployeeId` as its embedded ID.
- `EmployeeId` class contains two attributes: `employeeId` and `startDate`.
- `@AttributeOverride` is used to override the mapping of the `startDate` attribute within the `EmployeeId` embeddable class. It changes the column name to `start_date_override`.

By using `@AttributeOverride`, you can customize the mapping of individual attributes within an embeddable class, providing flexibility in how embedded IDs are mapped to the database schema.

In the example provided earlier, let's say we have an entity class `Employee` with an embedded ID `EmployeeId`, and we use `@AttributeOverride` to override the mapping of the `startDate` attribute within `EmployeeId`. Here's how the database table might look like after the schema generation or update:

```java
@Entity
public class Employee {
    @EmbeddedId
    @AttributeOverride(name = "startDate", column = @Column(name = "start_date_override"))
    private EmployeeId id;

    // Other properties and methods
}

@Embeddable
public class EmployeeId {
    private Long employeeId;
    private LocalDate startDate;

    // Getters and setters
}
```

After the schema generation or update, the database table for the `Employee` entity might look like this:

```
Employee Table:
| employeeId | start_date_override |
|------------|---------------------|
|            |                     |
```

Explanation:
- The `Employee` table contains two columns: `employeeId` and `start_date_override`.
- The `employeeId` column is inherited from the `EmployeeId` embeddable class, and it represents the primary key of the `Employee` entity.
- The `start_date_override` column is the overridden mapping of the `startDate` attribute within the `EmployeeId` embeddable class. It reflects the changes specified by `@AttributeOverride`, where the original column name `startDate` is replaced with `start_date_override`.

E.g.

Certainly! Here's an example demonstrating the use of `@EmbeddedId` along with `@MapsId`:

Let's assume we have two entities, `Employee` and `EmployeeDetails`, where `EmployeeDetails` holds additional details about an employee. Both entities are related, and we want to use the same primary key for both entities.

```java
import javax.persistence.*;

@Entity
public class Employee {

    @EmbeddedId
    private EmployeeId id;

    @OneToOne(mappedBy = "employee")
    private EmployeeDetails details;

    // Other properties and methods
}
```

```java
```

```
import javax.persistence.*;

@Embeddable
public class EmployeeId implements Serializable {

    private Long employeeId;

    // Getters and setters
}
```

```java
import javax.persistence.*;

@Entity
public class EmployeeDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long employeeId; // Maps to the primary key of Employee

    @MapsId
    @OneToOne
    @JoinColumn(name = "employee_id")
    private Employee employee;

    // Other properties and methods
}
```

In this example:

- `Employee` entity has an embedded ID `EmployeeId`, which contains the `employeeId`.
- `EmployeeDetails` entity has a primary key `employeeId` which is also used as a foreign key mapped by `@MapsId`.
- `@MapsId` ensures that `employeeId` in `EmployeeDetails` maps to the same column as the primary key of `Employee`.
- There's a one-to-one relationship between `Employee` and `EmployeeDetails`, where `Employee` is the owning side of the relationship.

With this setup, when you persist an `EmployeeDetails` instance, its `employeeId` will be automatically populated with the same value as the corresponding `Employee`'s `employeeId`, maintaining the relationship between the two entities.

Certainly! The statement " @MapsId is used in conjunction with @EmbeddedId

to specify derived primary key" can be justified as follows:

1. **Specifying a Derived Primary Key**:
   - When you have an entity with a composite primary key (i.e., primary key consisting of multiple attributes), you typically use `@EmbeddedId` to define an embeddable class that represents the composite key.
   - In some scenarios, you might want to use the same primary key value in another related entity. This is where `@MapsId` comes into play.
   - `@MapsId` is used to specify that an attribute in an entity is mapped to the same column(s) as the primary key of its parent entity. This allows you to share the primary key value between entities.

2. **Use with @EmbeddedId**:
   - When you use `@EmbeddedId` to define a composite primary key, you embed the primary key attributes within the entity.
   - You can then use `@MapsId` in conjunction with `@EmbeddedId` to indicate that an attribute within the entity should be mapped to the same column(s) as the primary key attributes defined in the embeddable class.
   - This effectively allows you to derive the primary key value of one entity from the primary key value of another related entity.

3. **Example**:
   - In the example provided earlier, `EmployeeDetails` has a primary key `employeeId`, and it's mapped to the same column(s) as the primary key of `Employee` using `@MapsId`.
   - The `EmployeeId` embeddable class is used as the composite primary key in `Employee`, and `@MapsId` ensures that the `employeeId` attribute in `EmployeeDetails` is derived from the primary key of `Employee`.

In summary, `@MapsId` in conjunction with `@EmbeddedId` allows you to specify a derived primary key, where an attribute in an entity is mapped to the same column(s) as the primary key of another related entity, typically defined using an embeddable class. This helps maintain consistency and relationships between entities with shared primary keys.

Important about @EmbeddedId

The class whose attributes we are using composite key should implement Serializable.

@Modifying

It supports update sql inside @Query jpql. It also needs the operation should be @Transactional.

@Modifying

```
@Transactional
@Query(“update Author a set a.age=:age where a.id=:id”)
Int updateAuthor(@Param(“age”) int age,@Param(“id”) int id)
```