Difference between @Embeddable and @OneToOne

https://stackoverflow.com/questions/13723914/hibernate-difference-between-embedded-annotation-technique-and-onetoone-ann

@Embeddable is used over the class to be embedded and @Embedded over the field in the class in which it is embedded.

https://www.geeksforgeeks.org/hibernate-embeddable-and-embedded-annotation/

So the fields will occur in same table for @Embeddable.

But they will be prefixed with name of field (where the embedded object is declared) with underscore (e.g. address_name)

But if we want that there is existing table and I want to map the column names then we can do with @AttributeOverrides which takes list of @AttributeOverride(name = “”, column = @Column(name = “”))

Here name is name of field in the class which we are embedding and column = helps in mapping the column existing in table in database.

@AttributeOverride and @AssociationOverride

https://docs.jboss.org/hibernate/orm/6.2/userguide/html_single/Hibernate_User_Guide.html#embeddables

In Jakarta Persistence (formerly known as JPA), when you use an embeddable type multiple times within an entity, you might need to customize the column names for each usage of the embeddable. This is where the `@AttributeOverride` and `@AssociationOverride` annotations come into play.

Here's what they do:

1. **@AttributeOverride**:
   - This annotation allows you to override the default column names defined by the embeddable type for specific attributes within the entity.
   - It's useful when you want to use the same embeddable type multiple times in an entity, but you need different column names for certain attributes each time.

2. **@AssociationOverride**:
   - While `@AttributeOverride` is used to customize column names for basic attributes within the embeddable, `@AssociationOverride` is used specifically for association attributes (such as `@OneToOne`, `@ManyToOne`,

`@OneToMany`, or `@ManyToMany`).
  - It allows you to customize the column names for association attributes when using the embeddable multiple times within an entity.

Here's a basic example to illustrate how they're used:

```java
@Entity
public class Employee {

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

    @Embedded
    @AttributeOverrides({
        @AttributeOverride(name = "street", column = @Column(name = "home_street")),
        @AttributeOverride(name = "city", column = @Column(name = "home_city"))
    })
    private Address homeAddress;

    @Embedded
    @AttributeOverrides({
        @AttributeOverride(name = "street", column = @Column(name = "work_street")),
        @AttributeOverride(name = "city", column = @Column(name = "work_city"))
    })
    private Address workAddress;

    // Other properties and methods
}

@Embeddable
public class Address {

    private String street;

    private String city;

    // Getters and setters
}
```

In this example:

- The `Employee` entity has two embeddable `Address` objects: `homeAddress` and `workAddress`.
- The `@AttributeOverrides` annotation is used to customize the column names for the `street` and `city` attributes of the `Address` embeddable within the `Employee` entity.
- Each usage of the `Address` embeddable has its own set of overridden column names.

Overall, `@AttributeOverride` and `@AssociationOverride` annotations provide flexibility when using embeddable types multiple times within an entity, allowing you to customize column names to suit your specific requirements.

It seems there is no annotation called `@AssociationAttribute` in Jakarta Persistence (JPA) or Hibernate. However, there are similar annotations like `@AssociationOverride` and `@AttributeOverride` which are commonly used for customizing associations and attributes within embeddable objects.

Let me provide you with an example demonstrating the usage of `@AssociationOverride`:

Suppose we have an `Employee` entity that contains multiple addresses: home and work addresses. Each address is represented by an embeddable `Address` object. We want to customize the column names for the association attributes when using the `Address` embeddable within the `Employee` entity.

```java
@Entity
public class Employee {

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

    @Embedded
    @AssociationOverrides({
        @AssociationOverride(name = "homeAddress",
            joinColumns = @JoinColumn(name = "home_address_id")),
        @AssociationOverride(name = "workAddress",
            joinColumns = @JoinColumn(name = "work_address_id"))
    })
    private AddressSet addresses;

    // Other properties and methods
}
```

```
@Embeddable
public class AddressSet {

    @ManyToOne
    private Address homeAddress;

    @ManyToOne
    private Address workAddress;

    // Getters and setters
}

@Entity
public class Address {

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

    private String street;

    private String city;

    // Getters and setters
}
```

In this example:

- The `Employee` entity has an embeddable `AddressSet` object that contains two association attributes: `homeAddress` and `workAddress`.
- The `@AssociationOverrides` annotation is used to customize the join column names for the association attributes within the `AddressSet` embeddable.
- Each usage of the `AddressSet` embeddable has its own set of overridden join column names.

While there isn't an annotation called `@AssociationAttribute` in JPA or Hibernate, `@AssociationOverride` and `@AttributeOverride` are commonly used for similar purposes to customize associations and attributes within embeddable objects.

Entity 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/

Persistence Context

https://www.baeldung.com/jpa-hibernate-persistence-context

https://medium.com/emlakjet/hibernate-persistence-context-c4d3bc1e84d3#:~:text=Unit%20of%20work%20is%20a,have%20to%20close%20a%20session

When Hibernate retrieves data from the database, it typically stores this data in an area of memory called the "persistence context." This persistence context acts as a first-level cache, holding onto entity instances that have been retrieved or persisted during the current transaction.

Now, let's consider the scenario described in your statement:

1. **Read-Committed Transaction Isolation**:
   - Read-committed isolation is a database transaction isolation level that ensures that any data read by a transaction is committed data. This means that if another transaction updates data that the current transaction is reading, the changes made by the other transaction won't be visible to the current transaction until it commits.

2. **Persistence Context in Hibernate**:
   - Hibernate utilizes the persistence context to manage entity instances retrieved from the database.
   - When Hibernate retrieves an entity from the database, it stores a reference to this entity in the persistence context.

3. **Behavior with Existing Entity Instances**:
   - If Hibernate fetches an entity from the database and finds that an instance of the same entity already exists in the persistence context (e.g., it was previously retrieved or persisted in the same transaction), it doesn't replace the existing instance with the fetched instance from the result set.
   - Instead, it reuses the existing instance already present in the persistence context. This is to ensure that any modifications made to the existing instance are retained and not lost due to replacement with potentially newer data from the database.

4. **Ignoring Potentially New Data**:
   - Due to read-committed transaction isolation at the database level, if another transaction updates data that the current transaction is reading, the changes made by the other transaction won't be visible to the current

transaction until it commits.
  - Therefore, even if there are potentially newer data in the result set fetched by Hibernate, it doesn't replace existing entity instances in the persistence context with this potentially newer data because it may not be visible or consistent yet according to the read-committed isolation level.

In summary, Hibernate's behavior ensures data consistency and transaction isolation by retaining existing entity instances in the persistence context and ignoring potentially newer data from the result set if an instance of the same entity is already present in the persistence context. This behavior aligns with the principles of transaction isolation and data consistency in a read-committed transaction isolation level.

In Java Persistence API (JPA), the `EntityManager` and the persistence context are closely related concepts that work together to manage entity instances and their lifecycle. Here's the relationship between them:

1. **EntityManager**:
  - The `EntityManager` interface is the primary interface through which you interact with the persistence context in JPA.
  - It serves as a gateway to perform CRUD (Create, Read, Update, Delete) operations on entities, execute JPQL (Java Persistence Query Language) queries, and manage transactions.

2. **Persistence Context**:
  - The persistence context is a collection of managed entity instances that are associated with the current unit of work or transaction.
  - It acts as a first-level cache, storing entity instances retrieved from the database or persisted during the current transaction.
  - The persistence context maintains the state of managed entities, tracks changes made to them, and ensures that these changes are synchronized with the database.

3. **EntityManager and Persistence Context Relationship**:
  - Each `EntityManager` instance is associated with a persistence context.
  - When you obtain an `EntityManager` from an EntityManagerFactory, it represents a new persistence context for the current transaction.
  - The `EntityManager` manages the lifecycle of entity instances within its associated persistence context.
  - Operations performed by the `EntityManager`, such as finding, persisting, merging, or removing entities, affect the state of entity instances within the persistence context.

4. **Managed Entity Instances**:
  - When you load an entity from the database using the `EntityManager`, it becomes a managed entity instance within the persistence context.

- Managed entities are tracked by the persistence context, and any changes made to them are automatically synchronized with the database during the transaction commit.

In summary, the `EntityManager` serves as the entry point for interacting with the persistence context in JPA. It manages the lifecycle of entity instances within the context of a transaction, while the persistence context tracks and maintains the state of managed entities. Together, the `EntityManager` and the persistence context provide a powerful mechanism for managing entities and their interactions with the underlying database in JPA applications.

In container-managed persistence contexts, where the `EntityManager` is typically managed by the Java EE container, the unit of work is managed within the boundaries of a transaction. Although you use the same instance of `EntityManager` for each operation, the unit of work is still effectively managed by the container through the lifecycle of a transaction. Here's how it works:

1. **Transaction Boundaries**:
   - The unit of work in container-managed persistence contexts is defined by the boundaries of a transaction.
   - When a transaction begins, the container ensures that the same instance of `EntityManager` is associated with the current transaction context.
   - All entity operations (e.g., persisting, updating, querying) performed within the same transaction share the same `EntityManager` instance.

2. **Persistence Context Lifecycle**:
   - The persistence context associated with the `EntityManager` is typically scoped to the duration of a transaction.
   - When the transaction begins, the container creates a new persistence context and associates it with the `EntityManager`.
   - All entity instances managed by the persistence context are tracked and managed within the scope of the transaction.

3. **Transaction Management**:
   - The container manages the lifecycle of transactions, including transaction demarcation, beginning, committing, and rolling back.
   - Entity operations performed within the transaction are part of the unit of work managed by the container.
   - The `EntityManager` delegates transactional operations to the underlying JPA provider, which coordinates with the container-managed transaction manager.

4. **Automatic Flushing and Synchronization**:
   - Changes made to managed entity instances within the persistence context are automatically synchronized with the database at the end of the transaction.
   - The container ensures that the persistence context is flushed and changes are propagated to the database before committing the transaction.

5. **Thread-Safety and Concurrency**:
   - Container-managed persistence contexts ensure thread-safety and concurrency by providing isolation between transactions.
   - Each transaction operates within its own persistence context and `EntityManager` instance, preventing interference or conflicts between concurrent transactions.

In summary, while you use the same instance of `EntityManager` for each operation in container-managed persistence contexts, the unit of work is effectively managed within the boundaries of a transaction. The container ensures transactional integrity, persistence context lifecycle management, and synchronization with the database, providing a consistent and reliable environment for managing entity operations in Java EE applications.