**Here is a more detailed explanation of the JPA annotations that help in developing a project with one and two tables for CRUD operations:**

**1. Entity Mapping Annotations (Single Table)**

**1.1 @Entity:**

**Purpose:** Marks a class as a persistent entity, meaning it will map to a database table.

**Example:**

@Entity

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

}

In this case, the `Employee` class represents a table in the database. Each instance of this class corresponds to a row in the table.

**1.2 @Table:**

**Purpose:** Specifies the name of the database table that the entity is mapped to.

**Example:**

@Entity

@Table(name = "employee\_details")

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

}

The entity `Employee` is mapped to a table called `employee\_details` instead of the default table name (`Employee`).

**1.3 @Id:**

**Purpose:** Specifies the primary key of an entity.

**Example:**

@Id

private Long id;

This marks the `id` field as the primary key for the `Employee` entity.

**1.4 @GeneratedValue:**

**Purpose:** Specifies how the primary key should be generated.

**Strategies:**

\* **IDENTITY:** The database generates the primary key for the entity.

\* **SEQUENCE:** Uses a database sequence to generate the primary key.

\* **AUTO:** The persistence provider (e.g., Hibernate) will automatically select a generation strategy.

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

**1.5 @Column:**

**Purpose:** Maps a class field to a specific database column. You can use this to specify column names, data types, constraints, etc.

**Example:**

@Column(name = "employee\_name", nullable = false)

private String name;

This maps the `name` field to the column `employee\_name` in the database.

**1.6 @Transient:**

**Purpose:** Marks a field that should not be persisted to the database. It is useful for fields used only in the business logic and not meant to be saved.

**Example:**

@Transient

private int age; // This field won't be stored in the database

#### 1.7 \*\*`@Lob`\*\*:

- \*\*Purpose\*\*: Used for fields that will store large objects like long text or binary data (e.g., images, documents).

- \*\*Example\*\*:

```java

@Lob

private String description; // Can store large text data

```

#### 1.8 \*\*`@Temporal`\*\*:

- \*\*Purpose\*\*: Used to map `java.util.Date` or `java.util.Calendar` fields to database date, time, or timestamp fields.

- \*\*Example\*\*:

```java

@Temporal(TemporalType.DATE)

private Date birthDate; // Maps to a DATE type in the database

```

### 2. \*\*Entity Mapping Annotations (Two Tables with Relationships)\*\*

#### 2.1 \*\*`@OneToOne`\*\*:

- \*\*Purpose\*\*: Defines a one-to-one relationship between two entities. This means that one instance of an entity is related to exactly one instance of another entity.

- \*\*Example\*\*:

```java

@OneToOne

@JoinColumn(name = "address\_id") // This specifies the foreign key column

private Address address;

```

This maps a one-to-one relationship between `Employee` and `Address`. The foreign key `address\_id` is used to link the two tables.

#### 2.2 \*\*`@OneToMany`\*\*:

- \*\*Purpose\*\*: Defines a one-to-many relationship. This means that one instance of an entity is related to many instances of another entity.

- \*\*Example\*\*:

```java

@OneToMany(mappedBy = "employee")

private List<Project> projects;

```

This means that an `Employee` can have many `Project` objects.

#### 2.3 \*\*`@ManyToOne`\*\*:

- \*\*Purpose\*\*: Defines a many-to-one relationship, meaning that many instances of one entity are associated with a single instance of another entity.

- \*\*Example\*\*:

```java

@ManyToOne

@JoinColumn(name = "department\_id") // foreign key column

private Department department;

```

This means that many `Employee` objects are related to one `Department`.

#### 2.4 \*\*`@ManyToMany`\*\*:

- \*\*Purpose\*\*: Defines a many-to-many relationship, meaning that many instances of one entity are related to many instances of another entity. A join table is used to manage the relationship.

- \*\*Example\*\*:

```java

@ManyToMany

@JoinTable(

name = "employee\_project",

joinColumns = @JoinColumn(name = "employee\_id"),

inverseJoinColumns = @JoinColumn(name = "project\_id")

)

private Set<Project> projects;

```

This means that many `Employee` objects can be related to many `Project` objects, with the `employee\_project` table serving as the join table.

#### 2.5 \*\*`@JoinColumn`\*\*:

- \*\*Purpose\*\*: Specifies the foreign key column for the relationship.

- \*\*Example\*\*:

```java

@ManyToOne

@JoinColumn(name = "department\_id")

private Department department;

```

#### 2.6 \*\*`@JoinTable`\*\*:

- \*\*Purpose\*\*: Used in a many-to-many relationship to define the join table and its columns.

- \*\*Example\*\*:

```java

@ManyToMany

@JoinTable(

name = "employee\_project",

joinColumns = @JoinColumn(name = "employee\_id"),

inverseJoinColumns = @JoinColumn(name = "project\_id")

)

private Set<Project> projects;

```

### 3. \*\*Additional Annotations for CRUD Operations\*\*

#### 3.1 \*\*`@Embeddable`\*\*:

- \*\*Purpose\*\*: Marks a class as embeddable, meaning its fields are part of the entity owning it. Often used for composite keys or embedded objects.

- \*\*Example\*\*:

```java

@Embeddable

public class Address {

private String street;

private String city;

}

```

#### 3.2 \*\*`@Embedded`\*\*:

- \*\*Purpose\*\*: Used to embed the fields of an embeddable class into an entity.

- \*\*Example\*\*:

```java

@Entity

public class Employee {

@Embedded

private Address address;

}

```

#### 3.3 \*\*`@EmbeddedId`\*\*:

- \*\*Purpose\*\*: Used to map a composite primary key, where the key consists of multiple fields.

- \*\*Example\*\*:

```java

@Entity

public class Employee {

@EmbeddedId

private EmployeeId employeeId; // composite key

}

```

#### 3.4 \*\*`@AttributeOverrides` & `@AttributeOverride`\*\*:

- \*\*Purpose\*\*: Used to override the mapping of an attribute in an embedded object.

- \*\*Example\*\*:

```java

@Embedded

@AttributeOverrides({

@AttributeOverride(name = "street", column = @Column(name = "home\_street")),

@AttributeOverride(name = "city", column = @Column(name = "home\_city"))

})

private Address homeAddress;

```

### 4. \*\*Annotations for Querying and Transactions\*\*

#### 4.1 \*\*`@Query`\*\*:

- \*\*Purpose\*\*: Used to define custom JPQL or native SQL queries in repository methods.

- \*\*Example\*\*:

```java

@Query("SELECT e FROM Employee e WHERE e.name = :name")

List<Employee> findByName(@Param("name") String name);

```

#### 4.2 \*\*`@Modifying`\*\*:

- \*\*Purpose\*\*: Indicates that a query is modifying the data (e.g., `INSERT`, `UPDATE`, `DELETE`).

- \*\*Example\*\*:

```java

@Modifying

@Query("UPDATE Employee e SET e.salary = :salary WHERE e.id = :id")

void updateSalary(@Param("id") Long id, @Param("salary") Double salary);

```

#### 4.3 \*\*`@Transactional`\*\*:

- \*\*Purpose\*\*: Used to declare a method or class as transactional.

- \*\*Example\*\*:

```java

@Transactional

public void updateEmployee(Employee employee) {

// Business logic

}

```

### Summary

- \*\*Single Table Operations\*\*: Use `@Entity`, `@Table`, `@Id`, `@GeneratedValue`, `@Column`, etc.

- \*\*Two Tables with Relationships\*\*: Use `@OneToOne`, `@OneToMany`, `@ManyToOne`, `@ManyToMany`, `@JoinColumn`, `@JoinTable`.

- \*\*Composite Keys and Embedded Objects\*\*: Use `@

Embeddable`, `@Embedded`, `@EmbeddedId`.

- \*\*Query and Transaction Management\*\*: Use `@Query`, `@Modifying`, `@Transactional`.

These annotations allow you to effectively manage CRUD operations, table relationships, and complex database mappings.