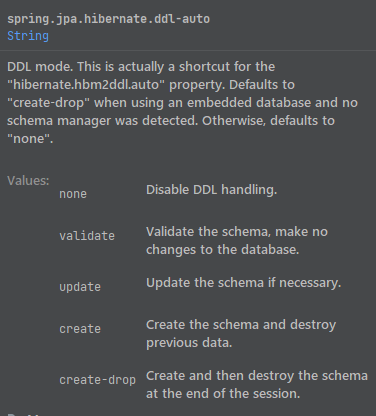
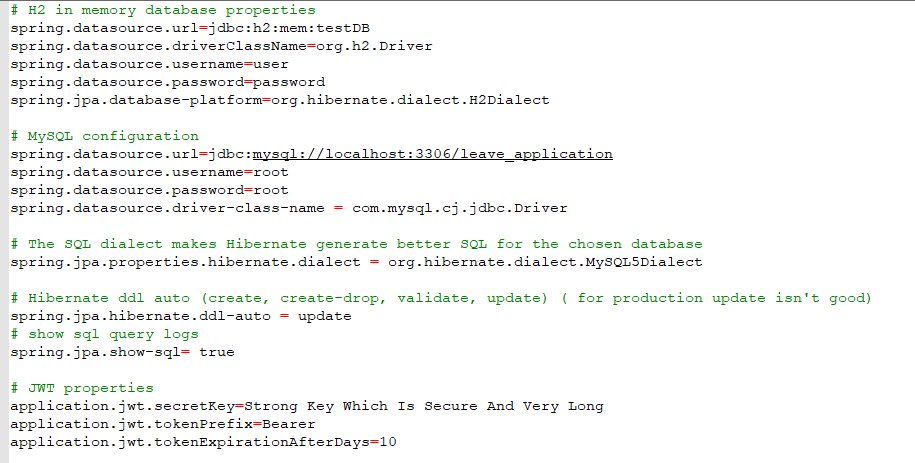
JPA is a standard specification, its implemented by different ORM provider (like hibernate, IBates etc). We need to use any of the ORM provider implementation. As we will be coding with interface it’s easy to switch between different providers.





**@Entity:** This class is reflected in the database.

**@Table:** We can define the table name, unique constraints etc here.

**@UniqueConstraint:** We can define the unique constraint name and column name here.

**@Id:** It defines the primary key of the table.

**@GeneratedValue:** Used to define auto generated value for primary key.

**AUTO:** Default value is AUTO (example: “8dd5f315-9788-4d00-87bb-10eed9eff566”). This generates a UUID or numerical.

**IDENTITY:** Values here gets generated by an identity column in the database. IDENTITY generation disables batch updates.

**TABLE:** This uses an underlying database table that holds segments of identifier generation values. The disadvantage of this method is that it doesn't scale well and can negatively affect performance.

**SEQUENCE:** Here hibernate provides the SequenceStyleGenerator class. If database supports this, this class is used otherwise It switch to table generation.

**@SequenceGenerator:** A sequence generator may be specified on the entity class or on the primary key field or property. The scope of the generator name is global to the persistence unit (across all generator types). We can define the names initial value allocation size (next sequence value \* allocation size = next id) etc. here. This generator can be used later in the @**GeneratedValue.**

**@Column:** Here we can define the column name and some other configuration.

In case we want to use a class inside the entity class we can embed it.

**@Embeddable:** Allows this class to be embedded as an entity property.

**@AttributeOverrides:** Here we can add list of attribute overrides to define their column.

**@Embedded:** When we use the class inside the entity, we use this annotation.

There are different repositories available for us. They implement common methods for us internally. The hierarchy of extension:

*JPARepository*

*extends PagingAndSortingRepository*

*extends CrudRepository*

*extends Repository*

<https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.repositories>

**@Repository:** used to work with database entity.

**@Query:** In case we want to create our own query. Here we use JPQL query. We don’t specify the database table/column name rather the class/property name here. If we want to use native SQL, we do that by setting the *nativeQuery*=*true*. We can use named parameter here as well as the ?1 type as well.

<https://www.baeldung.com/spring-data-jpa-query>

**@Modifying**: In case we want to update database, we must use **@Modifying.**

**@Transactional:** can be used in either class or method level. It has further configuration as well. If all the transactions successful, only then the changes will be committed in the database. Otherwise, it will be rolled back. Typically, is used in service layer.

**Cascading:**

Entity relationships often depend on the existence of another entity, for example the Person–Address relationship. Without the Person, the Address entity doesn't have any meaning of its own. When we delete the Person entity, our Address entity should also get deleted.

Cascading is the way to achieve this. When we perform some action on the target entity, the same action will be applied to the associated entity.

All JPA-specific cascade operations are represented by the javax.persistence.CascadeType Enum containing entries:

<https://www.baeldung.com/jpa-cascade-types>

**ALL**: propagates all operations — including Hibernate-specific ones — from a parent to a child entity.

**PERSIST**: Cascade Type PERSIST propagates the persist operation from a parent to a child entity. Example, when we save the person entity, the address entity will also get saved. (Creating new entry)

**MERGE**: MERGE propagates the merge operation from a parent to a child entity. In other words, for existing objects, to merge the existing data in the table with the data in my object. (Synchronization to database for updated data)

**REMOVE:** REMOVE propagates the remove operation from parent to child entity. Example, if person gets removed, address associated with the person also gets removed.

**REFRESH:** Refresh operations reread the value of a given instance from the database. (Synchronization from database for any database changes)

**DETACH:** The detach operation removes the entity from the persistent context.

**Fetch Type:**

**Eager:** It will bring the related data always.

**Lazy:** It will only bring the related data when asked.

**Uni and Bi Directional relationship:**

Uni directional is when one side has the information of other side. Bi directional is when both side has information of each other.

**One-To-One relationship:**

**@OneToOne:** This annotation is used for one-to-one relationship.

**@JoinColumn:** It is used to configure the names of the column and referencing column.

For foreign keys whoever owns the foreign key gets this annotation. Other side we don’t need to use Join column annotation.

For Join table, we need to define it in both sides, but the configuration will be opposite.

**@MappedBy:** The *mappedBy* attribute tells that the *@ManyToOne* side is in charge of managing the Foreign Key column, and the collection is used only to fetch the child entities

**One-To-Many Relationship:**

One-to-many mapping means that one row in a table is mapped to multiple rows in another table.

**@OneToMany:** This annotation is used for one-to-Many relationship. The one side will keep reference of the many sides.

Can be bi or uni-directional as well. We need to use @JoinColumn for configuring the foreign key, or join table keys.

**Many-To-One relationship:**

**@ManyToOne:** This annotation is used for many-to-one relationship. The many side will keep reference of the one side.

This relationship is recommended whenever possible instead of one to many by JPA specification.

**Many-To-Many relationship:**

members of each party can hold reference to arbitrary number of members of the other party.

**@ManyToMany:** This annotation is used for many-to-many relationship.

**Note:** One case: For uni-directional relationship, we can swap @ManyToOne and @OneToMany with same specification, it won’t change the database columns (see example for course and teacher class in the project). It is our definition in the class only.

**Paging and Sorting:**

JpaRepository extends PagingAndSortingRepository. Here there is two overload of findAll() method. One takes a Sort object and other takes a Pageable object as parameter.

**QueryByExample:**

It allows dynamic query creation and does not require you to write queries that contain field names. In fact, Query by Example does not require you to write queries by using store-specific query languages at all.

The Query by Example API consists of three parts:

* **Probe**: The actual example of a domain object with populated fields.
* **ExampleMatcher**: The ExampleMatcher carries details on how to match particular fields. It can be reused across multiple Examples.
* **Example**: An Example consists of the probe and the ExampleMatcher. It is used to create the query.

Query by Example is well suited for several use cases:

* Querying your data store with a set of static or dynamic constraints.
* Frequent refactoring of the domain objects without worrying about breaking existing queries.
* Working independently from the underlying data store API.

Query by Example also has several limitations:

* No support for nested or grouped property constraints, such as firstname = ?0 or (firstname = ?1 and lastname = ?2).
* Only supports starts/contains/ends/regex matching for strings and exact matching for other property type