For the record, the spring.jpa.hibernate.ddl-auto property is Spring Data JPA specific and is their way to specify a value that will eventually be passed to Hibernate under the property it knows, hibernate.hbm2ddl.auto.

The values create, create-drop, validate, and update basically influence how the schema tool management will manipulate the database schema at startup.

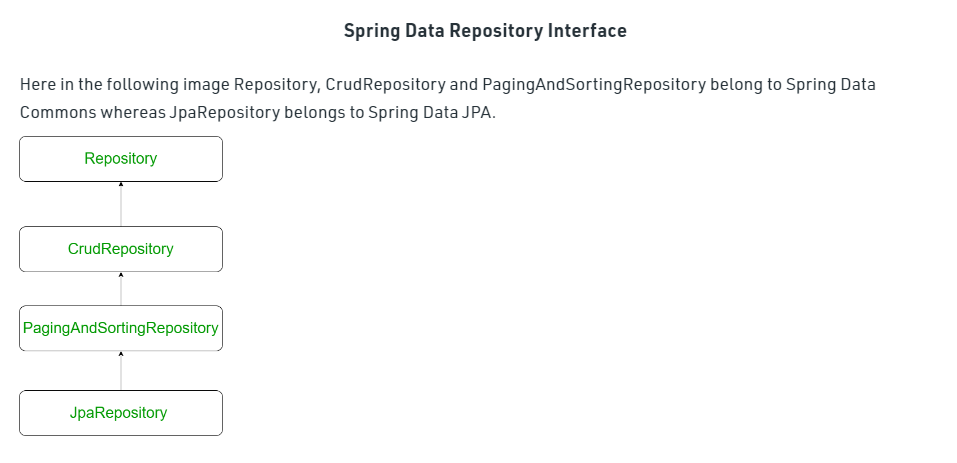
For example, the update operation will query the JDBC driver's API to get the database metadata and then Hibernate compares the object model it creates based on reading your annotated classes or HBM XML mappings and will attempt to adjust the schema on-the-fly.

The update operation for example will attempt to add new columns, constraints, etc but will never remove a column or constraint that may have existed previously but no longer does as part of the object model from a prior run.

Typically in test case scenarios, you'll likely use create-drop so that you create your schema, your test case adds some mock data, you run your tests, and then during the test case cleanup, the schema objects are dropped, leaving an empty database.

In development, it's often common to see developers use update to automatically modify the schema to add new additions upon restart. But again understand, this does not remove a column or constraint that may exist from previous executions that is no longer necessary.

In production, it's often highly recommended you use none or simply don't specify this property. That is because it's common practice for DBAs to review migration scripts for database changes, particularly if your database is shared across multiple services and applications.



**Difference between @Entity and @Table**  
  
@Table is optional. @Entity is needed for annotating a POJO class as an entity, but the name attribute is not mandatory.

If you have a class

@Entity

class MyEntity {}

A table with name "**MyEntity**" will be created and the Entity name will be **MyEntity**. Your JPQL query would be:

select \* from MyEntity

In JPQL you always use the Entity name and by default it is the class name.

if you have a class

@Entity(name="MyEntityName")

@Table(name="MyEntityTableName")

class MyEntity {}

then a table with name **MyEntityTableName** is created and the entity name is **MyEntityName**.

Your JPQL query would be :

select \* from MyEntityName

**What is @Access**

* The JPA Specification allows access the properties of a class **either directly through fields or indirectly through getter and setter methods.**
* JPA introduced standard annotation to control this behaviour in JPA 2.0 ([JSR-317](https://jcp.org/en/jsr/detail?id=317)).
* The default behaviour is, location of the mandatory id property of the POJO with @Id  annotation in hibernate determines the access level of this domain object.
* For example @Id annotation located at the field level then that entity access behaviour is **Field Access**  (OR) if @Id annotation located at the getter method entity access behaviour is **Property Access.**This is default behaviour.
* If access type is Property access JPA provider calls getter and setter methods runtime to load/store. For the Field access type JPA provider access fields directly, like how we can access fields within a class.

# **Spring JPA @Embedded and @EmbeddedId**

**3.1. *@Embeddable***

Let's define our *BookId* class in this section. The *author* and *name* will specify a unique *BookId* — the class is *Serializable* and implements both *equals* and *hashCode*methods:

@Embeddable

**public** **class** **BookId** **implements** **Serializable** {

**private** String author;

**private** String name;

// standard getters and setters

}

**3.2. *@Entity* and *@EmbeddedId***

Our *Book* entity has *@EmbeddedId* *BookId*and other fields related to a *book*. *BookId* tells JPA that the *Book* entity has a composite key:

@Entity

**public** **class** **Book** {

@EmbeddedId

**private** BookId id;

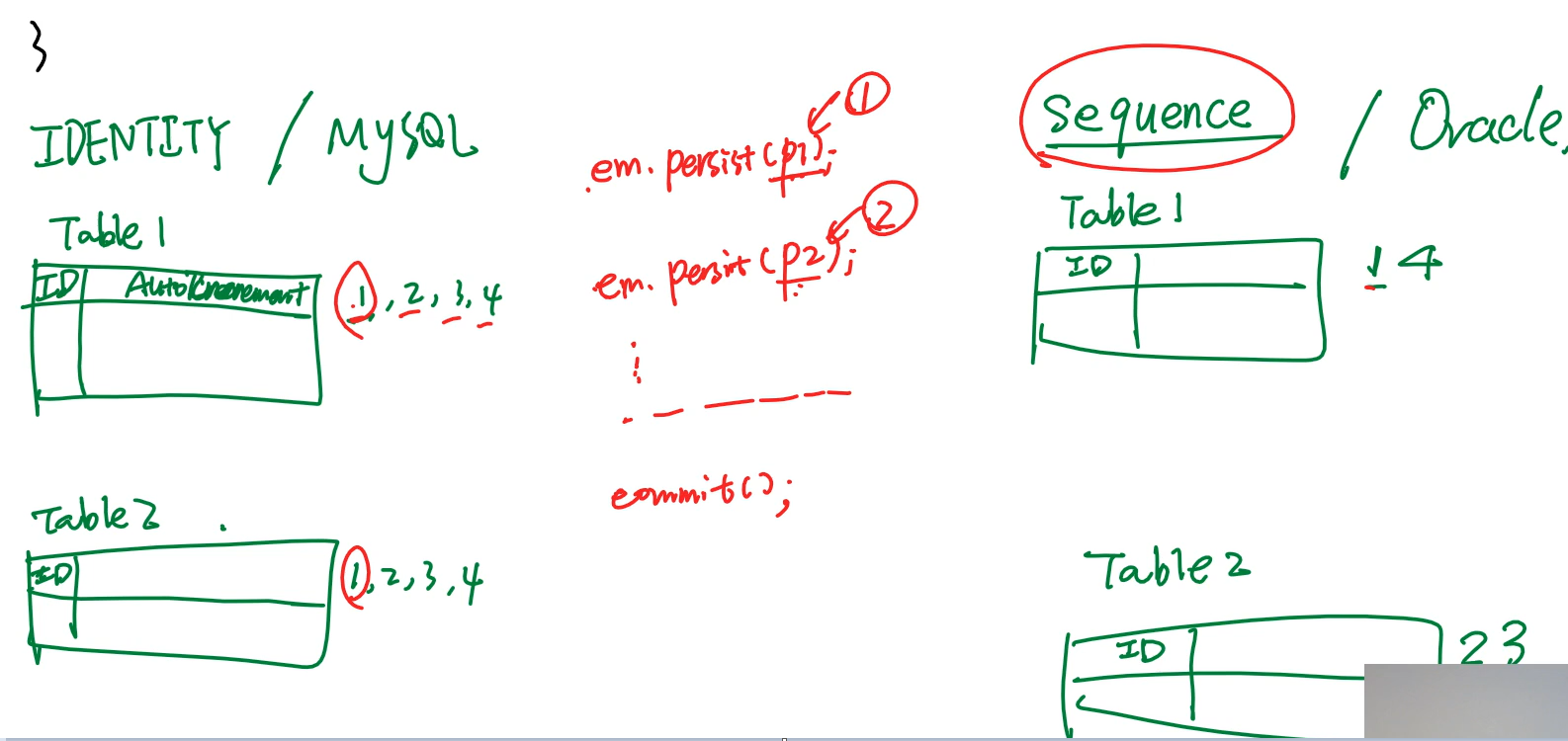
**private** String genre;

**private** Integer price;

//standard getters and setters

}

**Different types of @GeneratedValue**



IDENTITY

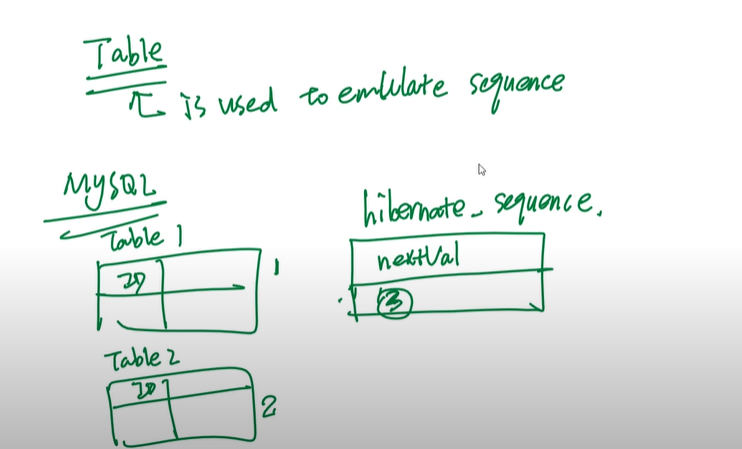
Generates incremental id for each table . There is no link between 2 tables

The problem here is hibernate cannot cache entities and then persist after sometime. Hibernate has to keep hiting the database to fetch the id each time and then increment

SEQUENCE

Generates incremental id and distributes between 2 tables.  
Here hibernate can hit the database once and then increment values for each entity and cache it

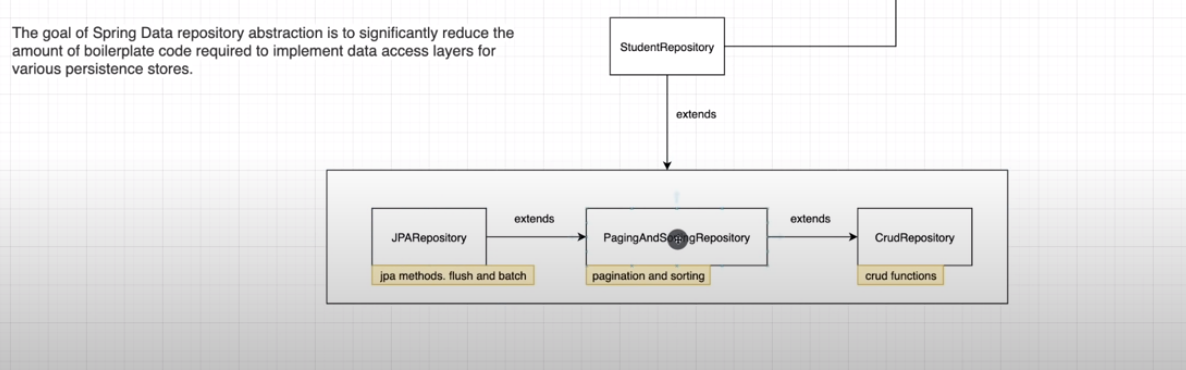
Not all databases support this type . Such databases can use TABLE generation type



TABLE

It is used to emulate Sequence generation type.

It creates a separate table to store the ids

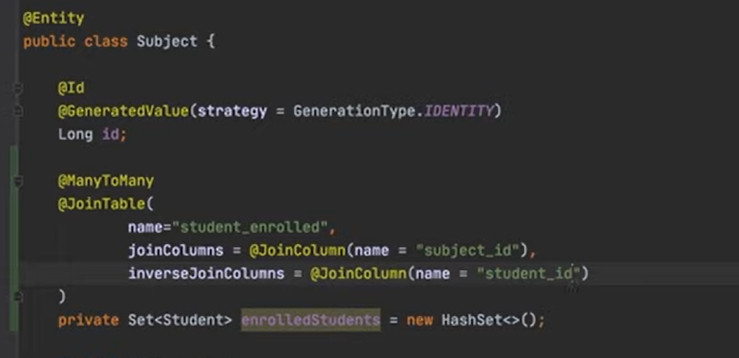


RELATIONSHIPS  
  
STUDENTS CAN HAVE MULTIPLE SUBJECTS

SUBJECTS CAN HAVE MULTIPLE STUDENTS  
  
MANY9 SUBJECT CAN HAVE ONLY ONE TEACHER

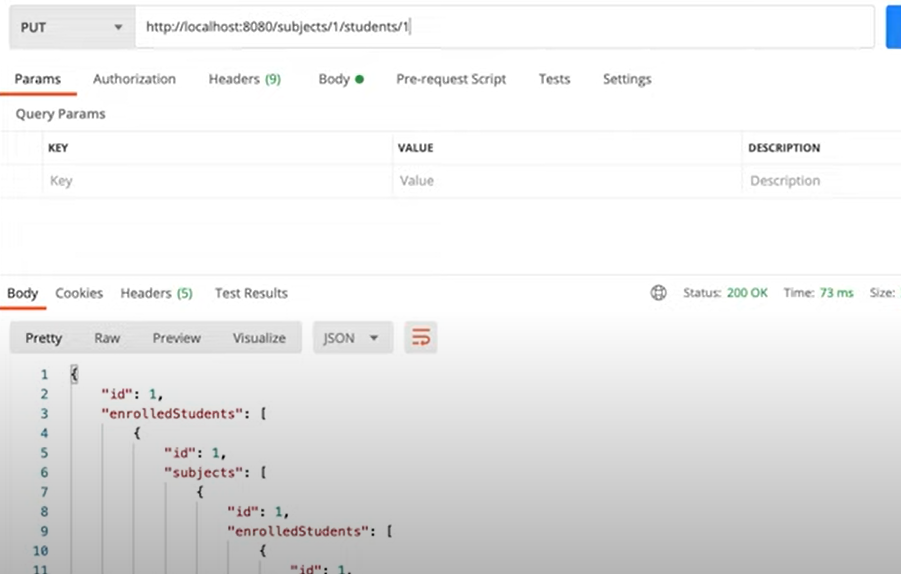
ONE TEACHER CAN HAVE MANY SUBJECTS

MANY TO MANY

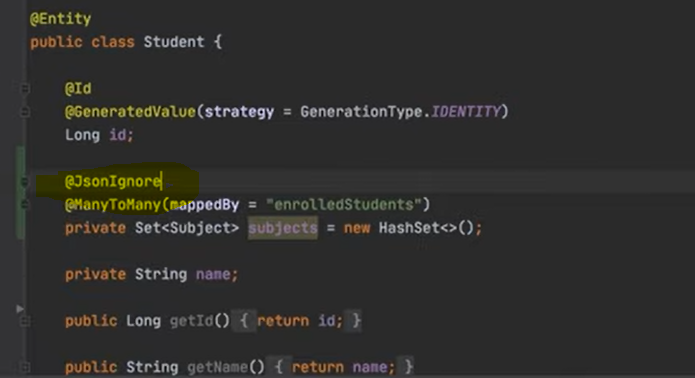




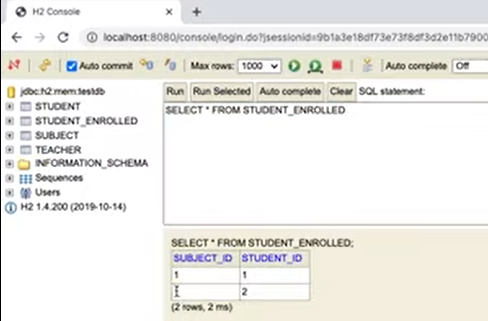
There is a recursive problem here



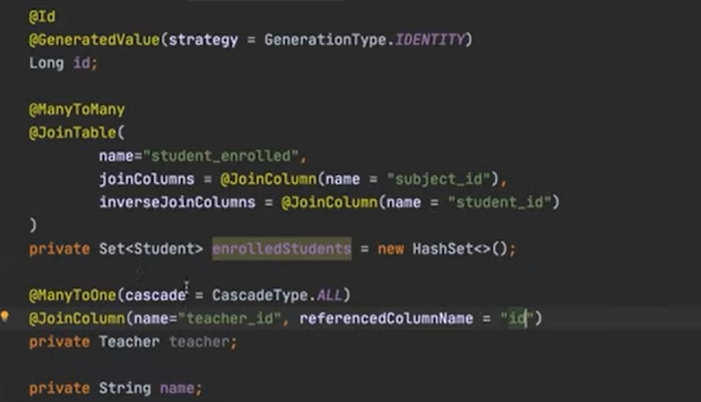
To avoid this just add jsonignore

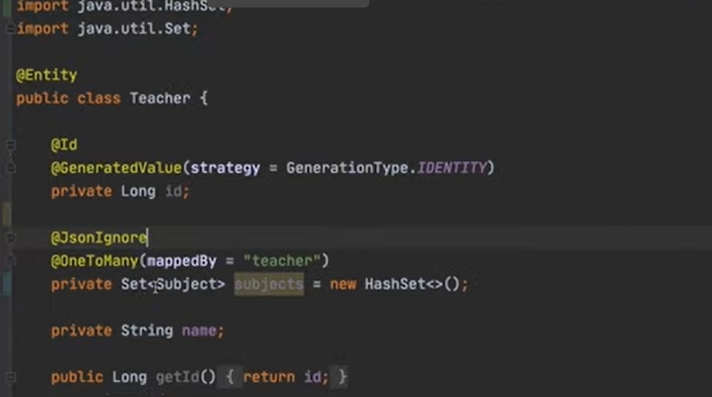


On the database side there is another table created to store the relations between student and subject

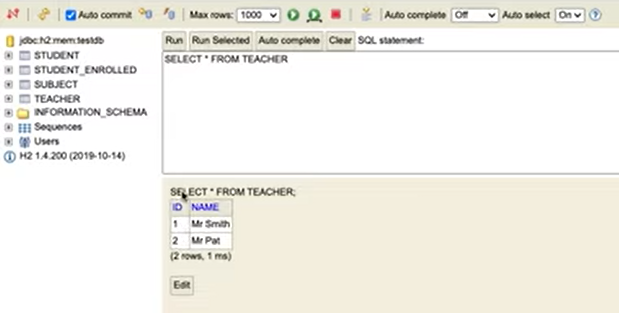


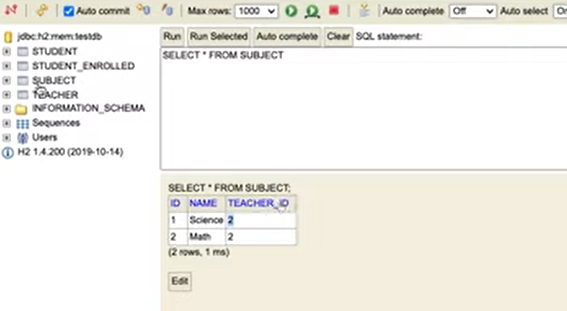
MANY TO ONE AND ONE TO MANY





How database looks





ONE TO ONE MAPPING ( ONE EMPLOYEE SHOULD HAVE ONE ADDRESS) UNIDIRECTIONAL (ONLY ON

