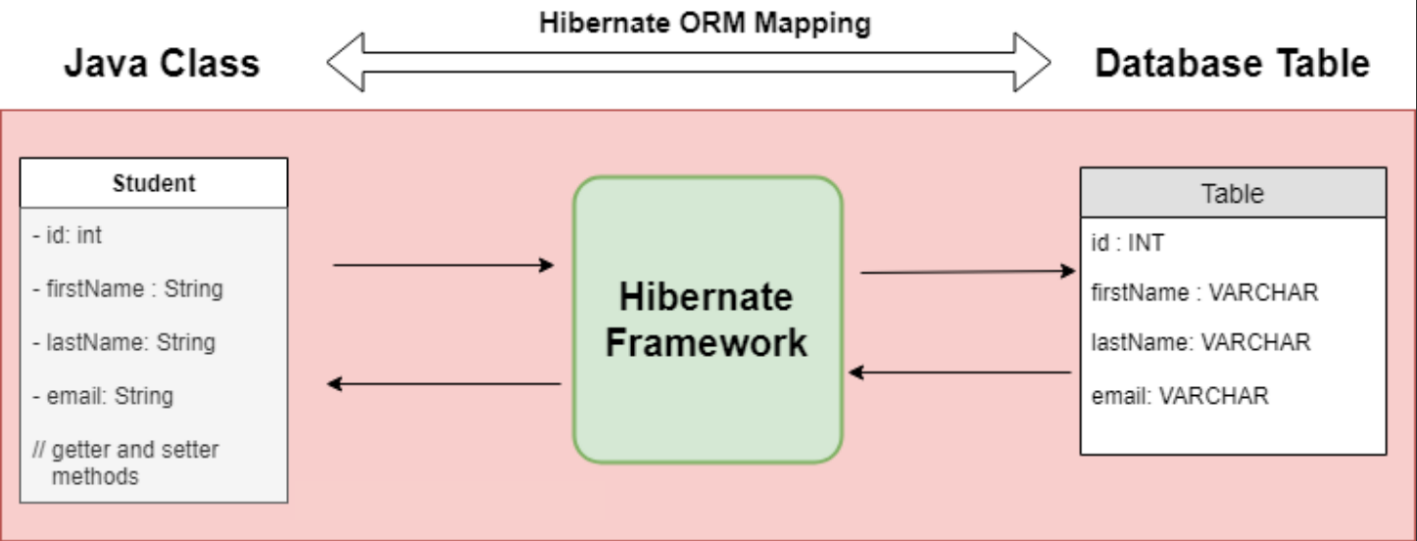
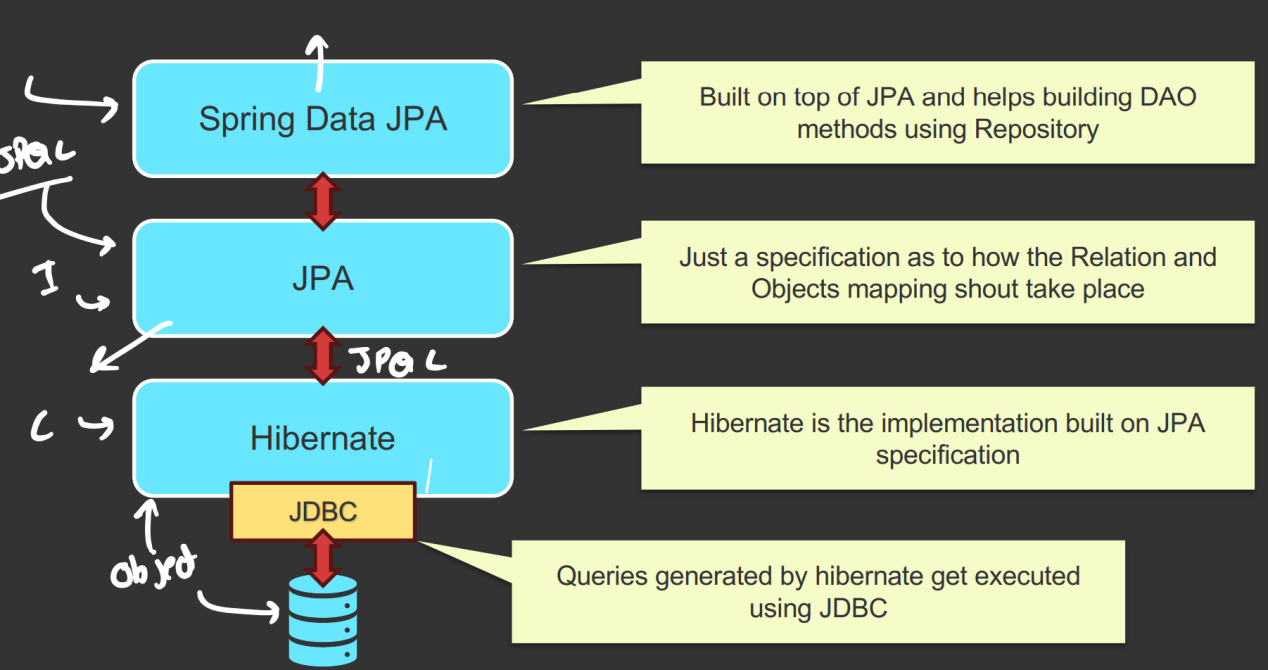
**Hibernate ORM Mapping**

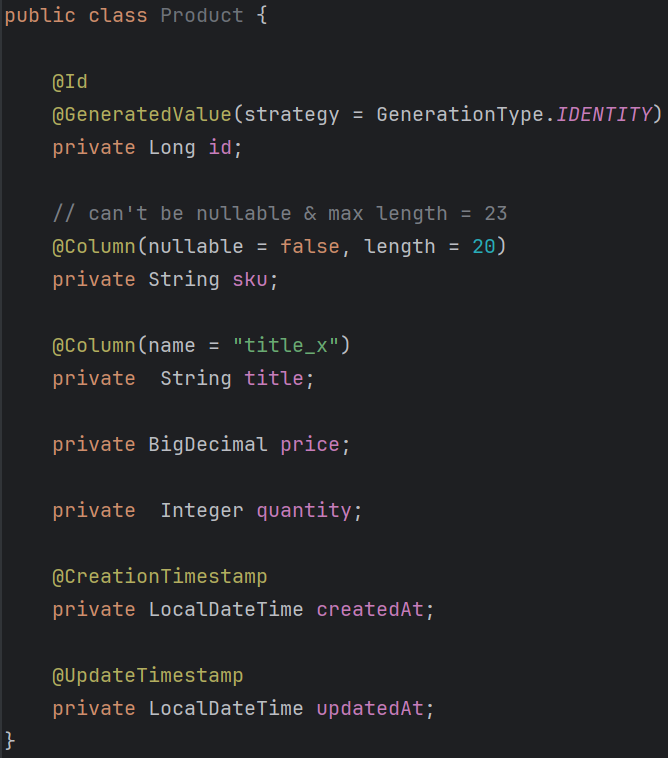
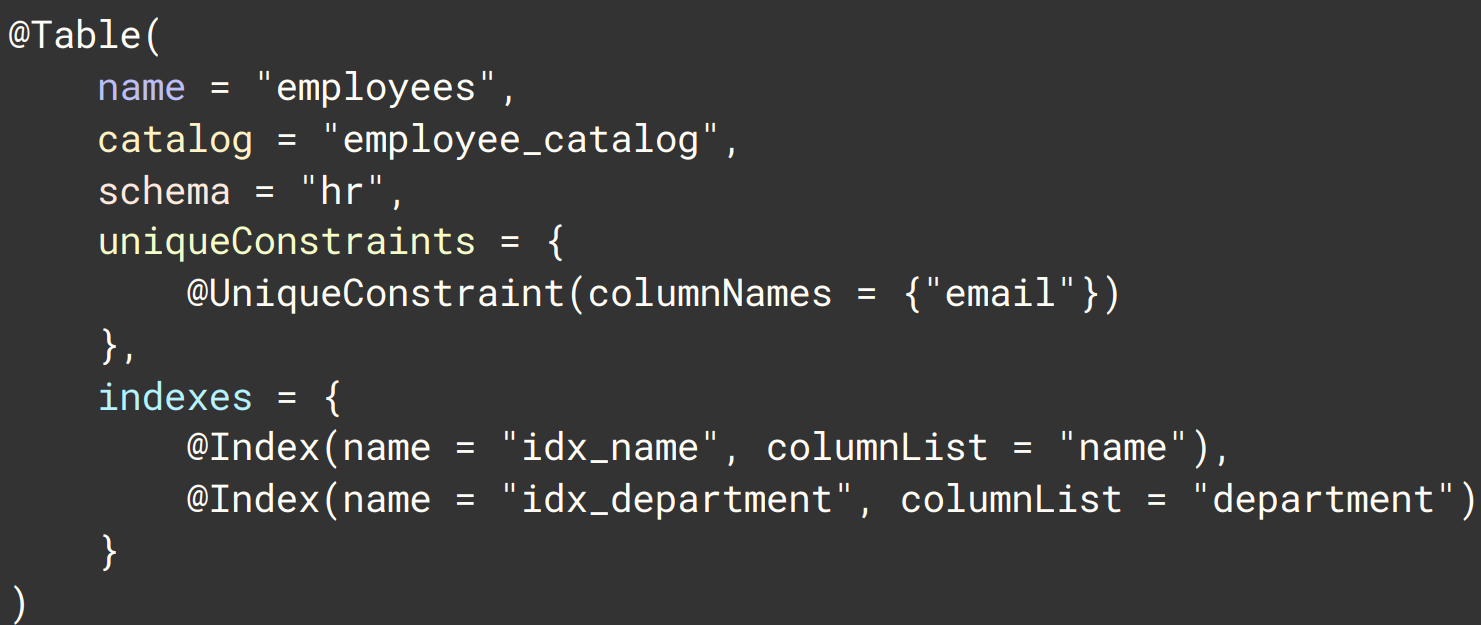
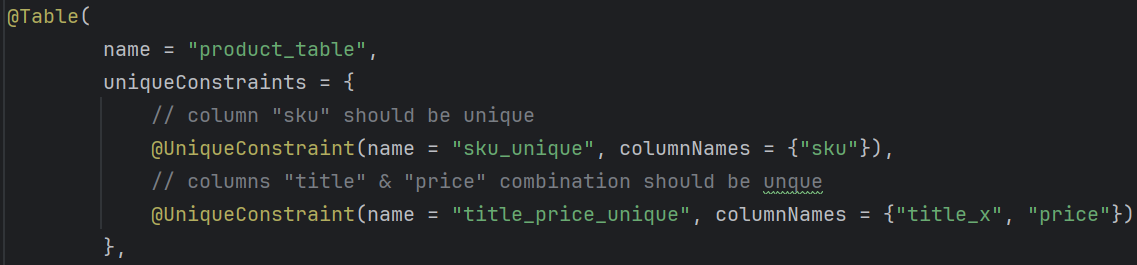
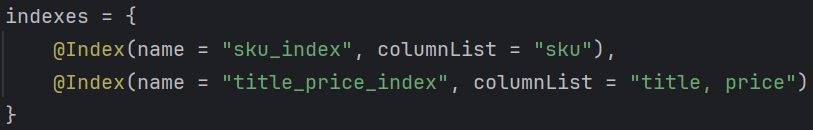


* 
* With the help of driver of particular database, we can connect JDBC to it.
  + Inside JDBC, we need to write SQL queries that will be supplied to Database.
* Then comes Hibernate, it is responsible for **Object-Relational-Mapping (ORM)**.
  + It’ll convert the specific Java object to Database relational entities.
  + Hibernate is the implementation built on JPA specification.
* JPA is just a specification as to how the Relation and Objects mapping shout take place.
* Then it comes Spring JPA.
  + It is built on top of JPA and helps building DAO (Data Object) methods using Repository.
* We can write on top of Spring Data JPA that is high-level data manipulation methods. Or also we can write JPQL on JPA level.

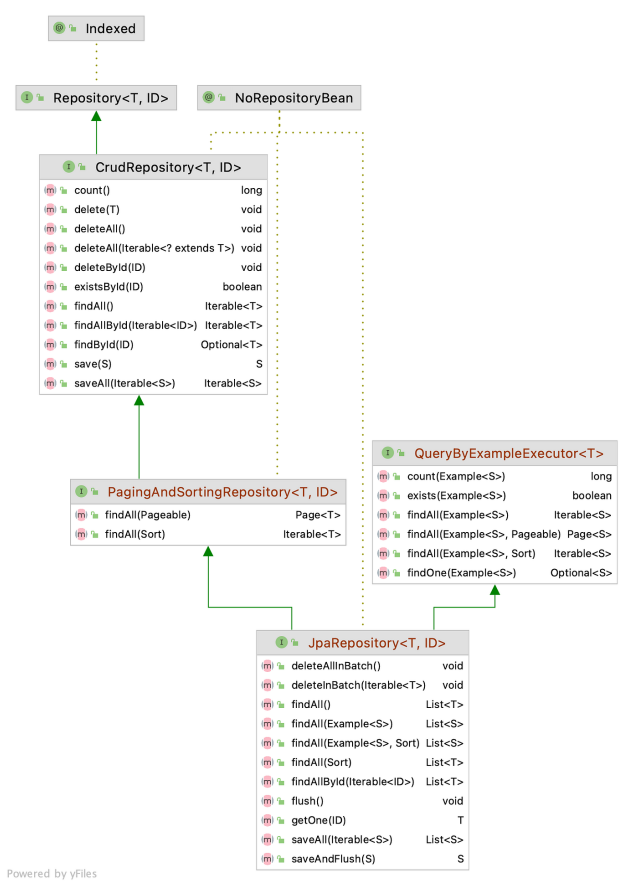
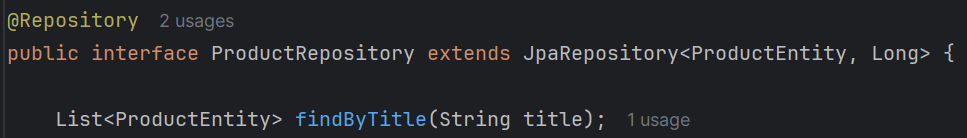
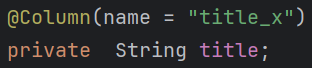
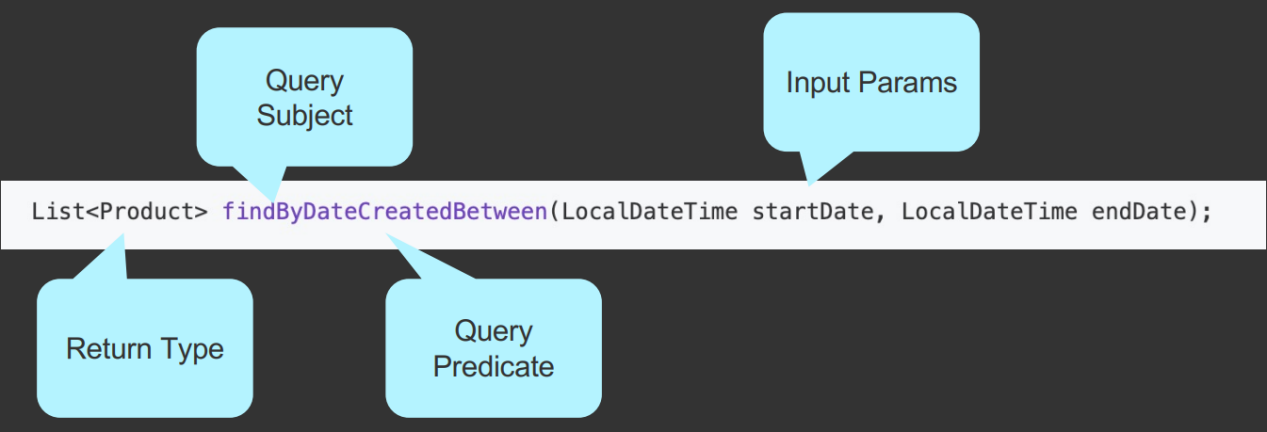
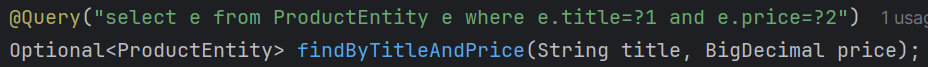
**The exact flow from Spring Data JPA to Database**

* First Layer is **SPRING DATA JPA**
  + Built by Spring on top of **JPA**.
  + Contains:
    - **JpaRepository interface** (extended by *user-defined repository interfaces*).
    - **SimpleJpaRepository class** (contains *method bodies of JpaRepository*).
  + SimpleJpaRepository already has implementations of CRUD methods (defined inside EntityManager interface of JPA).
  + No method implementation injection at runtime.
  + Spring creates a proxy that forwards repository method calls to **SimpleJpaRepository**.
  + Dependencies like **EntityManager** are **injected** at runtime.
* 2nd Layer is **JPA**
  + Pure Java specification.
  + Defines annotations and interfaces like **EntityManager**.
  + **SimpleJpaRepository** calls methods of **EntityManager**.
  + JPA provides only contracts, no implementations.
* 3rd Layer is **JPA Provider** (**Hibernate** is mainly used)
  + **Hibernate** implements **EntityManager** interface.
  + Provides actual method definitions.
  + Generates SQL queries.
  + Passes SQL to JDBC.
* 4th Layer is **JDBC**
  + Java API for DB communication.
  + Executes SQL generated by Hibernate.
  + Sends SQL to database drivers.
* 5th Layer is **Database Driver**
  + Executes SQL on the database.
  + Performs actual DB operations.
* Hibernate
  + It is a powerful, high-performance Object-Relational-Mapping (ORM) framework that is widely used with Java.
  + It provides a framework for mapping an object-oriented domain model to a relational database.
  + It is one of the implementations of Java Persistence API (JPA) which is a standard specification for ORM in Java.
* JPA
  + It is a specification for ORM in Java.
  + It defines a set of interfaces and annotations for mapping Java objects to database tables and vice versa.
  + It itself is just a guideline, doesn’t provide any implementations. Implementation is provided by JPA Provider framework like Hibernate.

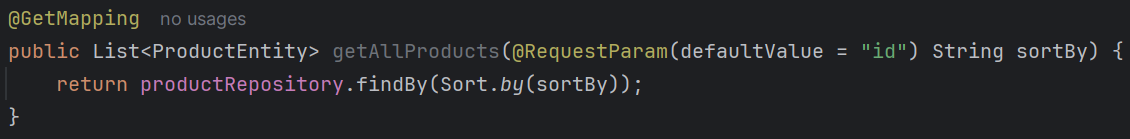
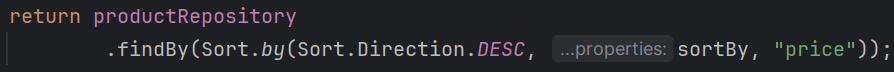
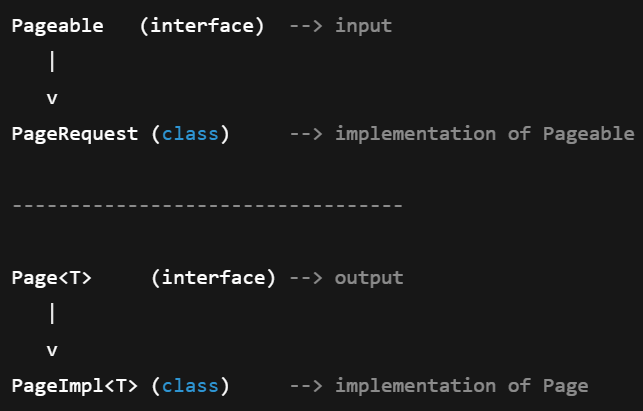
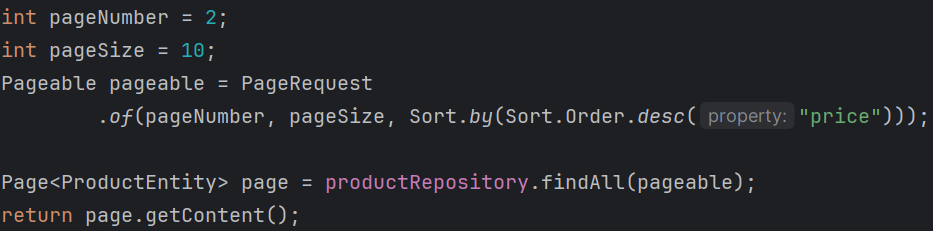
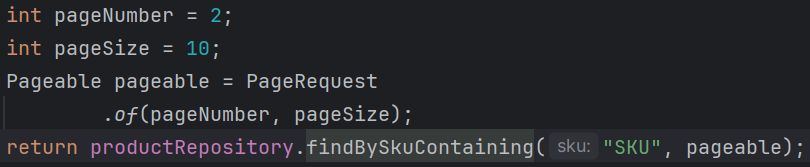
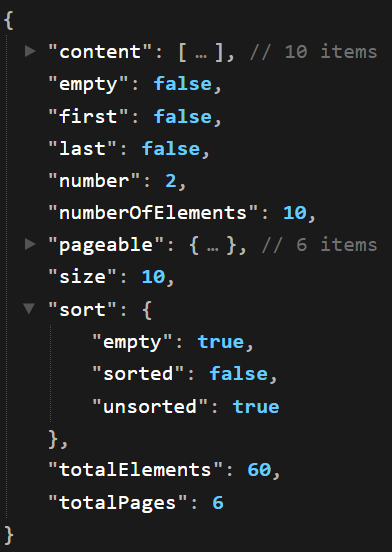
**Common Hibernate Configurations**

* **spring.jpa.hibernate.ddl-auto=update/create/validate/create-drop/none** (1)
  + Update: we want to update the table when we update the entity
  + Create: everytime we running the server, old table will be dropped and create a new.
  + Validate: the table that we have and entity that we have are matching or not
  + Create-drop: create table on running of server and drop that after stopping the server (not used in production)
* **spring.jpa.show-sql=true** (2)
  + If we want to see all the queries being generated underneath
* **spring.jpa.properties.hibernate.format\_sql=true**  (3)
  + The queries coming from the previous command (2) should be displayed after properly beautifying not in a single line.
* **spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySql5Dialect** (optional) (4)
  + Defines the rule that hibernate will use to convert JPQL to queries.
  + Database are having their own dialect.
  + Its optional because it’ll pick the proper dialect by itself.
* There are multiple annotations for **Entity** objects
  + 
  + **@Id**, **@GeneratedValue**, **@Column** (change name, nullable true or false, length if it’s a string, etc etc), **@CreationTimeStamp**, **@UpdateTimeStamp** …etc
* **@Table** annotation
  + 
  + There is something called **namespace** in database.
    - **auth.**user, **sales.**user
    - Here both have the same table name "user", but they do not conflict because they belong to different namespaces (auth, sales).
    - In MySQL, the database acts as the namespace (mapped using **catalog** in **@Table**).
    - In PostgreSQL and Oracle, the schema acts as the namespace (mapped using **schema** in **@Table**).
    - So, schema and catalog both represent the same concept (**namespace**), and which one is used depends on the database.
  + 
    - UniqueConstraint is also an annotation :).
    - 
    - 
    - (**title\_x** because we have changed the column name to **title\_x**; previous image)
    - Name is used to provide a specific name to the constraint. Otherwise it’ll generate some random unique name for the constraint.
    - **name** is useful during debugging.
    - (without name)
    - (with name)
  + **indexes** 
    - Here the **columnList** is a *String*not a *List*.
    - You should give comma separated column names.
    - 
* NOTE: **database** should already be present. It’ll not create the database inside the server by itself.
* **Indexing** in database.
  +  (JPA)
  +  (SQL)
  + An index is a *separate data structure* that stores indexed **column** values along with **row pointers**.
  + It is not a normal table; it is created and managed internally by the database..
    - 
  + But this is not a normal table, it is created and managed by database itself.
  + **Read** queries are ***faster***, but **create**, **update**, **delete** queries are ***slower*** as it needs to update the index table as well.

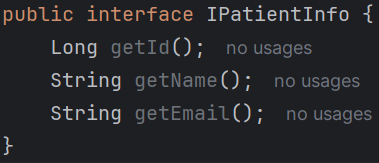
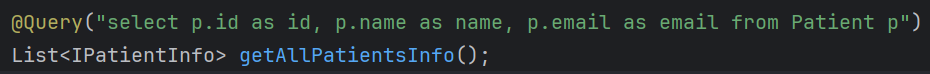
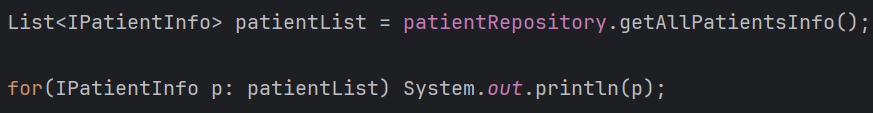
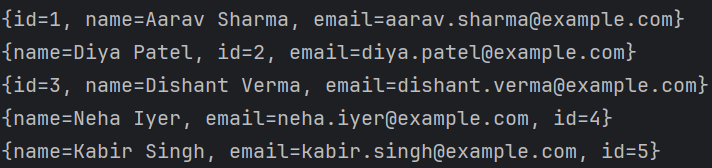
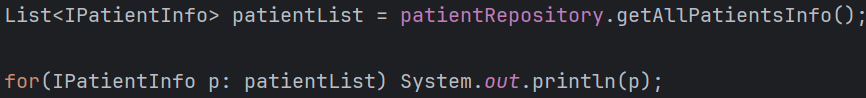
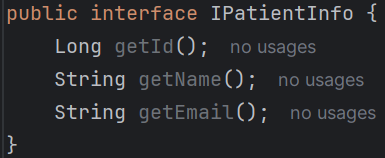
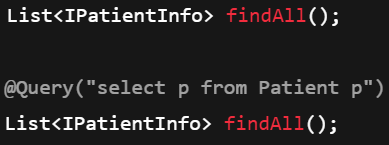
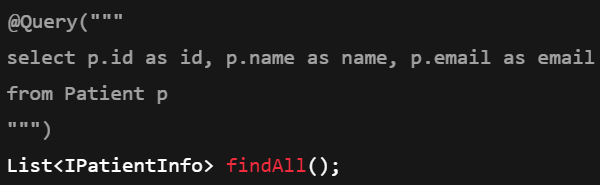
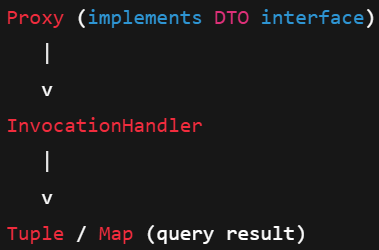
**Spring Data JPA**

* It is a part of the larger Spring Data Family.
* It builds on top of JPA, providing a higher-level and more convenient abstraction for data access.
* Spring data JPA makes it easier to implement JPA-based repositories by providing boilerplate code, custom query methods, and various utilities to reduce the amount of code you need to write.
* 
* **SimpleJpaRepository *class*** implements the *JpaRepository interface*. It contains implementation of all the methods of the JpaRepository and its parent interfaces.
* Key Features of Spring Data JPA
  + Repository Abstraction:
    - Provides a *Repository* interface with methods for common data access operations.
  + Custom Query Methods:
    - Allows defining custom query methods by simply declaring method names.
  + Pagination & Sorting:
    - Offers built-in support for pagination and sorting.
  + Query Derivation:
    - Automatically generates queries from method names.
* You’ll have to just write the method name in the Repository and no need to implement. It’ll be done automatically.
  + 
  + Just like this just write the method.
  + NOTE: If you remember the column name is **title\_x** not **title**.
    - 
    - **query generation takes place according to the Java object**; **not Database column**; If you write **findByTitleX** then it’ll not work;
* **Rules for Creating Query Methods**
  + 
    - Return type will be mostly **Entity**, **Optional<Entity>** or **List<Entity>**
    - In the diagram, **Query Subject** is **findBy**, and **Query Predicate** is **DateCreatedBetween**.
  + The name of the query method must start with one of the following prefixes
    - find..By, read..By, query..By, get..By
    - Examples: **findByName, readByName, queryByName, getByName**
  + If we want to limit the number of returned query results, we can add the ***First*** or the ***Top*** keyword before the first by word**.**
    - Examples: **findFirstByName**, **readFirst2ByName**, **findTop10ByName**
  + If we want to select unique results, we have to add the Distinct keyword before the first ***By*** word.
    - Examples: **findDistinctByName or findNameDistinctBy**--- both are same
  + Combine property expression with ***And*** and ***Or***
    - Examples: **findByNameOrDescription, findByNameAndDescription**
  + For more, refer to the link: **[query keyword reference](https://docs.spring.io/spring-data/jpa/reference/repositories/query-keywords-reference.html)**
* A few examples:
  + 
    - To get all the items that were created after a particular time.
  + **findByQuantityGreaterThanAndPriceLessThan(int quantity, int price)**
    - The argument orders should be same as the query.
* Writing **JPQL** query directly
  + 
  + **@Query** annotation is used to write the **JPQL** query.
  + **?1 , ?2 , ?3** etc are used to get the argument from the method. In the above image
    - **?1** means **title**
    - **?2** means **price**
  + Instead of **?1 ?2** you can also write **:title** and **:price**
    - 
  + **NOTE**: JPQL should be written according to Java object (inside Entity), not according to Database table/column name.
    - In database the table name is **product\_table** as I have mentioned inside the **@Table** annotation, but the entity name is **ProductEntity**, so we need to pass **ProductEntity** in the JPQL.
    - Also, in the database the column name is **title\_x** as I have mentioned **title\_x** in the annotation **@Column**, but the field name is **title** in side **ProductEntity** class. So we need to pass **title** in the JPQL query.
    - And also, we are not writing **select \*** just like SQL, rather we are writing **select e**. here *select \** will not work.

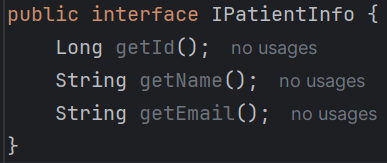
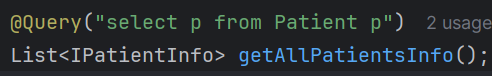
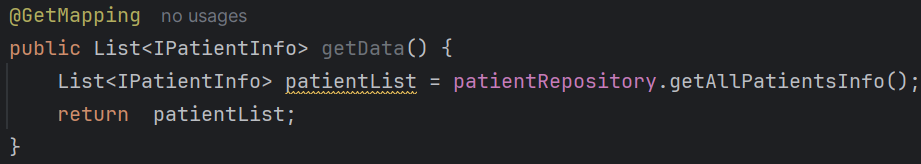
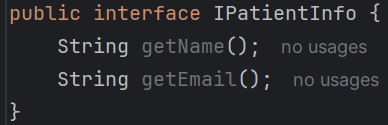
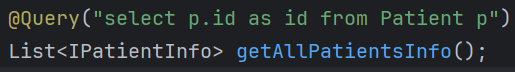
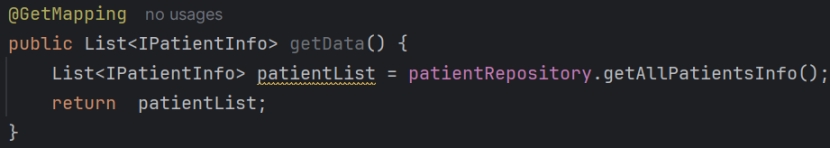
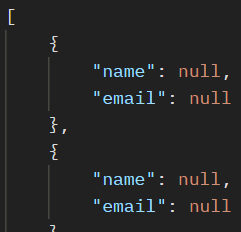
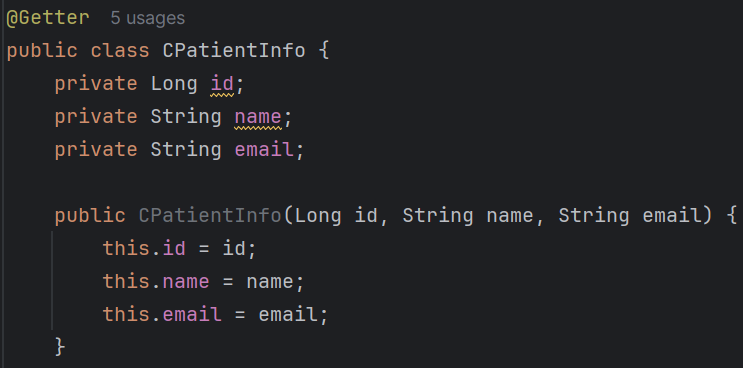
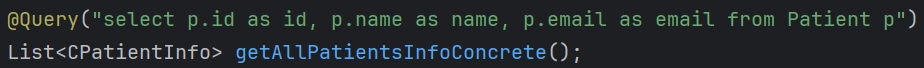
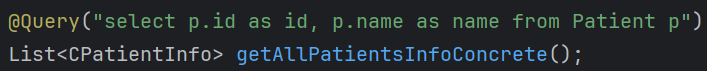
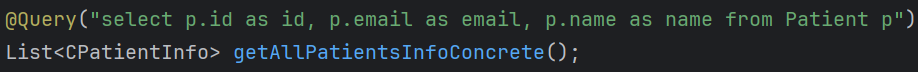
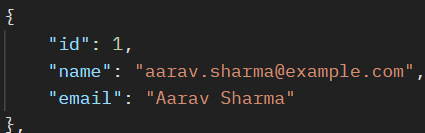
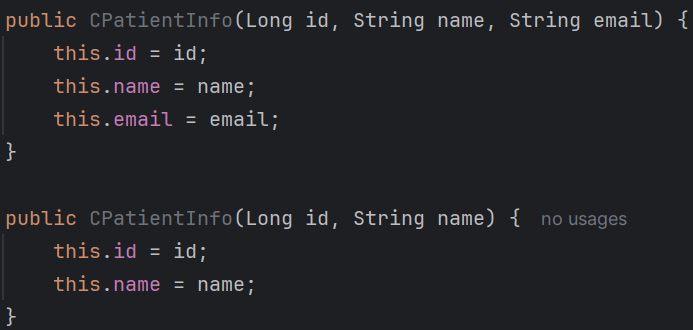
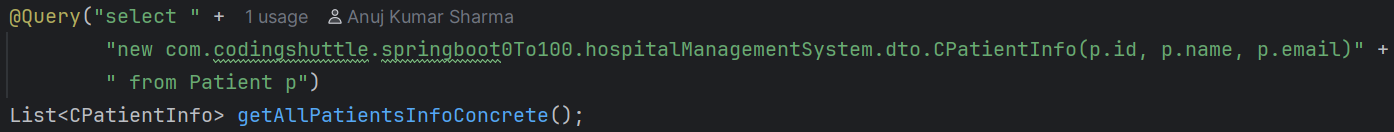
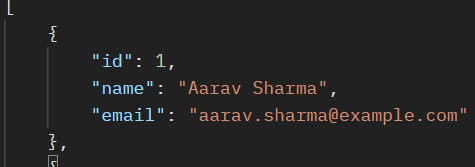
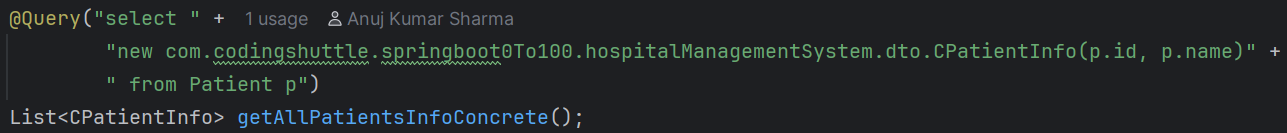
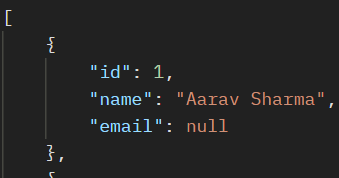
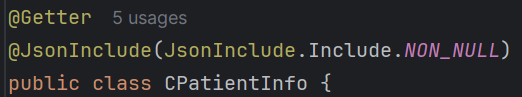
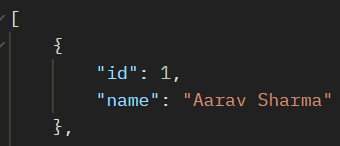
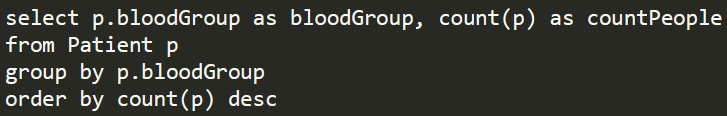
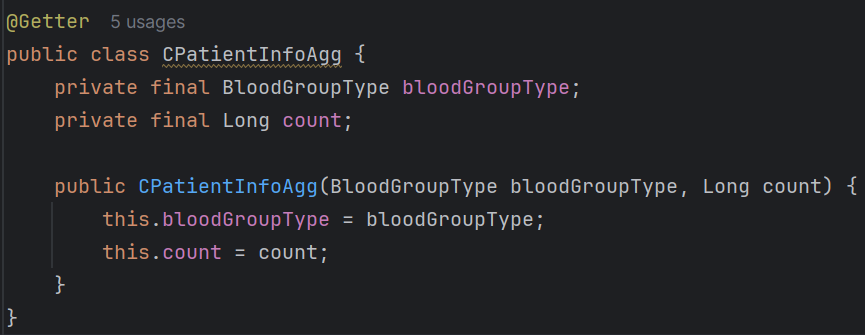
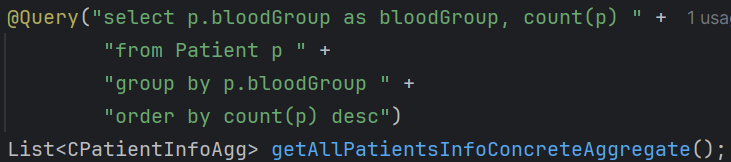
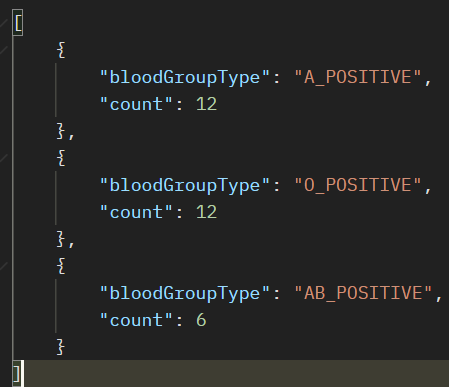
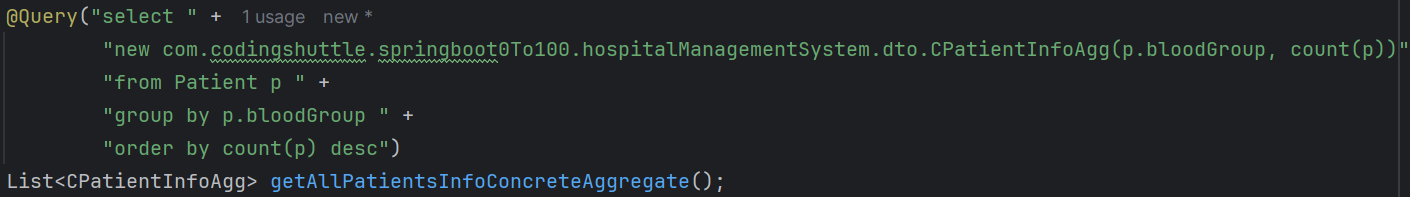
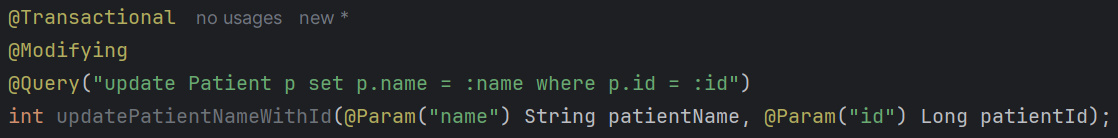
**Sorting & Pagination**

* **OrderBy** is used to *Sort*.
  + 
  + Instead of **findAll**, here we need to write **findBy**.
  + It’ll get all the rows and sort them according to the *price*. **OrderByPrice**.
  + 
  + It’ll also do the same, but it’ll sort in *reverse order*.
  + **Asc** is for ascending, **Desc** is for descending order.
* But it is not proper; because if we want to *sort by some column*, then for each column we need to write one method each. For example *findByOrderByPrice, findByOrderByTitle, findByOrderById* …etc
  + For this we can use **Sort** class.
  + 
    - Here we need to get the argument of type **Sort**.
  + 
    - We can call that method like this. Passing one **Sort** object.
  + Now, the API call should be made confirming according to which column it has to be sorted and its done.
    -  like this
  + 
    - If the **sortBy** column is same, then it’ll further sort according to “price”. like this we can give multiple properties.
  + 
    - You can give the direction like this as well.
  + 
    - If you want different direction i.e. ASC, DESC for different columns then use like this.
* 
  + **Pageable** interface represents: **page size, page number, sorting info**
  + **PageRequest** is used to create **Pageable** object.
  + The query after being run returns **Page** object.
    - Typically we never create object of **Page**.
* **Pageable** is the interface; **PageRequest** is the class which inherit **Pageable** (not direct parent, but ancestor).
  + - We need to create one **Pageable** object to do the pagination.
    - **PageRequest** is used to create the object (as of course it is the class; and Pageable type of variable can keep PageRequest type of object; because Pageable is the ancestor of PageRequest)
    - The query methods can return **Page** type of object (if you write return type as List then of course it’ll return List instead of Page)
* 
  + **findAll** method is already there.
  + 
  + It returns an object of type **Page**.
  + **getContent()** method is used to extract the **List of Entity** from the **Page**.
* **NOTE ----------------------------------------------------------------------------------**
* The return type of the query doesn’t only depends upon **Pageable** parameter. It depends on the return type of the method.
  + 
  + I wrote this method, and the return type is **List<ProductEntity>** so here **List** will be returned.
  + 
  + But if I write **Page<ProductEntity>** in that query then it’d have return Page type of object.
  + 
  + 
    - Now I need to use **getContent()** method to get the List out of the Page object.
* By default, no query method supports **sorting** or **pagination**. You need to pass **Sort** type of parameter to sort and **Pageable** type of parameter to make pagination.
* --------------------------------------------------------------------------------------------------------------
* If you return **Page<ProductEntity>** instead of **List<ProductEntity>** then you’ll get some **metadata** along with the **contents**.
*  (like this)

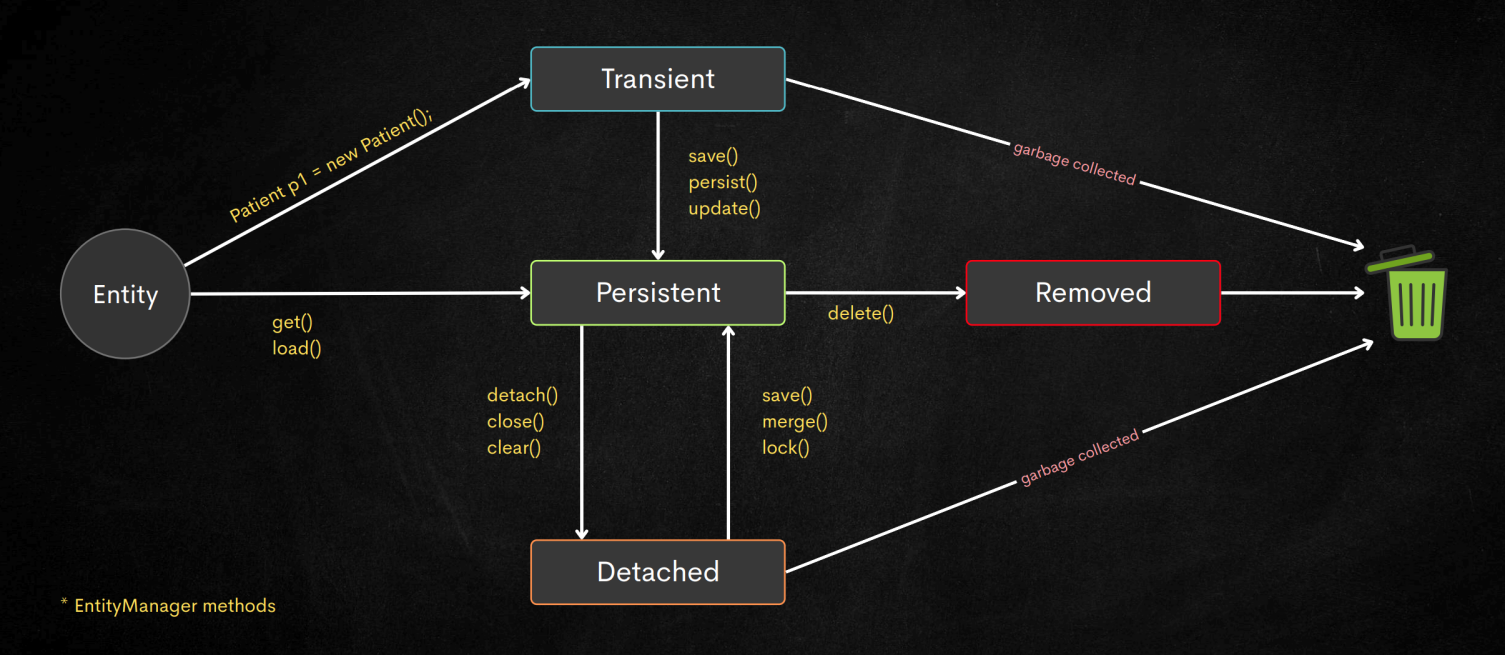
**Projection in Spring Data JPA**

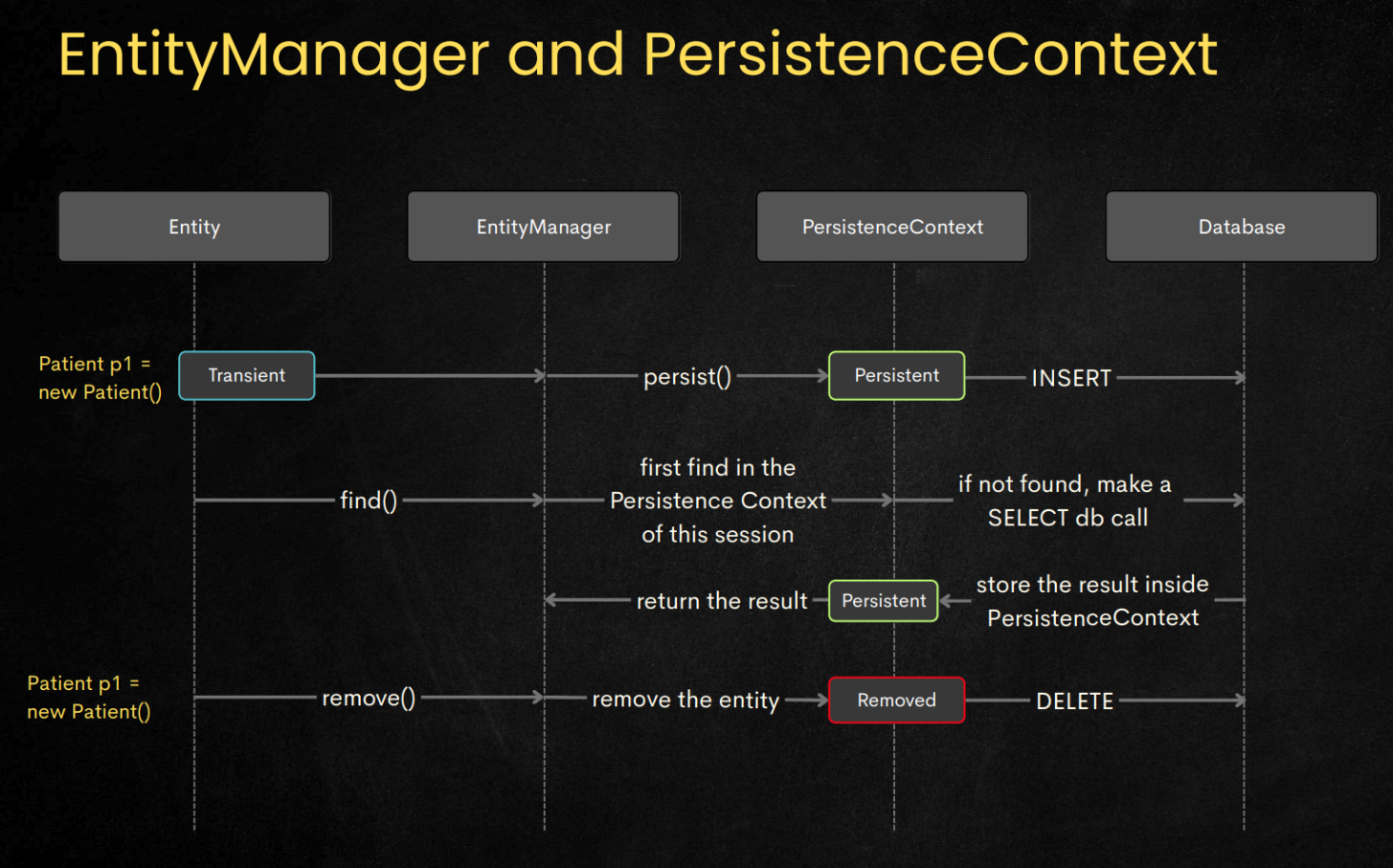
* A projection is a mechanism that allows you to define what data should be exposed to the **caller**, independent of how much data is fetched from the database.
* Returning full data can still be a projection if you are controlling what the caller can access.
* **Method-1 (DTO Interface)**
  + We are giving DTO directly to the repository; but as DTO interface is a view only model; so it doesn’t hamper the design pattern.
  + 
  + But there is no **variables** here so modification is not possible.
  + 
  + 
  + 
  + It’ll work.
  + In the above query i.e. **“select p.id as id, p.name as name, p.email as email from Patient p”** the aliases are important i.e. **id**, **name**, **email**,
    - **id** will be mapped to **getId**
    - **name** will be mapped to **getName** ..etc
* **NOTE**
  + In case of **interface DTO**, it doesn’t have any field. So **Spring Data** will not be able to create an object of this interface and return.
    - So, it creates one proxy class that implements the DTO interface.
    - Now DTO’s getter methods will be forwarded to Entity’s getter methods with the help of that Proxy.
    - Means the data you are getting is from the Entity itself.
    - 
    - 
    - When you execute this, You will see **entity-like output** because **toString()** is delegated to the entity. And it is basically **entity.toString()** .
      * For each object, **toString()** call was delegated to **toString()** of **entity**
    - Printed output ≠ actual object type
    - Actual object = proxy
    - 
  + In case of **class DTO** (lets assume only getters are there; no setters).
    - In this case, Spring/Hibernate doesn’t need to create one proxy class because it can directly create one object of type **DTO class** because it’s not an interface.
    - So, in case of **class DTO** (even if setters are not there), the object of type DTO class will be returned; No proxy class is required here.
* This is my **DTO Interface** (for reference of next explanations)
  + 
* 2 Types of projections are there:
  + **Entity-backed Projection**
  + **Tuple-backed Projection**
* **Entity-Backed Projection**
  + 
  + Both are same, if you don’t give the query then it’ll by default write that query only (which is being mentioned in the image)
  + So, here it is fetching the full **entity objects** which is of type **Entity**.
  + Flow will be like:
    - **Hibernate** fetches **Patient Entity** object from the database (As full **entity** has to be fetched according to the query)
    - **Spring** creates a **proxy** that implements the **DTO Interface** (IPatientInfo in our case)
    - **Proxy** holds a reference to the **InvocationHandler**.
    - The **InvocationHandler** holds a reference of the **Entity**.
    - Now, whenever the method will be called from the DTO will be delegated to Entity. Proxy is responsible for this.
      * **dto.getName()** ---- Proxy --- handler ----- **entity.getName()**
  + Here no data is copied;
  + No aliases are needed;
  + The data directly comes from the **entity**.
  + If you are wondering how it is able to know about the **Entity**, then you can remember we were passing **Entity** and **Id type** in the generics of **JpaRepository**.
* **Tuple-Backed Projection**
  + 
  + Here you need to give the query; and instead of fetching the whole **entity**, only select some specific columns that is being mentioned in the **DTO interface**.
  + In this case **aliases** are needed.
  + As here only some specific columns are being fetched from the table, so there is no need of keeping reference of **Entity**.
  + Flow:
    - **Hibernate** does not create entities.
    - Query returns selected column values.
    - **Spring** keeps the **tuple/map** inside the **InvocationHandler**.
    - **Spring** creates a **proxy** that implements **DTO Interface** (IPatientInfo in our case)
    - **Proxy** holds a reference of **InvocationHandler** (just like previous case)
    - Previously **InvocationHandler** was holding a reference of **Entity,** but in this case it is holding a reference of **Map/tuple**.
    - When **getter** is called, **proxy** delegates that method call to **InvocationHandler**, which reads from **tuple/map** and returns the result.
    - 

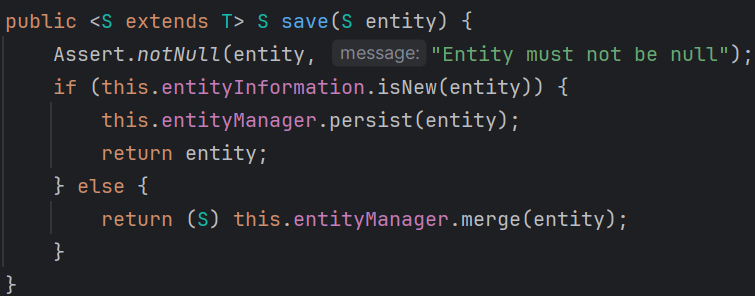
**Some experiments:**

* **Experiment-1**
  + Interface DTO (id, name, email) (IPatientInfo)
    - 
  + PatientRepository (fetching the whole **entities**)
    - 
  + PatientController (returning the IPatientInfo type object)
    - 
  + The response will be proper
    - 
  + Reason:
    - **IPatientInfo** (Interface DTO) is having the getters for **name, id, email** which is delegated to **Patient** (Entity).
    - From the **entity objects** the **getter** methods are being called and the result is properly being generated.
* **Experiment-2**
  + Interface DTO (name, email) (IPatientInfo)
    - 
  + PatientRepository (fetching the **id** of the entities)
    - 
    - **alias** is being used for **id** (here **getId()** will be called; otherwise it’d be null without alias)
  + PatientController (returning the IPatientInfo type object)
    - 
  + Now **name** and **email** will be **null**
    - 
  + Reason:
    - IPatientInfo (Interface DTO) is having the getters for name, email.
    - But now **getName()** and **getEmail()** are not there because now we don’t have **entity** objects; rather we have **map/tuple** which are containing only **id**. So, here only **getId()** will work.
    - But **IPatientInfo** contains **email, name** so it is null now.
* Interface DTOs are not flexible;
  + Only one type of response can be sent.
  + The object cannot be edited as it was coming from the Map(in case of few columns) or Entity(in case of whole entity). So, Interface DTO are only view-only.
  + So, class based DTO are used which can be modified, multiple types of responses are supported.
* **Class DTO**
  + If the DTO class is containing only one constructor, then the JPQL query can be written normally like Interface DTO.
    - (class DTO)
  + 
    - For this query, the response will be like the below:
    - (all fields are coming)
  + 
    - It’ll give “Internal Server Error” because the constructor is accepting 3 arguments but we are giving only one.
  + **NOTE:** In the JPQL query, the fields should be in same order as defined in the constructor; otherwise wrong fields will be passed in the constructor.
    - 
    - (it’ll give wrong response like this)
    - If the type of fields are different then it’ll give error instead of giving response.
* So, till now we were having only single constructor, but in case of multiple constructor we need to pass the **DTO class reference** in the JPQL query instead of writing like this normally.
  + 
    - So now I have 2 constructors.
  + 
    - Like this you need to write.
    - **new** keyword is must to create object.
    - No aliases are required here.
    - (response)
  + 
    - Here I passed just 2 fields (**id** and **name** as per the 2nd constructor).
    - 
    - Now if you see, all the fields are coming but having **null** values.
    - To avoid that **@JsonInclude(JsonInclude.Include.NON\_NULL)** is used.
  + 
    - Now I added this here.
    - 
    - Now if you see, only the non-null fields are being displayed.
* **Aggregate JPQL queries**
  + **NOTE:** Aggregate functions can have **aliases** but those aliases should not be used for **order by** and **group by**.
    - 
    - You can see here, I have given alias for **count(p)** which is **countPeople** but that alias name is not given on the side of **order by**.
    - I have mentioned **count(p)** on the side of **order by**.
    - Also, you cannot give **\*** just like SQL query. Here you need to give the object name (**p**)
    - Instead of field name, **p.fieldName** should be given in **camelCase** (in db it’ll be in **snake\_case**)
    - This is only valid for JPQL queries.
  + 
    - This is my **DTO class** for that aggregation query.
    - 
    - It’ll give the proper response.
    - 
  + 
    - It is preferable to use **class reference** to write the JPQL queries.
* **@Params** is used to bind a *parameter name* to a *particular JPQL value*.
  + 
  + Here I am executing one **update** query.
  + You can see the parameter names are **patientName** and **patientId** but I have bound those with **name** and **id** respectively using **@Params** annotation.
  + And also I am using **:name** and **:id** in the JPQL query.
* **@Modifying** is required in case of **modification** of database.
  + By default **Spring** thinks that the query is of **select** type. So, it expects some **data to be returned** from the database.
  + Hence, it’ll read those data and give response.
  + But, in case of **create/update/delete** method, no data is returned; only one number of *rows affected* will be returned.
  + So, **@Modifying** will tell spring that don’t expect any data to be returned.
  + **@Modifying** will always go with **@Transactional** because you are changing the database state. And it means its involving one **transaction**.

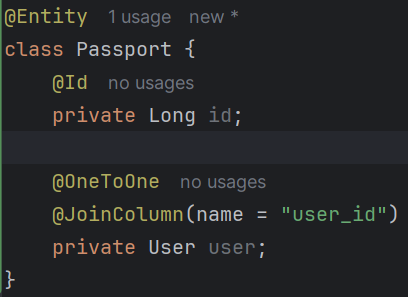
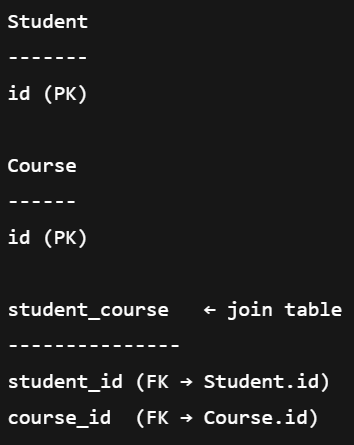
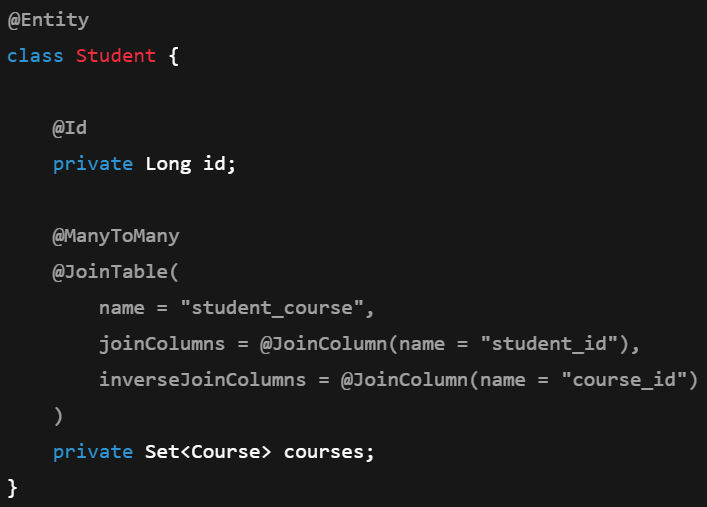
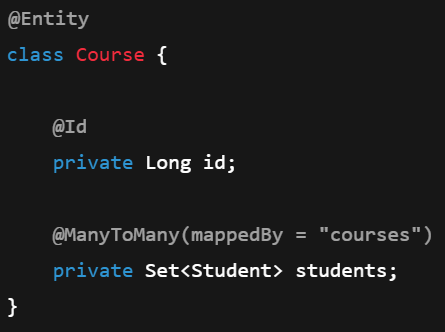
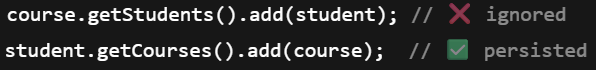
**Hibernate Entity Lifecycle, Entity Manager, Persistence Context**



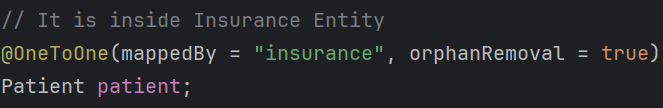
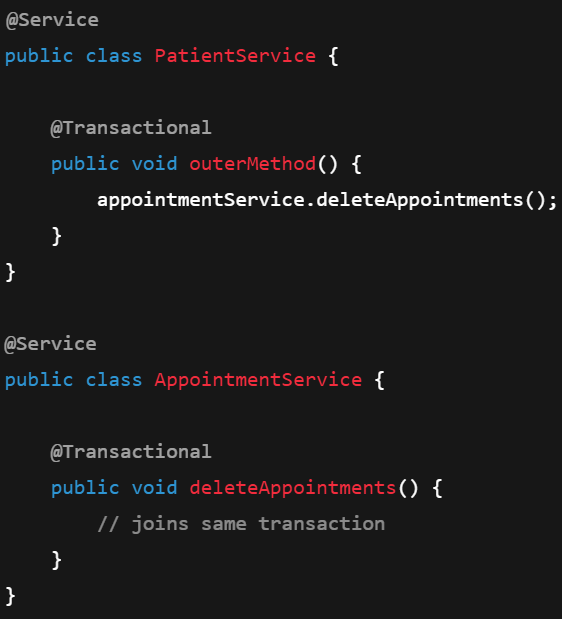
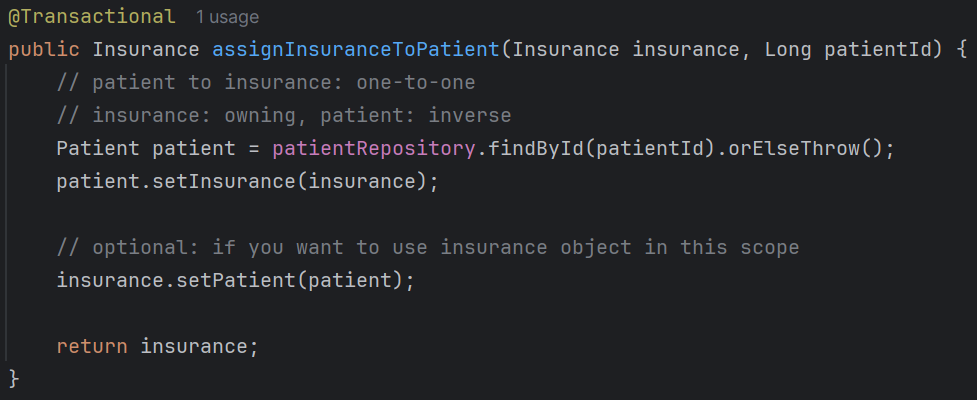
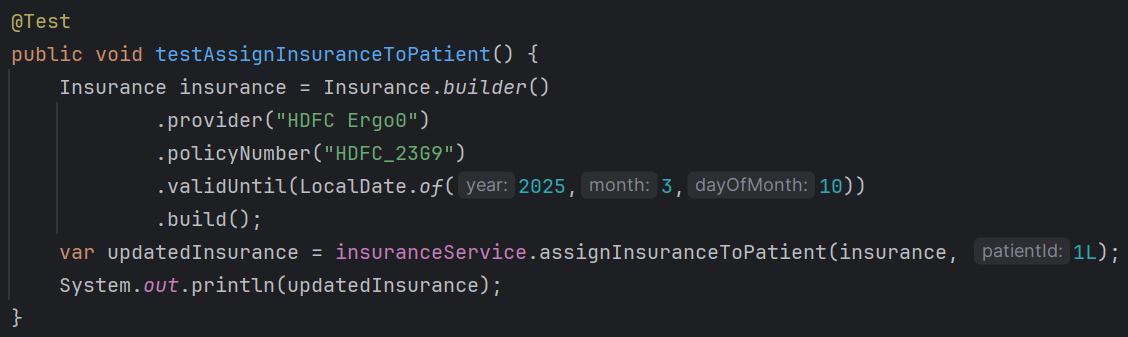
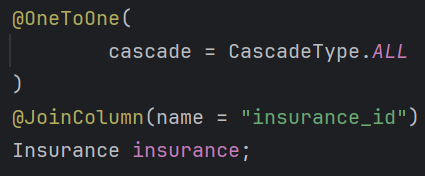
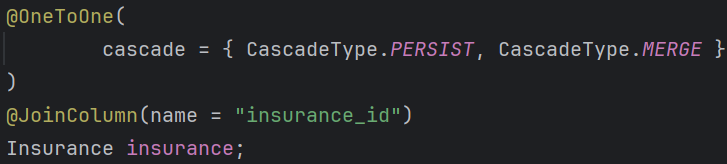
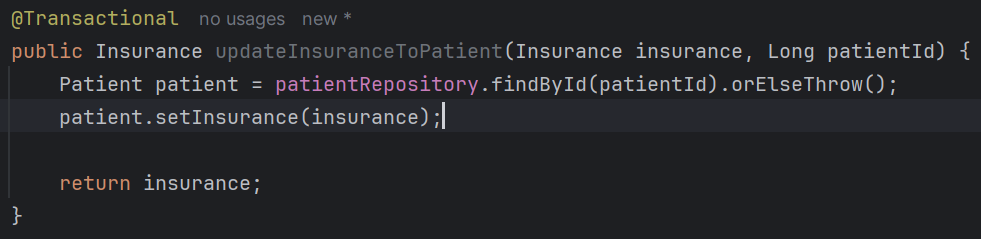
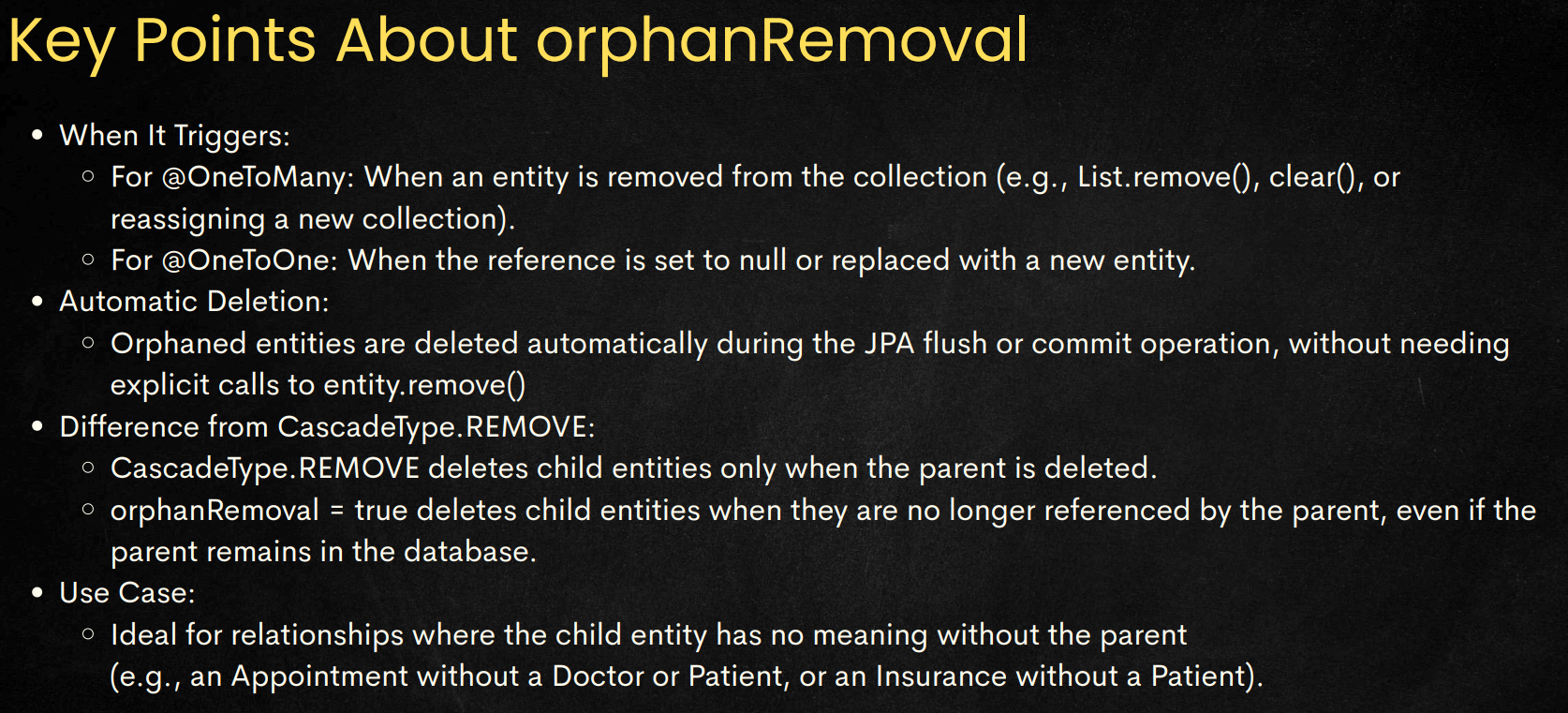
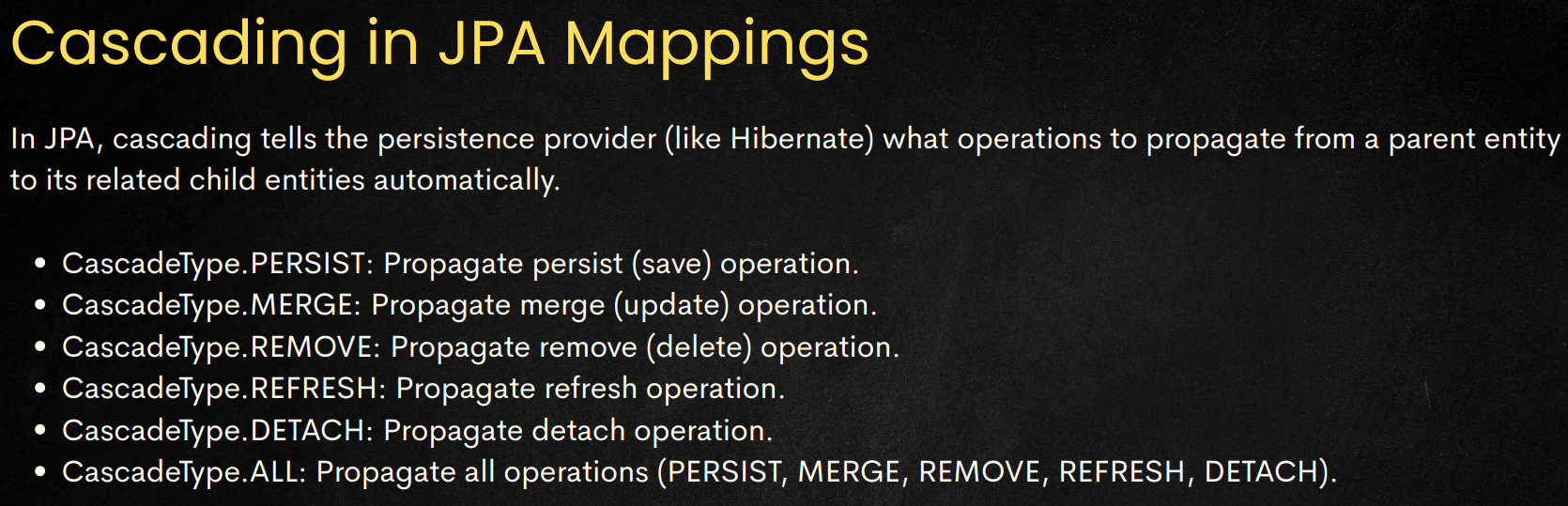
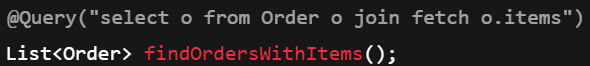


* **PatientEntity p = new PatientEntity()**
  + This will take the state to **Transient**.
  + This is only used for **creating new entry**.
  + So, basically only **create** is related to **transient state**, other operation **(update, delete, read)** doesn’t include **transient state** at all.
  + In **transient state**, it is only a pure Java object; nothing more.
  + **Transient state is not managed by EntityManager; because it is pure Java object**.
* Either you create one object just like above and call **repository.save(p)** or you are making the **get** call like **repository.findById(**id**)** to get the **entity**, it’ll be remaining in the **Persistence Context** only.
  + **Persistence Context** is nothing but a **run-time** state that holds all the **changes** being made, or **data** being fetched till now to **reduce the number of database calls**.
  + For read, it’ll check in the **Persistence Context** , if the entry is present then it’ll directly return; otherwise **query the db, fetch the data, store the data in Persistence Context, return the data**.
  + Once all the **requests are being processed** (it may be update, or delete, or create; because **read happens directly only**), when the **transaction is coming to the end**, it **commits all the changes to the database at once**. Hence reducing the number of **db calls**.
  + **persist()** puts an object into the Persistence Context **WITHOUT reading** from DB
  + **find()** puts an object into the Persistence Context **BY reading** from DB (**if needed**)
* Simple thing: if the operation is of type **create, update, delete**, then it’ll keep all the operations within it; and when the **transaction ends** it trigger those queries to **db**.
* If the **entity** is made **Detached** and left hanging there in the **Transactional method;** and after that method ends if the **reference of the entity** is destroyed; then the **entity will be garbage collected**.
* Same for **remove** and **transient(create)** as well;
* Means its normal Java concept; if the **variables** that were holding the reference of the **entity object** are destroyed; then the **entity object** will be garbage collected; it has nothing to do with the **transient** or **persistence** or **detach** state.
* **IMPORTANT:** **@Transactional** is required to make proper use of **Persistence Context**. If **@Transactional** is not mentioned, then there is no guarantee that each method will run in same **Persistence contexts**.
* **Some Points to Remember** 
  + Repository object will be having EntityManager’s methods.
  + **JpaRepository** will be implemented by **SimpleJpaRepository** class.
  + **SimpleJpaRepository** class will *call the methods* of **EntityManager**.
    - (Hibernate implements the EntityManager interface)
  + So, we call the methods like
    - **repository.save(entityObject)**
    - So, under the hood it is called like
    - **entityManagerObject.persist(entityObject)**
  + **persist** for *create*, **merge** for *update*, **delete** for *remove*.
  + 
    - It’s the SimpleJpaRepository class’s save method.
    - If it is **isNew** then **persist** else **merge** for update.
* **Garbage Collection**
  + Garbage Collection is **purely a Java concept**
  + **objects are created inside heap**
  + As variables are **created on stack** so after the execution of their respective scope, those variables will be destroyed.
  + So, when all the **variables holding reference to an object** are **destroyed** or **cleared**, and there is **no variable present which is containing the reference of that object** so that **JVM can reach that object**, then **the object will be eligible for garbage collection**.
  + JVM can not reach the object means: JVM is not able to find any variable(normal or static or anything) that contains the reference of the Object. So, the **reference of the object** is lost now. So In that case JVM send the Object for garbage collection.

**Relationship Owning side and Inverse Side**

* **In every bidirectional JPA relationship, exactly ONE side must be the owner.That owner is the side that contains the foreign key in the database.**
* **How to know which is OWNING and which is INVERSE?**
  + Just find out where should the **foreign key** can be stored?
  + Normally the decision making will be like:
    - For 2 entities **entity-1** and **entity-2**
    - If **entity-1** can exist without **entity-2** then **entity-2** should contain the **foreign key** (i.e. **primary** key of **entity-1**); so **entity-1** is **inverse** and **entity-2** is **owning**.
      * One catch will be here; lets say 2 entities are there **user** and **profile**.
      * **user** can exist without **profile** but vice-versa is not true.
      * So, **profile** should contain the FK.
      * But, its upon USER’s decision that he’ll create his profile or not so in this case PROFILE can be **null** as well.
      * So, in this case it is better to keep FK in **user** table instead of **profile**.
* **One-To-One** mapping in JPA
  + Lets say **USER** and **PASSPORT**.
  + PASSPORT is the *owning* and USER is the *inverse*.
  + **inverse** side will contain **mappedBy** in JPA.
  + **@OneToOne**
    - It’ll tell Hibernate that I need one **foreign key** column in my table.
    - If you pass **mappedBy** then it’ll think that “okay, my other half has already kept me; I don’t need to keep that”
    - without **mappedBy** hibernate thinks this is the owner. If you don’t write **mappedBy** in neither of 2 entities (User and Passport) then it’ll create foreign key column in both.
    - **mappedBy** will contain the **variable name (of object; not database column)** which is created in the **owning side** as the value.
      * 
        + Here owning type has variable name: **user**
      * 
        + So, **user** has to be given as the value of **mappedBy** in inverse.
  + **@JoinColumn**
    - it is just used to provide name of foreign key column; if you don’t give this annotation then by default the column name will be like the below:
      * Let inverse table is **Person** and **id** field is **myPmId**
      * The column name would be: **person\_my\_pm\_id**. (if @Column doesn’t contain any different column name; else that column name will be created)
    - **@JoinColumn(“person\_id”)** it means nothing but “store the primary key of inverse table in my table as foreign key under the column having name **person\_id**”
    - Means the **foreign key** column will be named as **person\_id**.
    - In case of **person\_id** I can even right **@JoinColumn(“**lalala\_lululu”**)**
* **One-to-Many** or **Many-to-One** mapping in JPA
  + **Many side will always be the owning type; hence foreign key will be there in the Many side only**.
  + **Many side**:
    - **@ManyToOne**
    - **@JoinColumn** (if you want a personalize column name)
  + **One side**:
    - **@OneToMany(mappedBy)**
* **Many-to-Many** mapping in JPA
  + This is slightly different than the previous mappings.
  + Here **No FK in either of tables**.
  + **Relationship will be stored in *another* table**.
  + **Any table can be *owning* and other will be *inverse*** (its completely up-to **you**)
  + (Student, Course)
  + I chose **Student** as the ***owning***type.
  + **(owning)**
    - Here **@JoinTable** is used; inside which **@JoinColumn** is used.
      * Again, here also **@JoinColumn** is not mandatory.
  + **(inverse)**
  +  

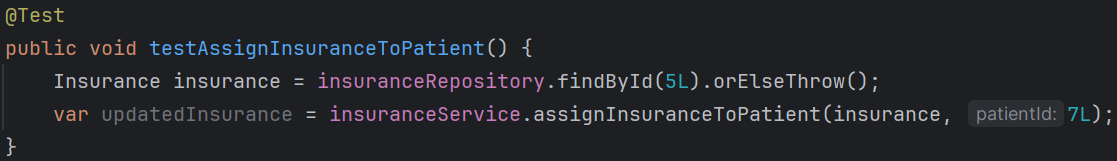
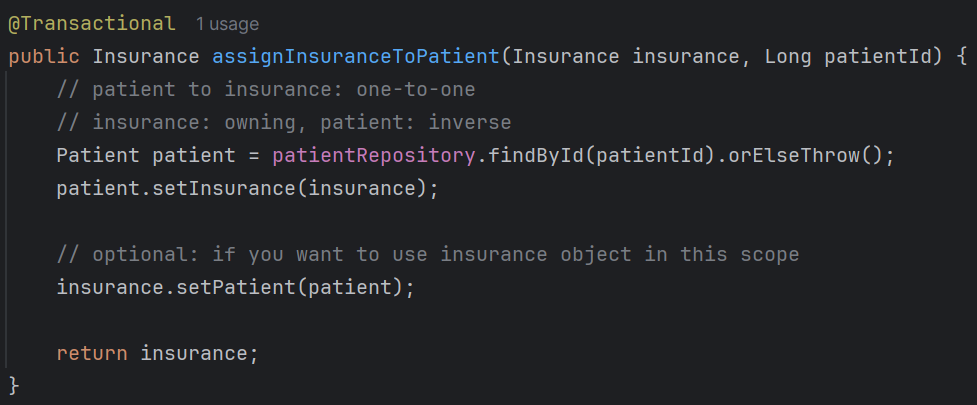
**Cascading**

* It tells persistence provider (Hibernate) what operations to propagate from a **Parent entity to Child entity**.
* NOTE: **owning** and **inverse** are no way related to **parent entity** and **child entity** concept.
* The **orphanRemoval** and **cascade** are written in **Entity** level. If you call the method like **deleteById** it is *not* *guaranteed* that those **cascade** will be used.
  + You need to call the method **delete()** which take an **entity object** as the argument.
* **cascade** vs **orphanRemoval** 
  + **cascade** means if some operation is being executed on **parent** entity (**create, update, delete**) then the same operation will be executed on **child** entities.
  + **orphanRemoval** means if the **reference is gone**, then delete the object that was being referenced (just like GC in Java)
    - 
    - If I set **patient=null** for the **insurance entity object**, and if there is no **entity that has reference of that particular patient entity object**, then that entity will be deleted.
    - In simple terms, when an **entity becomes orphan** (no parents are there), then it’ll be deleted.
    - Its independent of **owning/inverse** concept.
  + Simple analogy:
    - **cascade :** if I am being punished, all my children will also be punished.
    - **orphanRemoval :** I don’t want my children now; if no one has made them their children then remove them.
* **NOTE** 
  + In case of **cascade** or **orphanRemoval**, even the **repository method** will not be able to manage the **transaction** properly.
  + So, its better to use **@Transactional** in the method when doing **CUD** (create, update, delete) operations.
  + If the parent function contains **@Transactional**, and the child function (being called in the parent function) is also having **@Transactional**, then also it won’t give error. It is good practice to use @Transactional
    - 
    - Its safe.
* 
  + Here this service method is assigning one **insurance** to the **patient**.
  + **@Transactional** is required to make these whole queries run in a single **persistence context**.
  + If you remember, without **@Transactional**, all the method may not run in same **persistence context**.
  + 
    - It is the test method; but it’ll fail because we have not made the **patient entity** as **cascade** yet. So, even if the **patient** is created, it’ll not create the **insurance**.
    -  NOW IT WILL PASS
    - (or)
* There are 2 sides: **owning** & **inverse**.
  + As we know **owning** side will contain the foreign key.
  + So, if you try to delete the **inverse** entity then it’ll not happen; because it is being referenced in another table (**owning** table)
  + **You cannot delete the inverse side of a relationship while the owning side still points to it.**
* Some points to be remembered:
  + **orphanRemoval** on *inverse*side is of no use;
    - Because, **inverse** side doesn’t have a solid *foreign key* column;
    - If just keep one reference of the **owning** entity to fetch the data and store there;
  + **cascade** doesn’t care about **owning/inverse**. It’ll execute all the operation, that is being executed to the **parent entity,** to **every child entities whose references are there in current entity**.
* **orphanRemoval** example
  + 
    - Here I am **updating** the insurance.
    - So, **previous insurance object became orphan**, hence it’ll be deleted.
    - **new insurance object will be created, saved; then it’ll be added to patient**.
  + 
    - Same here as well; in stead of changing the insurance, here insurance is being removed;
* 
* 
* **N+1 Query Optimization** 
  + For **one to many** relations, the entity having **one-side** will be having a **List** of the entities of **many-side**.
  + In this case, by default the fetching behaviour is **Lazy**.
  + If you want to get all the things within the same object directly then you can use:
    - **fetch = FetchType.EAGER** (eager means it’ll directly call **join query** and give you one object having all the objects of the referenced type(s))
  + When you make this **FetchType.EAGER**, you can see there will be **N+1** queries; which is not optimized behaviour;
* **FetchType**
  + 2 types are there: **EAGER, LAZY**
    - In case of EAGER, child will be loaded immediately when parent is fetched.
    - In case of LAZY, child will only be loaded when we do **get** call from parent to get child data.
  + But sometimes we want both;
    - Like some method call may need full data i.e. parent and child in a single call.
    - And some method may not need full data; in this case making the Entity as **EAGER** will be very expensive; unnecessay a lot of data will be fetched from DB.
  + So, in this case we make the Entity **LAZY** and we’ll do something in method call to control the type of fetching.
    - One thing can be done:
      * Fetch the parent
      * Use a loop or something like that to get all the child objects;
      * But, it’ll result in **N+1** SQL queries; **1** for fetching *parent*, **N** for fetching *children* (considering N childs are there)
    - Instead of handling the FetchType from the Entity level, we’ll do this from **method level**.
    - So, when the method needs full data EAGER type call will happen; otherwise LAZY by default;
  + 2 methods of controlling the fetching type from methods:
    - **FETCH** join JPQL query
      * 
    - **@EntityGraph** annotation
      * 
      * “**items**” is object field name; not column name;

**Some Experiments**

* Lets say 3 entities are there **E1, E2, E3;** some objects are **e1, e2, e3**.
  + **E1 to E3** : One-to-One mapping
  + **E2 to E3** : One-to-One mapping
  + **orphanRemoval** is **true** for **E3** field in both **E1, E2**.
  + Lets say I did: **e1.setE3(null)**
    - Just removing the reference of **e3** from **e1**.
  + But, **e1**’s reference is still there in **e2**.
  + But, in this case Hibernate will try to delete **e3** from DB because in **e1**, its reference is gone.
    - Then DB will give error saying its reference is still there in **E2** table.
  + So, its design failure; The design of Database is wrong here;
* Let say 2 entities are there **E1, E2**.  **------ this is wrong; not sure why it happens -----**
  + **E1 to E2** : One-to-One
  + **orphanRemoval** is **true** for the ***E2*** *field* in ***E1*** entity.
  + No **cascade** is there.
  + Lets say I delete **e1** (object of type E1) which had reference of **e2** (object of type E2).
    - Even there is no **cascade** there, but because of deletion of **e1** the reference of **e2** is gone; hence it’ll delete **e2** as well.
* Ff

**Some Error Causes**

* ERROR-1
  + Consider the following codes:
    - 
    - 
  + In this case you’ll get an error like: “**detached entity passed to persist**”
  + In the first method i.e. *testAssignInsuranceToPatient*,
    - you are using **findById** to get the insurance.
    - So during this method call:
      * Spring opens a short lived EntityManager
      * Loads **Insurance**
      * Returns it
      * ***Closes the EntityManager***
    - Now, as the EntityManager is closed, so now **insurance** is in ***detached*** state now, not in ***managed*** (persistent context) state.
    - Now you are sending this ***detached*** object to next method i.e. *assignInsuranceToPatient* which is passing this object to *setInsurance***.**
    - So, in here, *setInsurance* expects one ***managed state*** object; but its gettnig ***detached state*** object;
    - Hence, it is giving that error.
  + Fix:
    - write **@Transactional** in the **@Test** method; Now, both the method will run in same EntityManager till the transaction is completed.
    - If you are writing **@Transactional** in **@Test** method, then no need to write **@Transactional** in the *assignInsuranceToPatient* method. Only the **outer method needs to have @Transactional**.
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F
* F