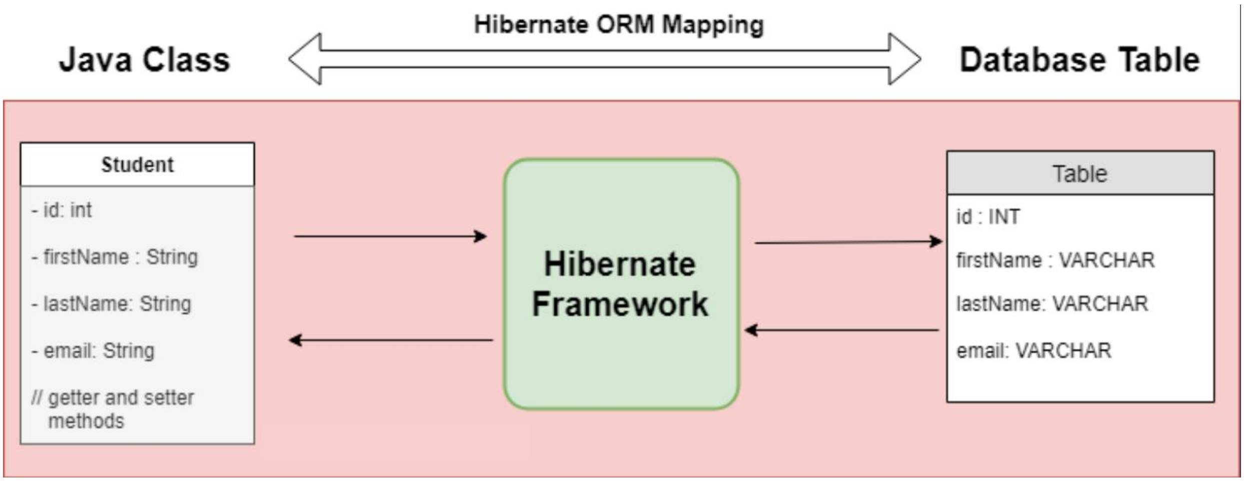
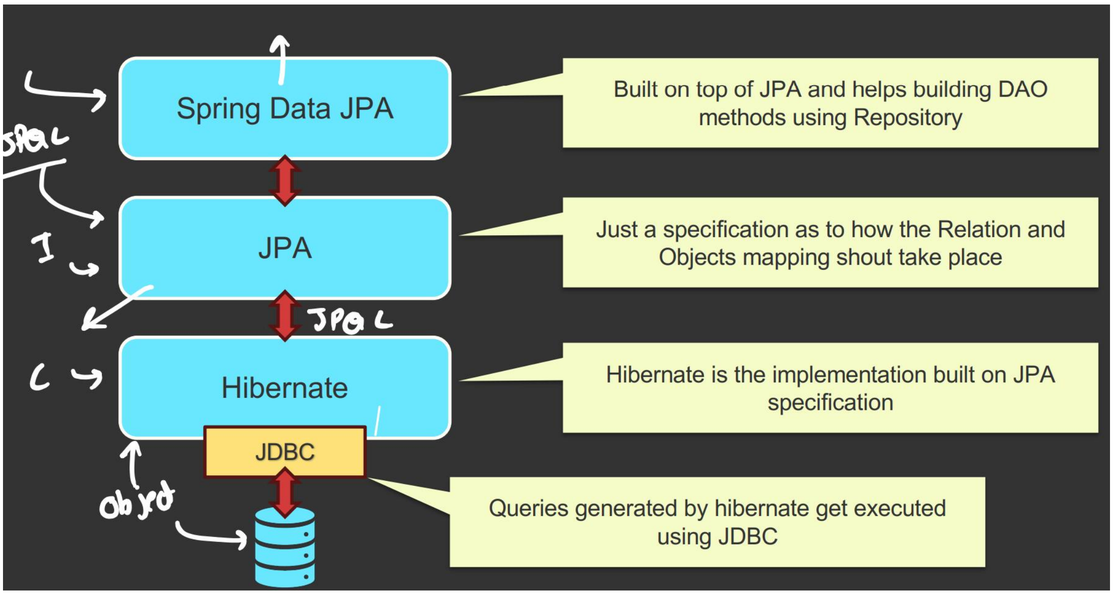
Hibernate ORM Mapping





Built on top of JPA and helps building DAO methods using Repository

Just a specification as to how the Relation and Objects mapping shout take place

Hibernate is the implementation built on JPA specification

Queries generated by hibernate get executed using JDBC

- 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

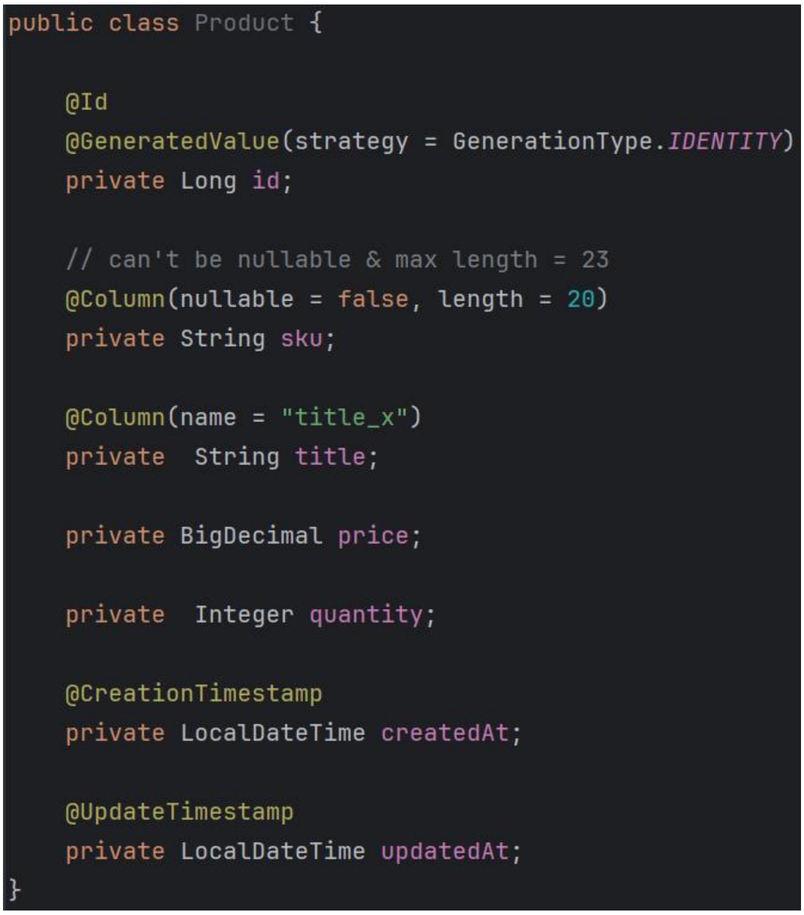
- First Layer is **SPRING DATA JPA**
  - Built by Spring on top of **JPA**.
  - Contains:
    - **JpaRepository interface** (extended by *user-defined repository interfaces*).
    - **SimpleJpaRepository class** (contains *method bodies of JpaRepository*).
  - SimpleJpaRepository already has implementations 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.

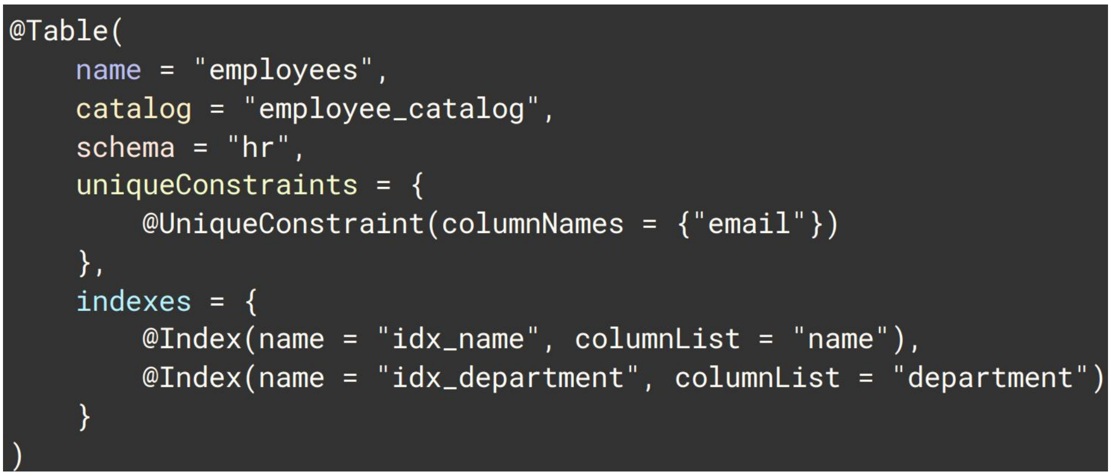
- There are multiple annotations for **Entity** objects

```java
public class Product {

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

    // can't be nullable & max length = 23
    @Column(nullable = false, length = 20)
    private String sku;

    @Column(name = "title_x")
    private  String title;

    private BigDecimal price;

    private  Integer quantity;

    @CreationTimestamp
    private LocalDateTime createdAt;

    @UpdateTimestamp
    private LocalDateTime updatedAt;
}
```

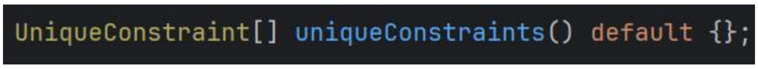
  - **@Id**, **@GeneratedValue**, **@Column** (change name, nullable true or false, length if it's a string, etc etc), **@CreationTimeStamp**, **@UpdateTimeStamp** …etc

- **@Table** annotation

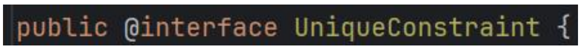
  

```java
@Table(
    name = "employees",
    catalog = "employee_catalog",
    schema = "hr",
    uniqueConstraints = {
        @UniqueConstraint(columnNames = {"email"})
    },
    indexes = {
        @Index(name = "idx_name", columnList = "name"),
        @Index(name = "idx_department", columnList = "department")
    }
)
```

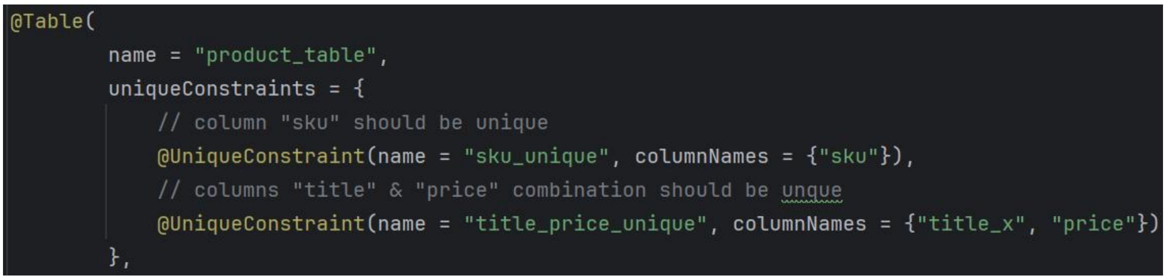
  - There is something called **namespace** in database.
    - **auth.**user, **sales.**user
    - Here both have the same table name "user", but they do not conflict because they belong to different namespaces (auth, sales).

    - 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.
  - 

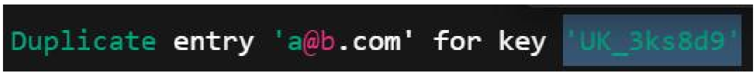
```
UniqueConstraint[] uniqueConstraints() default {};
```

    - UniqueConstraint is also an annotation :).
    - 

```
public @interface UniqueConstraint {
```

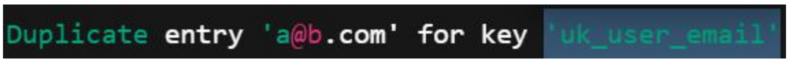
    - 

```
@Table(
        name = "product_table",
        uniqueConstraints = {
            // column "sku" should be unique
            @UniqueConstraint(name = "sku_unique", columnNames = {"sku"}),
            // columns "title" & "price" combination should be unque
            @UniqueConstraint(name = "title_price_unique", columnNames = {"title_x", "price"})
        },
```

    - (**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.
    - 

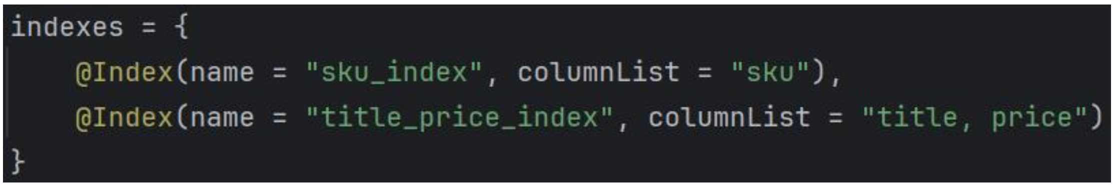
```
Duplicate entry 'a@b.com' for key 'UK_3ks8d9'
```

(without name)

    - 

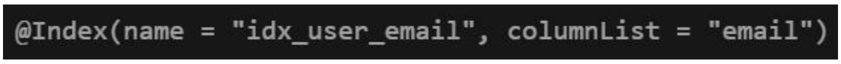
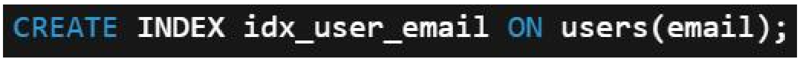
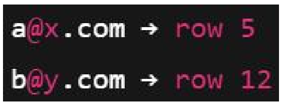
```
Duplicate entry 'a@b.com' for key 'uk_user_email'
```

(with name)

  - **indexes**
    - Here the **columnList** is a *String* not a *List*.
    - You should give comma separated column names.
    - 

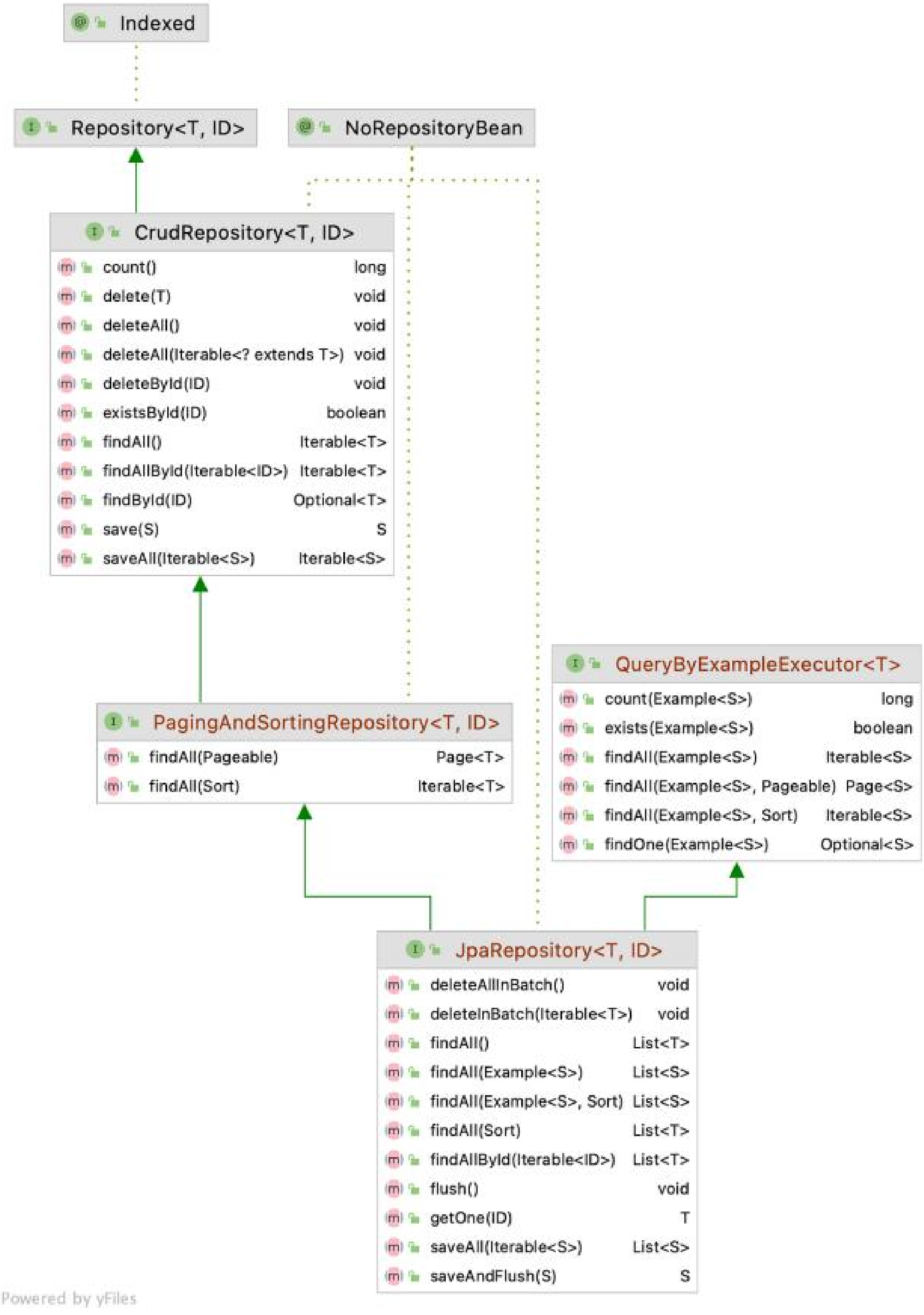
```
indexes = {
    @Index(name = "sku_index", columnList = "sku"),
    @Index(name = "title_price_index", columnList = "title, price")
}
```

- NOTE: **database** should already be present. It'll not create the database inside the server by itself.

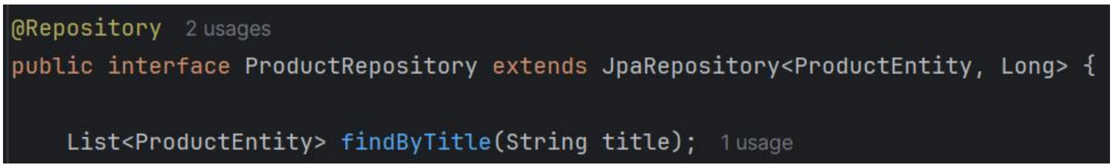
- **Indexing** in database.
  - 
    ```
    @Index(name = "idx_user_email", columnList = "email")
    ```
    (JPA)
  - 
    ```
    CREATE INDEX idx_user_email ON users(email);
    ```
    (SQL)
  - An index is a *separate data structure* that stores indexed **column** values along with **row pointers**.
  - It is not a normal table; it is created and managed internally by the database..
    - 
      ```
      a@x.com → row 5
      b@y.com → row 12
      ```
  - 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

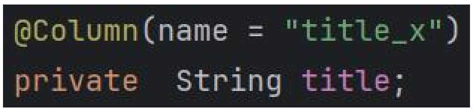
- It is a part of the larger Spring Data Family.
- It builds on top of JPA, providing a higher-level and more convenient abstraction for data access.
- Spring data JPA makes it easier to implement JPA-based repositories by providing boilerplate code, custom query methods, and various utilities to reduce the amount of code you need to write.


Indexed
Repository<T, ID>
NoRepositoryBean
CrudRepository<T, ID>
count() long
delete(T) void
deleteAll() void
deleteAll(Iterable<? extends T>) void
deleteById(ID) void
existsById(ID) boolean
findAll() Iterable<T>
findAllById(Iterable<ID>) Iterable<T>
findById(ID) Optional<T>
save(S) S
saveAll(Iterable<S>) Iterable<S>
QueryByExampleExecutor<T>
count(Example<S>) long
exists(Example<S>) boolean
findAll(Example<S>) Iterable<S>
findAll(Example<S>, Pageable) Page<S>
findAll(Example<S>, Sort) Iterable<S>
findOne(Example<S>) Optional<S>
PagingAndSortingRepository<T, ID>
findAll(Pageable) Page<T>
findAll(Sort) Iterable<T>
JpaRepository<T, ID>
deleteAllInBatch() void
deleteInBatch(Iterable<T>) void
findAll() List<T>
findAll(Example<S>) List<S>
findAll(Example<S>, Sort) List<S>
findAll(Sort) List<T>
findAllById(Iterable<ID>) List<T>
flush() void
getOne(ID) T
saveAll(Iterable<S>) List<S>
saveAndFlush(S) S
Powered by yFiles

-

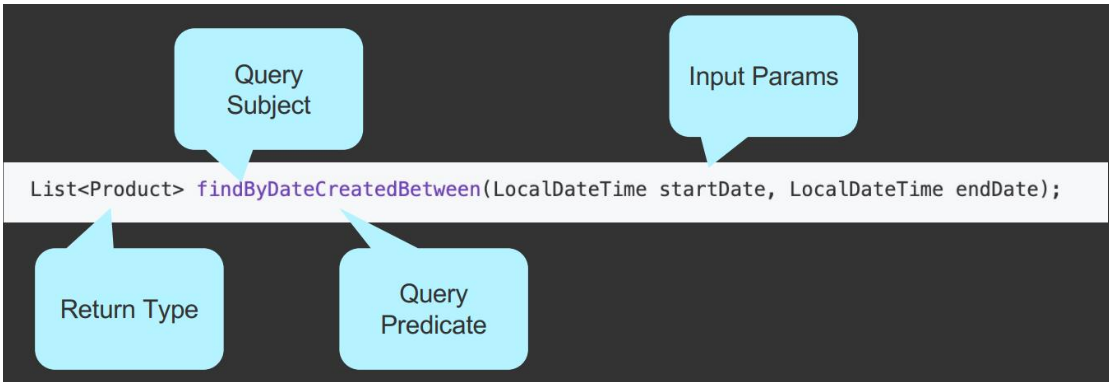
- **SimpleJpaRepository** ***class*** implements the *JpaRepository interface*. It contains implementation of all the methods of the JpaRepository and its parent interfaces.
- Key Features of Spring Data JPA
  - Repository Abstraction:
    - Provides a *Repository* interface with methods for common data 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.
  - 

```
@Repository  2 usages
public interface ProductRepository extends JpaRepository<ProductEntity, Long> {

    List<ProductEntity> findByTitle(String title);  1 usage
```

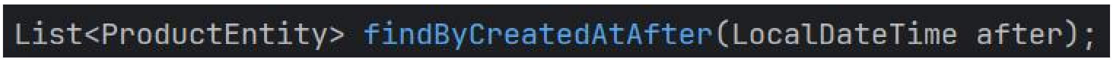
  - Just like this just write the method.
  - NOTE: If you remember the column name is **title_x** not **title**.
    - 

```
@Column(name = "title_x")
private  String 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**
  - 

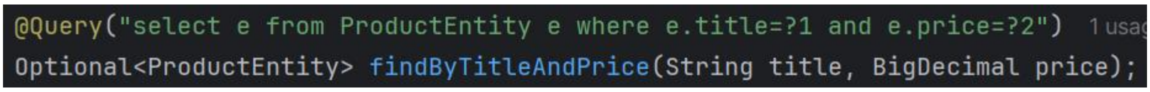
    ```
    List<Product> findByDateCreatedBetween(LocalDateTime startDate, LocalDateTime endDate);
    ```

    - 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**
- A few examples:
  - 

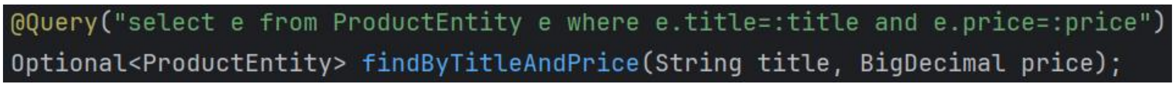
    ```
    List<ProductEntity> findByCreatedAtAfter(LocalDateTime after);
    ```

    - 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

  - 

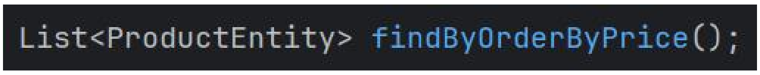
    ```
    @Query("select e from ProductEntity e where e.title=?1 and e.price=?2")  1 usag
    Optional<ProductEntity> findByTitleAndPrice(String title, BigDecimal price);
    ```

  - **@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**
    - 

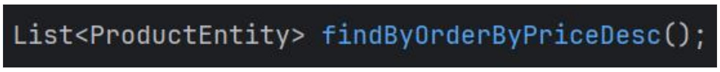
      ```
      @Query("select e from ProductEntity e where e.title=:title and e.price=:price")
      Optional<ProductEntity> findByTitleAndPrice(String title, BigDecimal price);
      ```

  - **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.

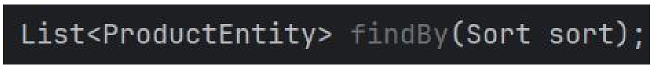
Sorting & Pagination

- **OrderBy** is used to *Sort*.
  - 

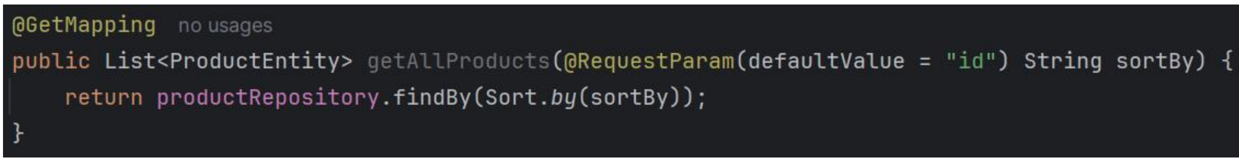
    ```
    List<ProductEntity> findByOrderByPrice();
    ```

  - Instead of **findAll**, here we need to write **findBy**.
  - It'll get all the rows and sort them according to the *price*. **OrderByPrice**.
  - 

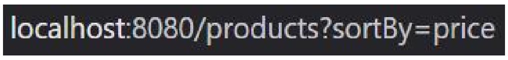
    ```
    List<ProductEntity> findByOrderByPriceDesc();
    ```

  - 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.
  - 

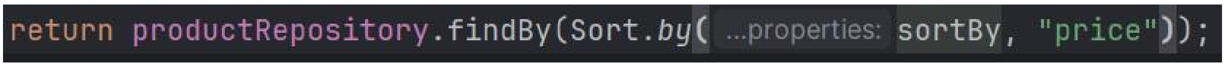
    ```
    List<ProductEntity> findBy(Sort sort);
    ```

    - Here we need to get the argument of type **Sort**.
  - 

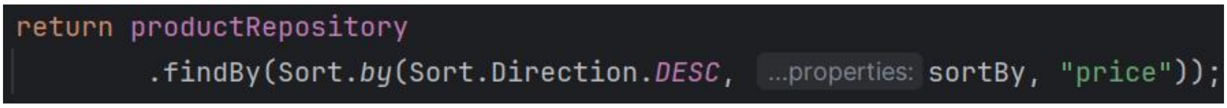
    ```
    @GetMapping  no usages
    public List<ProductEntity> getAllProducts(@RequestParam(defaultValue = "id") String sortBy) {
        return productRepository.findBy(Sort.by(sortBy));
    }
    ```

    - 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.
    - 

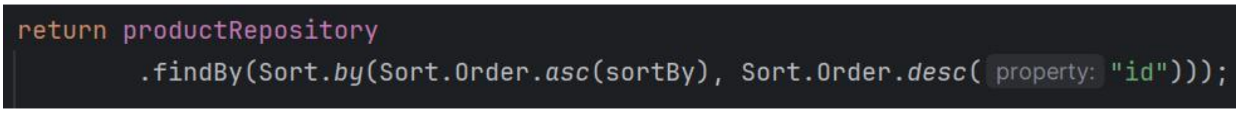
      ```
      localhost:8080/products?sortBy=price
      ```

      like this
  - 

    ```
    return productRepository.findBy(Sort.by( ...properties: sortBy, "price"));
    ```

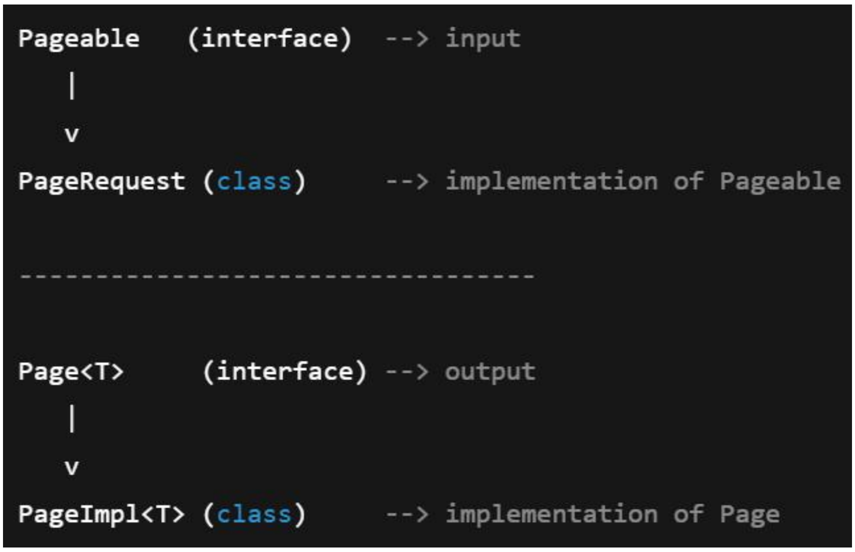
    - If the **sortBy** column is same, then it'll further sort according to "price". like this we can give multiple properties.
  - 

    ```
    return productRepository
            .findBy(Sort.by(Sort.Direction.DESC, ...properties: sortBy, "price"));
    ```

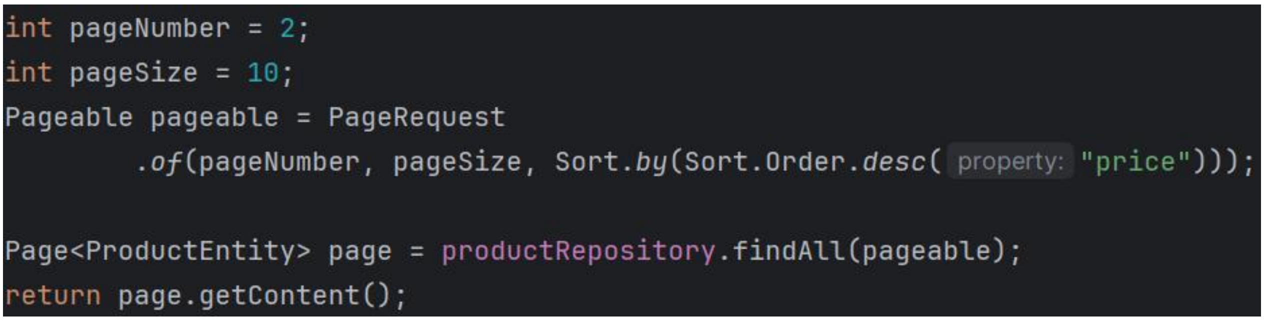
    - You can give the direction like this as well.
  - 

    ```
    return productRepository
            .findBy(Sort.by(Sort.Order.asc(sortBy), Sort.Order.desc( property: "id")));
    ```

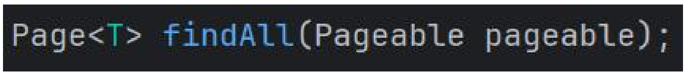
    - If you want different direction i.e. ASC, DESC for different columns then use like this.

- 

```
Pageable    (interface)  --> input
   |
   v
PageRequest (class)      --> implementation of Pageable

--------------------------------

Page<T>      (interface) --> output
   |
   v
PageImpl<T> (class)      --> implementation of Page
```

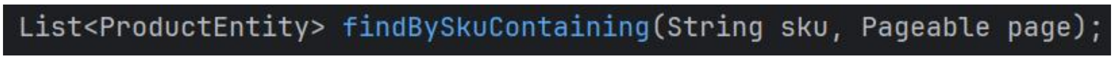
  - **Pageable** interface represents: **page size, page number, sorting info**
  - **PageRequest** is used to create **Pageable** object.
  - The query after being run returns **Page** object.
    - Typically we never create object of **Page**.
- **Pageable** is the interface; **PageRequest** is the class 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)
- 

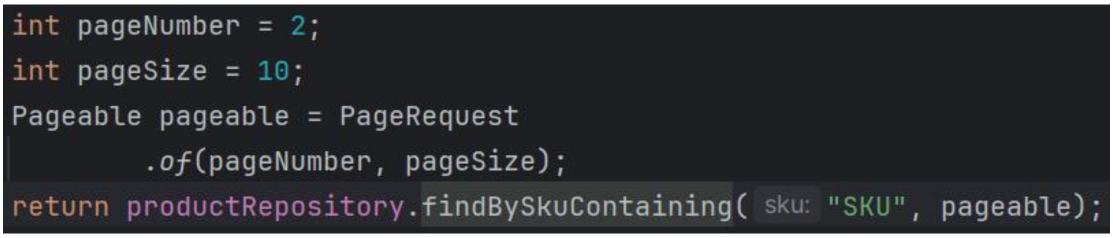
```
int pageNumber = 2;
int pageSize = 10;
Pageable pageable = PageRequest
        .of(pageNumber, pageSize, Sort.by(Sort.Order.desc( property: "price")));

Page<ProductEntity> page = productRepository.findAll(pageable);
return page.getContent();
```

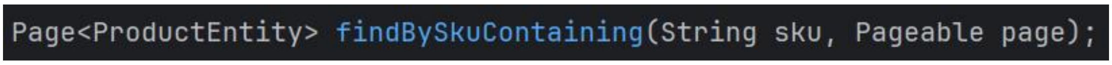
  - **findAll** method is already there.
  - 

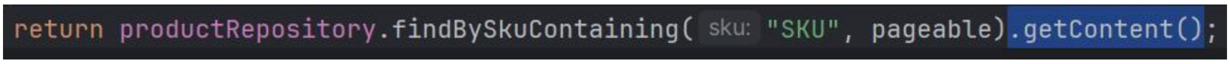
```
Page<T> findAll(Pageable pageable);
```

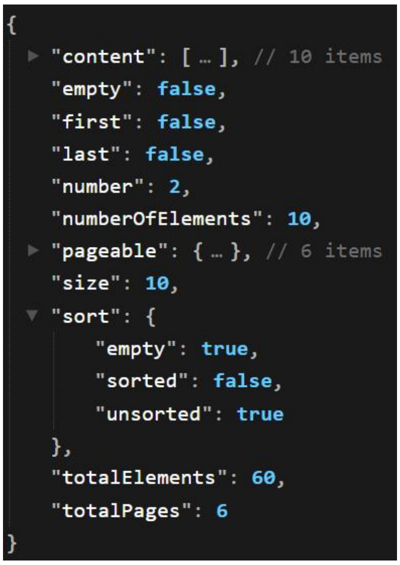
  - It returns an object of type **Page**.
  - **getContent()** method is used to extract the **List of Entity** from the **Page**.

- **NOTE** ---
- The return type of the query doesn't only depends upon **Pageable** parameter. It depends on the return type of the method.
  - 

    ```
    List<ProductEntity> findBySkuContaining(String sku, Pageable page);
    ```
  - I wrote this method, and the return type is **List<ProductEntity>** so here **List** will be returned.
  - 

    ```
    int pageNumber = 2;
    int pageSize = 10;
    Pageable pageable = PageRequest
            .of(pageNumber, pageSize);
    return productRepository.findBySkuContaining( sku: "SKU", pageable);
    ```
  - But if I write **Page<ProductEntity>** in that query then it'd have return Page type of object.
  - 

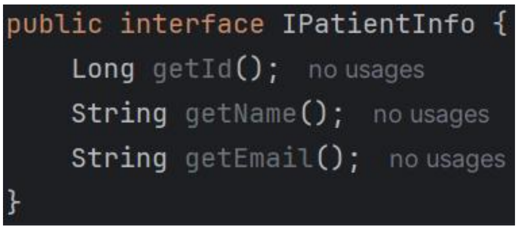
    ```
    Page<ProductEntity> findBySkuContaining(String sku, Pageable page);
    ```
  - 

    ```
    return productRepository.findBySkuContaining( sku: "SKU", pageable).getContent();
    ```
    - 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**.
- 

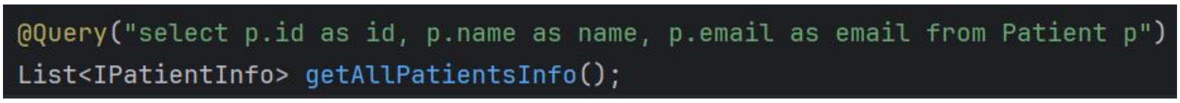
  ```
  {
  ▶ "content": [ … ], // 10 items
    "empty": false,
    "first": false,
    "last": false,
    "number": 2,
    "numberOfElements": 10,
  ▶ "pageable": { … }, // 6 items
    "size": 10,
  ▼ "sort": {
        "empty": true,
        "sorted": false,
        "unsorted": true
    },
    "totalElements": 60,
    "totalPages": 6
  }
  ```

  (like this)

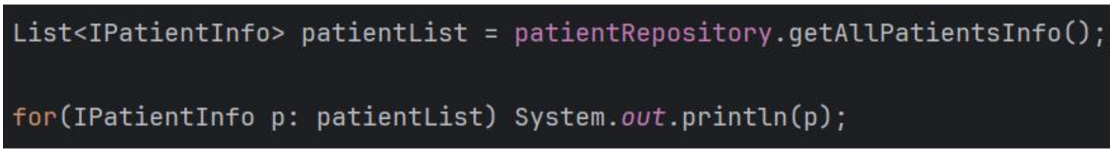
Projection in Spring Data JPA

- Lets say the the requirement is of some specific columns of the tables not the whole table.
  - In this case, we can get the Entity in the service file, then create one DTO containing required fields, and then return that DTO as response.
  - But in this case, We are fetching whole Table from the DB then we are filtering the columns. We don't want this.
- **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.**
  - 

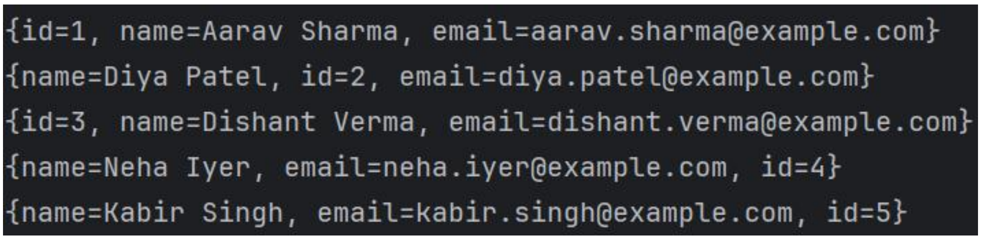
```
public interface IPatientInfo {
    Long getId();  no usages
    String getName();  no usages
    String getEmail();  no usages
}
```

  - But there is no **variables** here so modification is not possible.
  - 

```
@Query("select p.id as id, p.name as name, p.email as email from Patient p")
List<IPatientInfo> getAllPatientsInfo();
```

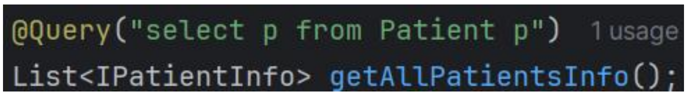
  - 

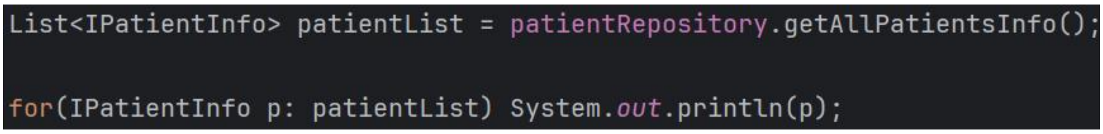
```
List<IPatientInfo> patientList = patientRepository.getAllPatientsInfo();

for(IPatientInfo p: patientList) System.out.println(p);
```

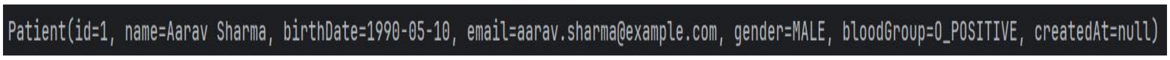
  - 

```
{id=1, name=Aarav Sharma, email=aarav.sharma@example.com}
{name=Diya Patel, id=2, email=diya.patel@example.com}
{id=3, name=Dishant Verma, email=dishant.verma@example.com}
{name=Neha Iyer, email=neha.iyer@example.com, id=4}
{name=Kabir Singh, email=kabir.singh@example.com, id=5}
```

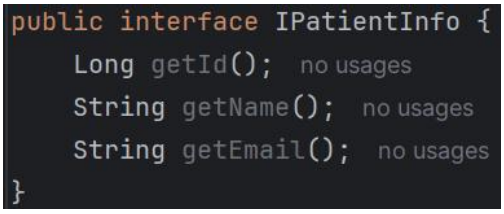
  - It'll work.
  - In the above query i.e. **"select p.id as id, p.name as name, p.email as email from Patient p"** the aliases are important i.e. **id**, **name**, **email**,
    - **id** will be mapped to **getId**
    - **name** will be mapped to **getName** ..etc

- 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.
    - 

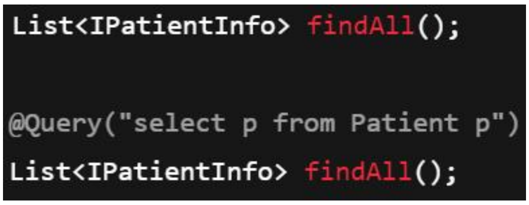
      ```
      @Query("select p from Patient p")  1 usage
      List<IPatientInfo> getAllPatientsInfo();
      ```
    - 

      ```
      List<IPatientInfo> patientList = patientRepository.getAllPatientsInfo();

      for(IPatientInfo p: patientList) System.out.println(p);
      ```
    - When you execute this, You will see **entity-like output** because **toString()** is delegated to the entity. And it is basically **entity.toString()** .
      - **patientList.toString()** was called;
      - for each object of this list, List internally calls **toString()**;
      - For each object, **toString()** call was delegated to **toString()** of **entity**
    - Printed output ≠ actual object type
    - Actual object = proxy
    - 

      ```
      Patient(id=1, name=Aarav Sharma, birthDate=1990-05-10, email=aarav.sharma@example.com, gender=MALE, bloodGroup=O_POSITIVE, createdAt=null)
      ```
  - 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.

- Projection
    - 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.
- This is my **DTO Interface** (for reference of next explanations)
    - 

```
public interface IPatientInfo {
    Long getId();  no usages
    String getName();  no usages
    String getEmail();  no usages
}
```

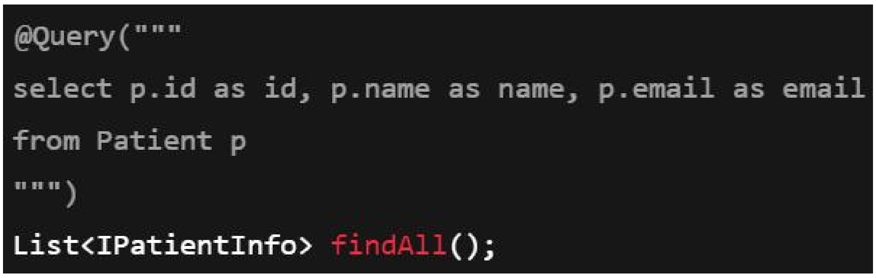
- 2 Types of projections are there:
    - **Entity-backed Projection**
    - **Tuple-backed Projection**
- **Entity-Backed Projection**
    - 

```
List<IPatientInfo> findAll();

@Query("select p from Patient p")
List<IPatientInfo> findAll();
```

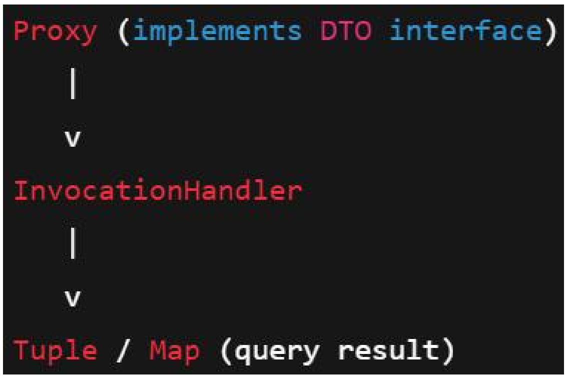
    - Both are same, if you don't give the query then it'll by default write that query only (which is being mentioned in the image)
    - So, here it is fetching the full **entity objects** which is of type **Entity**.
    - Flow will be like:
        - **Hibernate** fetches **Patient Entity** object from 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

- 

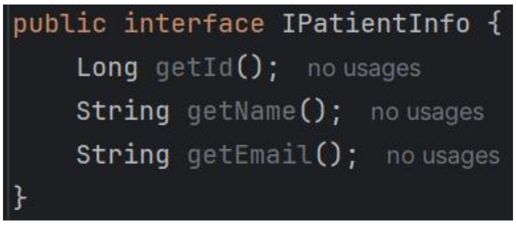
```
@Query("""
select p.id as id, p.name as name, p.email as email
from Patient p
""")
List<IPatientInfo> findAll();
```

- 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.
  - 

```
Proxy (implements DTO interface)
    |
    v
InvocationHandler
    |
    v
Tuple / Map (query result)
```

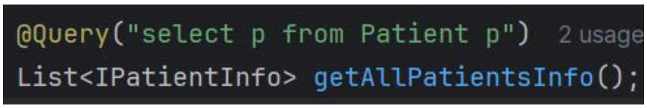
<u>Some experiments:</u>

- **Experiment-1**
  - Interface DTO (id, name, email)   (IPatientInfo)



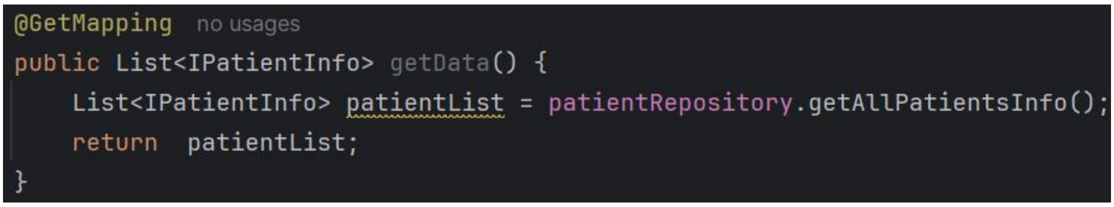
```
public interface IPatientInfo {
    Long getId();  no usages
    String getName();  no usages
    String getEmail();  no usages
}
```

  - PatientRepository (fetching the whole **entities**)



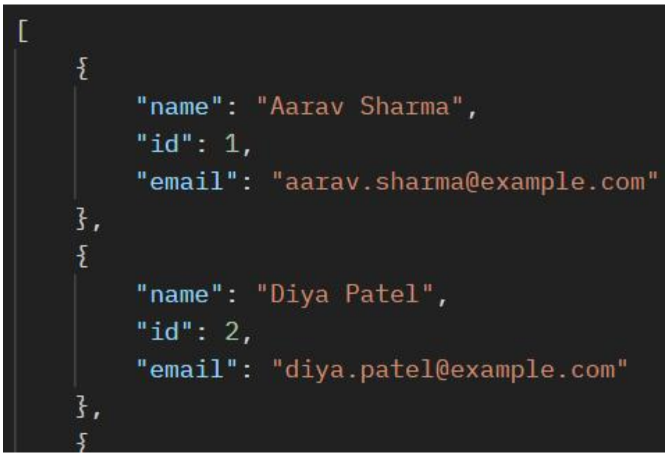
```
@Query("select p from Patient p")  2 usage
List<IPatientInfo> getAllPatientsInfo();
```

  - PatientController (returning the IPatientInfo type object)



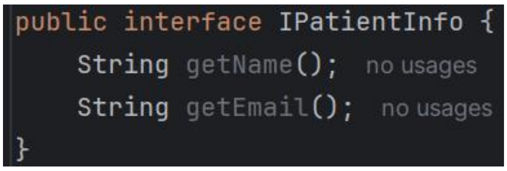
```
@GetMapping  no usages
public List<IPatientInfo> getData() {
    List<IPatientInfo> patientList = patientRepository.getAllPatientsInfo();
    return  patientList;
}
```

  - The response will be proper

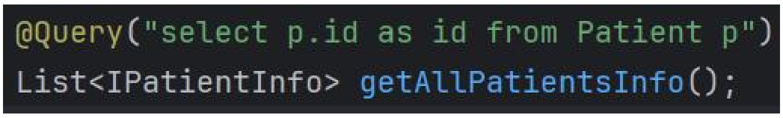


```
[
    {
        "name": "Aarav Sharma",
        "id": 1,
        "email": "aarav.sharma@example.com"
    },
    {
        "name": "Diya Patel",
        "id": 2,
        "email": "diya.patel@example.com"
    },
    {
```

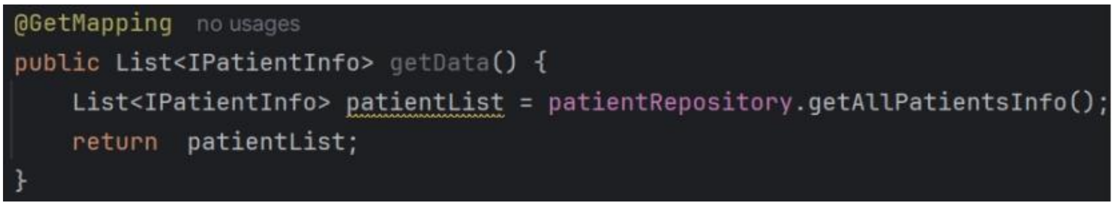
  - Reason:
    - **IPatientInfo** (Interface DTO) is having the getters for **name, id, email** which is delegated to **Patient** (Entity).
    - From the **entity objects** the **getter** methods are being called and the result is properly being generated.
- **Experiment-2**
  - Interface DTO (name, email)   (IPatientInfo)

    - 

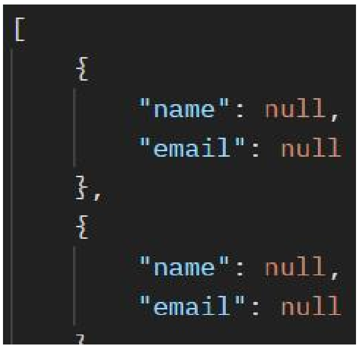
      ```java
      public interface IPatientInfo {
          String getName();  no usages
          String getEmail();  no usages
      }
      ```

  - PatientRepository (fetching the **id** of the entities)
    - 

      ```java
      @Query("select p.id as id from Patient p")
      List<IPatientInfo> getAllPatientsInfo();
      ```

    - **alias** is being used for **id** (here **getId()** will be called; otherwise it'd be null without alias)
  - PatientController (returning the IPatientInfo type object)
    - 

      ```java
      @GetMapping  no usages
      public List<IPatientInfo> getData() {
          List<IPatientInfo> patientList = patientRepository.getAllPatientsInfo();
          return  patientList;
      }
      ```

  - Now **name** and **email** will be **null**
    - 

      ```json
      [
          {
              "name": null,
              "email": null
          },
          {
              "name": null,
              "email": 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.
- F
- F
- F
- F
- F
- F
- F

- F
- F
- F
- F
- F
- F
- F
- F
- F
- F
- F
- F
- F
- F
- F
- fF

م

- F
- F
- F
- F

م

م