**Spring Data JPA**

*Nguyen Thuc Nguyen Chuong*

**I. What is ORM ?**

- ORM stands for Object-Relational Mapping. The problem is, when we work with an object-oriented system, there is a mismatch between the object model and the relational database. What if we need to modify the design of our database after having developed a few pages of our application?, etc…

Therefore, ORM was born to handle these problems.

- ORM is a programming technique for converting data between relational databases and object oriented programming languages such as Java, C#, etc…

- An ORM system has some advantages over plain JDBC:

| **No.** | **Advantages** |
| --- | --- |
| 1 | Let’s business code access objects rather than DB tables |
| 2 | Hides details of SQL queries from code logic |
| 3 | No need to deal with the database implementation |
| 4 | Entities based on business concepts rather than database structure |
| 5 | Transaction management and automatic key generation |
| 6 | Fast development of application |

- Furthermore, an ORM solution consists of the following four entities:

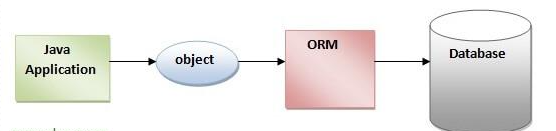
| **No.** | **Solutions** |
| --- | --- |
| 1 | An API to perform basic CRUD operations on objects of persistent classes |
| 2 | A language or API to specify queries that refer to classes and properties of classes |
| 3 | A configurable facility for specifying mapping metadata |
| 4 | A technique to interact with transactional objects to perform lazy association fetching, and other optimization functions |

- There are many persistent frameworks and ORM options in Java. A persistent framework is an ORM service that stores and retrieves objects into a relational database. We can have some: Castor, TopLink, Spring DAO, Hibernate, etc,...

**II. What is JPA & Hibernate?**

- JPA stands for Java Persistence API, is a **specification** that defines how to persist data in Java applications. It is the Java EE standard specification for ORM in Java EE.

- JPA is a high level API so that different ORM tools can implement.



- JPA provides tools that allow us to manipulate databases simply and quickly. JPA can be used to **persist** a Java object (POJO - Plain Old Java Object) into the database or get data from the database and **map** (mapping) Java objects in a simple way.

- In other way, Hibernate is a JPA provider, means it is an implementation of the Java Persistence API (JPA) specification. We can consider JPA as a collection of interfaces and Hibernate is one of the most implementations of this, to achieve ORM.

**III. JPA architecture**

Class level architecture of JPA:

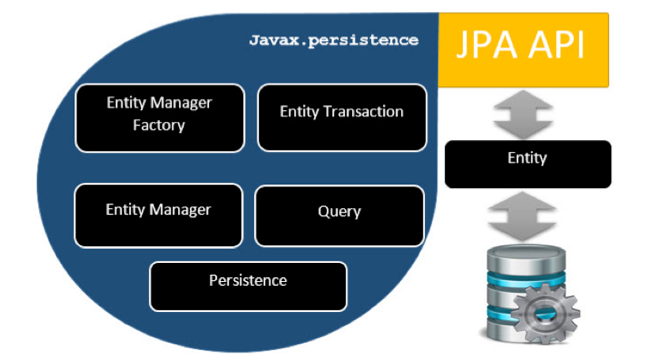
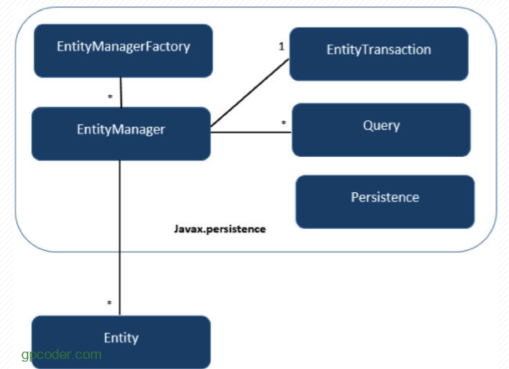


Table to describe each of unit in above picture:

| **Unit** | **Descriptions** |
| --- | --- |
| Entity | Entities are objects that represent a table in the database. Entities are usually simple POJO classes, consisting only of getter and setter methods, it can interact with database, support transaction and also inheritance like other java classes. |
| EntityManager | EntityManager is an Interface providing APIs for interacting with Entities. It works like factory for Query instance, some main functions like Persist(save new entity into db), Remove(delete an instance of Entity) |
| EntityManagerFactory | This is a factory class of EntityManager. It creates and manages multiple EntityManager instances. |
| Persistence | This is a class, contain static methods to obtain EntityManagerFactory instance |
| EntityTransaction | It has one-to-one relationship with EntityManager. For each EntityManager, operations are maintained by EntityTransaction class. Transaction consists of a set of SQL statements committed or rolled back in a unit. EntityManager helped to create an EntityTransaction. |
| Query | This is an interface, is implemented by each JPA vendor to obtain relational objects that meet the criteria |

This picture describe the relationship between the above components:

****

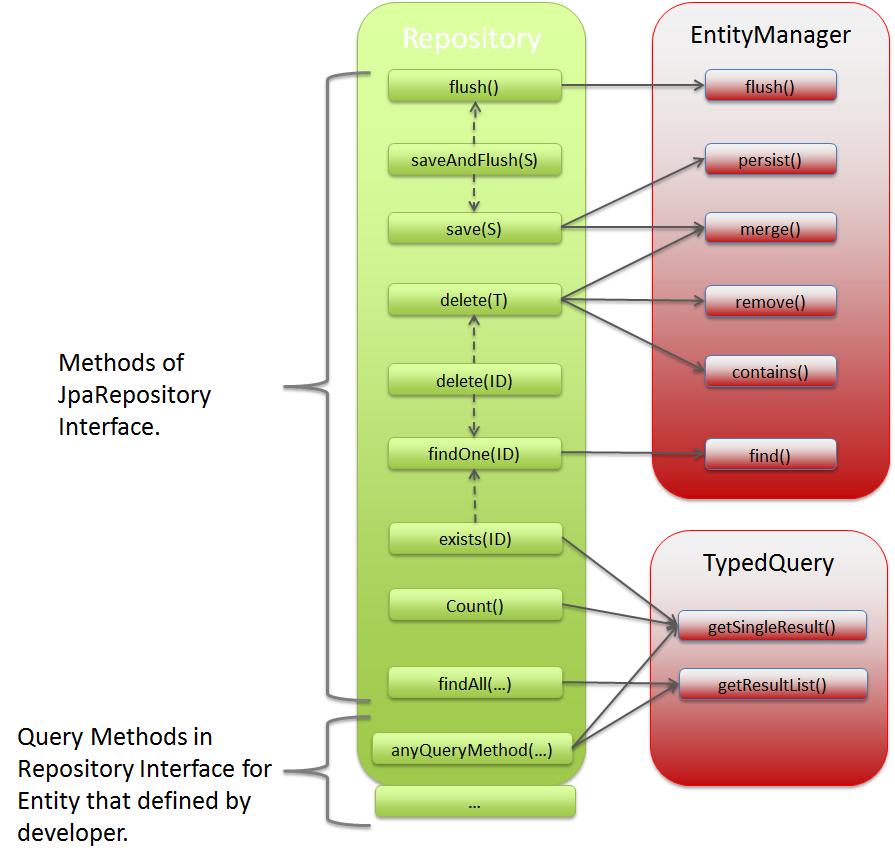
**IV. What is Spring Data JPA ?**

Spring Data is **a part of** Spring Framework. The goal of Spring Data repository abstraction is to significantly reduce the amount of boilerplate code required to implement data access layers for various persistence stores.

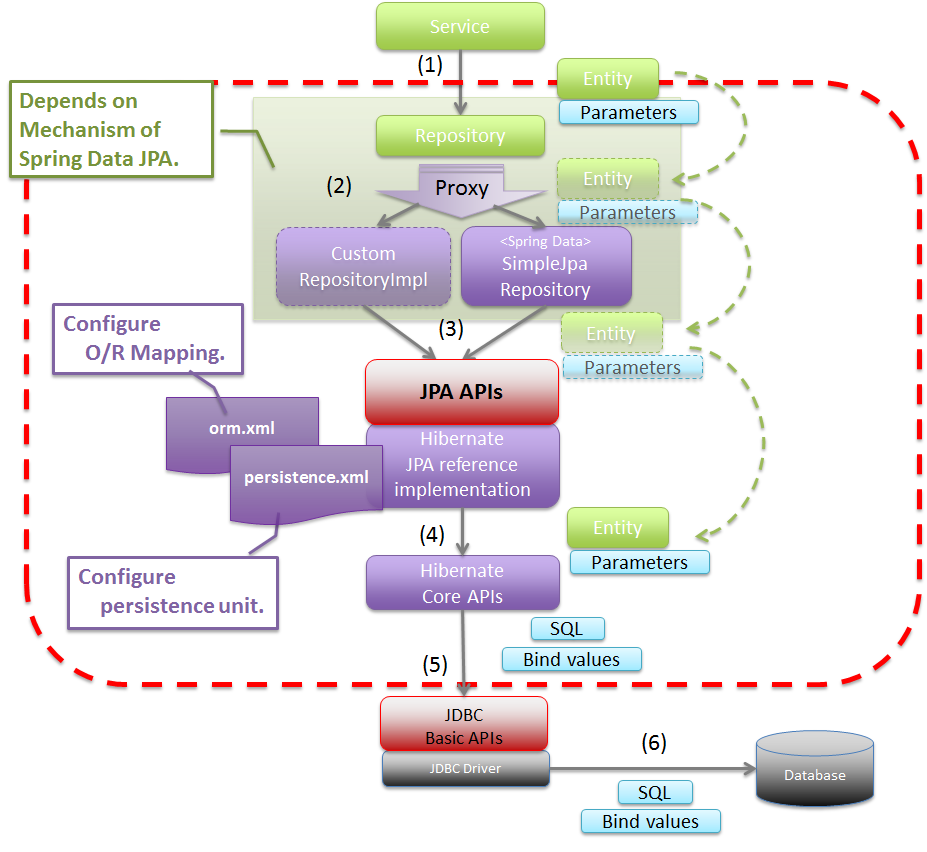
Spring Data JPA **is not a JPA provider**. It is a library/framework that adds an extra layer of abstraction on the top of our JPA provider (like Hibernate) and call the methods of JPA provider (its implementation) in runtime

The JpaRepository interfaces of Spring Data JPA use JPA EntityManager to map its method with JPA EntityManager interface methods.

We can view the picture below to know the overview of methods of EntityManager are called by the repository in Spring Data JPA:



**Basic Spring Data JPA Flow:**

****

(1). Service layer of project call to Repository layer

(2). Project Repository call to Spring Data JPA or Custom RepositoryImpl

(3). The Spring Data JPA call to JPA APIs, it contains Hibernate detail implementation

(4). The JPA APIs call to Hibernate Core APIs

(5). The Hibernate Core APIs call to JDBC Basic APIs, send SQL Bind values instead of Entity Parameters like above

(6). JDBC Basic APIs contains JDBC Drive can call to Database

**Huge Advantages**: No-code Repository, no need to deal with the hibernate configuration, open session, etc,..., reduced boilerplate code, generated queries follow the key words,...

**V. Spring Data JPA important concepts**

**\* SORTING IN JPA**

There are two ways to sort the query results in JPA: Static sorting(Add an ORDER BY clause to your JPQL or native SQL query) and Dynamic sorting(Add the special parameter Sort to your repository method).

**+ Sort with static sorting**

We can use static sorting by adding OrderBy keyword to the method name along with the property name and sort direction (Asc or Desc). Default direction is Ascending.

Example: List<Employee> findByOrderByIdDesc();

To sort query results by multiple attributes, we need to do is just reference multiple properties after the OrderBy clause and specify their sort directions.

Example: List<Employee> findByOrderByFullNameAscIdDesc();

### **Custom Queries with @Query Annotation**

Add the ORDER BY clause to the query definition.

Example: @Query("SELECT e FROM Employee e ORDER BY e.id")

List<Employee> findAllOrderById();

### **Named Queries**

@NamedQuery(name = "Employee.findByAgeGreaterThanNamedJPQL", query = "SELECT e FROM Employee e ORDER BY e.id ASC")

### **Native SQL**

@Query(value = "SELECT \* FROM Employee e ORDER BY e.id ASC", nativeQuery = true)

**+ Sort with dynamic sorting**

Spring Data JPA allows you to add a special Sort parameter to your query method. The Sort class is just a specification that provides sorting options for database queries.By using dynamic sorting, you can **choose the sorting column and direction at runtime** to sort the query results.

Example: List<Employee> findAllByFullName(String fullname, Sort sort);

### **Custom Queries with @Query Annotation**

Add a special Sort parameter to your query method.

@Query("SELECT e FROM Employee e")

List<Employee> findBySalaryRange(Sort sort);

Spring Data JPA **does not support dynamic sorting for native SQL queries.**

### **Using the Sort Parameter**

You have to create a Sort object to specify the entity attributes, you want to use for sorting, and their directions, and then pass this object as a parameter to call the above query methods.

Example: Sort sort = Sort.*by*("id").descending();

There is **no explicit limit** for the number of sorting attributes you can use in a single query. You can use as many as you want by add and() method

If you **want to skip the sorting** for a query that has a Sort parameter, just use the Sort.unsorted() method:

Example: List<Employee> emps = employeeRepository.findAllByFullName("Hieu",Sort.*unsorted*());

**\* NAMED QUERIES**

Your application grows with tens of hundreds of different kinds of queries scattered in many classes, it becomes harder to maintain such a large number of queries in Java codebase. So Spring Data JPA provides the concept of named queries, you can group related queries in one place and refer them in your code by their names

How to create:

- We can create named queries by using an external properties file, Java annotations, or an XML file. Spring Data JPA supports both JPQL and native SQL named queries.

- The definition of a named query is pretty much straightforward. It has two parts: the name and the query.

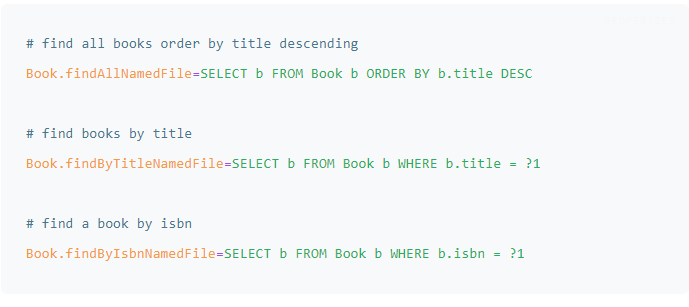
- Syntax: {EntityName}.{RepositoryMethodName}

\*name must be unique, its name should start with the name of the entity class

- There is three way to define a named queries: Using a Properties File, Using the orm.xml File, Using annotations

+ Using a properties file

You can define named queries by using a properties file called jpa-named-queries.properties inside the META-INF folder of your classpath. In a Spring Boot project, by default, this folder is not available. You need to first create META-INF folder inside /src/main/resources/

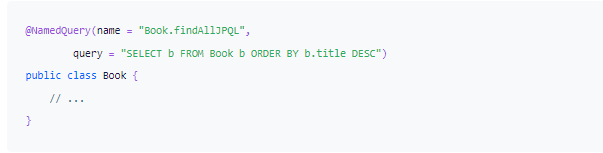


+ Using the orm.xml File

You can create an orm.xml file inside the same META-INF folder for declaring named queries

+ Using Annotations

@NamedQuery, @NamedNativeQuery



**\* DERIVED QUERY METHODS**

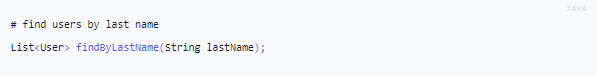
For simple use-cases, you can easily write a derived query method by simply looking at the corresponding method name in the entity class. Just define the query methods in a repository interface that extends one of the Spring Data's repositories such as CrudRepository. Spring Data JPA will create queries automatically by parsing these method names.

\* Derived Query Methods Structure

**- The introducer**: clauses like ***find****,* ***read****,* ***query****,* ***count***, or ***get*** which tells Spring Data JPA what you want to do with the method. This clause can contain further expressions, such as Distinct to set a distinct flag on the query to be created.

**- The criteria**: clause that starts after the first ***By*** keyword. The first By acts as a delimiter to indicate the start of the actual query criteria. The criteria clause is where you define conditions on entity properties and concatenate them with And and Or keywords.

**- Syntax Example**



Spring Data JPA will translate this to: SELECT u FROM User u WHERE u.lastName = ?1

**\* SPRING DATA JPA AUDITING USING SPRING BOOT AND MYSQL**

Auditing helps us in **tracking and logging** the persistence layer changes made by the user in an application. By using auditing, we can easily determine who created or updated the entity record or when it happened.

Spring Data JPA provides excellent support to transparently keep track of who created or changed an entity and the point in time this happened.

## **Create Auditable Abstract Class**

Let us now create an abstract Auditable class with the **createdBy, createdDate, lastModifiedBy**, and **lastModifiedDate** properties. This generic class acts as a base class with all the common auditing fields for the child entities.

To let the Spring Boot knows about these audit fields, you have to annotate the fields with **@CreatedBy** and **@LastModifiedBy** to track the user who created or updated the entity as well as **@CreatedDate** and **@LastModifiedDate** to log the time when these changes were made.

**Auditing Author with AuditorAware**

But how to tell the auditing infrastructure about the author who made these changes? It somehow needs to know this information since we have defined the **@CreatedBy** and **@LastModifiedBy** annotations in our Auditable abstract class.

## **Enable JPA Auditing**

Finally, we need to enable the JPA auditing feature by specifying **@EnableJpaAuditing** on one of our configuration classes. We also need to define a bean of type **AuditorAware** and return an instance of the **EntityAuditorAware** class.

## **Conclusion**

- Define the audit fields by using the @CreatedDate, @CreatedBy, @LastModifiedDate, and @LastModfiiedBy annotations. The best way to do so is by creating a generic abstract class and then extending the entities which need the auditing functionality.

- Implement the AuditorAware interface to let Spring Data JPA auditing infrastructure know about the currently logged-in user who is making the changes.

- Add the @EnableJpaAuditing annotation to any configuration class to enable JPA auditing.

- Expose a bean of type AuditorAware (only required if you need an auditing author).

**\* PAGINATION WITH SPRING DATA JPA**

In the modern web, the response time of your website is a critical factor for higher search engine ranking. Visitors are expecting pages to load quickly and only show the relevant information. For example, if you own an e-commerce website with tens of thousands of products, this means only displaying a small number of products at once, and not all of them.

To help you deal with such situations, Spring Data JPA provides the concepts of pagination. It makes it easy to deal with a large amount of data in the most efficient way.

### **+ Spring Data's Pageable Interface**

Spring Data JPA supports a special parameter called Pageable for paginating the query results. These special parameters are automatically recognized by the Spring Data infrastructure to apply pagination and sorting to database queries dynamically.

The Pageable interface contains the information about the requested page such as the size and the number of the page. It provides the following methods, among others, to add paging to statically define queries:

| [Pageable](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/domain/Pageable.html) | [first](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/domain/Pageable.html#first--)()  Returns the [Pageable](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/domain/Pageable.html) requesting the first page. |
| --- | --- |
| long | [getOffset](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/domain/Pageable.html#getOffset--)()  Returns the offset to be taken according to the underlying page and page size. |
| int | [getPageNumber](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/domain/Pageable.html#getPageNumber--)()  Returns the page to be returned… |

Whenever you want to apply pagination to query results, all you need to do is just add Pageable to the query method definition as a parameter and set the return by Page<T>:

| Page<Person> findByLastName(String lastName, Pageable pageable); |
| --- |

While calling the above method, you need to create an object of Pageable and pass it to the invoked repository method.

The simplest way to create an instance of Pageable is to use the PageRequest class:

| Pageable pageable = PageRequest.of(0, 10); |
| --- |

This will create a page request for the first page (page index is zero-based) with 10 as the size of the page to be returned.

### **+ Spring Data's Slice and Page**

Page extends Slice and knows the total number of elements and pages available by triggering a count query. From the Spring Data JPA documentation:

*“A Page knows about the total number of elements and pages available. It does so by the infrastructure triggering a count query to calculate the overall number. As this might be expensive depending on the store used, Slice can be used in return instead. A Slice only knows about whether there’s a next Slice available which might be just sufficient when walking through a larger result set.”*

| Slice<Person> findByAgeGreaterThan(int age, Pageable pageable); |
| --- |

### **+ Paginating Query Results with Pageable**

The PagingAndSortingRepository interface, we are extending above, provides us the findAll(Pageable pageable) method to paginate all Person entities available in the database. All you need to do is just create an instance of Pageable and pass it to this method to get a Page.

### **- Derived Queries**

You just need to pass the Pageable interface as a parameter to any derived query and set the desired return type as shown below:

| Page<Person> findByFirstName(String firstName, Pageable pageable);  Slice<Person> findByAgeBetween(int start, int end, Pageable pageable);  List<Person> findByLastNameIsNotNull(Pageable pageable); |
| --- |

### **- Custom Queries with @Query Annotation**

In this case, you’ll also just have to pass the Pageable interface as parameter

| @Query("SELECT p FROM Person p WHERE p.lastName = ?1")  Page<Person> findByLastNameJPQL(String lastName, Pageable pageable);  @Query("SELECT p FROM Person p WHERE p.age < :age")  Page<Person> findByAgeLessThanJPQL(@Param("age") int page, Pageable pageable); |
| --- |

To use pagination with native SQL queries declared using the @Query annotation, you need to define the count query by yourself, as shown in the following example:

| @Query(value = "SELECT \* FROM Person p WHERE p.firstName = :firstName",  countQuery = "SELECT count(\*) Person p WHERE p.firstName = :firstName",  nativeQuery = true)  Page<Person> findByFirstNameNativeSQL(@Param("firstName") String firstName, Pageable pageable); |
| --- |

If you want to skip the pagination for query methods that require an instance of Pageable, just use the Pageable.unpaged() method:

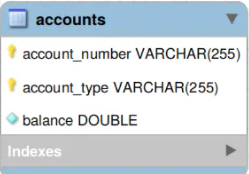
| Slice<Person> personSlice = personRepository.findByAgeBetween(20, 60, Pageable.unpaged()); |
| --- |

# **\* SPRING DATA JPA COMPOSITE PRIMARY KEY MAPPING**

In many cases, in real life, you may need more than one identity attribute for one object. Let’s take clothing for example. Two shirts might be distinguished by the product code, but between two similar shirts from the same brand, same color, they still might be different by the size (L, XL). So instead of using only one primary key, some time, we’ll have to deal with a primary key that contains several fields rather than one.

### **+ Mapping Composite Key using @IdClass Annotation**

Let us consider an application that manages different types of bank accounts. Each bank account has an account number and type (i.e. checking or saving), among other information. Now we want to create a compound key using this information to uniquely identify each account in the database.



The accounts table has a composite primary key, which consists of two columns:

* **account\_number**
* **account\_type**

To map this database relationship using Spring Data JPA, we need to create a separate composite primary key class with both these primary key columns:

Class ***AccountId***

| package com.attacomsian.jpa.composite.domains;  import java.io.Serializable;  import java.util.Objects;  public class AccountId implements Serializable {  private String accountNumber;  private String accountType;  public AccountId() {  }  public AccountId(String accountNumber, String accountType) {  this.accountNumber = accountNumber;  this.accountType = accountType;  }  @Override  public boolean equals(Object o) {  if (this == o) return true;  if (o == null || getClass() != o.getClass()) return false;  AccountId accountId = (AccountId) o;  return accountNumber.equals(accountId.accountNumber) &&  accountType.equals(accountId.accountType);  }  @Override  public int hashCode() {  return Objects.hash(accountNumber, accountType);  }  } |
| --- |

Once we’re finished, we create another class named ***Account***:

| package com.attacomsian.jpa.composite.domains;  import javax.persistence.Entity;  import javax.persistence.Id;  import javax.persistence.IdClass;  import javax.persistence.Table;  import java.io.Serializable;  @Entity  @Table(name = "accounts")  @IdClass(AccountId.class)  public class Account implements Serializable {  @Id  private String accountNumber;  @Id  private String accountType;  private double balance;  public Account() {  }  public Account(String accountNumber, String accountType, double balance) {  this.accountNumber = accountNumber;  this.accountType = accountType;  this.balance = balance;  }  // getters and setters, equals(), toString() .... (omitted for brevity)  } |
| --- |

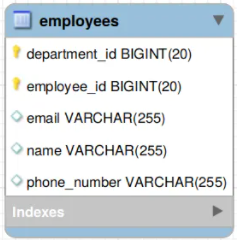
As you can see above, we have annotated the Account class with @IdClass to specify a composite primary key class that is mapped to multiple fields of the entity. The @Id annotation is then used to indicate all properties which are a part of the compound key.

### **+ Mapping Composite Key using @EmbeddedId Annotation**

In addition to @IdClass, Spring Data JPA provides another annotation — @EmbeddedId — to define a composite primary key.

Let us consider another example application that manages employees of a company with multiple departments. Each employee has a unique ID within his own department. But the same ID can be assigned to a different employee in another department. So we cannot uniquely identify an employee just by his employee ID.

To uniquely identify an employee, we need to know his employee ID as well as his department ID, as shown below:



To map the above relationship using Spring Data JPA, you need to create a separate composite primary key class annotated with @Embeddable:

| package com.attacomsian.jpa.composite.domains;  import javax.persistence.Column;  import javax.persistence.Embeddable;  import java.io.Serializable;  import java.util.Objects;  @Embeddable  public class EmployeeId implements Serializable {  @Column(name = "employee\_id")  private Long employeeId;  @Column(name = "department\_id")  private Long departmentId;  public EmployeeId() {  }  public EmployeeId(Long employeeId, Long departmentId) {  this.employeeId = employeeId;  this.departmentId = departmentId;  }  public Long getEmployeeId() {  return employeeId;  }  public void setEmployeeId(Long employeeId) {  this.employeeId = employeeId;  }  public Long getDepartmentId() {  return departmentId;  }  public void setDepartmentId(Long departmentId) {  this.departmentId = departmentId;  }  @Override  public boolean equals(Object o) {  if (this == o) return true;  if (o == null || getClass() != o.getClass()) return false;  EmployeeId that = (EmployeeId) o;  return employeeId.equals(that.employeeId) &&  departmentId.equals(that.departmentId);  }  @Override  public int hashCode() {  return Objects.hash(employeeId, departmentId);  }  } |
| --- |

The next step is to create the Employee class and embed the above composite primary class into it by using the @EmbeddedId annotation:

| package com.attacomsian.jpa.composite.domains;  import javax.persistence.Column;  import javax.persistence.EmbeddedId;  import javax.persistence.Entity;  import javax.persistence.Table;  import java.io.Serializable;  @Entity  @Table(name = "employees")  public class Employee implements Serializable {  @EmbeddedId  private EmployeeId employeeId;  private String name;  @Column(unique = true)  private String email;  private String phoneNumber;  public Employee() {  }  public Employee(EmployeeId employeeId, String name, String email, String phoneNumber) {  this.employeeId = employeeId;  this.name = name;  this.email = email;  this.phoneNumber = phoneNumber;  }  // getters and setters, equals(), toString() .... (omitted for brevity)  } |
| --- |

### 

### 

### **Compare @IdClass vs @EmbeddedId**

| **@IdClass** | **@EmbeddedId** |
| --- | --- |
| You need to specify the primary key columns twice — once in the composite primary key class and then again in the entity class with the @Id annotation. |  |
|  | You can access the entire primary key object using the field access method. This also gives a clear notion of the fields that are part of the composite key because they are all aggregated in a class that is only accessible through a field access method. |
| @IdClass, the query is a little simpler:  SELECT a.accountType FROM Account a | With @EmbeddedId, you have to write more text for a similar query:  SELECT e.employeeId.departmentId FROM Employee e |