JPA - PART5 (DTO - TABLE)

spring.jpa.hibernate.ddl-auto configuration tells hibernate regarding how to create and manage the DB Schema.

S.No.	Values	Create Schema	Update Schema	Delete Schema	Details
1.	none	no	no	no	Do nothing. Good for ***Production***
2.	update	yes	yes	no	Does update but without deleting any existing data or schema. Good for ***Development*** environment
3.	validate	no	no	no	During application startup, does matching between entities and DB Schema. If mismatch found, throws exception.
4.	create	yes	yes	yes	Drops and re-create the schema during application startup.
5.	create-drop	Yes	yes	yes	Creates the schema during startup and drops the schema when the application shutdown. *(generally by-default for in memory databases like H2)*

Mapping Classes to Tables

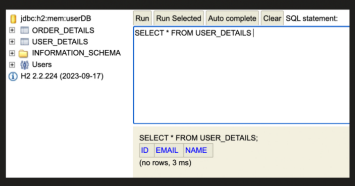
@Tables Annotation

- Its an Optional field, if not defined, hibernate will generate table name based on entity name.
- Generally it follows CamelCase to UPPER_SNAKE_CASE means: UserDetails -> USER_DETAILS

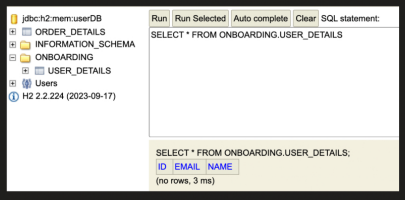
Report Abuse

```
@Target(TYPE)
@Retention(RUNTIME)
public @interface Table{
String name() default "";
String schema() default"";
UniqueConstraint[] uniqueConstraints() default{};
Index[] indexes() default{};
}
```

```
@Table(name= "USER_DETAILS")
@Entity
public class UserDetails {

//fields and their getters and setters

}
```




```
@Table(name= "USER_DETAILS", schema= "ONBOARDING")
@Entity
public class UserDetails {

//fields and their getters and setters

}
```




```
@Table(name= "USER_DETAILS",
        schema= "ONBOARDING",
         uniqueConstraints={
              @UniqueConstraint(columnNames="phone"), //single column unique constraint
              @UniqueConstraint(columnNames={"name","email"}) //composite unique constraint
       })
@Entity
public class UserDetails {

       @Id
       private Long id;
       private String name;
       private String email;
       private String phone;

       //Constructors
       public UserDetails(){
```

SELECT * FROM INFORMATION_SCHEMA.CONSTRAINT_COLUMN_USAGE;

```
    }

    //Getters and setters
}
```

TABLE_CATALOG	TABLE_SCHEMA	TABLE_NAME	COLUMN_NAME	CONSTRAINT_CATALOG	CONSTRAINT_SCHEMA	CONSTRAINT_NAME
USERDB	ONBOARDING	USER_DETAILS	ID	USERDB	ONBOARDING	CONSTRAINT_3
USERDB	ONBOARDING	USER_DETAILS	PHONE	USERDB	ONBOARDING	UKHWI9ICWRBOMPOWJO1L8MQQU49
USERDB	ONBOARDING	USER_DETAILS	EMAIL	USERDB	ONBOARDING	UKDRYA4RWKEMVJ1HS8D8P7N59FM
USERDB	ONBOARDING	USER_DETAILS	NAME	USERDB	ONBOARDING	UKDRYA4RWKEMVJ1HS8D8P7N59FM

```
@Table(name= "USER_DETAILS",
        schema= "ONBOARDING",
        uniqueConstraints={
            @UniqueConstraint(columnNames="phone"), //single column unique constraint
            @UniqueConstraint(columnNames={"name","email"}) //composite unique constraint
        },
        indexes={
            @Index(name="index_phone", columnList="phone"), //index on single column
            @Index(name="index_name_email", columnList="name, email") //index on composite column
    })
@Entity
public class UserDetails {

    @Id
    private Long id;
    private String name;
    private String email;
    private String phone;

    //Constructors
    public UserDetails(){
    }

    //Getters and setters
}
```

```
SELECT * FROM INFORMATION_SCHEMA.INDEX_COLUMNS
```

SELECT * FROM INFORMATION_SCHEMA.INDEX_COLUMNS;

INDEX_CATALOG	INDEX_SCHEMA	INDEX_NAME	TABLE_CATALOG	TABLE_SCHEMA	TABLE_NAME	COLUMN_NAME
USERDB	ONBOARDING	PRIMARY_KEY_3	USERDB	ONBOARDING	USER_DETAILS	ID
USERDB	ONBOARDING	INDEX_PHONE	USERDB	ONBOARDING	USER_DETAILS	PHONE
USERDB	ONBOARDING	INDEX_NAME_EMAIL	USERDB	ONBOARDING	USER_DETAILS	NAME
USERDB	ONBOARDING	INDEX_NAME_EMAIL	USERDB	ONBOARDING	USER_DETAILS	EMAIL
USERDB	ONBOARDING	UKHWI9ICWRBOMPOWJO1L8MQQU49_INDEX_3	USERDB	ONBOARDING	USER_DETAILS	PHONE
USERDB	ONBOARDING	UKDRYA4RWKEMVJ1HS8D8P7N59FM_INDEX_3	USERDB	ONBOARDING	USER_DETAILS	NAME
USERDB	ONBOARDING	UKDRYA4RWKEMVJ1HS8D8P7N59FM_INDEX_3	USERDB	ONBOARDING	USER_DETAILS	EMAIL

@Column Annotation

- Its an Optional field, if not defined, JPA will add it with default values.

```
@Table(name = "user_details")
@Entity
public class UserDetails {

    @Id
    private Long id;
    @Column(name = "full_name", unique = true, nullable = false, length = 255)
    private String name;
    private String email;
    private String phone;

    // Constructors
    public UserDetails() {
    }

    // Getters and setters
}
```

jdbc:h2:mem:userDB
- ORDER_DETAILS
- USER_DETAILS
- INFORMATION_SCHEMA
- ONBOARDING
- Users

H2 2.2.224 (2023-09-17)

Run | Run Selected | Auto complete | Clear | SQL statement:

```
SELECT * FROM USER_DETAILS
```

SELECT * FROM USER_DETAILS;

ID	EMAIL	FULL_NAME	PHONE

(no rows, 4 ms)

@Id Annotation and @GeneratedValue Annotation

Primary Key: must be unique, not null and used to uniquely identify each record.
- **@Id** annotation is used to mark the field as primary key.
- Each entity can have only 1 primary key.
- Only 1 field can be annotated with @Id.

Composite Primary key: combination of two or more columns to form a primary key.

Using **@Embeddable** and **@EmbeddedId** annotation.

Using **@IdClass** and **@Id** annotation

Rules to follow for both the approach:

- Must be a public class.

- Must Implement the Serializable interface.
- Must have no-arg constructor
- Must override the equals() and hashCode() methods

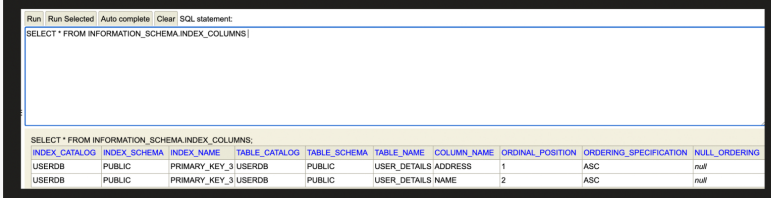
Using **@IdClass** and **@Id** annotation

```java
@Table(name = "user_details")
@Entity
public class UserDetails {

    private String name;
    private String address;
    private String phone;

    // Constructors
    public UserDetails() {
    }

    // Getters and setters

}
```

I want these 2 columns to be defined as Composite Key

```java
@Table(name = "user_details")
@IdClass(UserDetailsCK.class)
@Entity
public class UserDetails {

    @Id
    private String name;
    @Id
    private String address;
    private String phone;

    // Constructors
    public UserDetails() {
    }

    // Getters and setters
}
```

```java
public class UserDetailsCK implements Serializable {

    private String name;
    private String address;

    public UserDetailsCK() {
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (!(obj instanceof UserDetailsCK)) {
            return false;
        }
        UserDetailsCK userCK = (UserDetailsCK) obj;
        return this.name.equals(userCK.name) && this.address.equals(userCK.address);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, address);
    }
}
```



Why we need to override equals and hashCode methods?

- From previous video, we know that JPA internally maintains FIRST LEVEL CACHING.
- Also, we can implement SECOND LEVEL CACHING.
- And these caching uses HashMap and which relies on key (generally primary key becomes the key). So proper equals() and hashCode() method is required.

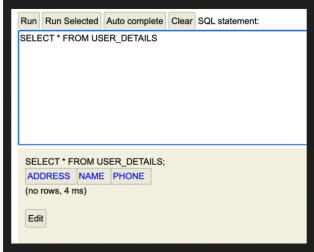
Whey we need to implement Serializable interface?

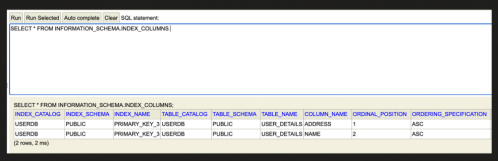
- Unlike Single primary key like String, Long etc. Composite key are custom classes.
- So, JPA need to make sure that those class should be properly serializable (in case if required like transfer over the network in case of distributed caching).

Using **@Embeddable** and **@EmbeddedId** annotation.

```java
@Table(name = "user_details")
@Entity
public class UserDetails {

    @EmbeddedId
    UserDetailsCK userDetailsCK;
    private String phone;

    // Constructors
    public UserDetails() {
    }

    // Getters and setters
}
```

```java
@Embeddable
public class UserDetailsCK implements Serializable {

    private String name;
    private String address;

    public UserDetailsCK() {
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (!(obj instanceof UserDetailsCK)) {
            return false;
        }
        UserDetailsCK userCK = (UserDetailsCK) obj;
        return this.name.equals(userCK.name) && this.address.equals(userCK.address);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, address);
    }
}
```





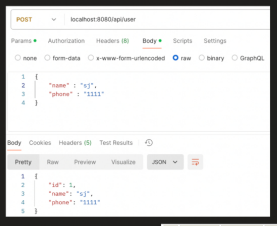
@GeneratedValue Annotation

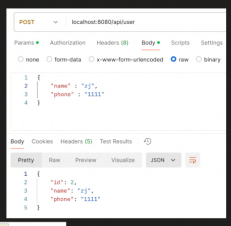
- Now we know, how to define Primary key.
- But we can also define its generation strategy too. By default, primary key columns are not autofill.
- It works with @Id annotation (only for single primary key not for composite one)

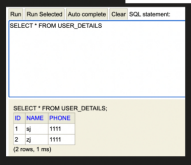
1. GenerationType.IDENTITY

- Each insert, generates a new identifier (auto-increment field)

```java
@Table(name = "user_details")
@Entity
public class UserDetails {

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

    // Constructors
    public UserDetails() {
    }

    public UserDetails(String name, String phone) {
        this.name = name;
        this.phone = phone;
    }

   //getters and setters
}
```







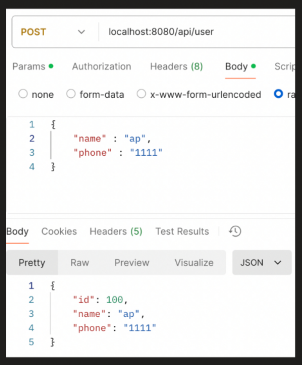
2. GenerationType.SEQUENCE

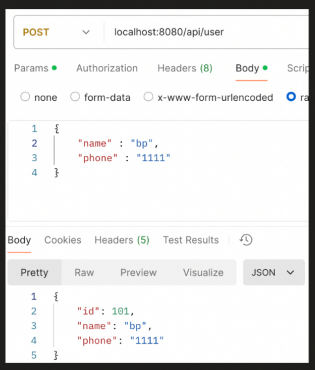
- Used to generate Unique numbers.
- Speed up the efficiency when we cache sequence values.
- More control than IDENTITY.

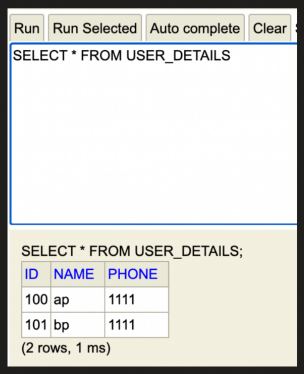
>> CREATE SEQUENCE user_seq INCREMENT BY 25 START WITH 100 MAXVALUE 9999;

```java
@Table(name = "user_details")
@Entity
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "unique_user_seq")
    @SequenceGenerator(name = "unique_user_seq", sequenceName = "db_seq_name", initialValue = 100, allocationSize = 5)
    private Long id;
    private String name;
    private String phone;

    // Constructors
    public UserDetails() {
    }

    public UserDetails(String name, String phone) {
        this.name = name;
        this.phone = phone;
```

```
    }

    //getters and setters
}
```







- ***name***: it's a unique name, internal for JPA, we can use it in different entity.
- ***sequenceName***: it's a name which is created in DB. Or if you have manually created the sequence in DB, then use that name here.
- ***initialValue***: sequence no starts from.
- ***allocationSize***: cache data, hibernate prefetch this much ids before hand, so that it will not query DB again and again.

We have given allocationSize = 5, so after 5 calls only, next db call will be made for sequence

```
Hibernate:
    insert
    into
        user_details
        (name, phone, id)
    values
        (?, ?, ?)
```
→ ***1st call***

```
Hibernate:
    insert
    into
        user_details
        (name, phone, id)
    values
        (?, ?, ?)
```
→ ***2nd call***

```
Hibernate:
    insert
    into
        user_details
        (name, phone, id)
    values
        (?, ?, ?)
```
→ ***3rd call***

```
Hibernate:
    insert
    into
        user_details
        (name, phone, id)
    values
        (?, ?, ?)
```
→ ***4th call***

```
Hibernate:
    insert
    into
        user_details
        (name, phone, id)
    values
        (?, ?, ?)
```
→ ***5th call***

```
Hibernate:
    select
        next value for db_seq_name
```
→ ***After 5th call, hibernate will fetch another 5 values.***

```
Hibernate:
    insert
    into
        user_details
        (name, phone, id)
    values
        (?, ?, ?)
```

Advantage of SEQUENCE over IDENTITY:

1. Custom logic (start point, increment etc.)
2. Sequence generation logic is independent of table, so multiple tables an use it.f
3. Range of IDs can be cached, so we can avoid hitting database each time a new id is required. (during IDENTITY, while INSERTION internally DB is auto generating the next ID , which require additional DB call)
4. Better portability, means IDENTITY is very DB specific while SEQUENCE can provide more consistent behavior across multiple DBs.

3. GenerationType.TABLE

- @TableGenerator annotation is used but its very less efficient.
- Because:
  - Separate Table is created, just for managing unique IDs.
  - Each time id is required, SELECT-UPDATE query is executed.
  - Complex concurrency handling, when multiple operations happening in parallel, it requires LOCK/UNLOCK functionality. Which can lead to performance bottleneck.
    In SEQUENCE type, its handle internally by DB using atomic counter, so its much more efficient.