3. Working with Entities

Contents

[1. Introducing Working with Entities 1](#_Toc27666)

[2. Demo: A Secondary Table with One Field 2](#_Toc22417)

[3. Demo: A Secondary Table with Multiple Fields 3](#_Toc5679)

[4. Demo: Multiple Secondary Tables 4](#_Toc15478)

[5. Entity Access Types 5](#_Toc13041)

[6. Entity Primary Keys and Entity Identity 5](#_Toc9344)

[7. Demo: Primary Keys with @GeneratedValue 6](#_Toc24061)

[8. Demo: Embeddable Primary Key and Embedded ID 7](#_Toc8052)

[9. Demo: Embeddable Primary Key and ID Class 8](#_Toc2037)

# 1. Introducing Working with Entities

=>slides: Pg. 1

Hello. Welcome to this module about working with entities. After having been introduced to the main concepts of ORM, JPA, and Hibernate, we'll move now to discuss how to effectively use entities in our programs and how to persist them in the database. So let's move farther.

=>slides: Pg. 2

This module will start by examining what an entity is and which are its particularities. We'll demonstrate how to map objects, analyze the entity access types, and define entity primary keys and entity identity. =>slides: Pg. 3

What is an entity and which are its particularities? An entity is a domain object that can be persisted. An entity class must be annotated with the Entity annotation or be defined as entity through XML configuration. Must be a top‑level class, and apps and interfaces cannot be entities. Must have a public or a protected no‑arguments constructor. It may define other constructors. The JPA specification requires an entity class not to be final. Hibernate is not so strict, and it will allow to declare final classes as entities. However, this is not a good practice as this will prevent Hibernate to use the proxy pattern for performance improvement.

=>slides: Pg. 4

An entity supports inheritance, polymorphic associations, and polymorphic queries.

=>slides: Pg. 5

The persistent state of the entity is represented by the instance variables. The persistent state may be of the following types. Primitive types, byte, short, int, long, float, double, boolean char. Serializable types, including here wrappers of the primitive types as integer or character or user‑defined types implementing the serializable interface. Other entity types, enums, embeddable types, and we'll demonstrate later what these embeddable types mean and how to work with them. Or collections of serializable types of embeddable types or of entity types.

=>slides: Pg. 6

The database tables to be mapped to an object to be persisted are determined by annotations, by the parameters of the annotations, and by the default rules. The annotations specifying the mapping of tables are Table, SecondaryTable, SecondaryTables.

=>slides: Pg. 7

The Table annotation specifies the primary table for the annotated entity. An entity class may lack the Table annotation, and, in this case, the default value will apply. This means that the corresponding table will have the same name as the entity class. The Table annotation may receive various parameters, the table name, which defaults to the entity name, the table catalog, which defaults to the default catalog, the table schema, which defaults to the default schema. You may also add the unique constraint to be placed on the table and the indexes, both used if the table generation is in effect.

=>slides: Pg. 8

The SecondaryTable annotation specifies a secondary table for the annotated entity class. If missing, all persistent fields of the entity are mapped to the Primary table. The parameters are the same as the ones from the Table annotation plus pkJoinColumns and foreignKey, which specify the columns that are used to join with the primary key and the foreign key constraint for the columns corresponding to the pkJoinColumns element used if the table generation is in effect.

=>slides: Pg. 9

The SecondaryTables annotation specifies multiple secondary tables for an entity class. If missing, all persistent fields of the entity are mapped to the primary table. It receives as parameters multiple SecondaryTable annotations, each one specifying one of the secondary tables foreign entity class. Each table specified by the SecondaryTable Annotation may keep the primary key name, or it may specify differently named primary keys.

# Demo: A Secondary Table with One Field

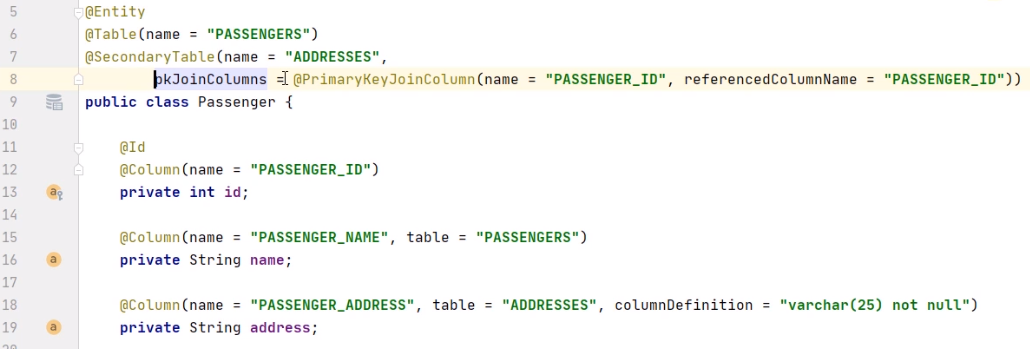
=>slides: Pg. 10

We move to the practical demonstration to show an example that uses a secondary table with one field. More exactly, it will use one passenger with one address, and the address will have one field. We go back to the code, and we have already set up a skeleton Maven project, including the pom.xml configuration. The project will be executed using Java 11. We added the hibernate‑core dependency version 5.4.20.Final, the latest stable version at the time of developing this course. And we have also added the mysql‑connector dependency version 8.0.20 as we interact with a MySQL database. We 

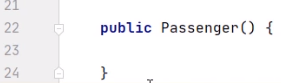
declare the persistence unit to be hibernatefundamentals.m03.ex01, and we choose Hibernate as persistence provider. We indicate here the JDBC properties. The driver is the MySQL JDBC driver.

The URL allows accessing the database. 3306 is the MySQL port, and M03\_EX01 will be the name of the database. On my machine, the user is root having the password root. We indicate the hibernate.dialect to be MySQL. While executing, we'll show the SQL code in a formatted way. Every time we execute the program, the database will be created from the scratch.

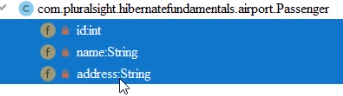




We move to the code to effectively work with the objects to be persisted, and first we create the Passenger class. We annotate the class with the Entity annotation. The corresponding table in the database for the Passenger entity will be the PASSENGERS table. In the Passenger class, we add the private int id field and the private\_String name field. Also, the passenger will have an address, and we keep it in the private\_String address field. We generate the constructor having all these fields, id, name, and address,

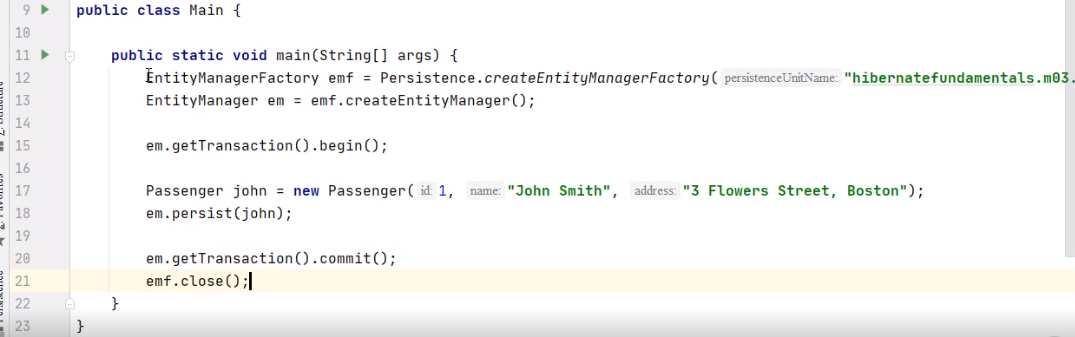


and we generate a no‑arguments constructor as any entity needs to have.



We finally generate the getters and setters for the id, name, and address fields. We annotate the id filled with the Id notation, meaning that it is a primary key. And we also annotate it with the Column name = PASSENGER\_ID annotation, indicating the name of the field in the database. We annotate the name field with the Column name = PASSENGER\_NAME annotation indicating the name of the column in the database. The column will belong to the PASSENGERS table. We annotate the address field with the Column name = PASSENGER\_ADDRESS annotation indicating the name of the column in the database. The column will belong to the ADDRESSES table, and its definition will be varcar of 25 not null. We annotate the passenger entity with SecondaryTable. The name of the secondary table is ADDRESSES. Primary key join column specifies the mapping of the foreign key PASSENGER\_ID column of a secondary table to the PASSENGER\_ID column of the primary table.





We move to the main class, for which we have already created the EntityMain method. We create an EntityManagerFactory object emf with the help of the Persistence.CreateEntityManagerFactory method. This method will need as argument the persistence unit name we inserted in persistence.xml. We create an EntityManager with the help of the createEntityManagerFactory method. Then we begin the transaction. We create the passenger with id1 and the name of John Smith having the address 3 Flower Street, Boston. We need to persist John and then to commit the transaction and close the EntityManagerFactory. We go on the side of the MySQL database that we have on the machine**. We need to create the database that we work with having the same name as indicated in the persistence unit from the persistence.xml file in the Java program.**



**So we execute the command CREATE DATABASE M03\_EX01**. And now we may run the Java program. We'll create the needed tables, we'll create the relationships between them, and we'll insert the information from our objects.



We go back to the MySQL database and commit to our database by executing the USE M03\_EX01 command. Let's check the content of the two expected tables. We execute SELECT all FROM PASSENGERS. We expect John Smith's to be there. And yes, we have the passenger inside the table. 

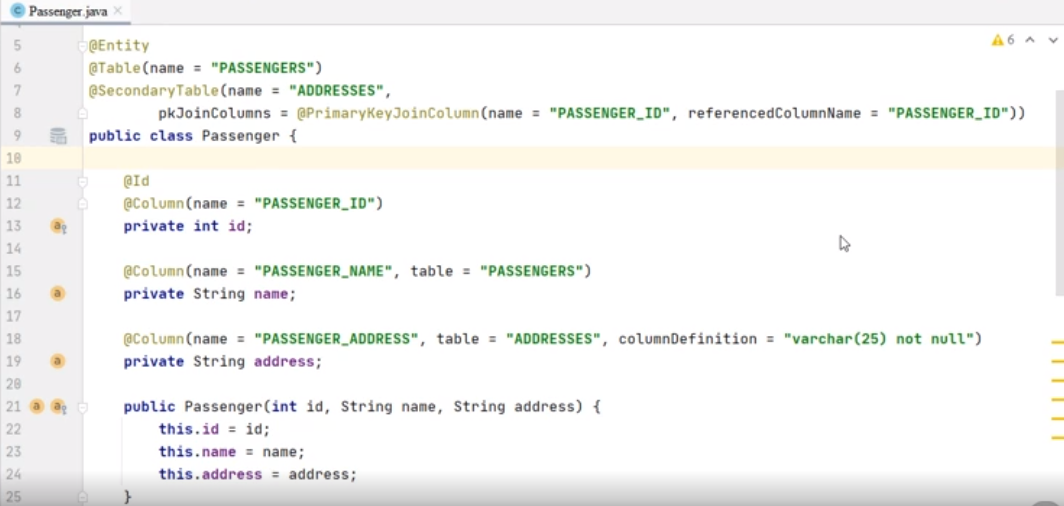
We finally verified the ADDRESSES table. We expect John Smith's address to be there. And yes, we have the 3 Flower Street, Boston address belonging to John. So this concludes our demonstration. We proved how we can create an entity that will generate in the database two different tables with a relationship between them.

# Demo: A Secondary Table with Multiple Fields

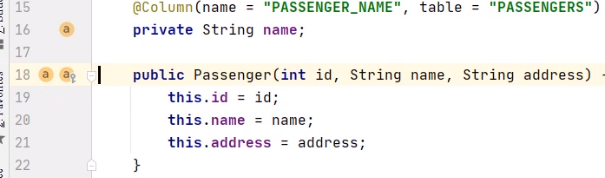
=>slides: Pg. 11

We'll move now to the next demonstration, a demonstration using a secondary table with multiple fields. More exactly, we have a passenger with an address, and the address has multiple fields.

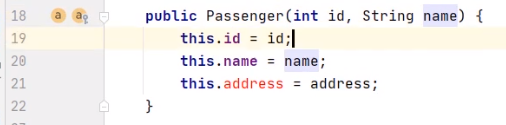


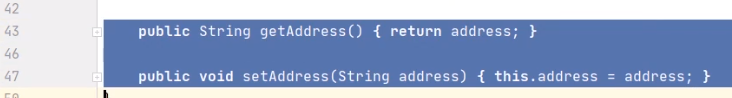


What if the address representation needs more than one field? The address maybe provided a street number, zip code, and city.

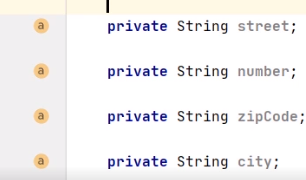


We removed this address field.

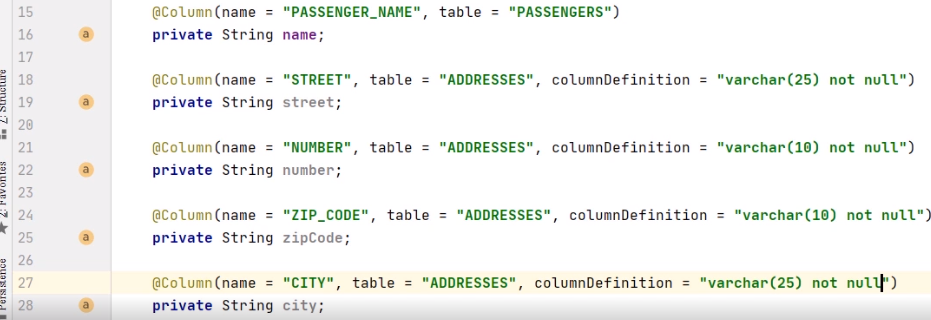


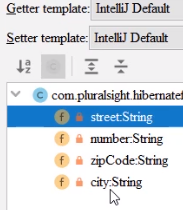
We fixed the constructor by eliminating the address parameter and by removing the assignment inside it. 

We also need to remove the setter and the getter for the address field.



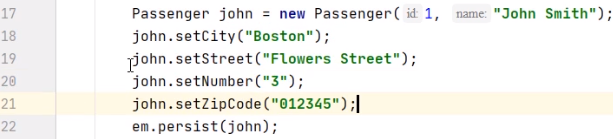
So the address is composed from the private String street field, the private String number field, the private String zipCode field, and the private String city field.

 We'll annotate the street field with the Column name = STREET annotation, indicating the name of the column in the database. The column will belong to the ADDRESSES table, and its definition will be varcar of 25 not null. We'll annotate the number field with the Column of name = NUMBER annotation, indicating the name of the column in the database. The column will belong to the ADDRESSES table, and its definition will be varcar of 10 not null. We'll annotate the zip code field with the Column = ZIP\_CODE annotation, indicating the name of the column in the database. The column will belong to the ADDRESSES table, and its definition will be varchar of 10 not null. We'll annotate the city field with the Column name = CITY annotation, indicating the name of the column in the database. The column will belong to the ADDRESSES table, and its definition will be varcar of 25 not null.



We finally generate the getters and setters for the street, number, zipCode, and city fields.



We move to the Main class. And first, we fix the call to the constructor of the Passenger class, which now only takes two parameters. We'll call the setters to set all the newly introduced fields, street, number, zip code, and city. On the MySQL database, we need to create a database with the same name as in the persistence unit from the persistence.xml file.



So we execute the command CREATE DATABASE M03\_EXO2. And now, we may run the Java program. We'll create the needed tables, we'll create the relationships between them, and we'll insert the information from our object.



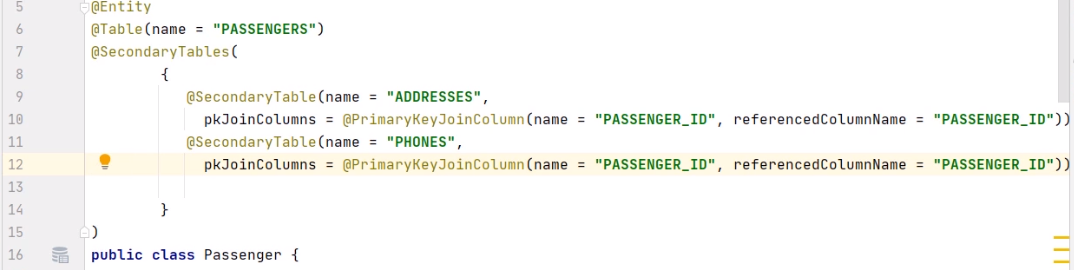
We go back to the MySQL database and commit to our database by executing the USE M03\_EX02 command. Let's check the content of the two expected tables. We execute SELECT all FROM PASSENGERS. We expect John Smith to be there. And yes, we have the passenger inside the table. We execute SELECT all FROM ADDRESSES. We expect John Smith address to be there, and yes, we have four fields, CITY, NUMBER, STREET, and ZIP\_CODE, representing the address belonging to John. So we proved how we can create an entity that will generate in the database two different tables with the second table containing multiple fields.

# Demo: Multiple Secondary Tables

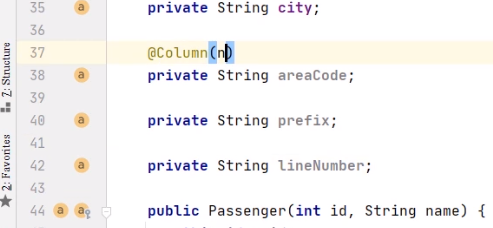
=>slides: Pg. 12

We'll move to the next demonstration, and we would like to show how to use multiple secondary tables. More exactly, we'll show how to use a passenger with an address and the phone.

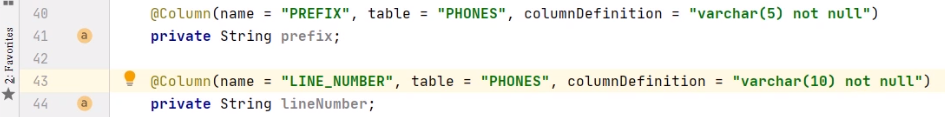
So there are different information that are independent of each other and may go to different tables. The passenger may have an address and the phone number. The logic will push the different information to different secondary tables.

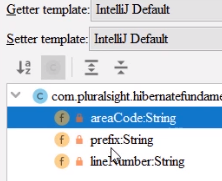


We remove the SecondaryTable annotation, and we replace it with the SecondaryTables one, and there will be two dependent tables now. The SecondaryTable number 1 will be ADDRESSES. PrimaryKeyJoinColumn specifies the mapping of the foreign key, PASSENGER\_ID column of the secondary table to the PASSENGER\_ID column of the primary table. The SecondaryTable number 2 will be PHONES. PrimaryKeyJoinColumn specifies the mapping of the foreign key PASSENGER\_ID column of the secondary table to the PASSENGER\_ID column of the primary table.



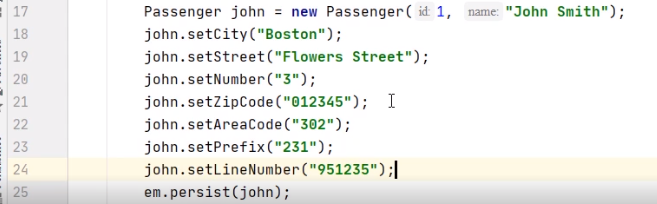
We'll add the new fields for the phone, the areaCode field, the prefix field, and the lineNumber field. We'll annotate the areaCode field with the Column name = AREA\_CODE annotation. The column will belong to the PHONES table, and its definition will be varchar of 5 not null.

We'll annotate the prefix field with the Column name = PREFIX annotation. The column will belong to the PHONES table, and its definition will be varchar of 5 not null. We'll annotate the lineNumber field with the Column name = LINE\_NUMBER annotation. The column will belong to the PHONES table, and its definition will be varchar of 10 not null.



We finally generate the getters and setters for the areaCode, prefix, and lineNumber fields.





We move to the Main class and set John's phone number using the setters for the area code, prefix, and lineNumber fields.



On the MySQL database, we need to create the database with the same name as in the persistence unit from the persistence.xml file. So we execute the command CREATE DATABASE M03\_EX03. And now we may run the Java program. We'll create the needed tables, we'll create the relationships between them, and we'll insert the information from our object.



We go back to the MySQL database and commit to our database by executing the USE M03\_EX03 command. Let's check the content of the three expected tables. We execute SELECT all FROM PASSENGERS. We expect John Smith to be there. And yes, we have the passenger inside the table. We verify the ADDRESSES table.



We execute SELECT all FROM ADDRESSES. And yes, John Smith's address is there in the format with four fields. We finally verify the PHONES table.



We execute SELECT all FROM PHONES. We expect John Smith's phone to be there. And yes, it is represented in the format with three fields. So we proved how we can create multiple secondary tables. One entity generates in the database three different tables with tables number 2 and 3 related to the first table as secondary tables.

# Entity Access Types

=>slides: Pg. 13

Let's examine the entity access types.

=>slides: Pg. 14

There are two ways the persistence provider runtime uses in order to access the persistent state of the entity, field access through the instance variables and property access through the getter and setter methods.

=>slides: Pg. 15

The access type is the methods through which the persistence runtime accesses the persistent state of the entity. Single access type, field or property, applies to an entity by default. The access type is determined by placing the mapping annotations. If annotations are on persistent fields, then field‑based access is used. If annotations are on getters, then property‑based access is used.

=>slides: Pg. 16

In this example, the access is field‑based.

=>slides: Pg. 17

In this other example, the access is property‑based.

=>slides: Pg. 18

If there are no mapping annotations, then the access is determined based on the position of the Id annotation. If it is on a field, then field‑based access is used for the entire class.

=>slides: Pg. 19

If it is on a property, then property‑based access is used for the entire class.

=>slides: Pg. 20

The access type can be explicitly defined via the Access annotation. Access of AccessType.Field applied to an entity class defines that default access type for this class being field‑based. The persistence runtime accesses persistent state via the instance variables.

=>slides: Pg. 21

It is possible to selectively designate individual attributes within the class for property access by specifying Access of AccessType.PROPERTY for needed properties. Access of AccessType.PROPERTY applied to an entity class defines access type defaults for this class being property‑based. The persistence runtime accesses persistent state via the properties. It is possible to selectively designate individual attributes within the class, for instance variable access by specifying Access of AccessType.FIELD for needed instance variables.

=>slides: Pg. 22

This example shows how to mix the access type for the passenger entity. The default access is provided at the level of the class using the Access of AccessType.FIELD annotation and is overridden at the level of the name attribute using the Access of AccessType.PROPERTY annotation.

=>slides: Pg. 23

Let's now compare the field‑based and the property‑based access. Why should you use one or another one? Using field‑based access, you may omit the getter methods for the fields that should not be exposed. The fields are declared on one single line, while the accessor methods will spread on multiple lines. Field‑based access will provide easier readability of the code. Accessor methods may execute additional logic. If this is what you would like to happen when persisting an object, you may use property‑based access. If the persistence would like to avoid these additional actions, you may use field‑based access.

# Entity Primary Keys and Entity Identity

=>slides: Pg. 24

Let's examine now how to define entity primary keys and entity identity.

=>slides: Pg. 25

Every entity must have a primary key. A primary key may correspond to one or more fields or properties of the entity class. A simple primary key always corresponds to a single persistence field or property. You may use the id annotation to define a simple primary key. A composite primary key may correspond to either a single persistence field or property, or to a set of such fields or properties. We'll demonstrate how to use the annotations, embedded id and id class, to define a composite primary key. =>slides: Pg. 26

There is a series of rules to apply for composite primary keys. First, a primary key class must be public and must have a public no‑arguments constructor. Then, a primary key class must be serializable. A primary key class must define the equals and hashCode methods. The semantics of logical equality must be consistent with the database equality. A primary key class can be represented as an embeddable class or as an id class. If a primary key is represented as an id class, the fields or properties must correspond to the entity fields and properties, referring here to names and types.

=>slides: Pg. 27

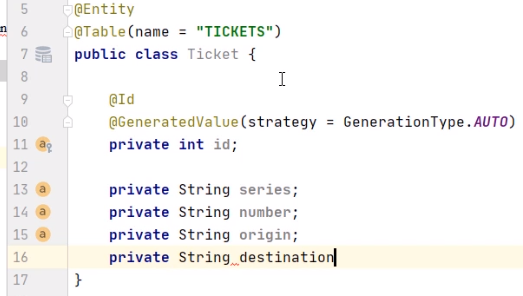
The GeneratedValue annotation is supported only for simple primary keys. It specifies a generation strategy for the primary key value and is used in conjunction with the id annotation. It may be applied to a persistent field or property. The GeneratedValue annotation may receive optional parameters, the strategy to use and the name of the generator to use.

# Demo: Primary Keys with @GeneratedValue

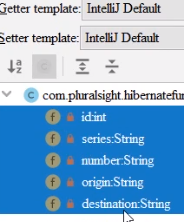
=>slides: Pg. 28

We move to another practical demonstration to show an example that uses the creation of a primary key. More exactly, the primary key will be created using the GeneratedValue annotation.





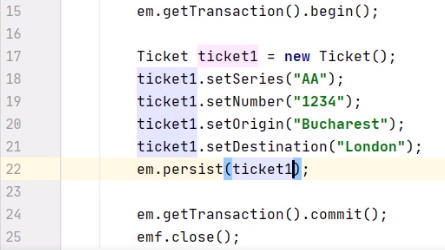
We'll demonstrate now how the primary key of the Ticket class can be created using the GeneratedValue annotation. We start with an already existing Maven project for which pom.xml and persistence.xml are configured similarly to what we presented in the previous demonstration. We made sure we only changed the persistence‑unit and the database name. We'll start the demonstration by creating the Ticket class that will serve as the entity to work on. We annotate the class with Entity. The corresponding table in the database for the Ticket entity will be the TICKETS table. We add the id field on the Ticket and annotate it with the Id annotation, as it is the primary key. The GeneratedValue annotation will configure the way of incrementing the column. By specifying the AUTO strategy, which anyway is the default one, we'll let the persistence provider choose the generation strategy. We'll add now the fields shaping the ticket, the series, and the number, the origin, and the destination. We will no longer annotate them with the Column annotation in order to demonstrate what will happen in such a case.



We generate the getters and setters for the series, number, origin, and destination fields.



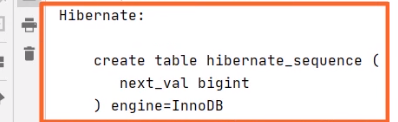
We move to the Main class that has already been configured to create the EntityManagerFactory and the EntityManager the way we saw in the previous demonstrations.

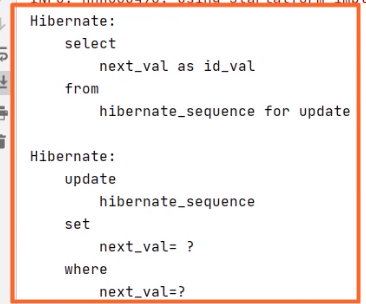


We also begin and commit the transaction and close the EntityManagerFactory. We'll need, of course, to write the code to create the Ticket and configure it using the previously created setters, and we'll need to persist the Ticket.



On the MySQL database, we need to create the database with the same name as in the persistence‑unit from the Persistence.xml file, so we execute the command, CREATE DATABASE M03\_EX04. And now we may run the Java program. We'll create the needed TICKETS table, we'll create the sequence that manages the primary key field, and we'll use this sequence to insert the information from our object.



Hibernate will generate a table called hibernate\_sequence to provide the next number for the id field. 

When inserting a new role, Hibernate will generate the next value that will be used for the id column in the Tickets database.



We go back to the MySQL database and commute to our database by executing the USE M03\_EX04 command. Let's check the content of the TICKETS expected table. We execute SELECT \* FROM TICKETS, and we have the ticket information inside the table. We notice that the id field has been automatically generated, and it took the first available value of 1, and as we did not annotate the fields inside the Ticket class using the Column annotation, the default values of the names of the fields were used, and the columns appear now as series, number, origin, and destination, so we proved the primary key of a class can be created using the GeneratedValue annotation.

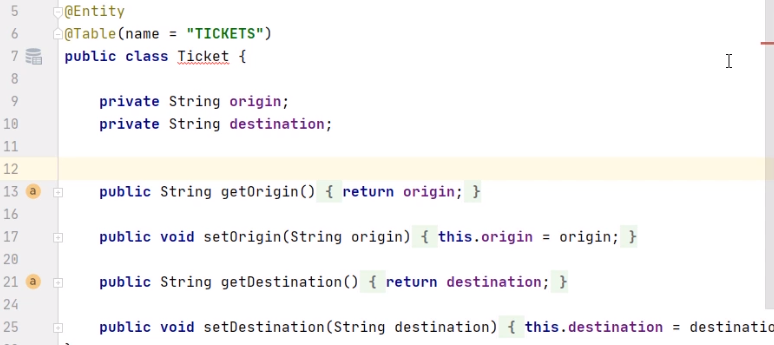
# Demo: Embeddable Primary Key and Embedded ID

=>slides: Pg. 19

We move to another practical demonstration to show an example that uses the creation of a primary key, and more exactly, the creation of an imbeddable primary key and of an imbedded ID.



We start with an already existing Maven project for which pom.xml and persistence.xml are configured similarly to what we presented in the previous demonstrations. We made sure we only changed the persistence unit and the database name. We'll also reuse parts of the previously created Ticket class and of the Main class that has already been configured to create the EntityManagerFactory and the EntityManager to begin and commit the transaction and close the EntityManagerFactory.



This Ticket class that needs to serve as the entity to work on already has the origin and destination fields. It needs an embedded key to contain the series and the number of the ticket.

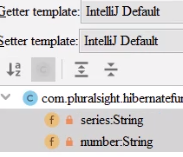
We create the TicketKey class that will serve as an embeddable ID.



We annotate this class with @Embeddable to mark that it will be embedded in some other entity.



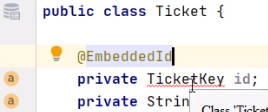
The TicketKey class contains as fields the series and the number,



and we'll generate the getters and setters for these fields.



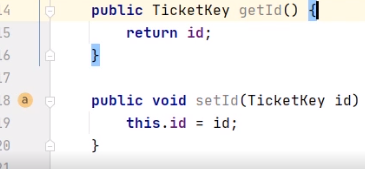
Back to the Ticket class,



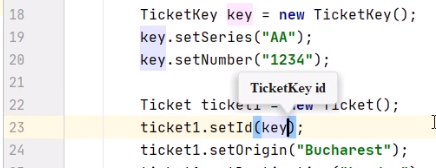
we'll declare a private TicketKey id field, we'll annotate it as @EmbeddedId because it will embed a composite primary key into the entity. We notice now a warning from the IDE saying that the TicketKey class should implement Serializable. This is required as the ID may be used as a key at the second level cache.



So, we go to the TicketKey class and mark it to implement Seralizable. Back in the Ticket class, the warning disappeared.

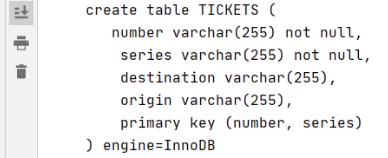


We generate the getter and the setter for the ID field. The Main class already created the ticket, but this ticket still needs the configuration of the field to provide the primary key.



We create a TicketKey instance. We configure it using the previously defined setter methods. And we set the ticket ID to be key.



On the MySQL side, we need to create the database with the same name as in the persistence unit from the persistence.xml file. So, we execute the command CREATE DATABASE M03\_EX05. And now, we may run the Java program. We'll create the needed TICKETS table. 

If we look at its fields, we see that it contains both the fields from the Ticket class, origin and destination, and the fields from the embedded ID, series and number.



We go back to the MySQL environment and commute to our database by executing the USE M03\_EX05 command. Let's check the content of the TICKETS expected table. We execute SELECT \* FROM TICKETS. And we have the ticket information inside the table containing four fields, origin and destination coming from the Ticket class, series and number coming from TicketKey. So, we proved how the primary key of a class can be created using an embeddable primary key and an embeddable ID.

# Demo: Embeddable Primary Key and ID Class

=>slides: Pg. 30

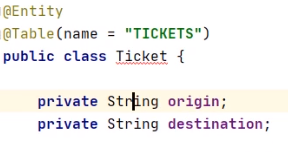
We move to the next practical demonstration to show an example that uses, again the creation of a primary key. More exactly, the creation of an embeddable primary key and of an ID class. We'll demonstrate now how the primary key of the Ticket class can be created using an embeddable primary key and an ID class.



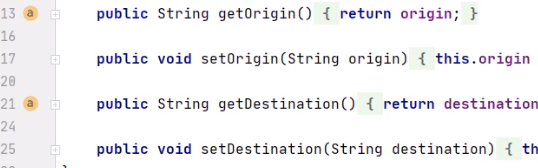
We start with the current Maven project, as it looked like at the end of the previous demonstration. The pom.xml and persistence.xml are configured similarly to what we presented in the previous demonstrations. We made sure we only changed the persistence unit and the database name.



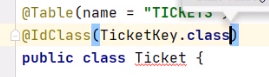
This TicketKey class served as an embeddable ID. It was annotated with embeddable to mark that this class will be embedded in some other entity, and we'll reuse it the way it is this time to become an ID class.



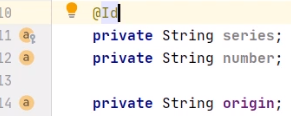
This Ticket class contained a TicketKey id serving as an embedded ID. We removed this field that will no longer be needed in the case of the ID class.



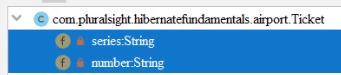
We'll also remove the getter and the setter for it.



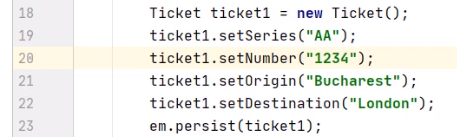
We'll annotate the Ticket class with the @IdClass annotation, having TicketKey.class as argument. This will mean that the fields from the TicketKey class will need to be mirrored in the Ticket class and will serve as a composite primary key.



We'll add in the Ticket class the private String series and the private String number field. We'll use the @Id annotation before the declaration of these two fields.



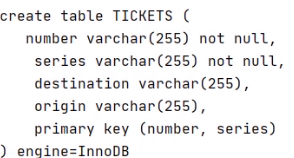
We'll generate the getter and the setter for the series and number fields.



We move to the Main class, and we need to fix the code according to the changes we made for the Ticket class. We'll remove the declaration of the ticket key instance, and it will no longer be used as an embedded ID. And, of course, we'll directly set the field serving as composite primary key for the ticket, the series, and the number.



On the MySQL side, we have already created the database with the same name as in the persistence unit with the help of the CREATE DATABASE M03\_EX06 command. We commuted to it by executing the USE M03\_EX06 command. And now, we may run the Java program. We'll create the needed tickets table.



If we look at its fields, we'll see that it contains both the fields from the Ticket class, origin and destination, and the fields from the ID class, series and number. The composite key of the TICKETS table is formed from the series and number columns. We go back to the MySQL environment. Let's check the content of the TICKETS expected table.



We execute SELECT \* FROM TICKETS. And we have the ticket information inside the table containing four fields, origin and destination coming from the Ticket class, series and number coming from the ID class ticket key. We remind that we did not annotate any field using the @Column annotation. So Hibernate used the default convention for generating the names of the columns. So, we proved how the primary key of a class can be created using an embeddable primary key and an ID class.

=>slides: Pg. 31

To conclude this module, let's have a look at what we covered so far. We defined what an entity is, and we analyzed its particularities. We demonstrated how to map objects. We examined the entity access types, field based or property based. We defined entity primary keys and entity identity using @GeneratedValue as annotation, an embedded primary key and an embedded ID, and an embeddable primary key and an ID class. Our next goal will be to work with entity relationships. See you in the next module.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*