2. Introducing ORM and Its Problems

Contents

[1. Introducing ORM and Its Problems 1](#_Toc19484)

[2. The Granularity Problem 3](#_Toc31283)

[3. The Inheritance Problem 4](#_Toc29465)

[4. The Identity Problem 4](#_Toc28695)

[5. The Associations Problem 5](#_Toc11900)

[6. The Data Navigation Problem 6](#_Toc19937)

[7. Overview of ORM tools 7](#_Toc12264)

[8. A Simple Hibernate Application 8](#_Toc6496)

# 1. Introducing ORM and Its Problems

=>slides: Pg. 1

Welcome to this Java Persistence Hibernate Fundamentals course. It targets Java developers who are already proficient in writing Java core code, have basic knowledge of the Apache Maven build tool, with good understanding of the database principles and that want to learn how to easily access databases from a Java program. So let's get started and first see what an ORM means and which its main problems are.

=>slides: Pg. 2

This course will show you how to write efficient applications that access a database from Java code using the Hibernate framework. This introductory module will first review the concepts of ORM, object‑relational mapping, and JPA, Java Persistence API. We'll examine the advantages and drawbacks of Hibernate as JPA implementation. We'll analyze the object‑relational impedance mismatch and the mismatch problems, granularity, inheritance, identity, associations, and data navigation . We'll finalize with a demonstration to create a simple Hibernate application with a PostgreSQLdatabase.

=>slides: Pg. 3

ORM, standing for object‑relational mapping, is a technique that converts data between an object‑oriented programming language and the relational database management system. These are incompatible type systems, as one is working with objects and their capabilities, while the second one is working with relations and their capabilities. The essential problem requires storing the representation of the objects so that they can be saved in a database while keeping the information from inside and querying the relational database and rebuilding the object after retrieving the information. The successful save and retrieve of the object to and from the database makes the object to be called persistent.

=>slides: Pg. 4

JPA, standing for Java persistence API, is the Java specification for object‑relational mapping and persistence. Hibernate is the most widely used framework implementing the JPA specification. Hibernate may use either XMLs or annotations in order to specify the mapping logic from the object‑oriented world to the relational databases world. Essentially, we have to map the classes and the fields of an object to tables and columns.

=>slides: Pg. 5

You may ask yourself, why should you use JPA and the framework like Hibernate that implements the JPA specification? You may know, or you may even have used the JDBC, Java Database Connectivity application programming interface, that successfully works with databases from within a Java application. Which are the advantages of using JPA and Hibernate? First, it will allow you to write less code. You will be working mostly on the side of the objects from the program, and you will have much less interaction with the database from within the Java program. Writing less code will mean that you will be able to boost your performance. You will do quicker development within the same timeframe. Then it will except you from necessary knowing SQL. Most of the Java programmers have at least some basic knowledge of the Structured Query Language. Using Hibernate, this knowledge will be even less required. Anyway, you will be able to see the SQL queries that Hibernate will generate while executing the program and eventually get more knowledge from them. You will be provided a consistent model for interacting with the database. As we were saying, you will work mostly on the side of the objects from the program, and you will be able to focus more about building the business logic, while the persistence burden will be on the side of the Hibernate framework. Last but not least, you will be independent of the database vendor. Your code will be portable, and Hibernate will translate it to be understood by the database individually for each vendor. If you would still like it, you will be able to use vendor‑specific features.

=>slides: Pg. 6

And now you may ask yourself, are there only advantages? Everything comes like a gift, just to take it? Of course not, and there are some potential drawbacks as well. Once you introduce Hibernate as JPA framework, you will have something more and something new to learn. You will first need a good command of the framework itself, and then you will be able to efficiently put it in practice. Then as your system is passing through an additional layer, it will be harder to debug. Problems may now occur in one more place. One more layer will also mean some delay in the execution, so performance may suffer in this case. There may be situations, especially for simpler applications, when you would like to still use JDBC as being closer to the database itself. Or you may desire in some situations to use the specific features of a vendor database, in which case you may also want to avoid passing the interaction with the database to a framework. Overall, for long‑time situations, we consider that it is really worth getting the knowledge to use a powerful framework as Hibernate. Its capabilities and advantages will pay back your effort.

=>slides: Pg. 7

The object‑relational impedance mismatch means that object and relational models do not work fine together. Object‑oriented programming keeps data as interconnected objects with fields and methods while an RDBMS keeps data as related tables. The object‑relational impedance mismatch can be divided into a few problems that we are going to examine separately. granularity, inheritance, identity, associations, and data navigation.

# The Granularity Problem

=>slides: Pg. 8

We discuss now about the granularity problem.

=>slides: Pg. 9

Let's presume that we are building a flights management application. We need a class to represent information about a passenger and a class to represent information about tickets. A passenger may be the owner of one or more tickets.

=>slides: Pg. 10

The classes representing this model may look like this, the Passenger class with A name and address and with a list of tickets, and the Ticket class with a number and a passenger. We care here only about the fields inside the classes and not about their behavior.

=>slides: Pg. 11

If we translate this in the relational model, the table structure may look like this, the table PASSENGERS with NAME and ADDRESS and primary key (NAME), the TICKETS table with the NUMBER and PASSENGER\_NAME and primary key (NUMBER), and the constraint between the passengers and the tickets. The foreign key FK\_PASSENGERS in TICKETS represents the relationship between the two tables.

=>slides: Pg. 12

So far, so good. The step from the 00P world to the RDBMS world was pretty straightforward. But what if we extend a little the initial diagram, and we would like to keep inside the address fields at street, number, zipCode, city, and country. Is an address table necessary? Usually not. In most situations, we keep the address information inside the passengers table using individual columns. This design does not use any join, so it will perform better. Most database vendors do not allow to create a new SQL data type to represent the newly introduced class address, so most likely you will add to the table the columns representing the street, the number, the ZIP code, the city, and the country. This is the granularity problem. The sizes of the types we are working with are different.

=>slides: Pg. 13  
So most likely we'll end up with an updated PASSENGERS table looking like this, A NAME, ADDRESS\_STREET, ADDRESS\_NUMBER, ADDRESS\_ZIPCODE, ADDRESS\_CITY, ADDRESS\_COUNTRY, then NAME as primary key.

# The Inheritance Problem

=>slides: Pg. 14

We'll discuss now about the inheritance problem, the second problem of the object‑relational impedance mismatch.

=>slides: Pg. 15

Inheritance is one of the characteristics of the object‑oriented programming and it is not well represented in the relational model. Let's presume that there are a few possible types of tickets, OneWayTicket, ReturnTicket, or RoundTripTicket. This situation may be represented in our program using this class's hierarchy. Looking on the side of the relational model, we cannot create a OneWayTicket table to extend the Ticket table. The Passenger class has an association to the ticket superclass. This is a polymorphic association. We keep a list of tickets on the side of the Passenger, but at runtime, a Passenger may reference an instance of any of the subclasses of Ticket, OneWayTicket, ReturnTicket, or RoundTripTicket. We'll need to write polymorphic queries that refer to the Ticket class, and the query will return instances of its subclasses. Relational databases do not possess a standard way to represent polymorphic association. Once you define a foreign key, it will refer exactly one target table, you cannot have the foreign key referring multiple tables. We'll examine the possible solutions provided by Hibernate to the inheritance problem and to the polymorphic associations.

# The Identity Problem

=>slides: Pg. 16

We'll discuss now about the identity problem, the third problem of the object‑relational impedance mismatch.

=>slides: Pg. 17

In our first approach, we define NAME as the primary key of the PASSENGERS table. If you are accustomed with the way tables and primary keys are defined, you may know that in most cases you will choose a column that is certainly unique, or you may create a special column to play the role of the primary key, sometimes also called a surrogate key, while the name has less chance to really be unique. A surrogate key column is a primary key column having as its unique purpose to identify the data inside the table. Otherwise, a surrogate key has absolutely no meaning to the application logic.

=>slides: Pg. 18

How do the object‑oriented model and the relational model represent uniqueness? Java uses two notions of equality‑‑objects identity, when two references point to the same object, and we check this using the == operator, logical equality when two objects may represent different references in memory, but they are equal from the logical point of view. You check this using the equals method. On the side of the relational database, the identity of two database rows is expressed by their primary keys. As we see, neither of the two ways Java verifies equality is connected to the way this happens inside the relational database. It is not impossible that several non‑identical instances in Java represent the same role of a database. This may happen by querying the database twice or in concurrently running application threads.

=>slides: Pg. 19

In the first table definition for PASSENGERS, NAME was the primary key. This approach will make it difficult to change the name of the passenger. Once you update it, this change will propagate in the TICKETS table. This problem may be solved by the use of surrogate keys, explained a little earlier. With this approach, the tables may be changed to look like this PASSENGERS with an ID integer not null and primary key and a separate NAME column. The table TICKETS with an ID integer not null and primary key and two separate NUMBER and PASSENGER\_ID columns. There is a constraint between the tables PASSENGERS and TICKETS. The foreign key (PASSENGER\_ID) is referenced by the PASSENGERS (ID) column. The ID columns were introduced only to be used by the data model. Should we represent them in the Java domain model? We'll discuss this while working with entities, and we'll find a solution with ORM.

# The Associations Problem

=>slides: Pg. 20

We discuss now about the associations problem, the fourth problem of the object‑relational impedance mismatch.

=>slides: Pg. 21

In the OOP model, associations represent the connections between classes. The passenger, address, and ticket classes are all associated.

=>slides: Pg. 22

On the side of the relational model, only the TICKETS entity is a separate one. The information originally contained in an address object will fully arrive to the PASSENGERS table as we previously stated. In the relational model, a foreign key constraint column represents an association, effectively copying the key values. Such a constraint intends to ensure the integrity of the association.

=>slides: Pg. 23

The OOP model represents associations using references to objects.

=>slides: Pg. 24

Joins and navigation in a given direction have no particular meaning for the relational model, we just follow the association given by the foreign key.

=>slides: Pg. 25

In the object oriented‑model, object references are directional, following the Has‑A relationship. Most of the relationships are unidirectional. If you would like a relationship to be bidirectional, you need to define the association twice, once on each side.

=>slides: Pg. 26

Java associations can be one‑to‑one. Here, a passenger has one ticket and a ticket is owned by one passenger.

=>slides: Pg. 27

One‑to‑many. Here, a passenger has a list of tickets and the ticket is owned by one passenger.

=>slides: Pg. 28

Or many‑to‑many. Here, a passenger has a list of tickets and a ticket has a list of passengers.

=>slides: Pg. 29

In the relational model, the foreign key declaration of the TICKETS table is always a many‑to‑one association. One passenger can have multiple tickets. One ticket may belong to only one passenger. =>slides: Pg. 30

Representing a many‑to‑many association in the relational model will require the introduction of a new table. Such a link table will look like this. The link table PASSENGERS\_TICKETS will have a PASSENGER\_ID field and the TICKET\_ID field, and the primary key will be composed of PASSENGER\_ID and TICKET\_ID. This is reflected in the FK\_PASSENGERS and FK\_TICKETS constraints.

# The Data Navigation Problem

=>slides: Pg. 31

We discuss now about the data navigation problem, the fifth problem of the object relational impedance mismatch.

=>slides: Pg. 32

Data navigation involves working through objects in the object‑oriented model

=>slides: Pg. 33

and joining tables in the relational model.

=>slides: Pg. 34

In Java, if you would like to iterate through the tickets of a passenger, you can write something like for(Ticket ticket: passenger.getTickets()). This is not efficient in the relational model. What we can better do in this case is to minimize the number of queries executed against the database and to use conditions and joins between tables. If we would like to retrieve a record from the PASSENGERS table, but we are not interested by the tickets in his possession, we can execute a query like SELECT \* FROM PASSENGERS WHERE ID = 727423. If we would like to retrieve a record from the PASSENGERS table and we are also interested by the tickets in his possession, we can execute a query like SELECT \* FROM PASSENGERS, TICKETS where PASSENGER.ID = 727423 AND PASSENGERS.ID= TICKETS.PASSENGER\_ID. To use joins in an efficient way, we have to know the part of the object network we need to access when we retrieve the initial instance before effectively navigating the object network. If we retrieve too much data, we may waste memory. The Cartesian product used when making joins is an expensive one. Executing such a Cartesian product on two tables, each having 1,000 rows, will result in 1,000,000 rows retrieved, 1,000 x 1,000.

# Overview of ORM tools

**Hibernate**

Hibernate is a Java persistence framework that simplifies the development of Java application to interact with the database. It is an open source, widely used, lightweight, ORM tool. Hibernate implements the specifications of [JPA](https://www.javatpoint.com/jpa-introduction) (Java Persistence API) for data persistence.

**TopLink** is an ORM tool developed by Oracle especially for Java developers. It is a persistence framework that is a part of Oracle's OracleAS, WebLogic, OC4J servers. It provides development tools and run-time functionalities that ease the development process and increases the functionality. Persistent object data is stored in relational databases that helps build high-performance applications. Storing data in either [XML](https://en.wikipedia.org/wiki/XML" \t "https://www.javatpoint.com/_blank) (Extensible Markup Language) or relational databases is made possible by transforming it from object-oriented data.

**OpenJPA**

Apache OpenJPA is a Java persistence project at [The Apache Software Foundation](https://www.apache.org/" \t "https://www.javatpoint.com/_blank) that can be used as a stand-alone [POJO](https://en.wikipedia.org/wiki/POJO" \t "https://www.javatpoint.com/_blank) persistence layer or [integrated](https://openjpa.apache.org/integration.html" \t "https://www.javatpoint.com/_blank) into any Java EE compliant container and many other lightweight frameworks, such as Tomcat and Spring.

**MyBatis**

MyBatis was formerly known as iBatis. It is also an open source persistence framework that simplifies the implementation of database. Basically, it performs the following four things:

It executes SQL safely and abstracts away all the intricacies of JDBC.

It maps parameter objects to JDBC prepared statement parameters.

It maps rows in JDBC result sets to objects.

It is different from other ORM tools. The main difference between MyBatis and other ORM tool is that MyBatis emphasis the use of SQL while other ORM tools uses custom query language (HQL).

**EclipseLink**

EclipseLink is the open source Eclipse Persistence Services tool introduced by Eclipse Foundation. It is an extensible framework that allows Java developers to interact with various data services such as databases, web services, Object XML mapping, and enterprise information systems. It supports the following four persistence standards:

JPA (Jakarta Persistence)

JAXB (Jakarta XML Bindings)

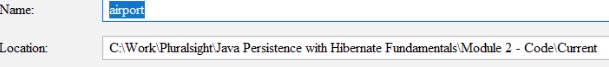
JCA (Jakarta Connectors)

SDO (Service Data Object)

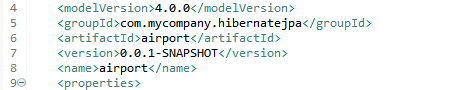
# A Simple Hibernate Application

=>slides: Pg. 35

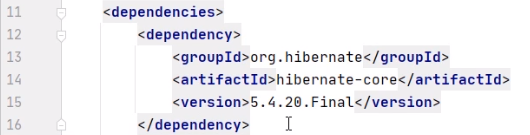
We would like to provide a first taste of Hibernate and we'll create a Java project, manage it with the help of Maven, create the entity classes, and demonstrate how the objects are persisted to the database. We create now a new project with the help of STS, and we are going to manage it with the help of Maven.



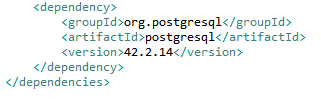
We choose here a Maven project, we name it airport, we press here Finish, we wait for the project to be loaded.. Here we have the pom.xml file from Maven,

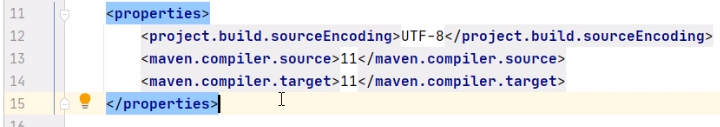


and we are changing here the groupId to become .pluralsight.hibernatefundamentals. We'll keep the artifactId to be airport. The next thing that we are going to do is to add the dependencies.

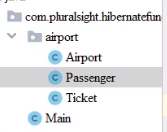


Of course, we are going to need Hibernate dependency and the PostgreSQLdependency. We added here the dependency org.hibernate, hibernate‑core, and we are going to choose the latest stable version, which is 5.4.20.



And we are going to add here the PostgreSQLdependency, and we are going to choose 8.0.20 as version. We are also adding the source code to be 11, Java 11 version, and we move our attention to adding source code.

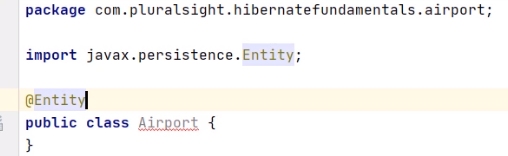




We create here a package called com.pluralsight.hibernatefundamentals,

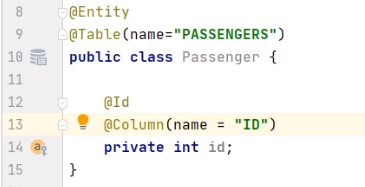


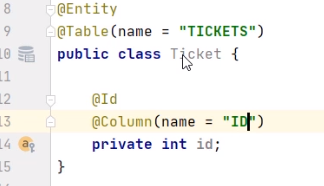
and here we are going to add our main class.



A new package, we call it airport, and here we are going to add a few classes. First, the Airport class and the other entity classes.



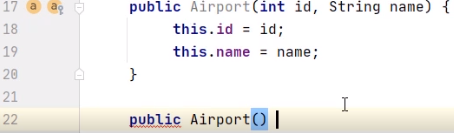


  
We take this Airport class and we annotate it with Entity. This is the way to mark that it is an entity, and it has a corresponding table into the database, and we name this table AIRPORTS, in plural. This class has a private int id field, and we are going to annotate it with Id, marking it is a primary key, and the name of the column that is corresponds to. And the name of the column will be ID. We mark this Access type to be AccessType.field, meaning that Hibernate will make the persistence at the level of the fields. We create a new class, and we call it Passenger. Of course, we need to annotate it with Entity, marking that it has a corresponding table into the database, and the name of the table is PASSENGERS. It has a private int id field, that it will be marked with the annotation Id, and it will correspond to the column in the database with the name ID. We add a new class called Ticket. This is corresponding to the tickets that our passengers will have. And of course, we need to mark it as Entity and also to annotate it as table with the name of the corresponding table to be TICKETS in plural. And we are adding the private int id field. Of course, we need to mark it with the Id annotation to tell it is a primary key and to tell that it has a corresponding column with the name ID.

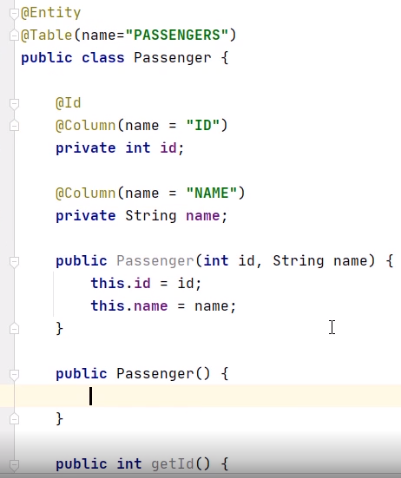
Now we get back to the Airport class, and we start to add more fields.

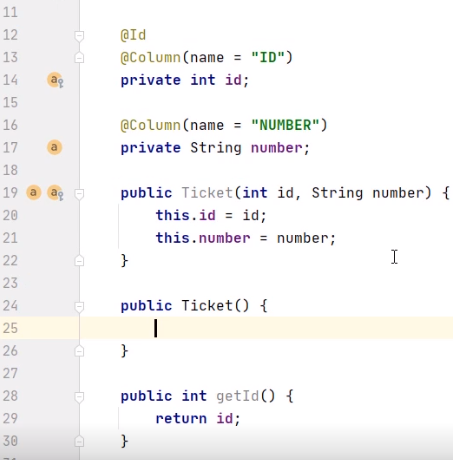


First, we are adding this name field, and we tell that it is corresponding to the column NAME. Then we are going to generate the getter and setter methods, both for the id and name fields.



And we are going to generate a constructor with the id and name as arguments. We are going also to add a constructor with no arguments because all entities will need constructors with no arguments.

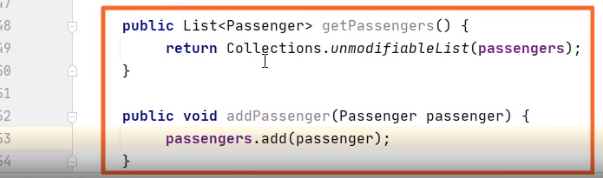


We move to the Passenger class, and we add a field called name, getter, setter, and the constructors. 

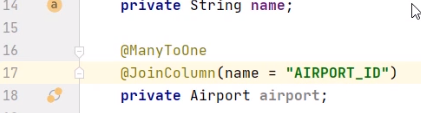
Then to the Ticket class we add a new field called NUMBER, corresponding to the column number, and for it we had getter, setter,

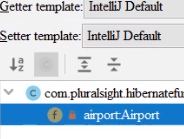


and we also add the constructors. In the Airport class, we add a list of passengers, and we annotate it with OneToMany, indicating that the relationship is mapped by the airport.



We add a getPassengers and add passenger method, and we move to the Passenger class.



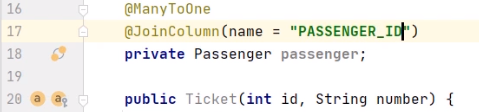
We'll insert a private Airport airport field and annotated with ManyToOne, indicating that many passengers use the same airport. The JoinColumn will have the name AIRPORT\_ID. 

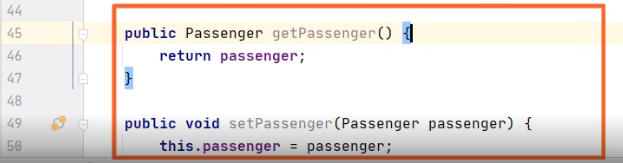
We generate the getter and the setter for this new airport field.

  
Then we add on the side of the passenger the list of the tickets.



We annotate this list with OneToMany, indicating that one passenger may have many tickets. The relationship is mapped by the passenger field on the ticket side.



In the class ticket, we add the field passenger and we annotate it with ManyToOne, indicating that one passenger may have many tickets. The JoinColumn will have the name PASSENGER\_ID, and we 

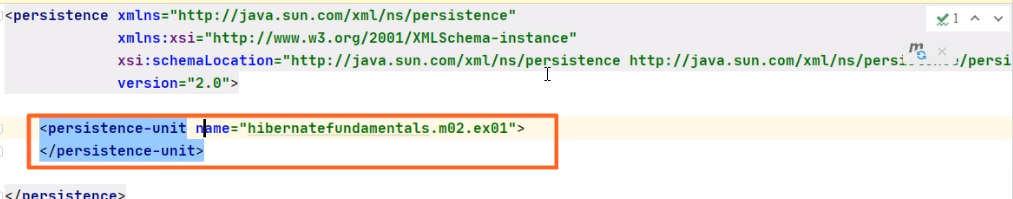
generate the getter and the setter for the Passenger. We need to provide the metainformation about how the persistence is made.



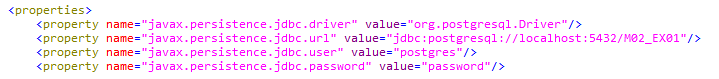
Into the resources folder, we create a new directory META‑INF.

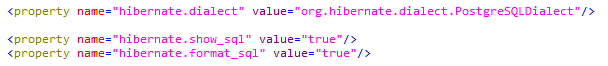


Inside this directory, we create the persistence.xml file. This is a standard name that Hibernate will consider in order to get the metainformation about the persistence.

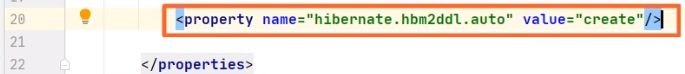


We provide the name of the persistence unit. The persistence will be backed by a Hibernate provider.

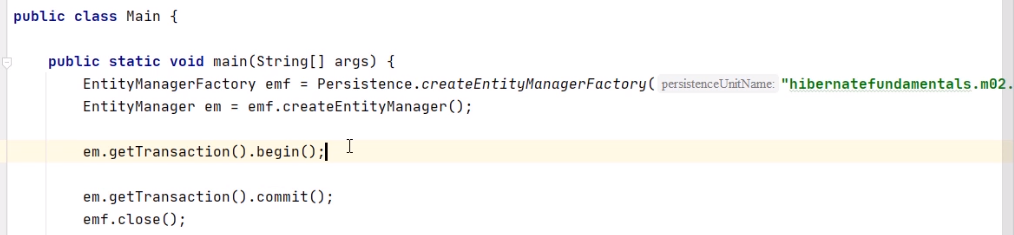
We'll indicate here the JDBC properties, the driver, the URL, the user name, and password to access the database.



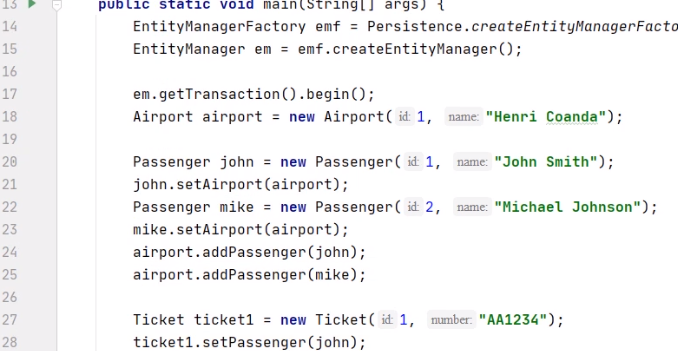
We indicate the hibernate.dialect to be my PostgreSQL. While executing, we'll show the SQL code in a formatted way.



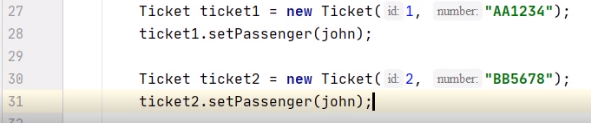
Every time we execute the program, the database will be created from the scratch.



We go back to the main class, and we create here the main method where we'll create our objects and we'll persist them in the database. First we'll 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 entity manager with the help of the Create EntityManagerFactory method. Then we begin the transaction. We'll write from now the end of our code that we'll need to commit the transaction, and that will close the EntityManagerFactory.

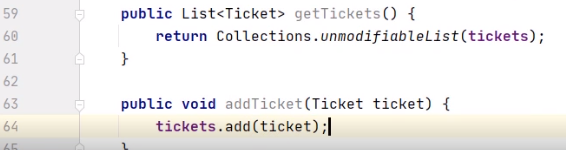


Now, we move to the code working with our objects. We create an airport having id 1 and with the name of Henri Coanda. We create a passenger with id 1 and the name of John Smith, and we set his airport. We create the passenger with id 2 and the name of Michael Johnson, and we set his airport. We add the passengers John and Mike on the list of passengers of the airport.

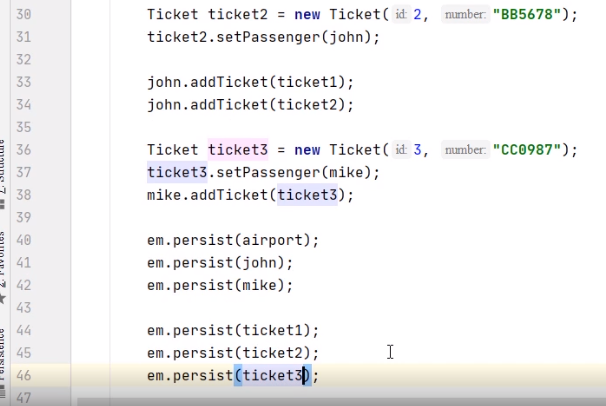


We create Ticket 1 and Ticket 2, and we need to go to the





Passenger class and add the getTickets and addTicket methods. The getTickets method will return an unmodifiableList of the tickets, as from outside the class we do not allow access to the original list. The addTicket method will add the Ticket 2 the passengers list of tickets.



Back in main, we add ticket1 and ticket2 on John's list of tickets. We create ticket3 with id 3, and we set the passenger for it. We add ticket3 on Mike's list of tickets, and we persist all created objects one by one, meaning that we insert into the database a corresponding record with their information. We persist the airport, we persist John and Mike, and we persist ticket1, ticket2, and ticket3 one by one.



Now we go on the side of the PostgreSQLdatabase that we have on the machine. We need to create the database that we work with, having the same name as indicating in the persistence unit. So we execute the command, create a database M02\_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 PostgreSQLdatabase and commute to our database by executing the USE M02\_EX01 command. Let's check the content of the three tables.



We execute SELECT \* from AIRPORTS. And yes, we retrieve the airport with the ID 1 and the name Henri Coanda. Let's verify the other two tables. We execute SELECT \* from PASSENGERS. We expect John and Mike to be there, and yes, we have the two passengers inside the table. Let's finally verify the TICKETS table.



We execute. SELECT \* from TICKETS. We expect three tickets to be there. And yes, we have the two tickets belonging to John and one ticket belonging to Mike. So this concludes our demonstration.

=>slides: Pg. 36

To conclude the first module of our journey with Hibernate, we take a brief look at what we covered so far. We introduced object‑relational mapping and Java Persistence API. We met Hibernate, its advantages and drawbacks. We examined in detail the problems of object‑relational impedance mismatch. Finally, we created the first Hibernate application to interact with a PostgreSQLdatabase. Our next goal will be to work with entities, map the tables to objects, and analyze the entity's access types.

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