Implementing the Database Layer

Prof. Dr. Peter Braun

17. Oktober 2024

1 Titlepage

Implementing the Database Layer

Prof. Dr. Peter Braun

thws Technical University of Applied Sciences Würzburg-Schweinfurt

No Text – maybe some music :-)

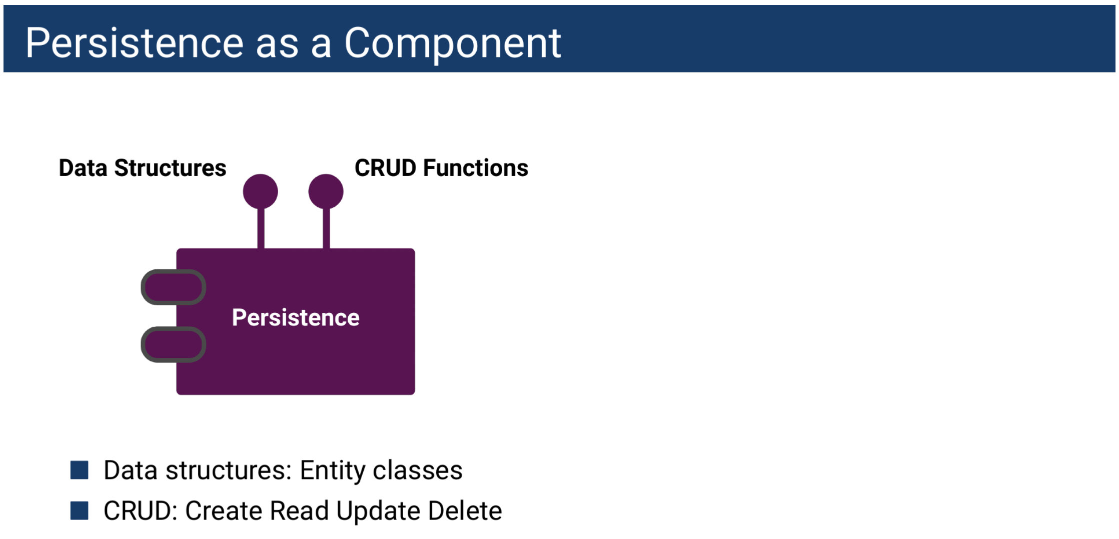
2 Learning Goals

Learning Goals

- You know JPA and Hibernate
- You can model database entities
- You can implement the CRUD operations
- You can model 1 to n relations

Welcome to this unit about how to implement a persistence layer in Java applications. I assume that you all have learned about relational database management systems already. You know how to use, for example, MySQL, and you can write SQL commands like select and insert statements. Maybe you have also learned about how to connect a Java application to MySQL using the JDBC technique, where you embed SQL code in Java code. Today, I will introduce a modern approach named JPA to connect a Java application to a database. JPA stands for Jakarta Persistency API. It is a technique where you can avoid writing plain SQL commands and instead only use Java classes and methods to write queries, for example. We will use Hibernate as one implementation of the JPA specification. At the end of this unit, you will be able to model data entities using Java classes, and you can implement all CRUD operations. CRUD stands for Create, Read, Update, and Delete and is the technical term that summarizes all database operations. In the end, I will shortly demonstrate how to model 1 to many relations in JPA.

3 Persistence as a Component



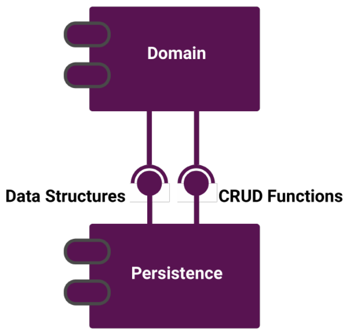
Persistence as a Component

- Data structures: Entity classes
- CRUD: Create Read Update Delete

Persistence is a key concept to understand when discussing the database layer in computer science. Persistence refers to the capability of saving, managing, and retrieving data from your system. When focused particularly on the database layer, it involves how the data remains intact and accessible even after the termination of a session. The essence of persistence is often associated with data structures, specifically entity classes. These are defined in your system to represent the schema of your tables, where each entity class corresponds to an individual table in your database. These classes hold the data and define the types of data variables or attributes and their interrelationships. To ensure persistence, your system must be equipped with the CRUD operations, which are Create, Read, Update, and Delete. These operations are the key activities within any database system. 'Create' introduces new data or records into the database, comparable to the INSERT statement in SQL. 'Read' corresponds to retrieving the data from the database, like the SELECT statement. 'Update' represents modifications you wish to make to the existing data. Finally, 'Delete' implies removing the selected data or records from the database. The update and delete commands are named identically in SQL. We are talking about software architecture right now. What is persistence in software architecture, then? I want you to think of it as a component. The persistence component has a clear job description: manage the data and make it persistent. Therefore, the two interfaces of this component are the data structures and the CRUD operations.

4 Using the Persistence Component

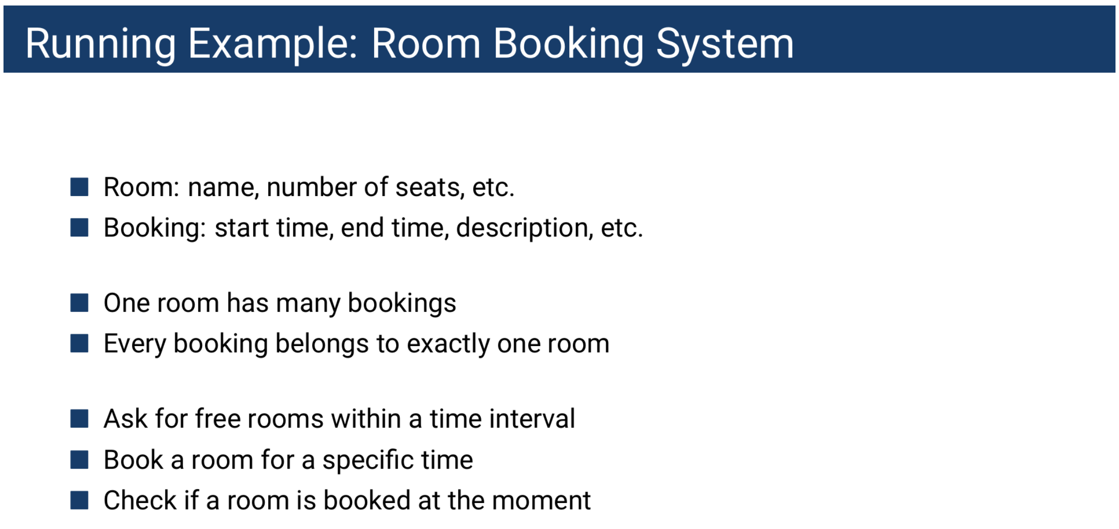
Using the Persistence Component



- Domain layer depends on the Persistence layer
- We show a solution by adapters later

The client of the persistence component is the Domain layer. In software design, we frequently observe a dependency of the Domain layer on the Persistence layer. Now, what does that mean? The Domain layer, which contains the core business logic, often needs to communicate with the Persistence layer to retrieve or store data in a database. The Domain layer uses the data structures that the Persistence component provides and the CRUD operations that are part of the interface of the Persistence component. Please don't complicate things here; think of it at the Java class level. The Persistence component defines a Java class to model a, let's say, University and a second class with methods that provide the CRUD operations. Now, the Domain layer or Domain component uses this very class named "University," which is provided by the Persistence component. We have two kinds of dependency here: First, the Domain component depends on the Persistence component because it is used. This type of dependency is inevitable to make the system work. The second type of dependency is on the level of data types. When the Domain component uses the Java class named "University" from the Persistence component, then the Persistence component is not free to change the name of this class or any object attributes because it directly influences the Domain layer. This is where we will introduce the concept of adapters later in this course. These adapters can act as bridges between the Domain and the Persistence layers. They allow us to modify the interface of one class to be used as another interface.

5 Running Example: Room Booking System



Running Example: Room Booking System

- Room: name, number of seats, etc.
- Booking: start time, end time, description, etc.

- One room has many bookings
- Every booking belongs to exactly one room

- Ask for free rooms within a time interval
- Book a room for a specific time
- Check if a room is booked at the moment

For this introduction to Java persistence, we will use a running example of a system to manage and book rooms in a university building. Think of rooms as lecture halls or conference rooms. The fundamental elements of a Room Booking System are Rooms and Bookings. A Room contains specific attributes such as its name and number of seats. Similarly, a Booking contains different variables, like the start time, end time, and a brief description. The relationship between these elements is quite straightforward - each room can have numerous bookings. However, it is vital to note that every booking must be associated with one and only one room. Implementing the database layer of such a system requires a few important functions. The first is a command to check for free rooms within a specified time interval. This operation ensures that a room isn't double-booked, maintaining integrity in the booking process. The second crucial function is the ability to book a room. This function involves creating a new booking, assigning it to a room, and storing the start and end times, among other details. Lastly, a function needs to check if a room is currently occupied. This function simplifies real-time bookings and provides users with immediate information about vacancies.

6 Introduction to Jakarta Persistence API (JPA)

Introduction to Jakarta Persistence API (JPA)

- JPA is a specification (current version 3.2)
- Hibernate is only one implementation
- Others: EclipseLink, TopLink, Open JPA (Apache)
- Try not to use Hibernate-specific features

- Advantage: you can get rid of embedded SQL
- You don't work with table and column names
- The source code becomes more readable

We now introduce the Jakarta Persistence API, or JPA for short. JPA, currently in its 3.2 version, is essentially a specification for interfacing with databases in Java. It isn't a tool or a library in and of itself; rather, it defines a set of rules or guidelines that tools or libraries can follow to interact with databases. One of the implementations of this specification might be familiar to you—Hibernate. However, it's important to note that Hibernate is just one of many implementations available. Other popular implementations of the JPA specification include EclipseLink, TopLink, and Open JPA, which were developed under the Apache software umbrella. When working with JPA, it's advisable to avoid implementation-specific features. For instance, while Hibernate might offer certain unique functionalities, leaning too heavily on these can lead to tighter coupling of your code with Hibernate, which detracts from the implementation independence that JPA is designed to provide. This is one of the main obstacles to learning JPA from tutorials or popular Websites like Stackoverflow. There, JPA and Hibernate are mixed, and you might have problems finding the correct answer. One of the most significant advantages of using JPA is its ability to abstract away the need for embedded SQL in your Java code. JPA allows you to manipulate and query databases using plain Java objects and methods instead of SQL statements, thus making your codebase cleaner and more maintainable. For example, you're no longer working directly with table and column names in your code. This shifts the focus from the specificities of your database schema to the more general concepts of entities and relationships. The final result is a significant improvement in the readability, maintenance, and extendibility of your source code.

7 JPA Project Set-up

JPA Project Set-up

Dependencies

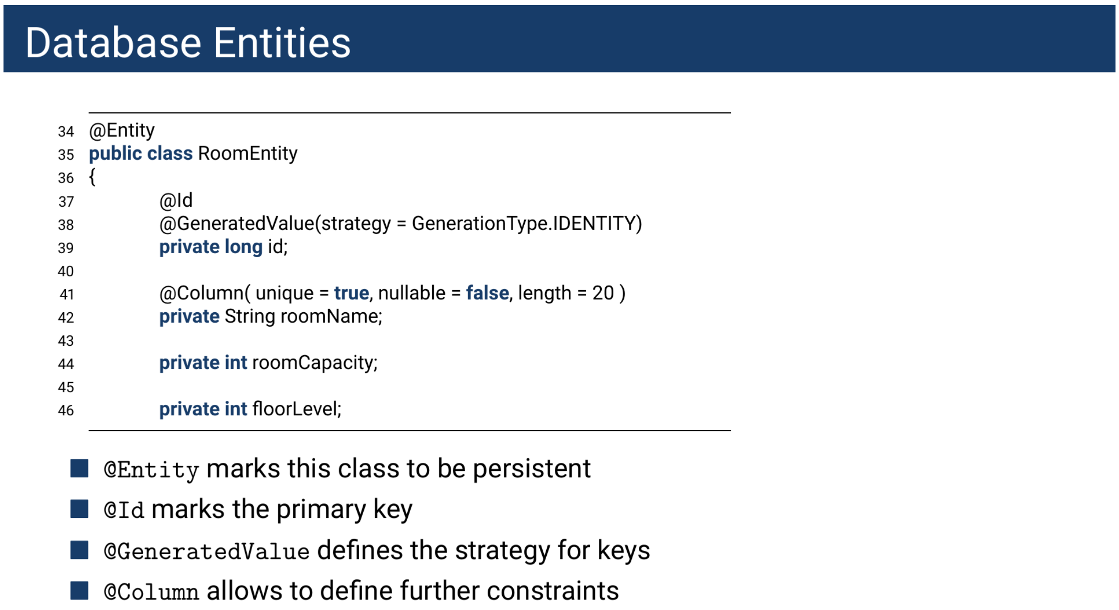
- Jakarta Specification
- Hibernate library
- H2 database
- HikariCP

Configuration

- File `main/resources/META-INF/persistence.xml`
- Configures JPA, Hibernate, and HikariCP

We will focus on setting up a JPA project for this part of our unit. As shown in the last unit, we will use Maven as a build tool. A template for a JPA project is on our GitLab server. The link should be published in the Moodle course. In a project that uses JPA, we first need the library containing the JPA specification. A key component of our project will be the Hibernate library. Our chosen database for this project is the H2 database. It is known for its lightweight and fast operations, presenting an ideal choice for our learning application. It will run in embedded mode, offering a simple yet effective database system with which to interface. In a production system, we would not use H2 but, for example, MySQL or Postgres. Installing and setting up a database is not part of this unit. HikariCP, a high-performance JDBC connection pooling library, will be integrated into our project to optimize database connectivity. This library ensures more efficient use of resources within our application for web traffic management and system scaling. Our primary configuration file, persistence.xml, will be in the main resources' directory under META-INF. It defines all our database settings and essentially links our Java application and the H2 database. This file, persistence.xml, configures JPA, Hibernate, and HikariCP. It contains properties such as the data source and database connection details. It provides Hibernate with necessary instructions and connection details to the H2 database through JPA, while HikariCP leverages this to manage database connections efficiently.

8 Database Entities



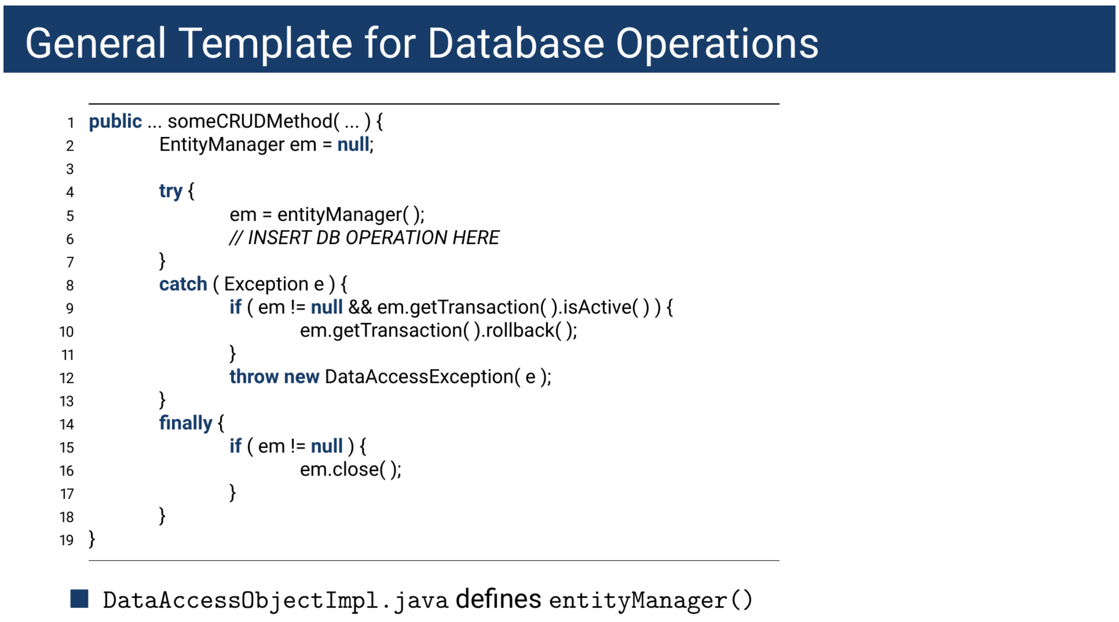
Database Entities

```
34 @Entity
35 public class RoomEntity
36 {
37         @Id
38         @GeneratedValue(strategy = GenerationType.IDENTITY)
39         private long id;
40
41         @Column( unique = true, nullable = false, length = 20 )
42         private String roomName;
43
44         private int roomCapacity;
45
46         private int floorLevel;
```

- `@Entity` marks this class to be persistent
- `@Id` marks the primary key
- `@GeneratedValue` defines the strategy for keys
- `@Column` allows to define further constraints

We're delving deeper into understanding the essential elements of the database layer, particularly Database Entities. When we speak of Database Entities, the cornerstone of it all is the @Entity annotation, which you can see in line 34 beside me. By the way, the source code that I show here on the slides comes from the demo application published on our GitLab server. The link to the Git repository should be published in the Moodle course. This "Entity" annotation marks any Java class as persistent. Then, we come to the @Id annotation. This one marks the primary key of our entity — the unique identifier every entity is required to have. Now, to control the generation of this primary key, we have the @GeneratedValue annotation. This annotation helps define the overall strategy for our keys, handling automatic, behind-the-scenes generation every time a new entity is created. Next on our list is the @Column annotation. With this, we get to define additional constraints for our columns in the database. We can manage properties, like specifying if a column can be nullable or deciding the column's length. This should be enough for the basic annotations needed in entity classes. In practice, you will need many more to fine-tune the mapping of Java classes to database tables.

9 General Template for Database Operations



General Template for Database Operations

```
 1 public ... someCRUDMethod( ... ) {
 2         EntityManager em = null;
 3 
 4         try {
 5                 em = entityManager( );
 6                 // INSERT DB OPERATION HERE
 7         }
 8         catch ( Exception e ) {
 9                 if ( em != null && em.getTransaction( ).isActive( ) ) {
10                         em.getTransaction( ).rollback( );
11                 }
12                 throw new DataAccessException( e );
13         }
14         finally {
15                 if ( em != null ) {
16                         em.close( );
17                 }
18         }
19 }
```

- `DataAccessObjectImpl.java` defines `entityManager()`

We continue with the general outline of all CRUD operations. The persistence component provides a Java class with at least four methods that implement the four CRUD operations. All CRUD operations share some common code but differ in the details, of course. What you can see next to me is this common structure. In line 5, we call a method named “entityManager,” defined in this class, but the source is not shown. This method uses JPA to get the entity manager, the root class to work with a database. The entity manager manages the entities from the database, sends commands to the database, and processes the result. Beginning in line 6, the implementation differs from one CRUD operation to the other, and we will show some examples later. During the processing of these database operations, an exception could be thrown that must be caught in line 8. If transaction management was necessary for this database operation, this is the place to roll back the transaction and throw a new database exception to inform the domain layer about this problem. In all cases, the entity manager must be closed in line 16, which cleans up everything.

10 Create an Entity in the Database

Create an Entity in the Database

- Method `void create( RoomEntity roomEntity)`

```
22  em = entityManager( );
23  em.getTransaction( ).begin( );
24  em.persist( roomEntity );
25  em.getTransaction( ).commit( );
```

- The primary key is set in object `roomEntity`

Okay, let's now look at some examples how to implement the CRUD operations in detail. We start with the "create" operation. The interface defines a method that accepts a parameter of type "RoomEntity." We only need for lines of code now to store this entity in the database. In line 22 we request an entity manager. In line 23 we start the transaction. In line 24 we only call method "persist" to store the entity. It is this method that creates an SQL CREATE statement and sends it to the database. And finally, in line 25, we commit the transaction. Remember that we configured this entity to let the database define the primary key. Because the primary key is of type long, the database will use a counter for this which is increased row by row. The "id" assigned by the database layer is stored in the object "roomEntity" so that the domain layer can continue to work with this id if necessary.

11 Load an Entity by Primary Key

Load an Entity by Primary Key

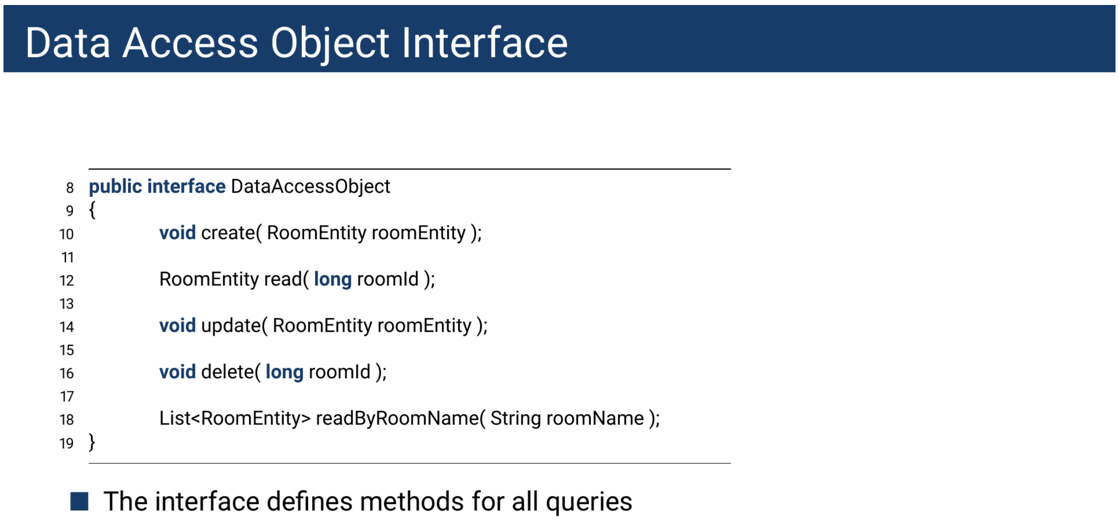
- Method `RoomEntity read( long roomId )`

```
em = entityManager( );
final RoomEntity result = em.find( RoomEntity.class, roomId );
return result;
```

- If the primary key is invalid, the return value is `null`

The next example shows loading an entity by its primary key. Remember, in a relational database, a primary key is an attribute or a set of attributes used to identify records in a table in a unique manner. When we talk about loading an entity by its primary key, we are fetching details or attributes of a single record using its unique identifier - the primary key. The persistence component provides a method for this named "read" and the "id" as a parameter. When you invoke this "read" method, the functional job of this method is to interact with the database, use the "find" method of the Entity Manager, and pass the provided "roomId" as a parameter to find the corresponding "RoomEntity," load that entity, and finally return it. It's important to understand the return possibilities for this method. If the primary key provided, in this case, the `roomId`, is invalid or doesn't correlate to any existing entity, the return value is `null`. No exception is thrown in this case in contrast to your expectations.

12 Data Access Object Interface



Data Access Object Interface

```
public interface DataAccessObject
{
	void create( RoomEntity roomEntity );

	RoomEntity read( long roomId );

	void update( RoomEntity roomEntity );

	void delete( long roomId );

	List<RoomEntity> readByRoomName( String roomName );
}
```

- The interface defines methods for all queries

Here, you can see the complete functional interface of the Persistence component. The name of this interface is "Data Access Object," which is also the name of a pattern we applied here. The pattern goes like this: Let's split the entities from the database functions. Keep both classes separated because it makes it easier to exchange the concrete database, but keep the data structures unmodified and reuseable. The Data Access Object interface operates as a contract between our database access code and the rest of our application, namely the Domain layer. The domain layer does not know how persistence is implemented; it does not even know what kind of database system is used. All technical details are transparent to the domain layer. The domain layer only knows and must only know these 4+ CRUD operations. These could be queries to create, read, update, or delete data. The interface must provide further methods to query data. This is where it might become complicated. How do you know what kind of queries the persistence component must provide? Well, that's the result of a deep analysis of the requirements and negotiations with the domain layer.

13 Filter by Attributes

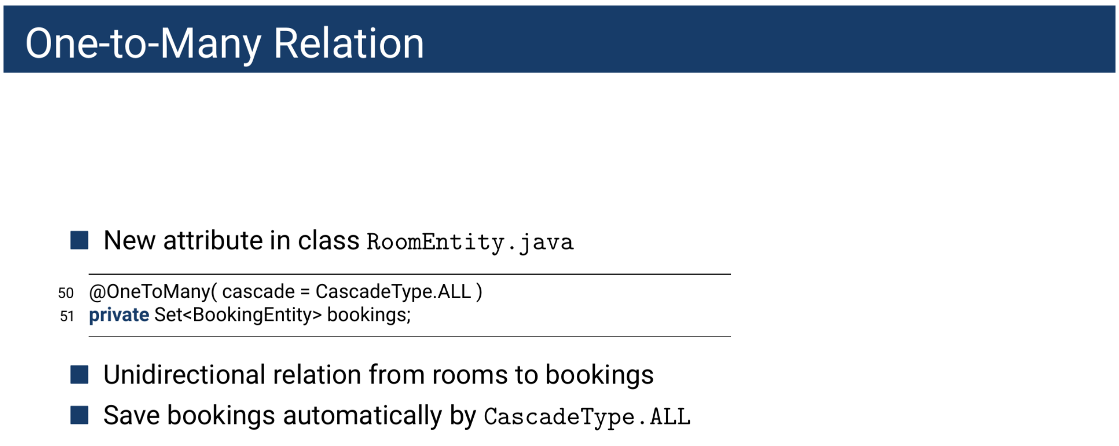
Filter by Attributes

- Method `List<RoomEntity> readByRoomName( String roomName )`

```
70 em = entityManager( );
71
72 final CriteriaBuilder cb = em.getCriteriaBuilder( );
73 final CriteriaQuery<RoomEntity> cq = cb.createQuery( RoomEntity.class );
74 final Root<RoomEntity> root = cq.from( RoomEntity.class );
75
76 final Predicate matchRoomName = cb.like( cb.lower( root.get( "
       roomName" ) ), roomName.toLowerCase( ) + "%" );
77 final CriteriaQuery<RoomEntity> filterByNames = cq.select( root ).where(
       matchRoomName );
78 final TypedQuery<RoomEntity> allQuery = em.createQuery( filterByNames
       );
79
80 return allQuery.getResultList( );
```

Implementing these kinds of complex queries is not so easy. When you write pure SQL commands, it results in a long SELECT statement with many filter conditions. In JPA, you can do everything with Java, but the source code is a bit longish. The source code beside me demonstrates the implementation of a simple filter by room name query. In line 72, we create the criteria builder, the base class for building queries. When we start defining the query in line 73 using the method "createQuery," we must pass the class name of the entity we expect. Here, you can see that we don't deal with table names; we only deal with class names. In line 74, we define the root entity from which we use attributes in the filter. This is the same class used to create the query in simple cases. But think of complex filters using "joins." Then, these are different classes. In line 76, we create a filter condition asking for the column used to store the object attribute "roomName," converting this to lowercase, and compare this using the "like" operator to the given parameter "roomName." We use the percentage sign as a placeholder here. In line 77, we create the final query using "select" and "where" and execute the query in line 78. Finally, in line 80, the result is returned. It is not difficult if you read it line by line, but it is a bit longish, as I said.

14 One-to-Many Relation



One-to-Many Relation

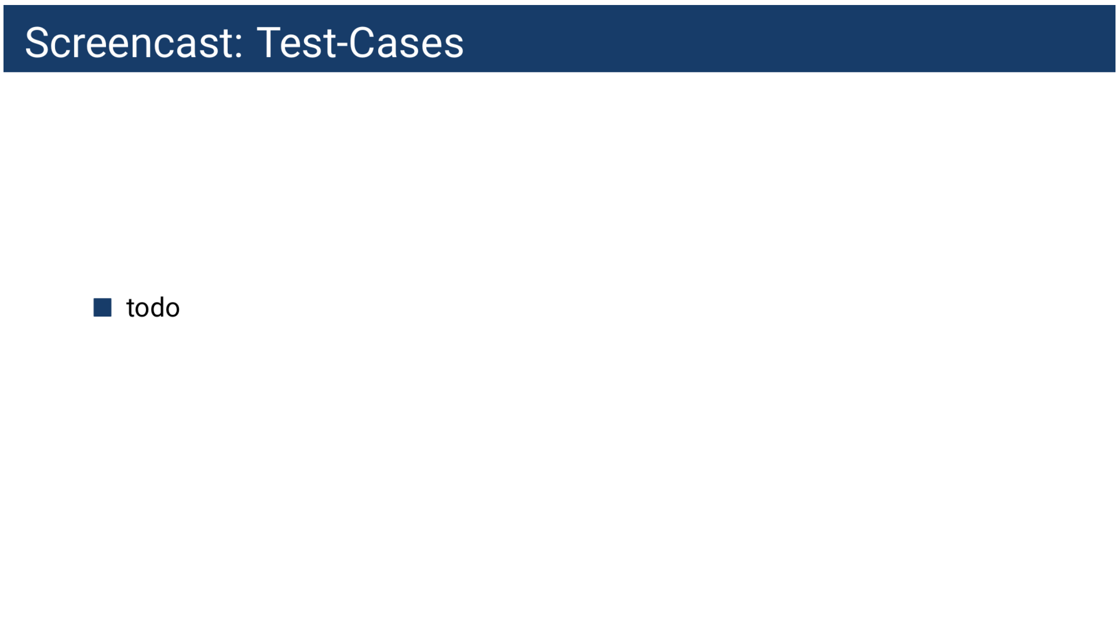
- New attribute in class `RoomEntity.java`

```
50 @OneToMany( cascade = CascadeType.ALL )
51 private Set<BookingEntity> bookings;
```

- Unidirectional relation from rooms to bookings
- Save bookings automatically by `CascadeType.ALL`

As the last topic for this unit, we are moving forward to discuss the One-to-Many relation. In the context of a simple room booking system for a university, let us consider how rooms can be related to bookings. We want to create a scenario where one room can have multiple bookings, but a single booking is specifically attached to one room only. Hence, we are dealing with a One-to-Many relation: one 'Room' to Many 'Bookings'. The implementation is surprisingly easy. It begins by defining a new attribute in our main class, "RoomEntity." The new attribute is named "bookings" and is a set of Booking objects. The annotation used here is called "OneToMany," and that would be enough to implement the most simple case. I will explain the parameter used in the annotation later. What we have defined here is a 'Unidirectional' relation. This implies that while a room knows its bookings, a booking does not hold knowledge about its room. From the room's perspective, there can be an array of bookings, but from a booking's perspective, it is simply an entity devoid of any relation. When we want to save such a relationship, we add the booking to the room's set of bookings and call the RoomEntity Data Access Object's "update" method. Unfortunately, it's not that easy, because we have to persist the booking manually before. That means we need a new CRUD method to persist a booking; call this method. Then, we can add this booking to the set of bookings of a room entity and update the room. This sounds too complicated. Fortunately, we can shorten this by adding the parameter "Cascade" to the annotation, as shown in line 50. Cascading is a technique where JPA or Hibernate will do whatever is necessary with related entities to be successful. That means we can add a new booking entity to the set of all bookings of a room and then simply save or update the room. JPA will find out that the booking was not created before and then create it in the database on the fly.

15 Screencast: Test-Cases



Screencast: Test-Cases

- todo

I would like to show you the template project for JPA projects. This is the one that is published on our GitLab server and you can use this as a starting point for your own JPA projects. First we have here the POM file. Let's start with the POM file. We have some more properties defined here. This is meant for the Jakarta JPA version here, for the Jupiter JUnit version, then for the database H2 version number and for the connection pool here. We only have to add a few more dependencies. This is the one for Jakarta, the one for Hibernate, the specific implementation of Jakarta, Persistency API, then the database itself. Then we have here a library that I often use in my projects. It's called Java, Faker. This is a nice little project to get some random data to just populate a database, for example, and the connection pool. And here in the plugin section there is nothing new, but we can fix one error here. This is wrong. So let's go to the models. We start with the room entity. This is what I showed you on the slide already. We have the properties here. We have this one-to-many relation here. And the rest is just getter and setter method. The most important line is here, this annotation at entity. Booking entity looks very similar. This is just an example here. You can also use other strategies or other types of ID. Can use string with a hash value, for example, not a long value. And the operations are implemented here. So the interface, as shown on the slide, just contains these CRUD operations. Then we have the implementation. So we start here with the create implementation. We have this typically try catch finally block. Because we have here transactions activated. And in case of an exception, we have to roll back everything here. And then at the end, you always have to close the connection to the entity manager. Method like read. Then this kind of a little bit too complex or let's say more complex method to do some filtering. The update method here and the delete method. The main class, this is just for setting up things when you just start this from the command line.

But actually, what I wanted to show you is the test case. I implemented some test cases here. So in this setup method that is called before each single test case, we initialize this data access object implementation object. And then we have test cases for create, which is like we create a room, we create a booking, and we add the booking to the room. And then here we call this create method from the DAO implementation. And what we check here is just very simple. You can extend this, of course, by more complex assertions. We just test here that there is a valid ID set. And we delete everything here. This deleting, I think, happens in every single test case. It would have been better to have this as a here, this after each method as well. So then there are test cases. Read through the test cases. This is very simple and this is by far not exhaustive, not complete. You have to add for your projects here definitely more test cases.

16 Summary

Summary

- You know how to define database entities
- You know how to implement database operations
- You know how to write test cases

That's it for today. You have learned today how to set up a Java project for using JPA and Hibernate. I showed you all the necessary Java annotations to make Java classes as entities. We discussed the general outline of database methods when using the Data Access Object pattern. You have seen a few examples of how to implement these methods in detail, and you should read the source code of the other methods in the project published for demonstration. Finally, I have shown you how to implement test cases for the persistence component. By this, you should now be able to implement a persistence component by yourself based on a given set of requirements.