Java Advanced – lesson 3 – Spring Data

Wim Wiltenburg

(willem.wiltenburg@inholland.nl)

Objectives

- Data
- Hibernate
- Entity
- Repository
- Query methods
- Live Demo

Persistence

What is an object:

- Data structure
- Has state (data)
- Behavior (methods)

We work with objects all the time, therefore we handle data all the time, but that data has to come from somewhere, and has to go somewhere.

We store data if we want to reuse it later -> persistence

Many types of data

We handle many types of data:

- Domain data – The data that our application is about
- Operational data – How does our application perform
- Session data – Who is using our application and what are they doing
- …

Data can be:

Fleeting	Persistent
Secret	Open
Structured	Unstructured
Time sensitive	The oppositie ☺
Or any combination thereof	

How to store data

There are a number of ways to store data:

- Memory
- File(s) on hard drive
- Database
- Combination of these

Use what fits your needs, not just what you think is cool, fun, trending or familiar

Databases

Databases can be of two types:

- Relational
  Examples: Oracle, MySQL, Postgres, DB2
- Non-Relational
  Examples: MongoDB, Cassandra, Key-Value stores, Graph Databsae

Databases are usually a tradeoff between internal consistency and performance and/or scalability

Relational Database problem

When we restrict our scope to relational databases we see that there is a discrepancy between the internal structure of an object and its relations (object graph), and the way it's represented in the database using normalization. This is called object-relational impedance mismatch. For example:

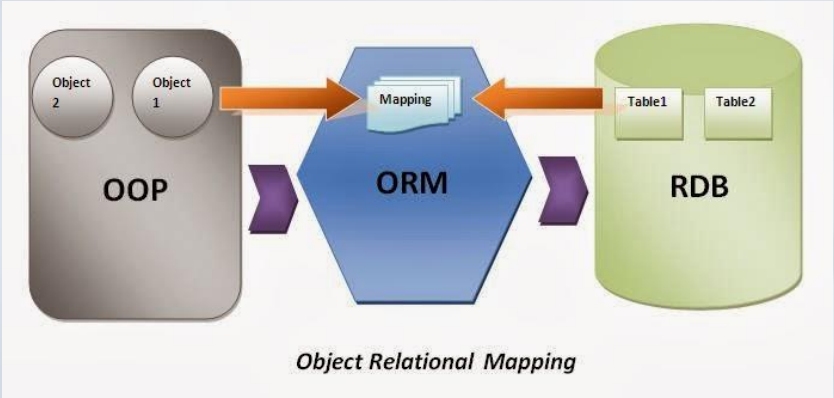
A Person object can contain an address object. A database can only contain rows with just integers, strings, and relations to other tables.

Problem: how to translate object state to database and vice versa

Object Relational Mapping

A solution to make the translation between objects and tables in the database is Object Relational Mapping (ORM).

It's an abstraction layer between application and database, that takes the object data and maps it to data that the database can handle… And vice versa!



Object Relational Mapping

Hibernate

The most widely used framework for ORM is Hibernate (by Red Hat):

- Open Source
- Translates object graph to normalized data. One to One, One to Many, Many to Many, etc.
- Provides a language HQL (Hibernate Query Language) for writing SQL queries
- Is integrated in the Spring Suite using Spring Data

Getting started with Hibernate

To start using ORM you have to include the following dependencies in our POM.xml:

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency
   <!– een in memory database →
   <groupId>com.h2database</groupId>
   <artifactId>h2</artifactId>
   <version>1.4.199</version>
</dependency>
```

The model

The ORM framework recognizes the objects that needs mapping to the database with class annotation:

```
@Entity
```

After that you need to design your mapping with a variety of possibilities:

- Id
- Column properties
- Mapping OneToOne or OneToMany, etc.

@Id

There are different ways to generate an id. First annotate the id field with @Id. The two most common methods:

- Easiest way is to just annotate the id with @GeneratedValue
  It creates an id starting at 1, incrementing with 1
- If you want something special, like a sequence starting from a number or a maximum range:
  @GeneratedValue(strategy=GenerationType.SEQUENCE)
  But you need a sequence generator on class level:
  @SequenceGenerator(name= , initialValue= , allocationSize=)
  That name goes into the @GeneratedValue annotation:
  @GeneratedValue(strategy=GenerationType.SEQUENCE, generator="seq")

@Column

The mapping of each field in the object can be configured with the @Column annotation. Then some properties can be assigned. The most common are:

- Name
- Nullable
- Unique
- Length

@Repository

Hibernate calls the data structure that holds the data of your objects a repository. To start using these repository, you have to create an interface that extends a Hibernate interface, like:

- CrudRepository<T, ID>
- PagingAndSortingRepository<T, ID>

Note: the T stands for your type of Object e.g. Guitar. The ID stands for the datatype of the id, wrapped if necessary:

```
@Repository
public interface GuitarRepository extends CrudRepository<Guitar, Long> {

}
```

@Component

A @Repository class is a @Component class.

@Components are scanned at startup and the Spring Container instantiates and configures them, and is responsible for the life cycle management. The Spring Container then returns that object for you to use.

So at startup there will be only 1 repository of that kind, up until the end of the application

CrudRepository

When you define a CrudRepository, you get a lot of functionality out of the box. Common methods are:

- <S extends T> S save(S var1);
  This saves a T the that database, or a subtype thereof
- Iterable<T> findAll();
  Creates a collection of all objects in the database of that type. This collection can be iterated over, and is sometimes cast as List<T>

Let's take it for a spin

Demo

JACK'S GUITAR SHOP

Strings attached

How to prevent duplicates

When you save an object that has most of the same attributes of other objects, that can lead to duplication in your database.

Example:

GET-request

```
{
"id": 1,
"brand": "Fender",
"model": "Stratocaster",
"price": 1750
}
```

POST-request

```
{
"brand": "Fender",
"model": "Stratocaster",
"price": 2000
}
```

Result:

```
{
"id": 6,
"brand": "Fender",
"model": "Stratocaster",
"price": 2000
}
```

Unique constraints

Apart from the ID that must always be unique, it's possible to apply additional constraints on table level:

```
@Table(uniqueConstraints =
{@UniqueConstraint(columnNames = {"brand",
"model"})})
```

This will result in an internal server error, but your data will retain its integrity::

{ "timestamp": "2019-05-05T11:57:07.004+0000", "status": 500, "error": "Internal Server Error", "message": "could not execute statement; SQL [n/a]; constraint [\"PUBLIC.UKJIQPK147GWJ0TYBRCNLELSORI_INDEX_7 ON PUBLIC.GUITAR(BRAND, MODEL) VALUES 2\"; SQL statement:\ninsert into guitar (brand, model, price, id) values (?, ?, ?, ?) [23505-199]]; nested exception is org.hibernate.exception.ConstraintViolationException: could not execute statement", "path": "/guitars"}

Foreign Keys

To let entities depend upon eachother you can apply relationships one foreign keys. They come in four flavors:

- @OneToOne
- @OneToMany
- @ManyToOne
- @ManyToMany

The annotation is applied to the object that is “composed” in the class.

@OneToOne

When you create a foreign key relation from one object to another, you can apply the @OneToOne annotation on the enclosing object. Example:

```
@Entity
public class Stock {

    @OneToOne
    private Guitar guitar;
```

If you want the relation to be two way, you can also annotate the other class with the parent class:

```
@Entity
@Table(uniqueConstraints = {@UniqueConstraint(columnNames = {"brand", "model"})})
public class Guitar {

    @OneToOne
    private Stock stock;
```

Customizing your query

CrudRepository provides custom methods for generic operations that you can use, that internally translates to an actual query depending on your model.

- You can start your query method names with find...By, read...By, query...By, count...By, and get...By. Before By you can add expression such as Distinct . After By you need to add property names of your entity.
  Example:

```
public Iterable<Stock> getAllByQuantityGreaterThanEqualOrderByQuantityDesc(int min);
```

- To get data on the basis of more than one property you can concatenate property names using And and Or while creating method names.
- If you want to use a completely custom query for your method, you can use @Query annotation to write query.

Query

@Query allow you to write your own SQL query. It annotates a specific method you wish to execute.

Example:

```
@Query("SELECT g.price * s.quantity from Guitar g, Stock s where s.guitarId = g.id and g.id = ?1")
public int getStockValueById(long id);
```

- Parameters are prefixed with a question mark. ?1 is the first parameter (long id). If there are more parameters, numbers go up ?2, ?3 etc.
- Objects are named, convention says first letter. Guitar becomes g, Stock becomes s
- Type the objects and name first. That will make it easier to retrieve the column names

Assignment

Refactor your application to use Spring Data with H2 database.

Add:

- Crud methods
- Secondary object
- @Query method