Using JPA in Spring

ABIS Training & Consulting
www.abis.be
training@abis.be

Document number: 1783_02.fm
1 October 2024

Address comments concerning the contents of this publication to:
ABIS Training & Consulting, Diestsevest 32 / 4b, B-3000 Leuven, Belgium
Tel.: (+32)-16-245610

TABLE OF CONTENTS

PREFACE VII

INTRODUCTION TO ORM AND (SPRING) JPA 9

1 Persisting objects 10

2 Object - relational mapping (ORM) 12

3 Features of persistence mechanisms 15

4 (Spring) JPA 17
- 4.1 JPA Architecture 18
- 4.2 Spring's JPA integration 20

5 Spring (Boot) - Recap 21
- 5.1 Spring's Philosophy 21
- 5.2 Ingredients 22
- 5.3 Tools 23
- 5.4 Spring Boot 24
- 5.5 Spring Boot Application - Example 26

CONFIGURATION 29

1 Spring JPA: Basic Configuration 30

2 Configuring a data source 31
- 2.1 General 31
- 2.2 Maven dependencies 32

3 Data Source Types 33
- 3.1 Embedded Data Source 33
- 3.2 Basic Data Source 34
- 3.3 Pooled Data Source 36
- 3.4 JNDI Data Source 38

4 Simplifications from Spring Boot 39

5 Extra configuration options 40

6 Testing The Connection 42

BASIC OR MAPPING AND JPA REPOSITORIES 43

1 Mapping concepts 44

2 Class to table mapping 45

3 Object identity 46
- 3.1 Object identity - mapping 47
- 3.2 Composite primary key 49

4 Property mapping 54
- 4.1 Type conversion - built-in Hibernate types 56
- 4.2 Property mapping - derived properties 58

5 JPA repositories 59
- 5.1 General 59
- 5.2 Repository Interfaces 60
- 5.3 Query Methods 62

6 Spring Data JUnit Testing 64

MAPPING ASSOCIATIONS 65

1 Value types 66

2 One class for two tables 68

3 Association mapping 69
3.1 One-to-one mapping 70
3.2 Many-to-one mapping 75
3.3 One-to-many mapping 77
3.4 Many-to-many associations 81

4 Additional options 84
4.1 Cascade strategy 84
4.2 Fetching strategy 85

QUERYING IN SPRING JPA 87

1 @Query 88

2 JPQL 89
2.1 Basic JPQL 90
2.2 Selection: From 91
2.3 Restriction: Where 92
2.4 Joins in JPQL 94
2.5 Parameter binding in JPQL 95

3 Named queries 96

4 Native queries 97

5 Changing the "select" object 98
5.1 Aggregate functions 99
5.2 Limiting/Changing selected columns 100

6 Using DTOs and mappers 101
6.1 Data Transfer Objects (DTOs) 101
6.2 Custom Mapper Class 102
6.3 @NamedNativeQuery and @SqlResultSetMapping 104
6.4 Mapping JPQL query results 105

TRANSACTION MANAGEMENT AND EXCEPTION HANDLING 107

1 Exception Handling 108

2 Modifying queries 109

3 Understanding transactions 110
3.1 Spring's Transaction Philosophy 111
3.2 Transaction Managers 112

4 Transaction Status 113

5 Transaction Definition 114

6 Declarative transactions 117

7 Programming transactions 120

ADVANCED TOPICS 123

1 Mapping collections 124
1.1 Set 125
1.2 List 126
1.3 Map 127

2 Hierarchy mapping 130
2.1 Table per class hierarchy 131
2.2 Table per concrete class 136
2.3 Table per subclass (joined) 140
2.4 Choose a hierarchy mapping strategy 142

APPENDIX A.EXERCISES 143

1 Configuration 143

2 Entity mappings and Basic JPA repositories 143

3 Association Mapping and @Query 144

4 More querying techniques and performance 144

5 ____ Exception Handling and Transactions_____ 145

6 ____ Advanced Topics_____ 145

PREFACE

Java applications accessing relational databases can use native JDBC calls, or are based on a good object-relational mapping O/R Mapping) solution.

The standard solution has been provided for the Java platform through the **Java Persistence API** (JPA). **Spring Boot Data JPA** will provide easy ways to integrate JPA to perform database access via Java Repositories

In this course, we will take a closer look at how to map our entities onto relational DB tables. We will also focus on the different query mechanisms. Special attention will go into some performance topics, like the use of DTOs and fetching strategies.

More information can be found in:

- https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#reference
- PLURALSIGHT
  - https://www.pluralsight.com/courses/java-persistence-api-21
  - https://www.pluralsight.com/courses/spring-data-jpa-getting-started
  - https://www.pluralsight.com/courses/data-transactions-spring
- BAELDUNG
  - https://www.baeldung.com/spring-data-rest-relationships
  - https://www.baeldung.com/hibernate-lazy-eager-loading
  - https://www.baeldung.com/hibernate-fetchmode
  - https://www.baeldung.com/spring-data-crud-repository-save
  - https://www.baeldung.com/spring-data-jpa-modifying-annotation
  - https://www.baeldung.com/spring-data-jpa-delete
  - https://www.baeldung.com/jpa-cascade-types

Introduction to ORM and (Spring) JPA

Objectives :

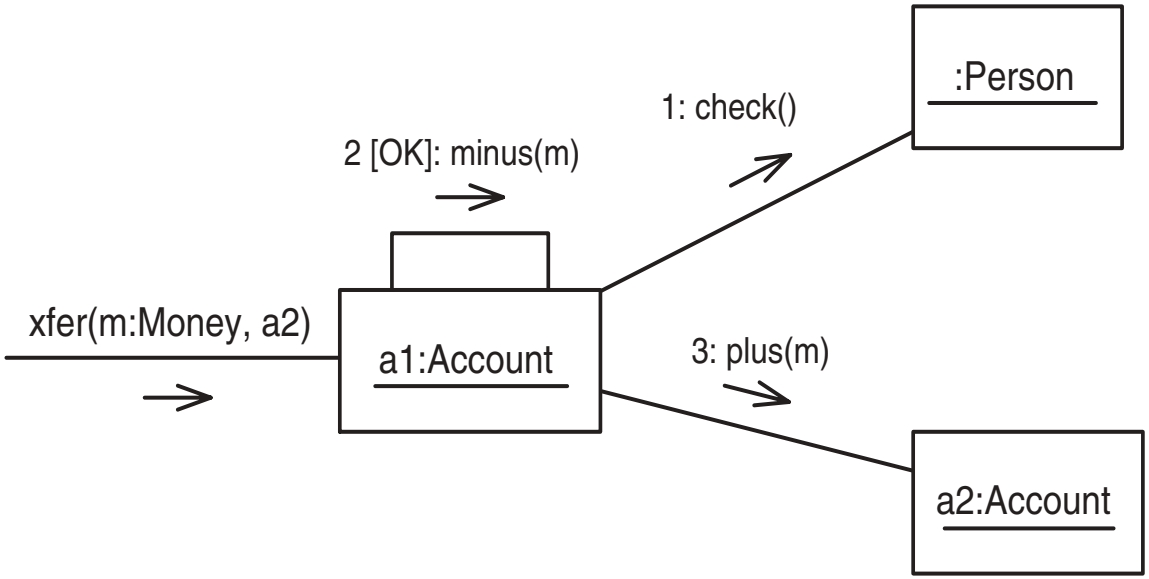
- **Persisting objects**
- **The object-relational mismatch**
- **Features of persistency mechanisms**
- **(Spring) JPA**

Persisting objects

OO-applications are composed of objects which

- consist of **data** and **behaviour**
- are connected to each other
- send messages to each other

Example: transfer money between accounts



Introduction to ORM and (Spring) JPA

1. Persisting objects
2. Object - relational mapping (ORM)
3. Features of persistence mechanisms
4. (Spring) JPA
5. Spring (Boot) - Recap

Object life cycle

- objects are described in classes (domain model)
- classes are instantiated at runtime
- instances are populated with data
- these data must be preserved i.e. persisted/saved

Object - relational mapping (ORM)

O/R mismatch

- granularity
  - object attributes
- mapping of inheritance trees
  - inheritance is (mostly) not supported in RDBMSs
  - map tree to table(s)
  - polymorphic queries
- identification of entities
  - in OO: not directly supported
  - in RDBMS: through primary keys (PK)

Introduction to ORM and (Spring) JPA

1. Persisting objects
2. Object - relational mapping (ORM)
3. Features of persistence mechanisms
4. (Spring) JPA
5. Spring (Boot) - Recap

Object - relational mapping (cont.)

- relationship management between entities (associations)
  - in OO: (navigational) links between objects
  - in RDBMS: PK - FK relations
  - **directional** problem in 1-many (or many-many) relationship
- transaction management
  - in application (application environment) or by data store?
  - concurrency control
- synchronization
  - how to keep data in objects in sync with database and vice-versa
- performance
  - caching
  - lazy loading

Object - relational mapping solutions

How to specify O/R mapping

- hard coding of DB schema in Java
- meta-data definitions in XML
- annotations

O/R mapping solutions (frameworks)

- do it yourself: extremely difficult and cost intensive
- use existing solutions:
  - JPA - Java Persistence Architecture
  - Java/Jakarta EE: Entity EJB (based on JPA)
  - Spring Data JPA (based on Hibernate)
  - alternatives: Hibernate, OJB, EclipseLink, Cayenne, MyBatis, ...

based on **Unit Of Work** Enterprise Application Architecture pattern

https://java-source.net/open-source/persistence

Introduction to ORM and (Spring) JPA

1. Persisting objects
2. Object - relational mapping (ORM)
3. Features of persistence mechanisms
4. (Spring) JPA
5. Spring (Boot) - Recap

Features of persistence mechanisms

- **persist** the information (data) in the object model, i.e.
  - the data in the objects described as attributes in class model
  - links between objects described as relationships in class model
- **synchronization** between application and data
  - data in memory must be synchronized with data in data store
  - data in data store must be synchronized with data in memory
  - important if different applications access the same data
- **transactions**
  - set of actions that move data from one consistent state to another
  - key features: **A**tomicity, **C**onsistency, **I**solation, **D**urability
- **concurrency control**
  - different users/applications must reach same data at same time...
  - ...while keeping the data in a consistent state

Introduction to ORM and (Spring) JPA

1. Persisting objects
2. Object - relational mapping (ORM)
3. Features of persistence mechanisms
4. (Spring) JPA
5. Spring (Boot) - Recap

Features of persistence mechanisms (cont.)

- **query mechanism**
  - need for some mechanism to retrieve data in a selective way from the data store
- **identity support**
  - unique identification of object
  - avoid multiple copies of the same data
- **security**
  - unauthorized people must not see sensitive data
- **performance optimisation**
  - use of caching

(Spring) JPA

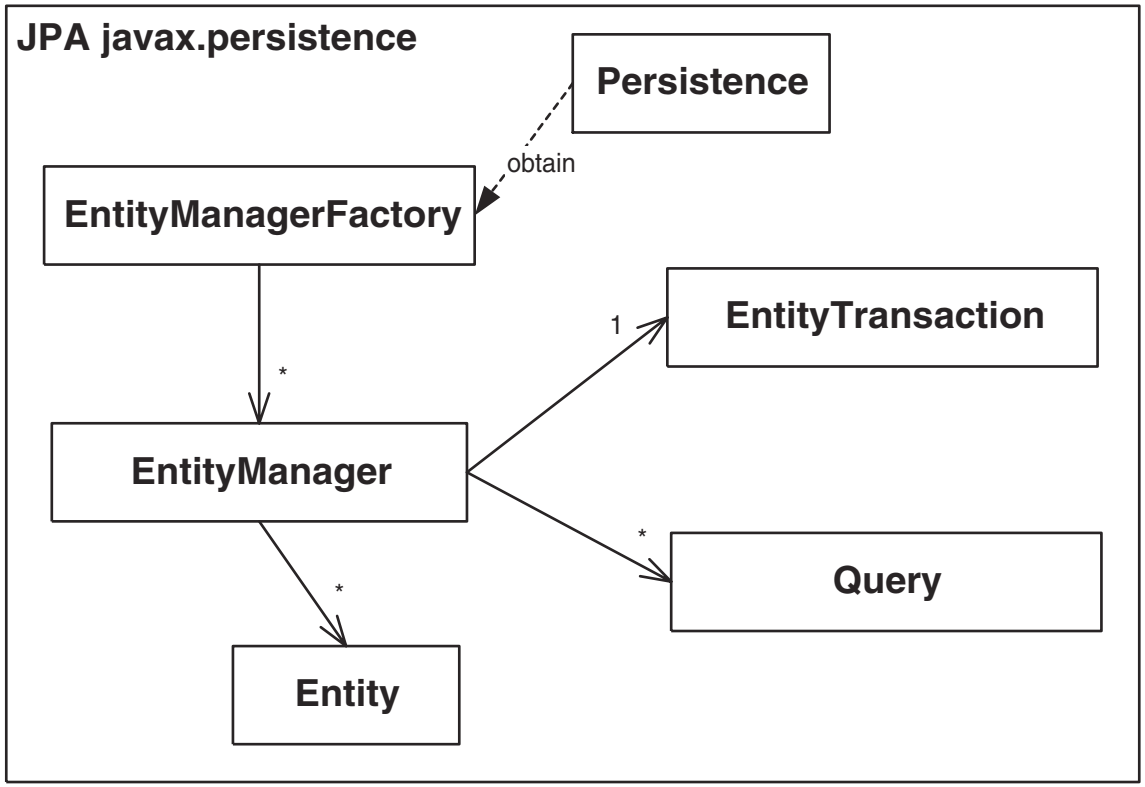
- **JPA: Java Persistence API (2006)**
  - **Java specification for accessing, persisting, and managing data between Java objects / classes and a RDBMS**
  - **defined as part of the EJB 3.0 specification as a replacement for the EJB 2 CMP Entity Beans specification**
  - **jakarta.persistence package (before javax.persistence)**
  - **specification, not implementation**
  - **implementations aka. JPA providers**

    EclipseLink, OpenJPA, **Hibernate,** Data Nucleus
- **Hibernate**
  - **started in 2001 as open source project**
  - **later further developed under JBoss (Red Hat)**
  - **used SessionFactory as basic interface**
  - **became certified implementation of JPA since version 3.5 (2010)**
  - **now mainly uses EntityManager as “interface”**

Introduction to ORM and (Spring) JPA

1. Persisting objects
2. Object - relational mapping (ORM)
3. Features of persistence mechanisms
4. (Spring) JPA
5. Spring (Boot) - Recap

JPA Architecture



JPA API

Introduction to ORM and (Spring) JPA

1. Persisting objects
2. Object - relational mapping (ORM)
3. Features of persistence mechanisms
4. (Spring) JPA
5. Spring (Boot) - Recap

package jakarta.persistence

- Persistence: get EntityManagerFactory instances (vendor-neutral)
- EntityManagerFactory: factory for EntityManagers
- EntityManager: manages a set of persistent objects

  acts also as factory for Query instances
- Entity: persistent object that represents datastore records
- EntityTransaction: (associated with 1 EntityManager) group operations on persistent data into consistent units of work
- Query: interface to find persistent objects that meet certain criteria

  uses the Java Persistence Query Language (JPQL), the Criteria API and/or the native Structured Query Language (SQL)

Spring's JPA integration

- Spring Data JPA
  - powerful repository and custom object-mapping abstractions
  - Query DSL:
    dynamic query derivation from repository method names
  - implementation domain base classes providing basic properties
  - support for transparent auditing (created, last changed)
  - possibility to integrate custom repository code
  - easy configuration in Spring Boot
  - advanced integration with Spring MVC controllers
  - experimental support for cross-store persistence

Introduction to ORM and (Spring) JPA

1. Persisting objects
2. Object - relational mapping (ORM)
3. Features of persistence mechanisms
4. (Spring) JPA
5. Spring (Boot) - Recap

Spring (Boot) - Recap

5

Spring's Philosophy

5.1

- **design:**
  - **Lasagna / ravioli code**
  - **Loose coupling**
  - **Coding to interfaces**
  - **Dependency Injection - Inversion of Control (IoC)**

    objects do not create other objects on which they rely to do the work. Instead, they get the objects that they need from an outside source.
- **DRY: avoid boilerplate code**

  -> templates, configuration taken out of the code, aop
- **easy to test**

  -> due to design + extra features on top of JUnit

Introduction to ORM and (Spring) JPA

1. Persisting objects
2. Object - relational mapping (ORM)
3. Features of persistence mechanisms
4. (Spring) JPA
5. Spring (Boot) - Recap

Ingredients

- Java Beans
  - reusable software components
  - classes conforming to a particular convention:
    default constructor, private variables with getters and setters, serializable
- Services - Repository
  - Service Layer decoupled from model
  - Repository: CRUD methods
- Configuration
  - Java based
  - annotations
- Container
  - ApplicationContext

Introduction to ORM and (Spring) JPA

1. Persisting objects
2. Object - relational mapping (ORM)
3. Features of persistence mechanisms
4. (Spring) JPA
5. Spring (Boot) - Recap

Tools

- IDE
  - IntelliJ
  - Eclipse + STS plugin
  - Spring Tool Suite (STS)
- Maven
  - build automation and software comprehension tool
  - uses Project Object Model (POM) to describe the software project being built, its dependencies on other external modules and components, and the build order
  - dynamically downloads Java libraries from one or more repositories https://search.maven.org/
- Server
  - only application server needed
  - Pivotal tc Server

Introduction to ORM and (Spring) JPA

1. Persisting objects
2. Object - relational mapping (ORM)
3. Features of persistence mechanisms
4. (Spring) JPA
5. Spring (Boot) - Recap

Spring Boot

- **simplifies the way of setting up a Spring application**
- **four main features:**
  - **Spring Boot Starters**
    aggregate common groupings of dependencies
  - **Auto Configuration**
    automatically configures beans needed by your application
  - **Command-line Interface (CLI)**
    simplifies even further, based on groovy
  - **Spring Boot Actuator**
    adds management features
- **integrated in STS/IntelliJ (Spring Initializr)**
- **running as Spring Boot App will bootstrap the embedded Tomcat server, deploy your app and map the URLs**

Introduction to ORM and (Spring) JPA

1. Persisting objects
2. Object - relational mapping (ORM)
3. Features of persistence mechanisms
4. (Spring) JPA
5. Spring (Boot) - Recap

Spring Starter Dependencies

- starters aggregate commonly needed dependencies
- based on the idea of transitive dependencies
- takes care of version compatibility
- more than 30 starters available
- for Spring Core, but also the portfolio projects
- examples
  - spring-boot-starter-web
  - spring-boot-starter-aop
  - spring-boot-starter-jdbc
  - spring-boot-starter-data-jpa
  - spring-boot-starter-ws
  - spring-boot-starter-rest

Spring Boot Application - Example

- **Create a new Spring Initializr project/module (check pom.xml!)**
- **Java Configuration Class (automatically added):**

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SpringBootRecapApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringBootRecapApplication.class, args);
    }

}
```

- **Model class**

```
public class Guest {

    private String name;
    private int age = 42;

    public Guest(String name) { this.name=name;}

    public String getName() {return name;}
    public void setName(String name) {this.name=name;}

    public int getAge() { return age;}
    public void setAge(int age) { this.age=age;}

}
```

Introduction to ORM and (Spring) JPA

1. Persisting objects
2. Object - relational mapping (ORM)
3. Features of persistence mechanisms
4. (Spring) JPA
5. Spring (Boot) - Recap

Spring Boot Application - Example (cont.)

- **Service: interface + implementation!**

```
public interface HelloService {
    Guest findGuest(int id);
    String sayHelloTo(Guest guest);
}
```

```
@Service
public class AbisHelloService implements HelloService {

    @Override
    public Guest findGuest(int id) {
        return new Guest("John");
    }

    @Override
    public String sayHelloTo(Guest guest) {
        return "Welcome at Abis, " +  guest.getName();
    }

}
```

Spring Boot Application - Example (cont.)

- **Junit Test**

```
@SpringBootTest
public class SpringBootRecapApplicationTests {

    @Autowired
    private HelloService helloService;

    private Guest guest;

    @BeforeEach
    void init(){
        guest = new Guest("Jane");
    }

    @Test
    void guestAgeIs42(){
        assertEquals(42,helloService.findGuest(1).getAge());
    }

    @Test
    void sayHelloToJane(){
        assertThat(helloService.sayHelloTo(guest),startsWith("Welcome"));
    }
}
```

Configuration

Objectives :

- **Configuring data sources**

Spring JPA: Basic Configuration

- **Maven dependency**

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

- **Spring Boot will do autoconfiguration**
  - will try to use embedded database by default
  - when own DataSource defined, this is disabled
  - disable manually via:

```
@SpringBootApplication(exclude={DataSourceAutoConfiguration.class})
```

Configuration

1. Spring JPA: Basic Configuration
2. Configuring a data source
3. Data Source Types
4. Simplifications from Spring Boot
5. Extra configuration options
6. Testing The Connection

Configuring a data source

2

General

2.1

- data source defines driver, database url, username and password
- central configuration makes it easy to change later on
- driver depends on the database, can be chosen when setting up Spring Boot Application for several supported DBs

Configuration

1. Spring JPA: Basic Configuration
2. Configuring a data source
3. Data Source Types
4. Simplifications from Spring Boot
5. Extra configuration options
6. Testing The Connection

Maven dependencies

H2

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
</dependency>
```

Oracle

```
<dependency>
    <groupId>com.oracle.database.jdbc</groupId>
    <artifactId>ojdbc11</artifactId>
</dependency>
```

PostgreSQL

```
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
</dependency>
```

SQL Server

```
<dependency>
    <groupId>com.microsoft.sqlserver</groupId>
    <artifactId>mssql-jdbc</artifactId>
</dependency>
```

Configuration

1. Spring JPA: Basic Configuration
2. Configuring a data source
3. Data Source Types
4. Simplifications from Spring Boot
5. Extra configuration options
6. Testing The Connection

Embedded Data Source 3.1

- embedded database runs as part of your application instead of a separate database
- invisible to end-user
- perfect choice for development and test purposes
- test data is reset every time you restart your application
- pass script that is used to set up the DB
- products: Derby, H2,..
- configuration:

```
@Bean
@Profile("development")
public DataSource testDataSource(){
    return new EmbeddedDatabaseBuilder()
            .setType(EmbeddedDatabaseType.H2)
            .addScript("classpath:h2.sql")
            .build();
}
```

Configuration

1. Spring JPA: Basic Configuration
2. Configuring a data source
3. Data Source Types
4. Simplifications from Spring Boot
5. Extra configuration options
6. Testing The Connection

Basic Data Source

- simplest possible, but limited data source
- provided by Spring (org.springframework.jdbc.datasource)
  - DriverManagerDataSource

    new connection every time one is requested
  - SingleConnectionDataSource

    returns same connection every time
- DBMS providers also have implementations available (e.g. OracleDataSource)
- no pooling possibilities!
- for small applications or at development phase

Configuration

1. Spring JPA: Basic Configuration
2. Configuring a data source
3. Data Source Types
4. Simplifications from Spring Boot
5. Extra configuration options
6. Testing The Connection

Basic Data Source - Configuration

- **application.properties**

```
spring.datasource.url=jdbc:oracle:thin:@//delphi:1521/TSTA
spring.datasource.username=tu00010
spring.datasource.password=tu00010
spring.datasource.driver-class-name = oracle.jdbc.OracleDriver
```

- **Java config (deprecated)**

```
@Bean
public DataSource dataSource() throws SQLException {
    OracleDataSource ds = new OracleDataSource();
    ds.setURL(env.getProperty("spring.datasource.url"));
    ds.setUser(env.getProperty("spring.datasource.username"));
    ds.setPassword(env.getProperty("spring.datasource.password"));
    ds.setDriverType(env.getProperty("spring.datasource.driver-class-name"));
    return ds;
}
```

Configuration

1. Spring JPA: Basic Configuration
2. Configuring a data source
3. Data Source Types
4. Simplifications from Spring Boot
5. Extra configuration options
6. Testing The Connection

Pooled Data Source

- directly configured in Spring
- provided by 3rd party (Apache Commons DBCP, c3po,Hikari,...)
- Hikari prefered (for Oracle/PostgreSQL/...), already provided in spring-boot-starter-data-jpa
- Maven dependencies for Apache Commons:
  - Apache Commons DBCP

```
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-dbcp2</artifactId>
</dependency>
```

RELIES ON:

  - Apache Commons Pool

```
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-pool2</artifactId>
</dependency>
```

Configuration

1. Spring JPA: Basic Configuration
2. Configuring a data source
3. Data Source Types
4. Simplifications from Spring Boot
5. Extra configuration options
6. Testing The Connection

Pooled Data Source - Configuration

- **application.properties**

```
// basic connection settings like before
spring.datasource.hikari.minimumIdle=5
spring.datasource.hikari.maximumPoolSize=20
spring.datasource.hikari.idleTimeout=30000
spring.datasource.hikari.maxLifetime=2000000
spring.datasource.hikari.connectionTimeout=30000
```

- **Java Config (deprecated)**

```
@Bean
public DataSource prodDatasource() throws SQLException {
    HikariDataSource hds = new HikariDataSource();
    hds.setJdbcUrl(env.getProperty("spring.datasource.url"));
    hds.setUsername(env.getProperty("spring.datasource.username"));
    hds.setPassword(env.getProperty("spring.datasource.password"));
    hds.setDriverClassName
            (env.getProperty("spring.datasource.driver-class-name"));
    hds.setMinimumIdle
            (Integer.parseInt(env.getProperty("spring.datasource.hikari.minimumIdle"));
    hds.setMaximumPoolSize(Integer.parseInt
            (env.getProperty("spring.datasource.hikari.maximumPoolSize"));
    hds.setIdleTimeout
            (Long.parseLong(env.getProperty("spring.datasource.hikari.maxLifetime"));
    hds.setConnectionTimeout(Long.parseLong
            (env.getProperty("spring.datasource.hikari.connectionTimeout"));
    return hds;
}
```

JNDI Data Source

- best to use in production
- data source managed by server (WebSphere, Tomcat,...) and retrieved via JNDI
- configuration
  - application.properties

```
spring.datasource.jndi-name=java:comp/env/jdbc/MyJndiName
```

  - Java Config (deprecated)

```
@Bean
public DataSource dataSource() throws NamingException {
    return (DataSource)  new JndiTemplate().lookup
                (env.getProperty("java:comp/env/jdbc/MyJndiName"));
}
```

Simplifications from Spring Boot

- Spring Boot can autoconfigure datasources based on properties files
- don’t need to configure them any more and load resources!
- create application-**myprofilename**.properties files when profiles used
- use pre configured properties of Spring Boot
- driver will be deduced from database url
- when nothing configured, automatically embedded db used based on definitions in pom.xml
- EntityManager will be used behind the scenes

Configuration

1. Spring JPA: Basic Configuration
2. Configuring a data source
3. Data Source Types
4. Simplifications from Spring Boot
5. Extra configuration options
6. Testing The Connection

Extra configuration options

application.properties file can contain many extra properties

- DB initialization: spring.jpa.hibernate.ddl-auto
  - create: existing tables are first dropped, and then recreated
  - create-drop: tables are created, and dropped after application is finished
  - validate: validates whether the tables and columns exist; otherwise, it throws an exception
  - update: object model created based on the mappings is compared with the existing schema, schema is updated according to the diff
  - none: no DDL is being done

    Spring Boot defaults the parameter value to create-drop if no schema manager has been detected, otherwise none for all other cases

Configuration

1. Spring JPA: Basic Configuration
2. Configuring a data source
3. Data Source Types
4. Simplifications from Spring Boot
5. Extra configuration options
6. Testing The Connection

Extra configuration options (cont.)

- formatting/logging SQL
  - spring.jpa.show-sql = true
  - spring.jpa.properties.hibernate.format_sql = true
- standard logging
  - logging.level.org.hibernate.SQL=DEBUG
  - logging.level.org.hibernate.type.descriptor.sql.BasicBinder =TRACE

Configuration

1. Spring JPA: Basic Configuration
2. Configuring a data source
3. Data Source Types
4. Simplifications from Spring Boot
5. Extra configuration options
6. Testing The Connection

Testing The Connection

- **DataSourceTest.java**

```
@SpringBootTest
public class DataSourceTest {

    @Autowired
    DataSource dataSource;

    @Test
    public void testConnectionViaDataSource() {

        try (Connection c = dataSource.getConnection()){
            System.out.println("connection succeeded via "
                                + c.getMetaData().getDatabaseProductName() + " . " );
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Configuration

1. Spring JPA: Basic Configuration
2. Configuring a data source
3. Data Source Types
4. Simplifications from Spring Boot
5. Extra configuration options
6. Testing The Connection

Basic OR Mapping and JPA repositories

Objectives :

- **Mapping of tables and properties**
- **Object identity**
- **JPA repositories and Query Methods**
- **Spring Data JUnit Testing**

Mapping concepts

What to map?

- map class to relational table
- map obj attribute / property to relational column + type conversion
- provide object identity
- map class association to relational FK
- inheritance/hierarchy mapping

Mapping is done using (JPA) annotations (since Hibernate 3)

using configuration by exception,
the number of annotations required to do simple mappings
can be very limited

Basic OR Mapping and JPA repositories

1. Mapping concepts
2. Class to table mapping
3. Object identity
4. Property mapping
5. JPA repositories
6. Spring Data JUnit Testing

Class to table mapping

Use annotations on class level

- `@Entity` enables class to become persistent
- `@Table` associates with RDBMS table name (default = class name)

Example

```
@Entity
@Table(name="GUESTS")
public class Guest {  }
```

Define schema

- use schema attribute in table annotation

  `@Table(name="GUESTS", schema="SPRINGJPA")`
- default schema in application.properties

  spring.jpa.properties.hibernate.default_schema = springjpa

Basic OR Mapping and JPA repositories

1. Mapping concepts
2. Class to table mapping
3. Object identity
4. Property mapping
5. JPA repositories
6. Spring Data JUnit Testing

Object identity

Distinction between

- identical objects: `a == b`
- equal objects: `a.equals(b)`
- database identity: same primary key (PK) in database

Distinction between

- natural keys (e.g. email)
- synthetic or surrogate keys (e.g. generated keys)

Criteria to choose a primary key

- not null
- unique
- value never changes, immutable (i.e. no `set()` method provided)

Basic OR Mapping and JPA repositories

1. Mapping concepts
2. Class to table mapping
3. Object identity
4. Property mapping
5. JPA repositories
6. Spring Data JUnit Testing

Object identity - mapping

Basic OR Mapping and JPA repositories

1. Mapping concepts
2. Class to table mapping
3. Object identity
4. Property mapping
5. JPA repositories
6. Spring Data JUnit Testing

Annotation `@Id` to (placement is important for access type for all attributes)

- key field -> *field access type* (for all attributes!)
- get method (property) -> *property access type* (for all attributes!)

If no key generator is provided

- application is responsible for providing/defining a value for the PK
- define constructor with argument for PK, e.g.

```
public Guest(String name) { this.name=name; }
```

Unique key/identifier value can be generated: specify strategy -> `@GeneratedValue`

- AUTO: selects generation strategy based on the DB specific dialect
- IDENTITY: use identity columns (if supported in RDBMS)
- SEQUENCE: sequential unique key
- TABLE: key based on value in separate table

Examples

- **Auto and Identity**

```
@GeneratedValue(strategy = GenerationType.AUTO)
@Column(name="gno")
private int id;
```

```
@GeneratedValue(strategy = GenerationType.IDENTITY)
@Column(name="gno")
private int id;
```

- **Sequence generator**

```
@SequenceGenerator(name="mySeqGen", sequenceName="guestsequence",
                                    allocationSize=1)
@Id  @GeneratedValue (strategy=GenerationType.SEQUENCE,
                                      generator="mySeqGen")

@Column(name="gno")
private Long id;
```

- **Table generator (**

```
@TableGenerator(name="myTabGen", table="keytable",
                        pkColumnName="keygenName", pkColumnValue="myGen",
                        valueColumnName="newValue", initialValue=100,
                          allocationSize=10)

@Id @GeneratedValue (strategy=GenerationType.TABLE, generator="myTabGen")

@Column(name="gno")
private Long id;
```

Composite primary key

- 2 techniques for creating a composite key
  - @IdClass
  - @EmbeddedId
- in both cases, an extra class is added for the Id “Object”
- class has to
  - implement Serializable (since possible session caching could occur)
  - override equals/hashCode (for testing key equality)
- usage in JpaRepository class will be slightly different for both techniques!

Basic OR Mapping and JPA repositories

1. Mapping concepts
2. Class to table mapping
3. Object identity
4. Property mapping
5. JPA repositories
6. Spring Data JUnit Testing

Composite key via @IdClass

```
public class PersonKey implements Serializable {
    private int nr;
    private String email;
    public PersonKey() { }
    public PersonKey(int nr, String email) {
        this.nr=nr;
        this.email=email;
    }
    public boolean equals(Object o){
        Person p= (Person)o;
        return this.nr==p.nr && this.email.equals(p.email);
    }
    public int hashcode(){
        return Objects.hash(nr,email);
    }
}
@Entity
@IdClass(PersonKey.class)
public class Person {
    @Id
    private int nr;
    @Id
    private String email;
    ...
}
```

Composite key via @IdClass (cont.)

```
public interface PersonJpaRepository extends JpaRepository<Person, PersonKey> {

    // Query DSL
    Person findByNrAndEmail(int nr, String email);

     //JPQL
    @Query("select p from Person p where p.nr=:nr and p.email = :email")
    Person findPersonById(@Param("nr") int nr, @Param("email") String email);

}
```

Composite Key via @EmbeddedId

```
@Embeddable
public class PersonKey implements Serializable {
    private int nr;
    private String email;
    public PersonKey() { }
    public PersonKey(int nr, String email) {
        this.nr=nr;
        this.email=email;
    }
    public boolean equals(Object o){
        Person p= (Person)o;
        return this.nr==p.nr && this.name.equals(p.name);
    }
    public int hashcode(){
        return Objects.hash(nr,email);
    }
}
@Entity
public class Person {
    @EmbeddedId
    private PersonKey persKey;
    ...
}
```

Composite Key via @EmbeddedId (cont.)

```
public interface PersonJpaRepository extends JpaRepository<Person, PersonKey> {

    // Query DSL
    Person findByPersKey_NrAndPersKey_Email(int nr, String email);

     //JPQL
    @Query("select p from Person p where p.persKey.nr=:nr and p.persKey.email = :email")
    Person findPersonById(@Param("nr") int nr, @Param("email") String email);

}
```

Property mapping

- persistent field (direct field access), or
- persistent property (property access via get/set)

```
@Column (name="column_name")
```

- name of column in relational table (default = property name)
- private, protected, package visibility for fields

Example

```
@Column(name="ctitle")
private String title;

@Column(name="cdur")
public short getDuration() { return duration; }
```

Configuration by exception: i.e. default configuration/specification for each (non static non transient) field!

Basic OR Mapping and JPA repositories

1. Mapping concepts
2. Class to table mapping
3. Object identity
4. Property mapping
5. JPA repositories
6. Spring Data JUnit Testing

Property mapping (cont.)

Additional attributes on `@Column`

- length (optional): column length (**default 255 !**)
- updatable/insertable: whether or not the column will be part of the update/insert statement (default true)
- nullable: set the column as nullable (default true)
- unique: set a unique constraint on the column (default false)
- precision: column decimal precision (default 0)
- scale: column decimal scale if useful (default 0)
- table: define the targeted table (default primary table)
- columnDefinition: e.g. CHAR(45)

Notes:

- `@Transient` -> field will not be persisted
- make use of Bean Validation annotations, e.g. `@NotNull`

Type conversion - built-in Hibernate types

4.1

Mapping type	Java type	ANSI Standard SQL type
big_decimal	java.math.BigDecimal	NUMERIC
big_integer	java.math.BigInteger	NUMERIC
boolean	boolean or java.lang.Boolean	BIT
byte	byte or java.lang.Byte	TINYINT
character	java.lang.Character	CHAR(1)
double	double or java.lang.Double	DOUBLE
float	float or java.lang.Float	FLOAT
integer	int or java.lang.Integer	INTEGER
long	long or java.lang.Long	BIGINT
short	short or java.lang.Short	SMALLINT
string	java.lang.String	VARCHAR
yes_no	boolean or java.lang.Boolean	CHAR(1) ('Y' or 'N')
true_false	boolean or java.lang.Boolean	CHAR(1) ('T' or 'F')
date	java.util.Date or java.sql.Date	DATE
time	java.util.Date or java.sql.Time	TIME
local date	java.time.LocalDate (Java 8)	DATE
local time	java.time.LocalTime (Java 8)	TIME
timestamp	java.util.Date or java.sql.Timestamp	TIMESTAMP
calendar	java.util.Calendar	TIMESTAMP

Type conversion - built-in Hibernate types (cont.)

Mapping type	Java type	Standard SQL type
calendar_date	java.util.Calendar	DATE
locale	java.util.Locale	VARCHAR
currency	java.util.Currency	VARCHAR
timezone	java.util.TimeZone	VARCHAR
class	java.lang.Class	VARCHAR
binary	byte[]	VARBINARY (or BLOB)
blob	java.sql.Blob	BLOB
clob	java.sql.Clob	CLOB
serializable	java.io.Serializable implementor	VARBINARY (or BLOB)
text	java.lang.String	CLOB

- type is derived by using reflection
- `@Temporal(TemporalType.DATE, TIME, or TIMESTAMP)` for date/time fields
- `@Lob` for large object (or serializable) properties
- `@Enumerated` for enums (EnumType.STRING or .ORDINAL)
- User-defined types can be specified with `@Type` annotation

Property mapping - derived properties

Property not mapped to database column

- annotation `@Formula`
- value of property calculated at runtime
- no column attribute (no insert or delete) -> **read-only (Select)**
- are allowed in formula (SQL fragment):
  - columns of database table
  - SQL functions
  - subselects

Example

```
@Formula("cprice*1.21")
private float finalPrice;
```

Basic OR Mapping and JPA repositories

1. Mapping concepts
2. Class to table mapping
3. Object identity
4. Property mapping
5. JPA repositories
6. Spring Data JUnit Testing

JPA repositories

5

General

5.1

- in any reasonable-sized application, same methods (add, remove,...) will reappear (for different classes) over and over again
- only domain types will be different
- solution: only write repository interface that extends JpaRepository

```
public interface GuestJpaRepository extends JpaRepository<Guest,Long> {}
```

- above example will persist **Guest** objects with id of type **Long**
- JpaRepository contains 18 methods for performing common persistence operations, such as save(=merge!), delete, find by id,...
- extra methods can be added, which are translated to SQL
- implementation is generated at application startup
- configure via
  - Java Config

    ```
    @EnableJpaRepositories(basePackages="be.abis.ch8jparepo.dao")
    ```

  - autimatically configured in @SpringBootApplication

Basic OR Mapping and JPA repositories

1. Mapping concepts
2. Class to table mapping
3. Object identity
4. Property mapping
5. JPA repositories
6. Spring Data JUnit Testing

Repository Interfaces

- CrudRepository<T,ID>
  - <S extends T> S save(S var1): saves/updates a given entity
  - Optional<T> findById(ID var1): retrieves an entity by its id
  - boolean existsById(ID var1): does entity with given id exists
  - Iterable<T> findAll(): all instances of the type
  - long count(): number of entities available
  - void deleteById(ID var1): deletes the entity with the given id
  - void delete(T var1): deletes a given entity
  - void deleteAll(): deletes all entities managed by the repository

Basic OR Mapping and JPA repositories

1. Mapping concepts
2. Class to table mapping
3. Object identity
4. Property mapping
5. JPA repositories
6. Spring Data JUnit Testing

Repository Interfaces (cont.)

- PagingAndSortingRepository (extends CrudRepository)
  - Iterable<T> findAll(Sort var1): returns all entities sorted by the given options
  - Page<T> findAll(Pageable var1): returns a Page of entities meeting the paging restriction provided in the Pageable object
- JpaRepository (extends PagingAndSortingRepository)
  - List<T> findAll()
  - List<T> findAll(Sort var1)
  - void flush(): flushes all pending changes to the database
  - <S extends T> S saveAndFlush(S var1): saves an entity and flushes changes instantly
  - void deleteInBatch(Iterable<T> var1): deletes the given entities in a batch which means it will create a single query
  - void deleteAllInBatch(): deletes all entities in a batch call
  - T getById(ID var1): returns a **reference** to the entity with the given identifier

Query Methods

- possibility to add methods to interface
- Spring can determine implementation based on the method name
- example

  ```
  List<Company> getCompanyByCountryOrderByName(String country);
  ```

- four verbs allowed in method name: get, read, find and count
- subject of repository is mostly optional
- syntax: query verb + subject (optional) + predicate + order by
- predicate
  - condition must reference one or more properties (combined with And and Or)
  - if no comparison operator, than equals is used
  - other comparison operators:

    IsAfter, IsGreaterThen, IsLessThan, IsBetween, IsNull, IsLike, Containing, IsNotIn, IsStartingWith, Contains, IsTrue,...
  - IgnoringCase can also be added

Basic OR Mapping and JPA repositories

1. Mapping concepts
2. Class to table mapping
3. Object identity
4. Property mapping
5. JPA repositories
6. Spring Data JUnit Testing

Example

```
public interface GuestJpaRepository extends JpaRepository<Guest, Long> {

    Guest findById(long id);
    List<Guest> findByName(String name);
    List<Guest> findByNameStartsWith(String name);
    List<Guest> findByAge(int age);
    List<Guest> findByAgeIsLessThan(int age);
    List<Guest> findByNameStartsWithAndAgeIsLessThan(String name, int age);
    long countByName(String name);
}
```

Spring Data JUnit Testing

6

- put @Transactional on test class or methods (org.springframework.transaction.annotation.Transactional)
- test-managed transactions will automatically be rollbacked
- test-managed transaction is one that is managed by TransactionalTestExecutionListener
- when running tests while starting up the Spring application context, a TransactionalTestExecutionListener is automatically configured
- @Transactional creates a new transaction that is then automatically rolled back after test completion
- can be changed, @Commit or @Rollback at the class or method level
- can use the static methods in TestTransaction, to start and end transactions, flag them for commit or rollback or check the transaction status
- Watch out: tests via RestTemplate or Postman are not automatically rolled back!

Basic OR Mapping and JPA repositories

1. Mapping concepts
2. Class to table mapping
3. Object identity
4. Property mapping
5. JPA repositories
6. Spring Data JUnit Testing

Mapping Associations

Objectives :

- **Value types**
- **One class for two tables**
- **Mapping associations**
- **Cascading and fetching strategies**

Value types

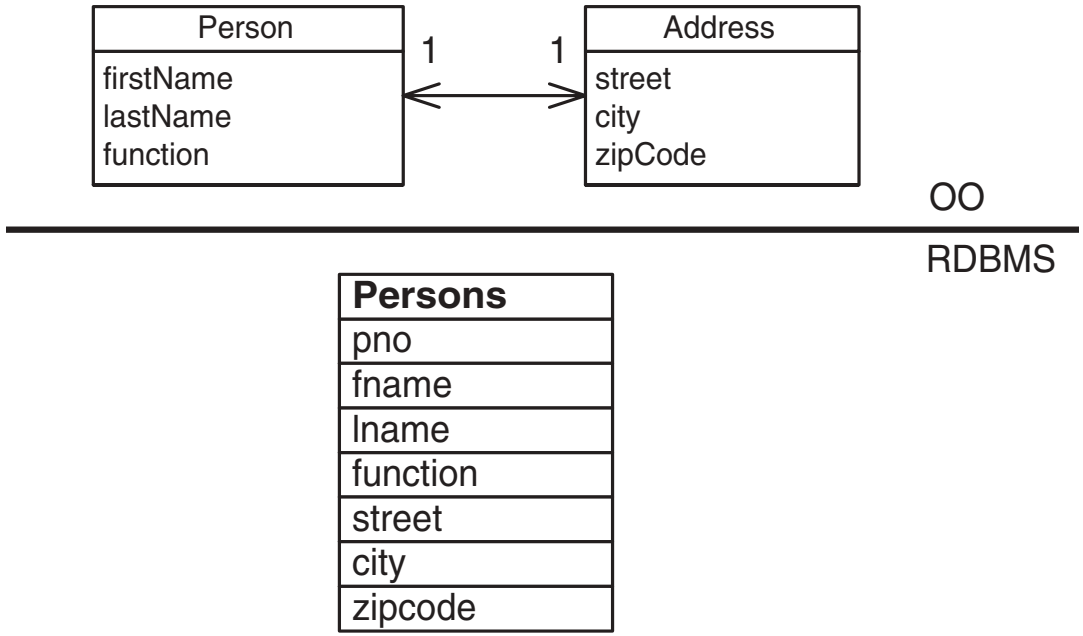
Mapping Associations

1. Value types
2. One class for two tables
3. Association mapping
4. Additional options

Difference between

- **entity type: has its own database identity**
- **value type: depends on database identity of owning entity type**

  2 (or more) classes map to 1 database table



Embedded objects

Use `@Embedded` annotation (in 'parent' class) on contained property

```
@Entity
@Table(name="Persons")
public class Person {
    @Embedded
    private Address address;
    ...
}
```

Use `@Embeddable` annotation on contained ('child') class

```
@Embeddable
public class Address {
    ...
    private String street;
    private String city;
}
```

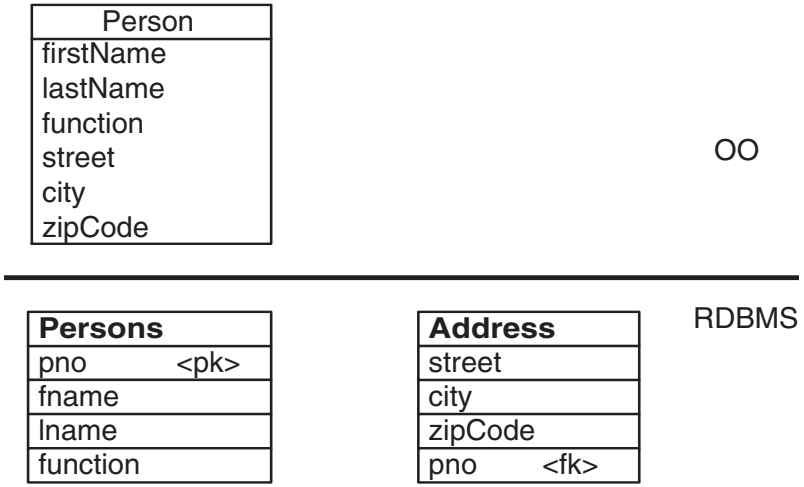
Notes:

- shared references to Address object is not possible (composition)
- `@Embedded` is optional if `@Embeddable` is defined

One class for two tables

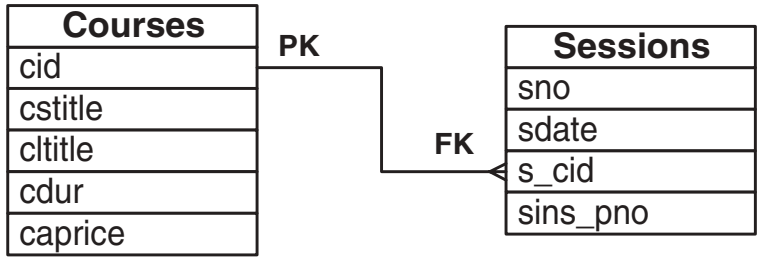
Use @SecondaryTable annotation

```
@Entity
@Table(name="Persons")
@SecondaryTable(name="Address",
pkJoinColumns = @PrimaryKeyJoinColumn
(name = "pno"))
public class Person {
    ...
    @Column(name="lname")
    private String lastName;
    ...
    @Column(table="Address", name="street")
    private String street;
    @Column(table="Address", name="city")
    private String city;
    ...
```



Association mapping

- multiplicity (one-to-one, one-to-many, many-to-one, many-to-many)
- directionality (navigational access to data)
  - uni- or bidirectional
  - must be implemented in Java code
  - no managed associations in JPA -> owner class is responsible for updates
- PK - FK relationship (no direction) in RDBMS



Mapping Associations

1. Value types
2. One class for two tables
3. Association mapping
4. Additional options

One-to-one mapping

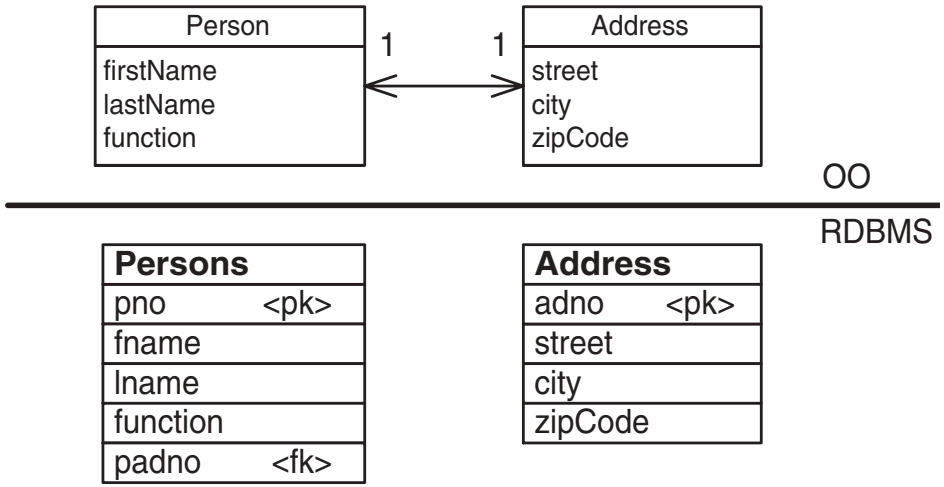
Use when:

- value-type mapping (embedding) not possible
- entity mapping needed - each class mapped to a table

Possibilities:

- foreign key association
- primary key association
- association table (rarely used)

One-to-one mapping (cont.)



- FK in Persons table refers to PK of Address table
- FK (padno) must be unique -> add unique constraint in database
- for bidirectional mapping:
  - use also one-to-one mapping in Address mapping
  - class Person, containing FK, is owning side

One-to-one mapping annotations

- `@OneToOne` annotation on association property
- naming of FK - PK columns
  - `@JoinColumn(name= ...)` on FK property
  - `@Column(name= ...)` on PK Id property

One-to-one mapping - Example

```
@Entity
@Table(name="Persons")
public class Person {
    ...
    @OneToOne
    @JoinColumn(name="padno")
    private Address homeAddress;
    ...
}

@Entity
@Table(name="Address")
public class Address {
    ...
    @Id @GeneratedValue
    @Column(name="adno")
    private int addressNr;
    ...
}
```

Bidirectional association

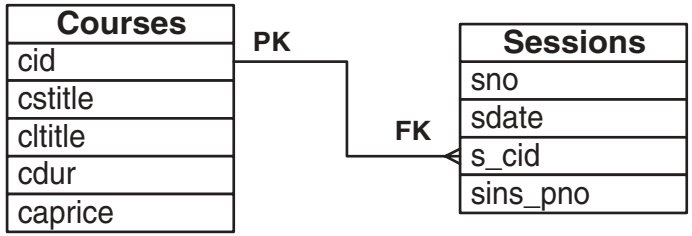
- *Owner (owning side)* class is responsible for the association column(s) update
- Specify additional property on the *target (inverse)* side
- Annotate with `@OneToOne` and attribute `mappedBy` at inverse side of the assocation (refer to property name in *owner* class)

Example

```
@Entity
@Table(name="Address")
public class Address {
    ...
    @OneToOne(mappedBy="homeAddress")
    private Person pers;
}
```

Many-to-one mapping

PK - FK relationship (no direction) in RDBMS



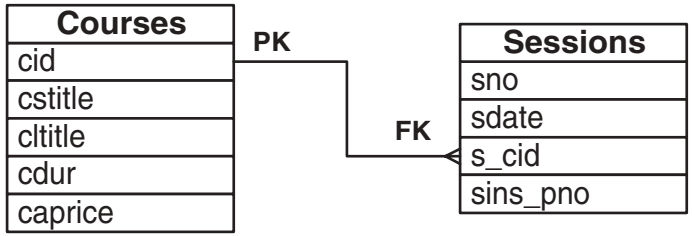
- Use `@ManyToOne` annotation on the Course object in the Session class

Many-to-one mapping - Example

```
public class Course {
    ...
    @Id @GeneratedValue
    @Column(name="cid")
    private int courseId;
    private String coursetitle;
    ...
}

public class Session {
    ...
    @ManyToOne
    @JoinColumn(name="s_cid")
    private Course course;
    ...
}
```

One-to-many mapping

PK - FK relationship (no direction) in RDBMS



Use `@OneToMany` annotation (on property with **interface** Set or List)

One-to-many mapping (cont.)

- Define foreign key column via `mappedBy` attribute, and (optionally) target of association via `targetEntity` attribute

  (not required if generic type is specified)
- reverse direction: use `@ManyToOne` and `@JoinColumn` annotations
- 'many' side is owning side

Example

```
public class Course {
...
    @OneToMany(targetEntity=Session.class, mappedBy="course")
    private Set<Session> courseSessions = new HashSet<Session>();
}

public class Session {
...
    @ManyToOne
    @JoinColumn(name="s_cid")
    private Course course;
}
```

One-to-many mapping - remarks

- don't forget to initialise the collection !
- override `equals()` and `hashCode()` for the entities used in a set identification based on "business key equality"
- control association inside application, e.g. via convenience method

```
public class Course{
...
    public void addSession(Session sess) {
        this.getCourseSessions().add(sess);
        sess.setCourse(this);
    }
    public void removeSession(Session sess) {
        this.getCourseSessions().remove(sess);
        sess.setCourse(null);
    }
```

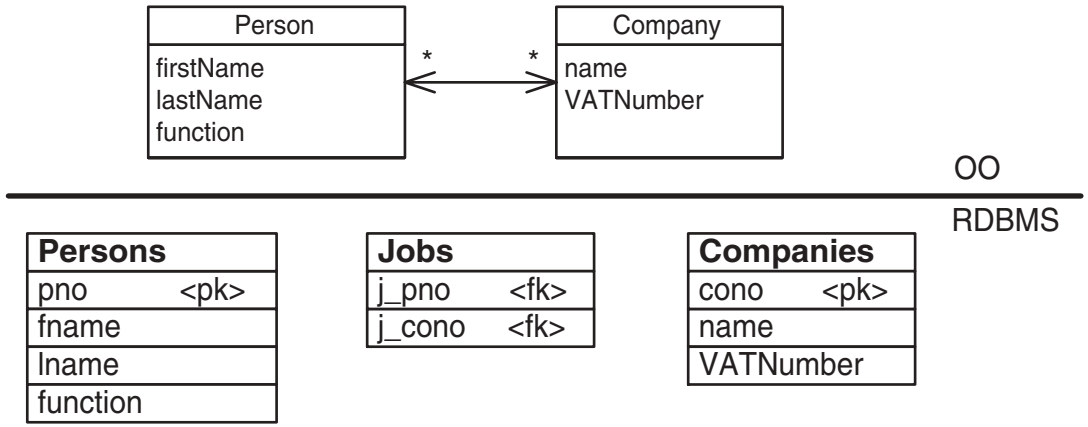
One-to-many mapping - remarks (cont.)

If the many-collection is implemented as a java.util.Map, use the `@MapKey` annotation for the primary key.

Example: person with multiple email addresses

```
@Entity
public class Person {
. . . .
@OneToMany
@MapKey(name="number")
private Map<String, Email> emails;
. . . .
}
```

Many-to-many associations



Can always be mapped with two many-to-one associations (recommended)

Use `@ManyToMany` annotation

Unidirectional and bidirectional associations possible

In the database, one additional table is needed

(intermediate association class)

Many-to-many mapping

Mapping table generated for each direction by default

- use `mappedBy` attribute to point to same mapping table
- PK is combination of both foreign keys

Definition of join table (and columns)

- `@JoinTable(name="...")`
- `@JoinColumns(joinColumns=@JoinColumn(name="..."), inverseJoinColumns=@JoinColumn(name="..."))`

Bidirectional mapping

- repeat `@JoinTable` and switch join column names

Example

```
public class Person {
    ...
    @ManyToMany
    @JoinTable(name = "jobs",
                joinColumns = @JoinColumn(name = "j_pno"),
                inverseJoinColumns = @JoinColumn(name = "j_cono"))
    private List<Company> listOfCompanies = new ArrayList<Company>();
    ...
}

public class Company {
    @ManyToMany(targetEntity=Person.class, mappedBy="listOfCompanies")
    private List<Person> listOfEmployees = new ArrayList<Person>();
    ...
}
```

Additional options

4

Cascade strategy

4.1

- how will cascading happen along the association path?
- default = no cascading !
- define `cascade=CascadeType.` (javax.persistence package)
  - PERSIST: save/persist corresponding record in other table
  - REMOVE: delete corresponding record in other table
  - MERGE: merge (after detach) corresponding record in other table
  - REFRESH: refresh corresponding record from other table
  - DETACH: detach corresponding record in other table
  - ALL: all the above options
- If cascade type does not correspond with database action:

  `TransientObjectException`

Additional options (cont.)

Fetching strategy

4.2

- how are associated records fetched?
- define `fetch=FetchType`.
  - EAGER: load corresponding record (default for one-to-one and many-to-one)
  - LAZY: do not load corresponding record (default for one-to-many and many-to-many)

    also called lazy initialisation

Mapping Associations

1. Value types
2. One class for two tables
3. Association mapping
4. Additional options

Querying in Spring JPA

Objectives :

- **@Query**
- **JPQL**
- **Named queries**
- **Native queries**
- **Using DTOs and mappers**

@Query

- @Query can be defined on repository methods to specify your own query, instead of using a generated one
- query can be written in JPQL or SQL

```
@Query("select c from Course c where c.title = 'Java' ")
List<Course> findJavaCourses();

@Query( value = "select * from courses where title = 'Java' ",
        nativeQuery=true)
List<Course> findJavaCoursesNative();
```

- positional or named parameters can be passed

```
@Query("select c from Course c where c.title = ?1 ")
List<Course> findCoursesWithTitle(String title);

@Query("select c from Course c where c.title = :title ")
List<Course> findCoursesWithTitle(@Param("title") String title);
```

Querying in Spring JPA

1. @Query
2. JPQL
3. Named queries
4. Native queries
5. Changing the “select” object
6. Using DTOs and mappers

JPQL

JPA Query Language (JPQL) characteristics

- use classes + properties instead of tables + columns
- polymorphic queries
- automatic join for associations
- full support of relational operations
- paging features

Querying is done in an ‘object oriented’ way!

Querying in Spring JPA

1. @Query
2. JPQL
3. Named queries
4. Native queries
5. Changing the “select” object
6. Using DTOs and mappers

Basic JPQL

projection SELECT specify selected data

selection FROM data source

restriction WHERE match records to criteria

Querying in Spring JPA

1. @Query
2. JPQL
3. Named queries
4. Native queries
5. Changing the “select” object
6. Using DTOs and mappers

Selection: From

- **Defines the class name + an alias**

  select cc from ConcreteCourse [as] cc

- **JPQL is case sensitive for class names and properties**
- **if @Entity(name="abc") attribute is used on class, use abc in JPQL from clause**
- **alias will be used in where clause**

JPQL: polymorphic queries

```
Select c from Course c
     //selects all ConcreteCourse and CourseGroup objects
Select o from java.lang.Object o
     //selects all persistent objects
Select s from java.io.Serializable s
     //selects all serializable objects
```

Restriction: Where

- Comparison operators
  - where column [= | <> | < | > | <= | >=] expression
  - use of NOT: negates the expression

```
select c from Course c where c.title = 'Java'
select c from Course c where c.title <> 'Java'
select c from Course c where c.duration > 3
select c from Course c where c.duration <= 4
select c from Course c where not(c.title = 'Java')
```

- (NOT) IN operator

```
select c from Course c where c.title IN ('JAVAPROG','SPRINGJPA')
select c from Course c NOT IN ('JAVAPROG','SPRINGJPA')
```

- (NOT) Between operator (border values are inclusive)

```
select c from Course c where c.price between 200 and 300
```

Querying in Spring JPA

1. @Query
2. JPQL
3. Named queries
4. Native queries
5. Changing the “select” object
6. Using DTOs and mappers

Restriction: Where (cont.)

- **(NOT) like operator**

```
select c from Course c where c.title like 'Java%'
select c from Course c where c.title like '%Java%'
select c from Course c where c.title not like '%Java%'
select c from Course c where lower(c.title) like '%java%'
```

- **Is (NOT) Null checking**

```
select c from Course c where c.title is null
select c from Course c where c.title is not null
```

- **AND, OR, NOT operators**

```
select c from Course c
where c.title not like '%java%'
   and (c.duration < 2 or c.duration > 4)
```

- **Collection operations**

```
select c from Company c, Person p
where c.employees is not empty

select c from Company c, Person p
where p member of c.employees

select c from Company c, Person p
where size(c.employees) > 100
```

- **functions : https://en.wikibooks.org/wiki/Java_Persistence/JPQL**

Joins in JPQL

- inner join automatically generated when traversing entities

  select p from Person p where p.address.country = 'B'

- join can also be mentioned explicitly
  - “comma” notation, with join condition in where statement

    select d from Employee e, Department d where e.department = d
  - (inner) join

    select distinct p from Person p join p.enrolments e where e.pricePerDayPayed > 400
  - left join

    select p from Person p left join Company c where p.age>40
  - join fetch (to override FetchType.LAZY)

    select p from Person p join fetch p.companies c where p.age>40

  mainly used in combination with @OneToMany and @ManyToMany associations

  watch out: if no where condition or traversing entity used, a cartesian product is generated!

  select p from Person p join Address a

Querying in Spring JPA

1. @Query
2. JPQL
3. Named queries
4. Native queries
5. Changing the “select” object
6. Using DTOs and mappers

Parameter binding in JPQL

- **Named parameters**
  - parameter name preceded by a ":"
  - in method, use @Param which states the name of the parameter

```
@Query("select c from Course c
        where c.duration = :dur and c.title like %:title%)
List<Course> getCourseByDurationAndTitle
      (@Param("dur") int duration, @Param("title") String title)
```

- **Positional parameters (deprecated!)**

```
@Query("select c from Course c
        where c.duration = ?1 and c.title like %?2%")
List<Course> getCourseByDurationAndTitle
                      (int duration, String title)
```

Note: watch out for SQL injection !

Named queries

- Predefine commonly used queries in the persistent class
- Add `@NamedQuery` annotation before entity class
- add a method in the repository interface which has exactly the same name as the name of the named query
- name obligatory prefixed with the class name
- uses JPQL by default
- possible to add parameters
- also @NamedNativeQuery, which will use SQL instead
- Example

```
@NamedQuery(name="Course.findByDurAndTitleNQ",
                    query="select c from Course c
                                where c.duration = :dur and c.title like %:title%")
public class Course {
    ...
}
public interface CourseJpaRepository extends JpaRepository<Course,Integer> {
    List<Course> findByDurAndTitleNQ
        (@Param("dur") int duration, @Param("title") String title);
}
```

Querying in Spring JPA

1. @Query
2. JPQL
3. Named queries
4. Native queries
5. Changing the “select” object
6. Using DTOs and mappers

Native queries

- use @Query(value="", nativeQuery=true)
- via `@NamedNativeQuery`
- specify result class in the repository method
- plain SQL that uses table names and columns
- JPA will not generate the query/queries in this case
- mostly used in case of limiting results when using joins
- gives you more control, but are DBMS specific
- Example

```
@Query(   value="select * from courses where cname like %:name% order by cname",
          nativeQuery=true)
List<Course> findCoursesOrderedByName(@Param("name") String name);
```

remark: it is possible that you will need to use a concat function when using a “like” combined with a parameter in native queries (certainly when used together with other functions):

```
select * from courses where cname like concat('%',:name,'%') order by cname
```

Querying in Spring JPA

1. @Query
2. JPQL
3. Named queries
4. Native queries
5. Changing the “select” object
6. Using DTOs and mappers

Changing the “select” object

- returned entity will change when select doesn’t fetch the full object
  - aggregate functions -> numerical value returned
  - select specific columns or group by -> List<Object[ ]> returned
- use Data Transfer Objects (DTOs) and mappers to convert to/from the “original” entity
- mainly needed when using native queries with joins, or limiting the result shown via a REST API
- better for performance, but comes with the “cost” of mapping the result

Querying in Spring JPA

1. @Query
2. JPQL
3. Named queries
4. Native queries
5. Changing the “select” object
6. Using DTOs and mappers

Aggregate functions

- **avg**

```
@Query("select avg(c.duration) from Course c")
Double getAverageDurationOfCourses();
```

- **min**

```
@Query("select min(c.duration) from Course c")
Integer getMinimumDurationOfCourses();
```

- **max**

```
@Query("select max(c.duration) from Course c")
Integer getMaximumDurationOfCourses();
```

- **sum**

```
@Query("select sum(c.duration) from Course c")
Integer getTotalDurationOfCourses();
```

- **count**

```
Integer countByDuration(int duration);
```

- **count distinct**

```
@Query("select count(distinct c.duration) from Course c")
Integer getNumberOfDifferentDurationOfCourses();
```

Limiting/Changing selected columns

- Object[ ] or List<Object[ ]> returned instead of original object
- Examples

```
@Query("select c.title, c.duration from Course c where c.id=:id")
Object[ ] getCourseTitleAndDurationById(@Param("id") int id);

@Query("select c.title, count(c) from Course c group by c.domain having count(c)>2")
List<Object[ ]> getNumberOfCoursesPerDomain();

@Query(value="p.pfname, p.plname, co.coname
                    from persons p inner join companies c on p.pa_cono=c.cono",
          nativeQuery=true)
List<Object[ ]> getPersonsWithCompanyName();
```

- will need to be mapped to a DTO
- techniques:
  - custom Mapper class
  - @Named(Native)Query and @SqlResultSetMapping
  - Interface-based DTO projections
  - Class-based DTO projections

  Only the first 2 techniques can be used for native queries!

Querying in Spring JPA

1. @Query
2. JPQL
3. Named queries
4. Native queries
5. Changing the “select” object
6. Using DTOs and mappers

Data Transfer Objects (DTOs) 6.1

- class that maps the result of a query
- provide a constructor with all arguments that will be passed
- used to limit the size of objects passed to the client (web or via JSON)
- will increase performance
- better design: separates entity layer and “view” layer objects
- will need transformation/mapping!
- Example

```
public class PersonDTO {
    private String firstName;
    private String lastName;
    private String companyName;
    public PersonDTO(String firstName, String lastName, String companyName){
        ...
    }
    ...
}
```

Querying in Spring JPA

1. @Query
2. JPQL
3. Named queries
4. Native queries
5. Changing the “select” object
6. Using DTOs and mappers

Custom Mapper Class

- user-defined class that maps the Object[ ] to a PersonDTO object
- contains a static method PersonDTO toDTO(Object[ ])
- service will then call the mapper object, and returns the PersonDTO object
- all array elements will need to be converted from Object to their correct type!

Querying in Spring JPA

1. @Query
2. JPQL
3. Named queries
4. Native queries
5. Changing the “select” object
6. Using DTOs and mappers

Custom Mapper Class - Example

```
public PersonJpaRepository extends JpaRepository<Person,Integer>{
    @Query(value="select p.pfname, p.plname, co.coname
                  from persons p inner join companies c on p.pa_cono=c.cono",
                nativeQuery=true)
    List<Object[ ]> getAllPersonsWithCompanyName();
}


public class PersonDTOMapper {
    public static PersonDTO toDTO(Object[ ] objArray){
        PersonDTO p = new PersonDTO(objArray[0].toString(),
                                    objArray[1].toString(),
                                    objArray[2].toString());
        return p;
    }
}


public class MyPersonService implements PersonService {
    @Autowired PersonJpaRepository pjr;
    public List<PersonDTO> findAllPersonsWithCompanyName(){
        List<PersonDTO> persons = new ArrayList<PersonDTO>();
        List<Object[ ]> resList = pjr.getAllPersonsWithCompanyName();
        for (Object[ ] objArray : resList){
            persons.add(PersonDTOMapper.toDTO(objArray));
        }
        return persons;
}
```

@NamedNativeQuery and @SqlResultSetMapping

- use a named (native) query on the class
- @SqlResultSetMapping then used to map the columns
- List of DTO objects can then directly be returned from the repository
- Example

```
@NamedNativeQuery(name="Person.getAllPersonsWithCompanyNameNQ",
                  query="p.pfname first, p.plname last, co.coname comp
                  from persons p inner join companies c on p.pa_cono=c.cono",
                  resultSetMapping = "PersonDtoMapping")

@SqlResultSetMapping(name = "PersonDtoMapping",
                     classes = @ConstructorResult(targetClass = PersonDto.class,
                                                  columns = {@ColumnResult(name = "first"),
                                                             @ColumnResult(name = "last"),
                                                             @ColumnResult(name = "comp"})))
@Entity
public class Person {
    ...
}


public PersonJpaRepository extends JpaRepository<Person,Integer>{
    @Query(nativeQuery=true)
    List<PersonDTO> getAllPersonsWithCompanyNameNQ();
}
```

Querying in Spring JPA

1. @Query
2. JPQL
3. Named queries
4. Native queries
5. Changing the “select” object
6. Using DTOs and mappers

Mapping JPQL query results

- Class-based DTO projections
  - use constructor call directly in the JPQL query
  - fully qualified name needed!

```
public PersonJpaRepository extends JpaRepository<Person,Integer>{
@Query
    (value="new be.abis.demo.dto.PersonDTO(p.firstName, p.lastName,
                        p.company.name) from Person p")
    List<PersonDTO> getAllPersonsWithCompanyName();
}
```

- Interface-based DTO projections
  - instead of using DTO class, use an interface with the columns in the exact order
  - watch out: easy to write, but bad performance!

```
public interface PersonDTO {
    String getFirstName();
    String getLastName();
    String getCompanyName();
}
public PersonJpaRepository extends JpaRepository<Person,Integer>{
    @Query(value="p.firstName, p.lastName,p.company.name from Person p")
    List<PersonDTO> getAllPersonsWithCompanyName();
}
```

Querying in Spring JPA

1. @Query
2. JPQL
3. Named queries
4. Native queries
5. Changing the “select” object
6. Using DTOs and mappers

Transaction Management and Exception Handling

Objectives :

- **Exception Handling**
- **Modifying queries**
- **Transaction Management**
- **Declarative and programmatic transactions**

Exception Handling

- Spring JPA predefines many (around 35 of them!) exceptions
- platform-agnostic
- super class = org.springframework.dao.DataAccessException
- descriptive names :
  IncorrectResultSizeDataAccessException,
  DataIntegrityViolationException,...
- all of them are RuntimeException -> unchecked
- still important to do correct exception handling!!!
- documentation:

  https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/dao/DataAccessException.html

Transaction Management and Exception Handling

1. Exception Handling
2. Modifying queries
3. Understanding transactions
4. Transaction Status
5. Transaction Definition
6. Declarative transactions
7. Programming transactions

Modifying queries

- JPQL/native queries can be used for DML
- add @Modifying annotation on the repository method
- obligatory to add @Transactional when calling them!
- Examples

```
public PersonJpaRepository extends JpaRepository<Person,Integer>{
    @Modifying
    @Query(value="update persons set ppwd=:newpwd where pno=:id ,
                nativeQuery=true)
    Person updatePassword(@Param("newpwd") String newpwd, @Param("id") int id);
}
public class MyPersonService implements PersonService {
    @Autowired PersonJpaRepository pjr;
    @Transactional
    public void updatePassword(String newpwd, int id){
        pjr.updatePassword(newpwd,id);
    }
}
```

Transaction Management and Exception Handling

1. Exception Handling
2. Modifying queries
3. Understanding transactions
4. Transaction Status
5. Transaction Definition
6. Declarative transactions
7. Programming transactions

Understanding transactions

- transactions ensure all-or-nothing behaviour
- data never left in inconsistent state
- ACID test
- Hibernate/JPA imposes transaction management
- global vs local transactions
  - global transactions
    - enable you to work with multiple transactional resources, typically relational databases and message queues
    - application server manages global transactions through JTA
  - local transactions
    - resource-specific, e.g. transaction linked to a JDBC connection
    - simpler, but no application server involved, so limited

Transaction Management and Exception Handling

1. Exception Handling
2. Modifying queries
3. Understanding transactions
4. Transaction Status
5. Transaction Definition
6. Declarative transactions
7. Programming transactions

Spring's Transaction Philosophy

- consistent programming model in any environment (whether global or local)
- only need an application server's JTA capability if application needs to handle transactions across multiple resources
- Spring supports both programmatic and declarative transaction management
- abstract away actual transactional implementation from code
- transaction strategy based on PlatformTransactionManager
- TransactionException is a RuntimeException
- By default everything is autocommitted, unless @Transactional is used

Transaction Management and Exception Handling

1. Exception Handling
2. Modifying queries
3. Understanding transactions
4. Transaction Status
5. Transaction Definition
6. Declarative transactions
7. Programming transactions

Transaction Managers

- different implementations of PlatformTransactionManager
- acts as facade to platform-specific transaction implementation
  - DataSourceTransactionManager (JDBC) -> dataSource
  - HibernateTransactionManager -> sessionFactory
  - JpaTransactionManager -> entityManager
  - JtaTransactionManager -> need dataSource via JNDI
- getTransaction() will return a TransactionStatus based on a TransactionDefinition
- Spring Boot automatically configures a JpaTransactionManager (if no JTA context found)
- use via
  - @Transactional (declarative transactions)
  - wired in transactionTemplate (programmatic transactions)

Transaction Management and Exception Handling

1. Exception Handling
2. Modifying queries
3. Understanding transactions
4. Transaction Status
5. Transaction Definition
6. Declarative transactions
7. Programming transactions

Transaction Status

- might represent a new transaction, or can represent an existing transaction if a matching transaction exists in the current call stack
- associated with a thread of execution
- code can use this to retrieve status information, and to programmatically request a rollback
- provides access to savepoint management facilities
- methods
  - flush(): flush the underlying session to the datastore, if applicable
  - hasSavepoint(): does transaction internally have a savepoint
  - isCompleted(): is transaction completed (commit or rollback)
  - isNewTransaction(): is present transaction new or nested
  - isRollbackOnly(): is transaction marked as rollback-only
  - setRollbackOnly(): set the transaction rollback-only.

Transaction Management and Exception Handling

1. Exception Handling
2. Modifying queries
3. Understanding transactions
4. Transaction Status
5. Transaction Definition
6. Declarative transactions
7. Programming transactions

Transaction Definition

- **isolation level**
  - **degree to which transaction is isolated from other transactions**
  - **possibilities:**
    - ISOLATION_READ_COMMITTED: dirty reads are prevented; non-repeatable reads and phantom reads can occur
    - ISOLATION_READ_UNCOMMITTED: dirty reads, non-repeatable reads and phantom reads can occur.
    - ISOLATION_REPEATABLE_READ: dirty reads and non-repeatable reads are prevented; phantom reads can occur.
    - ISOLATION_SERIALIZABLE: dirty reads, non-repeatable reads and phantom reads are prevented
    - (default!) ISOLATION_DEFAULT: use the default isolation level of the underlying datastore

Transaction Management and Exception Handling

1. Exception Handling
2. Modifying queries
3. Understanding transactions
4. Transaction Status
5. Transaction Definition
6. Declarative transactions
7. Programming transactions

Transaction Definition (cont.)

- **propagation behaviour**
  - **defines the transaction scope**
  - **Spring offers all of the transaction propagation options familiar from EJB CMT**
  - **possibilities**
    - PROPAGATION_NESTED: execute within a nested transaction if a current transaction exists, PROPAGATION_REQUIRED like behaviour else
    - PROPAGATION_NEVER: do not support a current transaction; throw an exception if a current transaction exists.
    - PROPAGATION_NOT_SUPPORTED: do not support a current transaction; rather always execute non-transactionally
    - (default!) PROPAGATION_REQUIRED: support a current transaction; create a new one if none exists.
    - PROPAGATION_REQUIRES_NEW: create a new transaction, suspending the current transaction if one exists.
    - PROPAGATION_SUPPORTS: support a current transaction; execute non-transactionally if none exists.

Transaction Definition (cont.)

- timeout
  - how long does transaction runs before timing out and being rolled back automatically
  - defined as int
  - default = - 1: use default timeout of the underlying transaction system
- read-only
  - defines whether underlying transaction is readonly
  - default = false
- rollback rules
  - defines for which Exceptions the transaction should be rolled back
  - default: RuntimeException and Error

Declarative transactions

- implemented through Spring's AOP framework
- annotation based:
  - @Transactional (org.springframework.transaction.annotation)
  - attributes define the transaction definition properties
- only need to override non-default options
- method annotations override class annotations
- use in the Service Layer for DML methods
- for delete: use myRepo.flush() such that the exception is thrown within the transaction boundary
- in JUnit tests
  - automatic rollback when putting @Transactional on class/method, use @Commit on test to change this behaviour
  - Make sure that the @Transactional annotation on the test does not change the behaviour of the method in the Service layer!
  - put @Transactional only on "happy flow" tests, exceptions should be handled from the Service layer

Transaction Management and Exception Handling

1. Exception Handling
2. Modifying queries
3. Understanding transactions
4. Transaction Status
5. Transaction Definition
6. Declarative transactions
7. Programming transactions

Example

- **AbisReceptionService.java**

```
@Service
public class AbisReceptionService implements ReceptionService {
    @Autowired
    private CompanyJpaRepository cjr;

    ...
    @Transactional(isolation = Isolation.SERIALIZABLE,
                        rollbackFor = CompanyException.class)
    public void doStuffWithCompanies() throws CompanyException{
        try {
            Company c = new Company
                    (1000, "My Company", "some street", "30", "3000", "Leuven", "B");
            cjr.save(c);
            c.setStreet("Diestsevest");
            cjr.save(c);
            System.out.println(cjr.findById(1000));
            cjr.deleteById(3);
            List<Company> companies = cjr.findAll();
            for (Company comp : companies) {System.out.println(comp);}
            cjr.flush();
        } catch (EmptyResultDataAccessException | DataIntegrityViolationException e){
                throw new CompanyException("oops");
        }
    }
}
```

Example (cont.)

- **ReceptionTest.java**

```
@SpringBootTest
public class ReceptionTest {

    @Autowired
    private ReceptionService receptionService;

    @Test
    @Transactional
    public void testdoStuffWithCompanies() {
        assertDoesNotThrow(()->receptionService.doStuffWithCompanies());
    }

    @Test
    public void testdoStuffWithCompanies() {
        assertThrows(CompanyException.class,
                                ()->receptionService.doStuffWithCompanies();
    }


}
```

Programming transactions

- control every step in the transaction yourself
- can be more fine-grained than declarative transactions
- uses TransactionTemplate that wires in TransactionManager
- implement TransactionCallback interface
- call doInTransaction() method
- wire template into your Service classes
- intrusive in your code!

Transaction Management and Exception Handling

1. Exception Handling
2. Modifying queries
3. Understanding transactions
4. Transaction Status
5. Transaction Definition
6. Declarative transactions
7. Programming transactions

Example

```
@Service
public class SimpleService implements Service {

    private final TransactionTemplate transactionTemplate;

    @Autowired
    public SimpleService(PlatformTransactionManager transactionManager) {
        this.transactionTemplate = new TransactionTemplate(transactionManager);
    }

    public Object someServiceMethod() {

        return transactionTemplate.execute(new TransactionCallback() {

            public Object doInTransaction(TransactionStatus status) {
                try {
                    updateOperation1();
                    updateOperation2();
                } catch (SomeBusinessExeption ex) {
                    status.setRollbackOnly();
                }
            }

        });

    }

}
```

Transaction Management and Exception Handling

1. Exception Handling
2. Modifying queries
3. Understanding transactions
4. Transaction Status
5. Transaction Definition
6. Declarative transactions
7. Programming transactions

Advanced Topics

Objectives :

- **Mapping collections**
- **Hierarchy mapping**

Advanced Topics

1. Mapping collections
2. Hierarchy mapping

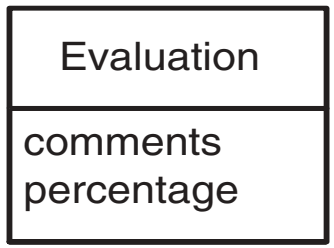
Difference between:

- collections of entity types

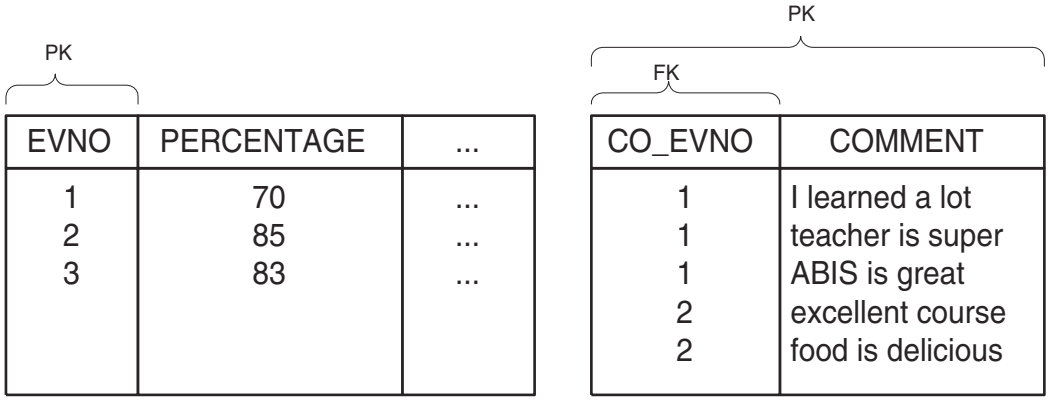
  mapped with true entity relationships

  - `@OneToMany`: see 3.3 One-to-many mapping on page 77
  - `@ManyToMany`: see 3.4 Many-to-many associations on page 81
  - mapped with collections semantics: not recommended, requires separate collections table (not covered here)
- collections of value types
  - different possibilities: set, list, map -> in separate table
  - use `@ElementCollection` and `@CollectionTable`

Example



Set



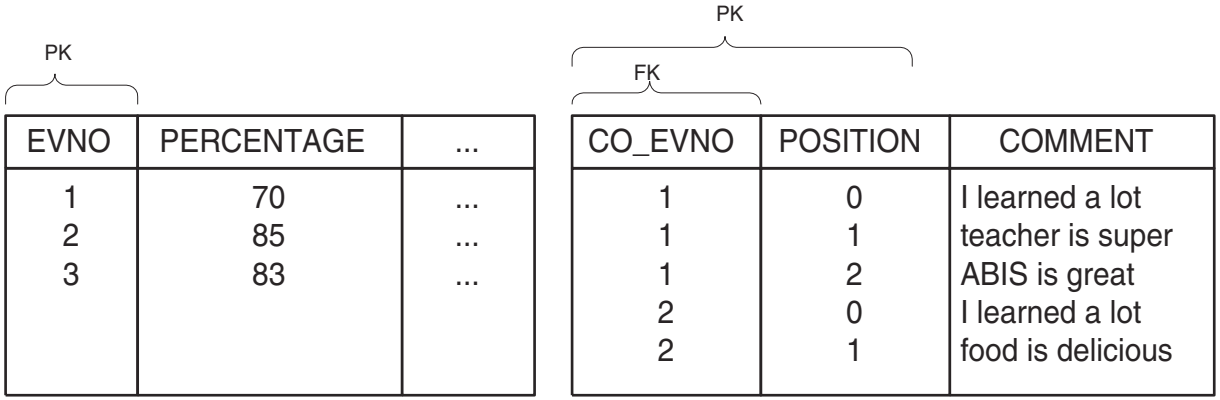
- **Java class is a `Set` implementation**
- **Set cannot contain duplicate elements -> two times same comment is impossible for same evaluation**

```
@Entity
public class Evaluation {

    ...
    @ElementCollection
    @CollectionTable(name="Comments",joinColumns=@JoinColumn(name="co_evno"))
    @Column(name="comment")
    private Set<String> comments;

    ...
}
```

Advanced Topics

1. Mapping collections
2. Hierarchy mapping

List



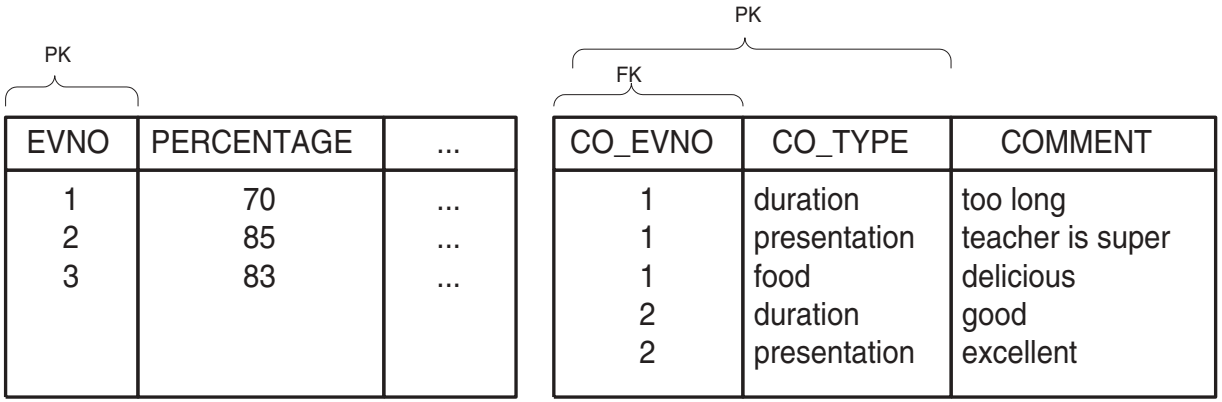
EVNO	PERCENTAGE	...
1	70	...
2	85	...
3	83	...

CO_EVNO	POSITION	COMMENT
1	0	I learned a lot
1	1	teacher is super
1	2	ABIS is great
2	0	I learned a lot
2	1	food is delicious

- Java class is a `List` implementation (e.g. `ArrayList`)
- duplicates are allowed
- preserves order

```
@Entity
public class Evaluation {

    ...
    @ElementCollection
    @CollectionTable(name="Comments",joinColumns=@JoinColumn(name="co_evno"))
    @OrderColumn(name="position")
    @Column(name="comment")
    private List<String> comments;
    ...
}
```

Map



EVNO	PERCENTAGE	...
1	70	...
2	85	...
3	83	...

CO_EVNO	CO_TYPE	COMMENT
1	duration	too long
1	presentation	teacher is super
1	food	delicious
2	duration	good
2	presentation	excellent

- **Java class is a *unordered* `Map`**
- **Key/value pair**
  - **key: basic, embeddable, or entity type**
    - unique
    - hashCode() and equals()
  - **value:**
    - basic or embeddable type: use @ElementCollection
    - entity type: use @OneToMany or @ManyToMany

Map - annotations

Key column specification

- @MapKeyColumn(name=...)

  default name: attribute_KEY

- @MapKeyEnumerated(EnumType.STRING)

  use enumerated type key with string names

- @MapKeyTemporal(DATE | TIME | TIMESTAMP)

  use Date type as key

- @MapKeyJoinColumn(name=...)

  use entity as key

Value column specification

- @Column(name=...)

Map - example

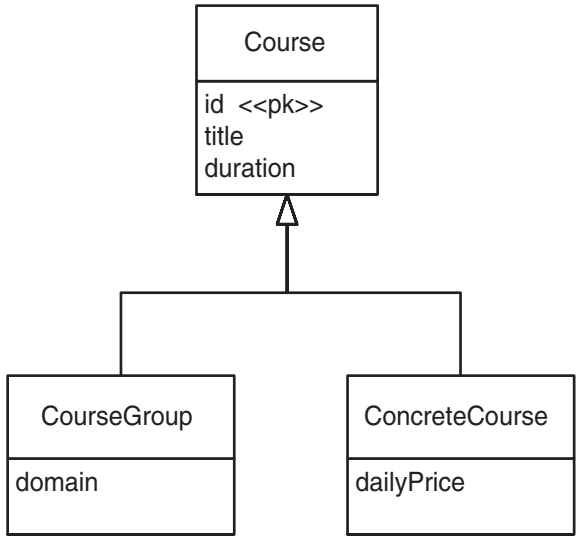
```
@Entity
public class Evaluation {
    ...
    @ElementCollection
    @CollectionTable(name="Comments")
    @MapKeyColumn(name = "co_type")
    @MapKeyEnumerated(EnumType.String)
    @Column(name = "comment")
    private Map<CommentType,String> comments;
    ...
}
public enum CommentType {
    DURATION,
    PRESENTATION,
    INFRASTRUCTURE,
    FOOD
}
```

Hierarchy mapping

Advanced Topics

1. Mapping collections
2. Hierarchy mapping

Hierarchical relations between entities not always supported in a relational database



Three alternatives:

1. table per class hierarchy (default)

2. table per concrete class

3. table per subclass (joined)

Table per class hierarchy

‘Single table’ inheritance

courses
id <<pk>>
title
duration
domain
dprice
ctype <<discriminator>>

- **define `@Entity` on each class in hierarchy**
- **define `@Inheritance` on root class and use `InheritanceType.SINGLE_TABLE`**
- **define `@DiscriminatorColumn` to define column name and type**
- **define `@DiscriminatorValue` in each class to define column value**

  default value = class name

notice discriminator (DTYPE) column in table

internal column used by JPA

-> no normalisation in RDBMS, nulls allowed in table!

Table per class hierarchy - example

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn (name="ctype", discriminatorType=DiscriminatorType.STRING)
@DiscriminatorValue("co")
@Table(name="Courses")
public class Course {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name="id")
    private int id;

    @Column(name="title")
    private String title;

    @Column(name="duration")
    private int duration;

    // DO NOT MENTION THE DISCRIMINATOR COLUMN AS PROPERTY !!!

    // constructors + getters/setters

    public String getCourseType(){
        return this.getClass().getAnnotation(DiscriminatorValue.class).value();
    }


}
```

Table per class hierarchy - example (cont.)

```
@Entity
@DiscriminatorValue("cg")
public class CourseGroup extends Course {
    @Column(name="domain")
    private String domain;
    // constructors + getter/setter
}

@Entity
@DiscriminatorValue("cc")
public class ConcreteCourse extends Course {
    @Column(name="dprice")
    private double dailyPrice;
    // constructors + getter/setter
}

public interface CourseJpaRepository extends JpaRepository<Course,Integer> {
    Course findById(int id);
    Query(value = "select ctype from courses where id=:courseId", nativeQuery = true)
    String findCourseType(@Param("courseId") int courseId);
}
```

Table per class hierarchy - example (cont.)

```
public interface CourseService {
    Course findCourseById(int id);
    Course addCourse(Course course);
    long count();
    String findCourseType(int courseId);
}


@Service
public class AbisCourseService {
    @Autowired
    CourseJpaRepository courseRepository;
    public Course findCourseById(int id){
        courseRepository.findById(id);
    }
    @Transactional
    public Course addCourse(Course course){
        return courseRepository.save(course);
    }
    public long count(){
        return courseRepository.count();
    }
    public String findCourseType(int id){
        return courseRepository.findCourseType(id);
    }
}
```

Table per class hierarchy - example (cont.)

```
@SpringBootTest
public class CourseServiceTest {
    @Autowired
    CourseService courseService;
    @Test
    public void course1IsAconcreteCourse(){
        Course found = courseRepository.findCourseById(1);
        assertTrue(found instanceof ConcreteCourse);
    }
    @Test
    @Transactional
    public void addCourseGroupWorks(){
        CourseGroup groupToAdd = new CourseGroup("new title", 5, "Java");
        long coursesBefore = courseService.count();
        Course added = courseService.addCourse(groupToAdd);
        long coursesAfter = courseService.count();
        assertEquals(1, coursesAfter-coursesBefore);
        assertEquals("cg", added.getCourseType());
        assertEquals("cg", courseService.findCourseType(added.getId()));
}
```

Table per concrete class

1. Union subclass

coursegroups
id <<pk>>
title
duration
domain

concretecourses
id <<pk>>
title
duration
dprice

- define `@Entity` (and `@Table`) on each class in hierarchy
- define `@Inheritance` on root class and use `InheritanceType.TABLE_PER_CLASS`
- parent class properties and id are inherited in child classes

Notes:

- Duplicate columns in different tables!
- do not use AUTO nor IDENTITY generation for the key field
- no polymorphic queries

Table per concrete class (union subclass) - example

```
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public class Course {
    ...
}

@Entity
@Table(name="CourseGroup")
public class CourseGroup extends Course {
    ...
}

@Entity
@Table(name="ConcreteCourse")
public class ConcreteCourse extends Course {
    ...
}
```

Table per concrete class

2. Implicit polymorphism

- parent class is not mapped (no `@Entity`). Use `@MappedSuperclass`
- parent properties are mapped/annotated
- define `@Entity` (and `@Table`) and map properties on each subclass in hierarchy
- parent class properties and id are inherited in child classes

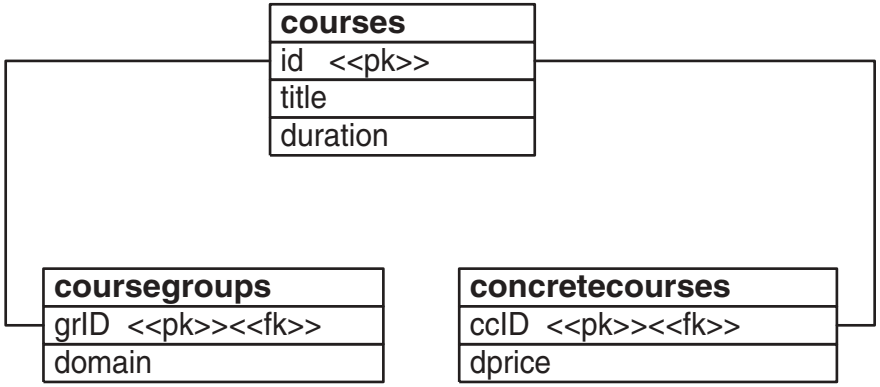
Notes:

- parent columns duplicated in different tables!
- do not use AUTO nor IDENTITY generation for the key field

Table per concrete class (implicit polymorphism) - example

```
@MappedSuperclass
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public class Course {
    ...
}
@Entity
@Table(name="CourseGroup")
public class CourseGroup extends Course {
    ...
}

@Entity
@Table(name="ConcreteCourse")
public class ConcreteCourse extends Course {
    ...
}
```

Table per subclass (joined)



- **define `@Entity` on each class in hierarchy**
- **define `@Inheritance` on root class and use `InheritanceType.JOINED`**
- **adds key in subclass**

  key can have the same name in all classes
  `@PrimaryKeyJoinColumn(name="...")`
- **create PK-FK relationships in database**

JPA uses outer join to access data!

Table per subclass (join) - example

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
@Table(name="Courses")
public class Course {
    @Id
    @Column(name="cid")
    private int courseNr;
    ...
}

@Entity
@PrimaryKeyJoinColumn(name="cid")
public class CourseGroup extends Course {
    ...
}
@Entity
@PrimaryKeyJoinColumn(name="cid")
public class ConcreteCourse extends Course {
    ...
}
```

Choose a hierarchy mapping strategy

No polymorphic queries or associations needed (e.g. if superclass is abstract)

- **prefer table-per-concrete-class**

Polymorphic queries or associations needed

- **not too many subclasses and not too many attributes in subclasses**

  -> table-per-class-hierarchy

  -> lot of nulls in table

- **many subclasses or many attributes in subclasses**

  -> table-per-subclass (join)

Note: it is possible to mix strategies (but DON'T use mixing)

APPENDIX A. EXERCISES

1 _____ Configuration

1. Clone the repository https://github.com/sschillebeeckx/springjpa.
   Copy the personsJpa.csv file under the c:\temp\javacourses directory.

   Check out the code. Run the JUnit tests. Make sure that they work. If you want, you can also test the API via Postman (as far as possible).
2.
   a. Add dependencies for spring-boot-starter-jpa and an Oracle driver in the pom.xml
   b. Add following DB settings in a application.properties file:
      Replace the "xx" by a number given to you by your instructor.

```
spring.datasource.url = jdbc:oracle:thin:@//delphi.abis.be:1521/TSTA
spring.datasource.username= tu000xx
spring.datasource.password= tu000xx
spring.datasource.driver-class-name=oracle.jdbc.OracleDriver

spring.jpa.hibernate.ddl-auto=none
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
```

   c. Add a DataSourceTest class, which checks whether the connection was set up correctly.
3.
   a. In the database view of IntelliJ, create a new Oracle datasource, by copying the datasource url. Go to advanced->expert options, and make sure that the "introspect using JDBC metadata" option is enabled.
   b. Run the createOracleDB.sql script via the query console. Check the tables via the explorer and perform some basic queries in the console.

2 ____ Entity mappings and Basic JPA repositories

1.
   a. Add correct mappings in the Course class to mimic the DB correctly.
      Use GenerationType.IDENTITY for the courseId.
   b. Replace the CourseRepository interface with a CourseJpaRepository, which has to be able to call the same methods as before. Use provided methods and query methods for this. Make sure to fetch a course (by id), before you update it.
      Wire the new repository into the CourseService class, and make sure that all methods are implemented. Let the addCourse() and updateCourse() methods return a Course object as from now.
   c. Use the CourseServiceTest class to test all methods (including the DML ones). What happens in the DB? Now add @Transactional on the DML tests, and check again. Also make sure that when running all tests, they nicely run in isolation. Don't use @Order annotations anymore.
   d. Test via the Course API. Take a look at the generated SQL via the console in IntelliJ. When adding a new course, make sure not to pass the id anymore!

3 _____ Association Mapping and @Query

1.
   a. Add mappings in the Company and Address classes to map to the companies table.
   b. Create a CompanyJpaRepository class. Make sure to add methods to find a company by id and to find by name.
   c. Define a method to find a company by name and town. Use @Query with JPQL for this.
   d. Use a CompanyRepositoryTest class to test these methods, as well as the DML methods.
2.
   a. Add the correct mappings in the Person class to mimic the DB correctly. Replace the PersonRepository with a PersonJpaRepository, which has to be able to call the same methods as before.
   b. Wire this in the PersonService. The "add" and "update" methods should return a Person object.
   c. Test via JUnit. Consider following use cases when adding a person:
      - person working for a new company
      - person working for an existing company
      - person working for new "location" of a company
      - person without a company
   d. Test via the API.
3. Add mappings in the Session class and create a SessionJpaRepository. Wire the repository in the AbisTrainingService. Implement the *findSessionsForCourse(String courseTitle)* method. Don't bother about cancelled sessions yet. Use JPQL. Add a JUnit test, and also test via the session API via the URL: *localhost:8085/api/sessions/query?title=DB2BAS* .
4. Add a *List<Person> employees* in the Company class and use a correct mapping for this. Add a test in the CompanyRepositoryTest class, which finds the employees of a company after calling *findCompanyById(int id)*. The other JUnit tests should also still work of course.
   What happens when you now call *http://localhost:8085/api/persons/1* via the API?

4 ____ More querying techniques and performance

1. Use lazy loading on the OneToMany relationship, but add a "fetch join" on the *findCompanyById(int id)* method. Also solve the API issue.
2.
   a. Add a PersonDTO class, such that the API will only show a person's id, first name, last name, email, birthdate, company name and company town.
   b. Provide a PersonForm class, which passes all person data (including company and address), but in a more flat structure.
   c. Make sure that all (necessary) JSON annotations are removed from the Person class, and now only appear in the PersonForm and PersonDTO classes.
   d. Foresee a PersonMapper class that transforms a Person object to and from the Form/DTO objects. Call the mapper in the PersonAPIController.
3. Check out the queries generated in exercise 3.3. Try to improve the performance by using a native query (with select *) in the SessionJpaRepository. Return a SessionDTO object from the TrainingController, which contains the session number, start date, instructor's first name and last name, location company name + town, kind, cancelled indicator and long course title. Make sure the query now only returns the non-cancelled sessions.
4.
   a. Add mappings in the Enrolment class. Use a "composite" id for the session and enrolmentInSession columns.
   b. Add a NamedQuery to find all (non-cancelled) enrolments for a person. Mention the person's first name and last name, his/her company name, the startdate of the session, the company name+town where the session takes place and the course title in the result. Use a native SQL query.
   c. Create an EnrolmentJpaRepository that calls the named query. Implement the *List<EnrolmentDTO> findEnrolments(int personId)* method in the AbisTrainingService and the TrainingApiController.
   d. Add a JUnit test in the TrainingServiceTest class and test the API.

5 _____ Exception Handling and Transactions

1. Implement the PersonNotFoundException, PersonCannotBeDeletedException (thrown when there are integrity violations) and PersonAlreadyExistsException (thrown when the person's emailaddress already exists in the DB) mentioned in the PersonService. Test via JUnit (and Postman).
2. Add a method in the SessionJpaRepository to cancel a session. Make sure to use a modifying, native query to do this, which only changes the scancel column. Also add the necessary in the AbisTrainingService class and test via JUnit.
   Extra: add a method in the TrainingApiController and test via Postman.
3. Implement the method *enrolForSession(Person person, int sessionId)* in the AbisTrainingService class. Make sure it is transactional. Throw the EnrolException if something goes wrong, and make sure everything is rolled back in that case. Add JUnit test cases in the TrainingServiceTest.
   Extra: implement the method in the TrainingApiController and test via Postman.

6 _____ Advanced Topics

1. Add List<**String**> hobbies in the Person class, linking to the *hobbies* table.
   Add methods to retrieve the hobbies of a person, and to add a new hobby. Test via JUnit.
2. Create 2 subclasses of the Session class: CompanySession and PublicSession. The classes can be left "empty" for now. Make sure that the *skind* column is used as the discriminator column. Test via the return type of a *findById(int id)* method in the SessionJpaRepository class. Also try to add a companySession object in the table.