Java Programming 3

Week 9: JPA

KdG

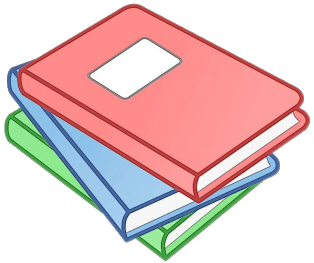
Agenda this week

What is ORM?
Spring configuration
The @Entity
CRUD operations with JPA
Implementing the repository
Relations with JPA

KdG

Tutorials

- JPA and Hibernate: https://www.baeldung.com/learn-jpa-hibernate
- Entities: https://www.baeldung.com/jpa-entities
- JPQL examples: https://docs.oracle.com/javaee/6/tutorial/doc/bnbtl.html
- Relationships: https://stackabuse.com/a-guide-to-jpa-with-hibernate-relationship-mapping/



KdG

Agenda this week

What is ORM?
Spring configuration
The @Entity
CRUD operations with JPA
Implementing the repository
Relations with JPA

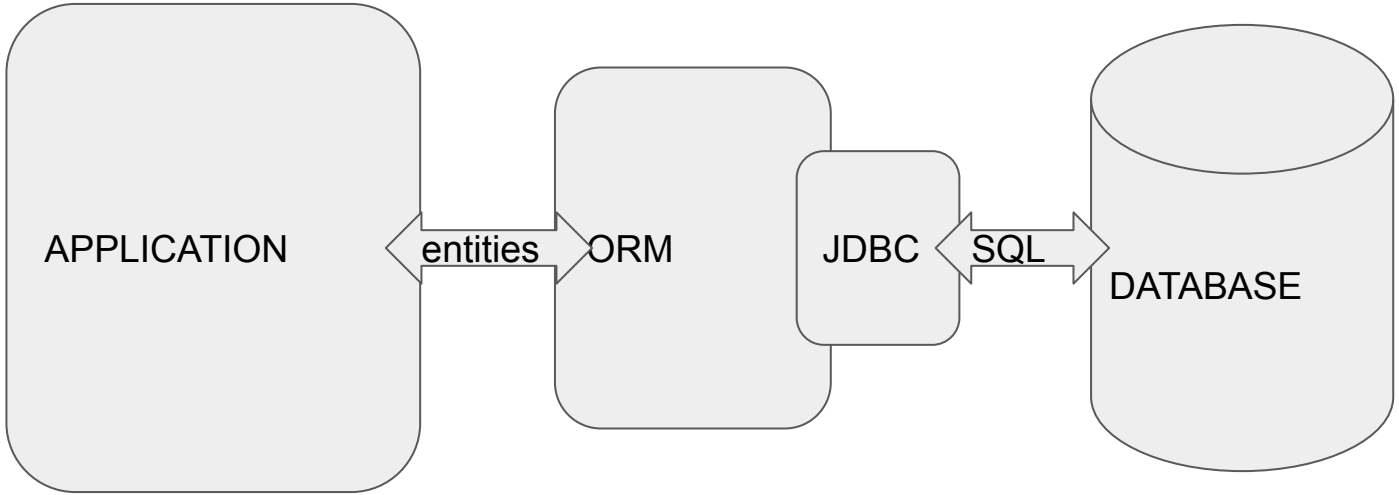
KdG

What is ORM?

- Object Relational Mapping
  - We use a framework that maps the objects from our domain model to tables in the relational database.
  - The application queries and updates the objects from the domain model instead of talking directly to the database
  - The ORM framework keeps the objects in sync with the database.
- JAP (“Jakarta Persistence API”) is a standard API defined for ORM
- Hibernate provides an implementation of this standard.

KdG

ORM

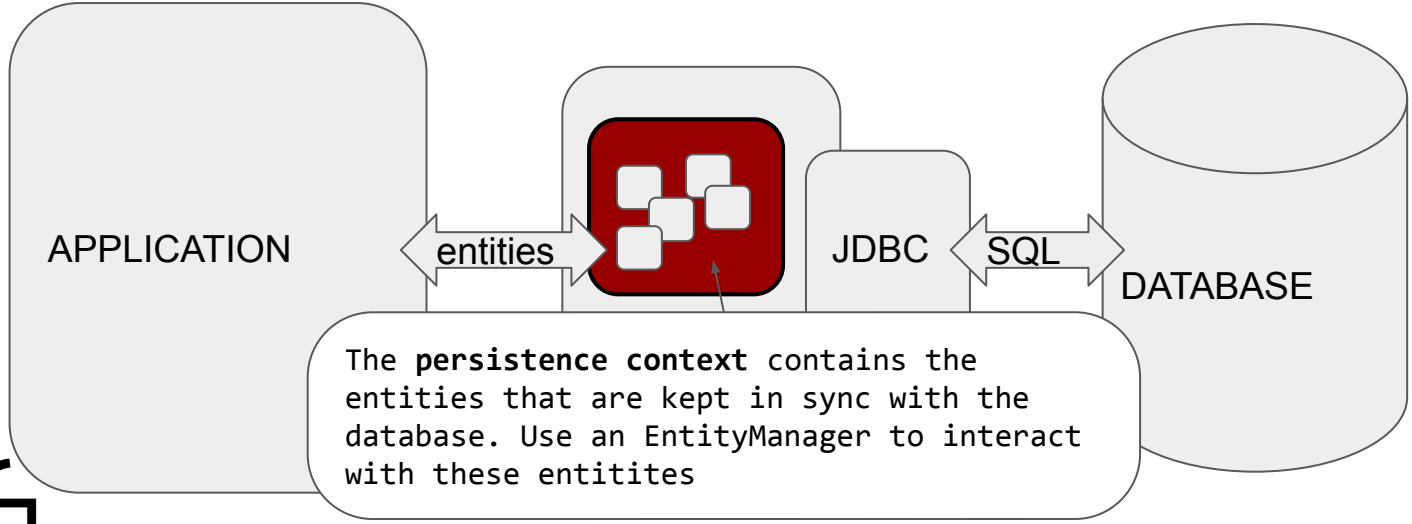


ORM Provides an extra abstraction layer on top of JDBC. The application uses the *entities* (= domain objects) to create, read, update and delete records from the database.

KdG

Persistence Context - EntityManager

- All the entities that are synced with the database reside in the *persistence context*
- Via an EntityManager you can manage the persistence context: you can create, remove, find and query entities.



Entities

- Domain objects that map to a database table get the @Entity annotation
- An @Entity needs
  - An id (@Id). It maps to the primary key of the table
  - A no arguments constructor (can be protected)

```
@Entity
public class Book {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private int id;

   private String title;

   protected Book() {
   }

   //other constructors, getters, setters, toString, ...
```

KdG

Agenda this week

What is ORM?
Spring configuration
The @Entity
CRUD operations with JPA
Implementing the repository
Relations with JPA

KdG

How to configure Spring?

- Add the **Spring Data JPA** and **Postgresql** driver dependencies
- Add following lines to the application.properties:

```
spring.datasource.url=jdbc:postgresql:demospring
spring.jpa.properties.hibernate.default_schema=demo_orm
spring.datasource.driverClassName=org.postgresql.Driver
spring.datasource.username=postgres
spring.datasource.password=postgres
spring.sql.init.mode=always
spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect
spring.jpa.hibernate.ddl-auto=create
spring.jpa.show-sql=true
```

This is the datasource configuration. We are using Postgresql this time. I'm using a schema in the database. Adapt to your Postgres configuration.

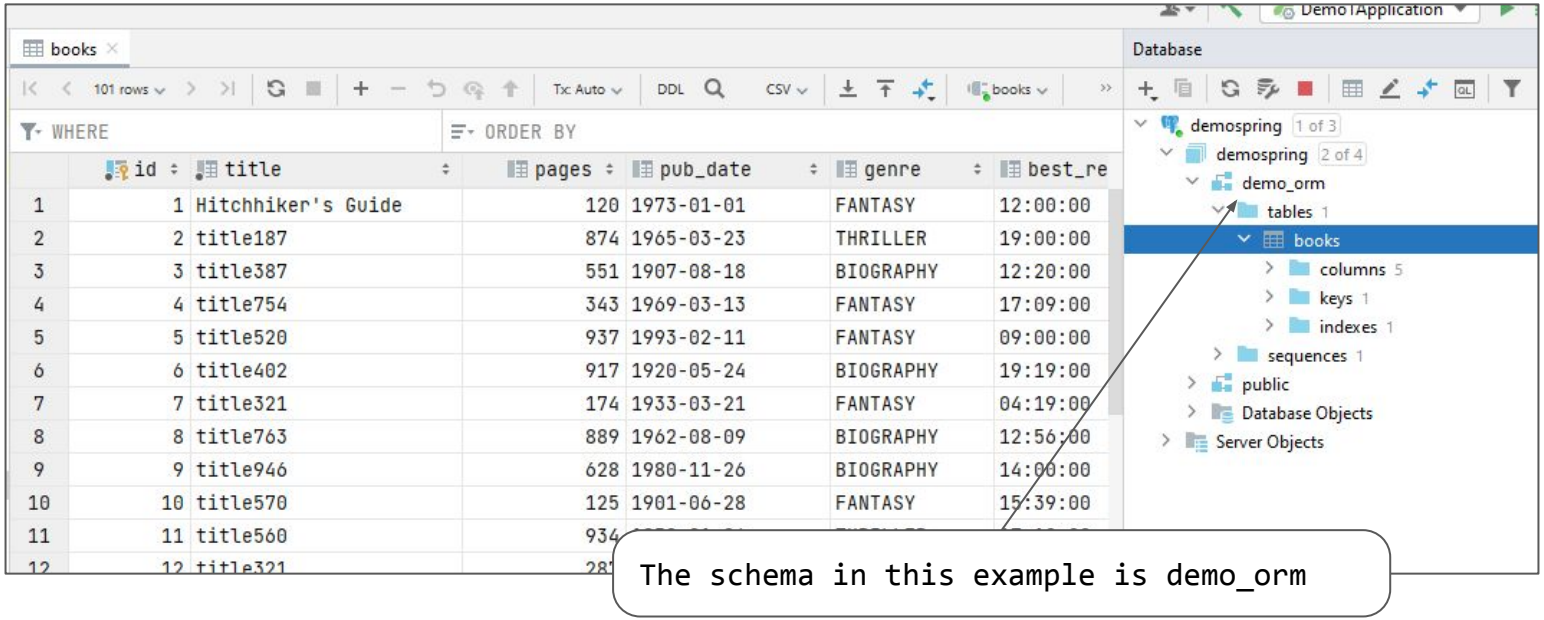
This tells hibernate what SQL dialect he has to use...

This will each time you start the application, re-create the database tables. **To be removed in production!!!**

This will show all executed SQL statements. Good for debugging!

KdG

Use the IntelliJ database tool to inspect the database



KdG

The EntityManagerFactory

- The create an EntityManager you need an EntityManagerFactory
- Spring injects it when you annotate it with @PeristenceUnit

```
@Component
public class DemoRunner implements CommandLineRunner {
   @PersistenceUnit
   private EntityManagerFactory entityManagerFactory;

   @Override
   public void run(String... args) throws Exception {
       EntityManager entityManager =
                 entityManagerFactory.createEntityManager();
       //...use the entitymanager to find entities etc...
```

KdG

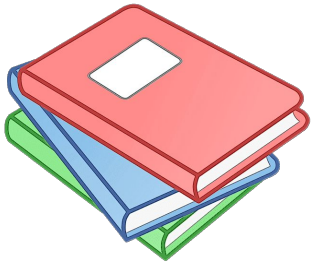
Agenda this week

What is ORM?
Spring configuration
The @Entity
CRUD operations with JPA
Implementing the repository
Relations with JPA

KdG

Entity details

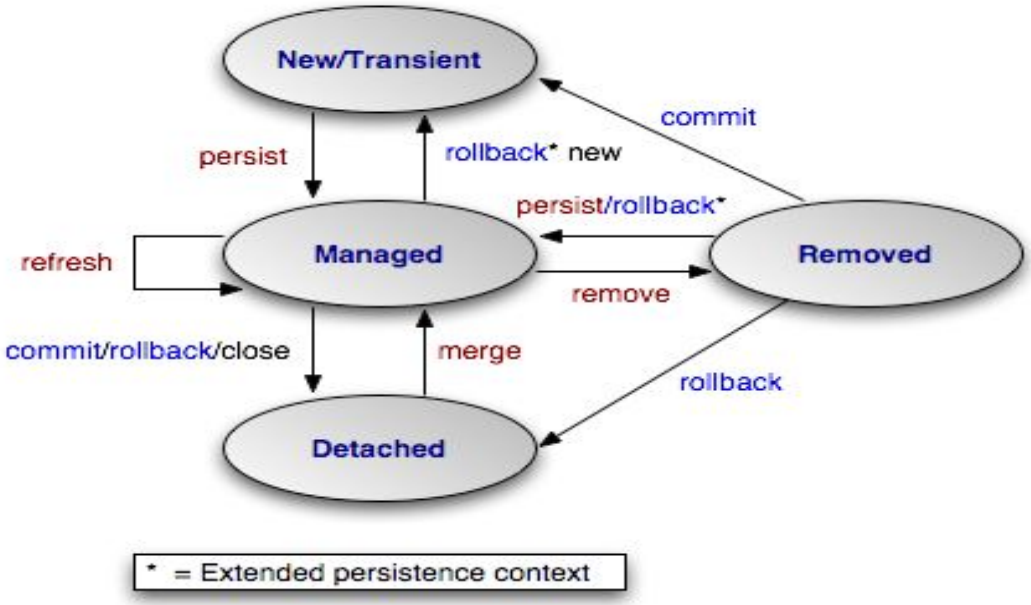
- By default the entity maps to the table with the same name, you can change via @Table(name=”table_name”)
- By default the attributes map to the table column with the same name, you can change via @Column(name=”column_name”)
- For more details: https://www.baeldung.com/jpa-entities



KdG

Entity Lifecycle

- An entity can have different states: **New/Transient, Managed, Detached, Removed**
- ***managed*** entities will sync memory data changes to the database
- When the `EntityManager` is closed/committed the entity becomes *detached*.
- It can become *managed* again via the `merge` operation
- **`Refresh`** will reload the data from the database into the managed entities

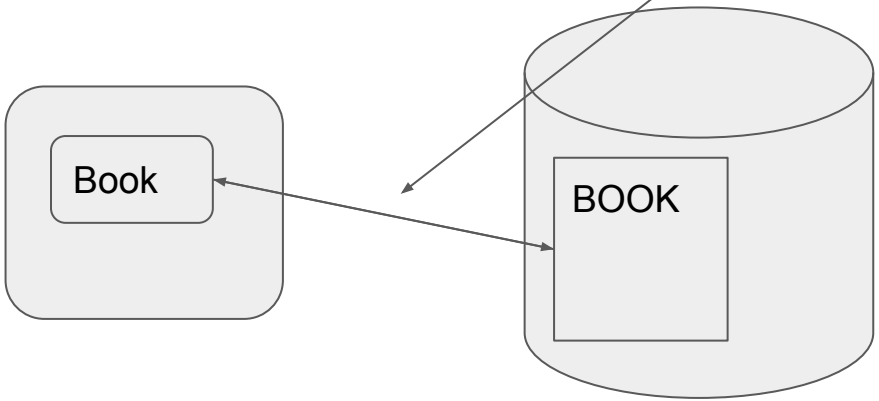


KdG

Example

```
EntityManager em = entityManagerFactory.createEntityManager();
em.getTransaction().begin();
Book book = em.find(Book.class, 23);
book.setGenre(Genre.FANTASY);
em.getTransaction().commit();
em.close();
```

The book entity resides in the persistence context. It is in a managed state. Any change to it will be synced to the database when the transaction is committed.



KdG

Use the @Entity to make it an entity

- Use @GeneratedValue to tell JPA that the database will generate the key

```
@Entity
@Table(name="books")
public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType. IDENTITY)
    private int id;

    protected Book(){}
    //...
```

Entity is saved to table "books" (default is "book"

Mandatory @Entity, @Id and no arguments constructor

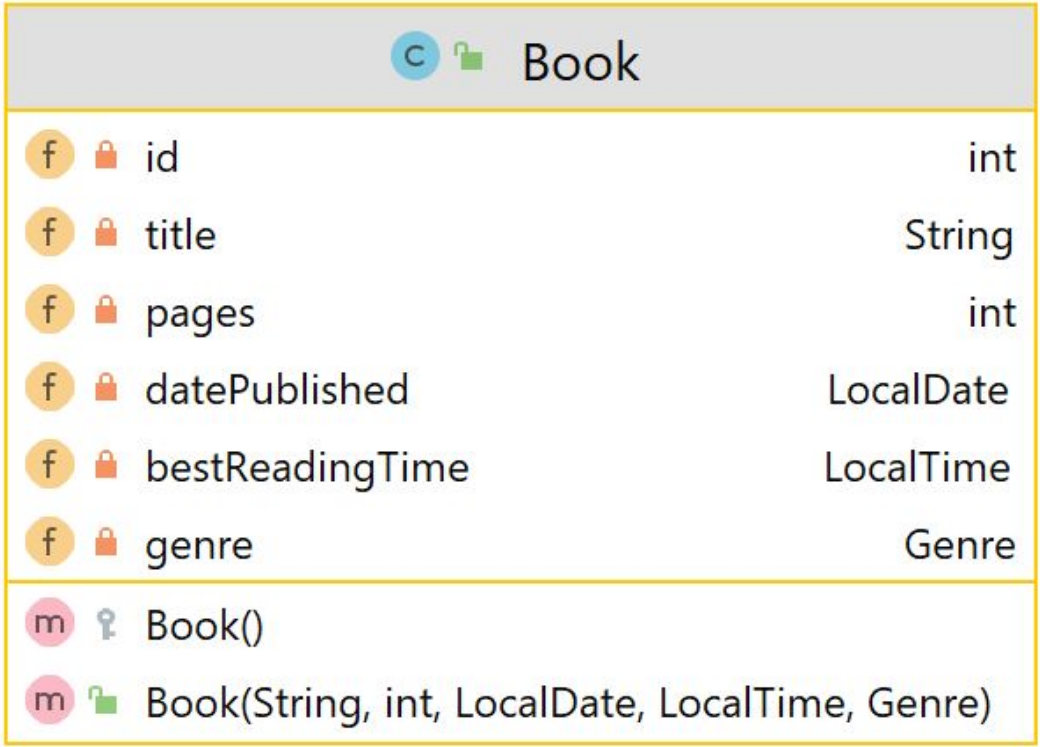
KdG

Exercise 1:

- create a Spring application with the Spring Data JPA and H2 dependencies
- configure it to use PostgreSQL
- inject an EntityManagerFactory using the @PersistenceUnit annotation
- create a Book entity class (with @Entity annotation)
- Start your application and inspect the hibernate SQL code

KdG

The example entity: Book



KdG

Annotate the fields

- Use @Column to change the column name if it does not match the name of the attribute
- You can also specify properties of the column

```
@Entity
@Table(name="books")
public class Book {
   //...

   @Column(name = "title", nullable = false, length = 50)
   private String title;

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

   //...
```

KdG

What about dates and times?

- LocalDate and LocalTime will be converted to SQL date and time...

```
@Entity
@Table(name="books")
public class Book {

   //...

   @Column(name = "pub_date")
   private LocalDate datePublished;

   @Column(name = "best_reading_time")
   private LocalTime bestReadingTime;

   //...
```

KdG

And enumerations?

- Use the @Enumerated annotation
- By default the ordinal is saved, but you can specify to save the value instead...

```
public enum Genre {
   THRILLER, BIOGRAPHY,
FANTASY
}
```

```
@Entity
@Table(name="books")
public class Book {

   //...

   @Enumerated(EnumType. STRING)
   private Genre genre;

   //...
```

KdG

Resulting database table

- Because of the `spring.jpa.hibernate.ddl-auto=create (or create-drop)` in the application.properties, the database is re-created each time.

```
-- auto-generated definition
CREATE TABLE books
(
   id                 SERIAL
       CONSTRAINT books_pkey PRIMARY KEY,
   best_reading_time TIME,
   pub_date           DATE,
   genre              VARCHAR(255),
   pages              INTEGER,
   title              VARCHAR(50) NOT NULL
);
```

KdG

Exercise 2:

- Add fields to the book entity:
  - A title, which is not nullable and has max length of 50
  - A LocalDate field and LocalTime field
  - An enumeration
- Start the application and inspect the generated database table...

KdG

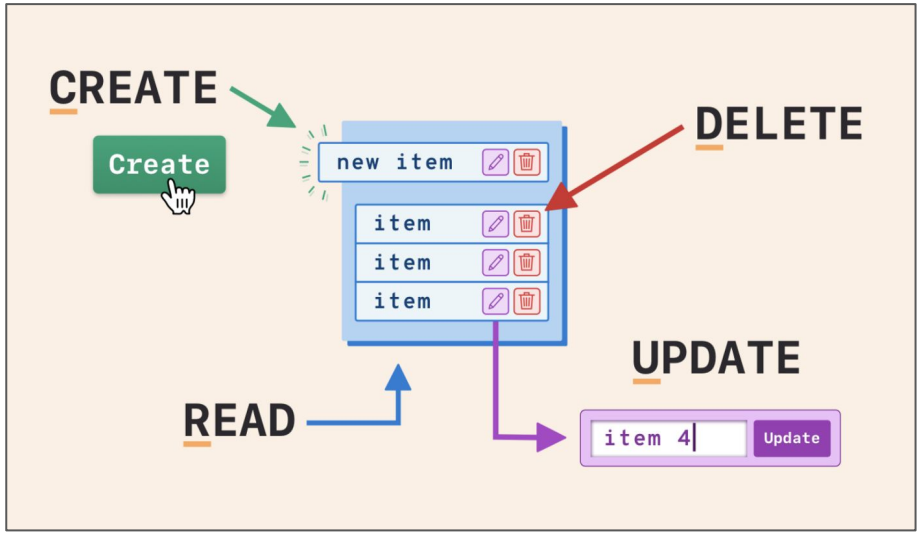
Agenda this week

What is ORM?
Spring configuration
The @Entity
CRUD operations with JPA
Implementing the repository
Relations with JPA

KdG

JPA CRUD operations

- We will have a look at how the different CRUD operations are defined in JPA:
  - Creating and saving objects
  - Reading objects
  - Updating objects
  - Deleting objects



KdG

Create an entity and save to the database

```
Book book = new Book("Hitchhiker's Guide", 120,
       LocalDate.of(1973, 1, 1),
       LocalTime.NOON,
       Genre.FANTASY);
try(EntityManager entityManager =
entityManagerFactory.createEntityManager()){
  entityManager.getTransaction().begin();
  entityManager.persist(book);
  entityManager.getTransaction().commit();
```

All operations on the persistence context are done using transactions: you first begin the transaction, then perform the operations and finally commit (or rollback) the transaction.

The persist method saves the entity to the database. Hibernate knows how to map it to the database using the annotations.

Try with resources since JPA 3.1 / Spring boot 3
For earlier versions call:

```
entityManager.close();
```

KdG

Primary key?

```
@Entity
@Table(name="books")
public class Book {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private int id;

   //...
```

We let the database generate the primary key values.
We can chose different strategies:

- GenerationType.AUTO
- GenerationType.IDENTITY
- GenerationType.UUID
- GenerationType.SEQUENCE
- GenerationType.TABLE

It depends on the database system but we will mostly use GenerationType.IDENTITY

It is possible to create your own key generation strategy

KdG

The book has a key now!

```
Book book = new Book("Hitchhiker's Guide", 120,
       LocalDate.of(1973, 1, 1),
       LocalTime.NOON,
       Genre.FANTASY);
try(EntityManager entityManager = EntityManagerFactory.createEntityManager()){
  entityManager.getTransaction().begin();
  entityManager.persist(book);
  logger.info("Created book:" + book);
  entityManager.getTransaction().commit();
}
```

The book is saved to the database, got an id there and the id is synced back to the entity...

Created book:Book{**id=1**, title='Hitchhiker's Guide', pages=120, ...

KdG

Add 100 objects to the database...

```
try (EntityManager entityManager = entityManagerFactory.createEntityManager())
{
   entityManager.getTransaction().begin();
   Stream.generate(Book::randomBook).limit(100)
         .forEach(entityManager::persist);
   entityManager.getTransaction().commit();
}
```

This code will add 100 random books to the database. I used a factory method in my entity class called randomBook to generate a random book.

```
public static Book randomBook() {
   Random random = new Random();
   return new Book("title" + random.nextInt(1000),
           random.nextInt(1000),
           LocalDate.ofEpochDay(random.nextLong(-10000,10000))
, random.nextInt(12) + 1, random.nextInt(28) + 1),
           LocalTime.of(random.nextInt(24), random.nextInt(60)),
           Genre.values()[random.nextInt(Genre.values().length)]);
}
```

Exercise 3:

- Create a static factory to generate random books
- Use the EnityManager to save some books to the database
- Inspect the data using the Database tool
- Is it possible to load initial data?
  - Yes, you can still use data.sql (and schema.sql)!
  - But you need to set (in application.properties):

```
spring.sql.init.mode=always

spring.jpa.defer-datasource-initialization=true
```

- Load some initial Books into the database using data.sql…

KdG

Retrieve an object from the database

```
try(EntityManager em = entityManagerFactory.createEntityManager()){
  //em.getTransaction().begin();
  book = em.find(Book.class, 23);
  logger.log("Found book:" + book);
  //em.getTransaction().commit();
}
```

We can retrieve an object by it's key. If the object is not found, this will return null

If we only retrieve data from the database, no transactions are needed!

KdG

Query entities from the persistence context

- You can use JPQL (JPA Query Language) to perform queries

```
try(EntityManager em = entityManagerFactory.createEntityManager()){
  List books = em
    .createQuery("SELECT b FROM Book b WHERE b.title LIKE 'title1%'")
    .getResultList();
  books.forEach(System.out::println);
}
```

```
The syntax of JPQL is very similar to SQL
syntax, except: you are querying Java
objects, not database tables!
You can even chan single valued
attributes further e.g
WHERE b.author.name like 'row%'
```

Examples of JPQL queries:https://docs.oracle.com/javaee/6/tutorial/doc/bnbtl.html

KdG

Query with parameters

- You can use JPQL (JPA Query Language) to perform queries

```
try (em = entityManagerFactory.createEntityManager()){
em.getTransaction().begin();
Query query = em.createQuery("select b from Book b where b.title = :title");
query.setParameter("title","Hitchhiker's Guide");
books = query.getResultList();
em.getTransaction().commit();
books.forEach(System.out::println);
}
```

Use a parameter, similar to PreparedStatements in JDBC...

Examples of JPQL queries:https://docs.oracle.com/javaee/6/tutorial/doc/bnbtl.html

KdG

Exercise 4:

- Check the JPA Query Language examples on https://www.tutorialspoint.com/jpa/jpa_jpql.htm
- Now try to perform some queries on your books:
  - Retrieve all thrillers
  - Find maximum number of pages a book has
  - Show all books ordered by date published
  - Show all books published before 1985
  - …

KdG

Update an object from the database

```
try(EntityManager em = entityManagerFactory.createEntityManager()){
  em.getTransaction().begin();
  book = em.find(Book.class, 23);
  //book is still managed by the EnityManager!
  book.setGenre(Genre.FANTASY);
  em.getTransaction().commit();
}
```

While the transaction is still open, the object is *managed* by the EntityManager. We can use setters to change the object. If we commit the transaction, the change will be saved to the database!

KdG

Update a *managed* object from the database

```
try(EntityManager em = entityManagerFactory.createEntityManager()){
  em.getTransaction().begin();
  book = em.find(Book.class, 23);
  //book is still managed by the EnityManager!
  book.setGenre(Genre.FANTASY);
  em.getTransaction().commit();
}
```

While the transaction is still open, the object is *managed* by the EntityManager. We can use setters to change the object. If we commit the transaction, the change will be saved to the database!

KdG

Update a *detached* object from the database

```
Book book;
try (EntityManager em = entityManagerFactory.createEntityManager()) {
   book = em.find(Book.class, 5);
}
book.setTitle("Updated title...");
try (EntityManager em = entityManagerFactory.createEntityManager()) {
   em.getTransaction().begin();
   em.merge(book);
   em.getTransaction().commit();
}
```

The EntityManager is closed, so the book is no longer managed. We call it *detached*…

If we want to update a detached book, we use the merge operation!

KdG

Delete a *managed* book: use the remove method

```
try(EntityManager em = entityManagerFactory.createEntityManager()){
  em.getTransaction().begin();
  book = em.find(Book.class, 23);
  em.remove(book);
  em.getTransaction().commit();
}
```

We find the book first, and then remove it. After the commit, the book is removed from the database.
You can also use a delete query to delete in bulk a lot of students...

Examples of JPQL queries:https://docs.oracle.com/javaee/6/tutorial/doc/bnbtl.html

KdG

Delete a *detached* book: merge first!

```
Book book;
try (EntityManager em = entityManagerFactory.createEntityManager()) {
   book = em.find(Book.class, 5);
}
try (EntityManager em = entityManagerFactory.createEntityManager()) {
   em.getTransaction().begin();
   book = em.merge(book);
   em.remove(book);
   em.getTransaction().commit();
}
```

KdG

Exercise 5:

- Perform the different update and delete operations in a CommandlineRunner
  - Update the book with id 5: change the title…
  - Update all thrillers: set the title to uppercase
  - Delete book with id 7
  - Delete all thrillers

KdG

Agenda this week

What is ORM?
Spring configuration
The @Entity
CRUD operations with JPA
Implementing the repository
Relations with JPA

KdG

Let’s implement a BookRepository with JPA…

```
public interface BookRepository {
   List<Book> findAll();
   Book findById(int id);
   Book create(Book book);
   void update(Book book);
   void delete(Book book);
}
```

We start from this interface…

KdG

Let’s implement a BookRepository with JPA…

```
@Repository
public class JPABookRepository implements BookRepository{
   @PersistenceUnit
   private EntityManagerFactory entityManagerFactory;

   @Override
   public List<Book> findAll() {
       try (EntityManager em = entityManagerFactory.createEntityManager()) {
           List<Book> books = em.createQuery("select b from Book b").getResultList();
           return books;
       }
   }

   @Override
   public Book findById(int id) {
       try (EntityManager em = entityManagerFactory.createEntityManager()) {
           Book book = em.find(Book.class, id);
           return book;
       }
   }

   //…
```

Always the same code. Can Spring help us?

Inject an EntityManager!

```
@Repository
public class JPABookRepository implements BookRepository{
   @PersistenceContext
   private EntityManager em;


   @Override
   public List<Book> findAll() {
       List<Book> books = em.createQuery("select b from Book b").getResultList();
       return books;
   }

   @Override
   public Book findById(int id) {
       Book book = em.find(Book.class, id);
       return book;
   }

   //…
```

Spring creates the EntityManager for us!

KdG

And the create, update and delete methods?

```
@Repository
public class JpaBookRepository implements BookRepository {
   //…

   @Override
   public Book create(Book book) {
       em.getTransaction().begin();
      em.persist(book);
      em.getTransaction().commit();
      return book;
   }

   @Override
   public void update(Book book) {
      em.getTransaction().begin();
      em.merge(book);
      em.getTransaction().commit();
   }

   @Override
   public void delete(Book book) {
      //…

}
```

Always the same code. Can Spring help us?

You can use @Transactional

```
@Repository
public class JpaBookRepository implements BookRepository {
   //…

   @Override
   @Transactional
   public Book create(Book book) {
      em.persist(book);
      return book;
   }

   @Override
   @Transactional
   public void update(Book book) {
      em.merge(book);
   }

   @Override
   @Transactional
   public void delete(Book book) {
      //…

}
```

Use @Transactional: spring will add the transaction logic (begin and commit of the transaction)

Exercise 6:

- Try to implement the BookRepository using JPA!
  - Start from the code on gitlab: https://gitlab.com/kdg-ti/programming-3/exercises/jpabookrepository
  - Run it: it is a small console application to list, insert, update and delete books
  - Inspect the code:
    - it is 3-layered: presentation - service - repository
    - It uses H2 database
  - Implement the repository using JPA
  - Implement the service layer: delegate to the correct JPA methods
    - For the deleteBook you need 2 steps:
      - Find the book and delete it → **use @Transactional in your service layer!**
  - Implement the presentation layer

KdG

@Transaction in Service Layer

- Often the service layer methods will contain business logic
- This can involve taking a few queries to one or more repositories that should be managed by a transaction
- You can use the @Transactional annotation on top of these Service layer methods
  - Spring will ensure that these steps are done in one transaction.
  - All entities created and loaded are part of the same persistence context

KdG

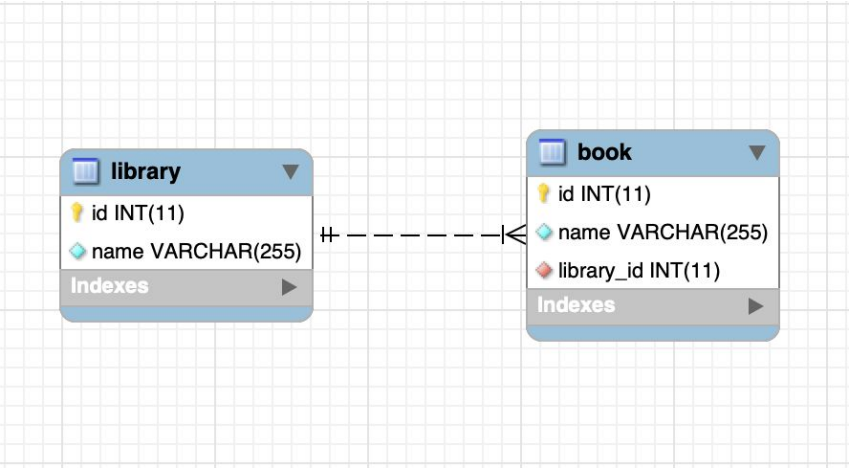
Agenda this week

What is ORM?
Spring configuration
The @Entity
CRUD operations with JPA
Implementing the repository
Relations with JPA

KdG

Advanced mappings

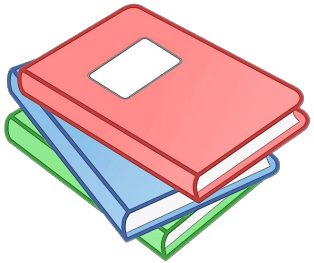
- What if we have
  - more than one table
  - Relationships between those tables?
- JPA has annotations for the different kinds of relationships:
  - One-to-One
  - One-to-Many and Many-to-One
  - Many-To-Many
- We will have a look at all of them...



KdG

Tutorial on mappings

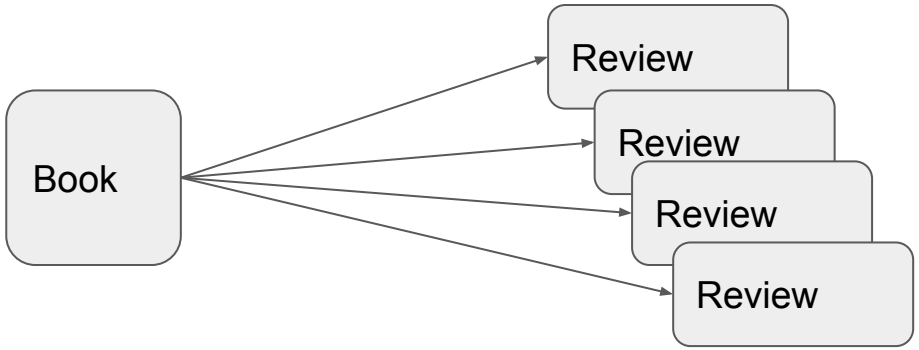
- https://stackabuse.com/a-guide-to-jpa-with-hibernate-relationship-mapping/
- Take the time to read through this tutorial!



KdG

Fetch types: Eager versus Lazy Loading

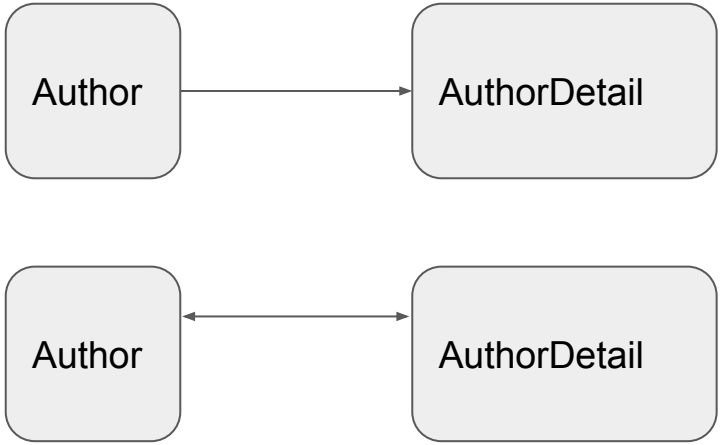
- When we retrieve data: should we retrieve all related entities…?
  - **Eager** will retrieve everything
  - **Lazy** will only retrieve on request



KdG

Uni-directional versus Bi-directional

- Should we be able to get the Author via the AuthorDetail or not?



KdG

@OneToOne - uni-directional

- Example: Author (firstName, lastName) - AuthorDetails (email)

```
@Entity
@Table(name="authors")
public class Author {
   @Id
   @GeneratedValue(strategy =
GenerationType.IDENTITY)
   private int id;

   private String firstName;
   private String lastName;

   @OneToOne(cascade = CascadeType.ALL)
   @JoinColumn(name="author_detail_id")
   private AuthorDetail authorDetail;
   //...getters and setters etc
}
```

We set the cascade type. If cascade is not specified, NO operations are cascaded.

@JoinColumn: only needed if you want to determine the name of the column with the FK in this table. author_detail_id is the default name (otherTablename_otherPKName)

@OneToOne - uni-directional

- Example: Author (firstName, lastName) - AuthorDetails (email)

```
@Entity
@Table(name="author details")
public class AuthorDetail {
   @Id
   @GeneratedValue(strategy =
GenerationType.IDENTITY)
   private int id;

   private String email;

   protected AuthorDetail() {
   }

   ///...
}
```

KdG

In the database

```
create table authors
(
   id                serial
       constraint authors_pkey
           primary key,
   first_name        varchar(255),
   last_name         varchar(255),
   author_detail_id integer
       constraint fk18v4mq362df6wh2xq5p60kvf7
           references author_details
);
```

```
create table author_details
(
   id    serial
       constraint author_details_pkey
           primary key,
   email varchar(255)
);
```

KdG

Using it in our Java code

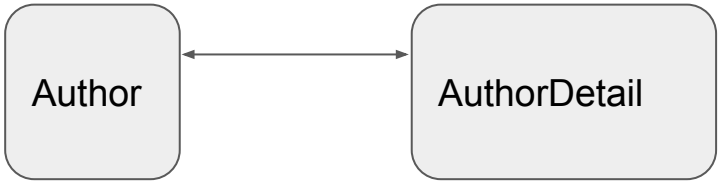
- We create the 2 entities, link them together and persist the main entity
- Because of the cascade, the sub-entity will also be saved!

```
try(EntityManager em = entityManagerFactory.createEntityManager()) {
 Author author = new Author("Herman", "Brusselmans");
 author.setAuthorDetail(new AuthorDetail("hman@brussel.be"));
 em.getTransaction().begin();
 em.persist(author);
 em.getTransaction().commit();
}
//...
```

If you remove an author, the authordetails will also be deleted!

KdG

OneToOne - bi-directional



- We only need to add the reference back from AuthorDetail...

```
@Entity
@Table(name="author_details")
public class AuthorDetail {
   @Id
   @GeneratedValue(strategy =
GenerationType.IDENTITY)
   private int id;

   private String email;

   @OneToOne(mappedBy = "authorDetail")
   private Author author;

//… getters and setters etc..
```

The mappedBy refers to the authorDetail field of the Author class.
mapped By indicates the inverse side. The other side (owning side) specifies the relation characteristics (JoinColumns…) and by default has the FK.
If you do NOT specify mappedBy you will have TWO unidirectional OneToOne relations with an FK in each table!

Using it in our Java code

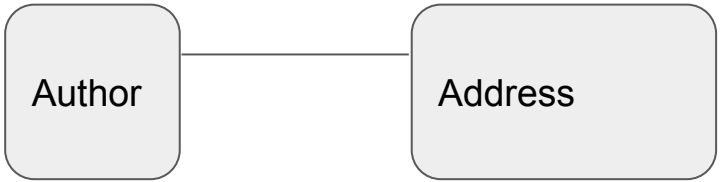
- We load AuthorDetails with id 1
- We can get the Author and show it

```
try (EntityManager em = entityManagerFactory.createEntityManager()) {
 AuthorDetail authorDetail = em.find(AuthorDetail.class, 1);
 System.out.println("Lazy or Eager?");
 System.out.println(authorDetail.getAuthor());
}
```

JPA uses Eager Loading: we don’t see any sql query running after the “Lazy or Eager” print...

KdG

OneToOne - Single table



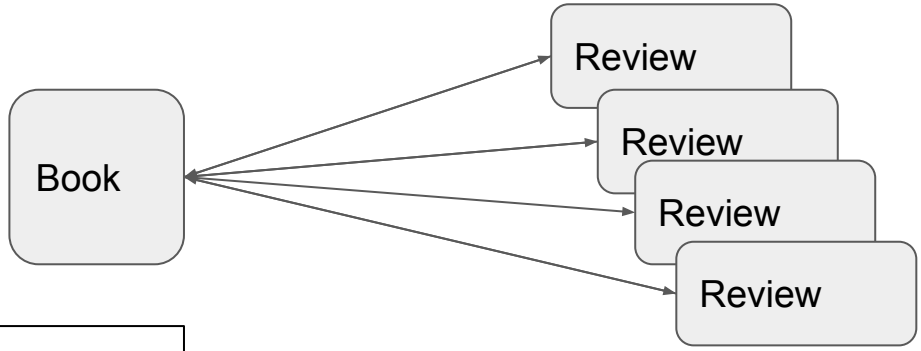
- All Address fields go into the Author Table

```
@Entity
@Table(name="authors")
public class Author {
 @Id
 @GeneratedValue(strategy =
GenerationType.IDENTITY)
 private int id;

 private String firstName;
 private String lastName;
 @Embedded
 private Address address;
  //…
}
```

```
@Embeddable
public class Address {
 private String street;
 private String number;
 private String zip;
 private String community;
//…
}
```

Specifying one of @Embeddable OR @Embedded is fine for embedding the fields of Address into the authors table.

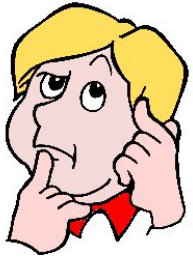
OneToMany - bi-directional



- In Review:

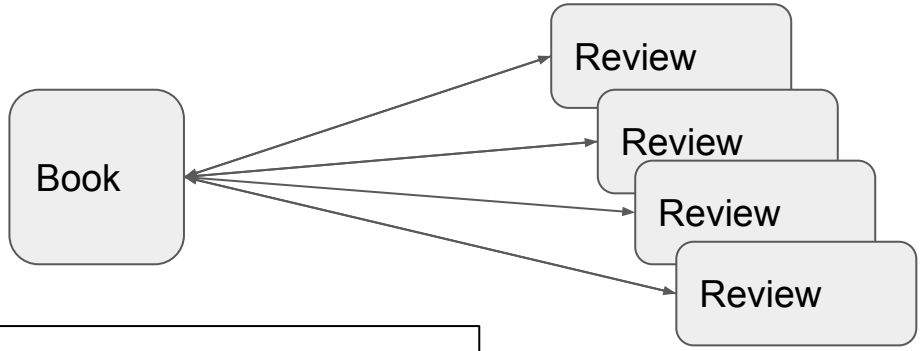
```
//...
@ManyToOne(cascade = {CascadeType. DETACH,
CascadeType.MERGE, CascadeType.PERSIST,
CascadeType.REFRESH})
private Book book;
//...
```

We cascade everything, **except** REMOVE!
Good idea?



KdG

OneToMany - bi-directional



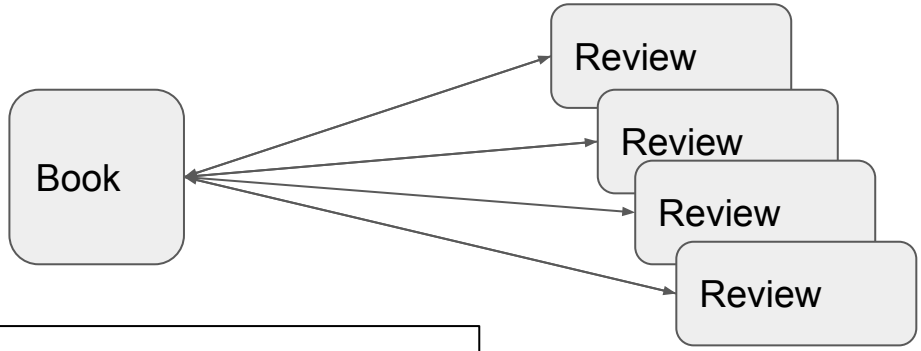
- In Book:

```
//...
@OneToMany(mappedBy = "book", cascade = CascadeType. ALL)
private List<Review> reviews = new ArrayList<>();

//…
```

```
Book contains a List of reviews...
```

KdG

OneToMany - bi-directional



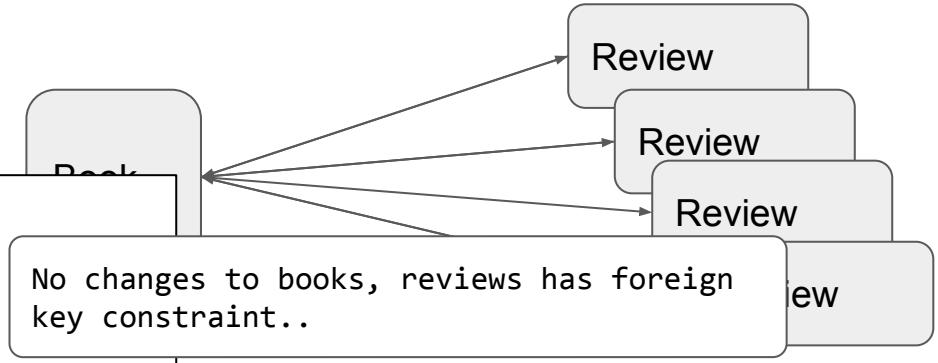
- In Book:

```
//...
public List<Review> getReviews() {   return reviews;}

public void setReviews(List<Review> reviews) {
   this.reviews = reviews;
   //TODO
}

public void addReview(Review review){
   reviews.add(review);
   review.setBook(this);
}
```

We add getter, setter and addreview method

In the database

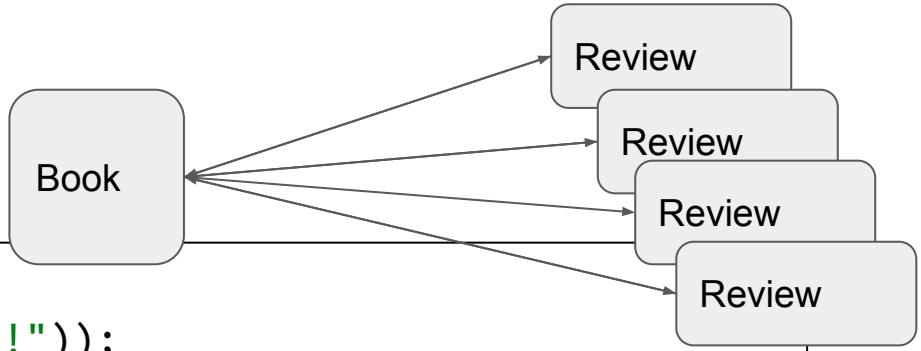
```
create table books
(
   id                 serial
       constraint books_pkey
           primary key,
   best_reading_time time,
   pub_date           date,
   genre              varchar(255
   pages              integer,
   title              varchar(50)
);
```




```
create table reviews
(
   id        serial
       constraint review_pkey
           primary key,
   contents varchar(255),
   book_id  integer
       constraint fk880c0rw9ffetwe2p1qb7x4icf
           references books
);
```

KdG

Using it in our Java code

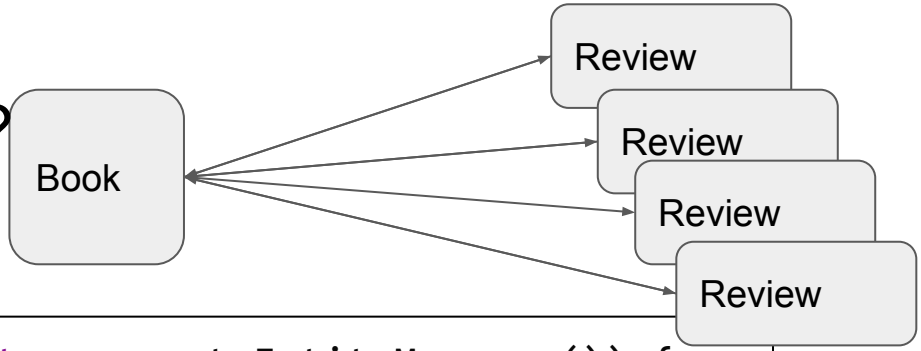



```java
Book book = Book.randomBook();
book.addReview( new Review("Great book!"));
book.addReview( new Review("Horrible book!"));

try(EntityManager em = entityManagerFactory.createEntityManager()) {
 em.getTransaction().begin();
 em.persist(book);
 em.getTransaction().commit();
}
```

Check the database: book and reviews are saved!

KdG

Are we Eager or Lazy loading?




```java
try(EntityManager em = entityManagerFactory.createEntityManager()) {
 book = em.find(Book.class, 1);
 System.out.println("Loading the reviews...");
 List<Review> reviews = book.getReviews();
 reviews.forEach(System.out::println);
}
```

```
Loading the reviews...
Hibernate: select reviews0_.book_id as book_id3_3_0_, reviews0_.id
Review{id=1, contents='Great book!'}
Review{id=2, contents='Horrible book!'}
```

Hibernate performs a select after the “Loading the reviews”: it uses lazy loading!

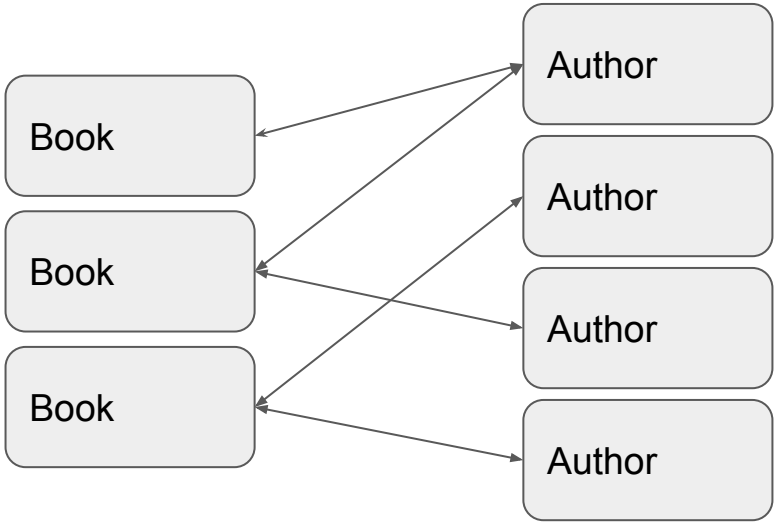
Default fetch types are LAZY for XxxToMany relations and EAGER for XxxToOne relations

You can specify the fetchtype yourself in the annotation:
@OneToOne(fetch=FetchType.LAZY)

KdG

@ManyToMany - bi-directional

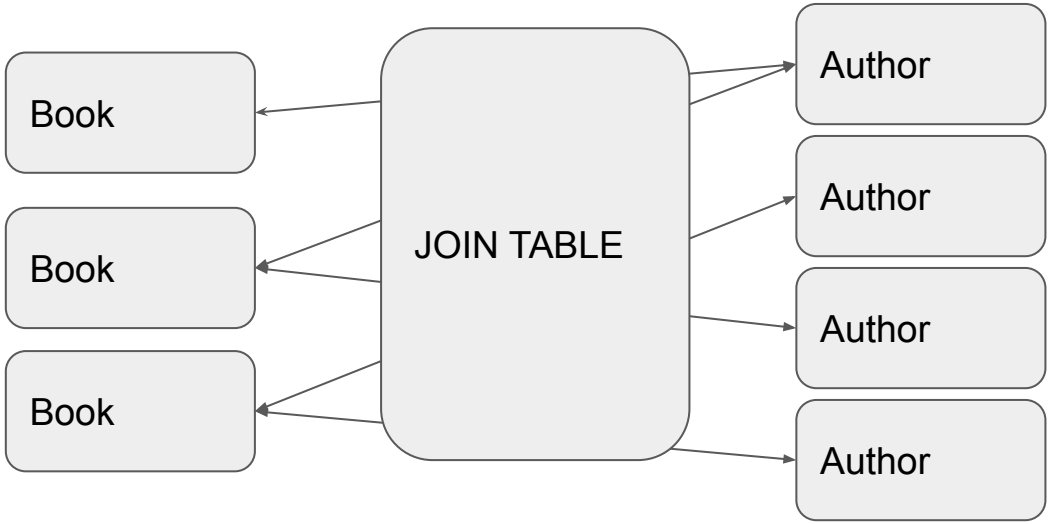
- A book can have more than one author, an author can write more than one book
- If we delete a Book, we don’t want to delete the Author
- If we delete an Author, we don’t want to delete the Book



KdG

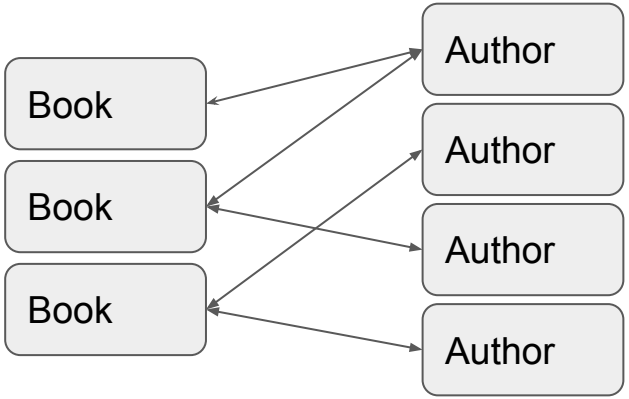
@ManyToMany - bi-directional

- We use a JOIN TABLE
  - Provides mapping between the two tables using 2 foreign keys



KdG

@ManyToMany - bi-directional



- In class Book:

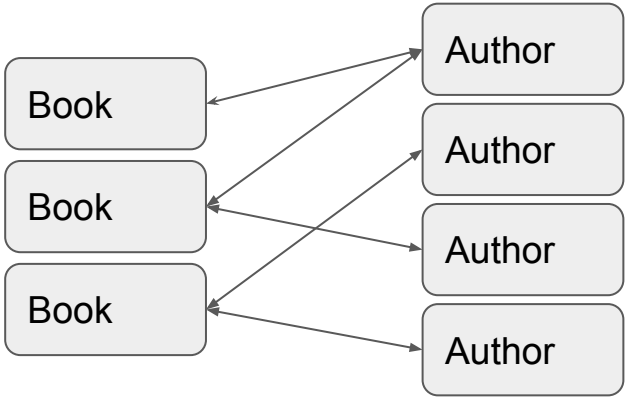
```
@ManyToMany(cascade = {CascadeType.DETACH, CascadeType.MERGE,
CascadeType.PERSIST, CascadeType.REFRESH})
@JoinTable(name="book_author",
       joinColumns = @JoinColumn(name = "book_id"),
       inverseJoinColumns = @JoinColumn(name="author_id"))
private List<Author> authors = new ArrayList<>();
```

```
And add getter and setter and addAuthor
method (addAuthor calls author.addBook)
...
```

```
If you omit @JoinTable…, defaults are
table books_authors(books_id,authors_id)
```

KdG

@ManyToMany - bi-directional



- In class Author:

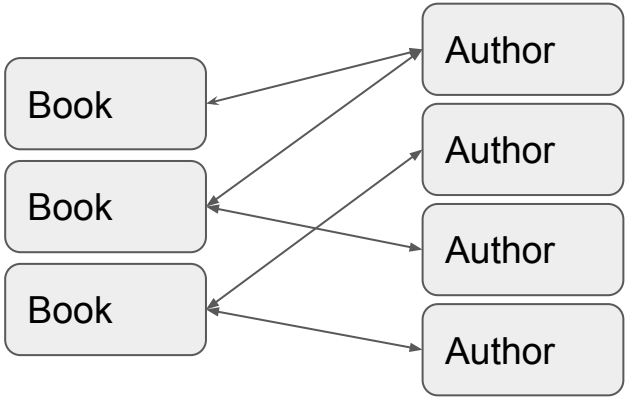
```
@ManyToMany(mappedBy = "authors", cascade =
{CascadeType.DETACH, CascadeType.MERGE, CascadeType.PERSIST,
CascadeType.REFRESH})
private List<Book> books = new ArrayList<>();
```

```
And add getter and setter and addBook
method...
```

KdG

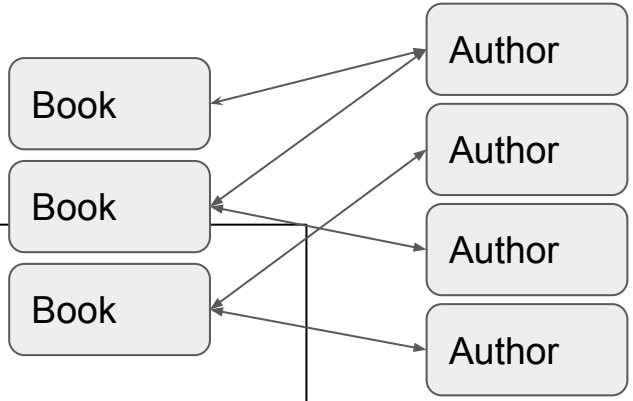
In the database

```
create table book_author
(
   book_id   integer not null
       constraint
fk91ierknt446aaqnjl4uxjyls3
           references books,
   author_id integer not null
       constraint
fkro54jqpth9cqm1899dnuu9lqg
           references authors
);
```



KdG

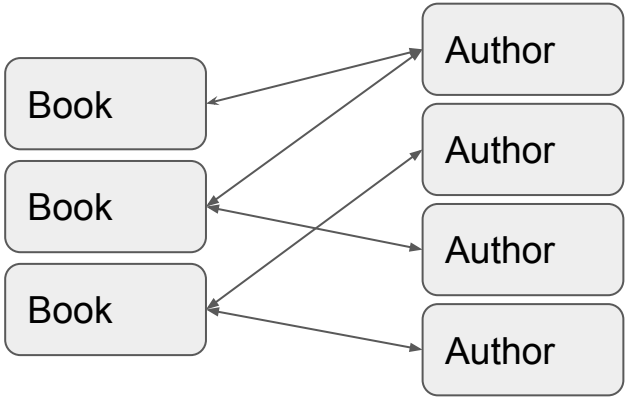
Using it in our Java code




```
Book book1 = Book.randomBook();
Book book2 = Book.randomBook();
Book book3 = Book.randomBook();
Author author1 = new Author("john","beck");
Author author2 = new Author("mario","vargas");
Author author3 = new Author("tine","verstrepen");
book1.addAuthor(author1);
book1.addAuthor(author2);
book2.addAuthor(author1);
book3.addAuthor(author3);
```

KdG

Using it in our Java code

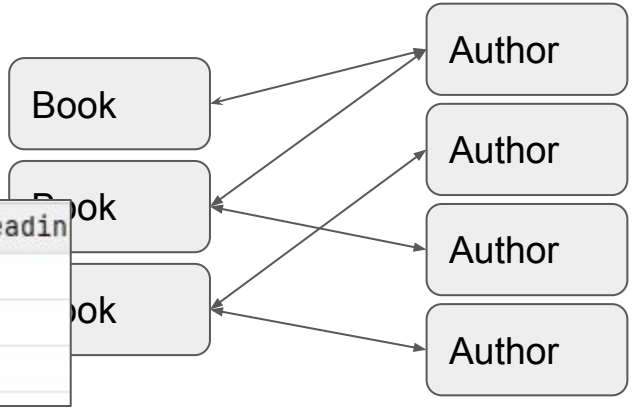



```
try(EntityManager em = entityManagerFactory.createEntityManager()) {
 em.getTransaction().begin();
 em.persist(book1);
 em.persist(book2);
 em.persist(book3);
 em.getTransaction().commit();
}
```

Check the database, do you find all the records?

KdG

Check the database...



id	title	pages	pub_date	genre	best_readin
1	title570	161	1990-02-10	FANTASY	19:46:00
2	title890	413	1995-05-11	THRILLER	16:06:00
3	title663	455	1975-04-03	FANTASY	06:40:00

id	first_name	last_name	author_detail_id
1	john	beck	<null>
2	mario	vargas	<null>
3	tine	verstrepen	<null>

book_id	author_id
1	1
1	2
2	1
3	3

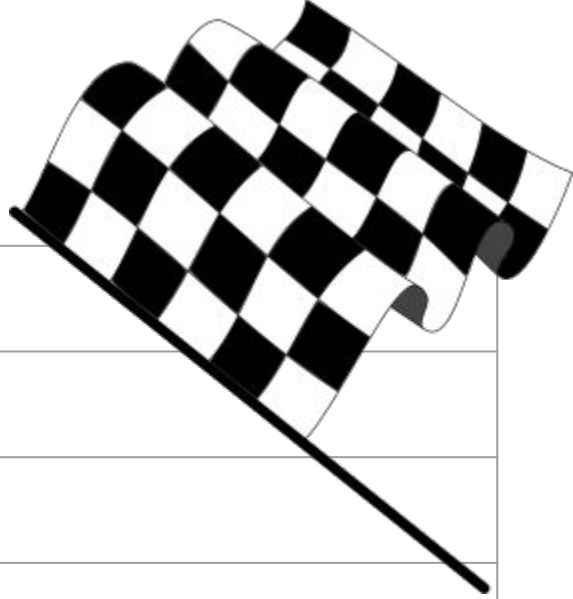
KdG

Exercise 7:

- Take the example of last week: https://gitlab.com/kdg-ti/programming-3/excercises/week9/solution/relationsdemo
- Now try to convert this demo to a JPA implementation

KdG

Agenda this week



What is ORM?
Spring configuration
The @Entity
CRUD operations with JPA
Implementing the repository
Relations with JPA

KdG

Project

- This week you will work out a third version of your repository layer: this time you will use JPA to implement the repository
- Add an extra profile to switch to this implementation
- Try to implement all the relationships between your entities
- Provide an application-dev.properties with H2 and application-prod.properties with PostgresQL configuration…

KdG