Developer diary:

18.06.2025:

* Starting to split it into enitities.
* Newly learned self join
* Get into crows notation

26.06.2025:

Using jakarta.validation-api, contributes the interfaces, with the Hibernate implementation , Hibernate- validator fully programmed annotations. Since this is a bean validation I made a DTO Object Login Request, which I use to run the validation on. Jakarta Expression Language depndency is needed for dynamic String interpolation and without it, it didn’t work in my programm.

xx.xx.2025:

Discovered that if you use static methods of a class. Then you can use it through the whole application like a Java Bean in the Spring Context.

05.07.2025:

Discovered extra line of defense with a private constructor in a utility class to prevent instantion.

Still need the Throw for clarity, but it is not strictly needed.

private SceneManager() {  
 throw new UnsupportedOperationException(" This is meant as a utility class only.");  
}

**06.07.2025:**

Sets up Hibernate with the important points in plain Java:

1. In die POM

* Jakarte Persistance
* Hibernate-core
* Postgres

2. In die Resources eine Java Persistence XML:

is the central piece of configuration. That makes it one of the most important files of your persistence layer.

Hier wird auch die Persistance Unit bennant.

A **persistence unit** defines a set of configuration and metadata for managing entities and connecting to the database. It is declared here in the persistence.xml.

<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"

version="2.2">

**<persistence-unit name="my-persistence-unit">**

<class>com.example.model.User</class>

<class>com.example.model.Order</class>

<properties>

<property name="javax.persistence.jdbc.url" value="jdbc:postgresql://localhost:5432/mydb"/>

<property name="javax.persistence.jdbc.user" value="postgres"/>

<property name="javax.persistence.jdbc.password" value="password"/>

<property name="javax.persistence.jdbc.driver" value="org.postgresql.Driver"/>

Ich muss sie dann verbinden wenn ich die Entity Manager Factory erstelle:

EntityManagerFactory emf = Persistence.createEntityManagerFactory("my-persistence-unit");

3. Die Entity Manager Factory erstellen.

Zum Beispiel vorläufig in der Main:

*//TIP To <b>Run</b> code, press <shortcut actionId="Run"/> or  
// click the <icon src="AllIcons.Actions.Execute"/> icon in the gutter.*public class Main {  
 public static void main(String[] args) {  
 try {  
 EntityManagerFactory emf = Persistence.*createEntityManagerFactory*("wifi-persistence-unit");  
 emf.close(); *// enough to trigger schema creation* } catch (Exception e) {  
 e.printStackTrace();  
 }  
  
 LoginView.*launch*(LoginView.class, args);  
 }  
}

Ohne dem erstellt er mir nicht die DB im PostgreSQL.

4. Die Klassen entsprechend annotieren.

**08.07.2025:**

Done self – referencing column in person entity.

CascadeType.PERSIST and CascadeType.MERGE. First means that the entity gets saved also the connected entitiy gets saved. Merge means the same with updated.

@ManyToOne  
@JoinColumn(name = "superior\_id")  
private Person superior;  
*//TODO try the subordinates with a set*@OneToMany(mappedBy = "superior", cascade = {CascadeType.*PERSIST*, CascadeType.*MERGE*})  
private List<Person> subordinates = new ArrayList<>();

Saving a Collection in the PostgresDatabase and the collection is also of enum elements:

Still person entity:

@ElementCollection(targetClass = Role.class)  
@Enumerated(EnumType.*STRING*)  
@CollectionTable(  
 name = "roles",  
 joinColumns = @JoinColumn(name = "person\_id")  
)  
@Column(name = "role")  
private Set<Role> roles;

@ElementCollection kennzeichnet mir das ganze als Collection vom Typ. In dem Fall role.

@Enumerated sagt speichere das als String.

@CollectionTable damit stelle ich die Normalform sicher. Hierin speichert er die Collection Elements. Ist eine eigene Tablle.

@Colum ist der Name der Tabelle

**14.07.2025:**

Habe mich bei dem Context für eine Singelton Klasse entschieden weil ich dann einfach klarere Dependencies in den einzelnen Klassen habe:

package org.fabianandiel.context;  
  
import org.fabianandiel.constants.Role;  
  
import java.util.Set;  
  
public class UserContext {  
 private static UserContext *instance*;  
 private String username;  
 private Set<Role> roles;  
  
 private UserContext() {  
 }  
  
 public static UserContext getInstance() {  
 if (*instance* == null) {  
 *instance* = new UserContext();  
 }  
 return *instance*;  
 }  
  
 public void initSession(String username, Set<Role> roles) {  
 this.username = username;  
 this.roles = roles;  
 }  
  
 public String getUsername() {  
 return username;  
 }  
  
 public Set<Role> getRoles() {  
 return roles;  
 }  
  
 public boolean hasRole(String role) {  
 return roles != null && roles.contains(role);  
 }  
  
 public void clearSession() {  
 this.username = null;  
 this.roles = null;  
 }  
  
}

No Eager when I do self reference in Hibernate:

Bescause then he goes into an infinte loop of loading data

Hibernate doesn't know that you only want to stop at **one level**. So:

1. It loads Person A
2. Sees EAGER → loads subordinates B and C
3. B and C are also Person → Hibernate sees EAGER on **their** subordinates fields
4. Hibernate now tries to load **B’s subordinates**, even though you never asked for it
5. That starts a **chain** (possibly infinite if the data structure is deep)

This is called **cascading eager loading** — and it happens **automatically** unless stopped.

…..

@Id  
@GeneratedValue(strategy = GenerationType.*UUID*)  
@Column(name="person\_id",nullable = false,updatable = false)  
private UUID id;  
@Column(name="firstname",nullable = false)  
private String firstname;  
@Column(name="lastname",nullable = false)  
private String lastname;  
@ManyToOne  
@JoinColumn(name = "address\_id")  
private Address address;  
@Column(name="telephone",nullable = false)  
private int telephone;  
@Column(name="email",nullable = false,unique = true)  
private String email;  
@Column(name="username",nullable = false,unique = true)  
private String username;  
@Column(name="password",nullable = false)  
private String password;  
@ManyToOne  
@JoinColumn(name = "superior\_id")  
private Person superior;  
  
**@OneToMany(mappedBy = "superior", cascade = {CascadeType.*PERSIST*, CascadeType.*MERGE*})  
private Set<Person> subordinates = new HashSet<>();  
  
@ElementCollection(targetClass = Role.class, fetch = FetchType.*EAGER*)  
@Enumerated(EnumType.*STRING*)  
@CollectionTable(  
 name = "roles",  
 joinColumns = @JoinColumn(name = "person\_id")  
)**  
@Column(name = "role")  
private Set<Role> roles;

…..

Why doesn’t this happen with the role table? Because it is a separate table and here he goes person->role

Great — let’s break it down clearly.

**❗What is a LazyInitializationException?**

It's a common Hibernate error:

org.hibernate.LazyInitializationException: could not initialize proxy - no Session

**✅ Meaning:**

You're trying to access a **lazy-loaded field** (like a Set<Person> or a List<Role>) **after the database session is already closed**.

Hibernate says:

"You asked me to delay loading this data (LAZY), but now I don’t have access to the database anymore — so I can't get it for you."

**🔍 When it happens:**

@OneToMany(mappedBy = "superior", fetch = FetchType.LAZY)

private Set<Person> subordinates;

Person person = personDAO.findById(id); // session is open here

System.out.println(person.getUsername()); // works

entityManager.close(); // session closed

System.out.println(person.getSubordinates()); // ❌ LazyInitializationException

**✅ The fix: JOIN FETCH**

**🛠️ What it means:**

A **JOIN FETCH query** tells Hibernate:

“Even if the field is lazy, **load it now while the session is still open**.”

**Example:**

SELECT DISTINCT p

FROM Person p

LEFT JOIN FETCH p.subordinates

WHERE p.id = :id

Now when you do:

Person person = personDAO.findByIdWithSubordinates(id);

System.out.println(person.getSubordinates()); // ✅ Works, because they were preloaded

**✅ Why it works**

| **Without JOIN FETCH** | **With JOIN FETCH** |
| --- | --- |
| Hibernate waits until you access the field | Hibernate loads it immediately in the same query |
| May fail if the session is closed | Works even after session closes |
| LazyInitializationException | ❌ |

**📦 Summary**

| **Term** | **What it means** |
| --- | --- |
| LazyInitializationException | You accessed a LAZY field after the session closed |
| JOIN FETCH | A query that tells Hibernate to load lazy fields immediately |
| Best Practice | Keep fields LAZY in the entity, and load them with JOIN FETCH only when needed |

Let me know if you want a utility method that safely fetches a person with roles and subordinates in one go!

**🔍 What is JOIN FETCH?**

JOIN FETCH is a special JPQL (Java Persistence Query Language) keyword that tells Hibernate:

"Join this related entity and load its data **immediately**, even if it’s marked as LAZY."

Trotzdem kann man sich nicht zu 100% drauf verlassen. Keine lazy geloadeden Fields in hash code and equals.