Software-Projekt WiSe 23/24

Aufgabenblatt 3

Abgabe bis: 10.12.2023 23:59 Uhr MEZ

Autoren: Karsten Hölscher, Marcel Steinbeck

In diesem Aufgabenblatt wollen wir das Thema Datenbanken bzw. *JPA*[1] (Jakarta Persistence API) näher beleuchten. Da ihr im Datenbankgrundlagenkurs bereits alle wesentlichen Grundlagen erworben habt (haben solltet), wird hier auf eine Einführung verzichtet, d. h. es wird vorausgesetzt, dass Konzepte wie z. B. relationale Datenbanken, Tabellen, Primär- und Fremdschlüssel, Transaktionen und SQL bekannt und verstanden sind. Stattdessen liegt der Fokus dieses Übungsblattes auf der Integration von Datenbanken (*H2*[2] im Speziellen) in Jakarta EE Anwendungen und der Abbildung von Java-Objekten auf relationale Datenbanken.

Aufgabe 1 Jakarta Persistence API

Wenn wir – wie hier in unserem universitären Kontext und auch ganz oft da draußen in der „echten“ Welt – Objekte unserer Programmiersprache in eine relationale Datenbank speichern (*persistieren*) möchten, dann stellen wir fest, dass das durchaus ein Problem darstellt. Es ist in der Literatur bekannt unter dem Namen *Object-relational impedance mismatch*[3]. Es gibt verschiedene Möglichkeiten, mit diesem Problem umzugehen. Offensichtlich könnte einfach eine *objektorientierte Datenbank* verwendet werden. Das ist aber nicht immer einfach so möglich und außerdem wäre dann dieses Aufgabenblatt sehr viel kürzer und langweiliger.

Wir beschäftigen uns daher mit einem alternativen Konzept, nämlich dem *ORM (Object-relational mapping)*. Dieses Konzept ermöglicht *objektrelationale Abbildungen*, d. h. die Objekte einer objektorientierten Programmiersprache (hier Java) auf relationale Datenbanken abzubilden.

ⓘ Eine Liste verschiedener ORM-Frameworks für diverse Programmiersprachen findet ihr unter: `https://en.wikipedia.org/wiki/List_of_object-relational_mapping_software`.

Das Jakarta Persistence API spezifiziert einen solchen ORM-Standard. JPA ist allerdings nur eine Spezifikation, die so erstmal nicht in einem Projekt benutzbar ist. Möchten wir JPA nutzen, müssen wir eigentlich eine konkrete Implementierung auswählen und in unser Projekt einbinden. In unserem Fall ist durch die Verwendung von WildFly diese Auswahl bereits erfolgt, denn WildFly bringt als Implementierung der Spezifikation (JPA ist Teil von Jakarta EE) das weit verbreitete und viel genutzte *Hibernate*[4] mit.

[1] `https://jakarta.eaworld.io/specifications/persistence/3.1/`
[2] `https://www.h2database.com`
[3] `https://de.wikipedia.org/wiki/Object-relational_impedance_mismatch`
[4] `https://hibernate.org`

Neben Hibernate als Implementierung für JPA gibt es noch *EclipseLink*[a], *Apache OpenJPA*[b], *DataNucleus*[c], *ObjectDB*[d] und weitere[e]. Ob die jeweilige Bibliothek bereits JPA 3.1 vollständig implementiert, muss allerdings im Einzelfall geprüft werden.

[a] `https://www.eclipse.org/eclipselink`
[b] `https://openjpa.apache.org`
[c] `https://www.datanucleus.org`
[d] `https://www.objectdb.com`
[e] `https://en.wikipedia.org/wiki/Jakarta_Persistence#JPA_2.0` ff.

JPA umfasst eine Vielzahl von Schnittstellen und Annotationen. Nicht ohne Grund gibt es ganze Bücher nur um das Thema JPA (Hibernate im Speziellen). Entsprechend können wir im Rahmen dieses Aufgabenblatts nur einen Bruchteil der bereitgestellten Funktionalitäten behandeln. Es empfiehlt sich daher weitere Lektüre heranzuziehen. Ihr findet z. B. in der Staats- und Universitätsbibliothek Bremen einige Bücher dazu. Gegenüber JPA 2.2 hat sich größtenteils nichts entscheidendes geändert, da der Schritt von JPA 2.2 zu JPA 3.0 lediglich eine Umbenennung der Pakete von `javax.persistence` zu `jakarta.persistence` beinhaltet. In JPA 3.1 sind dann wenige Dinge hinzugekommen – insofern ist also Literatur zu JPA 2 immer noch nützlich.

JPA abstrahiert zwar von der konkreten Implementierung, aber die Implementierungen bringen oft ihre spezifischen Erweiterungen und Ergänzungen mit sich. Wenn ihr auf die Spezifika von Hibernate verzichtet, kann die konkrete JPA-Implementierung leicht ausgewechselt werden: durch Änderungen der Projektkonfiguration und Anpassung der Konfigurationsdatei `persistence.xml` (siehe unten).

Durch die Verwendung von JPA ergeben sich im Wesentlichen zwei Vorteile:

- Die Menge an Glue-Code[5], der notwendig ist, um Java-Objekte in Datenbanken zu persistieren, wird erheblich reduziert (es sind keine speziellen `INSERT`-, `UPDATE`- und `DELETE`-Anweisungen notwendig).
- JPA abstrahiert von dem verwendeten Datenbankmanagementsystem (*DBMS*). Somit ist (sollte) ein Austausch des DBMS zu einem späteren Zeitpunkt problemlos möglich (sein). Dies wird insbesondere durch eine eigene (durch SQL inspirierte) Anfragesprache realisiert: *JPQL*[6] (*Java Persistence Query Language*). Wird kein Framework wie JPA (bzw. genauer: dessen Implementierung) eingesetzt, ist das DBMS nur dann leicht austauschbar, wenn im gesamten Projekt keine DBMS-Spezifika Verwendung finden.

Aufgabe 1.1 Datenquellen in Wildfly anlegen

Bevor ihr mit JPA loslegen könnt, braucht ihr zunächst eine Datenquelle (eine Verbindungskonfiguration für eine Datenbank) in WildFly, über die eure Daten persistiert werden können. Wir verwenden das von WildFly mitgebrachte Datenbankmanagementsystem H2. WildFly bringt alle notwendigen Treiber mit, um H2-Datenbanken in Jakarta EE Anwendungen anzubinden.

H2 kann in zwei verschiedenen Betriebsmodi verwendet werden: *embedded* und *server*. Während der *embedded*-Modus etwas einfacher aufzusetzen (und angeblich auch schneller) ist, erlaubt der

[5] `https://en.wikipedia.org/wiki/Glue_code`
[6] `https://jakarta.eaworld.io/specifications/persistence/3.1/jakarta-persistence-spec-3.1.html#a4665`

server-Modus die gleichzeitige Verwendung einer H2-Datenbank durch mehrere Prozesse (die Verbindungsart im *server*-Modus ist TCP). Dies ist insbesondere dann nützlich, wenn eine H2-Datenbank im Live-Betrieb (also während eine oder mehrere Anwendungen mit dieser interagieren) mit Hilfe externer Tools ausgewertet werden soll (z. B. um festzustellen, ob und welche Einträge getätigt wurden). Darüber hinaus bietet H2 noch einen sogenannten *mixed*-Modus an. Strenggenommen handelt es sich hierbei nicht um einen eigenen Modus, sondern stellt eine Kombination aus dem *embedded*- und *server*-Modus dar. Im *mixed*-Modus wird von der Anwendung, die eine Verbindung zu einer H2-Datenbank im *embedded*-Modus aufbaut, gleichzeitig (von der Anwendung selbst) ein H2-Server bereitgestellt, über den andere Anwendung eine Verbindung via TCP (vgl. *server*-Modus) aufbauen können, um so ebenfalls Zugriff auf die Datenbank zu erhalten.

Neue Datenquellen können in Wildfly entweder über das Webinterface (*Admin Console*) oder über das Kommandozeileninterface (*CLI*) angelegt werden. Der Einfachheit halber (und weil wir das schon in einem früheren Aufgabenblatt verwendet haben) wählen wir das CLI. Weitere Informationen zum Webinterface von Wildfly findet ihr unter: `https://docs.wildfly.org/29/Admin_Guide.html#web-management-interface`.

Denkt daran, dass zur Verwendung des CLI der WildFly gestartet sein muss und dass ihr im ersten Schritt `connect` eingebt, um euch mit dem Server zu verbinden.

Folgender Befehl fügt eine neue H2-Datenquelle (`--driver-name=h2`) mit dem Name `eksadat` und dem JNDI-Namen[7] `java:jboss/datasources/eksadat` hinzu.

Hinweise: Die abgebildete Tilde (∼) scheint von einigen PDF-Betrachtungsprogrammen nicht korrekt kopiert zu werden. Mitunter ist es notwendig, die Tilde manuell in das CLI einzugeben. Die Backslashes an den Zeilenenden bedeuten, dass trotz Zeilenschaltung die Eingabe noch nicht beendet ist und in der folgenden Zeile weitergeführt wird. Sie können direkt so eingegeben werden. Ohne die Backslashes am Zeilenende muss alles in eine Zeile kopiert werden.

```
1 data-source add --name=eksadat --jndi-name=java:jboss/datasources/eksadat \
2 --driver-name=h2 --connection-url=jdbc:h2:~/.eksadat/db;AUTO_SERVER=TRUE \
3 --user-name=eksadat --password=eksadat
```

Der Parameter `connection-url` setzt die Verbindungsdaten zur H2-Datenbank. Dessen Wert (`jdbc:h2:~/.eksadat/db;AUTO_SERVER=TRUE`) spezifiziert eine H2-Datenbank im *mixed*-Modus (`AUTO_SERVER=TRUE`[8]) mit dem Speicherort `~/.eksadat/db`. Neben absoluten Pfaden (hier ∼ für das Home-Verzeichnis einer Nutzer:in) für den Speicherort der Datenbank unterstützt H2 auch relative Pfade; beispielsweise: `./.eksadat/db`. Die Parameter `user-name` und `password` setzen, wenig überraschend, die Zugangsdaten; hier `eksadat` für Login und Passwort.

[7] `https://en.wikipedia.org/wiki/Java_Naming_and_Directory_Interface`
[8] `http://www.h2database.com/html/features.html#auto_mixed_mode`

Tücken mit relativen Pfaden

Relative Pfade zu H2-Datenbanken beziehen sich immer auf das Verzeichnis, von dem aus der H2-Server (*server*-Modus) bzw. die Anwendung (*embedded*- und *mixed*-Modus) gestartet wurde (*Arbeitsverzeichnis*). Das Problem mit relativen Pfaden ist, dass bei wechselnden Arbeitsverzeichnissen (z. B. weil der H2-Server nicht standardisiert gestartet wird) unerwartete Fehler auftreten können (und wahrscheinlich auch werden), wenn Anwendungen Verbindungen zu entsprechenden H2-Datenbanken aufbauen möchten. Die Verwendung von absoluten Pfaden ist also weniger fehleranfällig. Zudem bietet H2 mit ~ einen plattformübergreifenden Bezeichner (funktioniert auch unter Windows) für das Home-Verzeichnis einer Nutzer:in[a] an.

[a] `http://www.h2database.com/html/faq.html#database_files`

H2 Verbindungseinstellungen

Weitere Informationen zu den verschiedenen Verbindungseinstellungen (unter diesem Begriff fassen wir Verbindungs- und Zugangsdaten zusammen) von H2 findet ihr unter: `http://www.h2database.com/html/features.html#database_url`.

Wer legt die H2-Datenbank an?

Eine besondere Eigenschaft von H2 ist, dass Datenbanken nicht explizit angelegt werden müssen. Falls beim Verbindungsaufbau noch keine Datenbank mit den angegeben Verbindungsdaten existiert, wird ad-hoc eine neue angelegt. Weitere Informationen findet ihr unter: `http://www.h2database.com/html/tutorial.html#creating_new_databases`. **Hinweis:** *By default, if the database specified in the embedded URL does not yet exist [...]* meint nicht den *embedded*-Modus, d. h. auch im *server*-Modus wird seitens H2 bei Bedarf eine neue Datenbank angelegt.

Damit sind alle Vorbereitungen auf Seiten des Wildfly-Servers abgeschlossen. Ihr könnt nun das CLI mit dem Befehl `quit` beenden.

Aufgabe 1.2 Persistence.xml

Die Einbindung von Datenquellen in Jakarta EE Anwendungen ist denkbar einfach. Legt die Datei `src/main/resources/META-INF/persistence.xml` mit folgendem Inhalt an (die roten Pfeile markieren wie immer automatische Zeilenumbrüche):

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <persistence version="3.0"
3      xmlns="https://jakarta.ee/xml/ns/persistence"
4      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5      xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence
6      https://jakarta.ee/xml/ns/persistence/orm/orm_3_1.xsd">
7
8      <persistence-unit name="EksadatDB" transaction-type="JTA">
9        <jta-data-source>java:jboss/datasources/eksadat</jta-data-source>
10         <properties>
11             <property
12                    name="jakarta.persistence.schema-generation.database.action"
13                    value="drop-and-create"/>
14             <property name="hibernate.dialect"
                ↪ value="org.hibernate.dialect.H2Dialect"/>
15         </properties>
16     </persistence-unit>
17 </persistence>
```

Hervorzuheben sind hier `transaction-type` (Zeile 8), `jta-data-source` (Zeile 9) und die Properties `hibernate.dialect` und `jakarta.persistence.schema-generation.database.action` (Zeilen 11 – 14). In `jta-data-source` wird der JNDI-Name der zu verwendenden Datenquelle angegeben (vgl. `data-source add` in Aufgabe 1.1). Über `transaction-type` wird die Verwaltung der Transaktionen spezifiziert. Es gibt zwei mögliche Werte:

- **JTA:** JPA-Transaktionen werden vom Container verwaltet.
- **RESOURCE_LOCAL:** Die Verwaltung von JPA-Transaktionen obliegt der Anwendung.

Der Vorteil von `JTA` gegenüber `RESOURCE_LOCAL` ist, dass das Erstellen, Commit und Rollback von Transaktionen vollständig durch den Container übernommen wird. `RESOURCE_LOCAL` hingegen ist etwas flexibler (da die Applikation volle Kontrolle über alle Vorgänge hat) und wird insbesondere in JavaSE-Anwendungen genutzt (dort gibt es keinen Container, der etwas verwalten könnte). Es sei an dieser Stelle erwähnt, dass `JTA` nicht bedeutet, dass die Anwendung gar keine Kontrolle über das Transaktionsmanagement hat (vgl. Aufgabe 5.2).

Die Property `hibernate.dialect`[9] spezifiziert, welchen SQL-Dialekt die Datenquelle unterstützt. Üblicherweise kann Hibernate den Dialekt automatisch ermitteln[10], aber wir geben den Wert hier zur Sicherheit trotzdem an. Leider ist diese Konfiguration spezifisch für die verwendete JPA-Implementierung. Falls also eine andere JPA-Implementierung verwendet wird (z. B. EclipseLink), muss die Property entsprechend ersetzt werden.

Mit Hilfe der `jakarta.persistence.schema-generation.database.action` Property kann JPA (bzw. in unserem Fall Hibernate) angewiesen werden, bestimmte Operationen beim Start der Anwendung durchzuführen. Es stehen verschiedene Werte zur Auswahl[11]. Der Wert

[9] `https://docs.jboss.org/hibernate/orm/6.2/javadocs/org/hibernate/dialect/Dialect.html`
[10] `https://docs.jboss.org/hibernate/orm/6.2/userguide/html_single/Hibernate_User_Guide.html#configurations-general`
[11] Eine List aller Werte sowie eine umfassende Beschreibung findet ihr unter: `https://jakarta.eaworld.io/specifications/persistence/3.1/jakarta-persistence-spec-3.1.html#a12917`

`drop-and-create` gibt an, dass beim Start der Anwendung alle Tabellen gelöscht und neu erstellt werden sollen. Das ist insbesondere zu Beginn der Entwicklung hilfreich, da sich zu diesem Zeitpunkt die verwalteten Datenklassen häufig ändern.

Im nächsten Schritt konfigurieren wir die abzubildenden Datenklassen.

Aufgabe 2 JPA-Entitäten

Die von JPA zu verarbeitenden Datenklassen werden komfortabel über Annotationen konfiguriert. Die wichtigsten Annotation dürften wohl `Entity`[12] und `Embeddable`[13] sein. Auf den Unterschied beider Annotation wird hier nicht weiter eingegangen. Stattdessen sei die geneigte Leser:in auf die existierende Dokumentation: `https://en.wikibooks.org/wiki/Java_Persistence/Embeddables` und `https://www.objectdb.com/java/jpa/entity/types` verwiesen.

Mit `Entity` annotierte Klassen werden auch JPA-Entität genannt. Der Einfachheit halber fassen wir unter dem Begriff *JPA-Klassen* JPA-Entitäten und Klassen, die mit `Embeddable` annotiert sind, zusammen. JPA-Klassen werden von JPA automatisch auf eine (oder auch mehrere) Datenbanktabelle(n) abgebildet (genauer: die Attribute der annotierten Klassen werden auf Tabellenspalten abgebildet).

In einigen Tutorials werden in der Datei `persistence.xml` alle Klassen aufgelistet (XML-Tag `<class>`), die von JPA abgebildet werden sollen. Das ist für Hibernate nicht erforderlich, da Hibernate automatisch alle mit `Entity` und `Embeddable` annotierten Klassen eines Projektes ermittelt und registriert.

Wir passen jetzt die Klassen aus unserem Datenmodell entsprechend an.

Aufgabe 2.1 Entitäten im Datenmodell

In der Hoffnung, dass ihr euch bereits das nötige Wissen über den Unterschied zwischen `Entity` und `Embeddable` angeeignet habt, sollte die Verwendung von `Entity` für die Klassen `User` und `Student` nachvollziehbar sein. Annotiert also beide Klassen entsprechend.

IntelliJ zeigt daraufhin Fehler an. Diese Fehler ignorieren wir zunächst und gehen erst später darauf ein.

Wir haben jetzt war zwei Entities definiert, aber diese wissen im Entity-Modell noch nichts von den Eigenschaften, die sie im Java-Modell aus der Klasse `Person` erben. JPA bietet verschiedene Möglichkeiten zum Umgang mit Vererbung an[14].

Wir entscheiden uns hier für die Variante der `MappedSuperclass`[15] und versehen unsere Klasse `Person` mit der zugehörigen Annotation:

```
@MappedSuperclass
```

[12] `https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/entity`

[13] `https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/embeddable`

[14] `https://jakarta.eaworld.io/specifications/persistence/3.1/jakarta-persistence-spec-3.1.html#inheritance`

[15] `https://jakarta.eaworld.io/specifications/persistence/3.1/jakarta-persistence-spec-3.1.html#mapped-superclasses`

```
public abstract class Person {
   ...
```

Das ist hier sinnvoll, weil wir mit Personen allein ohnehin nichts anfangen können und sie also im Entity-Modell nicht benötigt werden.

Kommen wir zurück zu den Fehlermeldungen von IntelliJ für unsere beiden Entitäten. Für beide Klassen wird darüber gemeckert, dass sie keinen Primärschlüssel haben – und das stimmt ja auch. Es ist aber so, dass jede JPA-Entität (also jede mit `Entity` annotierte Klasse) einen *Primärschlüssel* benötigt[16].

Wir haben auf einem früheren Aufgabenblatt bereits über Kandidaten für einen Primärschlüssel nachgedacht und ihn zunächst auf die E-Mail-Adresse einer Person festgelegt. Das ist aber eigentlich wenig sinnvoll, denn eine E-Mail-Adresse könnte sich ändern und zudem werden E-Mail-Adressen im Kontext der Universität Bremen möglicherweise erneut vergeben, wenn die Vorbesitzer:in die Universität verlässt. Es ist daher sinnvoll, einen künstlichen Primärschlüssel einzuführen, der nur intern Verwendung findet und den Nutzer:innen verborgen bleibt.

Objektidentität und JPA

Die Verwendung eines internen Primärschlüssels – ohne Zusammenhang mit den eigentlichen Datenobjekten – erlaubt uns, diesen zu generieren bzw. generieren zu lassen. Die meisten DBMS unterstützen die Generierung von eindeutigen Primärschlüsseln, so dass sich dieses Konzept in der Praxis häufig findet.

Allerdings ist es in JPA ja so, dass Objekte in Java erzeugt werden und zunächst mal nichts von der Datenbank wissen – und umgekehrt genauso. Ein „frisch“ erzeugtes Objekt hat dann also noch gar keinen Primärschlüssel. Welche Probleme – insbesondere im Hinblick auf *equals()* und *hashCode()* sich daraus ergeben, könnt ihr im Artikel *Don't Let Hibernate Steal Your Identity* von James Brundege nachlesen. Der Artikel ist leider nur noch über die *Wayback Machine* von *archive.org* verfügbar: `https://web.archive.org/web/20180506060716/http://www.onjava.com/pub/a/onjava/2006/09/13/dont-let-hibernate-steal-your-identity.html` (die letzte erreichbare Fassung ist vom 06.05.2018).

Hinweis: In den Code-Beispielen des Artikels werden *Hibernate mapping files* (*hbms*) verwendet. In JavaEE solltet ihr darauf verzichten, da hbms zum einen spezifisch für Hibernate und zum anderen seit Einführung von Annotationen weitestgehend obsolet geworden sind. Einzig für `unsaved-value` muss in *reinem* JPA ein kleiner Umweg genommen werden.

Es sollte nicht unerwähnt bleiben, dass der Artikel von James Brundege nicht frei von Kritik ist; insbesondere seitens der Hibernate-Community. In dem Forumsbeitrag `https://forum.hibernate.org/viewtopic.php?f=1&t=967211` beispielsweise wird der Artikel mit den Worten *Incomplete, misleading, wrong conclusion.* bewertet. Selbiger Autor schreibt darüber hinaus:

> *Hibernate even _helps_ you with this when you require "one in-memory instance per database row", with it's freely extendable persistence and identity context.*

[16] `https://jakarta.eaworld.io/specifications/persistence/3.1/jakarta-persistence-spec-3.1.html#a132`

Dieser Aussage können wir nicht voll zustimmen, denn i) *Extended Persistence Contexts*[17] bergen (gerade im Kontext von Jakarta EE) ganz eigene Probleme (diese herauszufinden überlasse ich euch) und ii) eine Erklärung, was mit *identity context* gemeint ist, bleibt uns der Autor schuldig (wir konnten dazu nichts Eindeutiges in Erfahrung bringen). Ferner sind einige der genannten (Teil-)Lösungen nur mit einer direkten Abhängigkeit zu Hibernate umsetzbar (z. B. `IdentityMap`[18]). Insgesamt scheint die Diskussion in dem Forumsbeitrag emotional sehr aufgeladen zu sein (um es vorsichtig zu formulieren). James Brundege räumt schlussendlich zwar ein, dass:

> *I completely failed to mention the technique of extended sessions which eliminate the need to detach objects. Fair enough. I should have framed the context of the technique at the beginning of the article.*

Uns überzeugt der (seitens des Artikelkritikers als Allheilmittel angepriesene) Vorschlag: *Don't use detached objects if you don't want to*[19] jedoch nicht. Vielmehr haben wir den Eindruck, dass nicht nach Lösungen für existierende Probleme von Nutzer:innen von Hibernate gesucht wird, sondern sämtliche – unserer Meinung nach berechtigten – Anmerkungen mit: *sowas macht man mit Hibernate einfach nicht* abgeschmettert werden – alternative Meinungen sind natürlich zugelassen.

Wir könnten also wie im Artikel vorgeschlagen eine UUID[20] verwenden und direkt bei der Erzeugung eines Objektes setzen[21].

Das wäre dann aber wenig effizient, wie ihr im Artikel[22] von Vlad Mihalcea nachlesen könnt. Wir greifen daher den Vorschlag von Vlad auf und verwenden eine *TSID* (Time-Sorted Unique Identifier).

Da alle bzw. die meisten Entitäten einen derartigen Primärschlüssel benötigen, führen wir eine weitere abstrakte Oberklasse in unseren Datenmodell ein, die diesen Primärschlüssel entsprechend enthält.

Statt der im Artikel vorgeschlagenen Bibliothek verwenden wir deren Erweiterung von Vlad[23].

Fügt zunächst die Abhängigkeit in die `pom.xml` ein:

```
<hypersistence-tsid.version>2.1.1</hypersistence-tsid.version>
...
<dependency>
    <groupId>io.hypersistence</groupId>
    <artifactId>hypersistence-tsid</artifactId>
    <version>${hypersistence-tsid.version}</version>
</dependency>
...
```

Erzeugt dann im Unterpaket `model` die gemeinsame Basisklasse für alle Entitäten:

[17] https://docs.jboss.org/ejb3/app-server/tutorial/extended_pc/extended.html
[18] https://docs.jboss.org/hibernate/orm/6.2/javadocs/org/hibernate/internal/util/collections/IdentityMap.html
[19] https://forum.hibernate.org/viewtopic.php?p=2337195#p2337195
[20] https://www.uuidtools.com/what-is-uuid
[21] https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/UUID.html#randomUUID()
[22] https://vladmihalcea.com/uuid-database-primary-key/
[23] https://github.com/vladmihalcea/hypersistence-tsid

```
1  @Getter
2  @Setter
3  @NoArgsConstructor
4  @EqualsAndHashCode(onlyExplicitlyIncluded = true)
5  @MappedSuperclass
6  public abstract class DBEntity {
7
8      @EqualsAndHashCode.Include
9      @Id
10     private Long id = TSID.Factory.getTsid().toLong();
11
12 }
```

Wie üblich haben wir Getter- und Setter-Methoden für das einzige Attribut (Zeilen 1 und 2), einen Standardkonstruktor (Zeile 3) und wir verwenden jetzt die eineindeutige Id als einzigen Wert für *equals* und *hashCode* (Zeilen 4 und 8). Auch diese Klasse wird im Entity-Modell nicht isoliert benötigt, so dass wir uns auch hier für `MappedSuperclass` (Zeile 5) entscheiden.

Um die eineindeutige Id als Primärschlüssel für JPA zu kennzeichnen, annotieren wir sie mit `@Id`. Damit die Effizienz auf DBMS-Seite gewährleistet ist, speichern wir die Id allerdings direkt als `Long`. Alle Entities, die jetzt diese Klasse (direkt oder indirekt) erweitern, haben damit automatisch einen Primärschlüssel.

Das müssen wir jetzt also der Klasse `Person` noch mitteilen:

```
public abstract class Person extends DBEntity {
```

Da wir den Primärschlüssel in die Oberklasse verschoben haben und dort auch *equals* und *hashCode* für alle Unterklassen festlegen, müssen wir die entsprechenden Lombok-Annotationen noch aus der Klasse `Person` entfernen. Setzt das also um.

Aufgabe 2.2 Eigenschaften von Entities

Laut Abschnitt 2.1 im Standard[24] müssen JPA-Entitäten folgende Bedingungen erfüllen:

> ”
> The entity class must be annotated with the Entity annotation or denoted in the XML descriptor as an entity.
>
> The entity class must have a no-arg constructor. The entity class may have other constructors as well. The no-arg constructor must be public or protected.
>
> The entity class must be a top-level class. An enum or interface must not be designated as an entity.
>
> The entity class must not be final. No methods or persistent instance variables of the entity class may be final.
>
> If an entity instance is to be passed by value as a detached object (e.g., through a remote interface), the entity class must implement the Serializable interface.
>
> Entities support inheritance, polymorphic associations, and polymorphic queries.

[24] `https://jakarta.ee/specifications/persistence/3.1/jakarta-persistence-spec-3.1.html`

> Both abstract and concrete classes can be entities. Entities may extend non-entity classes as well as entity classes, and non-entity classes may extend entity classes.
>
> The persistent state of an entity is represented by instance variables, which may correspond to JavaBeans properties. An instance variable must be directly accessed only from within the methods of the entity by the entity instance itself. Instance variables must not be accessed by clients of the entity. The state of the entity is available to clients only through the entity's methods—i.e., accessor methods (getter/setter methods) or other business methods.
>
> ‘‘

Eine Entity benötigt also immer auch einen parameterlosen Konstruktor (dessen Fehlen IntelliJ auch für die Klasse `User` bemängelt). Behebt das Problem (Lombok hilft). Gemäß der obigen Auflistung darf unser Konstruktor mit Parametern erhalten bleiben.

Aufgabe 2.3 ElementCollections

In der Klasse `User` gibt es das Attribut `roles`, das als `Set` von Konstanten einer `enum` deklariert ist. Wir erinnern uns: Konstanten einer enum entsprechen in Java einfachen Zahlen. In so einem Fall – wenn wir eine Sammlung von einfachen Datentypen haben und diese überdies nicht als eigenständige Entities benötigt werden – bietet es sich an, die Sammlung als `ElementCollection` zu annotieren. Es wird dann eine eigene Tabelle für die Rollen (genauer die Zahlen, die den Rollen entsprechen) erzeugt, die dann auf die User-Tabelle verweist.

Annotiert also das Attribut wie folgt:

```
1 @ElementCollection
2 private Set<Role> roles = new HashSet<>();
```

Jetzt sind unsere Modellklassen zur Verwendung durch JPA vorbereitet.

Aufgabe 3 Die H2-Datenbank einsehen

Da wir unsere Web-Anwendung so konfiguriert haben, dass die Datenbank-Tabellen bei jedem Deployment gelöscht und neu erzeugt werden, können wir unsere Anwendung jetzt deployen und uns dann anschauen, welche Tabellenstruktur dann erzeugt wird.

Aufgabe 3.1 User verboten

Beim Deployment gibt es jetzt allerdings ein Problem. Die Anwendung lässt sich zwar erfolgreich deployen, aber in den Log-Ausgaben erkennen wir mehrere Exceptions. Gleich die erste (am weitesten oben im Log) enthält in der Meldung folgendes:

```
Syntax error in SQL statement "drop table if exists [*]User cascade ";
expected "identifier";
```

Hier erwartet H2 an der mit `*` markierten Stelle einen Bezeichner – nämlich den für die Tabelle, die gelöscht werden soll. Aus unserer Sicht steht dort ein Bezeichner für die Tabelle: *User*.

In H2 (und vielen anderen DBMSen) ist *User* aber kein gültiger Bezeichner, denn es ist ein Schlüsselwort[25].

Glücklicherweise müssen wir jetzt nicht unsere Java-Klasse umbenennen (auch wenn uns IntelliJ dabei durch das Feature „Refactoring“ die Arbeit abnehmen würde), sondern wir können JPA anweisen, statt des Klassennamens der Entity einen alternativen Namen zu verwenden. Das gelingt mit der Annotation `@Table`:

```
@Table(name = "EDUser")
```

Im Attribut `name` der Annotation können wir den gewünschten Tabellennamen angeben. Hier ist *EDUser* vorgeschlagen, um deutlich zu machen, dass es um die User der Anwendung **E**ksa**D**at geht.

Annotiert eure `User`-Klasse also und wählt einen alternativen Namen für die zu erstellende Tabelle. Deployt die Anwendung dann erneut. Es sollte keine Warnungen oder Fehler mehr geben.

Aufgabe 3.2 Datenquelle in IntelliJ

Wir können unsere Datenbank direkt in IntelliJ einsehen (und auch manipulieren). Dazu verwenden wir die `Database`-View (Datenbank-Sicht). Die könnt ihr typischerweise einblenden, indem ihr am rechten Rand das entsprechende Icon anklickt (siehe Abbildung 1).

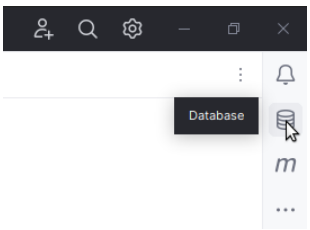

Abbildung 1: Icon für die Datenbank-Sicht in *IntelliJ*

Sollte das Icon bei euch nicht zu sehen sein (oder ihr findet es nicht), dann könnt ihr die Datenbank-Sicht auch durch Auswahl des Menüpunktes `View`→`Tool Windows`→`Database` anzeigen.

Es öffnet sich dann die Datenbank-Sicht. Klickt oben links auf das `+`-Symbol, um eine neue Datenquelle hinzuzufügen. Wählt dann `Data Source` und `H2` (siehe Abbildung 2).

Im sich öffnenden Fenster ändert ihr zunächst den *Connection type* von *Remote* auf *Embedded* (siehe Abbildung 3).

Jetzt könnt ihr einen eigenen Namen für die zu erstellende Datenquelle vergeben (z. B. *EksadatDB*). Für den `User`, das `Passwort` und die `URL` verwendet ihr die Angaben aus der WildFly-Datenquelle (siehe Aufgabe 1.1) und dann durch einen Klick auf `Test Connection` die Verbindung zur Datenbank überprüfen (siehe Abbildung 4).

Ein Klick auf `OK` schließt den Konfigurationsdialog und in der Datenbank-Sicht erscheint die soeben erstellte Datenquelle. Wenn ihr sie ausklappt und dann darunter `DB`, `Public` und `tables` ausklappt, dann seht ihr, dass JPA drei Tabellen angelegt hat: `EDUSER`, `STUDENT` und `USER_ROLES`. Ein Doppelklick auf den Tabellennamen öffnet einen neuen Tab-Reiter mit dem Inhalt der Tabelle (siehe Abbildung 5). Da die Tabellen allerdings noch leer sind, sehen wir hier nur die Spaltennamen. Es wird also Zeit, dass wir Daten dort eintragen...

[25] `http://h2database.com/html/advanced.html`, Abschnitt *Keywords / Reserved Words*

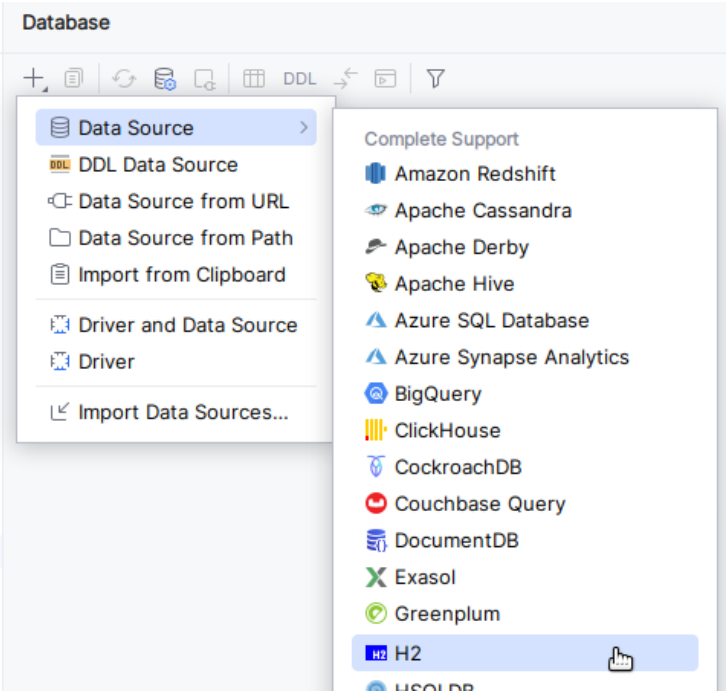

Abbildung 2: H2 Datenquelle hinzufügen

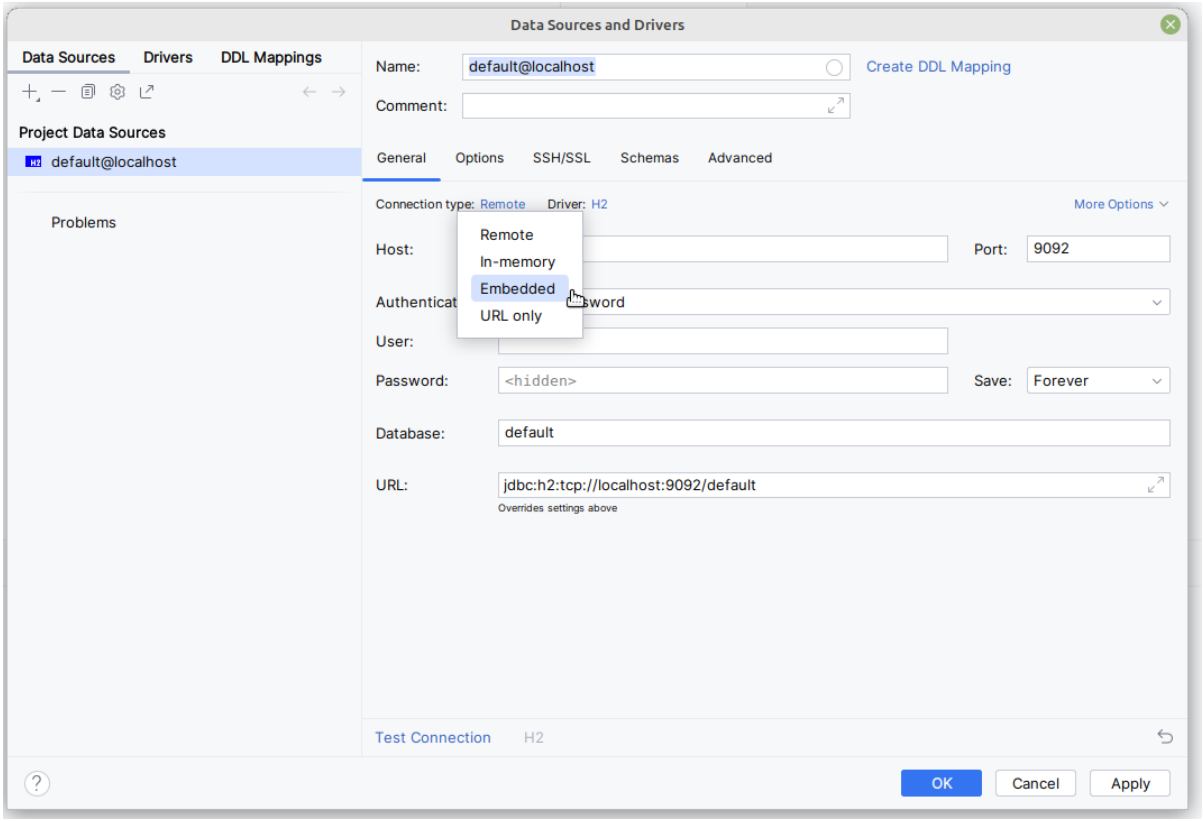

Abbildung 3: Connection type anpassen

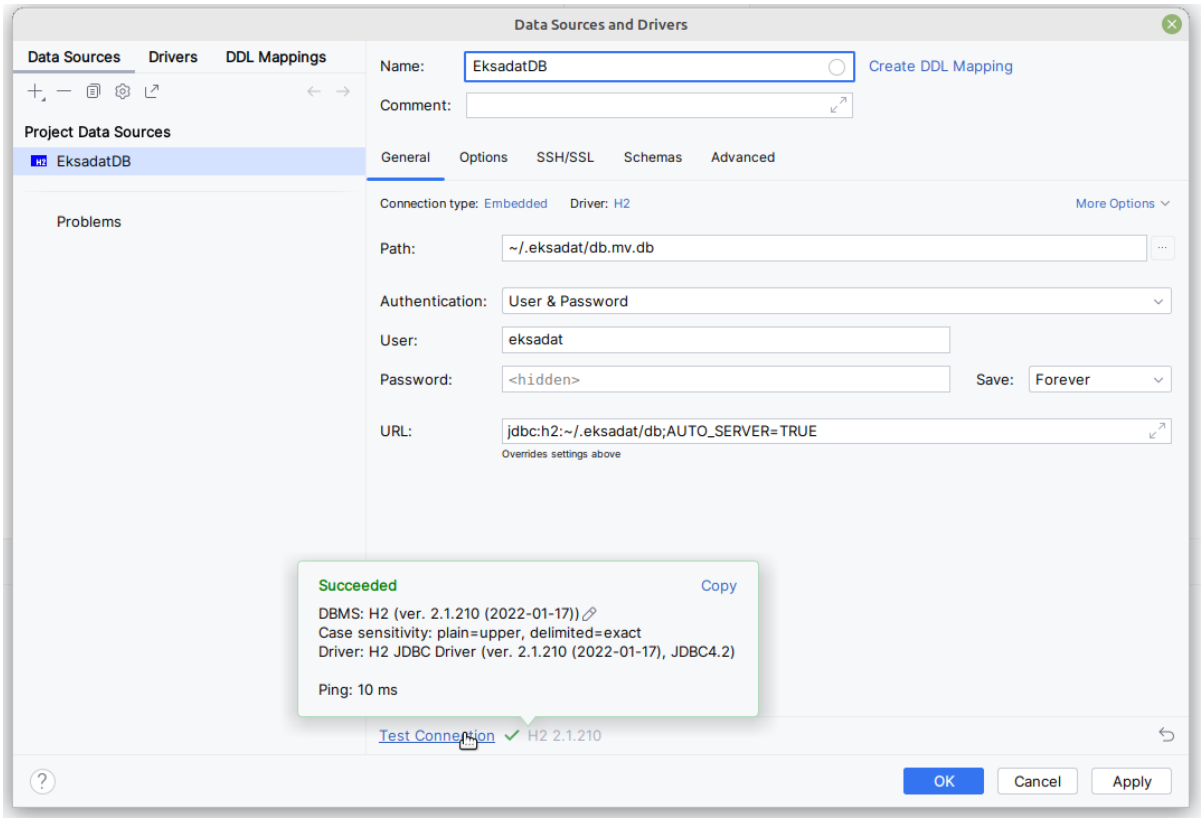

Abbildung 4: H2 Datenquelle konfigurieren

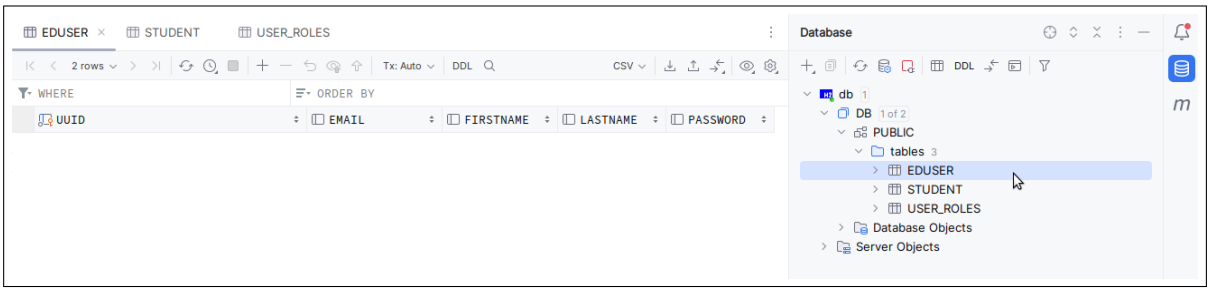

Abbildung 5: Tabelleninhalt anzeigen

Aufgabe 4 Der EntityManager

Der *Entity Manager* ist

> „
> Der Ressourcenmanager, der die aktive Sammlung der Entitätsobjekte verwaltet, die von der Anwendung verwendet werden. Der „EntityManager“ steuert die Datenbankinteraktion und die Metadaten für objektbezogene Zuordnungen. Eine EntityManager-Instanz stellt einen Persistenzkontext dar. [...] Wenn die Anwendung ihre Persistenz verwaltet, erhält sie den „EntityManager“ aus der „EntityManager-Factory“.[26]
> “

Der *Persistenzkontext*

> „Definiert die Gruppe aktiver Instanzen, die die Anwendung derzeit bearbeitet.“[26]

Der Entity Manager ist also das für unseren Persistenzmechanismus zentrale Objekt – denn darüber fügen wir Daten in die Datenbank ein und ändern oder löschen sie. Der Entity Manager kann in diesem Sinne als *Data Access Object (DAO)*[27] angesehen werden.

Ein DAO sollte mindestens die üblichen CRUD-Funktionalitäten anbieten und in der Tat enthält die Klasse `EntityManager` entsprechende Methoden[28].

In unserer vorgeschlagenen Architektur müssen die Repositories also jeweils den Entity Manager nutzen, um Zugriff auf die Datenbank zu erhalten.

Eine elegante Lösung wäre im Kontext von CDI natürlich, wenn die Repositories sich den Entity Manager einfach injizieren lassen könnten. Das ist in der Tat bereits unterstützt, nämlich durch die Annotation `@PersistenceContext`:

```
@PersistenceContext(name="EksadatDB")
private EntityManager entityManager;
```

Damit wird der zu unserer Datenquelle `EksadatDB` gehörende Entity Manager erzeugt und dem Attribut zugewiesen.

Jetzt muss dann allerdings an allen Stellen, die sich den Entity Manager auf diese Art injizieren lassen, der Name der Datenquelle angegeben werden. Daher greifen wir CDI jetzt unter die Arme und erzeugen einen *Producer*, der verwendet werden kann, um das bereits bekannte `@Inject` direkt zu nutzen.

Erzeugt dazu im Unterpaket `persistence` die Klasse `EntityManagerProducer` mit folgendem Inhalt:

[26] https://www.ibm.com/docs/de/was-liberty/base?topic=overview-java-persistence-api-jpa
[27] https://www.oracle.com/java/technologies/dataaccessobject.html
[28] https://jakarta.eaworld.io/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/EntityManager.html

```java
package de.unibremen.cs.swp.eksadat.persistence;

import jakarta.enterprise.inject.Produces;
import jakarta.persistence.EntityManager;
import jakarta.persistence.PersistenceContext;

public class EntityManagerProducer {

    @Produces
    @PersistenceContext(name="EksadatDB")
    private EntityManager entityManager;

}
```

Die Annotation `@Produces` kennzeichnet hier den *Producer* für den Typ `EntityManager`. Jetzt funktioniert eine Deklaration der Art

```java
@Inject
private EntityManager em;
```

Wenn CDI hier den EntityManager injizieren möchte, schaut es nach, wer ein Objekt dieses Typs bereitstellt. Dabei findet es dann den oben implementierten *Producer* und verwendet diesen. Es ist dabei beachtenswert, dass der *Producer* das Objekt allerdings gar nicht selbst erstellt, sondern es sich im Gegenteil selbst durch einen anderen Mechanimus injizieren lässt. Verrückt.

Manchmal kann die Erzeugung eines Objektes innerhalb eines *Producers* aufwändig sein und diverse Anweisungen erfordern. Daher könnt ihr auch eine Methode, die diese Anweisungen kapselt, durch die Annotation `@Produces` als *Producer* kennzeichnen und verwenden (lassen).

Neben `PersistenceContext` gibt es auch noch die Annotation `PersistenceUnit`. Der Unterschied ist zum einen, dass `PersistenceContext`[a] eine Instanz von `EntityManager` und `PersistenceUnit`[b] eine Instanz von `EntityManagerFactory` injiziert. Zum anderen wird der von `PersistenceContext` injizierte `EntityManager` durch den Container verwaltet (vgl. `transaction-type JTA` in `persistence.xml`). Bei `PersistenceUnit` obliegt die Verwaltung der `EntityManager`-Instanzen (z. B. die Erstellung[c] einer Instanz) der Applikation. Im Allgemeinen empfehlen wir, wann immer möglich, `PersistenceContext` zu verwenden. `PersistenceUnit` findet vornehmlich in JavaSE-Applikationen Anwendung – in aller Regel gibt es dort keinen Container, der etwas verwalten könnte.

[a] `https://javaee.github.io/javaee-spec/javadocs/javax/persistence/PersistenceContext.html`
[b] `https://javaee.github.io/javaee-spec/javadocs/javax/persistence/PersistenceUnit.html`
[c] `https://javaee.github.io/javaee-spec/javadocs/javax/persistence/EntityManagerFactory.html#createEntityManager`

Aufgabe 5 Repositories für JPA

Bisher haben wir zwei Repositories implementiert – je eins für `User` und für `Students`. Sie haben zwei sehr ähnliche Methoden, nämlich *save()* und *findAll()*. Die Wahrscheinlichkeit ist groß, dass alle oder zumindest die meisten Repositories diese Methoden benötigen. Und nicht nur diese, sondern noch einige andere, die sich aus dem CRUD-Konzept ergeben. Wir möchten natürlich den Code für diese Operationen nicht jedes Mal neu in den einzelnen Repositories erzeugen.

Der übliche Weg zur einer Lösung wäre eine gemeinsame, abstrakte Oberklasse für alle Repositories. Darin befinden sich dann die Methoden, die von allen Repositories benötigt werden (mindestens CRUD).

Aufgabe 5.1 Spring Data JPA

Glücklicherweise gibt es ein Framework (eigentlich mehrere, aber nur das hier verwendete arbeitet mit der aktuellen Jakarta EE Spezifikation erfolgreich zusammen), das uns derartige Repositories (und noch viel mehr) zur Verfügung stellt. Es handelt sich um *Spring Data JPA*[29].

Um das Framework zu nutzen, binden wir es wie üblich in unsere `pom.xml` ein:

```
...
<spring.data.version>3.1.3</spring.data.version>
...
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-jpa</artifactId>
    <version>${spring.data.version}</version>
</dependency>
...
```

Das Framework stellt uns Basis-Interfaces zur Verfügung, die wir erweitern können. Ändert daher die Klasse `StudentRepository` folgendermaßen:

```
1 package de.unibremen.cs.swp.eksadat.persistence;
2
3 import de.unibremen.cs.swp.eksadat.model.Student;
4 import org.springframework.data.jpa.repository.JpaRepository;
5
6 public interface StudentRepository extends JpaRepository<Student, Long> {
7 }
```

Wir „erweitern“ hier das gegebene `JpaRepository` aus dem Framework. Dafür müssen wir den Typ der Entitäten angeben, die verwaltet werden und den Typ des Primärschlüssels. Jetzt stehen uns in unserem `StudentRepository` alle Methoden aus dem erweiterten Interface[30] zur Verfügung. Da es sowohl eine *save()*- als auch eine *findAll()*-Methode zur Verfügung stellt, lässt sich unser Projekt trotz der Umstellung fehlerfrei übersetzen.

[29] `https://docs.spring.io/spring-data/jpa/reference/index.html`

[30] `https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html`

Im Hintergrund passiert folgendes: Zu dem von uns spezifizierten Interface erzeugt das Framework eine oder mehrere echte Klassen, die die gewünschten Fähigkeiten umsetzen und dann auch verwendet werden. Mit diesen Klassen haben wir aber nur dann zu tun, wenn Fehler (Exceptions) auftreten, denn dann sehen wir diese Klassen ggf. im Stacktrace.

Wir haben ja aber noch ein zweites Repository: `UserRepository`. Daraus wird jetzt analog ebenfalls ein Interface:

```
1  package de.unibremen.cs.swp.eksadat.persistence;
2
3  import de.unibremen.cs.swp.eksadat.model.User;
4  import org.springframework.data.jpa.repository.JpaRepository;
5
6  import java.util.Optional;
7
8  public interface UserRepository extends JpaRepository<User, Long> {
9
10     Optional<User> findByEmailAndPassword(final String email, final String
       ↪ password);
11
12 }
```

Die Besonderheit ist hier in Zeile 10 zu sehen. Dort haben wir die Signatur der Methode *findByEmailAndPassword* angegeben. Es gibt aber keine Implementierung des Methodenrumpfes. Woher weiß also das Repository, welche Datenbank-Abfrage zu dieser Methode gehört? Ganz einfach: das Framework ermittelt aus dem Namen der Methode und dem Rückgabetyp die zugehörige SQL-Abfrage. In diesem Fall ist das recht einfach, aber der Mechanismus dahinter kann auch deutlich komplizierte Anfragen generieren[31] und das funktioniert tatsächlich alles problemlos.

Trotz aller – bei Erstkontakt typischerweise überwältigenden – Magie, hat die automatische Abfragen-Generierung natürliche Grenzen. Daher können auch Abfragen explizit über die Annotation `@Query` direkt über der Methode vorgegeben werden. Wie das konkret funktioniert, findet ihr ebenfalls in der verlinkten Dokumentation[31]. Die verwendete Abfrage-Sprache für die `NamedQueries` ist übrigens `JPQL`[a] – insgesamt sehr nah an gewöhnlichem SQL.

[a] `https://jakarta.ee/specifications/persistence/3.1/jakarta-persistence-spec-3.1.html#a4665`

Grundsätzlich könnte unsere Web-Anwendung jetzt funktionieren...also deployen wir sie.

Aufgabe 5.2 Transaktionen mit CDI

Leider erhalten wir beim Deployment einen Fehler, der uns darauf aufmerksam macht, dass beim Speichern eines `Student`-Objektes ein Problem aufgetreten ist: `Transaction is required to perform this operation`.

[31] `https://docs.spring.io/spring-data/jpa/reference/jpa/query-methods.html`

Transaktionen werden (ihr erinnert euch hoffentlich an den Datenbankgrundlagenkurs) verwendet, um die Konsistenz des Datenbank-Inhaltes sicherzustellen. Beim Eintragen von Daten in die Datenbank könnte es ja sein, dass mehrere Tabellen betroffen sind, d. h. es wird mehr als ein `INSERT`-Statement ausgeführt. Sollte es dabei mittendrin z. B. zu einem Absturz der Anwendung kommen, ist es sehr wahrscheinlich, dass der Datenbestand inkonsistent ist.

Betrachten wir mal ganz konkret unseren Fall: Wenn ein `User`-Objekt persistiert werden soll, dann sind mindestens zwei `INSERT`-Statements nötig. Im ersten Statement werden die Attributwerte von Id, Vor- und Nachname sowie E-Mail-Adresse in die Tabelle `EDUSER` eingetragen. Im zweiten und ggf. weiteren Statements werden dann die Rollen mit Verweis auf den entsprechenden User-Eintrag in die Tabelle `USER_ROLES` eingetragen – eine Zeile pro Rolle des Users. Wenn jetzt die Anwendung nach der Ausführung des ersten Statements abstürzt, dann hätte nach einem Neustart die Tabelle `USER_ROLES` keine Einträge mit Verweis auf den User und dieser damit keine Rollen. Hier muss auch gar nicht der Extremfall eines Absturzes betrachtet werden – es genügt schon, wenn das erste Statement fehlschlägt, weil es z. B. schon einen User mit gleicher Id im Datenbestand gibt.

Aus diesem Grund (und noch anderen, z. B. nebenläufiger Zugriff) werden im Kontext von Datenbanken eigentlich immer Transaktionen eingesetzt. Vor dem Einfügen muss eine Transaktion erstellt bzw. gestartet werden und nach dem erfolgreichen Einfügen muss diese *committed* werden. Wenn während des Einfügens ein Problem auftritt, wird die Transaktion zurückgerollt (*rollback*), was insbesondere dazu führt, dass alle bisherigen Einfügungen innerhalb der aktuellen Transaktion rückgängig gemacht werden.

Wir müssen also eine Transaktion verwenden, um ein `Student`-Objekt zu persistieren. Auch wenn wir in der `persistence.xml` die Transaktionsverwaltung an den Container übergeben haben, müssen wir ihm dennoch helfen und anweisen, einen transaktionalen Kontext (*transactional context*) an konkreten Stellen herzustellen. Das gilt grundsätzlich für alle Bestandteile einer Jakarta EE Anwendung.

Seit JavaEE 7 bzw. JTA 1.2 gibt es einen einfachen Mechanismus für das deklarative Transaktionsmanagement mit CDI. Wenig überraschend handelt es sich hierbei um eine Annotation: `Transactional`[32]. Die Annotation kann sowohl auf Methoden- als auch auf Klassenebene angewendet werden und lässt sich (unter anderem) über die Eigenschaft `value` umfangreich konfigurieren[33]. Der Standardwert für `value` ist `Transactional.TxType.REQUIRED`:

> If called outside a transaction context, the interceptor must begin a new JTA transaction, the managed bean method execution must then continue inside this transaction context, and the transaction must be completed by the interceptor.
>
> If called inside a transaction context, the managed bean method execution must then continue inside this transaction context.

Achtung! *Spring Data JPA* bringt ebenfalls eine Annotation `Transactional` mit. Damit funktioniert es natürlich auch, aber es ist in der Service-Klasse sinnvoller, keine Abhängigkeit zu diesem Framework herzustellen. Stellt also sicher, dass ihr die Annotation aus dem Paket `jakarta.transaction` verwendet.

Eine mögliche Lösung unseres Problems wäre also die Annotation der Klasse `StudentService`

[32] `https://jakarta.eaworld.io/specifications/transactions/2.0/apidocs/jakarta/transaction/Transactional.html`

[33] `https://jakarta.eaworld.io/specifications/transactions/2.0/apidocs/jakarta/transaction/Transactional.TxType.html`

mit `@Transactional` (bzw. die Annotation der relevanten Methoden). Wir sprechen in diesem Fall von einer *Container Managed Transaction* (CMT).

Alternativ zu CMT kann das Transaktionsmanagement einzelner Klassen aber auch auf *BMT* (Bean Managed Transaction) umgestellt werden. Dazu muss eine Instanz der Klasse `UserTransaction` über die Annotation `Resource` injiziert werden (`UserTransaction` hat nichts mit unserer Klasse `User` zu tun). Mit Hilfe dieser Instanz können neue Transaktionen erstellt, abgeschlossen und bei Bedarf zurückgerollt werden. Folgendes Listing stellt dies beispielhaft dar:

```
1  public class UserDAOJPA implements UserDAO {
2
3      @Resource
4      private UserTransaction userTransaction;
5      ...
6      @Override
7      public void save(final @NonNull User user) throws IllegalStateException {
8          try {
9              userTransaction.begin();
10         } catch (final NotSupportedException | SystemException e) {
11             throw new IllegalStateException(e);
12         }
13         studentRepository.save(student);
14         try {
15             userTransaction.commit();
16         } catch (final Exception e) {
17             log.warn("Could not commit transaction", e);
18             try {
19                 userTransaction.rollback();
20             } catch (final Exception ex) {
21                 log.warn("Could not rollback transaction", ex);
22                 // Give up or whatever...
23             }
24             throw new RuntimeException(e.getMessage());
25         }
26     }
```

Obwohl CMTs in den meisten Fällen hinreichend flexibel (und im Allgemeinen wesentlich angenehmer zu nutzen) sind, gibt es auch Situationen, in denen BMTs die geeignetere Wahl sind – meistens dann, wenn ein hohes Maß an Kontrolle für die Erstellung, den Abschluss und das Zurückrollen von Transaktionen notwendig ist.

Wir empfehlen hier die Verwendung von CMTs für die Service-Klassen, d. h. ihr solltet eure Klassen jetzt mit `@Transactional` annotieren. Durch die Annotation auf Klassenebene sind dann alle nicht privaten Methoden automatisch in eine CMT eingebettet.

Wenn ihr jetzt ein erneutes Deployment versucht, werdet ihr feststellen, dass es immer noch nicht funktioniert. Objekte vom Typ `Student` werden problemlos persistiert (und sind auch in der Datenbanktabelle zu sehen) aber es tritt ein Fehler auf, wenn der `UserService` das `UserRepository` anweist, den ersten User zu speichern. Die Meldung in der zugehörigen Exception besagt, dass eine Transaktion benötigt wird. Aber wir haben doch unseren `UserService` auf Klassenebene mit `@Transactional` annotiert – wieso ist die Aktion dann nicht in eine CMT eingebettet?

Das liegt daran, dass CDI die verwalteten Klassen erst so spät wie möglich (*lazy*) initialisiert. Im

Zusammenspiel mit *Spring Data* Repositories ist das in diesem Fall **zu** spät. Wir müssen CDI also anweisen, das problematische Repository früher als üblich zu initialisieren. Dafür verwenden wir die Annotation `@Eager`[34]:

```
@Eager
public interface UserRepository extends JpaRepository<User, Long> {
```

Mit dieser Annotation lässt sich die Anwendung jetzt problemlos deployen und wir können danach auch die entsprechenden Einträge in den Datenbanktabellen sehen.

Warum muss denn das `StudentRepository` nicht *eager* initialisiert werden? Das liegt daran, dass das StudentRepository keine eigenen Abfrage-Methoden definiert. Es kann also direkt das Standard-Repository zum Einsatz kommen. Das `UserRepository` definiert aber eine zusätzliche Abfrage-Methode, daher muss das entsprechende Objekt durch CDI anders erzeugt werden.

Aufgabe 6 Lazy-Loading

Die Anwendung lässt sich zwar problemlos deployen und kommuniziert offensichtlich wie gewünscht mit der Datenbank, aber ein Login ist jetzt nicht mehr möglich. Wenn wir das versuchen (mit korrekten Anmeldedaten), dann wird eine Exception ausgelöst in deren Nachricht sich folgender Text findet: `could not initialize proxy - no Session`.

Puh – was ist jetzt wieder los?!?

Willkommen in der *Lazy-Loading-Hölle*. Per Default wird das `User`-Objekt *lazy* geladen, d. h. es werden nur die Werte der Basisattribute aus der Datenbank geladen und befüllt. Referenzen auf andere Objekte z. B. in Sammlungen (wie hier die Menge der Rollen) werden nicht automatisch mitgeladen. Das nervt hier zwar, ist aber grundsätzlich eine gute Sache.

Stellt euch folgendes Szenario vor: Ihr habt eine Klasse, die eine Universität repräsentiert. Jede Universität hat eine Liste von eingeschriebenen Student:innen. Nehmen wir jetzt an, ihr habt eine Anwendung, in der ihr in einem Auswahlfeld eine europäische Universität auswählen könnt, um näheres über sie zu erfahren. Um das Auswahlfeld zu befüllen, müssen dann offensichtlich alle europäischen Universitäten aus der Datenbank geladen werden. Ohne lazy-loading würden dann auch alle Student:innen pro Universität mit aus der Datenbank geladen (und für jede Student:in dann möglicherweise auch noch alle Kurse, die sie besucht haben mit Prüfungsergebnisen usw.). Vermutlich keine gute Idee im Hinblick auf Performanz und Speichernutzung...

JPA ist auf dieses Problem prinzipiell gut vorbereitet, denn innerhalb einer *Session* würden die entsprechenden Attribute bei Bedarf (d. h. hier Zugriff über eine Getter-Methode) automatisch aus der Datenbank nachgeladen. Das setzt allerdings voraus, dass es eine Session gibt.

Eine Session kann mehrere Transaktionen umfassen. Wenn wir z. B. eine Bank betreten, um diverse Überweisungen durchzuführen, dann wäre das Betreten und Verlassen der Bank die Session. Die einzelnen Überweisungen wären jeweils eine Transaktion.

[34] `https://docs.spring.io/spring-data/commons/docs/3.1.3/api/org/springframework/data/repository/cdi/Eager.html`

Um diese Automatik für unsere Anwendung zu verwenden, würden wir also einen Mechanismus benötigen, um eine Session bereitzustellen. Das würde in JPA über einen *Persistenzkontext* geregelt, der insbesondere an einen einzigen Entity Manager gekoppelt ist. Alle Aktionen der Session müssten sich also einen Entity Manager teilen. Gleichzeitig müssen natürlich verschiedene Sessions auch verschiedene Entity Manager nutzen, weil sonst u. a. das Transaktionskonzept ausgehebelt würde.

Jetzt ist ein guter Zeitpunkt (nach so vielen Seiten. . .), die möglichen Zustände einer JPA-Entität zu besprechen.

Eine JPA-Entität kann sich in einem der folgenden Zustände befinden[35]:

- A **new** entity instance has no persistent identity, and is not yet associated with a persistence context.
- A **managed** entity instance is an instance with a persistent identity that is currently associated with a persistence context.
- A **detached** entity instance is an instance with a persistent identity that is not (or no longer) associated with a persistence context.
- A **removed** entity instance is an instance with a persistent identity, associated with a persistence context, that will be removed from the database upon transaction commit.

Den Zusammenhang zwischen Entity-Manager und JPA-Entitäten können wir Abbildung 6 entnehmen.

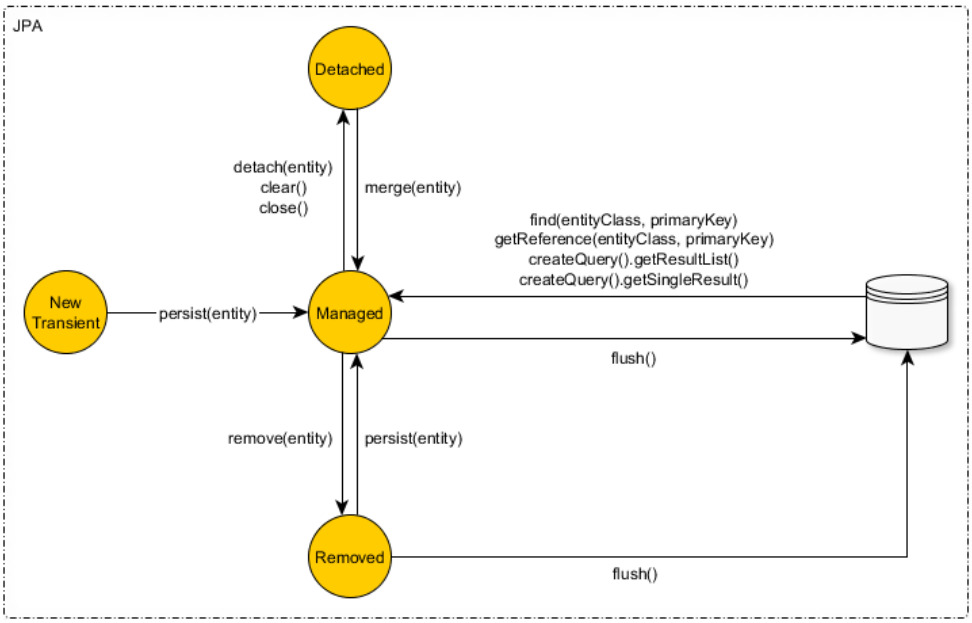

Abbildung 6: Zustände von JPA-Entitäten
[https://vladmihalcea.com/a-beginners-guide-to-jpa-hibernate-entity-state-transitions]

Was passiert also in unserer Anwendung? Wenn wir einen User aus der Datenbank holen, dann ist die Menge der Rollen nicht geladen. Für das eigentliche Login ist das kein Problem, denn es werden dazu nur die E-Mail-Adresse und das Passwort benötigt. Wenn aber im folgenden

[35] https://jakarta.ee/specifications/persistence/3.1/jakarta-persistence-spec-3.1.html#a1929

Facelet geprüft werden soll, welche Rolle der eingeloggte User hat, dann versucht JPA die Rollen nachzuladen. Das gelingt jetzt aber nicht, da das User-Objekt sich im Zustand *detached* befindet.

Wir könnten also versuchen, manuell eine übergeordnete Transaktion zu konfigurieren, die das Login inklusive Abfrage der User-Rollen beinhaltet. Das ist aber nicht so ganz einfach und würde den Rahmen der ohnehin schon sehr langen Aufgabenblätter sprengen (ihr sollt allerdings ein Bewusstsein für das Problem entwickeln).

Wir wählen hier der Einfachheit halber eine selten sinnvolle, aber zumindest funktionierende Option (auch schon, damit ihr das mal gesehen habt). Wir kennzeichnen das Rollen-Attribut als *eager* zu laden. Dann wird für jeden User immer auch gleich die Menge der Rollen mitgeladen und befüllt. Dazu konfigurieren wir die Annotation wie folgt:

```
@ElementCollection(fetch = FetchType.EAGER)
private Set<Role> roles = new HashSet<>();
```

Jetzt funktioniert die Anwendung wieder zuvor, allerdings mit dem entscheidenden Unterschied, dass die Daten jetzt tatsächlich in einer Datenbank persistiert werden und dort auch nach Beendigung der Anwendung „weiterleben“.

Info: Alternativen zu FetchType.EAGER

Wie so oft in der Welt der Software-Entwicklung gibt es verschiedene Wege, mit der Lazy-Loading-Problematik umzugehen. Wir betrachten hier Alternativen zu `FetchType.EAGER` in der Entität.

Um es noch einmal hervorzuheben: Im Fall der Rollen einer Nutzer:in ist das EAGER-Loading überhaupt kein Problem und auch sinnvoll. Es könnte aber andere Situationen geben, in denen ein EAGER-Loading nicht immer erwünscht ist.

Scheinbar sinnlose Zugriffe

Da – wie erwähnt – innerhalb einer Session die benötigten Werte automatisch nachgeladen werden, wenn sie benötigt werden, könnte ein Zugriff innerhalb der Transaktion der Service-Klasse erfolgen:

```
1  public Optional<User> findByEmailAndPassword(
2                                      final String email, final String password) {
3      final Optional<User> optionalUser =
4                              userRepository.findByEmailAndPassword(email, password);
5      if (optionalUser.isPresent()) {
6          final User user = optionalUser.get();
7          user.getRoles().size();
8      }
9      return optionalUser;
10 }
```

Im Rumpf dieser Methode haben wir noch eine Session, weil wir uns innerhalb einer Transaktion befinden. Wir prüfen in Zeile 5 ob die vorherige Abfrage ein `User`-Objekt geliefert hat. Wenn das so ist, holen wir uns in Zeile 6 die Referenz auf dieses `User`-Objekt. Dann holen wir uns in Zeile 7 die Rollen der Nutzer:in und rufen darauf die Methode *size()* auf – damit ist klar, dass die

Rollen aus der Datenbank nachgeladen werden müssen. Eine solche Aktion sollte dann natürlich durch einen Kommentar in unmittelbarer Nähe erläutert werden, denn die Anweisung an sich ist ja völlig sinnlos.

Wenn die Situation so ist, dass wir nicht immer die lazy-Attribute benötigen, dann sollten wir in der Service-Klasse mehrere Methoden haben. Die obige Methoden würden wir umbenennen in *findByEmailAndPasswordWithRoles()*, um anzuzeigen, dass die Rollen mitgeladen werden. Zusätzlich würden wir die ursprüngliche Methode ohne Rollen ebenfalls anbieten:

```
1 public Optional<User> findByEmailAndPassword(
2                                       final String email, final String password) {
3     return userRepository.findByEmailAndPassword(email, password);
4 }
```

Natürlich müsste dann an den Stellen im Quellcode, an denen die Rollen direkt benötigt werden, die Methode *findByEmailAndPasswordWithRoles()* aufgerufen werden.

Entity-Graphen

Eine andere Möglichkeit besteht in der Nutzung von Entity-Graphen. Ein Entity-Graph legt fest, welche Assoziationen (Fremdschüsselbeziehungen) mitgeladen werden. Wir können einen Entity-Graphen für die Entität selbst definieren oder die entsprechende Repository-Methode annotieren:

```
@EntityGraph(attributePaths = {"roles"})
Optional<User> findByEmailAndPassword(final String email, final String
 ↪ password);
```

Jetzt wird eine Abfrage generiert, die nicht nur das passende `User`-Objekt lädt, sondern auch das Attribut `roles`.

Wenn wir das für eine Methode mit einem speziellen Namen machen wollen, dann funktioniert der Methodenparser für das Repository nicht mehr. In diesem Fall müssen wir eine eigene Query definieren:

```
@EntityGraph(attributePaths = {"roles"})
@Query("select u from User u where u.email = ?1 and u.password = ?2")
Optional<User> findByEmailAndPasswordWithRoles(
                                       final String email, final String password);
```

Da über die Annotation `@Query` eine Abfrage in JPQL[36] für die Methode definiert wird, wird sie vom Methodenparser ignoriert. Würden wir die Annotation `@EntityGraph` weglassen, würde nur das `User`-Objekt ohne Rollen geladen. Mit der Annotation werden die Rollen direkt mitgeladen.

Es ist auch möglich, an der Entität selbst verschiedene Entity-Graphen zu definieren. Das könnte dann in unserem Fall so aussehen:

```
@NamedEntityGraph(
    name="User.Roles",
    attributeNodes = @NamedAttributeNode("roles")
)
public class User extends Person {
```

[36] https://jakarta.ee/specifications/persistence/3.1/jakarta-persistence-spec-3.1#a4665

Dann erhält der Entity-Graph einen Namen und die mit zu ladenden Assoziationen werden über das Attribut `attributeNodes` spezifiziert.

Sollen mehrere Assoziationen spezifiziert werden, dann werden sie durch Komma getrennt in geschweiften Klammern angegeben:

```
@NamedEntityGraph(
    name="Some.Thing",
    attributeNodes = {@NamedAttributeNode(...), @NamedAttributeNode(...)}
)
```

Die Verwendung in der Repository-Methode sieht dann so aus:

```
@EntityGraph(value = "User.Roles")
@Query("select u from User u where u.email = ?1 and u.password = ?2")
Optional<User> findByEmailAndPasswordWithRoles(
                                    final String email, final String password);
```

Es können übrigens verschiedene benannte Entity-Graphen definiert werden. Dazu werden sie von einer entsprechenden Annotation und in geschweifte Klammern eingeschlossen und durch Komma getrennt:

```
@NamedEntityGraphs(
    {
        @NamedEntityGraph(...),
        ...
        @NamedEntityGraph(...)
    }
)
```