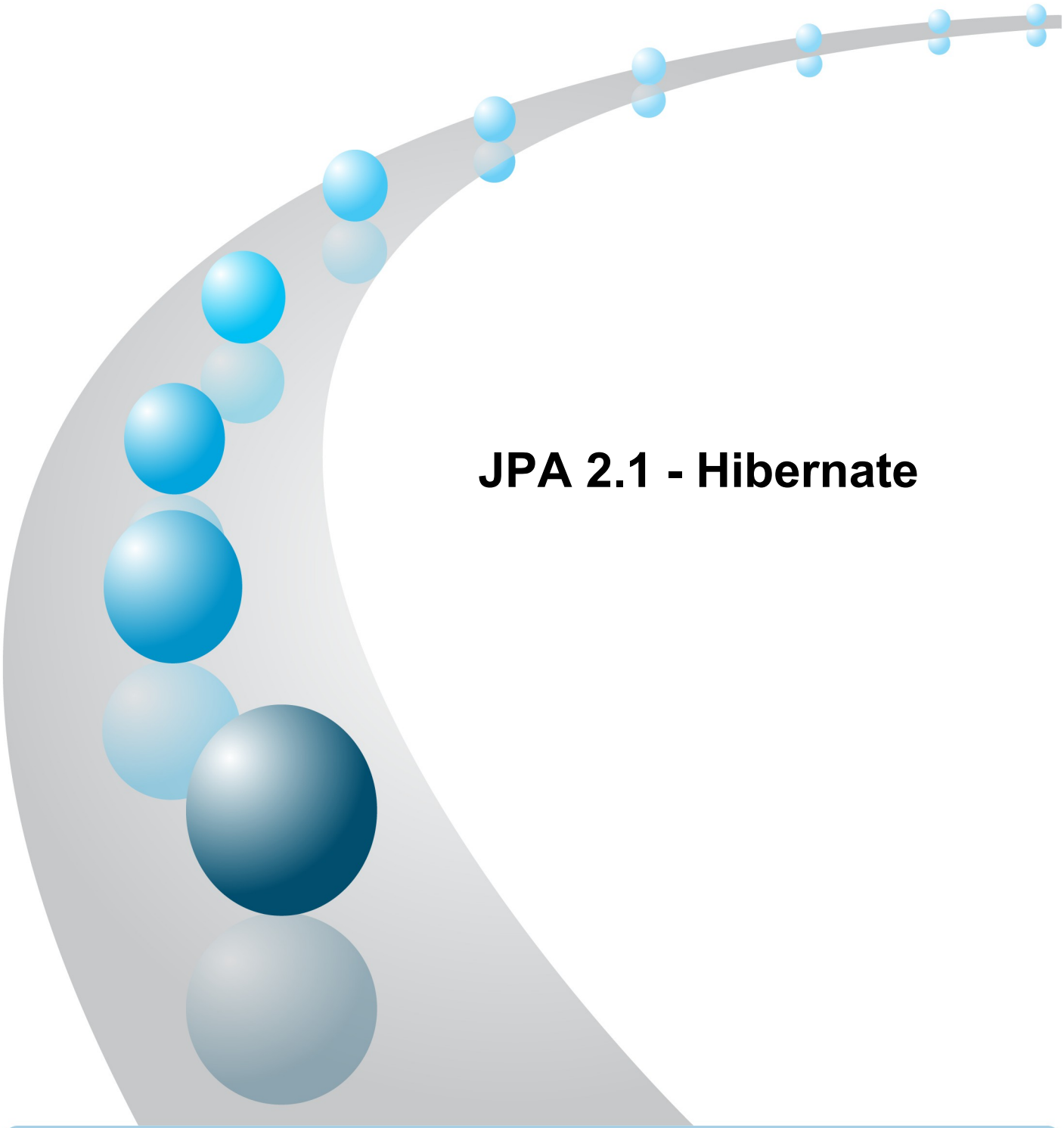
Noël Vaes

Java Trainer & Consultant



JPA 2.1 - Hibernate

Roode Roosstraat 5
3500 Hasselt
België

+32 474 38 23 94
noel@noelvaes.eu
www.noelvaes.eu

Noël Vaes
Roode Roosstraat 5
3500 Hasselt
België

Tel: +32 474 38 23 94

noel@noelvaes.eu
www.noelvaes.eu

02/01/2018

Inhoudsopgave

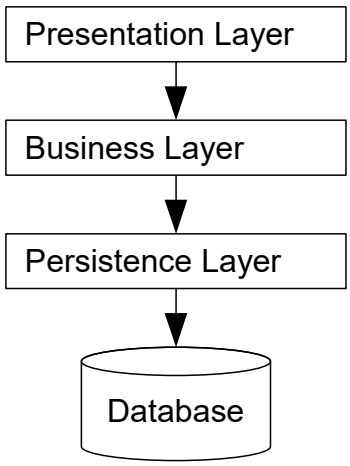
Hoofdstuk 1: Object-Relational Mapping

1.1 Inleiding

Software toepassingen maken doorgaans gebruik van gegevens die duurzaam bewaard moeten worden. Er zijn verschillende manieren om gegevens op te slaan en te gebruiken maar de meest gangbare wijze is het gebruik van een relationele databank. Deze technologie werd de laatste decennia uitvoerig uitgewerkt en heeft inmiddels geresulteerd in allerlei stabiele en betrouwbare databanksystemen.

Vanuit Java is het mogelijk deze relationele databanken te gebruiken met behulp van JDBC (*Java DataBase Connectivity*). Hiermee is het mogelijk SQL-commando's naar de databank te sturen en de resultaten ervan op te vragen. Doorgaans presenteren deze resultaten zich in tabelvorm; meer bepaald als `ResultSet`.

Om gegevens op te vragen en te manipuleren zijn heel wat databankcommando's noodzakelijk en bijgevolg ook heel wat code die gebruikmaakt van JDBC. Om te vermijden dat dergelijke JDBC-code verstrengeld wordt met de code die de zogenaamde *business logic* van de toepassing uitmaakt, gebruikt men vaak een gelaagde architectuur voor de toepassing waarbij iedere laag zijn eigen verantwoordelijkheid heeft:



1. ***Presentation layer***: Deze laag is verantwoordelijk voor de visuele presentatie van de toepassing en de interactie met de eindgebruiker.
2. ***Business layer***: Deze laag bevat de logica van de applicatie.
3. ***Persistence layer:*** Deze laag is verantwoordelijk voor het bewaren en opvragen van de gegevens in de toepassing.

In deze gelaagde architectuur is het enkel de *persistence layer* die communiceert met de databank en zorgt voor het opvragen en wegschrijven van de gegevens. Alle JDBC-code hoort hier dus thuis.

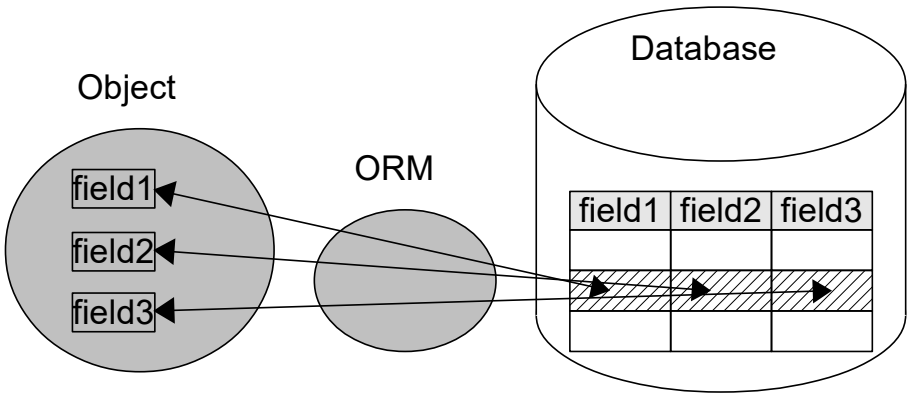
In de databank zijn de gegevens opgeslagen in de vorm van relationele tabellen. Deze gegevens moeten door de *persistence layer* op de een of andere manier aangeboden worden aan de *business layer* zodat ze gebruikt en gewijzigd kunnen worden waarna ze opnieuw worden aangepast in de databank. We sommen enkele mogelijke

voorstellingswijzen van de gegevens op:

1. **Tabellarisch:** De gegevens worden aangeboden in de vorm van een tabel. Dit komt overeen met de manier waarop ze zijn opgeslagen in de databank. Er kan hier bijvoorbeeld gebruikgemaakt worden van een mappenstructuur.
2. **XML:** De gegevens kunnen ook in de vorm van een XML DOM-structuur aangeboden worden zodat men tevens gebruik kan maken van alle mogelijkheden voor de manipulatie van XML-documenten (XPATH..).
3. **Java-objecten**: De gegevens worden voorgesteld in de vorm van objecten met *properties* die overeenkomen met de velden uit de databank.

Bij objectgeoriënteerde programmeertalen draait alles om objecten. De meeste voor de hand liggende keuze voor het voorstellen van gegevens is daarom ook in de vorm van een object. In deze cursus zullen we daarom deze techniek verder bekijken.

Er is echter een groot verschil tussen gegevens die vervat zijn in objecten en gegevens die opgeslagen zijn in relationele tabellen. Het gaat om een wezenlijk onderscheid tussen enerzijds het objectmodel dat gebruikt wordt in de code en het tabellarisch relationeel model dat gebruikt wordt in de databank. Om van het ene model naar het andere over te gaan is er een *mapping*-techniek nodig. Dit noemt men de **Object Relational Mapping**, afgekort ORM.



Deze ORM moet geïmplementeerd worden in de *persistence layer* zodat deze aan de ene kant objecten aan de *business layer* ter beschikking stelt en aan de andere kant deze gegevens uit relationele tabellen haalt.

Het is mogelijk dergelijke *persistence layer* met ORM-techniek zelf uit te werken. Dit kan gedaan worden door middel van ***Data Access Object (DAO)***. Voor kleine toepassingen kan dit een haalbare kaart zijn. Voor grote toepassingen met een complex objectmodel is het soms beter gebruik te maken van bestaande *frameworks*. Deze bevatten vaak veel meer mogelijkheden en zijn ook beter getest.

1.2 Enterprise JavaBeans 2.0

Als onderdeel van de *Java 2 Enterprise Editon 1.3 (J2EE)* werden *Entity Beans* uitgewerkt die onder andere een oplossing moesten bieden voor het ORM-probleem. Deze *Entity Beans* zijn componenten die door een *runtime container* beheerd worden en jammer genoeg erg vervlochten zijn met de container en de applicatieserver. De *Entity Beans* kunnen buiten

de omgeving van de container niet gebruikt worden. Bovendien zijn ze erg log en omslachtig in gebruik en configuratie.

De gecompliceerdheid van de *Entity Beans* deed programmeurs nostalgisch terugverlangen naar de *Plain Old Java Objects (POJO).* Intussen is de afkorting POJO een algemeen begrip geworden en staat symbool voor een nieuwe stijl van programmeren waarbij men teruggrijpt naar eenvoudige Java-objecten die geen afhankelijkheden hebben van andere systemen maar om het even waar gebruikt kunnen worden.

1.3 Hibernate – TopLink

Naast de officiële ORM-oplossing van SUN ontstonden er echter ook alternatieve producten die op een andere manier te werk gingen.
In de *open source* wereld ontstond zo het *framework* ***Hibernate*** dat ook een implementatie voorziet van een *persistence layer* met ORM. Momenteel is het een van de meest gebruikte technieken voor ORM.
Alle informatie over *Hibernate* is te vinden op de volgende website: www.hibernate.org.

In tegenstelling tot de EJB volgt *Hibernate* wel het POJO-principe; er wordt gebruikgemaakt van eenvoudige Java-objecten die persistent gemaakt worden in de databank.

Ook bij de commerciële producten werden er alternatieven ontwikkeld voor ORM. Zo is ***TopLink*** van *Oracle* een van de grote spelers op dat vlak.

1.4 Java Persistence API

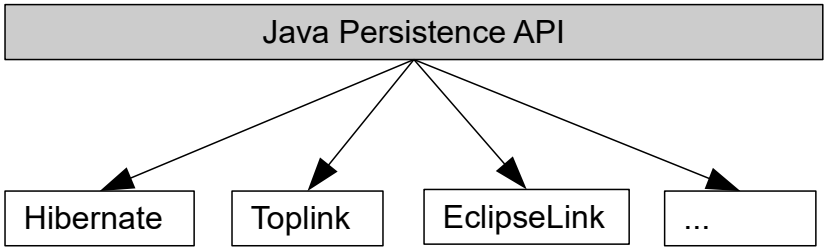
Het concept van *Hibernate* kon op veel bijval rekenen in de wereld van software-ontwikkelaars. Zelfs zij die gebruik maakten van *Enterprise JavaBeans* vervingen de gecompliceerde *Entity Beans vaak* door *Hibernate* om zo het beste van de twee werelden te combineren.

Bij het definiëren van de nieuwe specificatie EJB 3.0 werden de verzuchtingen en ervaringen van talloze Java-ontwikkelaars in rekening gebracht. Het oude model van de *Entity Beans* uit EJB 2.0 werd volledig achterwege gelaten en er werd resoluut gekozen voor het eenvoudigere POJO-model.

JEE is echter een specificatie en geen concrete implementatie (zoals *Hibernate*) en het doel van JEE is het programmeermodel voor *enterprise*-applicaties te standaardiseren. De herwerking van de ORM-oplossing in JEE versie 5 resulteerde in een geheel nieuwe specificatie: ***Java Persistence API (JPA)*** die een gestandaardiseerde oplossing moest bieden voor ORM. Deze specificatie is heel sterk geïnspireerd door *Hibernate*; een van de ontwikkelaars van *Hibernate* werd namelijk betrokken bij het opstellen van deze nieuwe specificatie.

De JPA biedt dus een programmeermodel voor een *persistence layer* die gebruikmaakt van POJO's. Zoals telkens bij dergelijke SUN-specificaties, wordt het aan de producenten overgelaten om een concrete implementatie hiervoor te voorzien. Dit maakt dat ontwikkelaars enerzijds slechts een API onder de knie hoeven te krijgen en dat ze anderzijds gebruik kunnen maken van alternatieve en uitwisselbare implementaties. Dit verhoogt de onafhankelijkheid van hun applicaties en stimuleert de concurrentie tussen de producenten van implementaties; hetgeen resulteert in hogere betrouwbaarheid, robuustheid en performantie.

Hibernate is bijgevolg een concrete implementatie van de JPA. Daarnaast zijn er nog andere implementaties beschikbaar. Zo is *Toplink* van *Oracle* ook een implementatie van JPA. Intussen zijn er nog andere implementaties zoals *EclipseLink, OpenJPA ...*



Bovendien dient iedere JEE applicatieserver vanaf versie 5 over een implementatie van de JPA te beschikken.

In JEE 5 werd de eerste versie JPA 1.0 geïntroduceerd. Daar JPA zowat een grootste gemene deler moest zijn van de onderliggende implementatie, bevatte deze niet alle mogelijkheden die wel beschikbaar waren in de originele *Hibernate* of *TopLink*.
In JEE 6 werd daarom JPA 2.0 voorzien waarin een aantal extra mogelijkheden werden toegevoegd en dit proces is verdergegaan in JEE 7 met JPA 2.1.

In deze cursus behandelen we JPA 2.1 en we zullen voor de concrete implementatie gebruikmaken van *Hibernate*, maar men kan op dezelfde manier gebruikmaken van een van de andere implementaties.

Hoofdstuk 2: Installatie & configuratie

2.1 Installatie van JPA met *Hibernate*

In deze cursus zullen we JPA illustreren met *Hibernate* als concrete implementatie. We kunnen hierbij de nodige bestanden afhalen van de website maar bij gebruik van *Maven* volstaat het een aantal afhankelijkheden toe te voegen aan de POM.

```
   <dependencies>
      <dependency>
         <groupId>org.hibernate</groupId>
         <artifactId>hibernate-entitymanager</artifactId>
         <version>5.2.6.Final</version>
      </dependency>
      <dependency>
         <groupId>mysql</groupId>
         <artifactId>mysql-connector-java</artifactId>
         <version>5.1.40</version>
      </dependency>
   </dependencies>
```

We voegen twee afhankelijkheden toe:

1. ***hibernate-entitymanager***: Deze afhankelijkheid bevat alles wat we nodig hebben om gebruik te maken van JPA met als concrete implementatie *Hibernate*. JPA 2.1 wordt ondersteund vanaf *Hibernate 4.3.0. Final.*
2. ***mysql-connector-java***: Voor de toegang tot de concrete databank moeten we een JDBC-*driver* hebben. In deze cursus maken we gebruik van een *MySQL* databank en zodoende voegen we deze afhankelijkheid toe. Bij gebruik van een andere databank dient de juiste *driver* hier toegevoegd te worden.

Opdracht 1: Een JPA-project maken

In deze opdracht maken we een nieuw project met behulp van *Maven.*

- Maak een nieuw *Maven* Java-project in je favoriete IDE.
- Voeg de volgende *dependencies* toe:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <groupId>eu.noelvaes.jpa</groupId>
   <artifactId>JPA</artifactId>
   <version>2.1.0</version>
   <properties>
      <maven.compiler.source>1.8</maven.compiler.source>
      <maven.compiler.target>1.8</maven.compiler.target>
      <project.build.sourceEncoding>
         UTF-8
      </project.build.sourceEncoding>
   </properties>
```

```
   <dependencies>
      <dependency>
         <groupId>org.hibernate</groupId>
         <artifactId>hibernate-entitymanager</artifactId>
         <version>5.2.6.Final</version>
      </dependency>
      <dependency>
         <groupId>mysql</groupId>
         <artifactId>mysql-connector-java</artifactId>
         <version>5.1.40</version>
      </dependency>
   </dependencies>
</project>
```

2.2 Configuratie van JPA

Bij *Object Relational Mapping* is er uiteraard wat configuratie nodig. Vooreerst moeten we ergens aangeven met welke databank we willen werken, wat de gebruikersnaam en het wachtwoord zijn en dergelijke. Deze configuratie gebeurt in een bestand met de naam ***persistence.xml*** dat zich in de map ***META-INF*** van het uiteindelijke JAR-bestand moet bevinden. We zullen later de details hiervan zien.

De structuur van het bestand is vastgelegd in een schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
       http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
   version="2.1">
</persistence>
```

Daarnaast is er ook configuratie nodig voor de mapping tussen de relationele tabellen en de Java-objecten. Dit kan op twee manieren gebeuren:

1. JPA kan gebruikmaken van **annotaties** die aan de programmeertaal werden toegevoegd sinds Java 5. We voorzien hierbij gewoon de nodige annotaties in de Java-klassen. Deze annotaties worden tijdens de uitvoering van het programma door JPA geïnterpreteerd.
2. JPA kan ook gebruikmaken van XML-bestanden die alle configuratiegegevens bevatten. Men noemt dit *deployment descriptors.* Het standaard-XML-bestand is ***orm.xml*** dat zich in de map ***META-INF*** van het uiteindelijke JAR-bestand moet bevinden. Het is evenwel ook mogelijk andere XML-bestanden te gebruiken die zich elders bevinden.

Annotaties werden in Java 5 geïntroduceerd om allerlei configuratiegegevens (*metadata*) rechtstreeks in de code te kunnen opnemen terwijl dat voorheen in een afzonderlijke *deployment descriptor* moest gebeuren. Dit maakt het de programmeur makkelijker omdat hij zo maar een bestand hoeft te onderhouden.

Nochtans heeft het gebruik van *deployment descriptors* ook een aantal voordelen die verloren zijn gegaan met de introductie van annotaties:

1. Met *deployment descriptors* heeft men een **perfecte scheiding** tussen de code en de configuratie. Indien de configuratie wijzigt, hoeft men enkel de XML-bestanden aan te

passen en blijft de broncode onaangeroerd. Dit is overigens een heilig principe als het gaat om het ontwikkelen van herbruikbare code. Deze mag namelijk niet aangepast worden.

2. JPA gaat uit van het POJO-model en dat impliceert dat heel veel **bestaande klassen** in aanmerking komen als *entity*-klasse, ook al zijn ze destijds niet met deze bedoeling ontwikkeld. Om ze tot *entity*-klassen te maken volstaan slechts enkele annotaties; maar vaak is het niet toegelaten of gewenst wijzigingen in bestaande code aan te brengen of in andere gevallen is de broncode zelfs niet beschikbaar. Ook hier brengen *deployment descriptors* soelaas. Deze kunnen gewoon toegevoegd worden aan bestaande klassen, zonder de broncode te wijzigen.

Bij JPA is om die reden het gebruik van een *deployment descriptor* mogelijk ter vervanging of ter aanvulling op de annotaties. In dit bestand kan men de gehele configuratie voorzien maar ook is het mogelijk deze te combineren met annotaties. De configuratie in de *deployment descriptor* vult hierbij de annotaties aan en kan zelfs de instellingen van de annotatie vervangen. De *deployment descriptor* heeft hierbij steeds voorrang op de annotatie. Dit is nodig om bestaande *entity*-klassen te kunnen voorzien van een nieuwe configuratie, zonder daarom de broncode hoeven te wijzigen.

De structuur van het bestand ***orm.xml*** is vastgelegd in een schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings version="2.1"
   xmlns="http://xmlns.jcp.org/xml/ns/persistence/orm"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence/orm
      http://xmlns.jcp.org/xml/ns/persistence/orm_2_1.xsd">

</entity-mappings>
```

In deze cursus zullen we hoofdzakelijk gebruikmaken van annotaties omdat dit voor de Java-programmeur de snelste manier is om vertrokken te raken met JPA. Voor een volledige beschrijving van de *deployment descriptor* verwijzen we naar de uitgebreide JPA-documentatie.

Hoofdstuk 3: Mijn eerste *entity*-klasse

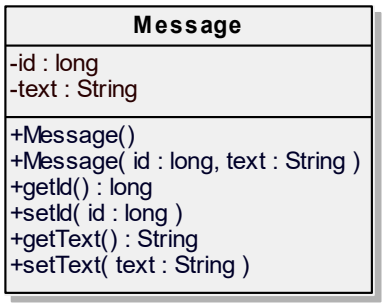
3.1 Daar komt hij weer ...

Tijd voor wat praktische oefeningen nu. Als programmeur[1] moeten de vingers nu toch al wat jeuken.

In dit hoofdstuk voorzien we een eerste concrete kennismaking met *JPA.* En ja, daar komt dus weer: de *Hello World*. Deze keer gaan we boodschap-objecten maken waarin we bijvoorbeeld de tekst "Hello World" steken en deze objecten gaan we wegschrijven naar een tabel in de databank.

3.2 De *entity*-klasse

Bij *object relational mapping* hebben we enerzijds Java-objecten die anderzijds worden bewaard in een relationele databank. We beginnen ditmaal met het Java-object. Een boodschap kunnen we voorstellen door de volgende klasse:



Zo'n object neemt de vorm aan van een klassieke *JavaBean*. Dat wil zeggen dat de klasse een aantal velden heeft die ingesteld en opgevraagd kunnen worden met *getters* en *setters* volgens de *JavaBeans*-specificatie.

Merk op dat we deze boodschap voorzien hebben van een uniek identificatienummer. Dit is nodig voor opslag in de databank (*primary key*).

De code:

```
package messages;
import javax.persistence.*;

@Entity
public class Message {
   private long id;
   private String text;

   public Message() {
   }

   public Message(long id, String text) {
```

1 Oh ja, deze cursus is geschreven met het oog op Java programmeurs die onmiddellijk *hands-on*-ervaring willen opdoen en al doende *Hibernate* willen verkennen. Geen ellenlange theoretische beschouwingen met eindeloos gecompliceerde voorbeelden en allerlei exotische *features* dus. Hiervoor wend je je best tot de plaatselijke boekhandel (mag ook online natuurlijk).

```
      this.id = id;
      this.text = text;
   }

   @Id
   public long getId() {
      return id;
   }

   public void setId(long id) {
      this.id = id;
   }

   public String getText() {
      return text;
   }

   public void setText(String text) {
      this.text = text;
   }
}
```

We gebruiken de annotatie `@Entity` om aan te geven dat het om een *entity*-klasse gaat. Verder geven we met de annotatie `@Id` aan dat dit veld de *primary key* is.

3.3 De *Persistence Unit*

Objecten van de klasse `Message` gaan we dadelijk wegschrijven naar een databank. Uiteraard moeten we eerst aangeven welke databank. Dit doen we in de configuratie van een *persistence unit*. Een *persistence unit* is een geheel van *entity*-klassen die samen gebruikmaken van dezelfde databaseconfiguratie. Deze configuratie wordt beschreven in een XML-bestand met de naam ***persistence.xml*** dat toegevoegd wordt aan het finale JAR-bestand waarin de *enitity*-klassen worden ondergebracht.

Dit bestand ziet er in ons geval als volgt uit:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
   version="2.1">
   <persistence-unit name="course"
                     transaction-type="RESOURCE_LOCAL">
      <properties>
         <property name="javax.persistence.jdbc.driver"
                value="com.mysql.jdbc.Driver" />
         <property name="javax.persistence.jdbc.url"
                value="jdbc:mysql://noelvaes.eu/StudentDB" />
         <property name="javax.persistence.jdbc.user"
                value="student" />
         <property name="javax.persistence.jdbc.password"
                value="student123" />
         <property
         name="javax.persistence.schema-generation.database.action"
```

```
            value="drop-and-create" />
            <!-- Hibernate specific -->
            <property name="hibernate.show_sql" value="true" />
        </properties>
    </persistence-unit>
</persistence>
```

Dit configuratiebestand maakt deel uit van JPA en is dus gestandaardiseerd. Alle properties die beginnen met de naam `javax.persistence` zijn gestandaardiseerd en staan los van de gebruikte JPA-*provider.*

Met de *property* `javax.persistence.schema-generation.database.action` kunnen we aangeven dat de tabellen automatisch gegenereerd en/of verwijderd worden door JPA. De mogelijke waarden zijn: `none, create, drop-and-create, drop`.

De JPA-*provider* zelf wordt overigens automatisch gedetecteerd. JPA gaat in het *classpath* namelijk op zoek naar een geschikte *provider* en zal de eerste die hij vindt ook gebruiken. Het is daarom ook niet nodig aan te geven welke onderliggende implementatie we gebruiken; het volstaat deze toe te voegen aan het *classpath.* Dit hebben wij gedaan door de juiste afhankelijkheid te definiëren in de POM. Wisselen van *provider* kan dus simpelweg door een andere afhankelijkheid te definiëren.

Afhankelijk van de gebruikte JPA-*provider* kunnen er wel specifieke *properties* toegevoegd worden. Zo is er voor *Hibernate* de *property* `hibernate.show_sql` waarmee we kunnen aangeven of we het SQL-commando willen zien in de *logging.*

In ons voorbeeld maken we gebruik van een MySQL databank met de volgende gegevens:

- **URL**: jdbc:mysql://noelvaes.eu/StudentDB
- **Login:** student
- **Wachtwoord**: student123

Indien we JPA willen gebruiken in een *managed* omgeving zoals een webcontainer, hebben we doorgaans de beschikking over *datasources.* In dit geval dienen we hiervan gebruik te maken. De naam van de *datasource* kunnen we opgeven met de *tag* `<jta-data-source>` of `<non-jta-datasource>` al naargelang deze *datasource* beheerd wordt door een *transaction manager* of niet. Dit valt echter buiten het bestek van deze cursus maar wordt verder behandeld in de cursus EJB.

Verder merken we op dat we de *entity*-klassen kunnen definiëren met de *tag* `<class>`. Dit is evenwel enkel nodig indien we gebruikmaken van meerdere *persistence units* of indien we bepaalde klassen willen uitsluiten. We kunnen hiermee dan aangeven welke klasse bij welke *unit* hoort. In het geval we maar een *persistence unit* in het *archive* (JAR-bestand) hebben, worden de aanwezige klassen gescand op annotaties en worden ze automatisch toegevoegd aan de *persistence unit*.
Indien we de klassen expliciet opgeven met de *tag* `<class>` en we alle andere klassen willen uitsluiten, kunnen we de *tag* `<exclude-unlisted-classes>` gebruiken.

3.4 Het hoofdprogramma

Rest ons nog een klein hoofdprogramma te maken waarmee we een boodschap wegschrijven naar de databank.

```
package messages;
```