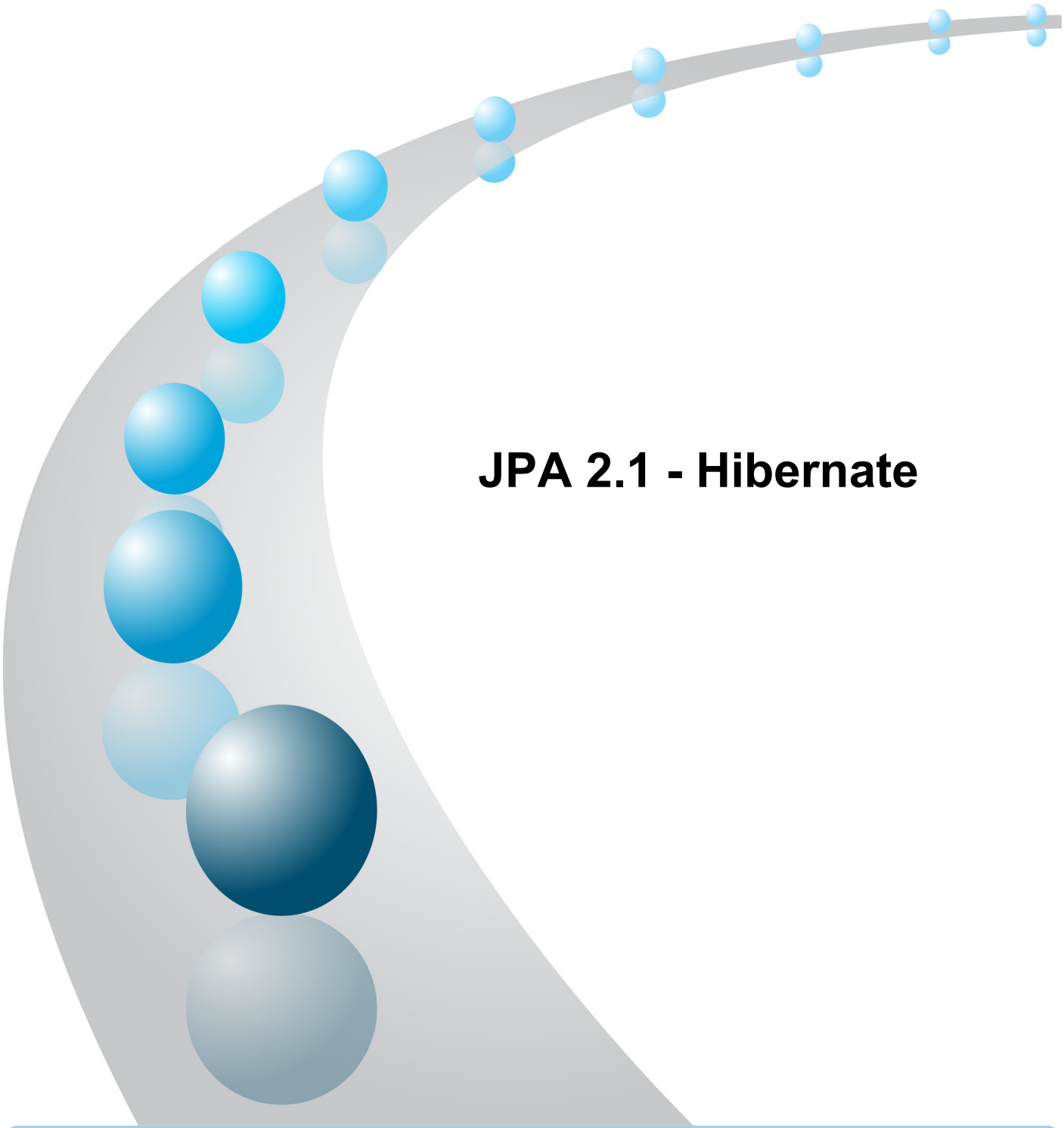
Noël Vaes

Java Trainer & Consultant



JPA 2.1 - Hibernate

Roode Roosstraat 5
3500 Hasselt
België

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

Correspondentie inzake overnemen of reproductie kunt u richten aan:

Noël Vaes
Roode Roosstraat 5
3500 Hasselt
België

Tel: +32 474 38 23 94

noel@noelvaes.eu
www.noelvaes.eu

05/02/2019

Inhoudsopgave

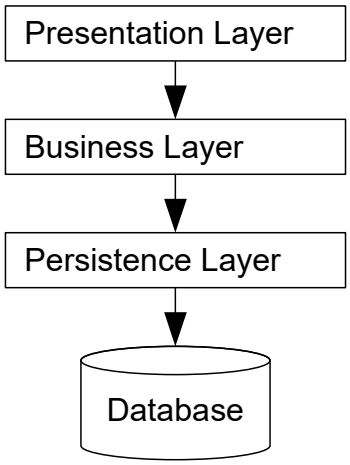
Hoofdstuk 1: Object-Relational Mapping

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.

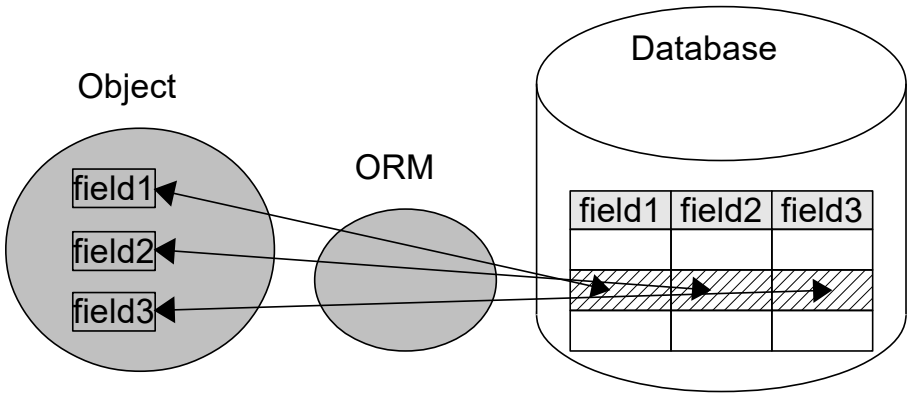
In deze gelaagde architectuur is het enkel de *persistence layer* die communiceert met de databank en zorgt voor het opvragen en wegschrijven van de gegevens. Alle JDBC-code hoort hier dus thuis.

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

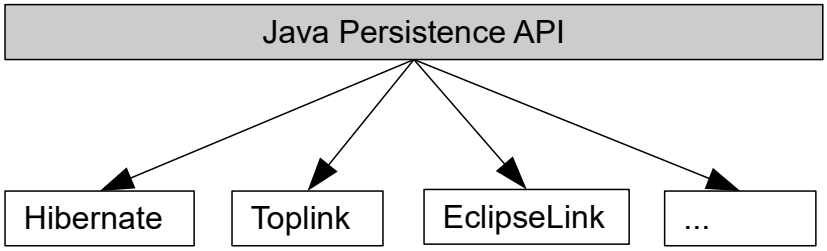
Het concept van *Hibernate* kon op veel bijval rekenen in de wereld van software-ontwikkelaars. Zelfs zij die gebruik maakten van *Enterprise JavaBeans* vervingen de gecompliceerde *Entity Beans vaak* door *Hibernate* om zo het beste van de twee werelden te combineren.

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>5.2.12.Final</version>
      </dependency>
      <dependency>
         <groupId>mysql</groupId>
         <artifactId>mysql-connector-java</artifactId>
         <version>5.1.45</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>
```