Java Persistence API - en samling av klasser som underlättar databas kommunikation

Skapar en brygga mellan javas objektorienterad modell och relationsmodellen som används i alla relationsdatabaser.

Relationsdatabaser består av tabeller medan object modellen består av sammanlänkade grafer, detta skapar svårigheter i anpassningen när man går från den ena modellen till den andra. Följande är de fem klassiska exemplen:

- Granularitet - andel detaljer
- Arv - Det finns inget som liknar arv i relationsmodellen, och om det finns så det stöds inte av alla relationsdatabaser
- Ekvivalens - RDBMS har endast likheten primärnyckel medan ex. java har både a==b och a.equals(b)
- Associationer - I relationsmodellen används främmande nyckel för att associera entiteter, i java måste detta definieras.
- Data navigering - Hur man hittar ett objekt respektive entitet är i grunden annorlunda för det bägge modellerna. I relationsmodellen vore det ineffektivt att gå från objekt till objekt för att komma åt datan. Solving ORM/Relational

Just JPA fokuserar på att definiera en POJO (Plain old java object/Java Bean) som en relation. Detta kallas vanligtvis för ORM (object relational mapping), dvs. att göra om ett objekt till en relation.

Det finns 5 nyckelkomponenter i JPA (Javax.persistence)

- *EntityManagerFactory*
- Kan skapa och hålla **många** *EntityManagers*
- *EntityManager*
- Kan skapa och hålla **en** *EntityTransaction*
- Kan köra **många** *Queries* till databasen
- Kan hålla många *Entities*

Arkitekturen (JPA)
Datalager (applikation) -> Javax.persistence -> Entity -> JDBC -> Relationsdatabas

Hibernate

Hibernate är en utveckling av JPA, dvs. allt som finns i JPA finns också i hibernate.
Medan JPA är mer av en specifikation är hibernate klasser, dvs. implementationer av JPA specifikationen.

Arkitekturen (Hibernate (JPA specifikation))
Datalager (applikation) -> Hibernate API/JPA -> JDBC -> relationsdatabas

För att se just hur alla klasser implementeras, se: [Hibernate overview]

Code-snippet

För att spara en användare

```
/* …. */
Session session = new Configuration().
                    configure().buildSessionFactory().openSession();

Transaction transaction = session.beginTransaction();

User user = new User();
user.setUsername("Jakob");

session.persist(user);

transaction.commit();

session.close();
/* …. */
```

förutsatt att det finns en klass User som har markerats som en entitet med annonteringen @Entity med metoden setUsername(String name). Samt måste det finns en MySQL Databas öppen med en tabell som matchar datatyperna i klassen User.

Utöver det måste det även finnas hibernate konfigurationer satta i en xml-fil under src mappen i ditt projekt. Filen ska heta hibernate.cfg.xml och innehålla data om vilken driver du använder, databas adress och inlogg plus vilka klasser du vill mappa till entiteter. Filen kan se ut ungefär såhär:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
   "-//Hibernate/Hibernate Configuration DTD//EN"
   "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>
          <property
name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
          <property
name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>

          <!-- Assume students is the database name -->
          <property
name="hibernate.connection.url">jdbc:mysql://localhost/demo</property>
          <property name="hibernate.connection.username">root</property>
          <property name="hibernate.connection.password">password</property>


                <mapping class="com.kavzor.hibernate.User" />
        </session-factory>
</hibernate-configuration>
```

Dialekten syftar på vilken SQL standard du använder.