Bazy danych – programowanie JDBC i Hibernate

Krzysztof Kasprowski

JDBC – Java DataBase Connectivity

Autor: Krzysztof Kasprowski

JDBC – Java DataBase Connectivity

JDBC to interfejs programowania opracowany w 1996 r. przez Sun Microsystems, umożliwiający niezależnym od platformy aplikacjom napisanym w języku Java porozumiewanie się z bazami danych za pomocą języka SQL. Interfejs ten jest odpowiednikiem standardu ODBC (Open DataBase Connectivity) opracowanego przez SQL Access Group.

[https://pl.wikipedia.org/wiki/Java_DataBase_Connectivity]

Autor: Krzysztof Kasprowski

JDBC

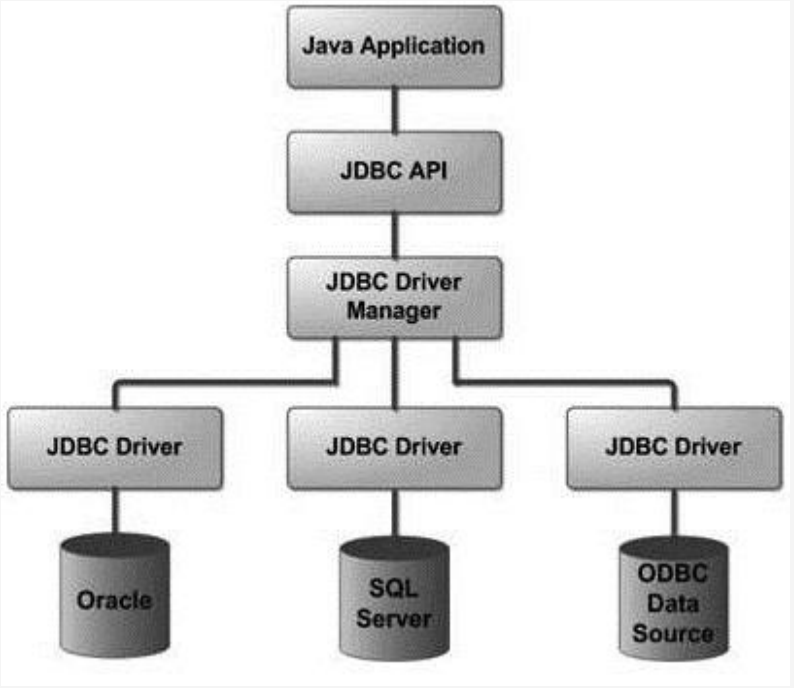
Biblioteka JDBC zawiera API umożliwiające:

- Utworzenie połączenia do bazy danych.
- Przygotowanie zapytania w języku SQL.
- Wykonanie zapytania SQL.
- Przeglądanie i modyfikację wyników.

Autor: Krzysztof Kasprowski

Architektura JDBC

JDBC API używa menadżera sterowników (Driver Manager), który wybiera specyficzny dla konkretnej bazy danych sterownik (Driver), konieczny do utworzenia transparentnego połączenia z bazą danych.



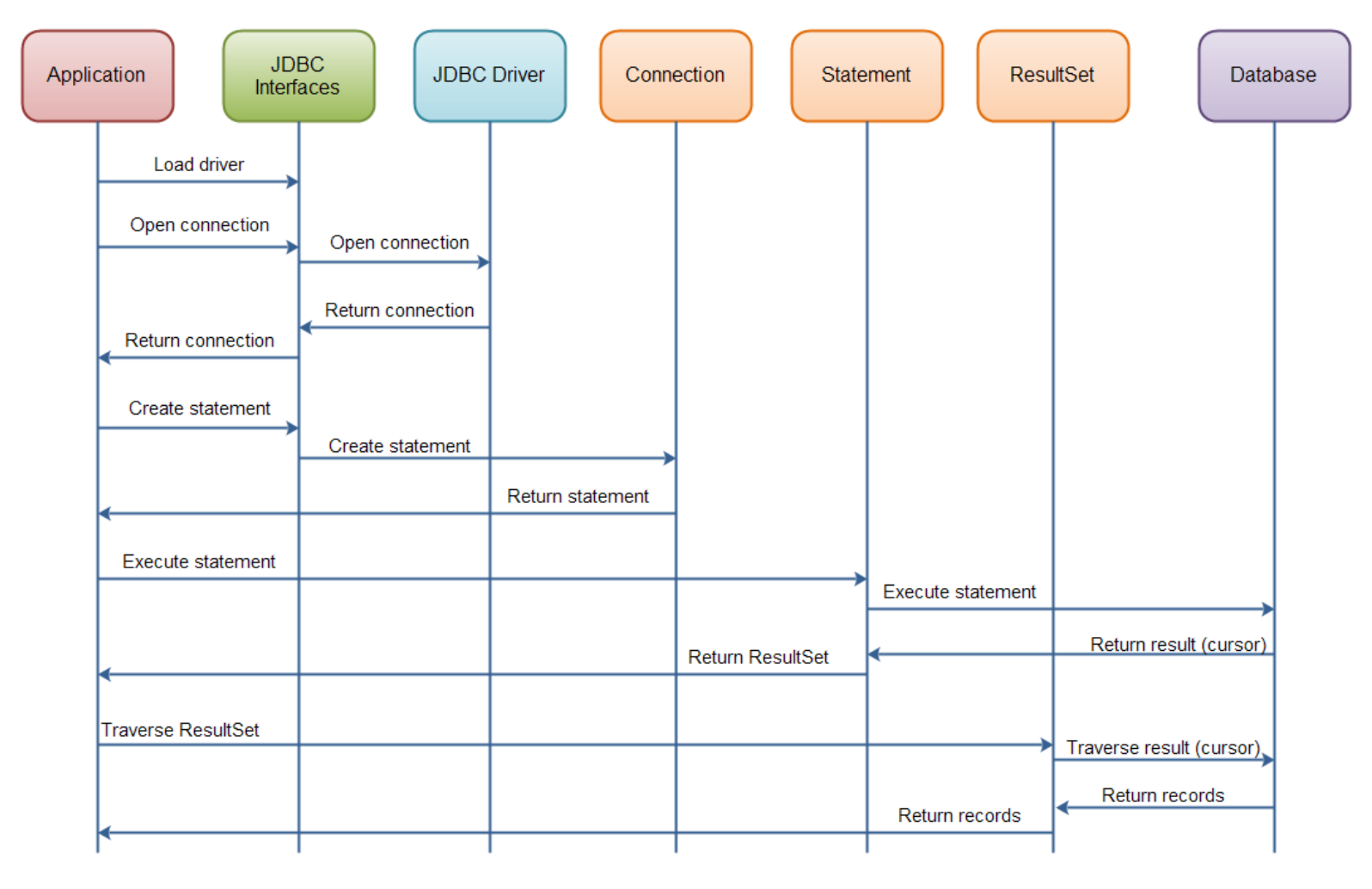
[https://www.tutorialspoint.com/jdbc/jdbc-introduction.htm]

Autor: Krzysztof Kasprowski

JDBC

JDBC API:

- JDBC Drivers
- Connection
- Statement
- PreparedStatement
- CallableStatement
- ResultSet



[http://tutorials.jenkov.com/images/java-jdbc/overview.png]

Sterowniki JDBC

- JDBC-ODBC Driver Bridge – tłumaczenie JDBC na ODBC
- JDBC-Native API – tłumaczenie JDBC na natywne zapytania w C/C++
- JDBC-Net pure Java – tłumaczenie na protokół specyficzny dla bazy danych przez komponent pośredniczący
- 100% Pure Java – bezpośrednia komunikacja z bazą danych przez socket

Dla aplikacji łączących się z jednym typem bazy danych odpowiednim wyborem jest 100% Pure Java. W przypadku aplikacji łączących się z różnymi typami baz danych odpowiednim wyborem jest JDBC-Net pure Java.

Autor: Krzysztof Kasprowski

Connection

- Class.forName(”DRIVER_CLASS_NAME”); //wymagane w wcześniejszych wersjach Javy
- Connection connection = DriverManager.getConnection(url, user, pass);
- connection.close();

Autor: Krzysztof Kasprowski

Statement - SELECT

- Statement statement = connection.createStatement();
- String query = ”SELECT * FROM db_table”
- ResultSet result = statement.executeQuery(query);
- while(result.next()) {result.getXXX(”columnName”);}
- result.close();
- statement.close();

Autor: Krzysztof Kasprowski

Statement – INSERT/UPDATE/DELETE

- Statement statement = connection.createStatement();
- String sql = ”DELETE FROM db_table”
- int result = statement.executeUpdate(sql);
- statement.close();

Autor: Krzysztof Kasprowski

PrepareStatement - SELECT

- String query = ”SELECT * FROM db_table WHERE ID = ?”
- PrepareStatement statement = connection.prepareStatement(query);
- statement.setLong(1, 2L);
- ResultSet result = statement.executeQuery();
- while(result.next()) {result.getXXX(”columnName”);}
- result.close();
- statement.close();

Autor: Krzysztof Kasprowski

PrepareStatement - INSERT/UPDATE/DELETE

- String query = ”DELETE FROM db_table WHERE ID = ?”
- PrepareStatement statement = connection.prepareStatement(query);
- statement.setLong(1, 2L);
- int result = statement.executeUpdate();
- statement.close();

Autor: Krzysztof Kasprowski

Batch updates - INSERT/UPDATE/DELETE

- String sql = ”DELETE FROM db_table WHERE ID = ?”
- PrepareStatement statement = connection.prepareStatement(sql);
- statement.setLong(1, 2L);
- statement.addBatch();
- statement.setLong(1, 44L);
- statement.addBatch();
- int result = statement.executeBatch();
- statement.close();

Transactions

JDBC Connection może działać w następujących trybach:

- Autocommit(true) - transakcja jest zatwierdzana automatycznie dla każdej pojedynczej operacji.
- Autocommit(false) - transakcja jest zatwierdzana przez programistę poprzez wywołanie connection.commit(). Transakcja może zostać wycofana poprzez connection.rollback().

Autor: Krzysztof Kasprowski

Ćwiczenia

Autor: Krzysztof Kasprowski

Podsumowanie

Zalety:

- Elastyczność i kontrola zapytań.
- Prosta składnia, dobre rozwiązanie dla małych aplikacji.
- Niezależny od bazy danych.

Wady:

- Złożoność w dużych aplikacjach.
- Zależny od SQL.
- Konieczność zarządzania połączeniem.

Autor: Krzysztof Kasprowski

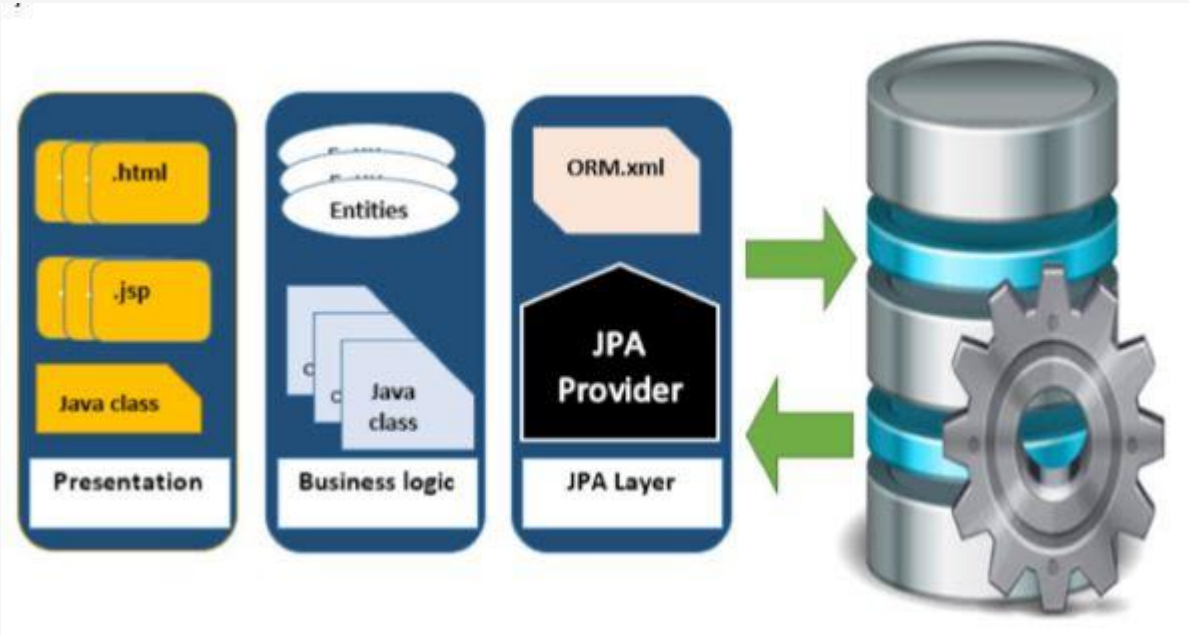
JPA – Java Persistence API

Autor: Krzysztof Kasprowski

Java Persistence API

Oficjalny standard mapowania obiektowo-relacyjnego (ORM) firmy Sun Microsystems dla języka programowania Java.

Definiuje jak tworzyć POJO (Plain Old Java Object) jako encję oraz jak zarządzać encjami z relacjami.



[https://pl.wikipedia.org/wiki/Java_Persistence_API]
[https://www.tutorialspoint.com/jpa/index.htm]

ORM – Object Relational Mapping

Mapowanie relacyjno-obiektowe to sposób odwzorowania obiektowej architektury systemu informatycznego na bazę danych (lub inny element systemu) o relacyjnym charakterze.

Implementacja takiego odwzorowania stosowana jest m.in. w przypadku, gdy tworzony system oparty jest na podejściu obiektowym, a system bazy danych operuje na relacjach.

[https://pl.wikipedia.org/wiki/Mapowanie_obiektowo-relacyjne]

Autor: Krzysztof Kasprowski

Dlaczego ORM?

- Domain Model
  - Skupiony na obiektach biznesowych, a nie na strukturze relacyjnej bazy danych.
  - Każdy obiekt ma indywidualne znaczenie.
  - Obiekty są ze sobą powiązane.
  - Koncepcje programowania obiektowego np. dziedziczenie.

[https://pl.wikipedia.org/wiki/Mapowanie_obiektowo-relacyjne]

Autor: Krzysztof Kasprowski

Dlaczego ORM?

- Redukcja ilości kodu i przyspieszenie kodowania
  - Brak konieczności „ręcznego” mapowania JDBC ResultSet na POJO.
  - Mniej pracy przy synchronizacji kodu i zmian w strukturze relacyjnej bazy danych.
  - Możliwość skupienia się na logice biznesowej.

[https://pl.wikipedia.org/wiki/Mapowanie_obiektowo-relacyjne]

Dlaczego ORM?

- Przenośność
  - Niezależny od bazy danych (z kilkoma wyjątkami np. generacja id).
  - Abstrakcja „query”.
  - Niezależny od SQL. Specyficzny dla danej bazy danych SQL jest generowany automatycznie.
- Wydajność, skalowalność i multi-tenantowość

[https://pl.wikipedia.org/wiki/Mapowanie_obiektowo-relacyjne]

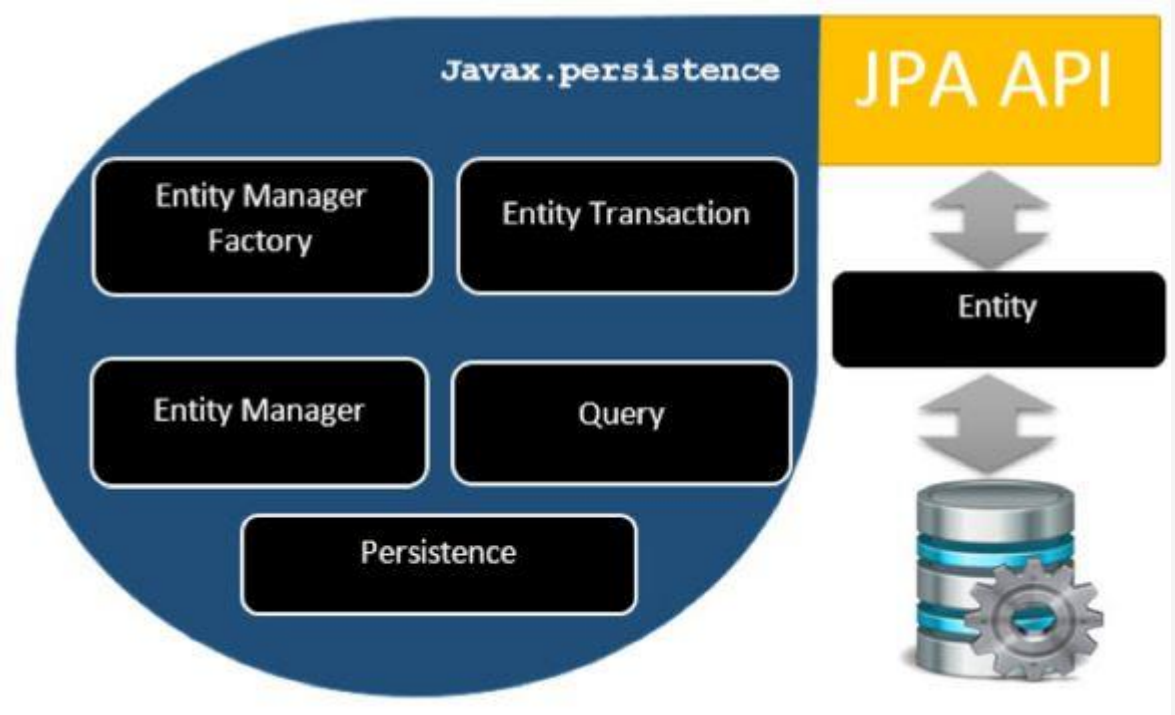
Implementacje JPA API (Providers):

- Hibernate
- Eclipselink
- Toplink
- Spring Data JPA
- Apache Torque

Autor: Krzysztof Kasprowski

Architektura

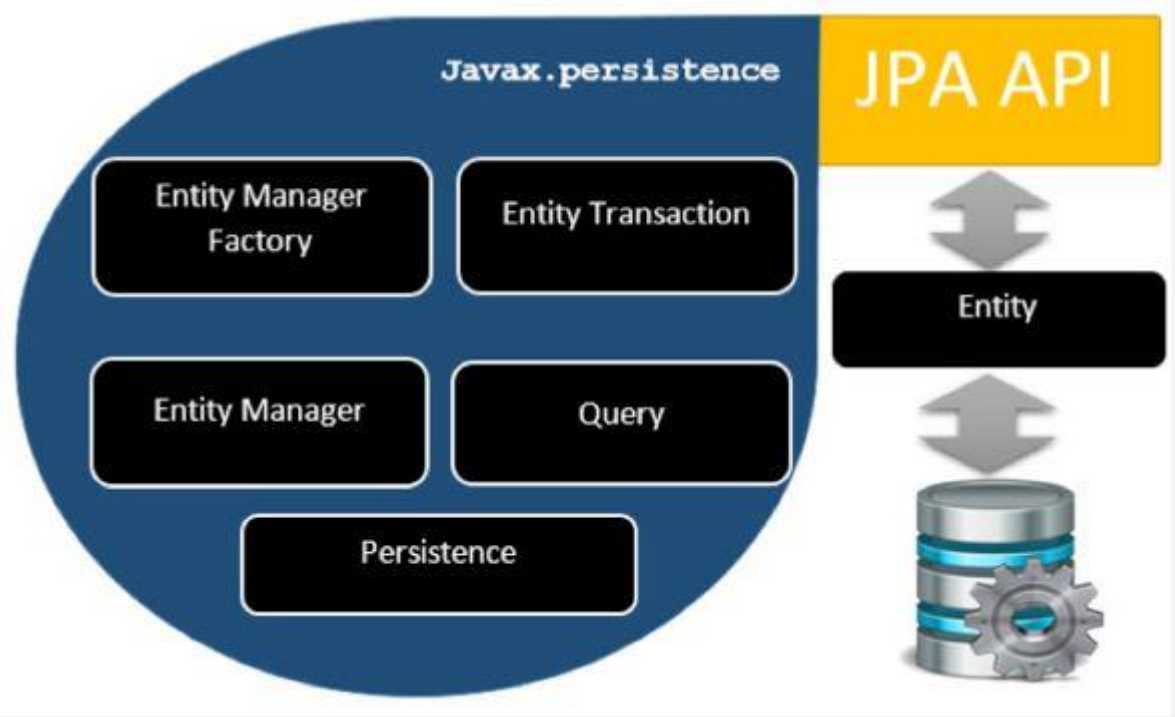
- Persistence – klasa konfiguracyjna, umożliwia utworzenie Entity Manager Factory.
- Entity Manager Factory – tworzy i zarządza wieloma instancjami Entity Manager.
- Entity Manager – zarządza operacjami utrwalania obiektów (persistance).



[https://www.tutorialspoint.com/jpa/index.htm]

Architektura

- Entity Transaction – zarządza transakcjami dla danego Entity Manager.
- Entity – są to trwałe obiekty, przechowywane jako rekordy w bazie danych.
- Query – interfejs zaimplementowany przez dostawcę JPA w celu uzyskania obiektów relacyjnych, które spełniają dane kryteria.



[https://www.tutorialspoint.com/jpa/index.htm]

Ćwiczenia

Autor: Krzysztof Kasprowski

HIBERNATE

Autor: Krzysztof Kasprowski

HIBERNATE

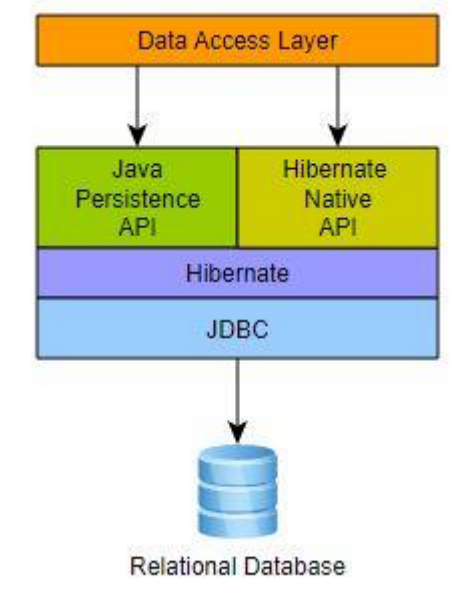
Hibernate framework jest implementacją Java Persistance API (JPA) wykorzystującą JDBC do komunikacji z bazą danych. Umożliwia mapowanie relacyjno-obiektowe (ORM) oraz wykonywanie zapytań.

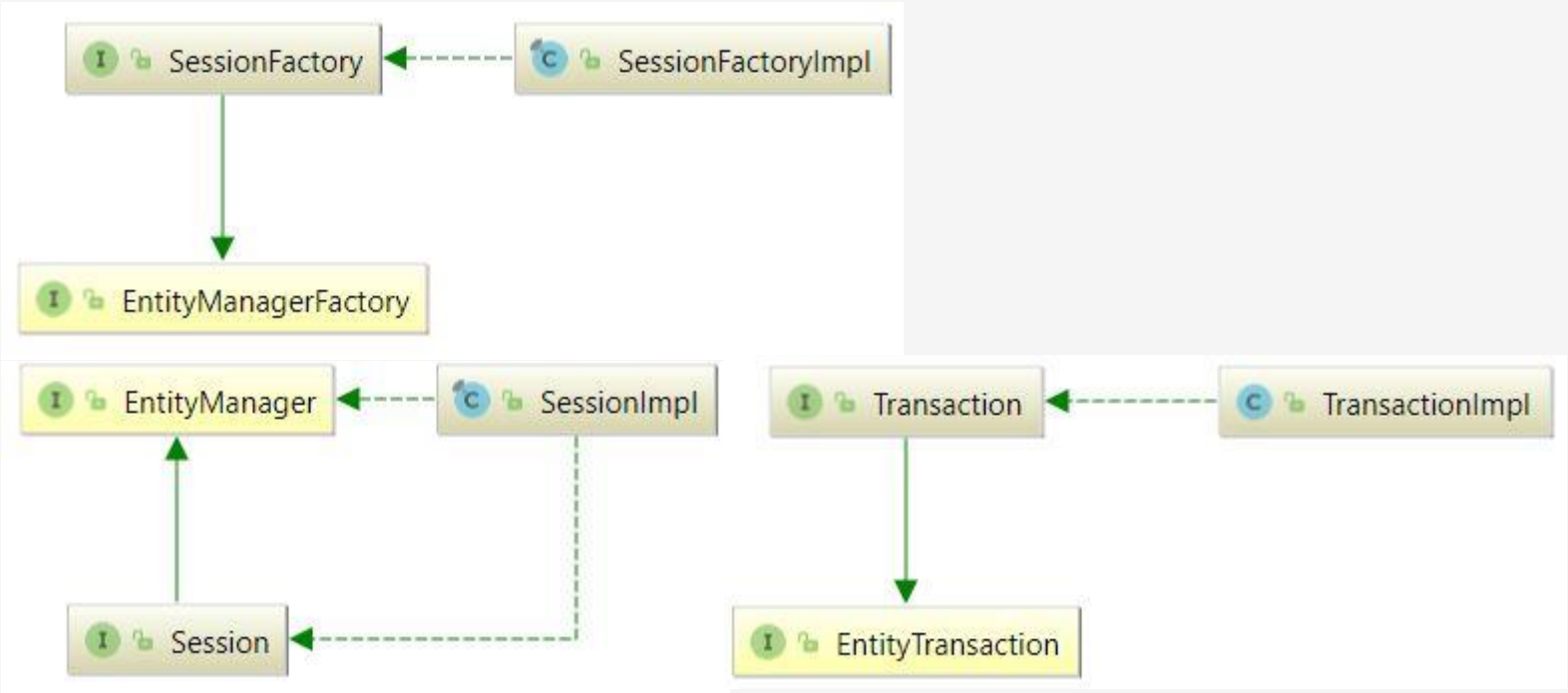
Opiera się na opisie danych z wykorzystaniem XML lub adnotacji.

[http://hibernate.org/]

HIBERNATE

Architektura





[http://docs.jboss.org/hibernate/orm/5.4/userguide/html_single/Hibernate_User_Guide.html#architecture]

Autor: Krzysztof Kasprowski

Architektura

- SessionFactory
  - Ze względu na duży koszt utworzenia w aplikacji powinna być inicjalizowana jedna instancja dla bazy danych.
  - Thread safe.
  - Wczytuje konfigurację ORM i tworzy obiekty Session.
- Session
  - Zapewnia połączenie z bazą danych.
  - Not thread safe.
  - First level cache

Architektura

- Transaction
  - Jednostka pracy z bazą danych, która grupuje operacje.
  - Zarządzany przez transaction manager.
- Entity
  - Reprezentacja danych, która jest mapowana na tabele w bazie danych.
- Query
  - Używa JPQL lub HQL do pobierania danych i utworzenia obiektów.
- Criteria
  - Wykorzystywany do wykonania zapytania w formie obiektowej.

Autor: Krzysztof Kasprowski

Entity

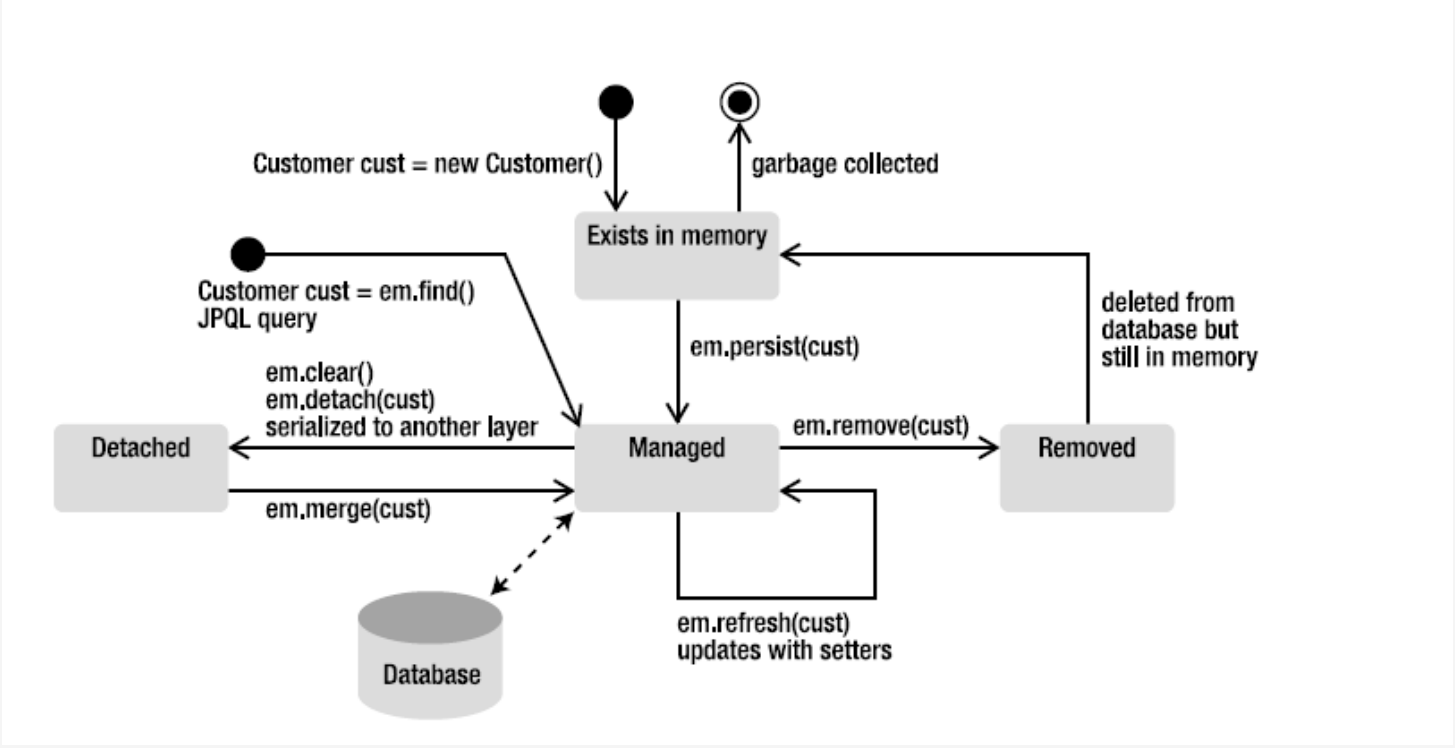
- Klasa POJO, która powinna spełniać założenia konwencji JavaBean. Zalecane jest używanie getterów i setterów oraz prywatnych pól. Konstruktor bezargumentowy jest wymagany. Klasy encji powinny implementować hashCode() i equals().
- @Entity
- @Table
- @Id
- @GeneratedValue
- @Column

Embeddable

- Kompozycja obiektów, w której cykl życia obiektu zagnieżdżonego jest zależny od rodzica.
- @Embeddable
- @Embedded
- @AttributeOverrides

Cykl życia encji

- Transient - nowo utworzony obiekt, niezarządzany i nieistniejący w bazie danych.
- Persistent - powiązany z session, zmiany są odzwierciedlane w bazie danych.
- Detached - niezarządzany i istniejący w bazie danych.
- Removed - oznaczony do usunięcia.



Konfiguracja – hibernate.cfg.xml

```xml
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="connection.driver_class">org.h2.Driver</property>
        <property name="connection.url">jdbc:h2:mem:db1;DB_CLOSE_DELAY=-1;MVCC=TRUE</property>
        <property name="connection.username">username</property>
        <property name="connection.password">password</property>
        <property name="connection.pool_size">1</property>
        <property name="dialect">org.hibernate.dialect.H2Dialect</property>
        <property name="cache.provider_class">org.hibernate.cache.internal.NoCacheProvider</property>
        <property name="show_sql">true</property>
        <property name="hbm2ddl.auto">create</property>
        <property name="hibernate.format_sql">true</property>

        <mapping class="org.sda.Entity"/>

        <mapping resource="org/sda/Entity.hbm.xml"></mapping>
    </session-factory>
</hibernate-configuration>
```

Autor: Krzysztof Kasprowski

Session Factory

```
final StandardServiceRegistry registry = new StandardServiceRegistryBuilder()
        .configure("hibernate.cfg.xml")
        .build();
final Metadata metadata = new MetadataSources(registry)
        .buildMetadata();
final SessionFactory sessionFactory = metadata.buildSessionFactory();
```

Autor: Krzysztof Kasprowski

Session

```
Session session = sessionFactory.openSession();
session.beginTransaction();
Entity entity = createEntity();
session.persist(entity);
session.getTransaction().commit();
session.close();

Session session2 = sessionFactory.openSession();
session.beginTransaction();
Entity foundEntity = session2.find(Entity.class, id);
session.getTransaction().commit();
session.close();
```

Ćwiczenia

Autor: Krzysztof Kasprowski

HQL – Hibernate Query Language

- Zbliżony do SQL
- Wykorzystuje notację obiektową

ClassName class property

```
String hql = "FROM Entity E WHERE E.id = :idParameter";
Query<Entity> query = session.createQuery(hql, Entity.class);
query.setParameter( name: "idParameter", parameterValue);
List<Entity> result = query.list();
```

Autor: Krzysztof Kasprowski

Criteria API

```
CriteriaBuilder builder = session.getCriteriaBuilder();
CriteriaQuery<Person> criteria = builder.createQuery(Person.class);
Root<Person> root = criteria.from(Person.class);
criteria.select(root);

Query<Person> query = session.createQuery(criteria);
List<Person> people = query.getResultList();
```

```
CriteriaBuilder builder = session.getCriteriaBuilder();
CriteriaQuery<String> criteria = builder.createQuery(String.class);
Root<Person> root = criteria.from(Person.class);
criteria.select(root.get("nickName"))
        .where(builder.equal(root.get("name"), o: "Krzysztof"));
List<String> nickNames = session.createQuery(criteria).getResultList();
```

Criteria API

```
CriteriaBuilder builder = session.getCriteriaBuilder();
CriteriaQuery<Person> criteria = builder.createQuery(Person.class);
Root<Person> root = criteria.from(Person.class);
Predicate equalName = builder.equal(root.get("name"), o: "Krzysztof");
Predicate graterAge = builder.greaterThan(root.get("age"), y: 20);
criteria.select(root)
        .where(builder.and(equalName, graterAge))
        .orderBy(builder.asc(root.get("name")));

Query<Person> query = session.createQuery(criteria);
List<Person> people = query.getResultList();
```

Autor: Krzysztof Kasprowski

Ćwiczenia

Autor: Krzysztof Kasprowski

Relacje

Wyróżniamy 4 rodzaje relacji:

- One-to-one
- One-to-many
- Many-to-one
- Many-to-many

Mogą występować w dwóch wariantach:

- Jednokierunkowe (unidirectional)
- Dwukierunkowe (bidirectional)

Autor: Krzysztof Kasprowski

Relacje jednokierunkowa - OneToOne

```
@Entity
public class Phone implements Serializable{

    @OneToOne
    @JoinColumn(name = "DETAILS_ID")
    private PhoneDetails phoneDetails;

}
```

Relacje dwukierunkowa - OneToOne

```
@Entity
public class PhoneDetails implements Serializable {

    @OneToOne
    @JoinColumn(name = "PHONE_ID")
    private Phone phone;

}
```

```
Phone phone = new Phone();
PhoneDetails phoneDetails = new PhoneDetails();
phone.addDetails(phoneDetails);
session.persist(phone);
```

```
@Entity
public class Phone implements Serializable{

    @OneToOne(mappedBy = "phone",
            cascade = CascadeType.ALL,
            orphanRemoval = true)
    private PhoneDetails phoneDetails;

    public void addDetails(PhoneDetails phoneDetails) {
        phoneDetails.setPhone(this);
        this.phoneDetails = phoneDetails;
    }

    public void removeDetails() {
        if (phoneDetails != null) {
            phoneDetails.setPhone(null);
            this.phoneDetails = null;
        }
    }

}
```

Autor: Krzysztof Kasprowski

Relacje jednokierunkowa - OneToMany

```
@Entity
public class Person implements Serializable {

    @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
    private List<Phone> phones = new ArrayList<Phone>();

}
```

Autor: Krzysztof Kasprowski

Relacje jednokierunkowa - ManyToOne

```
@Entity
public class Phone implements Serializable{

    @ManyToOne
    @JoinColumn(name = "PERSON_ID",
    foreignKey = @ForeignKey(name = "PERSON_ID_FK"))
    private Person person;

}
```

Autor: Krzysztof Kasprowski

Relacje dwukierunkowa - OneToMany

```java
@Entity
public class Phone implements Serializable{

    @ManyToOne
    @JoinColumn(name = "PERSON_ID")
    private Person person;

}
```

```java
Person person = new Person();
Phone phone = new Phone();
Phone phone2 = new Phone();

person.addPhone(phone);
person.addPhone(phone2);
session.persist(person);
```

```java
@Entity
public class Person implements Serializable {

    @OneToMany(mappedBy = "person",
            cascade = CascadeType.ALL,
            orphanRemoval = true)
    private List<Phone> phones = new ArrayList<>();

    public void addPhone(Phone phone) {
        phone.setPerson(this);
        this.phones.add(phone);
    }

    public void removePhone(Phone phone) {
        phone.setPerson(null);
        this.phones.remove(phone);
    }
}
```

Relacje jednokierunkowa - ManyToMany

```
@Entity
public class Person implements Serializable {

    @ManyToMany(cascade = CascadeType.PERSIST)
    private List<Address> addresses = new ArrayList<>();

}
```

Autor: Krzysztof Kasprowski

Relacje dwukierunkowa - ManyToMany

```java
@Entity
public class Address {

    @ManyToMany(mappedBy = "addresses")
    private List<Person> owners = new ArrayList<>();

    public List<Person> getOwners() {
        return owners;
    }
}
```

```java
@Entity
public class Person implements Serializable {

    @ManyToMany(cascade = CascadeType.PERSIST)
    @JoinTable(name = "PERSON_ADDRESS",
            joinColumns = @JoinColumn(name = "PERSON_ID"),
            inverseJoinColumns = @JoinColumn(name = "ADDRESS_ID"))
    private List<Address> addresses = new ArrayList<>();

    public void addAddress(Address address) {
        this.addresses.add(address);
        address.getOwners().add(this);
    }

    public void removeAddress(Address address) {
        this.addresses.remove(address);
        address.getOwners().remove( o: this);
    }

}
```

```java
Person person = new Person();
Person owner = new Person();

Address address = new Address();
Address address1 = new Address();

person.addAddress(address);
person.addAddress(address1);
owner.addAddress(address1);

session.persist(person);
session.persist(owner);
```

Autor: Krzysztof Kasprowski

Ćwiczenia

Autor: Krzysztof Kasprowski

Dziedziczenie @Inheritance – Single table

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(discriminatorType = DiscriminatorType.STRING, name = "TYPE")
@DiscriminatorValue("SUPER")
public class SuperClass {
}
```

```
@Entity
@DiscriminatorValue("TWO")
public class SubClassTwo extends SuperClass {
}
```

```
@Entity
@DiscriminatorValue("ONE")
public class SubClassOne extends SuperClass {
}
```

Dziedziczenie @Inheritance – Table per class

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class SuperClass {
}
```

```
@Entity
public class SubClassTwo extends SuperClass {
}
```

```
@Entity
public class SubClassOne extends SuperClass {
}
```

Autor: Krzysztof Kasprowski

Dziedziczenie @Inheritance – Joined

```java
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class SuperClass {
}
```

```java
@Entity
@PrimaryKeyJoinColumn(name = "subclass_id")
public class SubClassOne extends SuperClass {
}
```

```java
@Entity
public class SubClassTwo extends SuperClass {
}
```

Autor: Krzysztof Kasprowski

Implicit vs explicit polymorphism

```java
@Entity
@Polymorphism(type = PolymorphismType.EXPLICIT)
public class ExcludedFromQuery implements PolyInterface {
}
```

```java
@Entity
@Polymorphism(type = PolymorphismType.IMPLICIT) //default
public class IncludedInQuery implements PolyInterface {
}
```

Ćwiczenia

Autor: Krzysztof Kasprowski

Optimistic locking

- Pozwala na dostęp do danych w kilku transakcjach jednocześnie. Przed wprowadzeniem zmian (commit) weryfikuje, czy dane nie zostały zmodyfikowane. W przypadku konfliktu zmiany są wycofane (rollback).
- @Version
- @Source (DB vs. VM)
- @OptimisticLocking (All, Dirty, Version, None)

HIBERNATE

Zalety

- HQL (Hibernate Query Language) – rozszerzony JPQL.
- Criteria API.
- Optymalizacja wydajności – strategie pobierania, caching itd.
- Walidacja i automatyczna generacja struktury bazy danych.

Wady

- Dodatkowa warstwa na JDBC – dłuży czas dostępu.
- Generowanie dodatkowych zapytań.

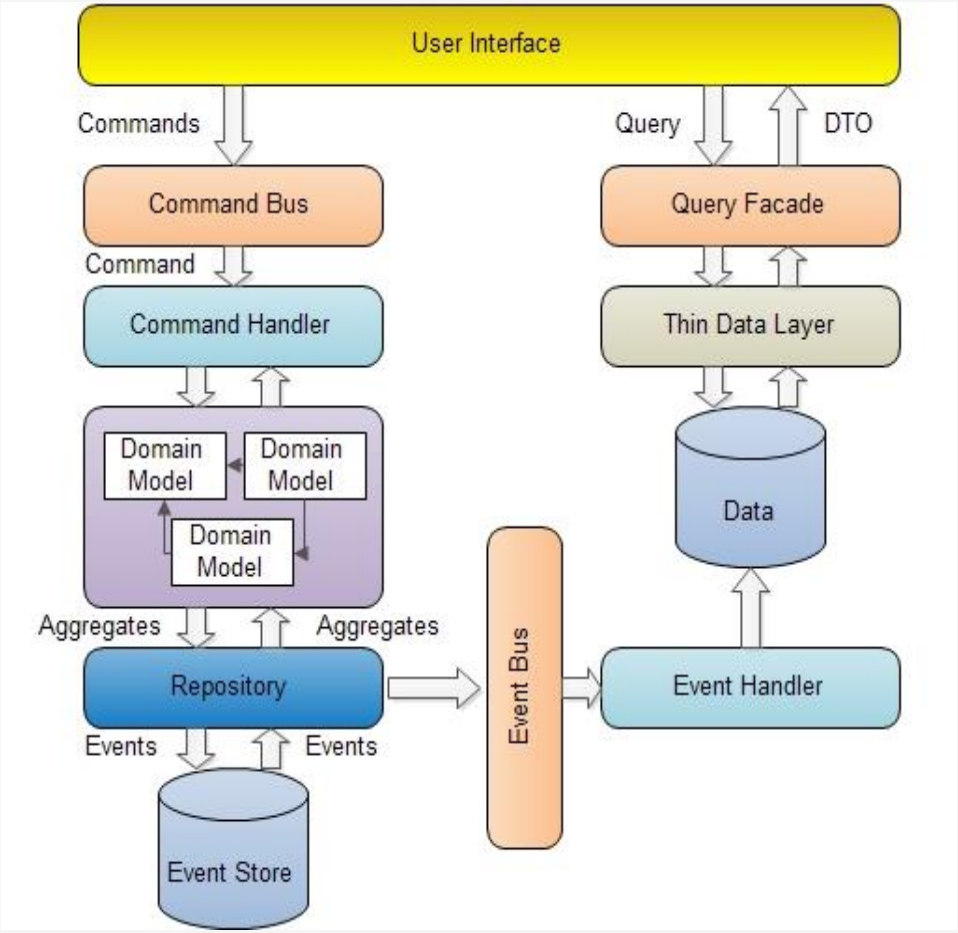
Autor: Krzysztof Kasprowski

CQRS – Command Query Responsibility Segregation

CQS – Command Query Separation

„Pytanie nie powinno zmieniać odpowiedzi.”

- Optymalizacja wydajności.
- Asymetryczna skalowalność.
- Dowolność doboru technologii.
- Mikroserwisy.
- Podział pracy nad dwoma modelami.
- Audyt i możliwość odtworzenia stanu (Event Sourcing)
- Nadmierna złożoność systemu.



[https://www.codeproject.com/]

Dziękuję za uwagę!

Autor: Krzysztof Kasprowski