Руководство по Hibernate. Архитектура.

Hibernate – это ORM фреймворк для Java с открытым исходным кодом. Эта технология является крайне мощной и имеет высокие показатели производительности.

Hibernate создаёт связь между таблицами в базе данных (далее – БД) и Java-классами и наоборот. Это избавляет разработчиков от огромного количества лишней, рутинной работы, в которой крайне легко допустить ошибку и крайне трудно потом её найти..

Схематично это можно изобразить следующим образом:

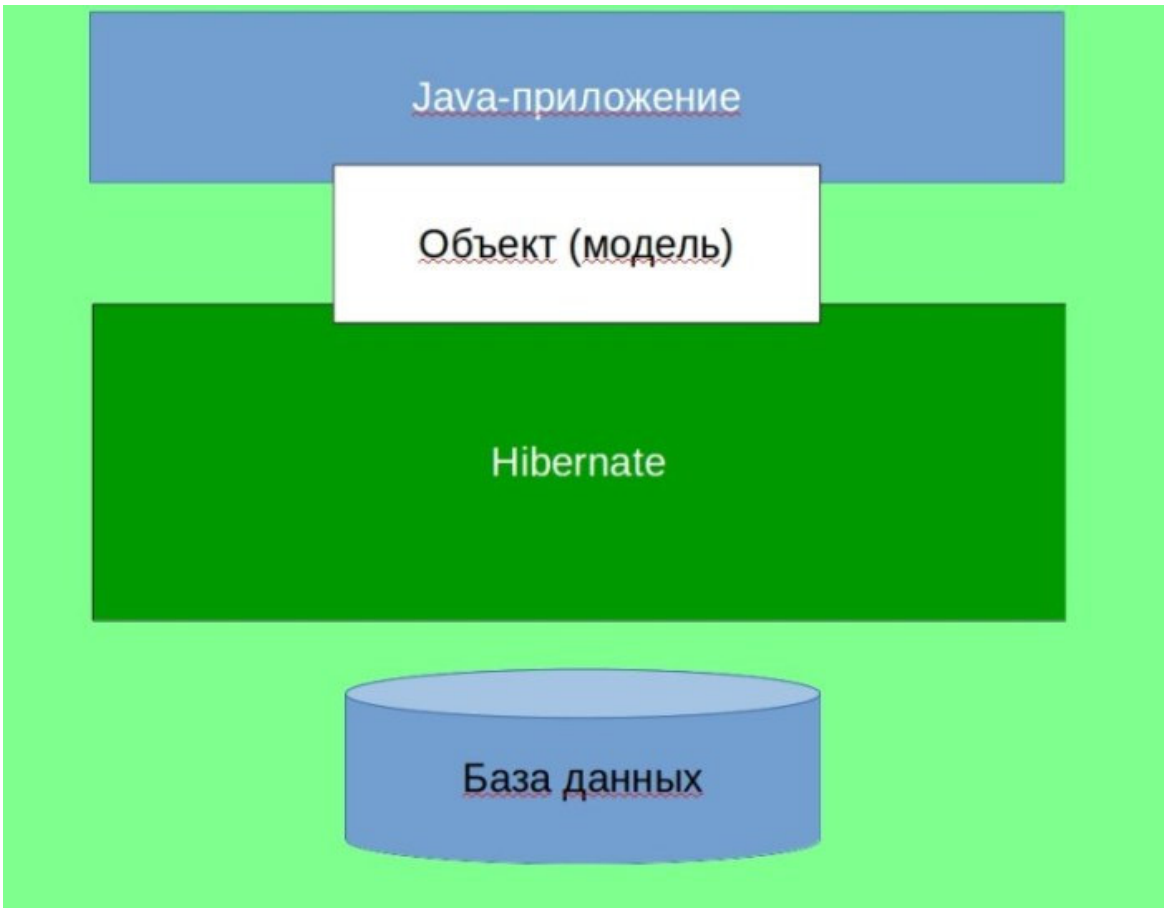
Какие же преимущества даёт нам использование Hibernate?

- Обеспечивает простой API для записи и получения Java-объектов в/из БД.
- Минимизирует доступ к БД, используя стратегии fetching.
- Не требует сервера приложения.
- Позволяет нам не работать с типами данных языка SQL, а иметь дело с привычыми нам типами данных Java.
- Заботится о создании связей между Java-классами и таблицами БД с помощью XML-файлов не внося изменения в программный код.
- Если нам неоходимо изменить БД, то достаточно лишь внести изменения в XML-файлы.

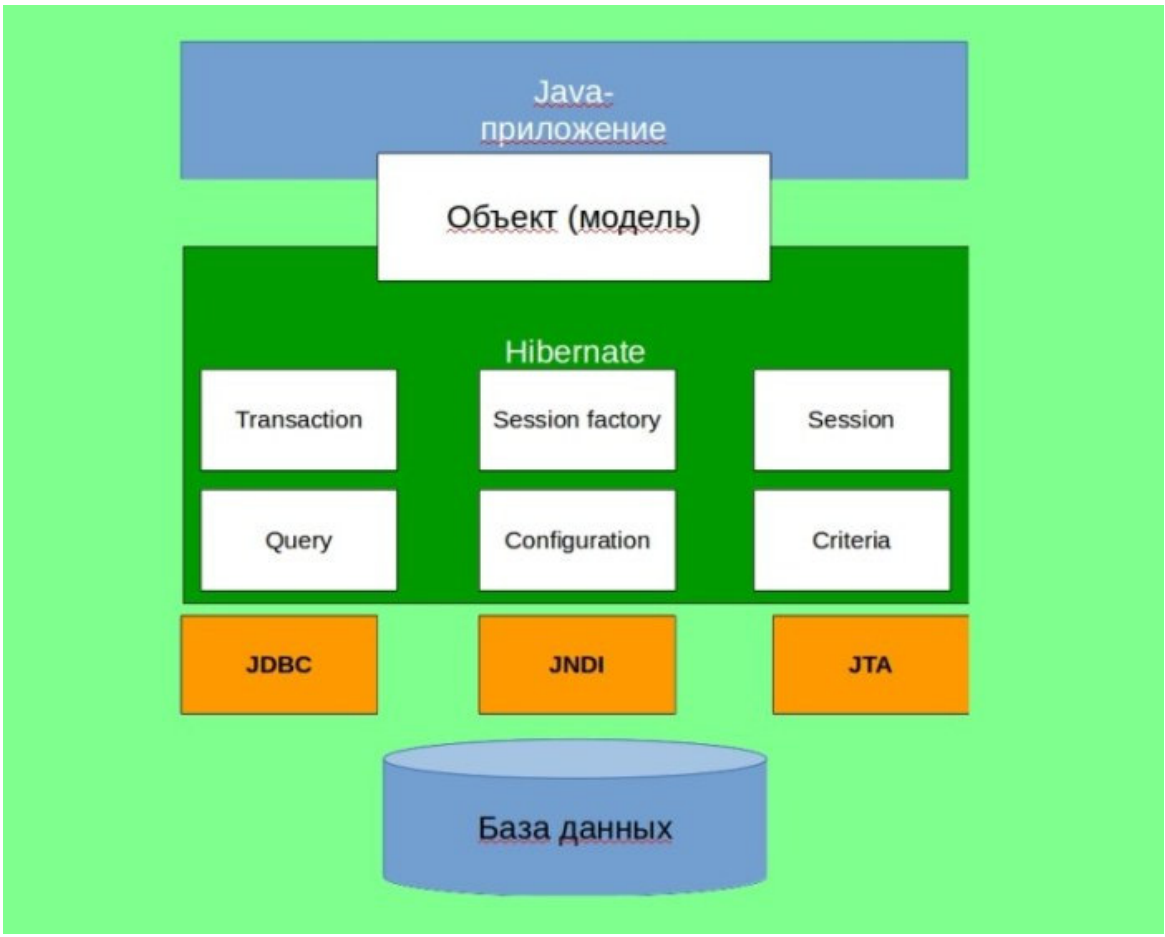
Hibernate поддердивает все основные СУБД: MySQL, Oracle, PostgreSQL, Microsoft SQL Server Database, HSQL, DB2.
Hibernate также может работать в связке с такими технологиями, как Maven и J2EE.

Архитектура

Приложение, которе использует Hibernate (в крайне поверхностном представлении) имеет такую архитектуру:



Если мы рассмотрим строение самого Hibernate более подробно, что этот же рисунок будет выглядеть следующим образом:



Hibernate поддерживает такие API, как JDBC, JNDI, JTA.
JDBC обеспечивает простейший уровень абстракции функциональности для реляционных БД. JTA и JNDI, в свою очередь, позволяют Hibernate использовать серверы приложений J2EE.

Давайье рассмотрим отдельно каждый из элементов Hibernate, которые указаны в схеме:

Transaction

Этот объект представляет собой рабочую единицу работы с БД. В Hibernate транзакции обрабатываются менеджером транзакций.

SessionFactory

Самый важный и самый тяжёлый объект (обычно создаётся в единственном экземпляре, при запуске приложения). Нам необходима как минимум одна SessionFactory для каждой БД, каждый из которых конфигурируется отдельным конфигурационным файлом.

Session

Сессия используется для получения физического соединения с БД. Обычно, сессия создаётся при необходимости, а после этого закрывается. Это связано с тем, что эти объекты крайне легковесны. Чтобы понять, что это такое, модно сказать, что создание, чтение, изменение и удаление объектов происходит через объект Session.

Query

Этот объект использует HQL или SQL для чтения/записи данных из/в БД. Экземпляр запроса используется для связывания параметров запроса, ограничения количества результатов, которые будут возвращены и для выполнения запроса.

Configuration

Этот объект используется для создания объекта SessionFactory и конфигурирует сам Hibernate с помощью конфигурационного XML-файла, который объясняет, как обрабатывать объект Session.

Criteria

Используется для создания и выполнения объектно-ориентированных запроса для получения объектов.

Руководство по Hibernate. Конфигурирование.

Для корректной работы, мы должны передать Hibernate подробную информацию, которая связывает наши Java-классы с таблицами в базе данных (далее – БД). Мы, также, должны указать значения определённых свойств Hibernate.

обычно, вся эта информация помещена в отдельный файл, либо XML-файл – **hibernate.cfg.xml**, либо – **hibernate.properties**.

Здесь мы рассмотрим конфигурирование приложение с помощью XML-файла **hibernate.cfg.xml**.

Для начала рассмотрим ключевые свойства, которые должны быть настроены в типичном приложении:

hibernate.dialect

Указывает HIebrnate диалект БД. Hibernate, в свою очередь, генерирует необходимые SQL-запросы (например, org.hibernate.dialect.MySQLDialect, если мы используем MySQL).

hibernate.connection-driver_class

Указывает класс JDBC драйвера.

hibernate.connection.url

Указывает URL (ссылку) необходимой нам БД (например, jdbc:mysql://localhost:3306/database).

hibernate.connection.username

Указывает имя пользователя БД (например, root).

hibernate.connection.password

Указывает пароль к БД (например, password).

hibernate.connection.pool_size

Ограничивает количество соединений, которые находятся в пуле соединений Hibernate.

hibernate.connection.autocommit
Указывает режим autocommit для JDBC-соединения.

Давайте рассмотрим пример конфигурационного XML-файла.

Исходные данные:
Тип БД: MySQL

Имя базы данных: database

Имя пользователя: root

Пароль: password

```
<?xml version="1.0" encoding="utf-8"?>

<!DOCTYPE hibernate-configuration SYSTEM
"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 test is the database name -->
   <property name="hibernate.connection.url">
      jdbc:mysql://localhost/database
   </property>
   <property name="hibernate.connection.username">
      root
   </property>
   <property name="hibernate.connection.password">
      password
   </property>

</session-factory>
</hibernate-configuration>
```

Руководство по Hibernate. Сессии.

В этой статье я решил более подробно остановиться на сессиях (Session). Напомню, что сессия используется для получения физического соединения с базой данных (далее – БД). Благодаря тому, что сессия является легковесны объектом, его создают (открывают сессию) каждый раз, когда возникает необходимость, а потом, когда необходимо, уничтожают (закрывают сессию). Мы создаём, читаем, редактируем и удаляем объекты с помощью сессий.

Мы стараемся создавать сессии при необходимости, а затем уничтожать их из-за того, что они не являются потоко-защищёнными и не должны быть открыты в течение длительного времени.

Экземпляр класса может находиться в одном из трёх состояний:

- **transient**
  Это новый экземпляр устойчивого класса, который не привязан к сессии и ещё не представлен в БД. Он не имеет значения, по которому может быть идентифицирован.
- **persistent**
  Мы модем создать переходный экземпляр класса, связав его с сессией. Устойчивый экземпляр класса представлен в БД, а значение идентификатора связано с сессией.
- **detached**
  После того, как сессия закрыта, экземпляр класса становится отдельным, независимым экземпляром класса.

В примитивном виде, транзакция выглядит примерно таким образом:

```
Session session = sessionFactory.openSession();
Transaction transaction = null;

    try{
        transaction = session.beginTransaction();

        /**
         * Here we make some work.
         * */

        transaction.commit();
    }catch(Exception e){
        if(transaction !=null){
            transaction.rollback();
            e.printStackTrace();
        }
        e.printStackTrace();
    }finally{
        session.close();
    }
```

В этом примере, в случае исключения, происходит откат (rollback).

В интерфейсе Session определены 23 метода, которые мы можем использовать:

Transaction beginTransaction()

Начинает транзакцию и возвращает объект Transaction.

---

void cancelQuery()

Отменяет выполнение текущего запроса.

---

void clear()

Полностью очищает сессию

Connection close()

Заканчивает сессию, освобождает JDBC-соединение и выполняет очистку.

Criteria createCriteria(String entityName)

Создание нового экземпляра Criteria для объекта с указанным именем.

Criteria createCriteria(Class persistentClass)

Создание нового экземпляра Criteria для указанного класса.

Serializable getIdentifier(Object object)

Возвращает идентификатор данной сущности, как сущности, связанной с данной сессией.

---

void update(String entityName, Object object)

Обновляет экземпляр с идентификатором, указанном в аргументе.

---

void update(Object object)

Обновляет экземпляр с идентификатором, указанном в аргументе.

---

void saveOrUpdate(Object object)

Сохраняет или обновляет указанный экземпляр.

Serializable save(Object object)

Сохраняет экземпляр, предварительно назначив сгенерированный идентификатор.

---

boolean isOpen()

Проверяет открыта ли сессия.

---

boolean isDirty()

Проверят, есть ли в данной сессии какие-либо изменения, которые должны быть синхронизованы с базой данных (далее – БД).

---

boolean isConnected()

Проверяет, подключена ли сессия в данный момент.

Transaction getTransaction()

Получает связанную с этой сессией транзакцию.

---

void refresh(Object object)

Обновляет состояние экземпляра из БД.

SessionFactory getSessionFactory()

Возвращает фабрику сессий (SessionFactory), которая создала данную сессию.

Session get(String entityName, Serializable id)

Возвращает сохранённый экземпляр с указанными именем сущности и идентификатором. Если таких сохранённых экземпляров нет – возвращает null.

---

void delete(String entityName, Object object)

Удаляет сохранённый экземпляр из БД.

---

void delete(Object object)

Удаляет сохранённый экземпляр из БД.

SQLQuery createSQLQuery(String queryString)

Создаёт новый экземпляр SQL-запроса (SQLQuery) для данной SQL-строки.

Query createQuery(String queryString)

Создаёт новый экземпляр запроса (Query) для данной HQL-строки.

Query createFilter(Object collection, String queryString)

Создаёт новый экземпляр запроса (Query) для данной коллекции и фильтра-строки.

Руководство по Hibernate. Сохраняемые классы.

Ключевая функция Hibernate заключается в том, что мы можем взять значения из нашего Java-класса и сохранить их в таблице базы данных (далее – БД). С помощью конфигурационных файлов мы указываем Hibernate как извлечь данные из класса и соединить с определённым столбцами в таблице БД.

Если мы хотим, чтобы экземпляры (объекты) Java-класса в будущем сохранялся в таблице БД, то мы называем их “сохраняемые классы” (persistent class). Для того, чтобы сделать работу с Hibernate максимально удобной и эффективной, мы следует использовать программную модель Простых Старых Java Объектов (Plain Old Java Object – POJO).

Существуют определённые требования к POJO классам. Вот самые главные из них:

- Все классы должны иметь ID для простой идентификации наших объектов в БД и в Hibernate. Это поле класса соединяется с первичным ключом (primary key) таблицы БД.
- Все POJO – классы должны иметь конструктор по умолчанию (пустой).
- Все поля POJO – классов должны иметь модификатор доступа **private** иметь набор getter-ов и setter-ов в стиле JavaBean.
- POJO – классы не должны содержать бизнес-логику.

Мы называем классы POJO для того, чтобы подчеркнуть тот факт, что эти объекты являются экземплярами обычных Java-классов.

Ниже приведён пример POJO – класса, которые соответствует условиям, написанным выше:

```
package net.proselyte.hibernate.pojo;

public class Developer {
    private int id;
    private String firstName;
```

```
    private String lastName;
    private String specialty;
    private int experience;

    /**
     * Default Constructor
     */
    public Developer() {
    }

    /**
     * Plain constructor
     */
    public Developer(int id, String firstName, String lastName, String
specialty, int experience) {
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
        this.specialty = specialty;
        this.experience = experience;
    }


    /**
     * Getters and Setters
     */
    public int getId() {
        return id;
    }

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

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getSpecialty() {
        return specialty;
    }
```

```
    public void setSpecialty(String specialty) {
        this.specialty = specialty;
    }

    public int getExperience() {
        return experience;
    }

    public void setExperience(int experience) {
        this.experience = experience;
    }

    /**
     * toString method (optional)
     */
    @Override
    public String toString() {
        return "Developer{" +
                "id=" + id +
                ", firstName='" + firstName + '\'' +
                ", lastName='" + lastName + '\'' +
                ", specialty='" + specialty + '\'' +
                ", experience=" + experience +
                '}';
    }
}
```

Руководство по Hibernate. Соединяющие файлы.

Чаще всего, когда мы имеем дело с ORM фреймворком, связи между объектами и таблицами в базе данных (далее – БД) указываются в XML – файле.
Давайте рассмотрим наш предыдущий POJO – класс Developer.java

```
package net.proselyte.hibernate.pojo;

public class Developer {
    private int id;
    private String firstName;
    private String lastName;
    private String specialty;
    private int experience;

    /**
     * Default Constructor
     */
    public Developer() {
    }

    /**
     * Plain constructor
     */
```

```
    public Developer(int id, String firstName, String lastName, String specialty, int
experience) {
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
        this.specialty = specialty;
        this.experience = experience;
    }


    /**
     * Getters and Setters
     */
    public int getId() {
        return id;
    }

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

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getSpecialty() {
        return specialty;
    }

    public void setSpecialty(String specialty) {
        this.specialty = specialty;
    }

    public int getExperience() {
        return experience;
    }

    public void setExperience(int experience) {
        this.experience = experience;
    }

    /**
     * toString method (optional)
     */
    @Override
    public String toString() {
        return "Developer{" +
                "id=" + id +
                ", firstName='" + firstName + '\'' +
                ", lastName='" + lastName + '\'' +
                ", specialty='" + specialty + '\'' +
                ", experience=" + experience +
                '}';
```

```
    }
}
```

Теперь создадим таблицу в БД под названием HIBERNATE_DEVELOPERS:

```
CREATE TABLE HIBERNATE_DEVELOPERS(
   ID INT NOT NULL AUTO_INCREMENT,
   FIRST_NAME VARCHAR(50) DEFAULT NULL,
   LAST_NAME VARCHAR(50) DEFAULT NULL,
   SPECIALTY VARCHAR(50) DEFAULT NULL,
   EXPERIENCE INT DEFAULT NULL,
   PRIMARY KEY(ID)
);
```

На данный момент у нас есть две независимых друг от друга сущности: POJO – класс Developer.java и таблица в БД HIBERNATE_DEVELOPERS.

Для того, чтобы связать их друг с другом и получить возможность сохранять значения полей класса, нам необходимо объяснить, как именно это делать Hibernate фреймворку.
Чтобы это сделать, мы создаём конфигурационной XML – файл Developer.hbm.xml
Вот этот файл:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD//EN"
        "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="net.proselyte.hibernate.pojo.Developer" table="HIBERNATE_DEVELOPERS">
        <meta attribute="class-description">
            This class contains developer's details.
        </meta>
        <id name="id" type="int" column="id">
            <generator class="native"/>
        </id>
        <property name="firstName" column="first_name" type="string"/>
        <property name="lastName" column="last_name" type="string"/>
        <property name="specialty" column="last_name" type="string"/>
        <property name="experience" column="salary" type="int"/>
    </class>
</hibernate-mapping>
```

На данный момент мы ещё не можем сказать, что наше приложение готово, но у нас есть часть необходимых конфигураций. Давайте разберем их.

- **<hibernate-mapping>**
  Это ключевой тег , который должен быть в каждом XML – файле для связывания (mapping). Внутри этого тега мы и конфигурируем наши связи.
- **<class>**
  Тег <class> используется для того, чтоы указать связь между POJO – классов и таблицей в БД. Имя класса указывается с помощью свойства **name**, имя таблицы в БД – с помощью свойства **table**.
- **<meta>**
  Опциональный (необязательный) тег, внутри которого мы можем добавить описание класса.

- **<id>**
  Тег <id > связывает уникальный идентификатор ID в POJO – классе и первичный ключ (primary key) в таблице БД. Свойство **name** соединяет поле класса со свойством **column**, которое указывает нам колонку в таблице БД. Свойство **type** определяет тип связывания (mapping) и используется для конфертации типа данных Java в тип данных SQL.
- **<generator>**
  Этот тег внутри тега <id> используется для того, что генерировать первичные ключи автоматически. Если мы указываем это свойство **native**, как в примере, приведённом выше, то Hibernate сам выберет алгоритм (**identity, hilo, sequence**) в зависимости от возможностей БД.
- **<property>**
  Мы используем этот тег для того, чтобы связать (map) конкретное поле POJO – класса с конкретной колонкой в таблице БД. Свойство **name** указывает поле в классе, в то время как свойство **column** указывает на колонку в таблице БД. Свойство **type** указывает тип связывания (mapping) и конвертирует тип данных Java в тип данных SQL.

Существуют также и другие теги, которые могут быть использованы в конфигурационном XML – файле, которые не были указаны в этой теме. Но в течение всего цикла статей, посвящённых Hibernate, мы постараемся поговорить о большинстве из них.

Руководство по Hibernate. Пример простого приложения.

В предыдущих статьях из цикла, посвященного Hibernate мы рассмотрели базовые понятия и ключевые функции этого ORM фреймворка. В этой статье мы создадим небольшое приложение, которое резюмирует всё, о чём мы говорили до этого.

Итак, во-первых, мы создадим POJO – класс Developer.java, который соответствует правилам POJO.

Developer.java

```
package net.proselyte.hibernate.example;

public class Developer {
    private int id;
    private String firstName;
    private String lastName;
    private String specialty;
    private int experience;

    /**
     * Default Constructor
     */
    public Developer() {
    }

    /**
     * Plain constructor
     */
    public Developer(int id, String firstName, String lastName, String specialty, int
experience) {
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
        this.specialty = specialty;
        this.experience = experience;
```

```
    }


    /**
     * Getters and Setters
     */
    public int getId() {
        return id;
    }

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

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getSpecialty() {
        return specialty;
    }

    public void setSpecialty(String specialty) {
        this.specialty = specialty;
    }

    public int getExperience() {
        return experience;
    }

    public void setExperience(int experience) {
        this.experience = experience;
    }

    /**
     * toString method (optional)
     */
    @Override
    public String toString() {
        return "Developer{" +
                "id=" + id +
                ", firstName='" + firstName + '\'' +
                ", lastName='" + lastName + '\'' +
                ", specialty='" + specialty + '\'' +
                ", experience=" + experience +
                '}';
    }
}
```

Вторым шагом будет создание таблицы в базе данных (далее – БД).
HIBERNATE_DEVELOPERS

```
CREATE TABLE HIBERNATE_DEVELOPERS(
   ID INT NOT NULL AUTO_INCREMENT,
   FIRST_NAME VARCHAR(50) DEFAULT NULL,
   LAST_NAME VARCHAR(50) DEFAULT NULL,
   SPECIALTY VARCHAR(50) DEFAULT NULL,
   EXPERIENCE INT DEFAULT NULL,
   PRIMARY KEY(ID)
);
```

Шаг 3 – создание конфигурационного файла hibernate.cfg.xml

hibernate.cfg.xml

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration SYSTEM
"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 ИМЯ ВАШЕЙ БД is the database name -->
<property name="hibernate.connection.url">
jdbc:mysql://localhost/ИМЯ_ВАШЕЙ_БАЗЫ_ДАННЫХ
</property>
<property name="hibernate.connection.username">
ВАШЕ ИМЯ ПОЛЬЗОВАТЕЛЯ
</property>
<property name="hibernate.connection.password">
ВАШ ПАРОЛЬ
</property>

<!-- List of XML mapping files -->
<mapping resource="Developer.hbm.xml"/>

</session-factory>
</hibernate-configuration>
```

Четвёртый шаг – создание конфигурационного XML – файла Developer.hbm.xml

Developer.hbm.xml

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD//EN"
        "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="net.proselyte.hibernate.example.model.Developer"
table="HIBERNATE_DEVELOPERS">
```

```
        <meta attribute="class-description">
            This class contains developer's details.
        </meta>
        <id name="id" type="int" column="ID">
            <generator class="native"/>
        </id>
        <property name="firstName" column="FIRST_NAME" type="string"/>
        <property name="lastName" column="LAST_NAME" type="string"/>
        <property name="specialty" column="SPECIALTY" type="string"/>
        <property name="experience" column="EXPERIENCE" type="int"/>
    </class>
</hibernate-mapping>
```

И финальный шаг – создание основного класса приложения DeveloperRunner.java

DeveloperRunner.java

```
package net.proselyte.hibernate.example;

import net.proselyte.hibernate.example.model.Developer;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;

import java.util.List;

public class DeveloperRunner {
    private static SessionFactory sessionFactory;

    public static void main(String[] args) {
        sessionFactory = new Configuration().configure().buildSessionFactory();

        DeveloperRunner developerRunner = new DeveloperRunner();

        System.out.println("Adding developer's records to the DB");
        /**
         *  Adding developer's records to the database (DB)
         */
        developerRunner.addDeveloper("Proselyte", "Developer", "Java Developer", 2);
        developerRunner.addDeveloper("Some", "Developer", "C++ Developer", 2);
        developerRunner.addDeveloper("Peter", "UI", "UI Developer", 4);

        System.out.println("List of developers");
        /**
         * List developers
         */
        List developers = developerRunner.listDevelopers();
        for (Developer developer : developers) {
            System.out.println(developer);
        }
        System.out.println("==================================");
        System.out.println("Removing Some Developer and updating Proselyte");
        /**
         * Update and Remove developers
         */
        developerRunner.updateDeveloper(10, 3);
        developerRunner.removeDeveloper(11);

        System.out.println("Final list of developers");
        /**
         * List developers
```

```
         */
        developers = developerRunner.listDevelopers();
        for (Developer developer : developers) {
            System.out.println(developer);
        }
        System.out.println("==================================");

    }

    public void addDeveloper(String firstName, String lastName, String specialty, int
experience) {
        Session session = sessionFactory.openSession();
        Transaction transaction = null;

        transaction = session.beginTransaction();
        Developer developer = new Developer(firstName, lastName, specialty,
experience);
        session.save(developer);
        transaction.commit();
        session.close();
    }

    public List listDevelopers() {
        Session session = this.sessionFactory.openSession();
        Transaction transaction = null;

        transaction = session.beginTransaction();
        List developers = session.createQuery("FROM Developer").list();

        transaction.commit();
        session.close();
        return developers;
    }

    public void updateDeveloper(int developerId, int experience) {
        Session session = this.sessionFactory.openSession();
        Transaction transaction = null;

        transaction = session.beginTransaction();
        Developer developer = (Developer) session.get(Developer.class, developerId);
        developer.setExperience(experience);
        session.update(developer);
        transaction.commit();
        session.close();
    }

    public void removeDeveloper(int developerId) {
        Session session = this.sessionFactory.openSession();
        Transaction transaction = null;

        transaction = session.beginTransaction();
        Developer developer = (Developer) session.get(Developer.class, developerId);
        session.delete(developer);
        transaction.commit();
        session.close();
    }

}
```

Если все настройки были сделаны правильно, то мы получаем следующий **результат работы программы**:

```
Developer:
id: 10
```

```
First Name: Proselyte
Last Name: Developer
Specialty: Java Developer
Experience: 2

Developer:
id: 11
First Name: Some
Last Name: Developer
Specialty: C++ Developer
Experience: 2

Developer:
id: 12
First Name: Peter
Last Name: UI
Specialty: UI Developer
Experience: 4

==================================
Removing Some Developer and updating Proselyte
Final list of developers
Developer:
id: 10
First Name: Proselyte
Last Name: Developer
Specialty: Java Developer
Experience: 3

Developer:
id: 12
First Name: Peter
Last Name: UI
Specialty: UI Developer
Experience: 4

==================================
```

Руководство по Hibernate. Виды связей.

До этого момента мы рассматривали только простейшие виды связей между классами и таблицами в базах данных (далее – БД). Но давайте рассмотрим более детально виды связей в ORM.
Связи в ORM делятся на 3 группы:

- **Связывание коллекций**
- **Ассоциативное связывание**
- **Связывание компонентов**

Рассмотрим каждую из них:

Связывание коллекций

Если среди значений класса есть коллекции (collections) каких-либо значений, мы можем связать (map) их с помощью любого интерфейса коллекций, доступных в Java.

В Hibernate мы можем оперировать следующими коллекциями:

---

java.util.List (ссылка на пример)

Связывается (mapped) с помощью элемента <list> и инициализируется с помощью java.util.ArrayList

---

java.util.Collection (ссылка на пример)

Связывается (mapped) с помощью элементов <bag> или <ibag> и инициализируется с помощью java.util.ArrayList

---

java.util. Set (ссылка на пример)

Связывается (mapped) с помощью элемента <set> и инициализируется с помощью java.util.HashSet

---

java.util.SortedSet (ссылка на пример)

Связывается (mapped) с помощью элемента <set> и инициализируется с помощью java.util.TreeSet. В качестве параметра для сравнения может выбрать либо компаратор, либо естественный порядок.

---

java.util.Map (ссылка на пример)

Связывается (mapped) с помощью элемента <map> и инициализируется с помощью java.util.HashMap.

---

java.util.SortedMap (ссылка на пример)

Связывается (mapped) с помощью элемента <map> и инициализируется с помощью java.util.TreeMap. В качестве параметра для сравнения может выбрать либо компаратор, либо естественный порядок.

Ассоциативное связывание

Связывание ассоциаций – это связывание (mapping) классов и отношений между таблицами в БД. Существует 4 типа таких зависимостей:

Many-to-One (ссылка на пример)

Связывание (mapping) отношений many-to-one с использованием Hibernate.

One-to-One (ссылка на пример)

Связывание (mapping) отношений one-to-one с использованием Hibernate.

One-to-Many (ссылка на пример)

Связывание (mapping) отношений one-to-many с использованием Hibernate.

Many-to-Many (ссылка на пример)

Связывание (mapping) отношений many-to-many с использованием Hibernate.

Связывание компонентов

Возможна ситуация, при которой наш Java – класс имеет ссылку на другой класс, как одну из переменных. Если класс, на который мы ссылаемся не имеет своего собственного жизненного цикла и полностью зависит от жизненного цикла класса, который на него ссылается, то класс, на который ссылаются называется *классом Компонентом (Component Class).*

Руководство по Hibernate. Аннотации.

Во всех предыдущих примерах мы использовали конфигурационные XML – файлы для конфигурирования Hibernate. В этих XML – файлах мы указывали Hibernate с какой таблицей в нашей базе данных (далее – БД) необходимо связать тот или иной POJO – класс и к каким колонкам относятся те или иные поля в этом классе.

Но в Hibernate предусмотрена возможность конфигурирования приложения с помощью аннотаций.

Аннотации являются мощным инструментом для предоставления метаданных, а также намного нагляднее при чтении нашего кода другим разработчиком.

Обязательными аннотациями являются следующие:

@Entity

Эта аннотация указывает Hibernate, что данный класс является сущностью (entity bean). Такой класс должен иметь конструктор по-умолчанию (пустой конструктор).

@Table

С помощью этой аннотации мы говорим Hibernate, с какой именно таблицей необходимо связать (map) данный класс. Аннотация **@Table** имеет различные атрибуты, с помощью которых мы можем указать *имя таблицы, каталог, БД и уникальность столбцов в таблиц БД*.

@Id

С помощью аннотации **@Id** мы указываем *первичный ключ (Primary Key)* данного класса.

@GeneratedValue

Эта аннотация используется вместе с аннотацией **@Id** и определяет такие параметры, как **strategy** и **generator**.

@Column

Аннотация **@Column** определяет к какому столбцу в таблице БД относится конкретное поле класса (аттрибут класса).

Наиболее часто используемые атрибуты аннотации **@Column** такие:

- **name**
  Указывает имя столбца в таблице
- **unique**
  Определяет, должно ли быть данное значение уникальным
- **nullable**
  Определяет, может ли данное поле быть NULL, или нет.
- **length**
  Указывает, какой размер столбца (например количество символов, при использовании String).

Для понимания того, как это работает на практике, рассмотрим пример небольшого приложения.

Пример:

Исходный код проекта можно скачать по ***ЭТОЙ ССЫЛКЕ***.

Шаг 1. Создадим таблицу в нашей БД.

HIBERNATE_DEVELOPERS

```
CREATE TABLE HIBERNATE_DEVELOPERS(
   ID INT NOT NULL AUTO_INCREMENT,
```

```
   FIRST_NAME VARCHAR(50) DEFAULT NULL,
   LAST_NAME VARCHAR(50) DEFAULT NULL,
   SPECIALTY VARCHAR(50) DEFAULT NULL,
   EXPERIENCE INT DEFAULT NULL,
   PRIMARY KEY(ID)
);
```

Шаг 2. Создадим POJO – класс.

Developer.java

```
package net.proselyte.hibernate.annotations;

import javax.persistence.*;

@Entity
@Table(name = "HIBERNATE_DEVELOPERS")
public class Developer {
    @Id
    @GeneratedValue (strategy = GenerationType.AUTO)
    @Column (name = "id")
    private int id;
    @Column (name = "FIRST_NAME")
    private String firstName;
    @Column (name = "LAST_NAME")
    private String lastName;
    @Column (name = "SPECIALTY")
    private String specialty;
    @Column (name = "EXPERIENCE")
    private int experience;

    /**
     * Default Constructor
     */
    public Developer() {
    }

    /**
     * Plain constructor
     */
    public Developer(String firstName, String lastName, String specialty, int
experience) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.specialty = specialty;
        this.experience = experience;
    }

    /**
     * Getters and Setters
     */
    public int getId() {
        return id;
    }

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

    public String getFirstName() {
        return firstName;
    }
```

```
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getSpecialty() {
        return specialty;
    }

    public void setSpecialty(String specialty) {
        this.specialty = specialty;
    }

    public int getExperience() {
        return experience;
    }

    public void setExperience(int experience) {
        this.experience = experience;
    }

    /**
     * toString method (optional)
     */
    @Override
    public String toString() {
        return "Developer:\n" +
                "id: " + id +
                "\nFirst Name: " + firstName + "\n" +
                "Last Name: " + lastName + "\n" +
                "Specialty: " + specialty + "\n" +
                "Experience: " + experience + "\n";
    }
}
```

Шаг 3. Создадим конфигурационные файлы

hibernate.cfg.xml

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration SYSTEM
        "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 PROSELYTE_TUTORIAL is the database name -->
        <property name="hibernate.connection.url">
```

```
            jdbc:mysql://localhost/ИМЯ_ВАШЕЙ_БД
        </property>
        <property name="hibernate.connection.username">
            ВАШЕ_ИМЯ_ПОЛЬЗОВАТЕЛЯ
        </property>
        <property name="hibernate.connection.password">
            ВАШ_ПАРОЛЬ
        </property>

        <!-- List of XML mapping files -->
        <mapping resource="Developer.hbm.xml"/>

    </session-factory>
</hibernate-configuration>
```

Developer.hbm.xml

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

<hibernate-mapping>
    <class name="net.proselyte.hibernate.annotations.Developer"
table="HIBERNATE_DEVELOPERS">
        <meta attribute="class-description">
            This class contains developer details.
        </meta>
        <id name="id" type="int" column="id">
            <generator class="native"/>
        </id>
        <property name="firstName" column="FIRST_NAME" type="string"/>
        <property name="lastName" column="LAST_NAME" type="string"/>
        <property name="specialty" column="SPECIALTY" type="string"/>
        <property name="experience" column="EXPERIENCE" type="int"/>
    </class>

</hibernate-mapping>
```

Шаг 4. Создадим класс DeveloperRunner.java

DeveloperRunner.java

```
package net.proselyte.hibernate.annotations;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;

import java.util.List;

public class DeveloperRunner {
    private static SessionFactory sessionFactory;

    public static void main(String[] args) {
        sessionFactory = new Configuration().configure().buildSessionFactory();

        DeveloperRunner developerRunner = new DeveloperRunner();
```

```
        System.out.println("Adding Developer's records to the database");
        Integer developerId1 = developerRunner.addDeveloper("Proselyte", "Developer",
"Java Developer", 2);
        Integer developerId2 = developerRunner.addDeveloper("Some", "Developer", "C++
Developer", 2);
        Integer developerId3 = developerRunner.addDeveloper("Peter", "Team Lead", "Java
Team Lead", 6);

        System.out.println("List of Developers:");
        developerRunner.listDevelopers();

        System.out.println("Removing \'Some Developer\' and updating \'Proselyte
Developer\''s experience:");
        developerRunner.removeDeveloper(developerId2);
        developerRunner.updateDeveloper(developerId1, 3);

        System.out.println("Final list of Developers:");
        developerRunner.listDevelopers();
        sessionFactory.close();
    }

    public Integer addDeveloper(String firstName, String lastName, String specialty,
int experience) {
        Session session = sessionFactory.openSession();
        Transaction transaction = null;
        Integer developerId = null;

        transaction = session.beginTransaction();
        Developer developer = new Developer(firstName, lastName, specialty,
experience);
        developerId = (Integer) session.save(developer);
        transaction.commit();
        session.close();
        return developerId;
    }

    public void listDevelopers() {
        Session session = sessionFactory.openSession();
        Transaction transaction = null;

        transaction = session.beginTransaction();
        List developers = session.createQuery("FROM Developer").list();
        for (Developer developer : developers) {
            System.out.println(developer);
            System.out.println("\n================\n");
        }
        session.close();
    }

    public void updateDeveloper(int developerId, int experience) {
        Session session = sessionFactory.openSession();
        Transaction transaction = null;

        transaction = session.beginTransaction();
        Developer developer = (Developer) session.get(Developer.class, developerId);
        developer.setExperience(experience);
        session.update(developer);
        transaction.commit();
        session.close();
    }

    public void removeDeveloper(int developerId) {
        Session session = sessionFactory.openSession();
        Transaction transaction = null;
```

```
        transaction = session.beginTransaction();
        Developer developer = (Developer) session.get(Developer.class, developerId);
        session.delete(developer);
        transaction.commit();
        session.close();
    }
}
```

Если всё было сделано правильно, то в результате работы программы вы получите, примерно, следующий результат:

```
/usr/lib/jvm/java-8-oracle/bin/java -Didea.launcher.port=7536 -
Didea.launcher.bin.path=/home/proselyte/Programming/Soft/IntellijIdea/bin -
Dfile.encoding=UTF-8 -classpath /usr/lib/jvm/java-8-oracle/jre/lib/management-
agent.jar:/usr/lib/jvm/java-8-oracle/jre/lib/plugin.jar:/usr/lib/jvm/java-8-
oracle/jre/lib/rt.jar:/usr/lib/jvm/java-8-oracle/jre/lib/jsse.jar:/usr/lib/jvm/java-8-
oracle/jre/lib/charsets.jar:/usr/lib/jvm/java-8-
oracle/jre/lib/jce.jar:/usr/lib/jvm/java-8-
oracle/jre/lib/resources.jar:/usr/lib/jvm/java-8-
oracle/jre/lib/deploy.jar:/usr/lib/jvm/java-8-
oracle/jre/lib/jfxswt.jar:/usr/lib/jvm/java-8-
oracle/jre/lib/javaws.jar:/usr/lib/jvm/java-8-oracle/jre/lib/jfr.jar:/usr/lib/jvm/java-
8-oracle/jre/lib/ext/dnsns.jar:/usr/lib/jvm/java-8-
oracle/jre/lib/ext/sunpkcs11.jar:/usr/lib/jvm/java-8-
oracle/jre/lib/ext/sunec.jar:/usr/lib/jvm/java-8-
oracle/jre/lib/ext/sunjce_provider.jar:/usr/lib/jvm/java-8-
oracle/jre/lib/ext/jaccess.jar:/usr/lib/jvm/java-8-
oracle/jre/lib/ext/nashorn.jar:/usr/lib/jvm/java-8-
oracle/jre/lib/ext/localedata.jar:/usr/lib/jvm/java-8-
oracle/jre/lib/ext/zipfs.jar:/usr/lib/jvm/java-8-
oracle/jre/lib/ext/cldrdata.jar:/usr/lib/jvm/java-8-
oracle/jre/lib/ext/jfxrt.jar:/home/proselyte/Programming/IdeaProjects/ProselyteTutorial
s/Hibernate/target/classes:/home/proselyte/.m2/repository/org/springframework/spring-
core/4.1.1.RELEASE/spring-core-
4.1.1.RELEASE.jar:/home/proselyte/.m2/repository/commons-logging/commons-
logging/1.1.3/commons-logging-
1.1.3.jar:/home/proselyte/.m2/repository/org/springframework/spring-
web/4.1.1.RELEASE/spring-web-
4.1.1.RELEASE.jar:/home/proselyte/.m2/repository/org/springframework/spring-
aop/4.1.1.RELEASE/spring-aop-
4.1.1.RELEASE.jar:/home/proselyte/.m2/repository/aopalliance/aopalliance/1.0/aopallianc
e-1.0.jar:/home/proselyte/.m2/repository/org/springframework/spring-
beans/4.1.1.RELEASE/spring-beans-
4.1.1.RELEASE.jar:/home/proselyte/.m2/repository/org/springframework/spring-
context/4.1.1.RELEASE/spring-context-
4.1.1.RELEASE.jar:/home/proselyte/.m2/repository/javax/servlet/servlet-api/2.5/servlet-
api-2.5.jar:/home/proselyte/.m2/repository/org/springframework/spring-
webmvc/4.1.1.RELEASE/spring-webmvc-
4.1.1.RELEASE.jar:/home/proselyte/.m2/repository/org/springframework/spring-
expression/4.1.1.RELEASE/spring-expression-
4.1.1.RELEASE.jar:/home/proselyte/.m2/repository/org/springframework/integration/spring
-integration-file/4.2.1.RELEASE/spring-integration-file-
4.2.1.RELEASE.jar:/home/proselyte/.m2/repository/org/springframework/integration/spring
-integration-core/4.2.1.RELEASE/spring-integration-core-
4.2.1.RELEASE.jar:/home/proselyte/.m2/repository/org/springframework/spring-
messaging/4.2.2.RELEASE/spring-messaging-
4.2.2.RELEASE.jar:/home/proselyte/.m2/repository/org/springframework/retry/spring-
retry/1.1.2.RELEASE/spring-retry-
1.1.2.RELEASE.jar:/home/proselyte/.m2/repository/org/springframework/spring-
tx/4.2.2.RELEASE/spring-tx-4.2.2.RELEASE.jar:/home/proselyte/.m2/repository/commons-
```

```
io/commons-io/2.4/commons-io-
2.4.jar:/home/proselyte/.m2/repository/org/hibernate/hibernate-
core/5.1.0.Final/hibernate-core-
5.1.0.Final.jar:/home/proselyte/.m2/repository/org/jboss/logging/jboss-
logging/3.3.0.Final/jboss-logging-
3.3.0.Final.jar:/home/proselyte/.m2/repository/org/hibernate/javax/persistence/hibernat
e-jpa-2.1-api/1.0.0.Final/hibernate-jpa-2.1-api-
1.0.0.Final.jar:/home/proselyte/.m2/repository/org/javassist/javassist/3.20.0-
GA/javassist-3.20.0-GA.jar:/home/proselyte/.m2/repository/antlr/antlr/2.7.7/antlr-
2.7.7.jar:/home/proselyte/.m2/repository/org/apache/geronimo/specs/geronimo-
jta_1.1_spec/1.1.1/geronimo-jta_1.1_spec-
1.1.1.jar:/home/proselyte/.m2/repository/org/jboss/jandex/2.0.0.Final/jandex-
2.0.0.Final.jar:/home/proselyte/.m2/repository/com/fasterxml/classmate/1.3.0/classmate-
1.3.0.jar:/home/proselyte/.m2/repository/dom4j/dom4j/1.6.1/dom4j-
1.6.1.jar:/home/proselyte/.m2/repository/xml-apis/xml-apis/1.0.b2/xml-apis-
1.0.b2.jar:/home/proselyte/.m2/repository/org/hibernate/common/hibernate-commons-
annotations/5.0.1.Final/hibernate-commons-annotations-
5.0.1.Final.jar:/home/proselyte/.m2/repository/javassist/javassist/3.12.1.GA/javassist-
3.12.1.GA.jar:/home/proselyte/.m2/repository/mysql/mysql-connector-java/5.1.38/mysql-
connector-java-5.1.38.jar:/home/proselyte/Programming/Soft/IntellijIdea/lib/idea_rt.jar
com.intellij.rt.execution.application.AppMain
net.proselyte.hibernate.annotations.DeveloperRunner
Feb 22, 2016 9:34:01 PM org.hibernate.Version logVersion
INFO: HHH000412: Hibernate Core {5.1.0.Final}
Feb 22, 2016 9:34:01 PM org.hibernate.cfg.Environment
INFO: HHH000206: hibernate.properties not found
Feb 22, 2016 9:34:01 PM org.hibernate.cfg.Environment buildBytecodeProvider
INFO: HHH000021: Bytecode provider name : javassist
Feb 22, 2016 9:34:01 PM
org.hibernate.annotations.common.reflection.java.JavaReflectionManager
INFO: HCANN000001: Hibernate Commons Annotations {5.0.1.Final}
Feb 22, 2016 9:34:02 PM
org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl
configure
WARN: HHH10001002: Using Hibernate built-in connection pool (not for production use!)
Feb 22, 2016 9:34:02 PM
org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl
buildCreator
INFO: HHH10001005: using driver [com.mysql.jdbc.Driver] at URL
[jdbc:mysql://localhost/PROSELYTE_TUTORIAL]
Feb 22, 2016 9:34:02 PM
org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl
buildCreator
INFO: HHH10001001: Connection properties: {user=root, password=****}
Feb 22, 2016 9:34:02 PM
org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl
buildCreator
INFO: HHH10001003: Autocommit mode: false
Feb 22, 2016 9:34:02 PM
org.hibernate.engine.jdbc.connections.internal.PooledConnections
INFO: HHH000115: Hibernate connection pool size: 20 (min=1)
Mon Feb 22 21:34:03 EET 2016 WARN: Establishing SSL connection without server's
identity verification is not recommended. According to MySQL 5.5.45+, 5.6.26+ and
5.7.6+ requirements SSL connection must be established by default if explicit option
isn't set. For compliance with existing applications not using SSL the
verifyServerCertificate property is set to 'false'. You need either to explicitly
disable SSL by setting useSSL=false, or set useSSL=true and provide truststore for
server certificate verification.
Feb 22, 2016 9:34:03 PM org.hibernate.dialect.Dialect
INFO: HHH000400: Using dialect: org.hibernate.dialect.MySQLDialect
Adding Developer's records to the database
List of Developers:
Feb 22, 2016 9:34:04 PM org.hibernate.hql.internal.QueryTranslatorFactoryInitiator
initiateService
INFO: HHH000397: Using ASTQueryTranslatorFactory
```

```
Developer:
id: 1
First Name: Proselyte
Last Name: Developer
Specialty: Java Developer
Experience: 2


================

Developer:
id: 2
First Name: Some
Last Name: Developer
Specialty: C++ Developer
Experience: 2


================

Developer:
id: 3
First Name: Peter
Last Name: Team Lead
Specialty: Java Team Lead
Experience: 6


================

Removing 'Some Developer' and updating 'Proselyte Developer''s experience:
Final list of Developers:
Developer:
id: 1
First Name: Proselyte
Last Name: Developer
Specialty: Java Developer
Experience: 3


================

Developer:
id: 3
First Name: Peter
Last Name: Team Lead
Specialty: Java Team Lead
Experience: 6


================
```

Так будет выглядеть наша таблица **HIBERNATE_DEVELOPERS** в БД

```
+----+------------+-----------+----------------+------------+
| ID | FIRST_NAME | LAST_NAME | SPECIALTY      | EXPERIENCE |
+----+------------+-----------+----------------+------------+
|  1 | Proselyte  | Developer | Java Developer |          3 |
|  3 | Peter      | Team Lead | Java Team Lead |          6 |
+----+------------+-----------+----------------+------------+
```

Руководство по Hibernate. Язык запросов Hibernate (HQL).

HQL (Hibernate Query Language) – это объектно-ориентированный (далее – ОО) язык запросов, который крайне похож на SQL.

Отличие между HQL и SQL состоит в том, что SQL работает таблицами в базе данных (далее – БД) и их столбцами, а HQL – с сохраняемыми объектами (Persistent Objects) и их полями (атрибутами класса).

Hibernate транслирует HQL – запросы в понятные для БД SQL – запросы, которые и выполняют необходимые нам действия в БД.

Мы также имеем возможность использовать обычные SQL – запросы в Hibernate используя Native SQL, но использование HQL является более предпочтительным.

Давайте рассмотрим основные ключевые слова языка HQL:

FROM

Если мы хотим загрузить в память наши сохраняемые объекты, то мы будем использовать ключевое слово **FROM**. Вот пример его использования:

```
   Query query = session.createQuery("FROM Developer");
   List developers = query.list();
```

Developer – это POJO – класс Developer.java, который ассоциирован с таблицей в нашей БД.

INSERT
Мы используем ключевое слово **INSERT**, в том случае, если хотим добавить запись в таблицу нашей БД.

Вот пример использования этого ключевого слова:

```
   Query query =
            session.createQuery("INSERT INTO Developer (firstName, lastName, specialty,
experience)");
```

UPDATE
Ключевое слово UPDATE используется для обновления одного или нескольких полей объекта. Вот так это выглядит на практике:

```
   Query query =
      session.createQuery(UPDATE Developer SET experience =: experience WHERE id =:
developerId);
   query.setParameter("expericence", 3);
```

DELETE
Это ключевоеслово используется для удаления одного или нескольких объектов. Пример использования:

```
Query query = session.createQuery("DELETE FROM Developer WHERE id = :developerId");
query.setParameter("developerId", 1);
```

SELECT
Если мы хотим получить запись из таблицы нашей БД, то мы должны использоваьб ключевое слово **SELECT**. Пример использования:

```
Query query = session.createQuery("SELECT D.lastName FROM Developer D");
List developers = query.list();
```

AS
В предыдущем примере мы использовали запись формы Developer D. С использованием опционального ключевого слова **AS** это будет выглядеть так:

```
Query query = session.createQuery("FROM Developer AS D");
List developers = query.list();
```

WHERE
В том случае, если мы хотим получить объекты, которые соответствуют определённым параметрам, то мы должны использовать ключевое слово **WHERE**. На практике это выглядит следующим образом:

```
Query query = session.createQuery("FROM Developer D WHERE D.id = 1");
List developer = query.list();
```

ORDER BY
Для того, чтобы отсортировать список объектов, полученных в результате запроса, мы должны применить ключевое слово **ORDER BY**. Нам необходимо указать параметр, по которому список будет отсортирован и тип сортировки – по возрастанию (ASC) или по убыванию (DESC). В виде кода это выглядит так:

```
Query query =
     session.createQuery("FROM Developer D WHERE experience > 3 ORDER BY
D.experience DESC");
```

GROUP BY
С помощью ключевого слова **GROUP BY** мы можем группировать данные, полученные из БД по какому-либо признаку. Вот простой пример применения данного ключевого слова:

```
Query query = session.createQuery("SELECT MAX(D.experience), D.lastName, D.specialty
FROM Developer D GROUP BY D.lastName");
List developers = query.list();
```

Методы аггрегации
Язык запросов Hibernate (HQL) пожжерживает различные методы аггрегации, которые доступны и в SQL. HQL поддерживает следующие методы:

---

min(имя свойства)

Минимальное значение данного свойства.

---

max(имя свойства)

Максимальное значение данного свойства.

---

sum(имя свойства)

Сумма всех значений данного свойства.

---

avg(имя свойства)

Среднее арифметическое всех значений данного свойства

---

count(имя свойства)

Какое количество раз данное свойство встречается в результате.

Руководство по Hibernate. Запросы с использованием Criteria

Hibernate поддерживает различные способы манипулирования объектами и транслирования их в таблицы баз данных (далее – БД). Одним из таких способов является **Criteria API**, который позволяет нам создавать запросы с критериями, программным методом.

Для создания **Criteria** используется метод createCriteria() интерфейса Session. Этот метод возвращает экземпляр сохраняемого класса (persistent class) в результате его выполнения.

Вот как это выглядит на практике:

```
Criteria criteria = session.createCriteria(Developer.class);
List developers = criteria.list();
```

Criteria имеет два важных метода:

---

public Criteria setFirstResult(int firstResult)

Этот метод указывает первый ряд нашего результата, который начинается с 0.

---

public Criteria setMaxResults(int maxResults)

Этот методограничивает максимальное количество оъектов, которое Hibernate сможет получить в результате запроса.

Для понимания того, как это работает на практке рассмотрим пример простого приложения.

Пример:

Шаг 1. Создадим таблицу HIBERNATE_DEVELOPERS в нашей БД.

```
CREATE TABLE HIBERNATE_DEVELOPERS (
id INT NOT NULL auto_increment,
FIRST_NAME VARCHAR(50) default NULL,
LAST_NAME VARCHAR(50) default NULL,
SPECIALTY VARCHAR(50) default NULL,
EXPERIENCE INT default NULL,
SALARY INT default NULL,
PRIMARY KEY (id)
);
```

Шаг 2. Создадим POJO – класс

Developer.java

```
package net.proselyte.hibernate.criteria;

public class Developer {
    private int id;
    private String firstName;
    private String lastName;
    private String specialty;
    private int experience;
    private int salary;

    /**
     * Default Constructor
     */
    public Developer() {
    }

    /**
     * Plain constructor
```

```
     */
    public Developer(String firstName, String lastName, String specialty, int
experience, int salary) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.specialty = specialty;
        this.experience = experience;
        this.salary = salary;
    }

    /**
     * Getters and Setters
     */
    public int getId() {
        return id;
    }

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

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getSpecialty() {
        return specialty;
    }

    public void setSpecialty(String specialty) {
        this.specialty = specialty;
    }

    public int getExperience() {
        return experience;
    }

    public void setExperience(int experience) {
        this.experience = experience;
    }

    public int getSalary() {
        return salary;
    }

    public void setSalary(int salary) {
        this.salary = salary;
    }

    /**
     * toString method (optional)
     */
    @Override
```

```
    public String toString() {
        return "id: " + id +
                "\nFirst Name: " + firstName +
                "\nLast Name: " + lastName +
                "\nSpecialty: " + specialty +
                "\nExperience: " + experience +
                "\nSalary: " + salary + "\n";
    }
}
```

Шаг 3. Создаём конфигурационные файлы

hibernate.cfg.xml

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration SYSTEM
        "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 PROSELYTE_TUTORIAL is the database name -->
        <property name="hibernate.connection.url">
            jdbc:mysql://localhost/ИМЯ_ВАШЕЙ_БД
        </property>
        <property name="hibernate.connection.username">
            ВАШЕ_ИМЯ_ПОЛЬЗОВАТЕЛЯ
        </property>
        <property name="hibernate.connection.password">
            ВАШ_ПАРОЛЬ
        </property>

        <!-- List of XML mapping files -->
        <mapping resource="Developer.hbm.xml"/>

    </session-factory>
</hibernate-configuration>
```

Developer.hbm.xml

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

<hibernate-mapping>
    <class name="net.proselyte.hibernate.criteria.Developer"
table="HIBERNATE_DEVELOPERS">
        <meta attribute="class-description">
            This class contains developer details.
        </meta>
        <id name="id" type="int" column="id">
```

```
            <generator class="native"/>
        </id>
        <property name="firstName" column="FIRST_NAME" type="string"/>
        <property name="lastName" column="LAST_NAME" type="string"/>
        <property name="specialty" column="SPECIALTY" type="string"/>
        <property name="experience" column="EXPERIENCE" type="int"/>
        <property name="salary" column="SALARY" type="int"/>
    </class>

</hibernate-mapping>
```

Шаг 4. Создаём класс DeveloperRunner.java

DeveloperRunner.java

```
package net.proselyte.hibernate.criteria;

import org.hibernate.Criteria;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
import org.hibernate.criterion.Projections;
import org.hibernate.criterion.Restrictions;

import java.util.List;

public class DeveloperRunner {
    private static SessionFactory sessionFactory;

    public static void main(String[] args) {
        sessionFactory = new Configuration().configure().buildSessionFactory();
        DeveloperRunner developerRunner = new DeveloperRunner();

        System.out.println("Adding developer's records to the database...");
        Integer developerId1 = developerRunner.addDeveloper("Proselyte", "Developer",
"Java Developer", 3, 2000);
        Integer developerId2 = developerRunner.addDeveloper("First", "Developer", "C++
Developer", 10, 2000);
        Integer developerId3 = developerRunner.addDeveloper("Second", "Developer", "C#
Developer", 5, 2000);
        Integer developerId4 = developerRunner.addDeveloper("Third", "Developer", "PHP
Developer", 1, 2000);

        System.out.println("List of Developers with experience more than 3 years:");
        developerRunner.listDevelopersOverThreeYears();

        System.out.println("Total Salary of all Developers:");
        developerRunner.totalSalary();
        sessionFactory.close();
    }

    public Integer addDeveloper(String firstName, String lastName, String specialty,
int experience, int salary) {
        Session session = sessionFactory.openSession();
        Transaction transaction = null;
        Integer developerId = null;

        transaction = session.beginTransaction();
        Developer developer = new Developer(firstName, lastName, specialty, experience,
salary);
        developerId = (Integer) session.save(developer);
```

```
        transaction.commit();
        session.close();
        return developerId;
    }

    public void listDevelopersOverThreeYears() {
        Session session = sessionFactory.openSession();
        Transaction transaction = null;

        transaction = session.beginTransaction();
        Criteria criteria = session.createCriteria(Developer.class);
        criteria.add(Restrictions.gt("experience", 3));
        List developers = criteria.list();

        for (Developer developer : developers) {
            System.out.println("======================");
            System.out.println(developer);
            System.out.println("======================");
        }
        transaction.commit();
        session.close();
    }

    public void totalSalary() {
        Session session  = sessionFactory.openSession();
        Transaction transaction = null;

        transaction = session.beginTransaction();
        Criteria criteria = session.createCriteria(Developer.class);
        criteria.setProjection(Projections.sum("salary"));

        List totalSalary = criteria.list();
        System.out.println("Total salary of all developers: " + totalSalary.get(0));
        transaction.commit();
        session.close();
    }


}
```

Если всё было сделано правильно, то в результате работы программы мы получим, примерно, следующий результат:

```
List of Developers with experience more than 3 years:
======================
id: 18
First Name: First
Last Name: Developer
Specialty: C++ Developer
Experience: 10
Salary: 2000

======================
======================
id: 19
First Name: Second
Last Name: Developer
Specialty: C# Developer
Experience: 5
Salary: 2000

======================
Total Salary of all Developers:
Total salary of all developers: 8000
```

Руководство по HIbernate. Нативный SQL.

Для того, чтобы формировать запросы для базы данных (далее – БД), при этом используя все возможности БД, мы можем использовать нативный SQL.

Таким образом наше приложение создаст нативный SQL – запрос, используя метод **createSQLQuery()** интерфейса **Session**, который выглядит следующим образом:

```
public SQLQuery createSQLQuery (String sqlString) throws HibernateException;
```

После того, как мы передаём методу **createSQLQuery()** строку (String), содержащую SQL – запрос, мы можем связать результат этого запроса с сохраняемым объектом (persistent object).

Для понимания того, как это работает на практике, рассмотрим пример простого приложения.

Пример:
Исходный код проекта можно скачать по ***ЭТОЙ ССЫЛКЕ***.

Шаг 1. Создадим таблицу в нашей БД

HIBRNATE_DEVELOPERS

```
CREATE TABLE HIBERNATE_DEVELOPERS (
id INT NOT NULL auto_increment,
FIRST_NAME VARCHAR(50) default NULL,
LAST_NAME VARCHAR(50) default NULL,
SPECIALTY VARCHAR(50) default NULL,
EXPERIENCE INT default NULL,
SALARY INT default NULL,
PRIMARY KEY (id)
);
```

Шаг 2. Создадим POJO – класс

Developer.java

```
package net.proselyte.hibernate.criteria;

public class Developer {
    private int id;
    private String firstName;
    private String lastName;
    private String specialty;
    private int experience;
    private int salary;

    /**
     * Default Constructor
     */
    public Developer() {
    }
```

```
    /**
     * Plain constructor
     */
    public Developer(String firstName, String lastName, String specialty, int
experience, int salary) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.specialty = specialty;
        this.experience = experience;
        this.salary = salary;
    }

    /**
     * Getters and Setters
     */
    public int getId() {
        return id;
    }

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

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getSpecialty() {
        return specialty;
    }

    public void setSpecialty(String specialty) {
        this.specialty = specialty;
    }

    public int getExperience() {
        return experience;
    }

    public void setExperience(int experience) {
        this.experience = experience;
    }

    public int getSalary() {
        return salary;
    }

    public void setSalary(int salary) {
        this.salary = salary;
    }

    /**
```

```
     * toString method (optional)
     */
    @Override
    public String toString() {
        return "id: " + id +
                "\nFirst Name: " + firstName +
                "\nLast Name: " + lastName +
                "\nSpecialty: " + specialty +
                "\nExperience: " + experience +
                "\nSalary: " + salary + "\n";
    }
}
```

Шаг 3. Создаём конфигурационные файлы

hibernate.cfg.xml

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration SYSTEM
        "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 PROSELYTE_TUTORIAL is the database name -->
        <property name="hibernate.connection.url">
            jdbc:mysql://localhost/ИМЯ_ВАШЕЙ_БД
        </property>
        <property name="hibernate.connection.username">
            ВАШЕ_ИМЯ_ПОЛЬЗОВАТЕЛЯ
        </property>
        <property name="hibernate.connection.password">
            ВАШ_ПАРОЛЬ
        </property>

        <!-- List of XML mapping files -->
        <mapping resource="Developer.hbm.xml"/>

    </session-factory>
</hibernate-configuration>
```

Developer.hbm.xml

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

<hibernate-mapping>
    <class name="net.proselyte.hibernate.nativesql.Developer"
table="HIBERNATE_DEVELOPERS">
        <meta attribute="class-description">
```

```
            This class contains developer details.
        </meta>
        <id name="id" type="int" column="id">
            <generator class="native"/>
        </id>
        <property name="firstName" column="FIRST_NAME" type="string"/>
        <property name="lastName" column="LAST_NAME" type="string"/>
        <property name="specialty" column="SPECIALTY" type="string"/>
        <property name="experience" column="EXPERIENCE" type="int"/>
        <property name="salary" column="SALARY" type="int"/>
    </class>

</hibernate-mapping>
```

Шаг 4. Создаём класс DeveloperRunner.java

DeveloperRunner.java

```
package net.proselyte.hibernate.nativesql;

import org.hibernate.*;
import org.hibernate.cfg.Configuration;
import org.hibernate.criterion.Projections;

import java.util.List;
import java.util.Map;

public class DeveloperRunner {
    private static SessionFactory sessionFactory;

    public static void main(String[] args) {
        sessionFactory = new Configuration().configure().buildSessionFactory();
        DeveloperRunner developerRunner = new DeveloperRunner();

        System.out.println("Adding developer's records to the database...");
        Integer developerId1 = developerRunner.addDeveloper("Proselyte", "Developer",
"Java Developer", 3, 2000);
        Integer developerId2 = developerRunner.addDeveloper("First", "Developer", "C++
Developer", 10, 5000);
        Integer developerId3 = developerRunner.addDeveloper("Second", "Developer", "C#
Developer", 5, 4000);
        Integer developerId4 = developerRunner.addDeveloper("Third", "Developer", "PHP
Developer", 1, 1000);

        System.out.println("List of Developers using Entity Query:");
        developerRunner.listDevelopers();

        System.out.println("List of Developers using Scalar Query:");
        developerRunner.listDevelopersScalar();
        sessionFactory.close();
    }

    public Integer addDeveloper(String firstName, String lastName, String specialty,
int experience, int salary) {
        Session session = sessionFactory.openSession();
        Transaction transaction = null;
        Integer developerId = null;

        transaction = session.beginTransaction();
        Developer developer = new Developer(firstName, lastName, specialty, experience,
salary);
        developerId = (Integer) session.save(developer);
        transaction.commit();
```

```
        session.close();
        return developerId;
    }

    public void listDevelopers() {
        Session session = sessionFactory.openSession();
        Transaction transaction = null;

        transaction = session.beginTransaction();
        SQLQuery sqlQuery = session.createSQLQuery("SELECT * FROM
HIBERNATE_DEVELOPERS");
        sqlQuery.addEntity(Developer.class);
        List developers = sqlQuery.list();

        for (Developer developer : developers) {
            System.out.println("======================");
            System.out.println(developer);
            System.out.println("======================");
        }
        transaction.commit();
        session.close();
    }

    public void listDevelopersScalar() {
        Session session = sessionFactory.openSession();
        Transaction transaction = null;

        transaction = session.beginTransaction();
        SQLQuery sqlQuery = session.createSQLQuery("SELECT * FROM
HIBERNATE_DEVELOPERS");
        sqlQuery.setResultTransformer(Criteria.ALIAS_TO_ENTITY_MAP);
        List developers = sqlQuery.list();
        for (Object developer : developers) {
            Map row = (Map) developer;
            System.out.println("======================");
            System.out.println("id: " + row.get("id"));
            System.out.println("First Name: " + row.get("FIRST_NAME"));
            System.out.println("Last Name: " + row.get("LAST_NAME"));
            System.out.println("Specialty: " + row.get("SPECIALTY"));
            System.out.println("Experience: " + row.get("EXPERIENCE"));
            System.out.println("Salary: " + row.get("SALARY"));
            System.out.println("======================");
        }
        transaction.commit();
        session.close();
    }

    public void totalSalary() {
        Session session = sessionFactory.openSession();
        Transaction transaction = null;

        transaction = session.beginTransaction();
        Criteria criteria = session.createCriteria(Developer.class);
        criteria.setProjection(Projections.sum("salary"));

        List totalSalary = criteria.list();
        System.out.println("Total salary of all developers: " + totalSalary.get(0));
        transaction.commit();
        session.close();
    }


}
```

Если всё было сделано правильно, в результате работы программы мы получим, примерно, следующий результат:

```
List of Developers using Entity Query:
======================
id: 69
First Name: Proselyte
Last Name: Developer
Specialty: Java Developer
Experience: 3
Salary: 2000

======================
======================
id: 70
First Name: First
Last Name: Developer
Specialty: C++ Developer
Experience: 10
Salary: 5000

======================
======================
id: 71
First Name: Second
Last Name: Developer
Specialty: C# Developer
Experience: 5
Salary: 4000

======================
======================
id: 72
First Name: Third
Last Name: Developer
Specialty: PHP Developer
Experience: 1
Salary: 1000

======================
List of Developers using Scalar Query:
======================
id: 69
First Name: Proselyte
Last Name: Developer
Specialty: Java Developer
Experience: 3
Salary: 2000
======================
======================
id: 70
First Name: First
Last Name: Developer
Specialty: C++ Developer
Experience: 10
Salary: 5000
======================
======================
id: 71
First Name: Second
Last Name: Developer
Specialty: C# Developer
Experience: 5
```

```
Salary: 4000
======================
======================
id: 72
First Name: Third
Last Name: Developer
Specialty: PHP Developer
Experience: 1
Salary: 1000
======================
Feb 23, 2016 9:44:06 PM
org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl stop
INFO: HHH10001008: Cleaning up connection pool
[jdbc:mysql://localhost/PROSELYTE_TUTORIAL]
```

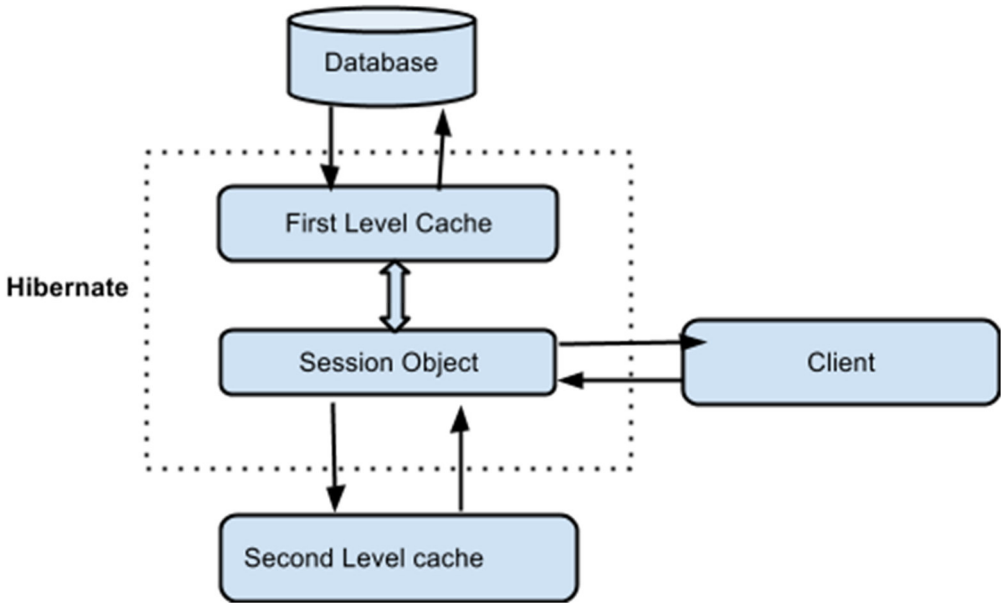
Наша таблица HIBERNATE_DEVELOPERS будет иметь такой вид:

```
+----+------------+-----------+----------------+------------+--------+
| id | FIRST_NAME | LAST_NAME | SPECIALTY      | EXPERIENCE | SALARY |
+----+------------+-----------+----------------+------------+--------+
| 65 | Proselyte  | Developer | Java Developer |          3 |   2000 |
| 66 | First      | Developer | C++ Developer  |         10 |   5000 |
| 67 | Second     | Developer | C# Developer   |          5 |   4000 |
| 68 | Third      | Developer | PHP Developer  |          1 |   1000 |
+----+------------+-----------+----------------+------------+--------+
```

Руководство по Hibernate. Кеширование.

Кеширование является одним из способов оптимизации работы приложения, ключевой задачей которого является уменьшить количество прямых обращений к базе данных (далее – БД).

Если речь идёт о Hibernate, то схематически кэширование можно представить в виде следующего рисунка:



Кэш первого уровня (First Level Cache)

Кэш первого уровня – это кэш **Сессии (Session)**, который является обязательным. Через него проходят все запросы. Перед тем, как отправить объект в БД, сессия хранит объект за счёт своих ресурсов.

В том случае, если мы выполняем несколько обновлений объекта, Hibernate старается отсрочить (насколько это возможно) обновление для того, чтобы сократить количество выполненных запросов. Если мы закроем сессию, то все объекты, находящиеся в кэше теряются, а далее – либо сохраняются, либо обновляются.

Кэш второго уровня (Second level Cache)

Кэш второго уровня является необязательным (опциональным) и изначально Hibernate будет искать необходимый объект в кэше первого уровня. В основном, кэширование второго уровня отвечает за кэширование объектов

Кэш запросов (Query Cache)

В Hibernate предусмотрен кэш для запросов, и он интегрирован с кэшем второго уровня. Это требует двух дополнительных физических мест для хранения кэшированных запросов и временных меток для обновления таблицы БД. Этот вид кэширования эффективен только для часто используемых запросов с одинаковыми параметрами.

Рассмотрим, как это выглядит на практике.

Developer.hbm.xml

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

<hibernate-mapping>
    <class name="net.proselyte.hibernate.nativesql.Developer"
table="HIBERNATE_DEVELOPERS">
        <meta attribute="class-description">
            This class contains developer details.
        </meta>
        <id name="id" type="int" column="id">
            <generator class="native"/>
        </id>
        <property name="firstName" column="FIRST_NAME" type="string"/>
        <property name="lastName" column="LAST_NAME" type="string"/>
        <property name="specialty" column="SPECIALTY" type="string"/>
        <property name="experience" column="EXPERIENCE" type="int"/>
        <property name="salary" column="SALARY" type="int"/>
    </class>

</hibernate-mapping>
```

hibenrate.cfg.xml

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration SYSTEM
```

```
        "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 PROSELYTE_TUTORIAL is the database name -->
        <property name="hibernate.connection.url">
            jdbc:mysql://localhost/ИМЯ_ВАШЕЙ_БАЗЫ_ДАННЫХ
        </property>
        <property name="hibernate.connection.username">
            ВАШЕ_ИМЯ_ПОЛЬЗОВАТЕЛЯ
        </property>
        <property name="hibernate.connection.password">
            ВАШ_ПАРОЛЬ
        </property>

        <!-- List of XML mapping files -->
        <mapping resource="Developer.hbm.xml"/>

    </session-factory>
</hibernate-configuration>
```

Два предыдущих фала нам уже знакомы, а теперь нам необходимо содать фалй ehcache.xml

ehcache.xml

```
<diskStore path="java.io.tmpdir"/>
<defaultCache
maxElementsInMemory="500"
eternal="false"
timeToIdleSeconds="60"
timeToLiveSeconds="60"
overflowToDisk="true"
/>

<cache name="Developer"
maxElementsInMemory="200"
eternal="true"
timeToIdleSeconds="0"
timeToLiveSeconds="0"
overflowToDisk="false"
/>
```

Для того, что кэширование стало доступным для нашего приложения мы должны активировать его следующим образом:

```
   Session session = sessionFactory.openSession();
   Query query = session.createQuery("FROM HIBERNATE_DEVELOPERS");
   query.setCacheable(true);
   query.setCacheRegion("developer");
   List developers = query.list();
   sessionFactory.close();
```

Руководство по Hibernate. Обработка “пакетов”.

Давайте рассмотрим ситуацию, в которой нам необходимо выполнить 100,000 записей в таблицу базы данных (далее – БД).

Рассмотрим примитивный способ:

```
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();

for(int i = 0; i < 100_000; i++){
   Developer developer = new Developer(/*Some parameters*/);
   session.save(developer);
}
transaction.commit();
session.close();
```

На первый взгляд кажется, что мы получим 100_000 записей в нашу БД, но на практике мы получим OutOfMemoryException примерно в тот момент, когда попытаемся выполнить 50_000 – тысячную запись. Это вызвано тем, что Hibernate кэширует все сохраняемые объекты в кэш сессии.

Как же нам решить данную проблему?
Для этого нам необходимо использовать **обработку пакетов (batch processing)**.
Например, мы говорим Hibernate, что хотим вставлять каждые 50 объектов, как единый пакет. Для этого нам необходимо установить **hibernate.jdbc.atch_size на 50** и написать, примерно, такой кусок кода:

```
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();
for(int i = 0; i < 100000; i++){
   Developer developer = new Developer(/*Some parameters*/);
   session.save(developer);
   if(i % 50 == 0){
      session.flush();
      session.clear();
   }
}
transaction.commit();
session.close();
```

Для понимания того, как это работает на практике – рассмотрим пример небольшого приложения.

Пример:
Исходный код проекта можно скачать по ***ЭТОЙ ССЫЛКЕ***.

Шаг 1. Создаём таблицу в нашей БД
HIBRNATE_DEVELOPERS

```
CREATE TABLE HIBERNATE_DEVELOPERS (
id INT NOT NULL auto_increment,
```

```
   FIRST_NAME VARCHAR(50) default NULL,
   LAST_NAME VARCHAR(50) default NULL,
   SPECIALTY VARCHAR(50) default NULL,
   EXPERIENCE INT default NULL,
   SALARY INT default NULL,
   PRIMARY KEY (id)
   );
```

Шаг 2 . Создадим POJO – класс
Developer.java

```
package net.proselyte.hibernate.batch;

public class Developer {
    private int id;
    private String firstName;
    private String lastName;
    private String specialty;
    private int experience;
    private int salary;

    /**
     * Default Constructor
     */
    public Developer() {
    }

    /**
     * Plain constructor
     */
    public Developer(String firstName, String lastName, String specialty, int
experience, int salary) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.specialty = specialty;
        this.experience = experience;
        this.salary = salary;
    }

    /**
     * Getters and Setters
     */
    public int getId() {
        return id;
    }

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

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }
```

```
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getSpecialty() {
        return specialty;
    }

    public void setSpecialty(String specialty) {
        this.specialty = specialty;
    }

    public int getExperience() {
        return experience;
    }

    public void setExperience(int experience) {
        this.experience = experience;
    }

    public int getSalary() {
        return salary;
    }

    public void setSalary(int salary) {
        this.salary = salary;
    }

    /**
     * toString method (optional)
     */
    @Override
    public String toString() {
        return "id: " + id +
                "\nFirst Name: " + firstName +
                "\nLast Name: " + lastName +
                "\nSpecialty: " + specialty +
                "\nExperience: " + experience +
                "\nSalary: " + salary + "\n";
    }
}
```

Шаг 3. Создаём конфигурационные файлы

hibernate.cfg.xml

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration SYSTEM
        "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 PROSELYTE_TUTORIAL is the database name -->
        <property name="hibernate.connection.url">
```

```
            jdbc:mysql://localhost/ИМЯ_ВАШЕЙ_БАЗЫ_ДАННЫХ
        </property>
        <property name="hibernate.connection.username">
            ВАШЕ_ИМЯ_ПОЛЬЗОВАТЕЛЯ
        </property>
        <property name="hibernate.connection.password">
            ВАШ_ПАРОЛЬ
        </property>

        <!--  Specifying batch size -->
        <property name="hibernate.jdbc.batch_size">
            50
        </property>

        <!-- List of XML mapping files -->
        <mapping resource="Developer.hbm.xml"/>

    </session-factory>
</hibernate-configuration>
```

Developer.hbm.xml

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

<hibernate-mapping>
    <class name="net.proselyte.hibernate.batch.Developer" table="HIBERNATE_DEVELOPERS">
        <meta attribute="class-description">
            This class contains developer details.
        </meta>
        <id name="id" type="int" column="id">
            <generator class="native"/>
        </id>
        <property name="firstName" column="FIRST_NAME" type="string"/>
        <property name="lastName" column="LAST_NAME" type="string"/>
        <property name="specialty" column="SPECIALTY" type="string"/>
        <property name="experience" column="EXPERIENCE" type="int"/>
        <property name="salary" column="SALARY" type="int"/>
    </class>

</hibernate-mapping>
```

Шаг 4. Создаём класс DeveloperRunner.java
DeveloperRunner.java

```
package net.proselyte.hibernate.batch;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;

public class DeveloperRunner {
    private static SessionFactory sessionFactory;

    public static void main(String[] args) {
        sessionFactory = new Configuration().configure().buildSessionFactory();
```

```
        DeveloperRunner developerRunner = new DeveloperRunner();

        System.out.println("Adding 100,000 developer's records to the database...");
        developerRunner.addDevelopers();
        System.out.println("100,000 developer's records successfully added to the
database...");
        sessionFactory.close();
    }

    public void addDevelopers() {
        Session session = sessionFactory.openSession();
        Transaction transaction = null;
        Integer developerId = null;

        transaction = session.beginTransaction();

        for (int i = 0; i < 100_000; i++) {
            String firstName = "First Name " + i;
            String lastName = "Last Name " + i;
            String specialty = "Specialty " + i;
            int experience = i;
            int salary = i * 10;
            Developer developer = new Developer(firstName, lastName, specialty,
experience, salary);
            session.save(developer);
            if (i % 50 == 0) {
                session.flush();
                session.clear();
            }
        }
        transaction.commit();
        session.close();
    }
}
```

Примерно, вот так будет выглядеть наша таблица HIBERNATE_DEVELOPERS

id	FIRST_NAME	LAST_NAME	SPECIALTY	EXPERIENCE	SALARY
100061	First Name 99988	Last Name 99988	Specialty 99988	99988	999880
100062	First Name 99989	Last Name 99989	Specialty 99989	99989	999890
100063	First Name 99990	Last Name 99990	Specialty 99990	99990	999900
100064	First Name 99991	Last Name 99991	Specialty 99991	99991	999910
100065	First Name 99992	Last Name 99992	Specialty 99992	99992	999920
100066	First Name 99993	Last Name 99993	Specialty 99993	99993	999930
100067	First Name 99994	Last Name 99994	Specialty 99994	99994	999940
100068	First Name 99995	Last Name 99995	Specialty 99995	99995	999950
100069	First Name 99996	Last Name 99996	Specialty 99996	99996	999960
100070	First Name 99997	Last Name 99997	Specialty 99997	99997	999970
100071	First Name 99998	Last Name 99998	Specialty 99998	99998	999980
100072	First Name 99999	Last Name 99999	Specialty 99999	99999	999990