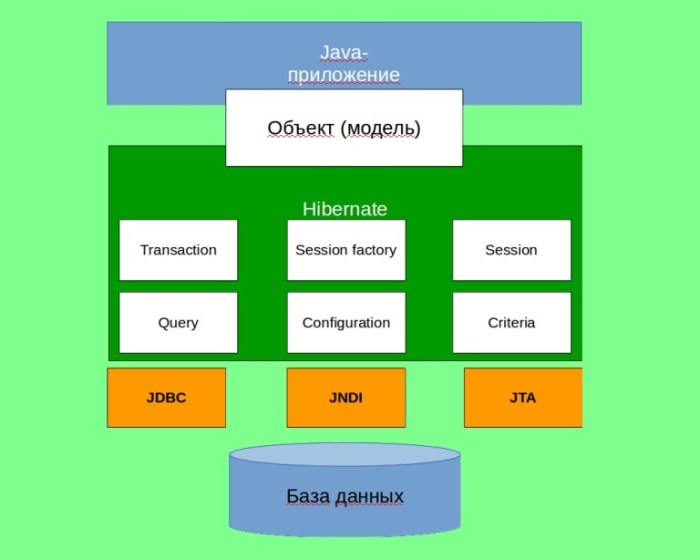
**Hibernate guide:**

**https://proselyte.net/tutorials/hibernate-tutorial/**

Hibernate – это ORM фреймворк для Java с открытым исходным кодом. Hibernate создаёт связь между таблицами в базе данных (далее – БД) и Java-классами и наоборот.

**Архитектура**



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 эземпляре, 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, можно 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 объекто-ориентированного запроса для получения объектов.

**Конфигурирование**

Для корректной работы, мы должны передать Hibernate подробную информацию, которая связывает наши Java-классы c 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**.

Ключевые свойства, которые должны быть настроены в типичном приложении:

**hibernate.dialect**

Указывает HIebrnate диалект БД. Hibernate, в своб 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.

Пример файла **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>

<property name="hibernate.connection.url"> jdbc:mysql://localhost/database </property>

<property name="hibernate.connection.username"> root </property>

<property name="hibernate.connection.password"> password </property>

<!-- List of XML mapping files -->

<mapping resource="Developer.hbm.xml"/>

</session-factory>

</hibernate-configuration>

# Сессии:

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 ни 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 можем 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.

Пример транзакции:

Transaction transaction = null;

try(Session session = sessionFactory.openSession()){

transaction = session.beginTransaction()

/\*\*

\* Here we make some work.

\* \*/

transaction.commit();

} catch(Exception e){

if(transaction != null){

transaction.rollback();

}

e.printStackTrace();

}

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, осовождает 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)**

Созраняет или обновляет указанный экземпляр.

**Serializable save(Object object)**

Сохраянет экземпляр, предварительно назначив сгенрированный 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 созранённый 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.

# 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 ключём (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.

Пример POJO класса:

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 +

'}';

}

}

# 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.

# Для примера, у нас есть POJO класс “Developer” и таблица “HIBERNATE\_DEVELOPER”, 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

<?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>

* **<hibernate-mapping>**Это ключевой тег , который должен быть в каждом XML – фалйе для связывания (mapping). Внутри этого тега мы и конфигурируем наши связи.
* **<class>**Тег <class> используется для того, чтобы указать связь между POJO – классом 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.

# Виды связей:

# Связи в ORM деляся на 3 гурппы:

1. **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**

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

**java.util.Collection**

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

**java.util.Set**

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

[**java.util.SortedSet**](https://proselyte.net/tutorials/hibernate-tutorial/mapping-types/sortedset-mapping-example/)

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**](https://proselyte.net/tutorials/hibernate-tutorial/mapping-types/map-mapping-example/)

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

[**java.util.SortedMap**](https://proselyte.net/tutorials/hibernate-tutorial/mapping-types/sortedset-mapping-example-2/)

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**

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

**One-to-One**

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

**One-to-Many**

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

**Many-to-Many**

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

# [****Связывание компонентов****](https://proselyte.net/tutorials/hibernate-tutorial/mapping-types/component-mapping-example/)

# 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).

# Аннотации:

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 аттрибуты, с помощью которых мы можем указать *имя таблицы, каталог, БД и уникальность столбцов в таблец 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 паметры, как **strategy** и **generator**.

**@Column**

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

# Пример начала POJO класса с аннотациями:

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;

# Язык запросов Hibernate (HQL):

HQL (Hibernate Query Language) – это объекто-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.

# 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 использовать 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 определенным 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 агрегации**

Язык запросов Hibernate (HQL) поддерживает 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.

# 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 сохряаняемого класса (persistent class) в результате его выполнения.

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 объектов, которое Hibernate сможет получить в результате запроса.

# 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:

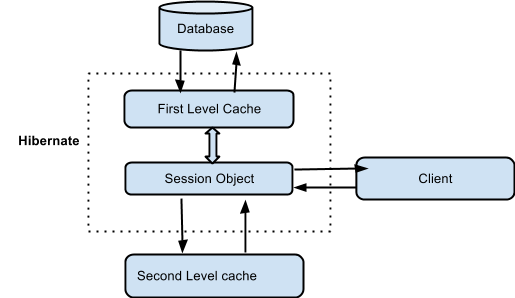
public SQLQuery createSQLQuery (String sqlString) throws HibernateException;

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

Кэширование:

Кэширование 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.

Пример создания кэша (имя файла **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();

**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 < 100\_000; i++){

Developer developer = new Developer(/\*Some parameters\*/);

session.save(developer);

if(i % 50 == 0){

session.flush();

session.clear();

}

}

transaction.commit();

session.close();

Пример метода который добавляет 100 000 записей:

void addDevelopers() {

Session session = sessionFactory.openSession();

Transaction 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();

}