<http://hibernate-refdoc.3141.ru/> - если нужно **ВСЕ**

**Hello, Hibernate (без JPA)**

[**Май 31, 2016**](https://easyjava.ru/data/hibernate/hello-hibernate-bez-jpa/)

Подготовка, Настройка, Использование - 2

Mapping сущностей - 6

Управление сущностями +аннотации - 9

Связанные сущности в Hibernate - 14

HQL, SQL, JPA Criteria – 27, 32

Первичные ключи - 37

Пользовательские типы - 43

Кэширование - 48

Транзакции и блокировки - 51

ENVERS: автоматический аудит изменений - 54

События и Интерцепторы - 58

Как склонировать объект в Hibernate - 68

Использование пулов соединений - 68

Multitenancy стратегии построения БД:

- Hibernate Database multitenancy - 70

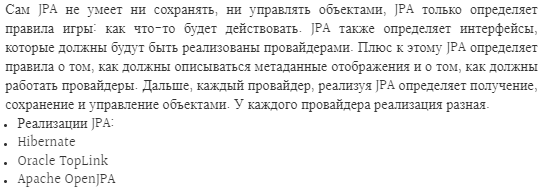
- Hibernate Schema Multitenancy - 76

- Filters и Discrimitator Multitenancy - 81

Совместимость с Flyway - 83

[JDBC](http://easyjava.ru/data/jdbc/) предоставляет вполне достаточный интерфейс для работы с базами данных в Java. Однако этот интерфейс весьма многословен и довольно неудобен и даже [Spring JDBC](http://easyjava.ru/spring/spring-data-access/) не делает его сильно лучше. По сути дела проблема в том, что реляционные базы данных работают с таблицами и отношениями между ними, в то время как в Java работают с объектами и их иерархиями. Поэтому приходится для каждого объекта или таблицы писать класс отображения одного в другое. Этот процесс называется ORM — object-relational mapping (объектно-реляционное отображение). И, к счастью, существуют готовые ORM решения, которые сами переводят данные из одного вида в другой и обратно.





Т.е. JPA это набор правил которым должны соответствовать ORM.

[Hibernate](https://easyjava.ru/tag/hibernate/) — один из старейших и уж точное наиболее распространённый ORM фреймворк в мире Java. Он может быть использован в качестве одной из [JPA реализаций](http://easyjava.ru/data/jpa/), либо с использованием его собственного API, которое, с одной стороны, сильно напоминает JPA, с другой стороны предоставляет больше возможностей и гибкости, чем строго регламентированный JPA.

**Подготовка**

В пустой maven проект добавим встраиваемую базу [H2](https://easyjava.ru/tag/h2/) и артефакты Hibernate:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | <dependency>    <groupId>com.h2database</groupId>    <artifactId>h2</artifactId>    <version>1.4.190</version>  </dependency>    <dependency>    <groupId>org.hibernate</groupId>    <artifactId>hibernate-core</artifactId>    <version>5.1.0.Final</version>  </dependency>    <dependency>    <groupId>org.hibernate</groupId>    <artifactId>hibernate-java8</artifactId>    <version>5.1.0.Final</version>  </dependency> |

*hibernate-core*это, собственно, сам Hibernate, API и его реализация, а hibernate-java8 добавляет поддержку новых типов данных (в основном даты/времени), появивишихся в java8.

**Entity**

Каждый заполненный объект класса, который сохранен в базе данных, называется entity(сущность). При этом на такой класс налагаются следующие требования:

* Наличие публично доступного конструктора без аргументов
* Класс, его методы и сохраняемые поля не должны быть final
* Если объект Entity класса будет передаваться по значению как отдельный объект (detached object), например через удаленный интерфейс (through a remote interface), он так же должен реализовывать Serializable интерфейс.
* Сохраняемые поля должны быть доступны только с использованием методов класса.

Эти требования в точности соответствуют [требованиям JPA](http://easyjava.ru/data/jpa/hello-jpa-i-hibernate/).

Каждый сохраняемый класс помечается аннотацией @Entity, говорящей Hibernate, что этот класс является сущностью. Помимо того, в каждом классе, помеченном @Entity должно быть поле, имеющее аннотацию @Id, говорящее Hibernate, что это поле может быть использовано как первичный ключ в базе данных и что по значению этого поля Hibernate может отличать один объект от другого. Честно говоря, полей с @Id может быть несколько и механизм первичного ключа несколько сложнее, но я рассмотрю это в отдельной статье. Стоит отметить, что Hibernate использует аннотации из JPA для определения сущностей. Раньше поддерживались и собственные аннотации Hibernate, но с недавних пор они признаны устаревшими.

В примере у нас будет простой entity класс, с тремя полями:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | @Entity  public class Greeter {      /\*\*       \* Primary key.       \*/      @Id      @GeneratedValue      @Getter      @Setter      private Integer id;        /\*\*       \* Greeting.       \*/      @Getter      @Setter      private String greeting;        /\*\*       \* Greeting target.       \*/      @Getter      @Setter      private String target;  } |

**Настройка Hibernate**

После создания классов данных и описания их отображения в базу данных, необходимо сконфигурировать Hibernate.

При запуске Hibernate считывает настройки из специальных файлов конфигурации XML (либо из Java конфигурации). Можно использовать любой из 2-х файлов:

- persistence.xml – для любой реализации JPA (в том числе и Hibernate) – ограничен спецификацией. Я так понял, что если будет использован язык не HQL, а JPQL, то нужен именно такой файл конфигурации. Он должен находиться в папке src/main/resources/META-INF/

- hibernate.cfg.xml – специальный для Hibernate – может содержать дополнительные настройки, которые будут работать только в Hibernate. !! Этот файл используется в Hibernate по умолчанию !! Он должен находиться в папке src/main/resources/

Конфигурация может быть выполнена программно, я это обязательно покажу в каком-нибудь из примеров, либо в файле *hibernate.cfg.xml,*который должен быть доступен в classpath. Строго говоря, имя файла может быть любым, а *hibernate.cfg.xml* принято по умолчанию.

|  |  |
| --- | --- |
|  | <?xml version='1.0' encoding='utf-8'?>  <!DOCTYPE hibernate-configuration PUBLIC          "-//Hibernate/Hibernate Configuration DTD//EN"          "<http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd>">    <hibernate-configuration>        <!-- a SessionFactory instance listed as /jndi/name -->      <session-factory>  //Подключение к БД, где “test” это имя БД к которой подключаемся        <property name="hibernate.connection.url">(или jdbc:mysql://127.0.0.1:3306/test или jdbc:h2.~/test или ---)</property>  //Логин и пароль для подключения к БД         <property name=" hibernate.connection.username">root</property>        <property name=" hibernate.connection.login">root</property>  //Драйвер, который необходимо использовать для подключения к MySQL БД         <property name=" hibernate.connection.driver\_class">com.mysql.fdbc.Driver</property>  //говорит Hibernate, что надо сканировать все классы, имеющие аннотацию @Entity и обновить схему таблиц базы данных сообразно этим классам(с этой опцией Hibernate сам создаёт таблицы для ваших классов).          <property name="hibernate.hbm2ddl.auto">update</property>  //каждый поставщик СУБД использует свое подмножество поддерживаемых синтаксисов(диалектом)          <property name="hibernate.dialect">org.hibernate.dialect.(или MySQLDialect или H2Dialect или ---)</property>   //Имена всех классов-сущностей которые необходимо отобразить в БД          <mapping class="example.Greeter"/>  -------------------------------------------------------------------------      </session-factory>    </hibernate-configuration> |

В [отличие от JPA](http://easyjava.ru/data/jpa/jpa-entity-mapping/) Hibernate требует явного перечисления каждого класса сущности в конфигурации. Кроме того, в Hibernate поддерживаются описание отображений классов и связей между ними на чистом XML, без аннотаций вовсе.

**Использование Hibernate**

Использование Hibernate состоит, вообщем-то, из двух частей — сохранение объектов в базу и чтение объектов из базы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | public void setUp() throws Exception {    Greeter greetJpa = new Greeter();    greetJpa.setGreeting("Bye");    greetJpa.setTarget("JPA");      Greeter greetHibernate = new Greeter();    greetHibernate.setGreeting("Hello");    greetHibernate.setTarget("Hibernate");   //следующий блок (Bootstraping) для запуска и конфигурирования Hibernate  *Этот блок называется* Bootstraping *и будет описан в Управлении сущностями*    final StandardServiceRegistry registry = new StandardServiceRegistryBuilder()      .configure()      .build();    try {      sessionFactory = new MetadataSources(registry)        .buildMetadata()        .buildSessionFactory();    } catch (Exception e) {      StandardServiceRegistryBuilder.destroy(registry);      throw e;  }      Session session = sessionFactory.openSession();    session.beginTransaction();    session.save(greetHibernate);    session.save(greetJpa);    session.getTransaction().commit();    session.close();  } |

Вначале создаётся ServiceRegistry, которое знает о всех настройках и сконфигурированных hibernate persistence units. Из ServiceRegistry создаётся уже SessionFactory. Процесс создания SessionFactory обставлен обработкой исключений, с тем, чтобы очистить ServiceRegistry в случае провала. В случае успешного создания SessionFactory она очистит за собой ServiceRegistry сама. Из SessionFactory уже открываются сессии для работы с объектами в базе данных:  Session. SessionFactory/Session могут рассматриваться как DataSource/Connection  из JDBC. Затем в Session открывается транзакция, новые объекты записываются в базу данных, транзакция подтверждается и, наконец-то, сессия закрывается.

Попробуем прочитать объекты обратно:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | Session session = sessionFactory.openSession();  session.beginTransaction();  List<Greeter> greetings = session.createQuery("from Greeter")    .list();  greetings.forEach(g -> System.out.println(String.format("%s, %s!",      g.getGreeting(),      g.getTarget())));  session.getTransaction().commit();  session.close(); |

Опять, открываем сессию, в сессии открываем транзакцию и делаем запрос, используя HQL, язык, подобный SQL, только ориентированный на ORM. Hibernate автоматически создаёт корректный запрос, получает данные из базы данных, создаёт экземпляры объектов и наполняет их данными.

Результат исполнения это подтверждает:

Shell

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | мая 30, 2016 5:04:20 PM org.hibernate.Version logVersion  INFO: HHH000412: Hibernate Core {5.1.0.Final}  мая 30, 2016 5:04:20 PM org.hibernate.cfg.Environment <clinit>  -------------  мая 30, 2016 5:04:21 PM org.hibernate.hql.internal.QueryTranslatorFactoryInitiator initiateService  INFO: HHH000397: Using ASTQueryTranslatorFactory  Hello, Hibernate!  Bye, JPA! |

Код примера доступен на [github](https://github.com/EasyJavaRu/data-hibernate-hello).

Обычное (стандартное – без доп.настроек) подключение выглядит следующим образом:

public class HibernateUtil {

private static final SessionFactory sessionFactory;

static {

try {

sessionFactory = new Configuration().configure().buildSessionFactory(); //метод “configure().” подгружает файл c конфигурацией “hibernate.cfg.xml”, а метод “buildSessionFactory()” настраивает сессионную фабрику

//Кроме того, я так пронимаю, что использование класса Configuration с методами .configure().buildSessionFactory() заменяет собой использование классов StandartServiceRegistryBuilder и MetadataSource.

} catch (Throwable e) {

throw new ExceptionInInitializerError(e);

}

}

public static Session getSession() throws HibernateException {

return sessionFactory.openSession();

}

}

а создание сессии, для автоматического закрытия *(без обязательного закрытия методом close())* производится с помощью открытия ее в скобках блока try:

public class Test {

public static void main (String[] args) {

try (Session session = HibernateUtil.getSession()) {

session.beginTransaction();

Employee employee = new Employee();

employee.setName(“Vasya”);

employee.setAge(30);

session.save(employee);

employee = new Employee();

employee.setName(“Petya”);

employee.setAge(40);

session.save(employee);

session.getTransaction().commit();

} catch (Trowable cause) {

cause.printStackTrace();

}

List<Employee> list = null;

try (Session session = HibernateUtil.getSession()) {

session.beginTransaction();

Query query = session.createQuery(“FROM Employee”);

list = (List<Employee>) query.list();

session.getTransaction().commit();

} catch(Throable cause) {

cause.printStackTrace();

}

if (list != null && !list.isEmpty()) {

for (Employee employee : lise() {

System.out.println(employee);

}

}

}

}

**Mapping сущностей в Hibernate**

Как и в JPA, в голом [Hibernate](https://easyjava.ru/tag/hibernate/) требуется описывать как сущности отображаются на таблицы базы данных. [Hibernate](https://easyjava.ru/tag/hibernate/) поддерживает три варианта описания отображения: с использованием аннотаций, описание в xml файле, динамическое отображение. В настоящий момент рекомендуется использовать аннотации, которые активно развиваются и имеют лучший функционал.

**Annotation mapping**

Раньше Hibernate использовал собственный набор аннотаций для описания отображения сущностей на таблицы. В последних версиях эти аннотации признали устаревшими и Hibernate использует JPA аннотации для описания[отображений сущностей](http://easyjava.ru/data/jpa/jpa-entity-mapping/) и [связей между ними](http://easyjava.ru/data/jpa/jpa-i-svyazi-mezhdu-obektami/).

Главное различие между JPA и Hibernate в том, что **JPA** может **самостоятельно** **находить классы сущностей**, сканируя аннотации, если в файле “persistence.xml” есть строчка <property name="hibernate.hbm2ddl.auto" value="update" /> . А **Hibernate** **требует перечисления** всех проаннотированных классов сущностей в конфигурации:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | <hibernate-configuration>    <session-factory>      <property name="hibernate.hbm2ddl.auto">update</property>      <property name="hibernate.dialect">org.hibernate.dialect.H2Dialect</property>      <property name="hibernate.connection.url">jdbc:h2:mem:test</property>        <mapping class="ru.easyjava.data.hibernate.entity.AbstractIdentifiableObject"/>      <mapping class="ru.easyjava.data.hibernate.entity.Address"/>      <mapping class="ru.easyjava.data.hibernate.entity.Company"/>      <mapping class="ru.easyjava.data.hibernate.entity.Passport"/>      <mapping class="ru.easyjava.data.hibernate.entity.Person"/>    </session-factory>  </hibernate-configuration> |

Кроме того, Hibernate поддерживает дополнительные аннотации для отображения, которые расширяют возможности ORM. Я расскажу о их использовании подробнее позже, в статьях, посвящённых этому дополнительному функционалу.

**XML mapping**

XML mapping исторически появился в Hibernate первым и долгое время оставлялся единственным механизмом описания отображения сущностей на таблицы. В настоящий момент рекомендуется использовать аннотации, которые активно развиваются и имеют лучший функционал. Но у XML есть и свои плюсы: код сущностей остаётся нетронутым и описание отображения отделено от кода. Соответственно, это единственная возможность применить ORM в том случае, когда код сущностей недоступен или его нельзя изменять.

Традиционно каждая сущность описывается в своём отдельном файле с именем *сущность.hbm.xml.* Разумеется, имя файла и его расширение могут быть любыми, но желательно придерживаться этого соглашения.

В простом случае сущность описывается перечислением полей и указанием id поля:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | <hibernate-mapping package="ru.easyjava.data.hibernate.entity">      <class name="Passport" table="PASSPORTS">          <id name="id" type="int" column="id">              <generator class="native"/>          </id>          <property name="series" type="string" column="SERIES"/>          <property name="no" type="string" column="NO"/>          <property name="issueDate" type="timestamp" column="ISSUE\_DATE"/>          <one-to-one name="owner" class="ru.easyjava.data.hibernate.entity.Person" cascade="save-update"/>      </class>  </hibernate-mapping> |

В примере выше описывают поля сущности, названия таблицы и столбцов в ней и, кроме того, отношение один-к-одному с сущностью Person. К сожалению в XML mapping-е нет аналога  @MappedSuperclass, поэтому пришлось включать id поле в каждый класс. Это хороший пример превосходства функционала аннотаций над XML.

Впрочем, XML не так уж и беден. По крайней мере стандартные отношения между классами в нём описываются весьма полноценно. Например один-ко-многим:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | <hibernate-mapping package="ru.easyjava.data.hibernate.entity">    <class name="Address" table="ADDRESSES">      <id name="id" type="int" column="id">        <generator class="native"/>      </id>      <property name="city" type="string" column="CITY"/>      <property name="street" type="string" column="STREET"/>      <property name="building" type="string" column="BUILDING"/>      <set name="tenants" table="PERSONS" inverse="true" lazy="false">        <key>          <column name="PRIMARY\_ADDRESS"/>        </key>        <one-to-many class="ru.easyjava.data.hibernate.entity.Person"/>      </set>    </class>  </hibernate-mapping> |

Или даже многие-ко-многим:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | <hibernate-mapping package="ru.easyjava.data.hibernate.entity">    <class name="Company" table="COMPANIES">      <id name="id" type="int" column="id">        <generator class="native"/>      </id>      <property name="name" type="string" column="NAME"/>      <set name="workers" table="WORKERS" inverse="true" lazy="true">        <key column="COMPANY\_ID"/>          <many-to-many column="PERSON\_ID" class="ru.easyjava.data.hibernate.entity.Person"/>        </set>    </class>  </hibernate-mapping> |

Или даже всё сразу:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | <hibernate-mapping package="ru.easyjava.data.hibernate.entity">    <class name="Person" abstract="true" table="PERSONS">      <id name="id" type="int" column="id">        <generator class="native"/>      </id>      <property name="firstName" type="string" column="FIRST\_NAME"/>      <property name="lastName" type="string" column="LAST\_NAME"/>      <property name="dob" type="timestamp" column="DOB"/>      <one-to-one name="passport" class="ru.easyjava.data.hibernate.entity.Passport" cascade="save-update"/>      <many-to-one name="primaryAddress" class="ru.easyjava.data.hibernate.entity.Address">        <column name="PRIMARY\_ADDRESS"/>      </many-to-one>      <set name="workingPlaces" table="WORKERS" inverse="false" lazy="true" cascade="all">        <key column="PERSON\_ID"/>        <many-to-many column="COMPANY\_ID" class="ru.easyjava.data.hibernate.entity.Company"/>      </set>    </class>  </hibernate-mapping> |

Все файлы отображений так же надо перечислить в конфигурации Hibernate:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | <hibernate-configuration>    <session-factory>      <property name="hibernate.hbm2ddl.auto">update</property>      <property name="hibernate.dialect">org.hibernate.dialect.H2Dialect</property>      <property name="hibernate.connection.url">jdbc:h2:mem:test</property>        <mapping resource="Passport.hbm.xml"/>      <mapping resource="Address.hbm.xml"/>      <mapping resource="Company.hbm.xml"/>      <mapping resource="Person.hbm.xml"/>    </session-factory>  </hibernate-configuration> |

Код, который работал с сущностями в примере с аннотациями, работает без изменений с теми же самыми сущностями, но описанными в XML.

**Длинный код теста**

**Dynamic mapping**

Последнее нововведение в Hibernate, это динамическое отображение. И это не отказ от необходимости перечислять сущности в конфигурации, вовсе нет. Это всего лишь возможность не определять конкретные структуры данных со стороны Java, заменяя их на Map. То есть, если есть база данных с таблицей, допустим, *persons,*можно не определять класс для данных, хранящихся в этой таблице, а загружать её в Map. При этом частичное отображение описывать всё равно нужно.

Т.е. насколько я понял, тут мы не создаем объект класса/заполняем его поля/сохраняем этот заполненный объект в таблицу класса , а вместо это создаем коллекцию Map*(типа HashMap или др.)* /заполняем ее/и ее сохраняем в таблицу класса*.*

Но при этом в первую очередь необходимо описать таблицу и её столбцы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | <hibernate-mapping>    <class entity-name="Passport">      <id name="id" type="long" column="ID">        <generator class="native"/>      </id>      <property name="series" column="SERIES" type="string"/>      <property name="no" column="NO" type="string"/>      <property name="issueDate" column="ISSUE\_DATE" type="timestamp"/>    </class>  </hibernate-mapping> |

во вторую очередь, добавить описание таблицы в конфигурацию Hibernate и включить поддержку dynamic mapping:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | <hibernate-configuration>    <session-factory>      <property name="hibernate.hbm2ddl.auto">update</property>      <property name="hibernate.dialect">org.hibernate.dialect.H2Dialect</property>      <property name="hibernate.connection.url">jdbc:h2:mem:test</property>      <property name="default\_entity\_mode">dynamic-map</property>        <mapping resource="Passport.hbm.xml"/>    </session-factory>  </hibernate-configuration> |

И только теперь можно создавать записи в базе из Map. Обратите внимание, что при сохранении указывается имя сущности из её описания:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | Session session = sessionFactory.openSession();  session.beginTransaction();    Map<String,Object> passport = new HashMap<>();  passport.put("id", 1L);  passport.put("series", "AS");  passport.put("no", "123456");  passport.put("issueDate", new Date());    session.save("Passport", passport);    session.getTransaction().commit();  session.close(); |

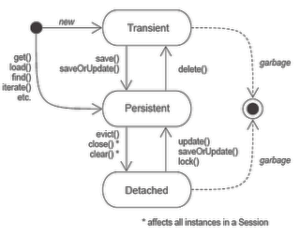
Сохранённую сущность можно и прочитать обратно в Map:

|  |  |
| --- | --- |
| 2  3 | session.createQuery("from Passport ")    .list()    .forEach(System.out::println); |
| 1 | {no=123456, series=AS, id=1, issueDate=2016-06-07 14:26:35.428, $type$=Passport} |

Код примера доступен на [github](https://github.com/EasyJavaRu/data-hibernate-mapping).

**Управление сущностями в Hibernate**

Опубликовано [**Июнь 16, 2016**](https://easyjava.ru/data/hibernate/upravlenie-sushhnostyami-v-hibernate/) автор [**EasyJava**](https://easyjava.ru/author/akashihi/) — [**2 комментария ↓**](https://easyjava.ru/data/hibernate/upravlenie-sushhnostyami-v-hibernate/#comments)

[](http://easyjava.ru/wp-content/uploads/2016/06/61dyc.png)Написав о [отображении классов в таблицы](http://easyjava.ru/data/hibernate/mapping-sushhnostej-v-hibernate/) можно написать и о работе с этими классами и таблицами.  Для управления сущностями [Hibernate](https://easyjava.ru/tag/hibernate/) использует подход, [схожий с JPA](http://easyjava.ru/data/jpa/jpa-entitymanager-upravlyaem-sushhnostyami/), когда каждая сущность имеет какое-то собственное состояние, а вызовами методов Hibernate это состояние изменяется, при этом одновременно изменяя и обновляя данные в JVM и в базе данных .

**Bootstrapping**

Как и всё остальное, Hibernate имеет начало и это начало называется Bootstrapping. Hibernate Boostrapping или, говоря простыми словами, запуск и конфигурирование Hibernate, выполняется в три шага.

В первую очередь необходимо создать ServiceRegistry, которая создаёт и предоставляет сервисы, которые нужны Hibernate для старта и дальнейшей работы. Можно построить ServiceRegistry самостоятельно, а можно воспользоваться стандартной. Второй вариант, очевидно, проще:

|  |  |
| --- | --- |
| 1  2  3 | final StandardServiceRegistry registry = new StandardServiceRegistryBuilder()    .configure()    .build(); |

Вызывая различные методы класса StandardServiceRegistryBuilder можно управлять, какой будет построенная  ServiceRegistry. В частности именно тут можно изменить имя файла конфигурации со значения по умолчанию *hibernate.cfg.xml*на своё собственное.

Следующим шагом настраивается связь между классами и таблицами:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | try {    MetadataSources metadataSources = new MetadataSources(registry);  } catch (Exception e) {    StandardServiceRegistryBuilder.destroy(registry);    throw e;  } |

Создание MetadataSources оборачивается в *try/catch* блок, чтобы очистить ServiceRegistry если MetadataSources не сможет создаться. В случае успеха же  MetadataSource сам будет управлять жизненным циклом ServiceRegistry.

Создать MetadatsSources можно и без ServiceRegistry, но тогда не будут автоматически прочитаны определения сущностей из конфигурации Hibernate и их придётся задавать вручную:

|  |  |
| --- | --- |
| 1  2  3 | metadataSources.addAnnotatedClass(ru.easyjava.data.hibernate.entity.Address.class);  metadataSources.addAnnotatedClassName("ru.easyjava.data.hibernate.entity.Address.class");  metadataSources.addResource("classpath:/Address.hbm.xml"); |

В коде выше показано, как можно добавить класс по его типу, класс по его имени или определение сущности из внешнего файла.

Наконец, последним шагом, из MetadataSources строится SessionFactory:

|  |  |
| --- | --- |
| 1 | sessionFactory = metadataSources.buildMetadata().buildSessionFactory(); |

**Session**

Из SessionFactory, построенной при запуске Hibernate, можно получить одну или несколько сессий. Если проводить аналогию с JDBC, то SessionFactory будет аналогом DataSource, а Session  — аналогом Connection. Для нас главное различие в том, что SessionFactory объект достаточно тяжёлый и его создание занимает довольно много времени. Поэтому в программе обычно используется один экземпляр SessionFactory. В тоже время объект Session весьма легковесный и конструируется быстро, поэтому их создают по мере необходимости.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | Session session = sessionFactory.openSession();  session.beginTransaction();    // Some changes here    session.getTransaction().commit();  session.close(); |

Каждый объект Session может так же иметь внутри себя объект Transaction, обращения к которому позволяют управлять транзакциями в рамках контекста сессии.

Контекст сессии понятие достаточно эфемерное. Один или несколько объектов Session могут образовывать *persistence context*. Я не буду переводить этот термин, попробую лучше его объяснить. Наличие p*ersistence context*означает, что для каждой существующей на данный момент сущности существует Session, которая с сущностью связана и следит за её состоянием. Что это значит? Смотри ниже.

**Сущности и состояния**

В Hibernate каждая сущность имеет своё состояние и может менять его подчиняясь строгим правилам перехода из состояния в состояние. Эти состояния и правила отображены на рисунке в начале статьи.

Когда будущая сущность создаётся оператором new, она чиста и невинна и Hibernate про неё ещё не знает. Такая свежесозданная сущность считается *«транзитной» (transient). Транзитная* сущность не имеет связи с базой данных, не имеет данных в базе данных и, за редким исключением, не имеет id.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | //Transient op  Operation op = new Operation();    op.setId(1L);  op.setAccountId(100500);  op.setAmount(BigDecimal.TEN);  op.setTimestamp(ZonedDateTime.now());  op.setDescription("Test operation");  op.setOpCode(9000); |

Такая сущность довольно бесполезна, поэтому сразу переведём её в состояние *«постоянная» (persistent):*

|  |  |
| --- | --- |
| 1  2 | Long opRowId = (Long)session.save(op); //в opRowId будет находиться id этой записи  System.out.println("Saved entity and assigned id: " + opRowId); |

Метод **save()** cохраняет объект в БД и возвращает id, который был присвоен сущности в момент сохранения. Кроме save() есть ещё несколько методов, способных перевести сущность из *«transient»* в *«persistent»:*

*- saveOrUpdate()*позволяет не задумываться, новая это сущность или просто изменённая старая и сам выбирает, сохранить новую или обновить существующую;

- persist() ведёт себя как save(), но он не возвращает id вновь сохранённой сущности.

Hibernate, как и JPA сам (по умолчанию с версии 5.2) следит за изменениями в объектах и не требует явного обновления сущности в базе данных, когда она меняется в коде (как это было раньше).

**Чтение из базы**

Разумеется, создание новых объектов и сохранение их в базу, это не единственный метод сделать какой-то объект *«persistent»*и, строго говоря, даже не самый популярный. Чаще всего объекты наоборот, читаются из базы данных с помощью методов ***get()*** и ***load()***. Оба метода принимают одинаковый набор параметров и в общем случае это будет тип загружаемого класса и его id:

|  |  |
| --- | --- |
| 1  2 | System.out.println(session.load(Operation.class, opRowId));  System.out.println(session.get(Operation.class, opRowId)); |

Отличаются эти методы лишь поведением: ***get()*** возвращает null, если объекта указанного типа и с указанным id в базе не нашлось. В то же время ***load()*** бросит исключение при попытке загрузить несуществующий объект. А если быть точным, то не при попытке загрузить объект, а при попытке обратиться к данным сущности. Дело в том, что load() загружает данные из базы *«лениво»,*откладывая фактическую загрузку на момент обращения к этим данным. А если к данным не обращаться, то и фактического запроса к базе тоже не произойдёт

**Ленивая загрузка**

Когда сущность стала *persistent,*она обретает связь с объектом Session, который её породил и включается в *persistent context.*Метод load() возвращает на самом деле не объект запрошенного класса, а ссылку на обёртку вокруг этого класса, которая знает, как ей обратиться к Session и загрузить данные в тот момент, когда они понадобятся. Очевидно, что такая обёртка бесполезна за пределами своего *peristence context.*

Однако не только load() может вернуть неполный класс. Любое поле может быть помечено для ленивой загрузки аннотацией @Basic( fetch = FetchType.LAZY ) и даже если загрузить объект с таким полем методом get(), для доступа к этому полю всё равно будет требоваться наличие *persistence context*. Используя аннотацию @LazyGroup("groupname") можно объединять лениво загружаемые поля в группы и тогда обращение к одному полю в группе автоматически вызовет загрузку всех полей в этой группе.

По умолчанию у любого класса, который не указывает явно, какие поля стоит загружать отложенно и как такие поля группировать, все поля которые содержат единственное значение, такие как String, BigDecimal, Long и другие, объединены в группу загрузки по умолчанию. В свою очередь поля с множественными значениями, такие как Set, List и другие, имеют собственную группу отложенной загрузки для каждого поля.

Таким образом можно сказать, что если у вас есть какое-то поле с множественными элементами в классе и вы явно не настраивали для него ленивую загрузку, оно, скорее всего, будет загружено лениво и для доступа к элементам этого поля будет требоваться наличие *persistence context* у сущности.

**Detached entities**

Итак, когда сущность только создана и записана в базу данных или когда наоборот, прочитана из базы данных, она входит в *persistence context* и обладает неким экземпляром Session, который ей управляет. Однако из этого состояния она может внезапно перейти в состояние *«отделённая» (detached).*В этом состоянии сущность не связана со своим контекстом (отделена от него) и нет экземпляра Session, который бы ей управлял.

Перейти в это состояние сущность может по следущим причинам:

* Явный перевод из *persisted*в*detached*вызовом метода evict() у Session.
* Сброс контекста методом clear() у Session.
* Явное закрытие сессии методом close().
* Неявное закрытие сессии связанное с удалением объекта Session.

Над *detached*объектом нельзя выполнять операции, которые требуют наличия *persistence context:*

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | session = sessionFactory.openSession();  session.beginTransaction();  Operation opD = session.load(Operation.class, 1L);  session.getTransaction().commit();  session.close();    System.out.println(opD.getDescription()); |

|  |  |
| --- | --- |
| 1  2  3 | org.hibernate.LazyInitializationException: could not initialize proxy - no Session    at org.hibernate.proxy.AbstractLazyInitializer.initialize(AbstractLazyInitializer.java:147) |

*detached* сущность можно вернуть в состояние *persisted* вызовами merge(), lock() или update(), но не saveOrUpdate()!

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | session = sessionFactory.openSession();  session.beginTransaction();  session.merge(opD);  System.out.println(opD.getDescription());  session.getTransaction().commit();  session.close(); |

Причем merge() не только возвращает сущность в состояние *persisted*, но и накатывает на нее все обновления сущности, которые произошли в этом классе, после того как сущность ушла в состояние *detached*

Book \_book = doInJPA(entityManager -> {

Book book = new Book()

.setIsbn("978-9730228236")

.setTitle("High-Performance Java Persistence")

.setAuthor("Vlad Mihalcea");

entityManager.persist(book);

return book;

});

LOGGER.info("Modifying the Book entity");

**\_book.setTitle(**

**"High-Performance Java Persistence, 2nd edition"**

);

doInJPA(entityManager -> {

Book book = entityManager.merge(\_book);

LOGGER.info("Merging the Book entity");

assertFalse(book == \_book);

});

При выполнении приведенного выше примера Hibernate выполнил следующие операторы SQL:

INSERT INTO book (

author,

isbn,

title,

id

)

VALUES (

'Vlad Mihalcea',

'978-9730228236',

**'High-Performance Java Persistence',**

1

)

-- Modifying the Book entity

SELECT

b.id,

b.author AS author2\_0\_,

b.isbn AS isbn3\_0\_,

b.title AS title4\_0\_

FROM

book b

WHERE

b.id = 1

-- Merging the Book entity

UPDATE

book

SET

author = 'Vlad Mihalcea',

isbn = '978-9730228236',

**title = 'High-Performance Java Persistence, 2nd edition'**

WHERE

id = 1

**JSON:**

Аннотация @JsonIgnoreProperties("usersList") (где "usersList" - это имя связанного поля в дочерней таблице) + такая-же аннотация в дочерней сущности (с обратной ссылкой) разрывают циклическую ссылку в работе JSON

**Удаление:**

**Для полей родительских экземпляров с аннотациями @ManyToMany, @ManyToOne применять** cascade = CascadeType.ALL **нельзя,** поскольку это приведет к удалению/изменению дочерних экземпляров, которые могут быть связаны с другими родительскими экземплярами. Поэтому для @ManyToMany ассоциаций вы обычно каскадируете операции используя =CascadeType.PERSIST или =CascadeType.MERGE.  Что касается Hibernate 5.2.8, похоже, нет способа достичь того же эффекта с аннотациями JPA. Например, @ManyToMany(cascade={PERSIST, MERGE, REFRESH, DETACH})(все, кроме REMOVE) не выполняет каскадные обновления, В этом случаи нужно использовать  =CascadeType.SAVE\_UPDATE.

**- всех объектов из сущности -** session.createQuery("delete from ForDelete").executeUpdate();

**- конкретного объекта из сущности -** session.createQuery("DELETE User WHERE id = :id"). setParameter("id", 2).executeUpdate() *(удалит объект в корневой/связанной сущности и записи в промежуточных таблицах, но оставит записи в связанных/корневой таблицах)*;

- **конкретного объекта из сущности и связанных с ним сущностей:**

**для HQL** User user = session. load(User.class, 50); if (user != null) { session.delete(user);}

**для JPQL** User user1 = entityManager. find(User.class, 50); if (user1 != null) { entityManager.

remove(user1);}

*или*

User user = new User(“a”); User user1 = entityManager. find(User.class, user.getId()); if (user1 != null) { entityManager.remove(user1);}

*(Удалит объект в корневой сущности и все записи по этому объекту в промежуточных и связанных таблице/ах, при установленном в корневой сущности "cascade=CascadeType.ALL". Если это не установить, то в связанной таблице/ах ничего не удалит.*

*Также это написание спокойно удаляет любой объект из связанной таблицы , при этом ни как не влияя на корневую и промежуточную таблицы. И на это не влияет наличие/осутствие в корневой сущности "cascade=CascadeType.ALL")*

- **дочерних записей без необходимости удалять родительскую запись** - заменить cascade = CascadeType.ALL на orphanRemoval = true в ассоциации @OneToMany

- **не связанной ни с кем сущности –** 1. удалил из Main.java весь код, что ссылался на эту сущность 2. Удалил из файла hibernate.cfg.xml ссылку на эту сущность 3. П.к.м. Delete-ом удалил сущность из списка в IDEA 4. П.к.м. DropTable-ом удалил эту таблицу в MySQL Workbench. – ***А до этого как-то пытался в другом порядке это делать, так Main.java перестал работать и требовало наличае удаленной сущности.***

**- мягкое удаление (просто отфильтровываем эту переменную при извлечении) –** для класса-сущности указывается аннотация @Where(clause = "DELETED = 0"), где DELETED это @Column(name = "DELETED") этой переменной.

**Удаление связанных сущностей**

Наличие или отсутствие связанной сущности в базе данных определяет способ удаления. Если связанная сущность отсутствует, то можно использовать оператор DELETE в HQL-запросе объекта Query. Но если сущность содержит связанный объект в таблице БД, то при выполнении транзакции удаления с использованием объекта Query будет вызвано соответствующее исключение. Удаление связанных сущностей необходимо выполнять с использованием объекта сессии Session. Подробнее об этом представлено при описании оператора DELETE в [HQL](http://java-online.ru/hibernate-hql.xhtml#delete)-запросе.

**Связанные сущности в Hibernate**

Статья является продолжением описания примера использования Sequence в Hibernate, в которой была рассмотрена только одна сущность и представлено описание сессии [Session](http://java-online.ru/hibernate-sequence.xhtml#session). В данной статье с использованием практического примера рассмотрим вопрос определения связей между сущностями.

***Примечание :****чтобы не вынуждать Вас обращаться к начальной статье, часть информации здесь будет повторена.*

Классы в Java могут не только [наследоваться](http://java-online.ru/java-oop.xhtml) друг от друга, но и включать в себя в качестве полей другие классы или [коллекции](http://java-online.ru/java-collection.xhtml) классов. В столбцах таблиц БД нельзя хранить сложные составные типы и коллекции таких типов (за некоторыми исключениями). Это не позволяет сохранять подобный объект в одну таблицу. Но можно сохранять каждый класс в свою собственную таблицу, определив необходимые связи между ними. C описания связей между объектами и начнем.

**Определение связей между сущностями**

Для определения связей между сущностями **Hibernate** использует аннотации @OneToOne, @OneToMany, @ManyToOne, @ManyToMany.

**@OneToOne**

Рассмотрим описание аннотации на примере, что каждый гражданин может иметь только один паспорт. И у каждого паспорта может быть только один владелец. Такая связь двух объектов в Hibernate определяется как @OneToOne (*один-к-одному*). В следующем листинге представлено описание двух сущностей Person (гражданин) и Password (паспорт). Лишние строки, не связанные с аннотацией **@OneToOne**, не включены в описания сущностей.

@Entity

@Table (name="users")

public class Person

{

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private int id;

@Column (name="namee")

private String namee;

@OneToOne (optional=false, cascade=CascadeType.ALL, orphanRemoval = true)

@JoinColumn (name="passport\_id")

private Passport passport;

}

@Entity

@Table (name="passports")

public class Passport

{

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private int id;

@Column (name="series")

private String series;

@Column (name="number")

private String number;

@OneToOne (optional=false, mappedBy="passport")

private Person owner;

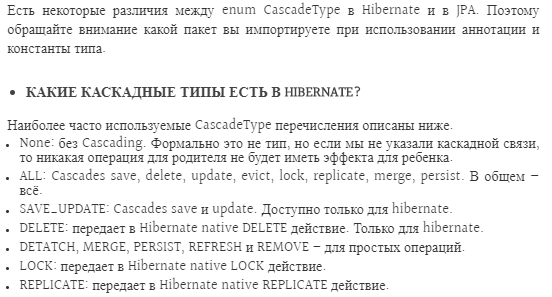
}

Для связи один к одному в обоих классах к соответствующим полям добавляется аннотация @OneToOne. Параметр optional говорит [JPA](http://java-online.ru/hibernate-jpa.xhtml), является ли значение в этом поле обязательным или нет. Связанное поле в User объявлено с помощью аннотации **@JoinColumn**, параметр name которой обозначает поле в БД для создания связи. Для того, чтобы объявить сторону, которая не несет ответственности за отношения, используется атрибут mappedBy в сущности Passport. Он ссылается на имя свойства связи (passport) на стороне владельца.

Со стороны владельца к аннотации @OneToOne добавляется параметр cascade, который говорит какое действие с “родительским” или “ребенком” объектом будет распространяться и на связанный объект “ребенка” или “родителя” соответственно (*save, delete, persist* и т.д.)**\***. В однонаправленных отношениях одна из сторон (и только одна) должна быть владельцем и нести ответственность за обновление связанных полей. В этом случае владельцем выступает сущность User. Каскадирование позволяет указать JPA, что необходимо «сделать со связанным объектом при выполнении операции с владельцем». То есть, когда удаляется Person из базы, JPA самостоятельно определит наличие у него паспорта и удалит вначале паспорт, потом гражданина.

Параметр *orphanRemoval* объявляет, что связанные экземпляры сущностей должны быть удалены, когда они отсоединены от родителя или когда родитель удален.

Связь в БД между таблицами users и passports осуществляется посредством поля passport\_id в таблице users.

**\*** - 

**@OneToMany и @ManyToOne**

Аннотации @OneToMany (*один-ко-многим*) и @ManyToOne (*многие-к-одному*) рассмотрим на примере гражданина и его места проживания. Гражданин имеет один основной адрес проживания, но по одному адресу могут проживать несколько человек. В следующем листинге представим эти сущности (лишние поля, не связанные с аннотациями, не отображаются) :

@Entity

@Table (name="users")

public class Person

{

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private int id;

private String name;

@OneToOne (optional=false, cascade=CascadeType.ALL)

@JoinColumn (name="passport\_id")

private Passport passport;

@ManyToOne (optional=false)

@JoinColumn (name="person\_id") //по аналогии с предыдущим примером наверное должно

быть "address\_id"

private Address addres;

}

@Entity

@Table (name="address")

public class Address

{

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private int id;

private String city;

private String street;

private String building;

@OneToMany (mappedBy="addres", fetch=FetchType.EAGER)

private Collection<Person> tenants;

}

Владельцем в этом примере также будет класс Person, который имеет поле address, связанное с соответствующим объектом. Поскольку адрес у гражданина только один, то используется аннотация @ManyToOne. Аннотацией @JoinColumn определяется поле связи в таблице БД. Таким образом, параметры этих аннотаций несут такую же смысловую нагрузку, что и у связи @OneToOne.

А вот у владеемого объекта на этот раз всё иначе. Поскольку по одному адресу может проживать несколько жильцов, то поле tenants представлено коллекцией, которая имеет аннотацию @OneToMany. Параметр mappedBy также указывает на поле в классе владельца. Параметр fetch=FetchType.EAGER говорит о том, что при загрузке владеемого объекта необходимо сразу загрузить и коллекцию владельцев.

Для чтения связанных объектов из БД используются следующие стратегии загрузок (fetch) : EAGER и LAZY. В первом случае объекты коллекции сразу загружаются в память, во втором случае — только при обращении к ним. Оба этих подхода имеют достоинства и недостатки.

В случае **FetchType.EAGER** в памяти будут находиться все загруженные объекты, даже если нужен только один объект из десятка (сотен/тысяч). При использовании данной стратегии необходимо быть внимательным, поскольку при загрузке какого-нибудь корневого объекта, который связан со всеми остальными объектами и коллекциями, можно случайно попытаться загрузить в память и всю базу.

Согласно стратегии **FetchType.LAZY** связанные объекты загружаются только по мере необходимости, т.е. при обращении. Но при этом требуется, чтобы соединение с базой (или транзакция) сохранялись. Если быть точно, то требуется, чтобы объект был attached. Поэтому для работы с lazy объектами тратится больше ресурсов на поддержку соединений.

**@ElementCollection**

В чем разница между использованием аннотации @OneToMany и @ElementCollectionаннотации, поскольку оба работают с отношением «один ко многим»?

- @ElementCollection используется, когда существование дочернего объекта бессмысленно без родительского объекта, IOW, всякий раз, когда родительский объект удаляется, ваши дети также будут ....

- ElementCollection- это стандартная аннотация JPA, которая теперь предпочтительнее проприетарной аннотации Hibernate CollectionOfElements.

Это означает, что коллекция - это не набор сущностей, а набор простых типов (строки и т. Д.) Или набор встраиваемых элементов (класс, помеченный @Embeddable).

Это также означает, что элементы полностью принадлежат содержащим сущностям: они модифицируются при изменении сущности, удаляются при удалении сущности и т. Д. У них не может быть собственного жизненного цикла.

- @ElementCollection это в основном для отображения не-сущностей (встраиваемых или базовых), а @OneToMany используется для сопоставления сущностей. Итак, какой из них использовать, зависит от того, чего вы хотите достичь.

**-** @ElementCollection позволяет упростить код, если вы хотите реализовать отношение «один ко многим» с помощью простого или встроенного типа. Например, в JPA 1.0, когда вы хотели иметь отношение «один ко многим» со списком Strings, вам нужно было создать простую сущность POJO ( StringWrapper), содержащую только первичный ключ и рассматриваемый String:

@OneToMany

private Collection<StringWrapper> strings;

//...

public class StringWrapper {

@Id

private int id;

private String string;

}

С JPA 2.0 вы можете просто написать в главной сущности:

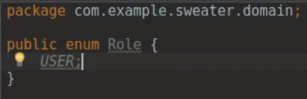
@ElementCollection

private Collection<String> strings;

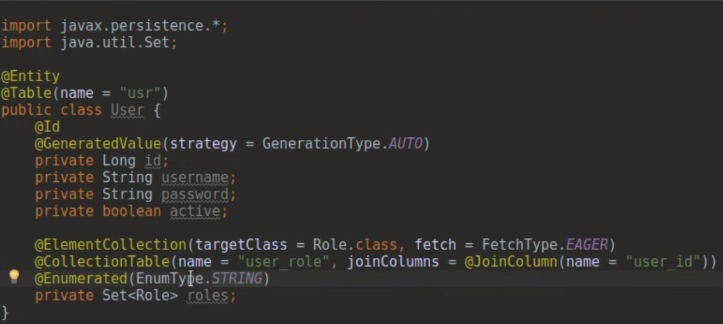
Проще, правда? Обратите внимание, что вы по-прежнему можете управлять именами таблиц и столбцов с помощью @CollectionTable аннотаций.

Пример:

Просто class или enum, для хранения этой доп. информации:



А связь + название таблицы + тип хранимой в ней информации, будем создавать в сущности:



**@ManyToMany *(создается не 2, а 3 таблицы)***

Примером ассоциации @ManyToMany (*многие-ко-многим*) могут быть отношения студентов и ВУЗов. В одном институте может быть много студентов, студент может учиться в нескольких ВУЗах. Рассмотрим с начала таблицы БД :

create table student (

id integer not null,

name varchar(255) default null,

CONSTRAINT PK\_STUDENT\_ID PRIMARY KEY (id)

);

create table university (

id integer not null,

name varchar(255) default null,

CONSTRAINT PK\_UNIVERSITY\_ID PRIMARY KEY (id)

);

create table student\_university (

student\_id integer not null,

university\_id integer not null,

CONSTRAINT FK\_STUDENT\_ID FOREIGN KEY (student\_id)

REFERENCES student (id),

CONSTRAINT FK\_UNIVERSITY\_ID FOREIGN KEY (university\_id)

REFERENCES university (id)

);

Для определения связи @ManyToMany в примере потребуется три таблицы : таблица студентов students, таблица ВУЗов university и таблица связей student\_university, в которой будут связаны студенты и ВУЗы. Кроме этого в таблице student\_university определены внешние ключи (FOREIGN KEY), предупреждающие появление непрошенных записей при отсутствии родительских.

Теперь можно представить описание сущностей :

@Entity

public class Student

{

@Id

private long id;

private String name;

@ManyToMany

@JoinTable (name="student\_university",

joinColumns=@JoinColumn (name="student\_id"),

inverseJoinColumns=@JoinColumn(name="university\_id"))

private List<University> universities; //List используется если нужна упорядоченность, двойники или null

// set/get не представлены

}

@Entity

public class University

{

@Id

private long id;

@Column

private String name;

@ManyToMany

@JoinTable(name="student\_university",

joinColumns=@JoinColumn(name="university\_id"),

inverseJoinColumns=@JoinColumn(name="student\_id"))

или, что равнозначно

@JoinTable(mappedBy = “universities”)

private List<Student> students; //List используется если нужна упорядоченность, двойники или null

// set/get не представлены

}

Список институтов в сущности Student аннотирован с помощью **@ManyToMany**. Далее следует аннотация **@JoinTable**, которая определяет таблицу и поля для связи. Параметр name указывает название таблицы (student\_university). Параметр **joinColumns** указывает на поле, которое используется для прямой связи (идентификатор student\_id). Параметр inverseJoinColumns указывает на поле, которое используется для обратной связи (идентификатор university\_id). Для указания столбцов связи из таблицы используется аннотация @JoinColumn.

Сущность университета University описана "зеркально".

**Пример связанных сущностей**

Рассмотрим пример использования аннотаций @OneToMane и @ManyToOne при определении связанных сущностей. В качестве первой сущности будет использоваться пользователь User. Второй сущностью будет автомобиль Auto. Пользователь может владеть несколькими автомобилями, поэтому сущность User будет связана с Auto связью @OneToMany (*один-ко-многим*). Сущность Auto будет связана с сущностью User связью @ManyToOne (*многие-к-одному*). Начнем с объектов базы данных :

**SQL-скрипты создания таблиц пользователей и автомобилей**

-- SQL-скрипт создания таблицы пользователей

create table USERS (

id Integer not null,

login varchar2 (16) null,

name varchar2 (64) null,

data timestamp default SYSDATE,

CONSTRAINT PK\_USERID PRIMARY KEY (id)

);

-- SQL-скрипт создания таблицы автомобилей

CREATE TABLE AUTOS (

aid Integer not null,

user\_id Integer,

name varchar2(32),

CONSTRAINT pk\_AUTOSK\_ID PRIMARY KEY (aid),

CONSTRAINT fk\_USERID FOREIGN KEY (user\_id)

REFERENCES USERS (id)

);

Записи таблицы пользователей ничего не знают о записях таблицы автомобилей. А записи таблицы Autos связаны с таблицей Users по внешнему ключу (поле user\_id). Синтаксис описания внешних ключей в базах данных представлен [здесь](http://java-online.ru/sql-foreignkey.xhtml).

**SQL-скрипт создания Sequence**

Генератор последовательностей SEQ\_USER используем для определении идентификаторов записей сущностей. Как работать с генераторами последовательностей **Sequence** в SQL подробно представлено [здесь](http://java-online.ru/sql-sequence.xhtml).

create sequence SEQ\_USER

minvalue 1

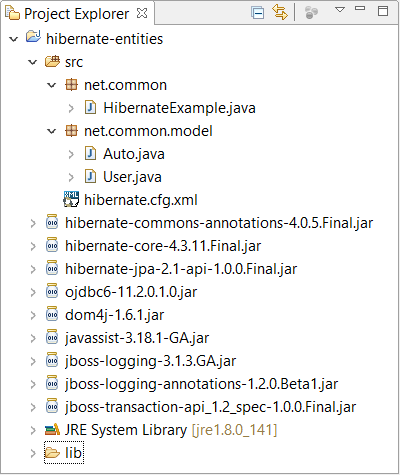
start with 10

increment by 1

cache 5;

**Проект тестирования связанных сущностей Hibernate**

На следующем скриншоте представлена структура проекта hibernate-entities в среде разработки Eclipse. В проекте необходимо определить файл конфигурации **hibernate.cfg.xml** и классы-сущности (User и Auto). Модуль *HibernateExample* будет тестировать настройки hibernate и сущностей. Все библиотеки, необходимые для работы с Oracle и hibernate, размещены в поддиректории lib. После включения их в CLASSPATH они отображаются в корне проекта Eclipse.



***Примечание :****в демонстрационном*[*примере*](http://java-online.ru/hibernate-example.xhtml)*hibernate был использован «файл-маппинг» person.cfg.xml сущности Person. В данном примере вместо «файл-маппингов» будем использовать*[*аннотации*](http://java-online.ru/java-annotation.xhtml)*Подробная информация об аннотациях JPA представлена*[*здесь*](https://docs.oracle.com/javaee/7/api/javax/persistence/package-summary.html)*.*

**Конфигурация hibernate**

В конфигурационном XML-файле **hibernate.cfg.xml** определяем сервер БД (драйвер, пул подключений, диалект, кодировку) и параметры подключения (url, login, password), а также дополнительные параметры, которые будут использованы при работе с сервером. В качестве сервера БД выбран Oracle c пулом подключений в одно соединение. В демонстрационном [примере](http://java-online.ru/hibernate-example.xhtml) в качестве сервера БД использовался MySQL.

Дополнительно определяем истиное значение свойства "show\_sql" для отображения в консоли SQL-скриптов, генерируемых библиотекой Hibernate. В заключении в обязательном порядке определяем маппинг сущностей/классов User и Auto, чтобы не вызывать исключений.

<?xml version='1.0' encoding='utf-8'?>

<!DOCTYPE hibernate-configuration PUBLIC

"-//Hibernate/Hibernate Configuration DTD//EN"

"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

<session-factory>

<!-- Database connection settings -->

<property name="hibernate.connection.driver\_class">

oracle.jdbc.OracleDriver</property>

<property name="hibernate.connection.url">

jdbc:oracle:thin:@localhost:1521:SE</property>

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

scott</property>

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

tiger</property>

<!-- JDBC connection pool (use the built-in) -->

<property name="hibernate.connection.pool\_size">1</property>

<!-- Echo all executed SQL to stdout -->

<property name="show\_sql">true</property>

<!-- SQL dialect -->

<property name="hibernate.dialect">

org.hibernate.dialect.Oracle10gDialect</property>

<property name="hibernate.current\_session\_context\_class">

thread</property>

<property name="hibernate.connection.CharSet">

utf8</property>

<property name="hibernate.connection.characterEncoding">

utf8</property>

<property name="hibernate.connection.useUnicode">

true</property>

<!-- Сущности User и Auto-->

<mapping class="net.common.model.User"/>

<mapping class="net.common.model.Auto"/>

</session-factory>

</hibernate-configuration>

**Листинг класса пользователя User**

Описание сущности/класса User незначительно изменилось. Добавилось поле List<Auto> autos, определяющее список автомобилей пользователя.

@Entity

@Table(name = "USERS")

public class User

{

@Id

@GeneratedValue(strategy=GenerationType.SEQUENCE,

generator="users\_seq")

@SequenceGenerator(name="users\_seq",

sequenceName="SEQ\_USER",

allocationSize=5)

@Column(name="id", updatable=false, nullable=false)

private Integer id;

@Column (name="name", nullable=true)

private String name;

@Column (name="login")

private String login;

@OneToMany(fetch = FetchType.LAZY,

mappedBy = "user",

cascade = CascadeType.ALL)

private List<Auto> autos;

public User() {}

public User(Integer id, String login, String name)

{

super();

this.id = id;

this.login = login;

this.name = name;

}

public Integer getId() {

return id;

}

public void setId(Integer id) {

this.id = id;

}

@Column

public String getLogin() {

return login;

}

public void setLogin(String login) {

this.login = login;

}

@Column (name="name", nullable=true)

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public List<Auto> getAutos() {

return autos;

}

public void setAutos(List<Auto> autos) {

this.autos = autos;

}

public String toString() {

String cars = "";

if ((autos != null) && (autos.size() > 0)) {

for (int i = 0; i < autos.size(); i++) {

if (i > 0)

cars += ",";

cars += autos.get(i).toString();

}

}

return "User {id = " + String.valueOf(id) +

", login = '" + login +

", name = '" + name + "', autos =[" + cars + "]}";

}

}

Описание аннотаций @Table, @Id, @Column, @GeneratedValue, @SequenceGenerator сущности User представлено в [предыдущей](http://java-online.ru/hibernate-sequence.xhtml) статье. Здесь дополним список описанием аннотации  
*@OneToMany*  
Атрибут *fetch* в аннотации, определяющий стратегию загрузки дочерних объектов, может принимать одно из двух значений перечисления javax.persistence.FetchType :

* FetchType.EAGER — загружать коллекцию дочерних объектов вместе с загрузкой родительских объектов;
* FetchType.LAZY — загружать коллекцию дочерних объектов при первом обращении к ней (вызове метода get) — это так называемая отложенная загрузка.

Атрибут *cascade* обозначает, какие из методов интерфейса *Session* будут распространяться каскадно к ассоциированным сущностям. Возможные варианты : CascadeType.ALL, CascadeType.PERSIST, CascadeType.MERGE. Необходимо правильно настроить CascadeType, чтобы не подгружать из базы данных лишних ассоциированных объектов-сущностей.

**Листинг класса автомобиля Auto**

При описании поля user используется аннотация *@ManyToOne*. Аннотация *@JoinColumn* определяет поле таблицы БД, по которому сущность Auto связана с пользователем User.

@Entity

@Table(name = "AUTOS")

public class Auto

{

@Id

@GeneratedValue (strategy=GenerationType.SEQUENCE,

aid Integer not null,

CONSTRAINT pk\_AUTOSK\_ID PRIMARY KEY (aid),

generator="users\_seq")

@SequenceGenerator (name="users\_seq",

sequenceName="SEQ\_USER",

allocationSize=5)

@Column (name="aid")

private Integer id;

user\_id Integer,

CONSTRAINT fk\_USERID FOREIGN KEY (user\_id) REFERENCES USERS (id)

@ManyToOne (fetch=FetchType.LAZY,

cascade=CascadeType.ALL)

@JoinColumn (name="user\_id")

private User user;

@Column(name = "name")

private String name;

public Auto()

{

super();

}

public Auto(Integer id, User user, String name)

{

super();

this.id = id;

this.user = user;

this.name = name;

}

public Integer getId() {

return id;

}

public void setId(Integer id) {

this.id = id;

}

public User getUser() {

return user;

}

public void setUser(User user) {

this.user = user;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String toString() {

return "Auto {id = " + String.valueOf(id) +

", name = '" + name +

", user\_id = " + user.getId() + "}";

}

}

**Листинг класса тестирования HibernateExample**

HibernateExample используется для тестирования связей между сущностями Hibernate. Сначала создается сессия в методе createHibernateSession. При создании session устанавливается соединение с БД Oracle. Если сессия создана успешо, то в методе saveUser создаются два объекта (user1, user2), открывается транзакция и объекты сохраняются в БД. Для сохранения объектов используются методы save класса Session. После этого создаются два объекта типа Auto, у которых полям user присваивается значение первого пользователя. Объекты автомобилей сохраняются в БД и транзакция завершается.

После сохранения объектов в БД, пользователь user1 обновляется с использованием метода refresh() объекта сессии. Описание методов Session представлено [здесь](http://java-online.ru/hibernate-sequence.xhtml#session).

package net.common;

import java.util.List;

import java.util.Iterator;

import net.common.model.User;

import org.hibernate.Query;

import org.hibernate.Session;

import org.hibernate.Transaction;

import org.hibernate.SessionFactory;

import org.hibernate.cfg.Configuration;

import org.hibernate.service.ServiceRegistry;

import org.hibernate.boot.registry.StandardServiceRegistryBuilder;

public class HibernateExample

{

private Session session = null;

//---------------------------------------------------------------

private Session createHibernateSession()

{

SessionFactory sf = null;

ServiceRegistry srvc = null;

try {

Configuration conf = new Configuration();

conf.configure("hibernate.cfg.xml");

srvc = new StandardServiceRegistryBuilder()

.applySettings(conf.getProperties()).build();

sf = conf.buildSessionFactory(srvc);

session = sf.openSession();

System.out.println("Создание сессии");

} catch (Throwable e) {

throw new ExceptionInInitializerError(e);

}

return session;

}

//---------------------------------------------------------------

private void saveUser()

{

if (session == null)

return;

User user1 = new User();

user1.setName ("Иван");

user1.setLogin("ivan");

User user2 = new User();

user2.setName ("Сергей");

user2.setLogin("serg" );

Transaction trans = session.beginTransaction();

session.save(user1);

session.save(user2);

Auto auto = new Auto();

auto.setName("Volvo");

1.

auto.setUser(user1);

session.saveOrUpdate(auto);

auto = new Auto();

auto.setName("Skoda");

2.

auto.setUser(user1);

session.saveOrUpdate(auto);

session.flush();

trans.commit();

/\*

\* Обновление данных по “user1”, после добавления автомобилей

\*/

session.refresh(user1);

System.out.println("user1 : " + user1.toString());

System.out.println("user2 : " + user2.toString());

}

//---------------------------------------------------------------

public HibernateExample()

{

// Создание сессии

session = createHibernateSession();

if (session != null) {

System.out.println("session is created");

// Добавление записей в таблицу

saveUser();

// Закрытие сессии

session.close();

} else {

System.out.println("session is not created");

}

}

//---------------------------------------------------------------

public static void main(String[] args)

{

new HibernateExample();

System.exit(0);

}

}

**Выполнение примера**

При выполнении примера в консоль выводится информация, представленная ниже. Поскольку установлен соответствующий флаг в файле конфигурации hibernate.cfg.xml, то формируемые библиотекой Hibernate SQL-скрипты также отображаются в консоли.

Информация, выведенная **Hibernate** в консоль, показывает, что сначала формируются SQL-скрипты (запросы к Sequence) для получения идентификаторов объектов пользователя и автомобиля, после этого создаются SQL-скрипты добавления пользователей и автомобилей в БД. И в заключение Hibernate создает SQL-скрипт select с использованием [left outer join](http://java-online.ru/sql-join.xhtml) для обновления объектов.

«Распечатка» описаний пользователей показывет, что первый user имеет автомобили, второй — нет. Как Hibernate с использованием Sequence определяет значения идентификаторов подробно представлено в предыдущей статье.

Создание сессии

session is created

Hibernate: select SEQ\_USER.nextval from dual

Hibernate: select SEQ\_USER.nextval from dual

Hibernate: insert into USERS (login, name, id) values (?, ?, ?)

Hibernate: insert into USERS (login, name, id) values (?, ?, ?)

Hibernate: insert into autos (name, user\_id, aid) values (?, ?, ?)

Hibernate: insert into autos (name, user\_id, aid) values (?, ?, ?)

Hibernate:

select user0\_.id as id1\_0\_1\_,

user0\_.login as login2\_0\_1\_,

user0\_.name as name3\_0\_1\_,

autos1\_.user\_id as user\_id3\_0\_3\_,

autos1\_.aid as aid1\_1\_3\_,

autos1\_.aid as aid1\_1\_0\_,

autos1\_.name as name2\_1\_0\_,

autos1\_.user\_id as user\_id3\_1\_0\_

from USERS user0\_

left outer join autos autos1\_

on user0\_.id=autos1\_.user\_id

where user0\_.id=?

user1 : User {id = 50, login = 'ivan, name = 'Иван',

autos =[Auto {id = 55, name = 'Volvo, user\_id = 50},

Auto {id = 56, name = 'Skoda, user\_id = 50}]}

user2 : User {id = 51, login = 'serg, name = 'Сергей',

autos =[]}

В [продолжении](http://java-online.ru/hibernate-load.xhtml) статьи рассмотрен вопрос чтения объектов с фильтрацией и без фильтрации.

**Удаление связанных сущностей**

Наличие или отсутствие связанной сущности в базе данных определяет способ удаления. Если связанная сущность отсутствует, то можно использовать оператор DELETE в HQL-запросе объекта Query. Но если сущность содержит связанный объект в таблице БД, то при выполнении транзакции удаления с использованием объекта Query будет вызвано соответствующее исключение. Удаление связанных сущностей необходимо выполнять с использованием объекта сессии Session. Подробнее об этом представлено при описании оператора DELETE в [HQL](http://java-online.ru/hibernate-hql.xhtml#delete)-запросе.

# Hibernate Query Language (хотя лучше для переносимости проекта использовать JPQL)

Опубликовано [**Июнь 29, 2016**](https://easyjava.ru/data/hibernate/hibernate-query-language/) автор [**EasyJava**](https://easyjava.ru/author/akashihi/) — [**2 комментария ↓**](https://easyjava.ru/data/hibernate/hibernate-query-language/#comments)

[Hibernate](https://easyjava.ru/tag/hibernate/) Query Language это аналог SQL в мире [Hibernate](https://easyjava.ru/tag/hibernate/), ориентированный на запросы не к таблицам, а к классам. Идеологически он очень похож на [Java Persistence Query Language](http://easyjava.ru/data/jpa/java-persistence-query-language/), так что любой JPQL запрос является одновременно и корректным [HQL](https://easyjava.ru/tag/hql/) запросом. Обратное может быть не верно.

Все запросы в данном примере основываются на модели данных использованной ранее в [примере JPQL](http://easyjava.ru/data/jpa/java-persistence-query-language/)

# Простые запросы

HQL запрос всегда начинается с получения объекта Query из Session вызовом метода createQuery(), в который передаётся текст запроса:

|  |  |
| --- | --- |
| 1  2  3 | session.createQuery("from Person")    .list()    .forEach(System.out::println); |
| 1 | Person{firstName='Test', lastName='Testoff', dob=2016-06-28, passport=Passport{series='AS', no='123456', issueDate=2016-06-28, validity=P20Y, owner=Testoff}, primaryAddress=Address{city='Kickapoo', street='Main street', building='1', tenants=Test}, workingPlaces=[Company{name='Acme Ltd', workers=Test}]} |

Запрос from Person  буквально означает «дай-ка мне сущностей класса Person без каких-либо ограничений». У объекта Query обычно вызывают метод list(), чтобы получить список выбранных сущностей или uniqueResult(), в случае, если сущность предполагается одна. При этом uniqueResult() проверяет, что запрос возвращает ровно одну сущность и если нет, кидает исключение NonUniqueResultException.

В HQL можно выбирать не просто объекты, но и конкретные поля объектов:

|  |  |
| --- | --- |
| 1  2  3  4 | System.out.println(    session.createQuery("select passport from Person ")      .setTimeout(100)      .uniqueResult()); |
| 1 | Passport{series='AS', no='123456', issueDate=2016-06-28, validity=P20Y, owner=Testoff} |

В данном примере я говорю Hibernate  «А дай-ка мне все поля passport из сущностей класса Person без каких-либо ограничений» и в ответ мне возвращаются именно сущности Passport, а не Person, даже не смотря на from Person.

Кроме того, вызывая методы объекта Query можно настраивать, как именно будет выполняться запрос. В примере выше я указал, что запрос должен выполниться не более чем за 100 секунд. Кроме таймаута можно настроить блокировки запроса, использованием запросом кэша, пейджинг с помощью setFirstResult()/setMaxResults()  и так далее. Обо всём этом я расскажу в соответствующих статьях.

# Условия и параметры

Запросы вида «выбери мне свойство» прикольные, но бесполезные. Полезнее было бы по этим свойствам фильтровать:

|  |  |
| --- | --- |
| 1  2  3  4  5 | ScrollableResults passports = session.createQuery("from Passport as p where p.owner.lastName='Testoff'") //найди в сущности Passport сущность owner, поле lastName которой равно Testoff    .scroll();  while (passports.next()) {    System.out.println(passports.get()[0]);  } |
| 1 | Passport{series='AS', no='123456', issueDate=2016-06-28, validity=P20Y, owner=Testoff} |

Этот запрос означает «найди мне все сущности класса Passport, в поле owner которых хранятся сущности, поле lastName которых равно Testoff». Практически как обычный SQL, только в условиях пишутся не имена столбцов в таблицах, а имена полей в классах.

В запросе выше показан вариант загрузки данных не целиком, а по одному, аналогично [JDBC Scrollable ResultSet](http://easyjava.ru/java/jdbc-resultset-i-rowset/). Метод scroll() возвращает объект особого типа,  ScrollableResults, по которому можно перемещаться и получать данные из базы данных по мере перемещения. Чаще всего ScrollableResults реализован с использованием курсоров на стороне базы данных или ещё какого-либо механизма, позволяющего получать результаты запроса по частям. Объект ScrollableResults должен быть либо закрыт явно, вызовом собственного метогда close() (он реализует Closable), либо вызовом Hibernate.close(), либо будет закрыт неявно при потере persistence context.

Последний вариант получения данных из объекта Query, это получение их в форме итератора:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | Iterator passportIterator = session.createQuery("from Passport as p where p.owner.lastName like :name")    .setString("name", "Te%")    .iterate();  while (passportIterator.hasNext()) {    System.out.println(passportIterator.next());  }  Hibernate.close(passportIterator); |
| 1 | Passport{series='AS', no='123456', issueDate=2016-06-28, validity=P20Y, owner=Testoff} |

В запросе выше показано, что в HQL поддерживаются и обычные SQL выражение, такие как is null, between, в данном случае like и так далее. Параметры в запросах поддерживаются автоматически и не требуют особого обращения, [в отличие от jdbc](http://easyjava.ru/data/jdbc/jdbc-statements/). Достаточно написать имя параметра, предварённое двоеточием, и задать его значение методом setParameter().

Вызов iterate() концептуально не отличается от вызова list(): в обоих случаях данные вначале будут загружены в память, список или итератор будет полностью сконструирован и только после этого его вернут. Только scroll() и ScrollableResults позволяют проходит по возвращённым данным без их полной загрузки. Разумеется во всех случаях, и со scroll(), и с list(), и с iterator() и даже с uniqueResult() работает [отложенная загрузка полей](http://easyjava.ru/data/hibernate/upravlenie-sushhnostyami-v-hibernate/), так что возвращённый объект может быть не полностью загружен до обращения к его полям.

Возвращаясь к параметризованным запросам, Hibernate поддерживает как [JPA стиль позиционных параметров](http://easyjava.ru/data/jpa/java-persistence-query-language/), так и [JDBC стиль](http://easyjava.ru/data/jdbc/jdbc-statements/). Последний, впрочем, не рекомендуется использовать:

|  |  |
| --- | --- |
| 1  2  3  4 | session.createQuery("Select p from Person as p, IN(p.workingPlaces) as wp where wp.name = ?")    .setParameter(0, "Acme Ltd")    .list()    .forEach(System.out::println); |
| 1  2 | WARN: [DEPRECATION] Encountered positional parameter near line 1, column 104 in HQL: [Select p from ru.easyjava.data.hibernate.entity.Person as p, IN(p.workingPlaces) as wp where wp.name = ?].  Positional parameter are considered deprecated; use named parameters or JPA-style positional parameters instead.  Person{firstName='Test', lastName='Testoff', dob=2016-06-28, passport=Passport{series='AS', no='123456', issueDate=2016-06-28, validity=P20Y, owner=Testoff}, primaryAddress=Address{city='Kickapoo', street='Main street', building='1', tenants=Test}, workingPlaces=[Company{name='Acme Ltd', workers=Test}]} |

В коде выше показано, что нет необходимости чётко указывать тип передаваемого параметра, как это было сделано в предыдущем примере с помощью setString(). Метод setParameter() готов принять параметр любого типа, но ценой проверки этого типа во время исполнения, а не во время компиляции.

# Именованные запросы (*есть еще хранимые на сервере* процедуры)

В Hibernate, помимо обычных HQL запросов, предусмотрены и статические именованные запросы, которые задаются отдельно, обычно поближе к сущности:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @NamedQueries({      @NamedQuery(          name = "findCompaniesWithWorkerPassport",          query = "Select c from Company as c, IN(c.workers) as w where w.passport.series = :series")    }) |
| 1  2  3  4 | session.getNamedQuery("findCompaniesWithWorkerPassport")    .setParameter("series", "AS")    .list()    .forEach(System.out::println); |

Такие запросы можно задавать и в XML описаниях сущностей:

|  |  |
| --- | --- |
| 1  2  3 | <query name="findCompaniesWithWorkerPassport">          <![CDATA[Select c from Company as c, IN(c.workers) as w where w.passport.series = :series]]>  </query> |
| 1 | Company{name='Acme Ltd', workers=Test} |

Именованные запросы позволяют повторно использовать один и тот же запрос в разных местах кода, тем самым уменьшая количество повторений в программе и упрощая его редактирование в будущем.  Кроме того, отделение кода запроса от места его исполнения позволяет последнему не заботиться вовсе о том, какой запрос фактически исполняется, уменьшая тем самым связность кода.

Для создания именованного запроса используется метод getNamedQuery() объекта Session, который принимает в себя имя запроса. В остальном с именованным запросом можно обходиться как и с обычным HQL запросом.

Код примера доступен на [github](https://github.com/EasyJavaRu/data-hibernate-hql).

# Hibernate SQL запросы

HQL запросы и прямая загрузка сущностей, это очень здорово, но хорошо бы иметь озможность и выполнять запросы напрямую, используя всю мощь [SQL](https://easyjava.ru/tag/sql/) и вашей базы данных. Однако, такие запросы могут вернуть данные которые [Hibernate](https://easyjava.ru/tag/hibernate/) не ожидает увидеть или которые даже не отображены на существующие сущности. Поэтому для поддержки таких запросов требуется особая реализация.

# Простые (скалярные) запросы

Самый простой запрос, это запрос который возвращает какой-либо один столбец.  Я использую модель данных из [примера с HQL](http://easyjava.ru/data/hibernate/hibernate-query-language/) и посчитаю в ней количество людей в базе:

|  |  |
| --- | --- |
| 1  2  3  4 | System.out.println("No persons: "+    session.createSQLQuery("select count(id) as c from Person")      .addScalar("c", IntegerType.INSTANCE)      .uniqueResult()); |
| 1 | No persons: 1 |

На SQL запросе я подробно останавливаться не буду, лучше рассмотрю особенности Hibernate. Вызов createSQLQuery() создаёт sql запрос, равно как и createQuery() создаёт HQL запрос. Вызов addScalar("c", IntegerType.INSTANCE) сообщает Hibernate что в результате запроса вернётся простой столбец (а не объект) и что он будет типа Integer.

В скалярных запросах можно возвращать и несколько столбцов и при этом вовсе не обязательно указывать их тип. В последнем случае, конечно, разработчику придётся делать приведение типа самому:

|  |  |
| --- | --- |
| 1  2  3 | List<Object[]> passportIds = session.createSQLQuery("select id, passport\_id from Person")    .list();  passportIds.forEach(p -> System.out.println("User id: "+p[0]+" Passport id: "+p[1])); |
| 1 | User id: 3 Passport id: 1 |

В это случае возвращается список из массива «сырых» объектов, которые Hibernate и не пытается обрабатывать. Этот подход похож на [возврат результатов в Spring JDBC](http://easyjava.ru/spring/spring-data-access/zaprosy-v-spring-jdbc/).

# Запросы сущностей

Сырые данные из базы это хорошо, но у нас всё таки объектный язык. Результаты SQL запроса можно преобразовать в сущность, выбрав поля, которые её составляют.

|  |  |
| --- | --- |
| 1  2  3  4 | session.createSQLQuery("select p.\* from Passport as p, Person as pe where p.id=pe.passport\_id and  pe.lastName='Testoff'")    .addEntity(Passport.class)    .list()    .forEach(System.out::println); |
| 1 | Passport{series='AS', no='123456', issueDate=2016-07-13, validity=P20Y} |

Вызов addEntity() указывает Hibernate, какую сущность из данных запроса надо построить. Имена столбцов в ответе должны совпадать с именами, заданными при отображении.

Разумеется, с SQL запросами (и с скалярными и с запросами сущностей) можно использовать стандартный функционал — параметры, пейджинг и т.д.:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | session.createSQLQuery("select \* from Person as p join Passport as pa on p.passport\_id=pa.id and p.lastName = :name")    .addEntity("p",Person.class)    .addJoin("pa", "p.passport") //говорит с каким полем в главной таблице объединены    .setResultTransformer( Criteria.ROOT\_ENTITY )    .setString("name", "Testoff")    .list()    .forEach(System.out::println); |

Запросы выше кроме использования параметров показывает, как загружать связанные объъекты. Вызов addEntity() указывает Hibernate, какой объект мы загружаем (корневой объект), вызов(ы) addJoin() говорят Hibernate, какие связанные объекты загружаются вместе с корневым и как они к нему относятся.

# Преобразование результатов

Вызов setResultTransformer() в предыдущем примере говорит Hibernate, что мы хотим получить конкретно экземпляр корневого объекта, а не набор загруженных данных. Но это не единственное и не главное предназначение setResultTransformer(). Если говорить обще, этот вызов позволяет задать код, который будет преобразовывать загруженные из базы данные.

Один из вариантов использования этого функционала — загрузка данных во временные объекты, которые даже не имеют отображения (DTO — data transfer objects)

Для начала я создам такой DTO:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | /\*\*  \* Company name extractor  \*/  @AllArgsConstructor  @NoArgsConstructor  public class CompanyNameDTO {      /\*\*       \* The name.       \*/      @Getter      @Setter      private String name;  } |

Для использования DTO с ResultTransfomer он должен иметь конструктор по умолчанию и сеттеры на полях. Затем мы трансформируем результат запроса в DTO:

|  |  |
| --- | --- |
| 1  2  3  4 | session.createSQLQuery("select name as \"name\"from Company")    .setResultTransformer(Transformers.aliasToBean(CompanyNameDTO.class))    .list()    .forEach(System.out::println); |
|  | CompanyNameDTO(name=Acme Ltd) |

Выражение “ *name as \"name\"* “ в данном случае критично важно, так как имя столбца в ответе должно совпадать с именем поля в DTO.

# Именованные запросы

Именованные запросы с SQL работают так же, как и с HQL и даже используют тот же вызов. Но определение запроса меняется: во-первых аннотация теперь называется @NamedNativeQuery, во-вторых необходимо явно задавать тип возвращаемой сущности:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @NamedNativeQueries({    @NamedNativeQuery(      name = "findCompanyWithName",      query = "select \* from Company where name like :name",      resultClass = Company.class    )  })  public class Company extends AbstractIdentifiableObject {  } |
| 1  2  3  4 | session.getNamedQuery("findCompanyWithName")    .setParameter("name", "Ac%")    .list()    .forEach(System.out::println); |
|  | Company{name='Acme Ltd', workers=Test} |

В остальном именованные SQL запросы ведут себя так же, как и именованные HQL запросы и поддерживают тот же самый функционал.

Кроме того, именованные SQL запросы поддерживают и DTO:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | @NamedNativeQueries({  @NamedNativeQuery(     name = "findCompanyNameOnly",     query = "select name from Company",     resultSetMapping = "company\_name\_dto"  )  })  @SqlResultSetMapping(  name = "company\_name\_dto",  classes = @ConstructorResult(     targetClass = CompanyNameDTO.class, //где CompanyNameDTO это класс временного (DTO) объекта, написанный выше в  Преобразовании результатов     columns = {       @ColumnResult(name="name")     }  )  )  public class Company extends AbstractIdentifiableObject {  } |

|  |  |
| --- | --- |
| 1  2  3 | session.getNamedQuery("findCompanyNameOnly")    .list()    .forEach(System.out::println); |
| 1 | CompanyNameDTO(name=Acme Ltd) |

Для использования DTO и именованных SQL запросов требуется определить соответствующее отображение аннотацией @SqlResultSetMapping и связать его с именованным запросом. Очевидно, что одно отображение может быть использовано в нескольких запросах.

Для того, чтобы DTO можно быть использовать с @ConstructorResult, класс DTO должен иметь конструктор, инициализирующий его значения.

Код примера доступен на [github](https://github.com/EasyJavaRu/data-hibernate-sql).

# Hibernate Criteria API

**Начиная с версии 5.2 Hibernate Criteria API объявлен deprecated и рекомендуется к использованию JPA Criteria**

**JPA Criteria**

[JPQL запросы](http://easyjava.ru/data/jpa/java-persistence-query-language/) штука весьма удобная, почти такая же, как и SQL, только с классами и атрибутами. Но, как и у всякой удобной вещи, у неё есть недостатки, даже два:

* Запросы жёстко определяются на стадии компиляции и во время исполнения их не изменить.
* Запросы совсем никак не связаны с реальными сущностями и если сущность изменяется, то никто не скажет, что запрос больше неверен. До тех пор, пока его не попытаются выполнить.

Хорошая новость — в [JPA](https://easyjava.ru/tag/jpa/) есть механизм, который решает обе эти проблемы.

Это база с полями и подполями для запросов:

Person{

firstName='Test',

lastName='Testoff',

dob=2016-06-28,

passport=Passport{

series='AS',

no='123456',

issueDate=2016-06-28,

validity=P20Y,

owner=Testoff

},

primaryAddress=Address{

city='Kickapoo',

street='Main street',

building='1',

tenants=Test

},

workingPlaces= [Company{

name='Acme Ltd',

workers=Test

}]

}

**Criteria API и программно создаваемые запросы.**

Примером запросов, создаваемых программно, может служить любой фильтр в любом приложении, который позволяет фильтровать данные по нескольким полям. Если рассматривать модель данных из примера с [отношениями между сущностями](http://easyjava.ru/data/jpa/jpa-i-svyazi-mezhdu-obektami/), то можно представить себе фильтр людей по:

* Серии и номеру паспорта
* Адресу проживания
* Месту работы
* Имени

Причём фильтроваться люди могут по любой комбинации этих полей, например по адресу проживания и месту работы или по имени и адресу проживания.

Вариантов реализации такого фильтра несколько. Можно сделать 16 различных запросов и в зависимости от того, какие значения фильтра установлены, выбирать подходящий запрос. Я даже не буду объяснять, почему это плохое решение и почему так делать никогда нельзя

Можно собирать запрос из составляющих, пользуясь тем, что запрос это строка. Недостатков у этого подхода тоже много: абсолютно нечитаемый код запроса в итоге, высокая вероятность составить кривой запрос, сложность поддержания генератора запроса в актуальном состоянии и т.д.

Наконец третий и наиболее правильный вариант, это использование программно определяемых запросов и JPA Criteria API.

Аналог JPQL запросу «**from Person**»:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | EntityManagerFactory emf = Persistence.createEntityManagerFactory(“my-pu”);  EntityManager em = emf.createEntityManager();  em.getTransaction().begin();  CriteriaBuilder **cb** = em.getCriteriaBuilder();    CriteriaQuery<Person> **cr** = cb.createQuery(Person.class);  Root<Person> **root** = cr.from(Person.class);  cr.select(root);  em.createQuery(cr).getResultList().forEach(System.out::println); |

тут мы:

1. (стр.**1-2**) [создаём EntityManager](http://easyjava.ru/data/jpa/jpa-entitymanager-upravlyaem-sushhnostyami/), где "my-pu" это самостоятельно придуманное в файле "persistence.xml" в блоке <persistence-unit> имя блока-сохраняемости
2. (стр.**3**) открываем транзацию.
3. (стр.**4**) создаём CriteriaBuilder (“**cb**”), который будет строить объекты запросов.
4. (стр.**5**) с помощью CriteriaBuilder создаём CriteriaQuery (“**cr**”), который параметризуется типом, который этот запрос возвращает.
5. (стр.**6**) создаём корневой объект (“**root**”), от которого производится обход дерева свойств при накладывании ограничений или указании что выбирать.
6. (стр.**7**) говорим, что же мы хотим выбрать.
7. (стр.**8**) запрос отправляется в EntityManager, где и выполняется как обычно.

Все шаги, перечисленные выше, являются обязательными для создания запроса с помощью Criteria API. Важно понимать, что корневой объект указывает JPQL, откуда будут браться данные, а CriteriaQuery указывает тип возвращаемых данных. И типы Root  и CriteriaQuery  могут отличаться.

Аналог JPQL запросу «**select passport from Person**» (тут мы из сущности **Person** хотим получить поле-коллекцию **passport**):

|  |  |
| --- | --- |
| 1  2  3  4 | CriteriaQuery<Passport> cr = cb.createQuery(Passport.class);  Root<Person> root = cr.from(Person.class);  cr.select(root.get("passport"));  em.createQuery(cr).getResultList().forEach(System.out::println); |

Этот запрос показывает, что класс, из которого запрашиваются данные и класс, который вернёт запрос, могут быть разными.

Пора переходить к самому интересному — ограничениям. Ограничения накладываются на CriteriaQuery и задаются программно, в виде вызова соответствующих функций.

Аналог JPQL запроса « **from Passport as p where p.owner.lastName='Testoff**'» (Этот запрос означает «найди мне все сущности класса Passport, в поле owner которых хранятся сущности, поле lastName которых равно Testoff»):

|  |  |
| --- | --- |
| 1  2  3  4 | CriteriaQuery<Passport> cr = cb.createQuery(Passport.class);  Root<Passport> root = cr.from(Passport.class);  cr.select(root).where(cb.equal(root.get("owner").get("lastName"), "Testoff"));  em.createQuery(cr).getResultList().forEach(System.out::println); |

Ключевое изменение здесь конечно же:

|  |  |
| --- | --- |
| 1 | cr.where(cb.equal(root.get("owner").get("lastName"), "Testoff")); |

Вначале из корневого объекта (“root”) ссылаемся на поле с объектами “owner”, потом из него на поле “lastName” и требуем его со значением "Testoff". Метод equal() создаёт экземпляр Predicate, сравнивающий значения полученной ссылки на поле  lastName и строки *«Testoff».*Этот Predicate отдаётся в метод CriteriaQuery.where() и теперь запрос вернёт только те объекты, для которых предикат истинен.

Различных предикатов в CriteriaBuilder запасено много и рассматривать каждый смысла наверное нет, поэтому я приведу всего лишь один пример с другим предикатом.

Аналог JPQL запросу « **from Passport as p where p.owner.lastName like 'Te%**'».

|  |  |
| --- | --- |
| 1  2  3  4 | CriteriaQuery<Passport> cr = cb.createQuery(Passport.class);  Root<Passport> root = cr.from(Passport.class);  cr.select(root).where(cb.like(root.get("owner").get("lastName"), "Te%"));  em.createQuery(cr).getResultList().forEach(System.out::println); |

Кстати, поскольку данные отправляются в предикаты как параметры функций, специальная поддержка параметров запроса становится излишней, всё происходит само по себе.

Обращение к связанным сущностям требует дополнительных усилий. Чтобы написать аналог « **Select p from Person as p, IN(p.workingPlaces) as wp where wp.name = 'Acme Ltd'**», необходимо явно указать связь между сущностями Company и Person:

|  |  |
| --- | --- |
| 1  2  3  4  5 | CriteriaQuery<Person> cr = cb.createQuery(Person.class);  Root<Person> root = cr.from(Person.class);  Join<Person, Company> company = root.join("workingPlaces");  cr.select(root).where(cb.equal(company.get("name"), "Acme Ltd"));  em.createQuery(cr).getResultList().forEach(System.out::println); |

Для фильтрации по полям коллекции мы вначале создаём объект связи, в котором указываем, кто с кем связан и по какому полю, а потом уже используется объект связи как источник ссылок на поля и создаём предикат для этих ссылок.

**Metamodel и типобезопасность.**

Все примеры выше решают проблему с программным созданием запросов, но всё ещё бессильны перед изменениями сущностей. В самом деле, изменю я в сущности Person  поле workingPlaces на jobs и развалится последний запрос. И не узнаю я, что он развалится, пока не попробую его исполнить.

Metamodel решает эту проблему, создавая специальные описательные классы, которые используются в Criteria API вместо имён полей.

Сам Metamodel класс выглядит примерно вот так:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @StaticMetamodel(Company.class)  public abstract class Company\_ extends AbstractIdentifiableObject\_ {    public static volatile SingularAttribute<Company, String> name;  public static volatile CollectionAttribute<Company, Person> workers;    } |

В Metamodel классе описываются, какие поля присутствуют в сущности, какого они типа, коллекция это или нет и т.д. Для каждой сущности создаётся свой класс Metаmodel.

Создаются классы Metamodel разумеется не вручную. То есть можно их и вручную создать, но тогда пропадает автоматичность проверки и теряется смысл всей этой затеи. Обычно же классы Metamodel генерируются на этапе компиляции тем или иным методом. Конкретная реализация генерации зависит от конкретной реализации JPA и может меняться.

Поскольку мы рассматриваем [Hibernate](https://easyjava.ru/tag/hibernate/) реализацию JPA, то генератор будет тоже от [Hibernate](https://easyjava.ru/tag/hibernate/). Сам генератор реализован как annotation processor и может быть подключен в maven проект следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <dependency>    <groupId>org.hibernate</groupId>    <artifactId>hibernate-jpamodelgen</artifactId>    <version>${hibernate.version}</version>    <scope>provided</scope>  </dependency> |

Генерация классов Metamodel в этом случае будет происходить автоматически.

У меня сработал такой блок:

<plugin>  
 <groupId>org.bsc.maven</groupId>  
 <artifactId>maven-processor-plugin</artifactId>  
 <executions>  
 <execution>  
 <id>process</id>  
 <goals>  
 <goal>process</goal>  
 </goals>  
 <phase>generate-sources</phase>  
 <configuration>  
 <processors>  
 <processor>org.hibernate.jpamodelgen.JPAMetaModelEntityProcessor</processor>  
 </processors>  
 </configuration>  
 </execution>  
 </executions>  
 <dependencies>  
 <dependency>  
 <groupId>org.hibernate</groupId>  
 <artifactId>hibernate-jpamodelgen</artifactId>  
 <version>${hibernate.version}</version>  
 </dependency>  
 </dependencies>  
 </plugin>  
</plugins>

Использование Metamodel в JPA Criteria достаточно очевидно: везде, где можно использовать имя поля, можно использовать Metamodel:

|  |  |
| --- | --- |
| 1  2  3  4 | CriteriaQuery<Passport> cr = cb.createQuery(Passport.class);  Root<Person> root = cr.from(Person.class);  cr.select(root.get(Person\_.passport));  em.createQuery(cr).getResultList().forEach(System.out::println); |

Или более сложный случай:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9. | CriteriaQuery<Company> cr = cb.createQuery(Company.class);  Root<Company> root = cr.from(Company.class);  Join<Company, Person> person = root.join(Company\_.workers);  cr.select(root).where(cb.equal(person.get(Person\_.passport).get(Passport\_.series), "AS"));  em.createQuery(cr)    .setFirstResult(0)    .setMaxResults(10)    .getResultList()    .forEach(System.out::println); |

Так как Metamodel генерируется автоматически при компиляции, то комплятор знает, какие поля вы передаёте в Criteria API и какие у них типы. Следовательно, как только будет внесено изменение, ломающее запрос, компиляция этих запросов тут же провалится и разработчик сразу увидит, где ещё надо внести изменения.

**Примеры Join**

@Entity

public class Address {

@Id @GeneratedValue

private Long id;

private String street;

private String city;

private String zipCode;

@ManyToOne

private User owner;

}

@Entity

public class User {

@Id @GeneratedValue

private Long id;

private String firstName;

private String lastName;

private int credits;

private Login login;

@OneToMany(cascade = CascadeType.PERSIST, mappedBy = "owner")

private Set<Address> addresses = new HashSet<Address>();

}

@Transactional

public class UserManager {

//get all users living in Paris

@Transactional

public List<User> getUsersFromParis() {

EntityManager em = lazyEM.get();

CriteriaBuilder cb = em.getCriteriaBuilder();

final CriteriaQuery<User> cr = cb.createQuery( User.class );

final Root<User> root = cr.from( User.class );

final SetJoin<User,Address> a = root.join(User\_.addresses ); // мы получили объект “a” из которого

// уже будем выбирать поля сущности “Address”

root.fetch(User\_.addresses ); //это доп.настройка, говорящая, что поля разных сущностей

// должны “выгребаться” одним запросом

cr.select(root).distinct( true ).where( cb.equal( a.get( Address\_.city ), "Paris" ) );

final TypedQuery<User> typedQuery = em.createQuery( cr );

typedQuery.setFirstResult( 0 ).setMaxResults( 20 );

final List<User> resultList = typedQuery.getResultList();

return resultList;

}

**Пример с каскадом из 3-х связанных таблиц**

**(получаем столбец из нижней таблицы)**

public static Specific<JpaAction> hasArtifact(final String controllerId, final String sha1Hash) {

return (actionRoot, query, criteriaBuilder) -> {

Join<Action, DistrSet> **dsJoin** = **root**.join(Action\_.distrnSet);

SetJoin<DistrSet, SoftwareModule> **modulesJoin** = **dsJoin**.join(DistrSet\_.modules);

ListJoin<SoftwareModule, Artifact> **artifactsJoin** = **modulesJoin**.[join](https://www.codota.com/code/java/methods/javax.persistence.criteria.SetJoin/join)(SoftwareModule\_.artifacts);

return criteriaBuilder.and

(criteriaBuilder.equal(**artifactsJoin**.get(JpaArtifact\_.sha1Hash), sha1Hash),

criteriaBuilder.equal(**root**.get(JpaAction\_.target).get(JpaTarget\_.controllerId),

controllerId));

};

}

**Пагинация**

Пагинация не относится ни к Criteria API, ни к JPQL, а работает с любыми запросами, которые создаёт метод createQuery(). Собственно речь идёт даже не о пагинации как таковой, хотя её обычно именно этими методами и делают, а о управлении количеством возвращаемых объектов и смещением от начала.

Все запросы, возвращаемые createQuery(), имеют два метода:**setFirstResult()** и **setMaxResults()**.

Первый метод, setFirstResult(), говорит, сколько результатов надо отбросить от начала списка результатов при выполнении запроса, задавая смещение от начала списка.

Второй метод, setMaxResults() ограничивает количество возвращаемых объектов. Аналог этих методов в SQL — операторы LIMIT и OFFSET.

Код примера доступен на [github](https://github.com/EasyJavaRu/data-jpa-hibernate-criteria).

**Первичные ключи в Hibernate**

Каждая [Hibernate](https://easyjava.ru/tag/hibernate/) сущность должна иметь идентификатор, который её однозначно идентифицирует. В мире SQL подобный идентификатор называется, с некоторыми допущениями,*первичный ключ.*В качестве такого идентификатора можно использовать примитивные типы и их обёртки, строки, BigDecimal/BigInteger, даты и т.д. Hibernate требует, чтобы каждый такой идентификатор был:

* уникальным (UNIQUE) — то есть однозначно идентифицировал строку в таблице.
* Имеющим значение (NOT NULL) — то есть идентификтор не может быть null, а в случае составного идентификатора ни какое из его полей не может быть null.
* Неизменным (IMMUTABLE) — значение идентификатора определяется при создании записи в базе и в последствии никогда не изменяется.

**Естественный ключ или суррогатный ключ**

Поле идентификатора помечается аннотацией@Id. Например, если у нас в модели данных есть сущность «*Компания*» (а я использую модель данных из [примера JPQL](http://easyjava.ru/data/jpa/java-persistence-query-language/), в которой эта сущность есть) у которой есть имя компании и имя это соответствует требованиям к идентификаторам, то есть имеет значение, не изменяется и уникально, то это имя можно использовать как идентификатор.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | @Entity  public class Company {      @Id      @Getter      @Setter      private String name;        @Getter      @Setter      @ManyToMany(mappedBy = "workingPlaces")      private Collection<Person> workers;  } |

Такой идентификатор называется *естественным*. Он вытекает непосредственно из модели данных приложения и, обычно, весьма хорошо в неё вписывается. Теория так же говорит нам, что для любой таблицы/сущности можно сформировать естественный ключ.

С другой стороны, естественный ключ может быть и неудобным в использовании. Пример выше не позволяет создать компанию с тем-же самым именем в будущем, когда существующая компания уже прекратит своё существование. Поэтому зачастую используются *суррогатные* ключи, которые не связаны явно с моделью данных приложения, а либо порождаются из неё, либо создаются каким-либо другим способом. Типичное решение — добавление к сущности дополнительного целочисленного поля.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | @Entity  public class Company extends AbstractIdentifiableObject {      @Id      @Getter      @Setter      private Long id;        @Getter      @Setter      private String name;        @Getter      @Setter      @ManyToMany(mappedBy = "workingPlaces")      private Collection<Person> workers;  } |

Суррогатные и натуральные ключи можно использовать параллельно:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | @Entity  public class Company {      @Id      @GeneratedValue      @Getter      @Setter      private Long id;        @NaturalId      @Getter      @Setter      private String name;        @Getter      @Setter      @ManyToMany(mappedBy = "workingPlaces")      private Collection<Person> workers;  } |

В этом случае поле id будет первичным ключом, а name — естественным ключом. С точки зрения базы это будет просто ещё один индекс, а вот Hibernate будет знать, что это тоже идентификатор и будет использовать это знание для оптимизации. Кроме того, по естественному ключу можно делать запросы:

|  |  |
| --- | --- |
| 1  2  3  4 | Company c  = session      .byNaturalId( Company.class )      .using( "name", "Acme Ltd." )      .load( ); |

**Простые и составные идентификаторы**

Все первичные ключи, которые были в примерах выше, являются *простыми* ключами, то есть состоящими из одного столбца. Но первичный ключ может быть и *составным*, то есть состоять из более чем одного столбца. Например вы можете решить идентифицировать пользователей по паспорту, то есть серии и номеру. Очевидно, что удобнее было бы хранить серию и номер в разных полях, но тогда гарантировать уникальность значений в каждом столбце не получится и на помощь приходит составной первичный ключ.

Hibernate, разумеется, составные ключи поддерживает, причём двумя разными методами, но требует при это дополнительной работы. В первую очередь необходимо определить класс ключа:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | @EqualsAndHashCode  @ToString  public class PassportKey implements Serializable {        static final long serialVersionUID = 1L;        @Getter      @Setter      private String series;        @Getter      @Setter      private String n;    } |

К классу ключа предъявляются некоторые требования:

* Класс должен быть public
* У класса должен быть публичный конструктор по умолчанию.
* Класс должен (корректно) реализовывать собственные equals()  и hashCode()
* Класс должен реализовывать Serializable

Первый метод использования составного ключа включает класс ключа целиком в класс сущности:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | @Entity  public class Passport {  **@EmbeddedId**  **private PassportKey key;**        @Getter      @Setter      private LocalDate issueDate;        @Getter      @Setter      private Period validity;        @Getter      @Setter      @OneToOne(optional = false, mappedBy = "passport")      private Person owner;    **@EqualsAndHashCode**  **@ToString**  **@Embeddable**  **public class PassportKey implements Serializable {**    **static final long serialVersionUID = 1L;**    **@Getter**  **@Setter**  **private String series;**    **@Getter**  **@Setter**  **private String n;**        }  } |

@EmbeddedId  указывает на поле составного первичного ключа, а @Embeddable объявляет класс составным ключом.

Второй вариант использования оставляет поля первичного ключа непосредственно в классе сущности, а класс составного ключа служит лишь для поддержки:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | @Entity  @IdClass(Passport.PassportKey.class)  public class Passport {      @Id  *Опять пишем то, что написанно в классе составного ключа (ниже)*      @Getter      @Setter      private String series;        @Id      @Getter      @Setter      private String n;        @Getter      @Setter      private LocalDate issueDate;        @Getter      @Setter      private Period validity;        @Getter      @Setter      @OneToOne(optional = false, mappedBy = "passport")      private Person owner;        @EqualsAndHashCode      @ToString      public class PassportKey implements Serializable {            static final long serialVersionUID = 1L;            @Getter          @Setter          private String series;            @Getter          @Setter          private String n;        }  } |

**Генерируемые автоматически и создаваемые вручную ключи**

Значения естественных ключей создаются естественным  образом при создании экземпляра сущности. А вот суррогатные ключи надо как-то заполнять самому и брать для них откуда-то уникальные значения, что может быть непростым делом в условиях параллельной обработки запросов.

В Hibernate на этот случай предсмотрены механизмы автоматической генерации значений суррогатных ключей, которые включается аннотацией **@GeneratedValue**. Эта аннотация без указания в скобках стратегии используется когда схема базы данных создаётся Hibernate по описанию сущностей, при этом стратегия генерации значений и необходимая для этой стратегии оснастка создаются автоматически, исходя из возможностей базы данных.

???При этом когда я изначально использовал для БД MySQL установленную в скобках после @GeneratedValue стратегию IDENTITY работало нормально. Потом захотел поменять генерацию ключа с числового на UUID и тип значения поля ключа с Long поменял на UUID и стратегию из скобок после @GeneratedValue удалил (чтобы определяло стратегию автоматически). А вот когда решил вернуться с UUID опять на генерацию числового ключа (со стратегией в скобках IDENTITY) и поменял назад UUID на Long, то работать перестало и чтобы заработало пришлось удалить из скобок стратегию IDENTITY и дать возможность системе самостоятельно определять стратегию как и при UUID. Т.е. возможно если система определяла стратегию самостоятельно, то потом переключиться на ручное указание стратегии не получится???

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | @Entity  public class Company extends AbstractIdentifiableObject {      @Id      @GeneratedValue      @Getter      @Setter      private Long id;        @Getter      @Setter      private String name;        @Getter      @Setter      @ManyToMany(mappedBy = "workingPlaces")      private Collection<Person> workers;  } |

Если же схема данных создаётся вручную, то описание таблиц должно совпадает с выбраной стратегией (и поддерживаться базой данных).

Hibernate поддерживает три страгегии генерации значений суррогатного ключа.

Первая стратегия, GenerationType.**IDENTITY**, работает с базами, у которых есть специальные IDENTITY поля, например с MySQL или DB2. В этом случае, для примера выше, таблицу необходимо было бы создавать как:

|  |  |
| --- | --- |
| 1  2  3  4 | CREATE TABLE COMPANY (    ID BIGINT PRIMARY KEY AUTO\_INCREMENT    -- Other fields  ); |

Вторая стратегия, GenerationType.**SEQUENCE**, использует встроенный в базы данных, такие как PostgreSQL или Oracle, механизм генерации последовательных значений (sequence). Использование этого генератора требует как создания отдельной sequence в базе данных:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | CREATE TABLE COMPANY(    ID BIGINT PRIMARY KEY    -- Other fields  );    CREATE SEQUENCE HIBERNATE\_SEQUENCE START WITH 1 INCREMENT BY 1 NOCACHE NOCYCLE; |

Так и задания имени этой sequence в описании ключа:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @Id  @SequenceGenerator(name = "hibernateSeq", sequenceName = "HIBERNATE\_SEQUENCE")  @GeneratedValue( strategy = GenerationType.SEQUENCE, generator = "hibernateSeq")  @Getter  @Setter  private Long id; |

Если явно не конфигурировать генератор последовательных значений, HIbernate будет использовать общую последовательность HIBERNATE\_SEQUENCE для всех сущностей.

Третья стратегия, GenerationType.**TABLE**, не зависит от поддержки конкретной базой данных и хранит счётчики значений в отдельной таблице. Hibernate использует таблицу из двух столбцов, для хранения имён последовательностей и текущих значений ключа для генерации:

|  |  |
| --- | --- |
| 1  2  3  4 | create table hibernate\_sequences(      sequence\_name VARCHAR NOT NULL,      next\_val INTEGER NOT NULL  ) |
| 1  2  3  4  5 | @Id  @GeneratedValue( strategy = GenerationType.TABLE)  @Getter  @Setter  private Long id; |

Параметры табличного генератора могут быть настроены аннотацией @TableGenerator, по умолчанию используется таблица HIBERNATE\_SEQUENCES и в ней последовательность *default*

**UUID идентификаторы**

Помимо обычных типов полей, перечисленных выше, Hibernate поддерживает использование UUID в качестве идентификатора.

UUID (Universally Unique Identifier) — это стандарт идентификации, основное назначение которого, это позволить распределённым системам уникально идентифицировать информацию без центра координации. Таким образом, любой может создать UUID и использовать его для идентификации чего-либо с приемлемым уровнем уверенности, что данный идентификатор непреднамеренно никогда не будет использован для чего-то ещё. Поэтому информация, помеченная с помощью UUID, может быть помещена позже в общую базу данных, без необходимости разрешения конфликта имен.

У использования UUID есть несколько достоинств, по сравнению с обычными автогенерируемыми целочисленными ключами:

* Обеспечивает уникальность идентификаторов не только в пределах одной таблицы, что для некоторых решений может быть важно
* Позволяет генерировать идентификатор записи на клиенте, до сохранения ее в базу
* Делает практически невозможным подбор ключа в случаях, когда запись можно получить, передав ее идентификатор в какой-нибудь публичный API
* Позволяет создавать записи в нескольких репликах базы данных и объединять их в последствии
* Типо-независим

За всё хорошее, с другой стороны, надо платить. В случае UUID ключей платить приходится производительностью:

* Генерация UUID обычно заметно медленнее, чем генерация целочисленного значения
* Выборка из таблиц, связанных друг с другом по UUID, зачастую производится медленнее.

Кроме того, UUID выглядит не очень человекочитаемым, например *452079be-cb27-4ceb-b29f-991e0c31b9e0.*

Использование UUID достаточно просто:

|  |  |
| --- | --- |
| 1  2  3  4  5 | @Id  @GeneratedValue  @Getter  @Setter  private UUID id; |

Hibernate сам переключится на UUID генератор с настройками по умолчанию и будет его использовать. При необходимости можно задать и ручную конфигурацию генератора:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | @Id  @GeneratedValue(generator = "uuid")  @GenericGenerator( //аннотация для подключения собственного алгоритма генерации ключей    name = "uuid",    strategy = "org.hibernate.id.UUIDGenerator",    parameters = {      @Parameter(        name = "uuid\_gen\_strategy\_class",        value = "org.hibernate.id.uuid.CustomVersionOneStrategy"      )    }  )  @Getter  @Setter  private UUID id; |

**Собственный генератор**

Наконец, для тех кому не хватает стандартных генераторов, в Hibernate предусмотрен механизм создания собственных генераторов. Для примера я сделаю целочисленный генератор, который создаёт случайное значение:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | /\*\*  \* Generates random INT ids.  \*/  public class RandomIdGenerator implements IdentifierGenerator {    private Random r = new Random();      @Override    public Serializable generate(SharedSessionContractImplementor s, Object o)        throws HibernateException {      return r.nextInt();    }  } |

Для создания нового генератора необходимо реализовать интерфейс IdentifierGenerator и его метод generate(), который должен вернуть нагенерированное. При необходимости можно использовать текущую сессию и объект, для которого генерируется значение.

Использовать собственный генератор можно с помощью аннотации  @GenericGenerator

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @Id  @GenericGenerator(name = "randomGen", strategy = "ru.easyjava.data.hibernate.RandomIdGenerator")  @GeneratedValue(generator = "randomGen")  @Getter  @Setter  private Long id; |

# 

# Пользовательские типы в Hibernate

Из коробки [Hibernate](https://easyjava.ru/tag/hibernate/) поддерживает некий общий набор типов данных SQL и типов данных Java, а также отображений между ними. В основном в этот набор входят базовые вещи, такие как даты, строки, числа, блобы и так далее. С полным списком, а он довольно большой чтобы приводить его здесь, можно ознакомиться в [документации Hibernate](http://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/Hibernate_User_Guide.html#basic-provided).

Как я уже сказал, набор этих типов довольно общий и поддерживается большинством баз данных, с которыми работает Hibernate. С другой стороны, каждая база может иметь свои собственные, уникальные и, следовательно, неподдерживаемые типы данных. Со стороны Java, в свою очередь, тоже можно представить собственные структуры данных, которые напрямую не поддерживаются в Hibernate, но желательно иметь возможность их сохранять.

Для решения этой проблемы в Hibernate существует поддержка пользовательских типов данных. То есть можно написать, как тот или иной класс в Java должен сохраняться в тот или иной тип данных (читай столбец в таблице) в SQL базе.

Например, в [PostgreSQL](https://easyjava.ru/tag/postgresql/) существует встроенный тип данных *inet* для хранения IPv4 и IPv6 адресов. И мы хотели бы использовать этот тип данных при разработке приложения, управляющего, например,  выделением ip сетей. И у нас есть проблема: Hibernate не знает про этот тип данных ровным счётом ничего. Попробуем реализовать поддержку этого типа вручную.

**Подготовка**

В первую очередь создадим Maven проект и добавим в него Hibernate и JDBC драйвер PostgreSQL:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | <properties>    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>    <lombok.version>1.16.6</lombok.version>    <postgresql.version>9.4-1206-jdbc42</postgresql.version>    <hibernate.version>5.1.0.Final</hibernate.version>      <junit.version>4.12</junit.version>    <hamcrest.version>1.3</hamcrest.version>  </properties>    <dependencies>    <dependency>      <groupId>org.projectlombok</groupId>      <artifactId>lombok</artifactId>      <version>${lombok.version}</version>      <scope>provided</scope>    </dependency>      <dependency>      <groupId>org.postgresql</groupId>      <artifactId>postgresql</artifactId>      <version>${postgresql.version}</version>    </dependency>      <dependency>      <groupId>org.hibernate</groupId>      <artifactId>hibernate-core</artifactId>      <version>${hibernate.version}</version>    </dependency>      <dependency>      <groupId>org.hibernate</groupId>      <artifactId>hibernate-java8</artifactId>      <version>${hibernate.version}</version>    </dependency>  </dependencies> |

Вторым шагом настроим Hibernate на работу с Postgresql:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | <hibernate-configuration>        <!-- a SessionFactory instance listed as /jndi/name -->      <session-factory>            <property name="hibernate.dialect">org.hibernate.dialect.PostgreSQL94Dialect</property>          <property name="hibernate.connection.url">jdbc:postgresql://127.0.0.1/test</property>          <property name="hibernate.connection.username">test</property>          <property name="hibernate.connection.password">test</property>            <mapping class="ru.easyjava.data.hibernate.entity.Network"/>      </session-factory>    </hibernate-configuration> |

По сравнению с H2 базой, [используемой ранее](http://easyjava.ru/data/hibernate/hello-hibernate-bez-jpa/), отличия минимальные — изменился Hibernate dialect, да настройки JDBC. Так же я отключил автоматическое создание таблиц, так как нам надо создать таблицу с типом столбцов неизвестным HIbernate.

**Модель данных**

Описание одной сохраняемой в базе сети будет состоять из двух классов. В первом классе определяется собственно тип «ip сеть»:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | /\*\*  \* Describes ip network as a pair of host and mask.  \*/  @Data  public class NetworkObject implements Serializable {    /\*\*     \* Serializable requirement.     \*/    private static final long serialVersionUID = 1L;      /\*\*     \* Address part of network.     \*/    private final String address;      /\*\*     \* Bitmask part of the network.     \*/    private final short bitmask;  } |

Сеть состоит из адреса сети и маски сети. Во втором классе мы будем использовать объект сети как часть хранимой сущности:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | /\*\*  \* Network storing entity.  \*/  @Entity  @ToString  public class Network {    /\*\*     \* Entity id.     \*/    @Id    @GeneratedValue    @Getter    @Setter    private Long id;      /\*\*     \* Network data.     \*/    @Type(type = "ru.easyjava.data.hibernate.type.NetworkObjectType")    @Getter    @Setter    private NetworkObject network;  } |

Обратите внимание, что поле network встроено в сущность Network, а не связано через *один-к-одному* или иным типом связи. Кроме того, NetworkObject не является и сущностью. Это всего лишь обычный класс, наподобие BigDecimal или LocalDateTime. Аннотация @Type объясняет Hibernate, к кому обращаться, для преобразования этого класса в SQL столбец и обратно.

**Пользовательский тип**

Чтобы реализовать поддержку собственного типа данных в Hibernate, необходимо реализовать интерфейс UserType и его методы (***11 штук***):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | /\*\*  \* Mapping of NetworkObject to PGSQL cidr type.  \*/  public class NetworkObjectType implements UserType {        /\*\*       \* Reference to the type for Hibernate.       \*/      public static final NetworkObjectType INSTANCE = new NetworkObjectType();        //Methods  } |

Поле INSTANCE используется, например, для спецификации типов возвращаемых из SQL запросов. Методов же в интерфейсе UserType немало:

|  |  |
| --- | --- |
| 1  2  3  4 | @Override  public int[] sqlTypes() {    return new int[]{StringType.INSTANCE.sqlType()};  } |

sqlTypes() собщает Hibernate, как представлять данные со стороны базы данных. В моём случае для простоты я буду использовать строковое представление.

|  |  |
| --- | --- |
| 1  2  3  4 | @Override  public Class returnedClass() {    return NetworkObject.class;  } |

returnedClass() наоборот, сообщает какой класс будет отдан при чтении из базы.

|  |  |
| --- | --- |
| 1  2  3  4 | @Override  public boolean equals(Object o, Object o1) throws HibernateException {    return Objects.equals(0, 01);  } |

equals(), очевидно, сравнивает два объекта. В моей реализации я полагаюсь на [корректную реализацию equals() от project lombok](http://easyjava.ru/java/lombok/equals-tostring-i-sushhnosti/). Равно как и для hashCode():

|  |  |
| --- | --- |
| 1  2  3  4 | @Override  public int hashCode(Object o) throws HibernateException {    return o.hashCode();  } |

Наконец, самое главное происходит в двух методах. Первый, nullSafeGet(), восстанавливает объект при чтении его из базы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | @Override  public Object nullSafeGet(      ResultSet rs,      String[] names,      SessionImplementor si,      Object owner)      throws HibernateException, SQLException {    String network = rs.getString(names[0]);    if (null != network) {      String[] buf = network.split("/");      //Host addresses are stored without bitmask      //So we have check, where we are equipped with      //bitmask and, if not, find a proper value.      Short prefix = 128;      if (buf.length > 1) {        prefix = Short.valueOf(buf[1]);      } else {        if (!buf[0].contains(":")) {          prefix = 32;        }      }      return new NetworkObject(buf[0], prefix);    }    return null;  } |

В nullSafeGet() передаётся самый обычный [JDBC ResultSet](http://easyjava.ru/java/jdbc-resultset-i-rowset/), из которого мы получаем значение поля, как оно представлено в базе данных и строим каким-то образом необходимый объект из этого значения. В нашем частном случае адрес сети представляется как строка вида *«адрес/маска»*, а адрес хоста как строка вида *«адрес»*. Поэтому я разбиваю значение поля по символу *«/»,*проверяю наличие маски и если она отсутствует, угадываю тип адреса (IPv4 или IPv6) и добавляю соответствующую маску хоста.

Вторая функция, nullSafeSet(), наоборот, преобразовывает значения класса в вид, пригодный для записи в базу:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @Override  public void nullSafeSet(PreparedStatement st, Object value, int i, SessionImplementor sessionImplementor) throws HibernateException, SQLException {    if ( null == value ) {      st.setNull(i, StringType.INSTANCE.sqlType());    } else {      NetworkObject network = (NetworkObject) value;      st.setString(i, network.getAddress() + "/" + network.getBitmask());    }  } |

В nullSafeSet() передаётся [JDBC PreparedStatement](http://easyjava.ru/data/jdbc/jdbc-statements/), аргументы которого заполняются. В моём конкретном примере я строю строку вида *«адрес/маска»*

Далее следуют функции, которые нужные для поддержки кэширования и отслеживания изменений.

|  |  |
| --- | --- |
| 1  2  3  4 | @Override  public boolean isMutable() {    return false;  } |

isMutable() сообщает Hibernate, могут ли поля объекта менять свои значения после создания или нет. Для упрощения кода я сделал NetworkObject *immutable*и, соответственно, возвращаю здесь false.

|  |  |
| --- | --- |
| 1  2  3  4 | @Override  public Object deepCopy(Object o) throws HibernateException {    return o;  } |

deepCopy() создаёт полную копию объекта. Так как мой объект *immutable*, я возвращаю оригинал.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @Override  public Serializable disassemble(Object o) throws HibernateException {    return (Serializable) o;  }    @Override  public Object assemble(Serializable cached, Object owner) throws HibernateException {    return cached;  } |

assemble() и disassemble() конвертируют объект в вид, пригодный для хранения в кэше второго уровня и восстанавливают объект из вида, пригодного для хранения в кэше второго уровня. Так как NetworkObject реализует Serializable, я просто привожу его к этому интерфейсу. Как вариант реализации, я мог бы преобразовывать объект в строку и возвращать её, а впоследствии создавать из строки по новой.

|  |  |
| --- | --- |
| 1  2  3  4 | @Override  public Object replace(Object original, Object target, Object owner) throws HibernateException {    return original;  } |

Наконец replace() копирует изменения из нового объекта в старый. Так как объект *immutable*, достаточно вернуть оригинал.

Для тех, кому интересно, полный код пользовательского типа приведён под катом.

**Тестирование**

Для теста понадобится запущенный PostgreSQL, в котором мы создадим пользователя, базу данных и таблицу:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | CREATE ROLE test WITH PASSWORD 'test';  ALTER ROLE test WITH LOGIN;    CREATE DATABASE test OWNER test;    \c test    CREATE TABLE network (id BIGINT PRIMARY KEY, network INET);    GRANT ALL ON network TO test;    CREATE SEQUENCE hibernate\_sequence;    GRANT ALL ON hibernate\_sequence TO test;    CREATE CAST (CHARACTER VARYING AS inet) WITH INOUT  AS ASSIGNMENT; |

Кроме таблицы и sequence для [автогенерации первичных ключей](http://easyjava.ru/data/hibernate/pervichnye-klyuchi-v-hibernate/) я создаю автоматическое преобразование из *inet* в строку и обратно. Это позволяет мне использовать строковый тип данных со стороны Hibernate.

Для сохранения в базу я использую адреса example.org:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | NetworkObject example = new NetworkObject("93.184.216.34", (short)24);  NetworkObject example6 = new NetworkObject("2606:2800:220:1:248:1893:25c8:1946", (short)64);    Network n4 = new Network();  n4.setNetwork(example);    Network n6 = new Network();  n6.setNetwork(example6);    session.save(n4);  session.save(n6); |

Которые потом прочитаю из базы:

|  |  |
| --- | --- |
| 1  2  3  4 | session.createCriteria(Network.class)    .list()    .stream()    .forEach(System.out::println); |
| 1  2 | Network(id=1, network=NetworkObject(address=93.184.216.34, bitmask=24))  Network(id=2, network=NetworkObject(address=2606:2800:220:1:248:1893:25c8:1946, bitmask=64)) |

И проверю, что в базе данных они корректно представлены как ip сети:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | test=# select \* from network;  id |                network  ----+---------------------------------------    3 | 93.184.216.34/24    4 | 2606:2800:220:1:248:1893:25c8:1946/64  (2 строки)    test=# \d network      Таблица "public.network"  Колонка |  Тип   | Модификаторы  ---------+--------+--------------  id      | bigint | NOT NULL  network | inet   |  Индексы:      "network\_pkey" PRIMARY KEY, btree (id) |

Код примера доступен на [gitbub](https://github.com/EasyJavaRu/data-hibernate-usertype). Код из примера использует PostgreSQL. В hibernate.cfg*.xml* примера следует заменить адрес сервера с **‘127.0.0.1’** на адрес вашего PostgreSQL сервера, если он установлен не на локальной машине.

**Кэширование в Hibernate**

Кэши первого и второго уровней работают с объектами загружаемыми по id. А если в cache нужно занести данные полученные после запроса по условию, то необходимо использовать Кэш запросов.

В статье о [поддержке пользовательских типов в Hibernate](http://easyjava.ru/data/hibernate/polzovatelskie-tipy-v-hibernate/) упоминается о поддержке кэширования. В этой статье я постараюсь рассказать о кэшровании подробнее.

Идея кэширования (не только в [Hibernate](https://easyjava.ru/tag/hibernate/)) основывается на мнении, что из всех данных, доступных для обработки, работа ведётся только над некоторым небольшим набором и если ускорить доступ к этому набору, то в среднем программа будет работать быстрее. Говоря конкретно о [Hibernate](https://easyjava.ru/tag/hibernate/), доступ к базе занимает на порядки больше времени, чем доступ к объекту в памяти JVM. И поэтому, если какое-то время хранить в памяти загруженные из БД объекты, то при их повторном запросе [Hibernate](https://easyjava.ru/tag/hibernate/) сможет вернуть их гораздо быстрее.

**Кэш первого уровня**

С объектом Session, а точнее с *persistence context*, в Hibernate всегда связан кэш первого уровня. При помещении объекта в *persistence context*, то есть при его загрузке из БД или сохранении, объект так же автоматически будет помещён в кэш первого уровня и это невозможно отключить. Соответственно, при запросах того же самого объекта несколько раз в рамках одного *persistence context*, запрос в БД будет выполнен один раз, а всё остальные загрузки будут выполнены из кэша.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | Session session = sessionFactory.openSession();  session.beginTransaction();    // Database will be queried  System.out.println(session.get(Person.class, 123456L));    // Cached object will be returned  System.out.println(session.get(Person.class, 123456L));    session.getTransaction().commit();  session.close(); |

В примере выше только первый вызов get() инициирует запрос к базе, второй вызов будет обслужен уже из кэша и обращения к базе не произойдёт.

Интересно поведение кэша первого уровня при использовании [ленивой загрузки](http://easyjava.ru/data/hibernate/upravlenie-sushhnostyami-v-hibernate/). При загрузке объекта методом load() или объекта с лениво загружаемыми полями, лениво загружаемые данные в кэш не попадут. При обращении к данным будет выполнен запрос в базу и данные будут загружены и в объект и в кэш. А вот следующая попытка лениво загрузить объект приведёт к тому, что объект сразу вернут из кэша и уже полностью загруженным.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | session = sessionFactory.openSession();  session.beginTransaction();    // Database is not queried, only reference is returned  Person p1 = session.load(Person.class, 123456L);    // Query triggered, object is filled with data  System.out.println(p1);    // Cached, fully populated object will be returned  Person p2=session.load(Person.class, 123456L);    System.out.println(p2);    session.getTransaction().commit();  session.close(); |

**Кэш второго уровня**

Если кэш первого уровня существует только на уровне сессии и *persistence context,*то кэш второго уровня находится выше — на уровне SessionFactory и, следовательно, один и тот же кэш доступен одновременно в нескольких *persistence context.*Кэш второго уровня требует некоторой настройки и поэтому не включен по умолчанию. Настройка кэша заключается в конфигурировании реализации кэша и разрешения сущностям быть закэшированными.

**Конфигурирование кэша**

Hibernate не реализует сам никакого in-memory сache, а использует существующие реализации кэшей. Раньше Hibernate самостоятельно поддерживал интерфейс с этими кэшами, но сейчас существует [JCache](https://jcp.org/en/jsr/detail?id=107) и корректнее будет использовать этот интерфейс. Реализаций у [JCache](https://easyjava.ru/tag/jcache/) множество, но я выберу [ehcache](http://www.ehcache.org/), как одну из самых распространённых.

В первую очередь надо добавить поддержку JCache и [ehcache](https://easyjava.ru/tag/ehcache/) в зависимости:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | <properties>  <hibernate.version>5.2.0.Final</hibernate.version>  <ehcache.version>3.1.1</ehcache.version>  </properties>    <dependencies>    <dependency>      <groupId>org.ehcache</groupId>      <artifactId>ehcache</artifactId>      <version>${ehcache.version}</version>    </dependency>      <dependency>      <groupId>org.hibernate</groupId>      <artifactId>hibernate-core</artifactId>      <version>${hibernate.version}</version>    </dependency>      <dependency>      <groupId>org.hibernate</groupId>      <artifactId>hibernate-jcache</artifactId>      <version>${hibernate.version}</version>    </dependency>    </dependencies> |

Затем настроить hibernate на использование ehcache для кэширования:

|  |  |
| --- | --- |
| 1  2 | <property name="hibernate.cache.region.factory\_class">org.hibernate.cache.jcache.JCacheRegionFactory</property>  <property name="hibernate.javax.cache.provider">org.ehcache.jsr107.EhcacheCachingProvider</property> |

В первой строке мы говорим Hibernate, что хотим использовать JCache интерфейс, а во второй строке выбираем конкретную реализацию JCache: ehcache.

Наконец, включим кэш второго уровня:

|  |  |
| --- | --- |
| 1 | <property name="hibernate.cache.use\_second\_level\_cache">true</property> |

**@Cacheable и @Cache**

@Cacheable это аннотация JPA и позволяет объекту быть закэшированным. Hibernate поддерживает эту аннотацию в том же ключе.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @Entity  @Cacheable  public class Person extends AbstractIdentifiableObject {      @Getter      @Setter      private String firstName;        //Other fields  } |

@Cache это аннотация Hibernate, настраивающая тонкости кэширования объекта в кэше второго уровня Hibernate. Аннотации @Cacheable  достаточно, чтобы объект начал кэшироваться с настройками по умолчанию. При этом @Cache использованная без @Cacheable не разрешит кэширование объекта.

@Cache  принимает три параметра:

* include, имеющий по умолчанию значение *all* и означающий кэширование всего объекта. Второе возможное значение, *non-lazy,*запрещает кэширование лениво загружаемых объектов. Кэш первого уровня не обращает внимания на эту директиву и *всегда*кэширует лениво загружаемые объекты.
* region позволяет задать имя региона кэша для хранения сущности. Регион можно представить как разные кэши или разные части кэша, имеющие разные настройки на уровне реализации кэша. Например, я мог бы создать в конфигурации ehcache два региона, один с краткосрочным хранением объектов, другой с долгосрочным и отправлять часто изменяющиеся объекты в первый регион, а все остальные во второй.
* usage задаёт стратегию одновременного доступа к объектам.

Последний пункт достаточно объёмен, чтобы рассматривать его внутри списка. Проблема заключается в том, что кэш второго уровня доступен из нескольких сессий сразу и несколько потоков программы могут одновременно в разных транзакциях работать с одним и тем же объектом. Следовательно надо как-то обеспечивать их одинаковым представлением этого объекта.

Стратегий одновременного доступа к объектам в кэше в hibernate существует четыре:

* *translactional*— полноценное разделение транзакций. Каждая сессия и каждая транзакция видят объекты, как если бы только они с ним работали последовательно одна транзакция за другой. Плата за это — блокировки и потеря производительности.
* read-write — полноценный доступ к одной конкретной записи и разделение её состояния между транзакциями. Однако суммарное состояние нескольких объектов в разных транзакциях может отличаться.
* *nonstrict-read-write*— аналогичен *read-write,*но изменения объектов могут запаздывать и транзакции могут видеть старые версии объектов. Рекомендуется использовать в случаях, когда одновременное обновление объектов маловероятно и не может привести к проблемам.
* *read-only —*объекты кэшируются только для чтения и изменение удаляет их из кэша.

Список выше отсортирован по нарастанию производительности, *transactional* стратегия самая медленная, *read-only* самая быстрая. Недостатком *read-only* стратегии является её бесполезность, в случае если объекты постоянно изменяются, так как в этом случае они не будут задерживаться в кэше.

Использование кэша второго уровня требует изменений в конфигурации Hibernate и в коде сущностей, но не требует изменения кода запросов и управления сущностями:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | session = sessionFactory.openSession();  session.beginTransaction();    // Database will be queried  System.out.println(session.get(Person.class, 3L));    session.getTransaction().commit();  session.close();    session = sessionFactory.openSession();  session.beginTransaction();    // Database will not be queried, 2nd level cache will provide the data  System.out.println(session.get(Person.class, 3L));    session.getTransaction().commit();  session.close(); |

**Кэш запросов**

Кэши первого и второго уровней работают с объектами загружаемыми по id. Но в дикой природе к базе чаще выполняются запросы с условиями, чем загружаются какие-то заранее известные объекты:

|  |  |
| --- | --- |
| 1  2  3 | session.createCriteria(Passport.class)    .add(Restrictions.eq("series", "AS"))    .uniqueResult() |

И результат выполнения таких запросов тоже может потребоваться кэшировать. Например если вы делаете поисковый сайт по автозапчастям, то можете кэшировать запросы пользователей, которые, скорее всего, ищут одни запчасти гораздо чаще других. У кэша запросов есть и своя цена — Hibernate будет вынужден отслеживать сущности закешированные с определённым запросом и выкидывать запрос из кэша, если кто-то поменяет значение сущности. То есть для кэша запросов стратегия параллельного доступа всегда *read-only.*

Чтобы включить кэш запросов надо настроить внешний кэш, так же как и для кэша второго уровня, и разрешить Hibernate кэшировать запросы:

|  |  |
| --- | --- |
| 1 | <property name="hibernate.cache.use\_query\_cache">true</property> |

Но даже с этим разрешением Hibernate не будет кэшировать все запросы, а только те, кэширование которых явно запрошено методом setCacheable()

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | session = sessionFactory.openSession();  session.beginTransaction();    // Database will be queried  System.out.println(session.createCriteria(Passport.class)    .add(Restrictions.eq("series", "AS"))    .setCacheable(true)    .uniqueResult());    session.getTransaction().commit();  session.close();    session = sessionFactory.openSession();  session.beginTransaction();    // Database will not be queries, query cache will provide the data  System.out.println(session.createCriteria(Passport.class)    .add(Restrictions.eq("series", "AS"))    .setCacheable(true)    .uniqueResult());    session.getTransaction().commit();  session.close(); |

Кэш запросов, так же как и кэш второго уровня, существует на уровне SessionFactory и доступен во всех persistence context.

Код пример доступен на [github](https://github.com/EasyJavaRu/data-hibernate-cache).

**Транзакции и блокировки в Hibernate**

**Транзакции в**[**Hibernate**](https://easyjava.ru/tag/hibernate/)

Транзакция это набор операций, которые могут быть либо целиком и успешно выполнены, либо полностью не выполнены.

Транзакции в базах данных соответствуют свойствам ACID:

* Атомарность — транзакция может быть либо целиком выполнена, либо целиком отменена.
* Согласованность — состояние данных должно быть логически согласованным после выполнения (или отмены) транзации
* Изолированность — в процессе работы транзакции другие выполняющиеся в это время транзакции не влияют на неё.
* наДёжность — что-бы не произошло, транзакция останется атомарной.

Если вдруг транзакции в вашей базе не соответствуют этим принципам, то это плохие, негодные транзакции и они не заслуживают чести носить это имя.

Использование транзакций в дикой природе проще показать на примере: положим у нас есть система, в которой клиенты размещают какие-то заказы. За каждый заказ со счёта клиента снимается какая-то сумма. Таким образом, одна логическая единица работы разбивается на три операции в базе данных:

* Добавить заказ.
* Получить текущее значение счёта клиента.
* Уменьшить счёт клиента и обновить значение в базе.

Оборачивание этих трёх операций в одну транзакцию гарантирует нам, что либо мы добавим заказ и спишем деньги, либо ничего не произойдёт вообще. Третьего не дано.

Hibernate построен поверх [JDBC API](http://easyjava.ru/data/jdbc/) и реализует модель транзакций JDBC. Если быть точным, Hibernate способен работать или с JDBC транзакциями или с JTA транзакциями.О JTA — Java Transaction API я напишу как-нибудь позже, а пока сосредоточимся на JDBC транзакциях, тем более что с точки зрения использования их отличий не так и много.

Транзакцию можно начать вызовом beginTransaction() объекта Session, либо запросить у Session связанный с ней объект Transaction и позвать у последнего метод begin(). С объектом Session всегда связан ровно один объект Transaction, доступ к которому может быть получен вызовом getTransaction():

|  |  |
| --- | --- |
| 1  2 | Session session = sessionFactory.openSession();  Transaction t=session.getTransaction(); |

Методов для подтверждения или отката транзакции у объекта Session нет, необходимо всегда обращаться к объекту Transaction:

1. session.beginTransaction();

2. session.getTransaction().commit();

1. session.beginTransaction();

2. session.getTransaction().rollback();

Код выше подтверждает первую транзакцию и откатывает вторую. В [отличие от JDBC](http://easyjava.ru/data/jdbc/jdbc-tranzakcii/) в Hibernate не поддерживаются Savepoints и транзакция может только быть подтверждена или откачена, без промежуточных вариантов.

**Операции над транзакциями**

У объекта Transaction есть ещё несколько методов, кроме commit() и rollback(), которые позволяют тонко управлять поведением транзакции. Метод isActive() позволяет проверить, есть ли в рамках объекта Transaction управляемая им транзакция. Очевидно, что такая транзакция существует в промежутке времени между вызовами begin() и commit()/rollback().

Метод setRollbackOnly() помечает транзакцию как откаченную в будущем. В отличие от rollback() этот метод *не*закрывает транзакцию, и все последующие запросы к базе будут продолжать выполняться в рамках той же самой транзакции, но завершить эту транзакцию можно будет только откатом и вызовом rollback(). Вызов commit() на такой транзакции выбросит исключение. Проверить состояние транзакции можно вызовом getRollbackOnly().

**Блокировки в Hibernate**

Транзакции, как средство разграничения параллельной работы с данными, идут рядом с аналогичным средством разграничения, блокировками.

Блокировки, это механизм, позволяющий параллельную работу с одними и теми же данными в базе данных. Когда более чем одна транзакция пытается получить доступ к одним и тем же данным в одно и то же время, в дело вступают блокировки, которые гарантируют, что только одна из этих транзакций изменит данные.

Существует два основных подхода к блокированию транзакций:*оптимистичный* и *пессимистичный. Оптимистичный* подход предполагает, что параллельно выполняющиеся транзакции редко обращаются к одним и тем же данным и позволяет им спокойно и свободно выполнять любые чтения и обновления данных. Но, при окончании транзакции *(то есть записи данных в базу)*, производится проверка, изменились ли данные в ходе выполнения данной транзакции и если да, транзакция обрывается и выбрасывается исключение.

*Пессимистичный* подход напротив, ориентирован на транзакции, которые постоянно или достаточно часто конкурируют за одни и те же данные и поэтому блокирует доступ к данным превентивно, в тот момент когда читает их. Другие транзакции останавливаются, когда пытаются обратиться к заблокированным данным и ждут снятия блокировки (или кидают исключение).

Разница в том, что в первом случае обеспечивается более высокий уровень конкурентности при доступе к базе, который оплачивается необходимостью переповтора транзакций, в случае коллизии. Во втором случае транзакции гарантируется, что только она будет иметь полный доступ к данным, но за счёт понижения уровня конкурентности и затрат времени на ожидание блокировки.

**Оптимистичное блокирование**

[Как и в JPA](http://easyjava.ru/data/jpa/blokirovki-v-jpa/), *оптимистичное* блокирование выполнено на уровне Hibernate, а не базы данных. Для поддержки таких блокировок в класс вводится специально поле версии, которое анализирует Hibernate при сохранении изменений.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | @Entity  public class Company extends AbstractIdentifiableObject {      @Version      private long version;        @Getter      @Setter      private String name;        @Getter      @Setter      @ManyToMany(mappedBy = "workingPlaces")      private Collection<Person> workers;  } |

Поле, аннотирование @Version может быть целочисленным или временным. Hibernate разрешает доступ к объектам всем транзакциям сразу, без каких-либо ограничений, но при сохранении объектов проверяет, нет ли изменений, внесённых другими транзакциями. В случае, если обнаружится конкурирующее изменение, транзакция откатывается.

**Пессимистичное блокирование**

*Пессимистичное* блокирование выполняется на уровне базы и поэтому не требует вмешательств в код сущности. Блокировка в случае пессимистичного блокирование всегда запрашивается для конкретного объекта во время его загрузки или позднее. Тут показаны 3-и возможных варианта использования:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | Person **p** = session.load(Person.class, 3L, **LockMode.PESSIMISTIC\_READ**); //тут блокируется строка с id = 3L в таблице Person    session.lock(**p**, **LockMode.PESSIMISTIC\_WRITE**);    session.createCriteria(Person.class)    .setLockMode(**LockMode.PESSIMISTIC\_READ**)    .uniqueResult(); |

В примере выше блокировка запрашивается при загрузке объекта методом load(), накладывается другая блокировка на уже загруженный объект методом lock() и, наконец, все объекты, соответствующие критерию будут загружены с блокировкой, указанной в setLockMode().

Если не говорить о тонкостях, в Hibernate поддерживаются две главных *пессимистичных* блокировки:

* LockMode.PESSIMISTIC\_READ — данные блокируются в момент чтения и это гарантирует, что никто в ходе выполнения транзакции не сможет их изменить. Остальные транзакции, тем не менее, смогут параллельно читать эти данные. Использование этой блокировки может вызывать долгое ожидание блокировки или даже выкидываниеOptimisticLockException.
* LockMode.PESSIMISTIC\_WRITE — данные блокируются в момент записи и никто с момента захвата блокировки не может в них писать и не может их читать до окончания транзакции, владеющей блокировкой. Использование этой блокировки может вызывать долгое ожидание блокировки.

LockMode.PESSIMISTIC\_READ может поддерживаться не всеми базами данных и в этом случае автоматически будет применён LockMode.PESSIMISTIC\_WRITE

**ENVERS: автоматический аудит изменений в Hibernate и JPA**

«…и мы должны иметь возможность видеть, когда, как и кто изменил данные» — распространённая хотелка при разработке приложения, работающего с БД.

Заказчик обычно хочет видеть, для некоторых таблиц, какие изменения в них вносились, когда вносились, кем вносились и какие были предыдущие значения. Существует множество решений этой задачи: можно обвесить необходимые таблицы триггерами, можно сохранять объекты с помощью хранимых процедур, можно перехватывать запросы к базе с помощью AOP и так далее. [Hibernate](https://easyjava.ru/tag/hibernate/) предлагает собственно решение, *envers*, решающее задачу ведения истории с помощью одной аннотации.

**Любые изменения проведенные в таблице на стороне БД *(например удалить строки)* - не увидит и будет продолжать показывать изменения *(которые через него прошли раньше)* по уже удаленным данным + добавит новые изменения, но удаление данных не покажет!**

**Настройка envers**

Чтобы включить автоматическое ведение истории изменений, необходимо выполнить два шага:

Добавить библиотеку *envers* в classpath:

|  |  |
| --- | --- |
| 1  2  3  4  5 | <dependency>    <groupId>org.hibernate</groupId>    <artifactId>hibernate-envers</artifactId>    <version>5.2.0.Final</version>  </dependency> |

Добавить аннотацию @Audited к классам, для которых требуется отслеживание истории изменении. Рекоменуется добавить эту аннотацию так же и к классам, имеющим связи с классами, уже помеченными @Audited. Далее в статье я  буду использовать сущности из [примера HQL](http://easyjava.ru/data/hibernate/hibernate-query-language/).

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @Entity  @Audited  public class Person extends AbstractIdentifiableObject {      @Getter      @Setter      private String firstName;        /\* other fields \*/  } |

Hibernate автоматически (если конечно включено *hbm2ddl.auto)*создаст следующие таблицы:

* имя\_сущности\_AUD — для хранения исторических данных. Эти таблицы состоят из поля *id* отслеживамой сущности, поля *rev*, в котором хранится номер редакции, поля *revtype*, в котором хранится тип изменения (создано/модифицировано/удалено) и все отслеживаемые поля оригинальной сущности.
* REVINFO — для хранения списка ревизий. В этой таблице хранятся номера ревизий и время их создания. Так же в этой таблице можно хранить дополнительные данные, такие как имя пользователя, создавшего ревизию, например.

**Когда**

Для начала ответим на вопрос «Когда были изменены данные». Создадим пробный запрос и проверим, как это отразилось в истории.

**Длинный код создания запроса**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | Object[] result = (Object[]) AuditReaderFactory    .get(session)    .createQuery()    .forRevisionsOfEntity(Person.class, false, true)    .getSingleResult()**;**  System.out.println(result[0]);  System.out.println(result[1]);  System.out.println(result[2]); |

Запросы к истории делаются с помощью AuditQuery, который можно запросить у AuditReaderFactory.

Запрос forRevisionsOfEntity() возвращает данные о изменениях сущности:

- первый параметр задаёт класс сущности;

- второй параметр регулируют, хотим ли мы получить только ревизии ( true) или ревизии и данные сущностей ( false). В первом случае возвращаются объекты ревизий, во втором — массив из трех объектов: данные сущности в ревизии, ревизия, тип изменения;

- третий параметр регулирует, включать ли в вывод удалённые записи или нет.

|  |  |
| --- | --- |
|  | Person{firstName='Test', lastName='Testoff', dob=2016-08-25,  passport=Passport{series='AS', no='123456', issueDate=2016-08-25, validity=P20Y, owner=Testoff}, primaryAddress=Address{city='Kickapoo', street='Main street', building='1', tenants=Test},  workingPlaces=[Company{name='Acme Ltd', workers=Test}]}  DefaultRevisionEntity(id = 1, revisionDate = 25.08.2016 11:02:15)  ADD |

В результате мы видим, что в ревизии 1, созданной сегодня, 25 августа, в 11:02, была добавлена сущность Person и отображены её данные.

**Как**

Когда изменились данные мы видим, но интересно было бы посмотреть в динамике — какие были значения, какие стали. Попробуем изменить ранее созданную персону:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | // Set new name  session = sessionFactory.openSession();  session.beginTransaction();    Person p = (Person) session    .createCriteria(Person.class)    .add(Restrictions.eq("firstName", "Test"))    .uniqueResult();  p.setFirstName("Johan");  p.setLastName("von Testow");    session.save(p);    session.getTransaction().commit();  session.close(); |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | AuditReaderFactory    .get(session)    .createQuery()    .forRevisionsOfEntity(Person.class, false, true)    .getResultList()    .forEach(r -> {      Object[] v = (Object[])r;        System.out.println(v[0]);        System.out.println(v[1]);        System.out.println(v[2]);  }); |

|  |  |
| --- | --- |
|  | Person{firstName='Test', lastName='Testoff', dob=2016-08-25, passport=Passport{series='AS', no='123456', issueDate=2016-08-25, validity=P20Y, owner=Testoff}, primaryAddress=Address{city='Kickapoo', street='Main street', building='1', tenants=Test}, workingPlaces=[Company{name='Acme Ltd', workers=Test}]}  DefaultRevisionEntity(id = 1, revisionDate = 25.08.2016 11:16:14)  ADD  Person{firstName='Johan', lastName='von Testow', dob=2016-08-25, passport=Passport{series='AS', no='123456', issueDate=2016-08-25, validity=P20Y, owner=von Testow}, primaryAddress=Address{city='Kickapoo', street='Main street', building='1', tenants=Johan}, workingPlaces=[Company{name='Acme Ltd', workers=Johan}]}  DefaultRevisionEntity(id = 2, revisionDate = 25.08.2016 11:16:14)  MOD |

В этот раз я запрашиваю список ревизий и сразу вижу, что над сущностью Person было проведено две операции — создание записи и изменение данных. В обоих случаях видно когда была проведена операция и как стала выглядеть изменяемая сущность.

Очевидно, что выборка по всей истории изменений не так интересна, как выборка только по интересующему объекту. В AuditQuery можно задавать критерии выборки используя похожий на [Criteria API](http://easyjava.ru/data/hibernate/hibernate-criteria-api/) подход. Создадим новую персону и проведём над ней несколько действий:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | session = sessionFactory.openSession();  session.beginTransaction();    //Следующий блок кода загрузит объект из базы и сохранит его точную копию, но с другим id. Т.е. новой персоне оставим тот же адресс  Address address = p.getPrimaryAddress(); //загружаем объект из БД  session.detach(address); //отключаем связь между загруженным объектом и БД  address.setId(null); //сбрасывем id    Passport passport = new Passport();  passport.setSeries("TT");  passport.setNo("250816");  passport.setIssueDate(LocalDate.now());  passport.setValidity(Period.ofYears(20));    Person person = new Person();  person.setFirstName("Brunhild");  person.setLastName("Testonkowski");  person.setDob(LocalDate.now());  person.setPrimaryAddress(p.getPrimaryAddress());  person.setPassport(passport);  passport.setOwner(person);    session.save(person);    session.getTransaction().commit();  session.close();            session = sessionFactory.openSession();          session.beginTransaction();            person.setLastName("von Testow");          person.getPassport().setSeries("VT");          person.getPassport().setNo("101731");            session.merge(person);            session.getTransaction().commit();          session.close(); |
| 1  2  3  4  5  6  7  8  9  10  11  12 | AuditReaderFactory    .get(session)    .createQuery()    .forRevisionsOfEntity(Person.class, false, true)    .add(AuditEntity.property("firstName").eq("Brunhild"))    .getResultList()    .forEach(r -> {      Object[] v = (Object[])r;        System.out.println(v[0]);        System.out.println(v[1]);        System.out.println(v[2]);  }); |

В запросе к истории я делаю ограничение по значению поля firstName, что позволяет мне просмотреть не всю историю изменений, а только по персоне с именем *Брунгильда.*

|  |  |
| --- | --- |
|  | Person{firstName='Brunhild', lastName='Testonkowski', dob=2016-08-25, passport=Passport{series='TT', no='250816', issueDate=2016-08-25, validity=P20Y, owner=Testonkowski}, primaryAddress=Address{city='Kickapoo', street='Main street', building='1', tenants=Brunhild}, workingPlaces=[]}  DefaultRevisionEntity(id = 3, revisionDate = 25.08.2016 11:16:14)  ADD  Person{firstName='Brunhild', lastName='von Testow', dob=2016-08-25, passport=Passport{series='VT', no='101731', issueDate=2016-08-25, validity=P20Y, owner=von Testow}, primaryAddress=Address{city='Kickapoo', street='Main street', building='1', tenants=Brunhild}, workingPlaces=[]}  DefaultRevisionEntity(id = 4, revisionDate = 25.08.2016 11:16:14)  MOD |

Можно так же убедиться, что изменения в связанных таблицах так же записываются:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | AuditReaderFactory    .get(session)    .createQuery()    .forRevisionsOfEntity(Passport.class, false, true)    .getResultList()    .forEach(r -> {      Object[] v = (Object[])r;        System.out.println(v[0]);        System.out.println(v[1]);        System.out.println(v[2]);  }); |
|  | Passport{series='AS', no='123456', issueDate=2016-08-25, validity=P20Y, owner=Testoff}  DefaultRevisionEntity(id = 1, revisionDate = 25.08.2016 11:16:14)  ADD  Passport{series='TT', no='250816', issueDate=2016-08-25, validity=P20Y, owner=Testonkowski}  DefaultRevisionEntity(id = 3, revisionDate = 25.08.2016 11:16:14)  ADD  Passport{series='VT', no='101731', issueDate=2016-08-25, validity=P20Y, owner=von Testow}  DefaultRevisionEntity(id = 4, revisionDate = 25.08.2016 11:16:14)  MOD  Passport{series='null', no='null', issueDate=null, validity=null, owner=None}  DefaultRevisionEntity(id = 5, revisionDate = 25.08.2016 11:16:14)  DEL |

Наконец, кроме просмотра истории ревизий, можно запросить и конкретную ревизию:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | session = sessionFactory.openSession();  session.beginTransaction();    System.out.println(    session      .createCriteria(Person.class)      .add(Restrictions.eq("firstName", "Johan"))      .uniqueResult()  );    System.out.println(    AuditReaderFactory      .get(session)      .createQuery()      .forEntitiesAtRevision(Person.class, 1) //тут мы применяем обратный метод forEntitiesAtRevision, а не forRevisionsOfEntity      .getSingleResult()  );    session.getTransaction().commit();  session.close(); |
|  | Person{firstName='Johan', lastName='von Testow', dob=2016-08-25, passport=Passport{series='AS', no='123456', issueDate=2016-08-25, validity=P20Y, owner=von Testow}, primaryAddress=Address{city='Kickapoo', street='Main street', building='1', tenants=Johan}, workingPlaces=[Company{name='Acme Ltd', workers=Johan}]}  Person{firstName='Test', lastName='Testoff', dob=2016-08-25, passport=Passport{series='AS', no='123456', issueDate=2016-08-25, validity=P20Y, owner=Testoff}, primaryAddress=Address{city='Kickapoo', street='Main street', building='1', tenants=Test}, workingPlaces=[Company{name='Acme Ltd', workers=Test}]} |

forEntitiesAtRevision() создаёт запрос, достающие значение сущности заданного класса для заданной ревизии. На него так же можно накладывать критерии, как и на предыдущий запрос. В примере выше мы видим, что текущее состояние сущности и её историческое состояние отличаются.

**Кто**

Самый интересный вопрос — кто же внёс изменения в таблицу. Hibernate сам не записывает такую информацию, так как не знаком со структурой каждого приложения в мире, поэтому для записывания данных пользователя придётся немного потрудится.

В первую очередь расширим сущность ревизии с тем, чтобы добавить в неё дополнительное поле:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | /\*\*  \* Custom revision entity that holds additional data.  \*/  @Entity  @RevisionEntity(UserRevisionListener.class)  class UserRevision extends DefaultRevisionEntity {      /\*\*       \* User who created that revision.       \*/      @Getter      @Setter      private String username;  } |

Аннотация @RevisionEntity ссылается на UserRevisionListener, который будет вызываться для заполнения дополнительных полей ревизии. Пока что я буду писать в него константную строку, но в реальной жизни там будет больше кода:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | /\*\*  \* Custom revision entity processor.  \*/  class UserRevisionListener implements RevisionListener {      private final static String USERNAME = "vpupkin"; //тут используется стационарное значение идентификации пользователя, для добавления его к ревизии и времени создания в таблицу REVINFO. В реалии эти данные нужно будет получать.        @Override      public void newRevision(Object o) {          UserRevision r= (UserRevision) o;          r.setUsername(USERNAME);      }  } |

Наконец, надо не забыть добавить собственную сущность ревизии в конфигурацию маппинга:

|  |  |
| --- | --- |
| 1 | <mapping class="ru.easyjava.data.hibernate.envers.UserRevision"/> |

Результат на лицо:

|  |  |
| --- | --- |
| 1  2  3 | Person{firstName='Test', lastName='Testoff', dob=2016-08-25, passport=Passport{series='AS', no='123456', issueDate=2016-08-25, validity=P20Y, owner=Testoff}, primaryAddress=Address{city='Kickapoo', street='Main street', building='1', tenants=Test}, workingPlaces=[Company{name='Acme Ltd', workers=Test}]}  UserRevision(super=DefaultRevisionEntity(id = 5, revisionDate = 25.08.2016 12:10:08), username=vpupkin)  ADD |

**Envers и JPA**

Механизм envers будет работать и при использовании Hibernate в качестве [JPA Implementation](http://easyjava.ru/data/jpa/hello-jpa-i-hibernate/), но с небольшими изменениями. При создании AuditQuery надо передавать объект EntityManager, а не Session и не требуется явно регистрировать собственную сущность ревизий.

Код примера доступен на [github](https://github.com/EasyJavaRu/data-hibernate-envers).

# Интерцепторы и события в Hibernate

Бывает необходимо реагировать на какие-то события происходящие внутри [Hibernate](https://easyjava.ru/tag/hibernate/) - например, контролировать как [сущности переходят из одного состояния в другое](http://easyjava.ru/data/hibernate/upravlenie-sushhnostyami-v-hibernate/). Применений этой возможности можно придумать множество — вести статистику обращений, количество вызовов метода save(), проверять уровень доступа, [вести историю изменений](http://easyjava.ru/data/hibernate/envers-avtomaticheskij-audit-izmenenij-v-hibernate/) и так далее. Hibernate поддерживает два разных, но похожих, механизма для вмешательства в свои внутренние дела — интерцепторы и события.

# События

# *Сработал только Длинный пример!!!*

События это интерфейс реагирования на изменения состояния сущностей. Отличия от Интерцепторов в основном в более удобном api (*просто 1.выбираем из списка какой* \*EventListener*интерфейс будем имплементировать в наш класс и в нем переопределяем его метод, после чего 2.связываем наш класс с событием*) и возможности конфигурирования через xml.

Т.е. есть уже готовые шаблоны действий(интерфейсы), которые просто “имплементятся” к создаваемому классу-слушателю. А для применения такого написанного нами класса, IDEA автоматически подвяжет его под необходимые события, без дополнительного прописания в коде (как в Интерцепторах).

Событий существует великое множество. Грубо можно предположить,что на каждый вызов у Session есть своё собственное событие, плюс pre-событие и post-событие. Например, посчитаем, сколько раз в ходе выполнения кода делается вызов save():

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | /\*\*  \* Counts save events.  \*/  public class SaveCounter implements SaveOrUpdateEventListener {      AtomicInteger counter = new AtomicInteger();      @Override      public void onSaveOrUpdate(SaveOrUpdateEvent e) throws HibernateException {          System.out.println("Saving: " + e.getResultId());          System.out.println("Totally seen " + counter.incrementAndGet() + " save events");      }  } |

Каждому типу события сооветствует свой \*EventListener интерфейс, поэтому для save() вызова я использую SaveOrUpdateListener.

Стоит отметить, что хранить данные (состояние) внутри обработчика событий дурной тон и следует этого избегать. С другой стороны, я использую атомарный счётчик, который можно безопасно использовать в конкуретной среде.

Обработчик события надо, помимо прочего, связать с событием. Это можно сделать сконфигурировав Hibernate программно, а можно используя xml конфигурацию:

|  |  |
| --- | --- |
| 1  2  3 | <event type="save">    <listener class="ru.easyjava.data.hibernate.events.SaveCounter"/>  </event> |

Теперь любой вызов save() приведёт к такой записи на экране:

|  |  |
| --- | --- |
| 1  2 | Saving: 1  Totally seen 1 save events |

Код примера доступен на [github](https://github.com/EasyJavaRu/data-hibernate-interceptors).

**Короткий пример**

/////////////////////////////////////////////////////////////////////////

import java.io.Serializable;

import java.util.\*;

import java.sql.\*;

import org.hibernate.\*;

import org.hibernate.cfg.\*;

import org.hibernate.criterion.\*;

import org.hibernate.event.\*;

import org.hibernate.event.def.\*;

public class Main {

public static void main(String[] args) throws Exception {

HibernateUtil.setup("create table User ( id int, username VARCHAR);");

Configuration config = new Configuration();

config.setListener("save-update", **new UseringSaveOrUpdateEventListener()**); //непонятно почему, но в подгружаемом классе Configuration не подгружает этот метод.

SessionFactory factory = config.configure().buildSessionFactory();

Session session = factory.openSession();

Transaction tx = session.beginTransaction();

session.saveOrUpdate(new User("R1"));

session.saveOrUpdate(new User("R2"));

tx.commit();

session.close();

}

}

class **UseringSaveOrUpdateEventListener** extends DefaultSaveOrUpdateEventListener

{

public Serializable onSaveOrUpdate(SaveOrUpdateEvent event) throws HibernateException {

if( event.getObject() instanceof User ) {

User user = (User)event.getObject();

System.out.println("Preparing to save or update user " + user.getUsername());

if( user.getUsername().equalsIgnoreCase("R1")) {

System.out.println("Обычный пользователь не зарегистрирован.");

// Здесь мы не допускаем, чтобы вызов saveOrUpdate в сеансе оказывал какое-либо влияние на базу данных

return null;

}

}

return super.onSaveOrUpdate(event);

}

}

/////////////////////////////////////////////////////////////////////////

<?xml version='1.0' encoding='utf-8'?>

<!DOCTYPE hibernate-mapping

PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"

"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>

<class name="User">

<id name="id" type="int">

<generator class="increment"/>

</id>

<property name="username" type="string" length="32"/>

</class>

</hibernate-mapping>

/////////////////////////////////////////////////////////////////////////

@Entity

@Table(name = "user")

@Audited /\* подключаем Envers \*/

@Getter

@Setter

@ToString

@NoArgsConstructor

public class User{

private int id;

private String username;

public User(String u){ username = u; }

}

**Длинный пример**

Добавьте следующие jar-зависимости в pom.xml файл вашего проекта maven.

**pom.xml**

<project xmlns="<http://maven.apache.org/POM/4.0.0>" xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>"

xsi:schemaLocation="<http://maven.apache.org/POM/4.0.0> <http://maven.apache.org/xsd/maven-4.0.0.xsd>">

<modelVersion>4.0.0</modelVersion>

<groupId>com.boraji.tutorial.hibernate</groupId>

<artifactId>hibernate-event-example</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>hibernate-event-example</name>

<dependencies>

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-core</artifactId>

<version>5.2.12.Final</version>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>6.0.6</version>

</dependency>

<dependency>

<groupId>org.apache.logging.log4j</groupId>

<artifactId>log4j-core</artifactId>

<version>2.8.2</version>

</dependency>

<dependency>

<groupId>org.apache.logging.log4j</groupId>

<artifactId>log4j-api</artifactId>

<version>2.8.2</version>

</dependency>

</dependencies>

<build>

<sourceDirectory>src/main/java</sourceDirectory>

<plugins>

<plugin>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.5.1</version>

<configuration>

<source>1.8</source>

<target>1.8</target>

</configuration>

</plugin>

</plugins>

</build>

</project>

Класс сущности

Создайте простой  @Entity класс следующим образом.

**package** com.boraji.tutorial.hibernate.entity;

**import** javax.persistence.Entity;

**import** javax.persistence.GeneratedValue;

**import** javax.persistence.GenerationType;

**import** javax.persistence.Id;

@Entity

**public** **class** **Book** {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

**private** **long** id;

**private** String title;

**private** String author;

*//Getter and Setter methods*

**public** String **toString**() {

**return** "Book [id=" + id + ", title=" + title + ", author=" + author + "]";

}

}

Реализация слушателей событий

В Hibernate все интерфейсы прослушивателя событий доступны в org.hibernate.event.spiпакете. Вы можете использовать любой приемник событий, основанный на org.hibernate.event.spi.EventType.

В этом примере я покажу вам, как реализовать LoadEventListener, SaveOrUpdateEventListenerи RefreshEventListenerследующим образом.

**LoadEventListenerImp.java**

**package** com.boraji.tutorial.hibernate.event;

**import** org.apache.logging.log4j.LogManager;

**import** org.apache.logging.log4j.Logger;

**import** org.hibernate.HibernateException;

**import** org.hibernate.event.spi.LoadEvent;

**import** org.hibernate.event.spi.LoadEventListener;

**import** com.boraji.tutorial.hibernate.entity.Book;

**public** **class** **LoadEventListenerImp** **implements** **LoadEventListener** {

**private** **static** **final** **long** serialVersionUID = 1L;

**private** **static** Logger logger = LogManager.getLogger(LoadEventListenerImp.class);

@Override

**public** **void** **onLoad**(LoadEvent e, LoadType type) **throws** HibernateException {

logger.info("onLoad is called.");

Object obj = e.getResult();

**if** (obj **instanceof** Book) {

Book book = (Book) obj;

logger.info(book);

}

}

}

**SaveUpdateEventListenerImp.java**

**package** com.boraji.tutorial.hibernate.event;

**import** org.apache.logging.log4j.LogManager;

**import** org.apache.logging.log4j.Logger;

**import** org.hibernate.HibernateException;

**import** org.hibernate.event.spi.SaveOrUpdateEvent;

**import** org.hibernate.event.spi.SaveOrUpdateEventListener;

**import** com.boraji.tutorial.hibernate.entity.Book;

**public** **class** **SaveUpdateEventListenerImp** **implements** **SaveOrUpdateEventListener** {

**private** **static** Logger logger = LogManager

.getLogger(SaveUpdateEventListenerImp.class);

**private** **static** **final** **long** serialVersionUID = 1L;

@Override

**public** **void** **onSaveOrUpdate**(SaveOrUpdateEvent e) **throws** HibernateException {

logger.info("onSaveOrUpdate is called.");

Object obj = e.getEntity();

**if** (obj **instanceof** Book) {

Book book = (Book) obj;

logger.info(book);

}

}

}

**RefreshEventListenerImp.java**

**package** com.boraji.tutorial.hibernate.event;

**import** java.util.Map;

**import** org.apache.logging.log4j.LogManager;

**import** org.apache.logging.log4j.Logger;

**import** org.hibernate.HibernateException;

**import** org.hibernate.event.spi.RefreshEvent;

**import** org.hibernate.event.spi.RefreshEventListener;

**import** com.boraji.tutorial.hibernate.entity.Book;

**public** **class** **RefreshEventListenerImp** **implements** **RefreshEventListener** {

**private** **static** **final** **long** serialVersionUID = 1L;

**private** **static** Logger logger = LogManager

.getLogger(RefreshEventListenerImp.class);

@Override

**public** **void** **onRefresh**(RefreshEvent e) **throws** HibernateException {

logger.info("onRefresh is called.");

Object obj = e.getObject();

**if** (obj **instanceof** Book) {

Book book = (Book) obj;

logger.info(book);

}

}

@SuppressWarnings("rawtypes")

@Override

**public** **void** **onRefresh**(RefreshEvent e, Map refreshedAlready)

**throws** HibernateException {

logger.info("onRefresh is called.");

}

}

Регистрация слушателей событий

Чтобы зарегистрировать прослушиватели событий, вам нужно создать класс интегратора, реализовав org.hibernate.integrator.spi.Integrator интерфейс следующим образом:

**EventListenerIntegrator.java**

**package** com.boraji.tutorial.hibernate;

**import** org.hibernate.boot.Metadata;

**import** org.hibernate.engine.spi.SessionFactoryImplementor;

**import** org.hibernate.event.service.spi.EventListenerRegistry;

**import** org.hibernate.event.spi.EventType;

**import** org.hibernate.integrator.spi.Integrator;

**import** org.hibernate.service.spi.SessionFactoryServiceRegistry;

**import** com.boraji.tutorial.hibernate.event.LoadEventListenerImp;

**import** com.boraji.tutorial.hibernate.event.RefreshEventListenerImp;

**import** com.boraji.tutorial.hibernate.event.SaveUpdateEventListenerImp;

**public** **class** **EventListenerIntegrator** **implements** **Integrator** {

@Override

**public** **void** **integrate**(Metadata metadata, SessionFactoryImplementor

sessionFactory, SessionFactoryServiceRegistry serviceRegistry) {

EventListenerRegistry eventListenerRegistry =

serviceRegistry.getService(EventListenerRegistry.class);

eventListenerRegistry.getEventListenerGroup(EventType.SAVE)

.appendListener(**new** **SaveUpdateEventListenerImp**());

eventListenerRegistry.getEventListenerGroup(EventType.LOAD)

.appendListener(**new** **LoadEventListenerImp**());

eventListenerRegistry.getEventListenerGroup(EventType.REFRESH)

.appendListener(**new** **RefreshEventListenerImp**());

}

@Override

**public** **void** **disintegrate**(SessionFactoryImplementor sessionFactory,

SessionFactoryServiceRegistry serviceRegistry) {

}

}

Hibernate Utility класс

Создайте вспомогательный класс  HibernateUtil для начальной загрузки Hibernate.

Добавьте метод  EventListenerIntegrator используя  BootstrapServiceRegistryBuilder#applyIntegrator() для применения к реестру начальной загрузки.

Сопоставьте  Book сущность, используя  #MetadataSources.addAnnotatedClass() метод.

**HibernateUtil.java**

**package** com.boraji.tutorial.hibernate;

**import** java.util.HashMap;

**import** java.util.Map;

**import** org.hibernate.SessionFactory;

**import** org.hibernate.boot.Metadata;

**import** org.hibernate.boot.MetadataSources;

**import** org.hibernate.boot.registry.BootstrapServiceRegistry;

**import** org.hibernate.boot.registry.BootstrapServiceRegistryBuilder;

**import** org.hibernate.boot.registry.StandardServiceRegistry;

**import** org.hibernate.boot.registry.StandardServiceRegistryBuilder;

**import** org.hibernate.cfg.Environment;

**import** com.boraji.tutorial.hibernate.entity.Book;

**public** **class** **HibernateUtil** {

**private** **static** StandardServiceRegistry registry;

**private** **static** SessionFactory sessionFactory;

**public** **static** SessionFactory **getSessionFactory**() {

**if** (sessionFactory == **null**) {

**try** {

BootstrapServiceRegistry bootstrapRegistry =

**new** BootstrapServiceRegistryBuilder()

.applyIntegrator(**new** EventListenerIntegrator())

.build();

StandardServiceRegistryBuilder registryBuilder =

**new** StandardServiceRegistryBuilder(bootstrapRegistry);

Map<String, Object> settings = **new** HashMap<>();

settings.put(Environment.DRIVER, "com.mysql.cj.jdbc.Driver");

settings.put(Environment.URL, "jdbc:mysql://localhost:3306/BORAJI?useSSL=false");

settings.put(Environment.USER, "root");

settings.put(Environment.PASS, "admin");

settings.put(Environment.HBM2DDL\_AUTO, "update");

registryBuilder.applySettings(settings);

*или если конфигурацию необходимо брать из файла* ***hibernate.cfg.xml****, то просто:*

registryBuilder.configure();

registry = registryBuilder.build();

MetadataSources sources = **new** MetadataSources(registry)

.addAnnotatedClass(Book.class); //где “Book” это наша сущность

Metadata metadata = sources.getMetadataBuilder().build();

sessionFactory = metadata.getSessionFactoryBuilder().build();

} **catch** (Exception e) {

**if** (registry != **null**) {

StandardServiceRegistryBuilder.destroy(registry);

}

e.printStackTrace();

}

}

**return** sessionFactory;

}

**public** **static** **void** **shutdown**() {

**if** (registry != **null**) {

StandardServiceRegistryBuilder.destroy(registry);

}

}

}

Регистрация слушателей событий

Чтобы включить ведение журнала, создайте  log4j2.xml файл в  src/main/resources папке и напишите в нем следующий код.

**log4j2.xml**

**<?xml version="1.0" encoding="UTF-8"?>**

<Configuration>

<Appenders>

*<!-- Console Appender -->*

<Console name="Console" target="SYSTEM\_OUT">

<PatternLayout

pattern="%d{yyyy-MMM-dd HH:mm:ss a} %-5level %logger{36} - %msg%n" />

</Console>

</Appenders>

<Loggers>

*<!-- Log everything in hibernate -->*

<Logger name="com.boraji.tutorial.hibernate" level="trace"

additivity="false">

<AppenderRef ref="Console" />

</Logger>

<Root level="error">

<AppenderRef ref="Console" />

</Root>

</Loggers>

</Configuration>

Запустить приложение

Создайте основной класс для проверки зарегистрированных слушателей событий.

**MainApp.java**

**package** com.boraji.tutorial.hibernate;

**import** org.apache.logging.log4j.LogManager;

**import** org.apache.logging.log4j.Logger;

**import** org.hibernate.Session;

**import** org.hibernate.Transaction;

**import** org.hibernate.query.Query;

**import** com.boraji.tutorial.hibernate.entity.Book;

**public** **class** **MainApp** {

**private** **static** Logger logger = LogManager.getLogger(MainApp.class);

**public** **static** **void** **main**(String[] args) {

Session session = **null**;

Transaction transaction = **null**;

**try** {

session = HibernateUtil.getSessionFactory().openSession();

transaction = session.getTransaction();

transaction.begin();

*// Save Book*

Book book = **new** Book();

book.setTitle("Thinking in Java");

book.setAuthor("Bruce Eckel");

session.save(book);

*// Load Book*

Book book2 = session.get(Book.class, 1l);

logger.info("Auther name before update - "+book2.getAuthor());

*// Update Book*

Query<?> query=session.createQuery("update Book set author=LOWER(author)"

+ " where id=:id");

query.setParameter("id", book2.getId());

query.executeUpdate();

*//Refresh Book*

session.refresh(book2);

logger.info("Auther name after update - "+book2.getAuthor());

transaction.commit();

} **catch** (Exception e) {

**if** (transaction != **null**) {

transaction.rollback();

}

e.printStackTrace();

} **finally** {

**if** (session != **null**) {

session.close();

}

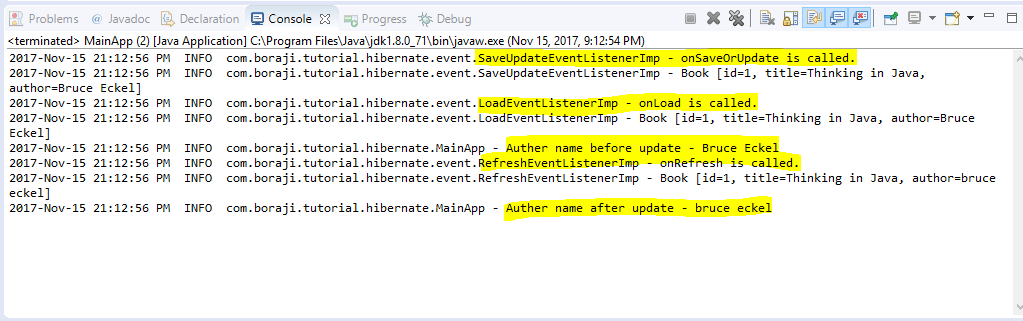
}

HibernateUtil.shutdown();

}

}

После выполнения  MainApp, вывод вашей программы будет выглядеть следующим образом.



# Интерцепторы

Интерцепторы использовать когда недостаточно готовых интерфейсов в Событиях.

Интерцептор это интерфейс для вызовов пользовательского кода из Session во время изменения состояния какой-либо сущности. Для использования интерцепторов надо реализовать интерфейс Interceptor или, что удобнее, расширить класс EmptyInterceptor. Последний уже содержит пустые реализации методов интерфейса Interceptor, что позволяет реализовывать только нужные методы.

Для примера я напишу интерцептор, который будет проверять корректность серии/номера паспорта (на самом деле нет) и кидать исключение, если данные ему не нравятся:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | /\*\*  \* Check that passport series/no is valid.  \*/  public class PassportValidationInterceptor extends EmptyInterceptor {        @Override      public boolean onFlushDirty(Object entity, Serializable id, Object[] currentState, Object[] previousState, String[] propertyNames, Type[] types) {          if (entity instanceof Passport) {              Passport p = (Passport) entity;              if (p.getSeries().length() !=2 || p.getNo().length() != 6) {                  throw new IllegalArgumentException("Incorrect passport data");              }          }          return super.onFlushDirty(entity, id, currentState, previousState, propertyNames, types);      }  } |

В интерцепторе я перехватываю сохранение сущности, проверяю её тип и если это паспорт, проверяю данные на корректность.

В результате попытка сохранить паспорт с неверными данными приводит к провалу:

|  |  |
| --- | --- |
| 1  2  3  4 | //Try to save invalid passport  Passport p = (Passport) session.createCriteria(Passport.class).add(Restrictions.eq("series", "AS")).uniqueResult();  p.setSeries("INVALID");  session.save(p); |
| 1 | ERROR: HHH000346: Error during managed flush [Incorrect passport data] |

Интерцептор может быть применён:

а). на уровне SessionFactory:

|  |  |
| --- | --- |
| 1  2  3  4  5 | sessionFactory = new MetadataSources(registry)    .buildMetadata()    .getSessionFactoryBuilder()    .applyInterceptor(new PassportValidationInterceptor())    .build(); |

б). при конструировании конкретной Session:

|  |  |
| --- | --- |
| 1  2  3  4 | sessionFactory    .withOptions()    .interceptor(new PassportValidationInterceptor())    .openSession(); |

Понятно, что в первом случае интерцептор будет назначаться каждой новой Session автоматически.

# Как склонировать объект в Hibernate

Опубликовано [**Август 25, 2016**](https://easyjava.ru/data/hibernate/kak_sklonirovat_object_v_hibernate/) автор [**EasyJava**](https://easyjava.ru/author/akashihi/) — [**Нет комментариев ↓**](https://easyjava.ru/data/hibernate/kak_sklonirovat_object_v_hibernate/#disqus_thread)

Вопрос небольшой, но довольно популярный — как загрузить объект из базы и сохранить его точную копию, но с другим id?

Решение простое — необходимо загрузить требуемый объект, через Session.get()  или запрос, не важно. Затем отсоединить его от persistence context вызовом  detach(), сбросить id и сохранить:

Java

|  |  |
| --- | --- |
| 1  2  3  4 | SomeEntity e = session.get(SomeEntity.class, 1);  session.detach(e);  e.setId(null);  session.save(e); |

**Использование пулов соединений в Hibernate**

Опубликовано [**Сентябрь 15, 2016**](https://easyjava.ru/data/pool/ispolzovanie-pulov-soedinenij-v-hibernate/) автор [**EasyJava**](https://easyjava.ru/author/akashihi/) — [**Нет комментариев ↓**](https://easyjava.ru/data/pool/ispolzovanie-pulov-soedinenij-v-hibernate/#disqus_thread)

Когда мы [настраиваем связь с базой данных в Hibernate](http://easyjava.ru/data/hibernate/hello-hibernate-bez-jpa/), мы подкладываем под приложение мину замедленного действия. Дело в том, что как и в случае в JDBC, [Hibernate](https://easyjava.ru/tag/hibernate/) открывает ровно *одно*соединение с базой данных и разрешает только *одному*объекту Session использовать его. Это прекрасно работает в тестовых примерах, где только одно соединение и требуется. Это будет работать в однопоточных программах, которые не смогут использовать больше одной Session в единицу времени, но это приведёт к проблемам в реальном многопоточном приложении.  В самом деле, если у вас web приложение, то скорее всего в нём будет несколько нитей, которые обслуживают запросы. Если речь идёт о ETL приложении, то наверняка данные, над которыми оно работает, будут разделены на блоки, которые будут перерабатывать несколько параллельных потоков. Даже настольные приложения сейчас имеют несколько потоков. А соединение то у нас с базой одно. И это проблема.

К счастью решение этой проблемы известно: [пулы соединений с базой](http://easyjava.ru/data/puly-soedinenij/) и Hibernate их отлично поддерживает. Я возьму [пример управления сущностями в Hibernate](http://easyjava.ru/data/hibernate/upravlenie-sushhnostyami-v-hibernate/) и добавлю к нему два пула соединений.

[DBCP](https://easyjava.ru/tag/dbcp/) это ещё одна библиотека для создания [пулов соединений](http://easyjava.ru/data/puly-soedinenij/). Вместе с [HikariCP](http://easyjava.ru/data/pool/nastrojka-hikaricp/) и [c3p0](http://easyjava.ru/data/pool/nastrojka-c3p0/) они составляют триумвират наиболее популярных библиотек пулов для java. DBCP разрабатывается [The Apache Foundation](http://apache.org/), что сделало его некоторым образом тяжёловесным.

К сожалению поддержка DBCP в Hibernate не очень работоспособна и я не могу рекомендовать её использовать.

[**HikariCP**](http://easyjava.ru/data/pool/nastrojka-hikaricp/)

Hibernate имеет встроенную поддержку [HikariCP](https://easyjava.ru/tag/hikaricp/). Достаточно лишь указать [HikariCP](https://easyjava.ru/tag/hikaricp/) в качестве источника соединений и настроить сам [HikariCP](https://easyjava.ru/tag/hikaricp/). Начать, конечно, нужно с включения [HikariCP](https://easyjava.ru/tag/hikaricp/) в сборку:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | <properties>    <hikaricp.version>2.4.3</hikaricp.version>  </properties>    <dependencies>    <dependency>      <groupId>com.zaxxer</groupId>      <artifactId>HikariCP</artifactId>      <version>${hikaricp.version}</version>    </dependency>  </dependencies> |

Подключение HikariCP и настройка его параметров осуществляется в файле конфигурации Hibernate:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | <hibernate-configuration>    <session-factory>      <property name="hibernate.hbm2ddl.auto">update</property>      <property name="hibernate.dialect">org.hibernate.dialect.H2Dialect</property>      <property name="hibernate.connection.provider\_class">com.zaxxer.hikari.hibernate.HikariConnectionProvider</property>      <property name="hibernate.hikari.dataSourceClassName">org.h2.jdbcx.JdbcDataSource</property>      <property name="hibernate.hikari.dataSource.url">jdbc:h2:mem:test</property>        <mapping class="ru.easyjava.data.hibernate.entity.Operation"/>    </session-factory>  </hibernate-configuration> |

Опция *hibernate.connection.provider\_class*явно включает в Hibernate поддержку HikariCP. Опции начинающиеся на *hibernate.hikari.\**передаются непосредственно в HikariCP и [настраивают его](http://easyjava.ru/data/pool/nastrojka-hikaricp/).

[**c3p0**](http://easyjava.ru/data/pool/nastrojka-c3p0/)

Hibernate имеет встроенную первоклассную поддержку c3p0. Достаточно лишь положить необходимые библиотеки в classpath и задать любую опцию c3p0. Необходимых библиотек две: сам c3p0 и библиотека поддержки от hibernate.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | <properties>    <c3p0.version>0.9.5.2</c3p0.version>    <hibernate.version>5.2.1.Final</hibernate.version>  </properties>    <dependencies>    <dependency>      <groupId>com.mchange</groupId>      <artifactId>c3p0</artifactId>      <version>${c3p0.version}</version>    </dependency>      <dependency>      <groupId>org.hibernate</groupId>      <artifactId>hibernate-core</artifactId>      <version>${hibernate.version}</version>    </dependency>      <dependency>      <groupId>org.hibernate</groupId>      <artifactId>hibernate-c3p0</artifactId>      <version>${hibernate.version}</version>    </dependency>  </dependencies> |

Hibernate настраивается как обычно, [параметры c3p0](http://easyjava.ru/data/pool/nastrojka-c3p0/) передаются с префиксом *hibernate.c3p0*

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | <hibernate-configuration>    <session-factory>      <property name="hibernate.hbm2ddl.auto">update</property>      <property name="hibernate.dialect">org.hibernate.dialect.H2Dialect</property>      <property name="hibernate.connection.url">jdbc:h2:mem:test</property>      <property name="hibernate.c3p0.min\_size">5</property>        <mapping class="ru.easyjava.data.hibernate.entity.Operation"/>    </session-factory>  </hibernate-configuration> |

Hibernate не анализирует содержимое параметров с префиксом *hibernate.c3p0*, но требует наличия по меньшей мере одного такого параметра для автоматического включения пула c3p0.

Код примера доступен на [github](https://github.com/EasyJavaRu/data-hibernate-pools).

**Hibernate Database multitenancy**

[Multitenancy](https://easyjava.ru/tag/multitenancy/) (мультиарендность) — это подход к проектированию приложения, когда один экземпляр приложений обслуживает несколько клиентов с непересекающимися наборами данных. Например сайт по учёту персональных финансов имеет одну копию кода, одно хранилище данных и много клиентов, при этом каждому клиенту доступны только его собственные данные. Наиболее популярен этот подход, по очевидным причинам, в облачных SaaS решениях — каждый клиент видит общий, единый, экземпляр приложения как свою собственную копию.

Чаще всего проблемы с реализацией multitenancy кроются не на уровне логики приложения, а на уровне хранения данных: мы должны уметь отделять данные одного клиента от другого и не давать им шанса смешаться. Существует три основных подхода к решению этой проблемы:

* Разделение на уровне базы данных — под каждого клиента создаётся (или запускается) отдельный экземпляр БД, уникальный для этого клиента и в нём распологаются данные только этого клиента.
* Разделение на уровне схемы — в одной и той же базе данных создаются разные схемы (schema/namespace) в которых создаются копии структуры таблиц, необходимых для работы клиента и данные каждого клиента живут в отдельно схеме.
* Разделение на уровне таблиц — и база данных одна, и схема одна, и даже таблицы те же самые. Но в каждой таблице заводится столбец (дискриминатор), указывающий, какому клиенту принадлежит какая таблица.

Как я понял, при использовании мультиарендного подхода к проектированию выполняются следующие шаги:

1. создаются классы с даными,
2. на основе классов с данными на сервере создается БД тремя разными способами описанными выше,
3. создается собственный класс, в котором написаны механизмы работы,
4. в файл *hibernate.cfg.xml* или программно в ходе создания *SessionFactory* прописывается созданный класс и наименование типа проектирования,
5. при открытии сессии *(Session session = sessionFactory .withOptions().****tenantIdentifier("au")****.openSession();)* в методе *tenantIdentifier()* прописывается название БД или схемы.

**Подготовка базы данных**

В режиме *database multitenancy* hibernate сам *не* создаёт базы и *не* создаёт структуру хранения, всё это авторы приложения должны сделать вручную. Для этой статьи я использую [PostgreSQL](https://easyjava.ru/tag/postgresql/), в котором создам три базы данных, одна по умолчанию, две другие для двух разных клиентов. Клиентские базы я наполню таблицами вручную, а базу по умолчанию оставлю пустой.

**SQL скрипт, создающий структуру данных**

CREATE ROLE test WITH PASSWORD 'test';

ALTER ROLE test WITH LOGIN;

CREATE DATABASE test OWNER test;

CREATE DATABASE AU OWNER test;

CREATE DATABASE DE OWNER test;

**\c AU**

CREATE TABLE address (

    id bigint NOT NULL,

    building character varying(255),

    city character varying(255),

    street character varying(255)

);

ALTER TABLE address OWNER TO test;

CREATE TABLE company (

    id bigint NOT NULL,

    name character varying(255)

);

ALTER TABLE company OWNER TO test;

CREATE SEQUENCE hibernate\_sequence

    START WITH 1

    INCREMENT BY 1

    NO MINVALUE

    NO MAXVALUE

    CACHE 1;

ALTER TABLE hibernate\_sequence OWNER TO test;

CREATE TABLE passport (

    id bigint NOT NULL,

    issuedate date,

    no character varying(255),

    series character varying(255),

    validity bytea

);

ALTER TABLE passport OWNER TO test;

CREATE TABLE person (

    id bigint NOT NULL,

    dob date,

    firstname character varying(255),

    lastname character varying(255),

    passport\_id bigint NOT NULL,

    person\_id bigint NOT NULL

);

ALTER TABLE person OWNER TO test;

CREATE TABLE person\_companies (

    person\_id bigint NOT NULL,

    company\_id bigint NOT NULL

);

ALTER TABLE person\_companies OWNER TO test;

ALTER TABLE ONLY address

    ADD CONSTRAINT address\_pkey PRIMARY KEY (id);

ALTER TABLE ONLY company

    ADD CONSTRAINT company\_pkey PRIMARY KEY (id);

ALTER TABLE ONLY passport

    ADD CONSTRAINT passport\_pkey PRIMARY KEY (id);

ALTER TABLE ONLY person

    ADD CONSTRAINT person\_pkey PRIMARY KEY (id);

ALTER TABLE ONLY person

    ADD CONSTRAINT uk\_33yyiniy3o7irjyb5nbmidt4u UNIQUE (passport\_id);

ALTER TABLE ONLY person\_companies

    ADD CONSTRAINT fk2xnkexg6vm6l0346lru8r6qaa FOREIGN KEY (company\_id) REFERENCES company(id);

ALTER TABLE ONLY person

    ADD CONSTRAINT fk701a8b9b2kw01q32ws1rc42bp FOREIGN KEY (person\_id) REFERENCES address(id);

ALTER TABLE ONLY person\_companies

    ADD CONSTRAINT fkj1do3s4ycpku0fpmm3pinysau FOREIGN KEY (person\_id) REFERENCES person(id);

ALTER TABLE ONLY person

    ADD CONSTRAINT fkn3sbguup9nkxgiqxgfc0sigxj FOREIGN KEY (passport\_id) REFERENCES passport(id);

**\c DE**

CREATE TABLE address (

    id bigint NOT NULL,

    building character varying(255),

    city character varying(255),

    street character varying(255)

);

ALTER TABLE address OWNER TO test;

CREATE TABLE company (

    id bigint NOT NULL,

    name character varying(255)

);

ALTER TABLE company OWNER TO test;

CREATE SEQUENCE hibernate\_sequence

    START WITH 1

    INCREMENT BY 1

    NO MINVALUE

    NO MAXVALUE

    CACHE 1;

ALTER TABLE hibernate\_sequence OWNER TO test;

CREATE TABLE passport (

    id bigint NOT NULL,

    issuedate date,

    no character varying(255),

    series character varying(255),

    validity bytea

);

ALTER TABLE passport OWNER TO test;

CREATE TABLE person (

    id bigint NOT NULL,

    dob date,

    firstname character varying(255),

    lastname character varying(255),

    passport\_id bigint NOT NULL,

    person\_id bigint NOT NULL

);

ALTER TABLE person OWNER TO test;

CREATE TABLE person\_companies (

    person\_id bigint NOT NULL,

    company\_id bigint NOT NULL

);

ALTER TABLE person\_companies OWNER TO test;

ALTER TABLE ONLY address

    ADD CONSTRAINT address\_pkey PRIMARY KEY (id);

ALTER TABLE ONLY company

    ADD CONSTRAINT company\_pkey PRIMARY KEY (id);

ALTER TABLE ONLY passport

    ADD CONSTRAINT passport\_pkey PRIMARY KEY (id);

ALTER TABLE ONLY person

    ADD CONSTRAINT person\_pkey PRIMARY KEY (id);

ALTER TABLE ONLY person

    ADD CONSTRAINT uk\_33yyiniy3o7irjyb5nbmidt4u UNIQUE (passport\_id);

ALTER TABLE ONLY person\_companies

    ADD CONSTRAINT fk2xnkexg6vm6l0346lru8r6qaa FOREIGN KEY (company\_id) REFERENCES company(id);

ALTER TABLE ONLY person

    ADD CONSTRAINT fk701a8b9b2kw01q32ws1rc42bp FOREIGN KEY (person\_id) REFERENCES address(id);

ALTER TABLE ONLY person\_companies

    ADD CONSTRAINT fkj1do3s4ycpku0fpmm3pinysau FOREIGN KEY (person\_id) REFERENCES person(id);

ALTER TABLE ONLY person

    ADD CONSTRAINT fkn3sbguup9nkxgiqxgfc0sigxj FOREIGN KEY (passport\_id) REFERENCES passport(id);

В скрипте выше и в коде примера используется схема данных из [примера управления сущностями в Hibernate](https://easyjava.ru/data/hibernate/upravlenie-sushhnostyami-v-hibernate/).

**Настройка Hibernate**

Включение multitenancy в hibernate производится в два действия — выбираем стратегию разделения данных и реализуем дружественный этой стратегии метод переключения соединений.

Стратегию проще всего назначить в *hibernate.cfg.xml,*хотя можно и программно, в ходе создания SessionFactory.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | <hibernate-configuration>    <session-factory>      <property name="hibernate.multiTenancy">DATABASE</property>      <property name="hibernate.multi\_tenant\_connection\_provider">ru.easyjava.data.hibernate.multitenancy.MultitenantDatabaseProvider </property>        <property name="hibernate.hbm2ddl.auto">update</property>      <property name="hibernate.dialect">org.hibernate.dialect.PostgreSQL94Dialect</property>        <!-- mappings skipped -->    </session-factory>    </hibernate-configuration> |

Свойство «*hibernate.multiTenancy*» выбирает стратегию разделения данных. В рамках моего примера я выбираю «*DATABASE*«, так как планирую данные клиентов хранить в разных базах данных. Обратите внимание, что в названии свойства «*hibernate.multiTenancy»*присутствует заглавная *T*,  если написать всё в нижнем регистре, multitenancy не включится.

Второе свойство, «*hibernate.multi\_tenant\_connection\_provider*» выбирает реализацию провайдера *(самостоятельно написанный кдасс с механизмами работы* MultitenantDatabaseProvider*)* соединений с базой данных, который будет возвращать соединение сообразно переданному в него *tenant id —* идентификатору клиента.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40 | /\*\*  \* Database selecting multitenant provider.  \*  \* Works with H2 database and HikariCP pool.  \*/  public class MultitenantDatabaseProvider extends AbstractMultiTenantConnectionProvider {      /\*\*       \* Provides common db parameters.       \* @return parameters map.       \*/      private Map<String,String> dbParameters() {          Map<String,String> parameters = new HashMap<>();          parameters.put("hibernate.hikari.dataSourceClassName", "org.postgresql.ds.PGSimpleDataSource");          parameters.put("hibernate.hikari.username", "test");          parameters.put("hibernate.hikari.password", "test");          parameters.put("hibernate.hikari.maximumPoolSize", "2");            return parameters;      }        @Override      protected ConnectionProvider getAnyConnectionProvider() {          Map<String,String> parameters = dbParameters();          parameters.put("hibernate.hikari.dataSource.url", "jdbc:postgresql://127.0.0.1:5432/test");            HikariConnectionProvider p =new HikariConnectionProvider();          p.configure(parameters);          return p;      }        @Override      protected ConnectionProvider selectConnectionProvider(String s ) { //s – это id записи          Map<String,String> parameters = dbParameters();          parameters.put("hibernate.hikari.dataSource.url", "jdbc:postgresql://127.0.0.1:5432/" + s);            HikariConnectionProvider p =new HikariConnectionProvider();          p.configure(parameters);          return p;      }  } |

Провайдер должен реализовать два метода: selectConnectionProvider() и getAnyConnectionProvider(). Первый метод достаточно очевиден — он получает *tenant id*, то есть идентификатор клиента, и строит по нему соединение. В моём случае наименование базы и идентификатор клиента совпадают.

Второй метод используется как метод получения соединения «по умолчанию», когда *tenant id* неизвестен или не требуется, например при старте Hibernate и сканировании таблиц. При этом в базе данных, которую вернёт getAnyConnectionProvider(), будет сгенерирована структура данных, если Hibernate соответствующим образом настроен. Именно поэтому при создании трёх баз данных, структуру данных я создал только в двух.

Так же в провайдере выше показывается, что можно создавать любые конфигурации соединений, [использовать пулы соединений](https://easyjava.ru/data/pool/ispolzovanie-pulov-soedinenij-v-hibernate/) и вообще всё, что придёт в голову.

**Использование Multitenancy**

Использовать multitenancy просто: при открытии сессии достаточно указать *tenant id* и всё остальное Hibernate сделает сам. Для примера я создам два разных объекта с двумя разными *tenant id* и потом выберу все объекты с разными *tenant id*

**Код создания объектов**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | @Test  public void testGreeter() {    writeAUPerson();    writeDEPerson();      Session session = sessionFactory      .withOptions()      .tenantIdentifier**("au"**)      .openSession();    session.beginTransaction();      session      .createCriteria(Person.class)      .list()      .stream()      .forEach(System.out::println);      session.getTransaction().commit();    session.close();      session = sessionFactory      .withOptions()      .tenantIdentifier**("de")**      .openSession();    session.beginTransaction();      session      .createCriteria(Person.class)      .list()      .stream()      .forEach(System.out::println);      session.getTransaction().commit();    session.close();  } |

Запуск примера показывает, что данные для одного и того же запроса возвращаются разные, в зависимости от *tenant id*:

|  |  |
| --- | --- |
| 1  2 | Person{firstName='Test', lastName='Testoff', dob=2016-10-03, passport=Passport{series='AS', no='123456', issueDate=2016-10-03, validity=P20Y, owner=Testoff}, primaryAddress=Address{city='Kickapoo', street='Main street', building='1', tenants=Test}, workingPlaces=[Company{name='Acme Ltd', workers=Test}]}  Person{firstName='Johan', lastName='von Testoff', dob=2016-10-03, passport=Passport{series='RY', no='654321', issueDate=2016-10-03, validity=P20Y, owner=von Testoff}, primaryAddress=Address{city='Oberdingeskirchen', street='Hbf Platz', building='1', tenants=Johan}, workingPlaces=[Company{name='Acme Ltd', workers=Johan}]} |

Код примера доступен на [github](https://github.com/EasyJavaRu/data-hibernate-multitenancydatabase). Для запуска примера требуется установить PostgreSQL сервер и разрешить к нему доступ. Если сервер будет установлен не на локальной машине, требуется изменить его адрес в jdbc url.

**Hibernate Schema Multitenancy**

**Подготовка базы данных**

В режиме schema *multitenancy* hibernate сам *не* создаёт схемы для разных клиентов и *не* создаёт структуру хранения, всё это авторы приложения должны сделать вручную. Для этой статьи я использую [PostgreSQL](https://easyjava.ru/tag/postgresql/), в котором создам пустую базу данных и две схемы для двух разных клиентов. Клиентские схемы я наполню таблицами вручную.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | //ROLE – роль (список разрешенных действий с БД)  CREATE ROLE test WITH PASSWORD 'test'; //создаем роль test с паролем test  ALTER ROLE test WITH LOGIN; //возможно это обнуление пароля    CREATE DATABASE test OWNER test; // OWNER test означает, что владельцем БД назначается роль test  CREATE DATABASE AU OWNER test;  CREATE DATABASE DE OWNER test;    \c test   // SCHEMA это пространство имен в БД  CREATE SCHEMA au;  ALTER SCHEMA au OWNER TO test; // TO test означает, что владельцем схемы назначается роль test  GRANT USAGE ON SCHEMA au TO public; //воозможно разрешение всем пользоваться данными находящимися в схеме    CREATE SCHEMA de;  ALTER SCHEMA de OWNER TO test;  GRANT USAGE ON SCHEMA de TO public; |

**SQL скрипт, создающий структуру данных**

**SET SCHEMA 'au';**

CREATE TABLE address (

    id bigint NOT NULL,

    building character varying(255),

    city character varying(255),

    street character varying(255)

);

ALTER TABLE address OWNER TO test;

CREATE TABLE company (

    id bigint NOT NULL,

    name character varying(255)

);

ALTER TABLE company OWNER TO test;

CREATE SEQUENCE hibernate\_sequence

    START WITH 1

    INCREMENT BY 1

    NO MINVALUE

    NO MAXVALUE

    CACHE 1;

ALTER TABLE hibernate\_sequence OWNER TO test;

CREATE TABLE passport (

    id bigint NOT NULL,

    issuedate date,

    no character varying(255),

    series character varying(255),

    validity bytea

);

ALTER TABLE passport OWNER TO test;

CREATE TABLE person (

    id bigint NOT NULL,

    dob date,

    firstname character varying(255),

    lastname character varying(255),

    passport\_id bigint NOT NULL,

    person\_id bigint NOT NULL

);

ALTER TABLE person OWNER TO test;

CREATE TABLE person\_companies (

    person\_id bigint NOT NULL,

    company\_id bigint NOT NULL

);

ALTER TABLE person\_companies OWNER TO test;

ALTER TABLE ONLY address

    ADD CONSTRAINT address\_pkey PRIMARY KEY (id);

ALTER TABLE ONLY company

    ADD CONSTRAINT company\_pkey PRIMARY KEY (id);

ALTER TABLE ONLY passport

    ADD CONSTRAINT passport\_pkey PRIMARY KEY (id);

ALTER TABLE ONLY person

    ADD CONSTRAINT person\_pkey PRIMARY KEY (id);

ALTER TABLE ONLY person

    ADD CONSTRAINT uk\_33yyiniy3o7irjyb5nbmidt4u UNIQUE (passport\_id);

ALTER TABLE ONLY person\_companies

    ADD CONSTRAINT fk2xnkexg6vm6l0346lru8r6qaa FOREIGN KEY (company\_id) REFERENCES company(id);

ALTER TABLE ONLY person

    ADD CONSTRAINT fk701a8b9b2kw01q32ws1rc42bp FOREIGN KEY (person\_id) REFERENCES address(id);

ALTER TABLE ONLY person\_companies

    ADD CONSTRAINT fkj1do3s4ycpku0fpmm3pinysau FOREIGN KEY (person\_id) REFERENCES person(id);

ALTER TABLE ONLY person

    ADD CONSTRAINT fkn3sbguup9nkxgiqxgfc0sigxj FOREIGN KEY (passport\_id) REFERENCES passport(id);

**SET SCHEMA 'de';**

CREATE TABLE address (

    id bigint NOT NULL,

    building character varying(255),

    city character varying(255),

    street character varying(255)

);

ALTER TABLE address OWNER TO test;

CREATE TABLE company (

    id bigint NOT NULL,

    name character varying(255)

);

ALTER TABLE company OWNER TO test;

CREATE SEQUENCE hibernate\_sequence

    START WITH 1

    INCREMENT BY 1

    NO MINVALUE

    NO MAXVALUE

    CACHE 1;

ALTER TABLE hibernate\_sequence OWNER TO test;

CREATE TABLE passport (

    id bigint NOT NULL,

    issuedate date,

    no character varying(255),

    series character varying(255),

    validity bytea

);

ALTER TABLE passport OWNER TO test;

CREATE TABLE person (

    id bigint NOT NULL,

    dob date,

    firstname character varying(255),

    lastname character varying(255),

    passport\_id bigint NOT NULL,

    person\_id bigint NOT NULL

);

ALTER TABLE person OWNER TO test;

CREATE TABLE person\_companies (

    person\_id bigint NOT NULL,

    company\_id bigint NOT NULL

);

ALTER TABLE person\_companies OWNER TO test;

ALTER TABLE ONLY address

    ADD CONSTRAINT address\_pkey PRIMARY KEY (id);

ALTER TABLE ONLY company

    ADD CONSTRAINT company\_pkey PRIMARY KEY (id);

ALTER TABLE ONLY passport

    ADD CONSTRAINT passport\_pkey PRIMARY KEY (id);

ALTER TABLE ONLY person

    ADD CONSTRAINT person\_pkey PRIMARY KEY (id);

ALTER TABLE ONLY person

    ADD CONSTRAINT uk\_33yyiniy3o7irjyb5nbmidt4u UNIQUE (passport\_id);

ALTER TABLE ONLY person\_companies

    ADD CONSTRAINT fk2xnkexg6vm6l0346lru8r6qaa FOREIGN KEY (company\_id) REFERENCES company(id);

ALTER TABLE ONLY person

    ADD CONSTRAINT fk701a8b9b2kw01q32ws1rc42bp FOREIGN KEY (person\_id) REFERENCES address(id);

ALTER TABLE ONLY person\_companies

    ADD CONSTRAINT fkj1do3s4ycpku0fpmm3pinysau FOREIGN KEY (person\_id) REFERENCES person(id);

ALTER TABLE ONLY person

    ADD CONSTRAINT fkn3sbguup9nkxgiqxgfc0sigxj FOREIGN KEY (passport\_id) REFERENCES passport(id);

Схема данных, код сущностей и код примера идентичны таковым из [примера *database multitenancy*](https://easyjava.ru/data/hibernate/hibernate-database-multitenancy/).

**Настройка Hibernate**

Включение multitenancy в hibernate производится в два действия — выбираем стратегию разделения данных и реализуем дружественный этой стратегии метод переключения соединений.

Стратегию проще всего назначить в *hibernate.cfg.xml,*хотя можно и программно, в ходе создания SessionFactory.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | <hibernate-configuration>    <session-factory>      <property name="hibernate.multiTenancy">SCHEMA</property>      <property name="hibernate.multi\_tenant\_connection\_provider">ru.easyjava.data.hibernate.multitenancy.MultitenantSchemaProvider</property>        <property name="hibernate.dialect">org.hibernate.dialect.PostgreSQL94Dialect</property>        <!-- mappings skipped -->    </session-factory>    </hibernate-configuration> |

Свойство «*hibernate.multiTenancy*» выбирает стратегию разделения данных. В рамках моего примера я выбираю «*SCHEMA*«, так как планирую данные клиентов хранить в разных схемах одной базы данных. Обратите внимание, что в названии свойства «*hibernate.multiTenancy»*присутствует заглавная *T*,  если написать всё в нижнем регистре, multitenancy не включится.

Второе свойство, «*hibernate.multi\_tenant\_connection\_provider*» выбирает реализацию провайдера *(самостоятельно написанный кдасс с механизмами работы* MultitenantSchemaProvider*)* соединений с базой данных, который будет возвращать соединение сообразно переданному в него *tenant id —* идентификатору клиента.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82 | /\*\*  \* Schema selecting multitenant provider.  \*  \* Works with PgSQL database and HikariCP pool.  \*/  public class MultitenantSchemaProvider implements MultiTenantConnectionProvider {    /\*\*     \* Connection pool.     \*/    private HikariDataSource connectionProvider;      /\*\*     \* Устанавливам схему в БД.     \* @param c connection to operate on     \* @param schema tenant id     \* @throws SQLException is thrown when not able     \* to return connection to the pool.     \*/    private void **setSchemaTo**(Connection c, String schema) throws SQLException {      try {        c.createStatement().execute("SET SCHEMA **'**" + schema.toLowerCase() + "**'**"); //где "SET SCHEMA '" + schema.toLowerCase() + "'" – это SQL запрос. И вся строка целиком отправляет этот запрос в БД и выведет “true” если такой запрос выполним, иначе “false”      } catch (SQLException e) {        connectionProvider.evictConnection(c); //принудительно закрывает соединение и удаляет из пула        throw new HibernateException("Error while switching schema", e);      }    }      /\*\*     \* Here we instantiate database connection pool.     \*/    public MultitenantSchemaProvider() {      HikariConfig parameters = new HikariConfig();        parameters.setDataSourceClassName("org.postgresql.ds.PGSimpleDataSource");      parameters.setUsername("test");      parameters.setPassword("test");      parameters.setMaximumPoolSize(2);      parameters.addDataSourceProperty("databaseName", "test");      parameters.addDataSourceProperty("serverName", "192.168.75.5");        connectionProvider = new HikariDataSource(parameters);    }      @Override    public Connection getAnyConnection() throws SQLException {      return connectionProvider.getConnection();    }      @Override    public void releaseAnyConnection(Connection connection) throws SQLException {      connectionProvider.evictConnection(connection);    }      @Override    public Connection getConnection(String s) throws SQLException {      Connection c = getAnyConnection();      setSchemaTo(c, s);      return c;    }      @Override    public void releaseConnection(String s, Connection connection) throws SQLException {      setSchemaTo(connection, "public");      releaseAnyConnection(connection);    }      @Override    public boolean supportsAggressiveRelease() {      return false;    }      /\* Spi related mandatory methods \*/    @Override    public boolean isUnwrappableAs(Class aClass) {      return false;    }      @Override    public <T> T unwrap(Class<T> aClass) {      return null;    }  } |

В этот раз я напрямую реализовал интерфейс MultiTenandProvider. Мой собственный код по переключению схем в базе находится в методе **setSchemaTo**, который переключает схему в базе и, если что-то идёт не так, удаляет соединение из пула и кидает исключение.

Из методов интерфейса следует обратить внимание на пары getAnyConnection()/releaseAnyConnection() и getConnection()/releaseConnection(). Первая пара использутся для получения соединения «по умолчанию», когда *tenant id* неизвестен или не требуется, например при старте Hibernate и сканировании таблиц. Вторая пара используется для работы с tenant специфичными соединениями.

Так же в моём собственном провайдере используется [пул HikariCP](https://easyjava.ru/data/pool/nastrojka-hikaricp/) для поддержания нескольких соединений с базой.

**Использование Multitenancy**

Использовать multitenancy просто: при открытии сессии достаточно указать *tenant id* и всё остальное Hibernate сделает сам. Для примера я создам два разных объекта с двумя разными *tenant id* и потом выберу все объекты с разными *tenant id*

**Код создания объектов**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | @Test  public void testGreeter() {    writeAUPerson();    writeDEPerson();      Session session = sessionFactory      .withOptions()      .tenantIdentifier**("au")**      .openSession();    session.beginTransaction();      session      .createCriteria(Person.class)      .list()      .stream()      .forEach(System.out::println);      session.getTransaction().commit();    session.close();      session = sessionFactory      .withOptions()      .tenantIdentifier**("de")**      .openSession();    session.beginTransaction();      session      .createCriteria(Person.class)      .list()      .stream()      .forEach(System.out::println);      session.getTransaction().commit();    session.close();  } |

Запуск примера показывает, что данные для одного и того же запроса возвращаются разные, в зависимости от *tenant id*:

|  |  |
| --- | --- |
| 1  2 | Person{firstName='Test', lastName='Testoff', dob=2016-10-03, passport=Passport{series='AS', no='123456', issueDate=2016-10-03, validity=P20Y, owner=Testoff}, primaryAddress=Address{city='Kickapoo', street='Main street', building='1', tenants=Test}, workingPlaces=[Company{name='Acme Ltd', workers=Test}]}  Person{firstName='Johan', lastName='von Testoff', dob=2016-10-03, passport=Passport{series='RY', no='654321', issueDate=2016-10-03, validity=P20Y, owner=von Testoff}, primaryAddress=Address{city='Oberdingeskirchen', street='Hbf Platz', building='1', tenants=Johan}, workingPlaces=[Company{name='Acme Ltd', workers=Johan}]} |

Код примера доступен на [github](https://github.com/EasyJavaRu/data-hibernate-multitenancyschema). Для запуска примера требуется установить PostgreSQL сервер и разрешить к нему доступ. Если сервер будет установлен не на локальной машине, требуется изменить его адрес коде создания пула соединений.

**Hibernate Discrimitator Multitenancy**

**Filters**

Фильтр это зараннее написанный, загружаемый с диска и параметризированный **where**

*(используется при использовании подхода разделения данных на уровне таблиц - Discrimitator Multitenancy)*

**Discrimitator Multitenancy**

*(Ниже, в теме Фильтры, будет пример использования фильтров в Hibernate, где будет показано, как ограничивать запросы по tenant id, хранящемуся в таблицах.)*

*Discrimitator (tenant id)* надо хранить и корректно заполнять.

В отличие от других подходов к multitenancy, этот подход требует внесения изменений в схему данных, в частности добавления столбца дискриминатора. Это проще всего сделать, создав отдельный @MappedSuperclass для хранения этого столбца и унаследовать все остальные сущности от него.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | @MappedSuperclass  public class AbstractDiscriminatorObject extends AbstractIdentifiableObject {        /\*\*       \* Actual tenant id value.       \*/      @Getter      @Setter      private String tenantId;  } |

@MappedSuperclass определяет класс - предок для Entity классов, описанные и аннотированные свойства и методы которого будут применены в классах-наследниках. Описанный супер класс не отображается на отдельную таблицу. Данная аннотация используется для классов, которые не описывают сущности как таковые. Задача состоит в следующем. Сущности (Entity) нашего проекта должны обладать какими-то общими свойствами и методами. В то же время, если исходить из понимания объектов самой базы данных, эти объекты не должны быть связаны между собой. Для модели базы это будут совершенно разные объекты ни связанные между собой.

Я использовал схему данных из [других примеров multitenancy](https://easyjava.ru/data/hibernate/hibernate-schema-multitenancy/) и добавил в неё дополнительный класс. В коде выше хорошо видно, что классы с @MappedSuperclass могут быть унаследованы друг от друга и создавать достаточно сложные иерархии наследования.

Последней частью имитации discriminator based multitenancy станет интерцептор, который будет заполнять *tenant id* при сохранении сущности в базу.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | public class DiscriminatorSettingInterceptor extends EmptyInterceptor {      private TenantIdResolver resolver = new TenantIdResolver();        @Override      public void preFlush(Iterator entities) {          entities.forEachRemaining(o -> {              if (o instanceof AbstractDiscriminatorObject) {                  AbstractDiscriminatorObject t = (AbstractDiscriminatorObject) o;                  if (t.getTenantId() == null) {                      String tenantId = resolver.resolveCurrentTenantIdentifier();                      if (tenantId == null) {                          throw new IllegalStateException("No tenant id specified");                      }                      t.setTenantId(tenantId);                  }              }          });          super.preFlush(entities);      }  } |

В метод *preFlush()* передаётся итератор с сохраняемыми сущностями, для каждой из которых проверяется тип и, если она содержит столбец дискриминатора и он пуст, он заполняется текущим значением *tenant id*. В качестве источника *tenant id*используется реализация интерфейса CurrentTenantIdentifierResolver, которая (в моём случае) просто хранит заранее заданное значение.

**Фильтры в**[**Hibernate**](https://easyjava.ru/tag/hibernate/)

Hibernate позволяет определить простое SQL условие, которое будет прикладываться к любом запросу. Фактически фильтр это просто параметризованное SQL условие с именем, которое в рамках одной сессии прикладывается к каждому выполняемому запросу.

Определяется фильтр в два шага:

- вначале задаётся имя фильтра и список его параметров + их типы,

- затем собственно текст запроса для фильтра.

Например, зная что мы делаем основанную на дискриминаторе multitenancy, и что в каждой таблице есть столбец дискриминатора, который называется, допустим, *tenantId*, мы могли бы написать следующий фильтр:

|  |  |
| --- | --- |
|  | @Entity  1. @FilterDefs({     @FilterDef(name = "discriminator\_filter", parameters={@ParamDef( name="tenantIdValue", type="string" )} )  })  2. @Filters({     @Filter(name="discriminator\_filter", condition="tenantId = :tenantIdValue")  })  public class Forest { ... } |

Фильтр называется discriminator\_filter, принимает параметр tenantIdValue типа String. Фильтр добавит к запросу SQL условие tenantId = :tenantIdValue.

Чтобы использовать фильтр, необходимо разрешить его применение в сессии:

|  |  |
| --- | --- |
| 1  2  3  4 | session = sessionFactory    .openSession();  session.enableFilter("discriminator\_filter")    .setParameter("tenantIdValue", "de"); |

Наконец попробуем использовать всё что мы написали выше:

**Код создания объектов**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | @Test  public void testGreeter() {    writeAUPerson();    writeDEPerson();      Session session = sessionFactory      .openSession();      session.enableFilter("discriminator\_filter")      .setParameter("tenantIdValue", "au");    session.beginTransaction();      session      .createCriteria(Person.class)      .list()      .stream()      .forEach(System.out::println);    session.getTransaction().commit();    session.close();      session = sessionFactory      .openSession();    session.enableFilter("discriminator\_filter")      .setParameter("tenantIdValue", "de");    session.beginTransaction();      session      .createCriteria(Person.class)      .list()      .stream()      .forEach(System.out::println);    session.getTransaction().commit();    session.close();  } |

Запуск примера показывает, что данные для одного и того же запроса возвращаются разные, в зависимости от *tenant id*:

|  |  |
| --- | --- |
| 1  2 | Person{firstName='Test', lastName='Testoff', dob=2016-11-04, passport=Passport{series='AS', no='123456', issueDate=2016-11-04, validity=P20Y, owner=Testoff}, primaryAddress=Address{city='Kickapoo', street='Main street', building='1', tenants=Test}, workingPlaces=[Company{name='Acme Ltd', workers=Test}]}  Person{firstName='Johan', lastName='von Testoff', dob=2016-11-04, passport=Passport{series='RY', no='654321', issueDate=2016-11-04, validity=P20Y, owner=von Testoff}, primaryAddress=Address{city='Oberdingeskirchen', street='Hbf Platz', building='1', tenants=Johan}, workingPlaces=[Company{name='Acme Ltd', workers=Johan}]} |

Код примера доступен на [github](https://github.com/EasyJavaRu/data-hibernate-multitenancydiscriminator).

**Совместимость с Flyway**

Как уже упоминалось в комментариях, Flyway и Hibernate не должны использоваться вместе для обновления схемы, но это не означает, что вы не можете использовать Hibernate вообще, чтобы помочь вам поддерживать свою схему.

Basic : Эта функция определенно не предназначена для предотвращения выполнения миграции в пустых базах данных. Это следует использовать, когда ваша база данных уже существует (т. е. у нее уже есть таблицы и данные, и вы хотите сохранить эти данные). Для пустых баз данных это бесполезно.

Пример: Предположим, у вас есть существующая база данных, созданная с помощью 2 скриптов:

*V1\_\_create\_tables.sql*

*V2\_\_create\_constraints.sql*

Теперь вы хотите управлять дальнейшими обновлениями схемы с flyway:

*V3\_\_First\_update.sql*

*V4\_\_Second\_update.sql*

*V2* - это ваш базовый уровень, означающий, что миграция базы данных не будет выполнять миграции *V1* и *V2*, потому что они уже существуют.

Если вы хотите объединить Spring Boot + Hibernate + Flyway:

- отключите автоматическое обновление схемы на hibernate - как уже упоминалось, это может быть опасно (hbm2ddl.auto=false),

- используйте Hibernate SchemaGenerator для создания большого SQL-файла, назовем его например *V1\_\_initial\_schema.sql*,

- если у вас есть база данных *V1* с данными, которые вы хотите сохранить, вы должны создать ее базовый уровень . В противном случае просто начните с пустой схемы. ( flyway clean ) и мигрируют (flyway migrate ). WARNING: “flyway clean” отбросит все ваши таблицы!

Теперь вы готовы использовать как Hibernate, так и Flyway для каждой предстоящей модификации схемы. Предположим, вы только что обновили свою модель:

- используйте Hibernate SchemaGenerator для создания такого же большого SQL-файла, назовем его *generator-output.sql*,

- сделайте сравнение side-by-side *V1\_\_initial\_schema.sql* и *generator-output.sql* . Это поможет вам определить различия между двумя файлами. На основе этих различий можно создать новый файл миграции, назовем его *V2\_\_update.sql*,

- сделайте flyway migrate на вашей базе данных.

**Hibernate и MySQL: примеры настройки**

Перед тем, как соединяться из [Hibernate](https://easyjava.ru/tag/hibernate/) с [MySQL](https://easyjava.ru/tag/mysql/), не забудьте создать пользователя, базу данных и предоставить пользователю права на неё:

|  |  |
| --- | --- |
| 1  2  3  4 | CREATE USER 'test'@'%' IDENTIFIED BY 'test';  CREATE DATABASE test;  GRANT ALL PRIVILEGES ON test.\* TO test;  FLUSH PRIVILEGES |

Команды выше создают пользователя *test* с паролем *test*, создают базу данных *test* и, наконец, дают пользователю все права на эту базу. В условиях реального приложения разумеется следует выбрать пароль посложнее, да и права раздавать поаккуратнее.

**Использование MySQL непосредственно с Hibernate**

Перед тем, как использовать MySQL, необходимо добавить его [JDBC](https://dev.mysql.com/downloads/connector/j/) драйвер в зависимости maven:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | <properties>    <mysql.version>6.0.5</mysql.version>  </properties>    <dependencies>    <dependency>      <groupId>mysql</groupId>      <artifactId>mysql-connector-java</artifactId>      <version>${mysql.version}</version>    </dependency>  </dependencies> |

С драйвером MySQL в classpath можно настраивать Hibernate:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | <hibernate-configuration>    <session-factory>        <property name="hibernate.connection.url">jdbc:mysql://127.0.0.1:3306/test</property>      <property name="hibernate.connection.username">test</property>      <property name="hibernate.connection.password">test</property>      <property name="hibernate.dialect">org.hibernate.dialect.MySQL5Dialect</property>        <!-- other settings -->    </session-factory>  </hibernate-configuration> |

JDBC url у MySQL имеет следующий формат: *jdbc:mysql://хост:порт/имябазы,*например *jdbc:mysql://127.0.0.1:3306/test.*Имя пользователя и пароль передаются отдельными параметрами. Наконец, надо не забыть переключить диалект Hibernate на MySQL, что делается в последней строке.

**Hibernate, MySQL и пул c3p0**

Однако, как мы знаем, [одно соединение не очень удобно использовать в реальном приложении](https://easyjava.ru/data/puly-soedinenij/). Поэтомунастроим [пул c3p0](https://easyjava.ru/data/pool/nastrojka-c3p0/) для совместного использования с Hibernate и MySQL.

Для этого, кроме MySQL JDBC драйвера,  надо добавить непосредственно c3p0 и библиотеку hibernate-cp30 к зависимостям maven:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | <properties>    <hibernate.version>5.2.1.Final</hibernate.version>    <c3p0.version>0.9.5.2</c3p0.version>  </properties>    <dependencies>    <dependency>      <groupId>com.mchange</groupId>      <artifactId>c3p0</artifactId>      <version>${c3p0.version}</version>    </dependency>      <dependency>      <groupId>org.hibernate</groupId>      <artifactId>hibernate-c3p0</artifactId>      <version>${hibernate.version}</version>    </dependency>  </dependencies> |

Для включения c3p0 достаточно передать по меньшей мере один параметр c3p0 в настройки Hibernate:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | <hibernate-configuration>    <session-factory>      <property name="hibernate.connection.url">jdbc:mysql://192.168.75.5:3306/test</property>      <property name="hibernate.connection.username">test</property>      <property name="hibernate.connection.password">test</property>      <property name="hibernate.dialect">org.hibernate.dialect.MySQL5Dialect</property>      <property name="hibernate.c3p0.min\_size">5</property>      <!--Other settings-->    </session-factory>  </hibernate-configuration> |

Остальные параметры остаются такими же, как и для MySQL без c3p0.

**Hibernate, MySQL и пул**[**HikariCP**](https://easyjava.ru/tag/hikaricp/)

К сожалению, в Hibernate нет настолько высококачественной поддержки HikariCP, аналогичной c3p0. Поэтому настройка HikariCP требует чуть чуть больше работы.

В первую очередь, конечно же, надо добавить ещё и артефакт HikariCP в зависимости (не забываем про MySQL JDBC драйвер):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | <properties>    <hikaricp.version>2.4.3</hikaricp.version>  </properties>    <dependencies>    <dependency>      <groupId>com.zaxxer</groupId>      <artifactId>HikariCP</artifactId>      <version>${hikaricp.version}</version>    </dependency>  </dependencies> |

Потом необходимо изменить конфигурацию Hibernate для работы с HikariCP:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | <hibernate-configuration>    <session-factory>      <property name="hibernate.connection.provider\_class">com.zaxxer.hikari.hibernate.HikariConnectionProvider</property>      <property name="hibernate.hikari.dataSourceClassName">com.mysql.cj.jdbc.MysqlDataSource</property>      <property name="hibernate.hikari.dataSource.url">jdbc:mysql://192.168.75.5:3306/test</property>      <property name="hibernate.hikari.username">test</property>      <property name="hibernate.hikari.password">test</property>      <property name="hibernate.dialect">org.hibernate.dialect.MySQL5Dialect</property>    <!-- Other settings -->    </session-factory>  </hibernate-configuration> |

Все настройки соединения с MySQL переименовываются и к ним добавляется два дополнительных параметра. Вначале в Hibernate настраивается использование HikariCP, затем указывается какой класс использовать для соединения с MySQL.