ТЕМА 1. ВВЕДЕНИЕ В HIBERNATE

1.1 ВВЕДЕНИЕ В МЕХАНИЗМ СОХРАНЕНИЯ ДАННЫХ.

Hibernate — библиотека для языка программирования Java, предназначенная для решения задач объектно-реляционного отображения (object-relational mapping — ORM). Она представляет собой свободное программное обеспечение с открытым исходным кодом (open source), распространяемое на условиях GNU Lesser General Public License. Данная библиотека предоставляет лёгкий в использовании каркас (Фреймворк) для отображения объектно-ориентированной модели данных в традиционные реляционные базы данных.

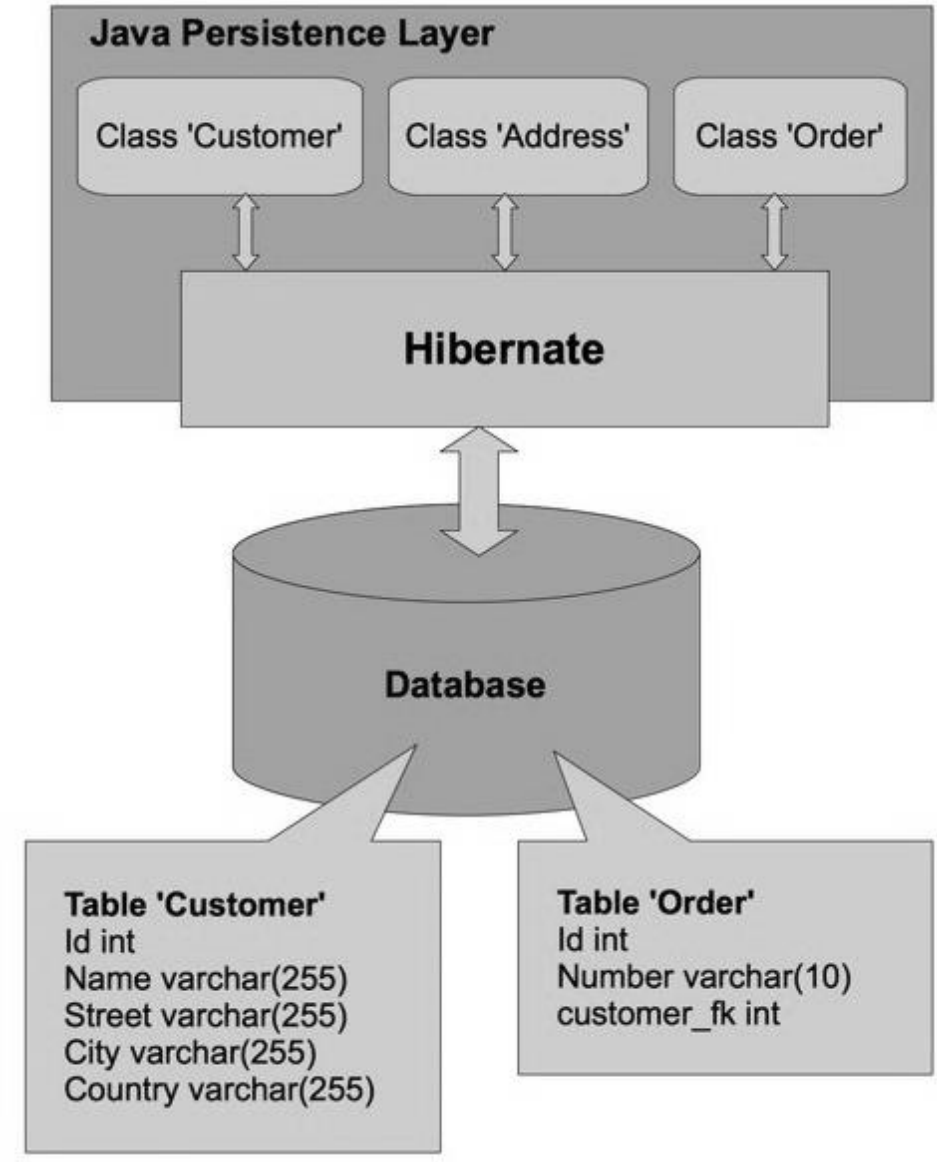

Рисунок 1.1.1 Структура проекта с Hibernate

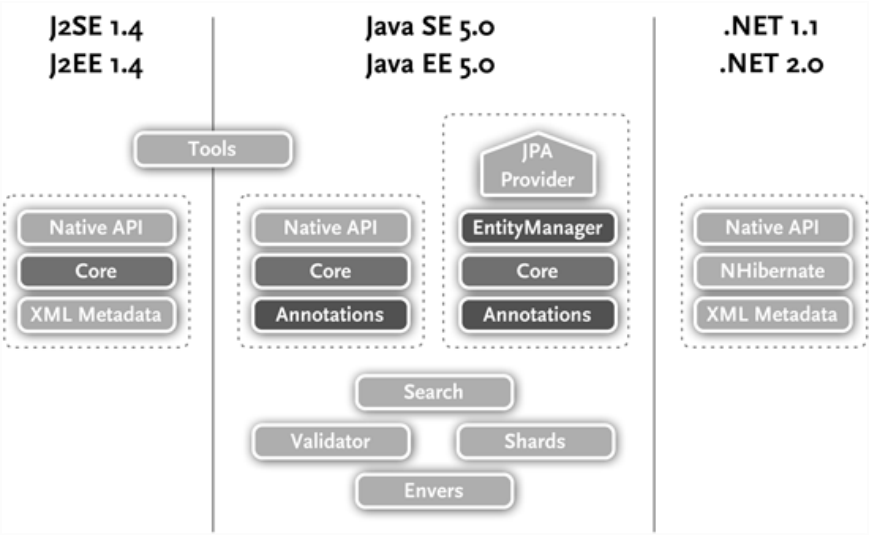

Рисунок 1.1.2 Модули Hibernate

- ***Hibernate Core*** — ядро Hibernate для Java, собственный API и метаданные отражение в формате XMHibernate Core — ядро Hibernate для Java, собственный API и метаданные отражение в формате XML.
- ***Hibernate Annotations*** — отображение с помощью аннотаций JDK 5.0, как стандартных для JPA, так и собственных расширений.
- ***Hibernate EntityManager*** — реализация Java Persistence API для Java SE и Java EE.

1.2 Объектно-реляционная модель.

POJO (англ. Plain Old Java Object) — «простой Java-объект в старом стиле», простой Java-объект, не унаследованный от какого-то специфического объекта и не реализующий никаких служебных интерфейсов сверх тех, которые нужны для бизнес-модели.

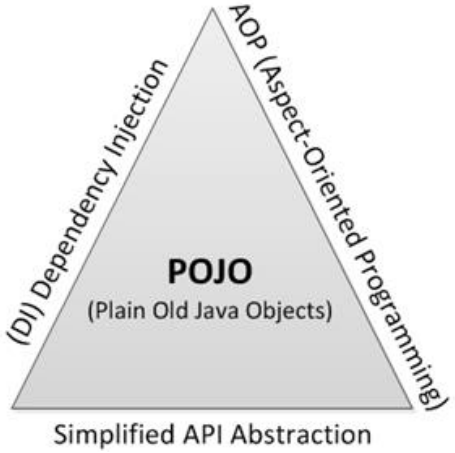

Рисунок 1.2.1 Применение POJO

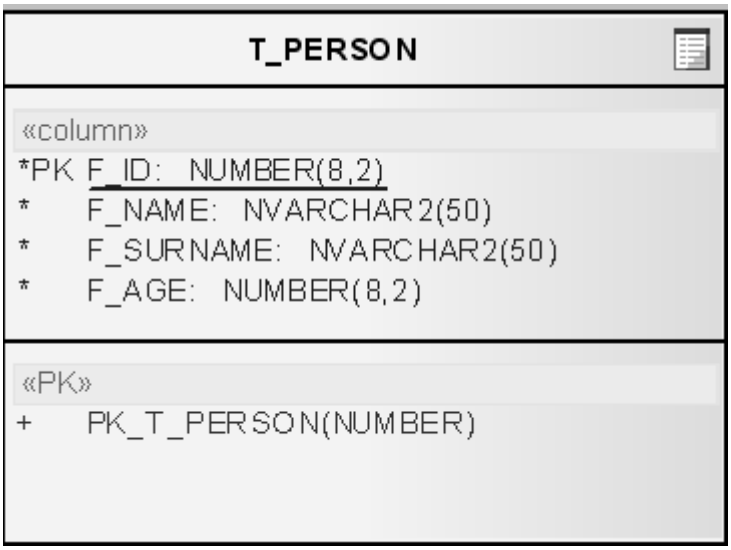

Рисунок 1.2.2 Таблица T_PERSON

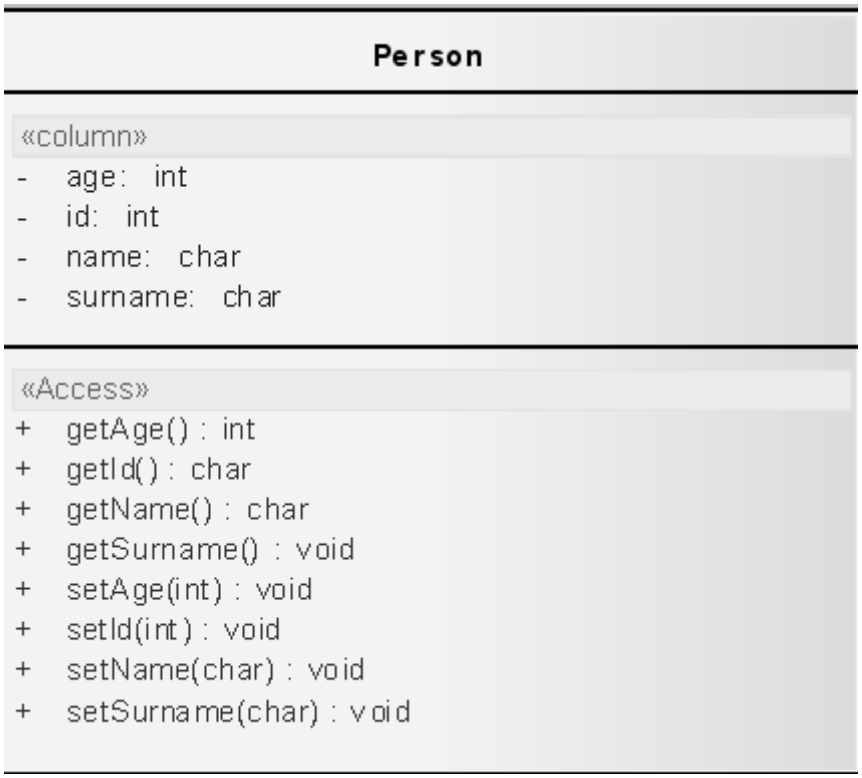

Рисунок 1.2.3 POJO Person класс

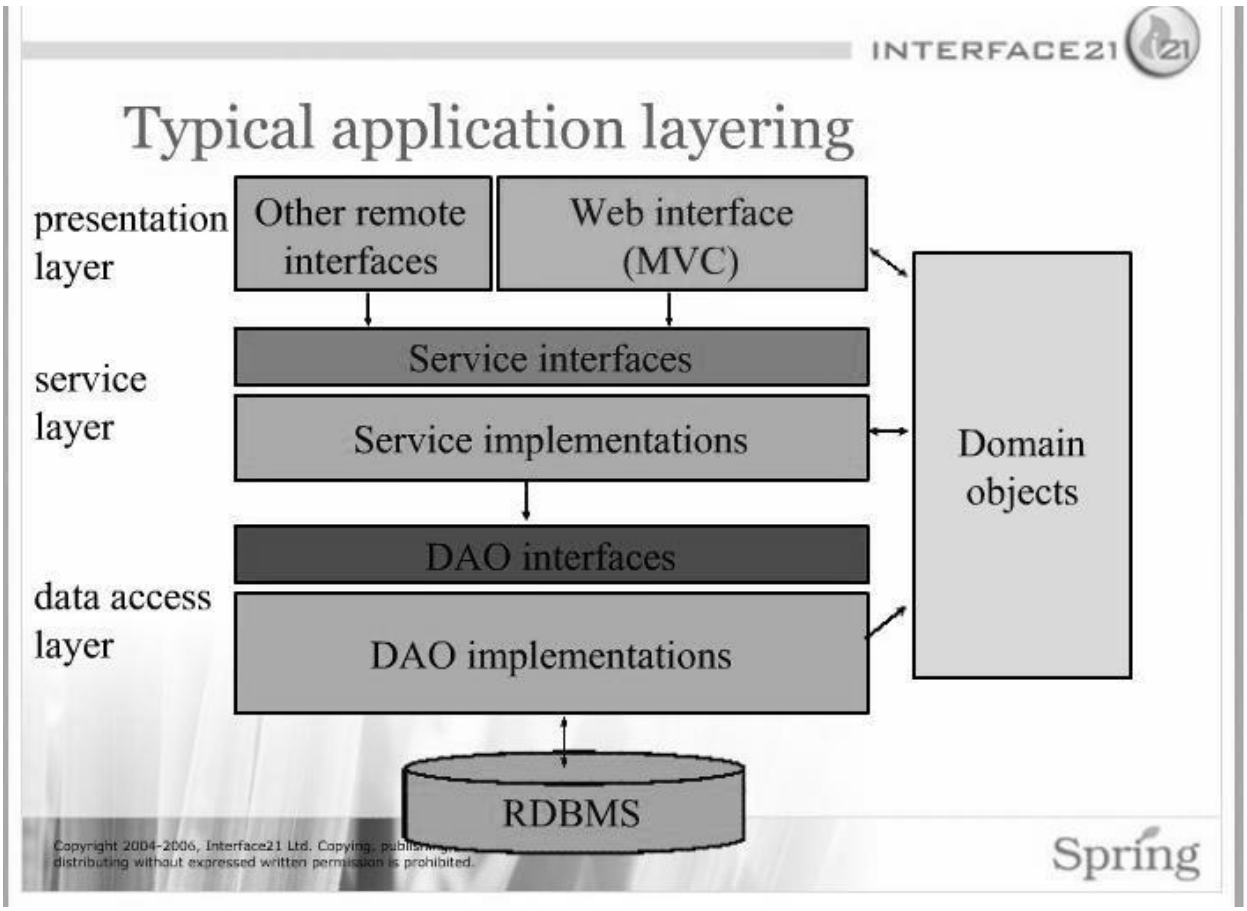

Рисунок 1.2.3 Dao архитектура

Для решения вопроса построения связей между таблицами СУБД и объектами Java существует множество подходов:

- Применение JDBC и SQL выражений для создания соотношения и обеспечения трансфера данных. Данный подход требует внимательного отношения к структуре запросов и правилам написания кода, доступности и блокировки общих ресурсов (соединений с СУБД, транзакций и т.д.). Необходимо четко и ясно представлять модель обработки исключительных ситуаций и алгоритмов освобождения критичных ресурсов, что само по себе требует незаурядных способностей в области проектирования и опыта работы с конкретными СУБД.
- Применение систем и библиотек и систем для сохранения данных из уровня java на уровень СУБД. Благодаря высокой проработки основных сложностей, возникающих при обращении к СУБД (слабые навыки работы с SQL запросами, незнание особенностей СУБД и механизмов обработки структур данных, подходы ООП языков и СУБД при обработке данных и т.д.), данные системы позволяют более быстро и качественно разрабатывать и строить сложные бизнес модели данных, при этом обеспечивая высокую производительность при работе с СУБД.

1.3 Первое приложение с использованием Hibernate.

Для сохранения данных и их обработки мы начнем строить приложение. Слой, отвечающий за работу с СУБД, будет использовать библиотеки Hibernate. Для сборки мы будем использовать maven, СУБД – MySQL.

В качестве POJO мы будем использовать Person POJO класс.

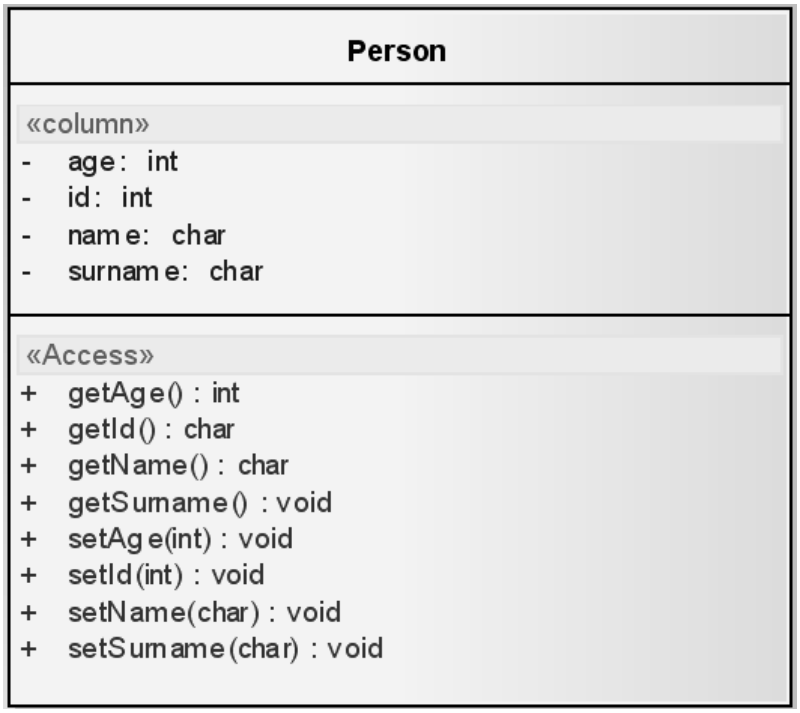

Рисунок 1.3.1 POJO Person класс

Структура DAO будет выглядеть следующим образом:

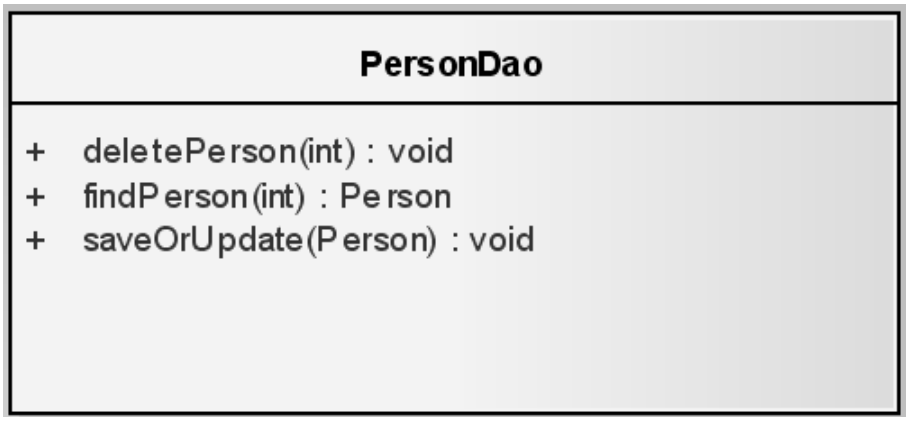

Рисунок 1.3.2 PersonDao класс

Для реализации данного приложения давайте сначала рассмотрим структуру проекта.

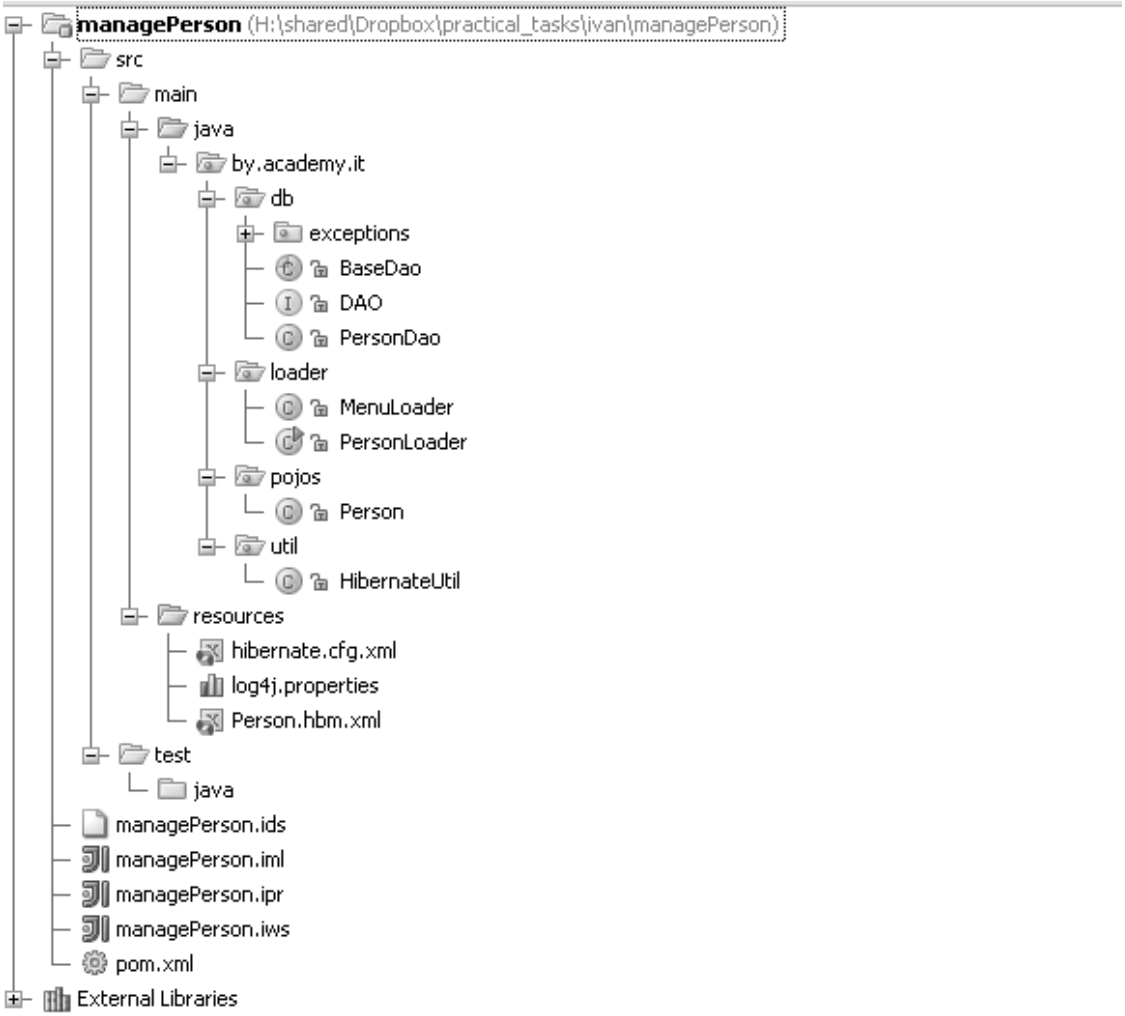

Рисунок 1.3.3 Структура проекта

Как вы можете видеть на рисунке 1.3.3, перед вами стандартный проект maven.

Исходя из содержимого pom-файла, можно сделать вывод, что данный проект зависит от следующих библиотек:

- log4j – библиотека, предназначенная для логирования;
- slf4j-log4j12 – специализированная библиотека, предназначенная для улучшения процесса логирования; работает вместе с log4j;
- hibernate-core – ядро Hibernate для Java, собственный API и метаданные отражение в формате XML;
- hibernate-entitymanager - реализация Java Persistence API для Java SE и Java EE
- mysql-connector-java драйвера.

Давайте теперь рассмотрим файл-mapping между Person и T_PERSON.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="by.academy.it.pojos" auto-import="false" catalog="my_db">
    <class name="Person" table="PERSON">
        <id name="id" type="int" column="F_ID">
            <generator class="increment"/>
        </id>

        <property name="name" column="F_NAME" type="string"/>
        <property name="surname" column="F_SURNAME" type="string"/>
        <property name="age" column="F_AGE" type="int"/>
    </class>
</hibernate-mapping>
```

Рисунок 1.3.4 Person.hbm.xml

Для генерации id применяется вызов oracle sequence. Также задается соответствие между полями класса Person и колонками таблицы T_PERSON.

После того, как мы создали mapping, пришло время создать конфигурационный файл, в котором указана вся информация о базе данных, с которой предстоит работать (адрес для соединения с СУБД, параметры отображения sql и т.д.)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
        <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/</property>
        <property name="hibernate.connection.username">root</property>
        <property name="hibernate.connection.password">yuli</property>
        <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
        <property name="hibernate.connection.pool_size">10</property>
        <property name="show_sql">true</property>
        <property name="hibernate.hbm2ddl.auto">validate</property>
        <mapping resource="Person.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

Рисунок 1.3.5 hibernate.cfg.xml

Основными параметрами в этом файле являются:

- hibernate.connection.driver_class – класс драйвера, с помощью которого приложение соединяется с базой данных;
- hibernate.connection.url – адрес для соединения с базой данных;
- hibernate.connection.username – имя пользователя для соединения с СУБД;
- hibernate.connection.password – пароль для соединения с СУБД;
- hibernate.default_schema - схема СУБД по умолчанию;

- hibernate.dialect – специфическая реализация языка по отношению к конкретной СУБД (например Oracle 11, Oracle 10 и т.д.);
- hibernate.connection.pool_size – максимальное число соединений в пуле соединений;
- show_sql – включение отображение SQL-запросов в логах приложения;
- hibernate.hbm2ddl.auto – автоматически обновляет (или пересоздает) схему в соответствии с mapping-файлом; существуют следующие значения – validate, update, create, create-drop

Из настроек осталось лишь настроить логгер, для отображения сообщений о состоянии приложения. Для этого необходимо добавить log4j.properties.

```
log4j.rootLogger=INFO, CONSOLE
#log4j.rootLogger=INFO, FILE
log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
log4j.appender.CONSOLE.Target=System.err
log4j.appender.CONSOLE.layout=org.apache.log4j.PatternLayout
log4j.appender.CONSOLE.layout.conversionPattern=%d{yyyy-MM-dd HH:mm:ss,SSS} %-5p - %m%n
log4j.appender.FILE=org.apache.log4j.RollingFileAppender
log4j.appender.FILE.File=log4j.log
log4j.appender.FILE.MaxFileSize=512KB
log4j.appender.FILE.MaxBackupIndex=3
log4j.appender.FILE.layout=org.apache.log4j.PatternLayout
log4j.appender.FILE.layout.conversionPattern=%d{yyyy-MM-dd HH:mm:ss,SSS} %-5p - %m%n
```

Рисунок 1.3.6 log4j.properties

Как вы можете увидеть, мы не отображаем на консоли сообщения в LOG.DEBUG режиме, потому что данный уровень содержит много несущественной информации, которая не ускоряет процесс поиска причин ошибок.

ТЕМА 2. РАБОТА С ОБЪЕКТАМИ В HIBERNATE

2.1 АРХИТЕКТУРА HIBERNATE.

Разобравшись с конфигурационными файлами, пора переходить к написанию кода. Для этого начнем с определения SessionFactory.

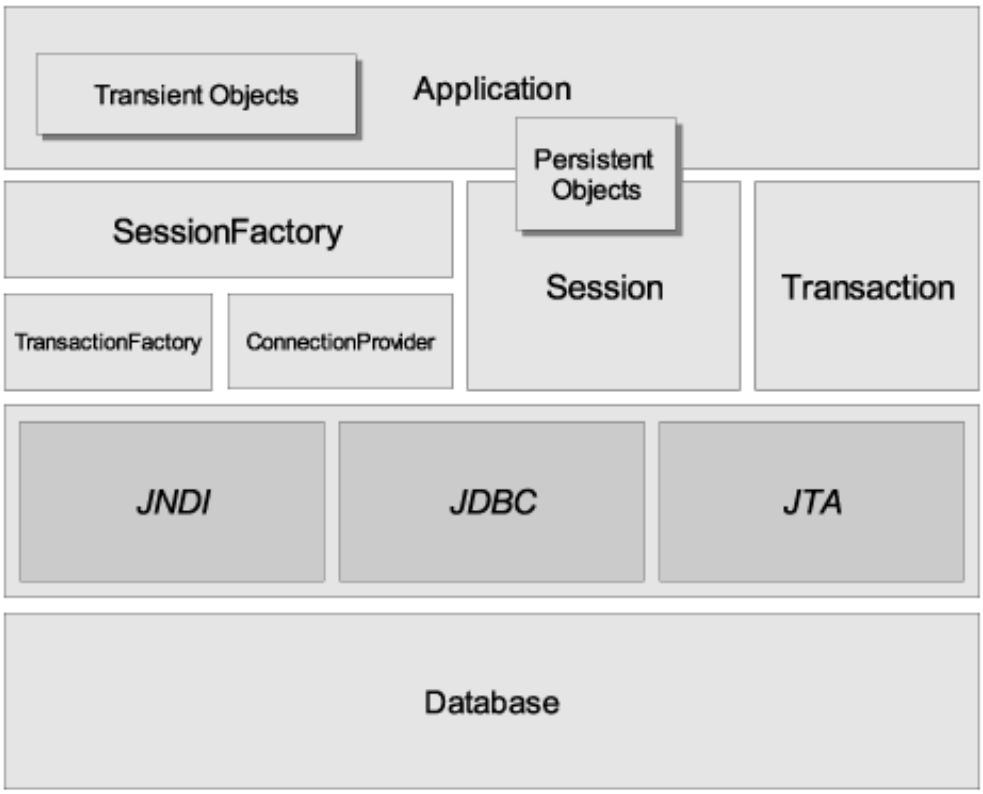

Рисунок 1.4.1 Внутренняя архитектура Hibernate

- SessionFactory (net.sf.hibernate.SessionFactory)

Потокобезопасный, неизменяемый (threadsafe, immutable) кэш откомпилированных мэппингов для одной базы данных. Фабрика для создания объектов класса Session (Hibernate сессий). SessionFactory является клиентом для ConnectionProvider. SessionFactory может поддерживать опциональный кэш для данных, которые используются несколькими транзакциями, этот кэш называют кэш второго уровня (second-level cache) или кэш уровня JVM (JVM-level cache). Данный кэш можно настроить таким образом чтобы он работал как для одного процесса (process-level) так и для нескольких процессов в кластере (cluster-level).

- Session (net.sf.hibernate.Session)

Однопоточный, короткоживущий объект, являющийся посредником между приложением и хранилищем долгоживущих объектов. Обертка вокруг JDBC-соединения. Фабрика для объектов класса Transaction. Содержит обязательный кэш первого уровня (first-level cache) для долгоживущих объектов, используется для навигации по объектному графу или для поиска объектов по идентификатору.

- Persistent Objects and Collections

Долгоживущие Объекты и Коллекции. Однопоточные, короткоживущие объекты содержащие сохраняемое состояние и бизнес функции. Это могут быть обычные JavaBean/POJO объекты, одна их отличительная особенность, это то, что они ассоциированы с одной сессией (Session). Как только их сессия закрыта, эти объекты становятся отсоединенными (detached) и свободными для использования на любом слое/уровне (layer) приложения, например непосредственно как объекты передачи данных (data transfer objects) на уровень представления и с него.

- Transient Objects and Collections

Временные Объекты и Коллекции. Экземпляры долгоживущих (persistent) классов, которые в данный момент не ассоциированы с сессией (Session). Это могут быть объекты инстанциированые приложением и в данный момент еще не переведенные в долгоживущее (persistent) состояние или они могли быть инстанциированы закрытой сессией

- Transaction (net.sf.hibernate.Transaction)

Транзакция. Опциональный однопоточный, короткоживущий объект, используется приложением для указания атомарной единицы выполняемой работы (atomic unit of work). Абстрагирует приложение от нижележащих JDBC, JTA или CORBA транзакций. В некоторых случаях одна сессия (Session) может породить несколько транзакций.

- ConnectionProvider (net.sf.hibernate.connection.ConnectionProvider)

Поставщик соединений. Опциональная фабрика и пул для JDBC-соединений. Абстрагирует приложение от нижележащих объектов Datasource или DriverManager. Это внутренний объект Hibernate, он недоступен для приложения, но может быть расширен/реализован разработчиком.

- TransactionFactory (net.sf.hibernate.TransactionFactory)

Фабрика транзакций. Опциональная фабрика для экземпляров класса Transaction. Это внутренний объект Hibernate, он недоступен для приложения, но может быть расширен/реализован разработчиком.

2.2 Состояние объектов в жизненном цикле Hibernate.

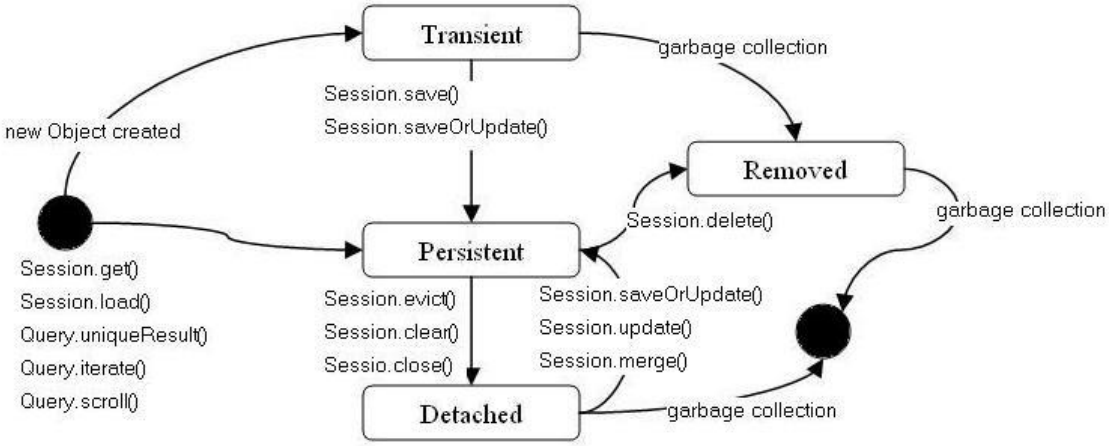

Рисунок 2.2.1 Состояние объектов в жизненном цикле Hibernate.

Сначала создаются Transient объекты (объекты, хранящиеся в памяти, и не ассоциированные с сессией). После того как объекты попадают в поле ответственности сессии (при сохранении или обновлении), то они становятся Persistent, т.е. теми объектами, которые имеют отображение на базу данных и попадают в поле ответственности сессии (могут быть сохранены или обновлены). Если же сессия была закрыта, то объекты становятся Detached – объекты, для которых существует отображение в базе данных, но сессия была закрыта. Detached объекты могут быть ассоциированы с другой сессией во время вызова одного из следующих методов load, refresh, merge, update или save.

2.3 Операции над данными в Hibernate.

Для манипуляции над данными необходимо создать объекты классов SessionFactory и Session. Для создания SessionFactory применяется методы класса:

- org.hibernate.cfg.Configuration;
- org.hibernate.boot.registry.StandardServiceRegistryBuilder.

```
private HibernateUtil() {
    try {
        Configuration configuration = new Configuration().configure();
        StandardServiceRegistryBuilder builder = new StandardServiceRegistryBuilder()
                .applySettings(configuration.getProperties());
        sessionFactory = configuration.buildSessionFactory(builder.build());
    } catch (Throwable e) {
        log.error("Initial SessionFactory creation failed. " + e);
        throw new ExceptionInInitializerError(e);
    }
}
```

Рисунок 2.3.1 Создание SessionFactory

Еще одним важным методом является создание net.sf.hibernate.Session.

```
public Session getSession() {
    Session session = (Session) sessions.get();
    if (session == null) {
        session = sessionFactory.openSession();
        sessions.set(session);
    }
    return session;
}
```

Рисунок 2.3.2 Создание Session

Имея в наличии session, мы можем легко начинать сохранять, удалять и доставать из таблиц данные. Для этого нужно использовать методы класса net.sf.hibernate.Session.

```
public void delete(Integer id) throws DaoException {
    try {
        Session session = util.getSession();
        transaction = session.beginTransaction();
        T t = (T) session.get(getPersistentClass(), id);
        session.delete(t);
        transaction.commit();
    } catch (HibernateException e) {
        log.error("Error was thrown in DAO:" + e);
        transaction.rollback();
        throw new DaoException(e);
    }

}
```

Рисунок 2.3.3 Удаление данных

Для сохранения данных легко подойдет следующий метод.

```
public T saveOrUpdate(T t) throws DaoException {
    try {
        Session session = util.getSession();
        transaction = session.beginTransaction();
        session.saveOrUpdate(t);
        session.update(t);
        transaction.commit();
    } catch (HibernateException e) {
        log.error("Error was thrown in DAO:" + e);
        transaction.rollback();
        throw new DaoException(e);
    }
    return t;
}
```

Рисунок 2.3.4 Сохранение данных

Для нахождения данных подойдет вот такой подход.

```
public T find(Integer id) throws DaoException {
    T t = null;
    try {
        Session session = util.getSession();
        transaction = session.beginTransaction();
        t = (T) session.get(getPersistentClass(), id);
        transaction.commit();
    } catch (HibernateException e) {
        log.error("Error was thrown in DAO:" + e);
        transaction.rollback();
        throw new DaoException(e);
    }
    return t;
}
```

Рисунок 2.3.5 Нахождение данных.

2.4 Сохранение объектов.

Объекты классов, которые имеют отображение, не сохраняют состояние автоматически в базу данных. Пока они не ассоциированы с определенной сессией и не сохранены с помощью её, они являются Transient.

Существуют следующие методы класса net.sf.hibernate.Session:

- public Serializable save(Object object) throws HibernateException - сохраняет transient объект , используя сгенерированный id, возвращает сгенерированный id;

- public void save(Object object, Serializable id) throws HibernateException - сохраняет transient объект, используя параметр id;
- public void saveOrUpdate(Object object) throws HibernateException - создает объект, если он не существует в базе данных, иначе – обновляет его.

2.5 Загрузка Объектов.

Сессия предоставляет несколько load() методов для загрузки объектов из вашей базы данных. Каждый load() метод требует в качестве параметра первичный ключ как идентификатор, класс которого должен реализовывать интерфейс Serializable. В результате вы должны привести возвращаемый Object тип к типу вашего класса.

- public Object load(Class theClass, Serializable id) throws HibernateException;
- public void load(Object object, Serializable id) throws HibernateException;

Также существует более безопасный способ загрузки объекта из базы данных, при котором HibernateException не генерируется, если объекта нету в базе данных.

- public Object get(Class class, Serializable id) throws HibernateException;

В случае если объект не найден в базе данных, метод вернет null.

2.6 Обновление объектов

Hibernate автоматически управляет всеми изменениями в объектах, ассоциированных с сессией. С точки зрения разработчика вам не нужно будет делать ничего, кроме как вызывать commit всех изменений в базу данных. Бывает, что состояние объектов в сессии не всегда совпадают с состоянием в базе. Это может быть например, если вы сохранили объекты в базе, а затем начали менять или ассоциировать новые Transient объекты с сессией, предварительно ее не очистив. Такое состояние сессии называется *грязным*. Если вы используете транзакции, то hibernate заботится об этом автоматически, т.е. перед commit, он очищает сессию.

- public void flush() throws HibernateException;

Данный метод синхронизирует состояние сессии с состоянием базы данных.

Вы также можете проверить грязная ли сессия, при помощи следующего метода:

- public boolean isDirty() throws HibernateException;

Этот метод возвращает true, если сессия – грязная, а false – в противном случае.

Очистка Hibernate сессии происходит только в следующих случаях:

- когда Transaction фиксируется(коммит);

- иногда перед выполнением запроса;
- когда приложение вызывает Session.flush() явно;

Очистка состояния сессии в БД в конце транзакции БД необходимо для того, чтобы сделать изменения долговечными. Hibernate не очищает сессию перед каждым запросом. Однако, если есть изменения хранимые в памяти, которые могут повлиять на результат запроса, то Hibernate, по умолчанию, синхронизирует их в первую очередь.

Вы можете контролировать это поведение явно, устанавливая Hibernate FlushMode через вызов Session.setFlushMode(). Режимами очистки сессии являются:

- FlushMode.AUTO – по умолчанию. Устанавливает поведение, описанное только что.
- FlushMode.COMMIT – указывает, что сессия не будет очищена перед выполнением запроса (она будет очищена только в конце транзакции БД). Имейте в виду, что этот параметр может затронуть устаревшие данные: изменения, внесенные в объекты только в памяти, могут вступать в противоречие с результатами запроса.
- FlushMode.NEVER – позволяет вам самим указать, что только явные вызовы flush() приведут к синхронизации состояния сессии с БД.
- FlushMode.ALWAYS – указывает, что сессия не будет очищаться всегда.

Hibernate предоставляет механизм синхронизации состояния объекта с состоянием базы данных. Существуют несколько сценариев, когда это понадобится:

- Сценарий, когда приложение выполняет SQL напрямую с базой данных, и после этого данные в базе и в объектах будут отличаться.
- Сценарий, когда ваша база данных использует триггеры для заполнения свойств объекта в базе данных, и после этого данные в базе и в объектах будут отличаться.

Для выше упомянутых целей в Session существует метод:

- public void refresh(Object object) throws HibernateException

Этот метод обновляет состояние объекта из базы данных.

2.7 Удаление объектов.

Для удаления объектов из базы данных в Hibernate существует следующий метод:

- public void delete(Object object) throws HibernateException;

Этот метод позволяет удалить объект без связей с другими объектами. Много объектов связаны с другими объектами, поэтому удаления могут быть каскадными, т.е. удаление одного объекта влечет к удалению связанных с ним других объектов. Задание опции cascade в отображениее, позволяет добиться эффекта каскадного удаления.

Существуют несколько вариантов cascade:

- *cascade="none"* - значение по умолчанию. Hibernate будет игнорировать ассоциации, поэтому управлять зависимостями придется самостоятельно.
- *cascade="save-update"* говорит Hibernate'у, что управлять зависимостями необходимо при комите транзакции в которой делается save() или update() объекта. Суть управления заключается в том, что новые объекты, с которыми есть ассоциации у нашего, будут сохранены до него. Это позволяет обойти constraint-violations.
- *cascade="delete"* говорит Hibernate'у, что надо управлять зависимостями при удалении объекта.
- *cascade="all"* обозначает выполнение каскадных операций при save-update и delete.
- *cascade="all-delete-orphan"* обозначает то же самое, что и cascade="all", но к тому же Hibernate удаляет любые связанные сущности, удаленные из ассоциации (например, из коллекции).
- *cascade="delete-orphan"* обозначает, что Hibernate будет удалять любые сущности, которые были удалены из ассоциации.

ТЕМА 3. ОСНОВЫ ОТОБРАЖЕНИЯ ОБЪЕКТНО-РЕЛЯЦИОННОГО МОДЕЛИ.

3.1 Объявление отображения.

Объектно-реляционный отображение описывается в виде XML документа.

```
<hibernate-mapping
        schema="schemaName"                                          1
        catalog="catalogName"                                        2
        default-cascade="cascade_style"                              3
        default-access="field|property|ClassName"                    4
        default-lazy="true|false"                                    5
        auto-import="true|false"                                     6
        package="package.name"                                       7
/>
```

Рисунок 3.1.1 Hibernate-mapping.

1) schema (необязательный): Наименование схемы БД.
2) catalog (необязательный): Наименование имени каталога БД.
3) default-cascade (необязательный - по умолчанию none): Правила каскадов по умолчанию.
4) default-lazy (необязательный – по умолчанию true): Правила lazy загрузки для классов и коллекций.
5) auto-import (необязательный – по умолчанию true): Определяет возможность использования неполных имен классов (описанных в данном документе) в языке запросов.
6) package (необязательный): Префикс пакета для определения полного наименования классов.

Если Hibernate обнаружит два класса в разных пакетах соответсвующих указаному имени, при разборе файла отображения будет сгенерировано исключение.

```
<class
        name="ClassName"                                                    (1)
        table="tableName"                                                   (2)
        discriminator-value="discriminator_value"                           (3)
        mutable="true|false"                                                (4)
        schema="owner"                                                      (5)
        catalog="catalog"                                                   (6)
        proxy="ProxyInterface"                                              (7)
        dynamic-update="true|false"                                         (8)
        dynamic-insert="true|false"                                         (9)
        select-before-update="true|false"                                   (10)
        polymorphism="implicit|explicit"                                    (11)
        where="arbitrary sql where condition"                               (12)
        persister="PersisterClass"                                          (13)
        batch-size="N"                                                      (14)
        optimistic-lock="none|version|dirty|all"                            (15)
        lazy="true|false"                                                   (16)
        entity-name="EntityName"                                            (17)
        check="arbitrary sql check condition"                               (18)
        rowid="rowid"                                                       (19)
        subselect="SQL expression"                                          (20)
        abstract="true|false"                                               (21)
        node="element-name"
```

Рисунок 3.1.2 Class-mapping.

1) *name:* Полное наименование Java класса персистентного класса или интерфейса.
2) *table:* Наименование таблицы БД.
3) *discriminator-value* (необязательный - по умолчанию имя класса): Значение для определения индивидуальных подклассов. Используется при полиморфизме. Допустимые значения включают null и not null
4) *mutable* (необязательный, по умолчанию true): Используется для определения (не)изменчивости класса (mutable/not mutable)
5) *schema* (необязательный): Переопределяет наименование схемы которая была специфицирована в корневом элементе <hibernate-mapping>.
6) *proxy* (необязательный): Определяет интерфейс используемый для ленивой инициализации прокси-класса. В качестве прокси-класса можно использовать тот же (персистетный) класс.

7) *dynamic-update* (необязательно, по умолчанию false): Если параметр установлен в true Hibernate будет генерировать UPDATE запрос в процессе выполнения, при этом запрос будет сгенерирован только на те столбцы, которые были изменены.

8) *dynamic-insert* (необязательный, по умолчанию false): Если параметр установлен в true Hibernate будет генерировать INSERT запрос в процессе выполнения, при этом запрос будет сгенерирован только на те столбцы, значение которых не null.

9) *select-before-update* (необязательный, по умолчанию false): Обязывает Hibernate перед выполнением SQL UPDATE всегда проверять действительно ли изменился объект. В определенных случаях (в действительности, только когда трансиентный объект ассоциируется с новой сессией с использованием update()), это означает что Hibernate генерирует SQL SELECT чтобы убедиться в том, что UPDATE действительно необходим.

10) *polymorphism* (необязательный, по умолчанию implicit): Определяет использования явного (explicit) или неявного (implicit) полиморфизма.

11) *where* (необязательный) определяет произвольное условие WHERE SQL запроса, который используется для получения обьектов определяемого класса.

12) *persister* (необязательный): Используется для указания реализации ClassPersister.

13) *batch-size* (необязательный, по умолчанию 1) определяет "размер пакета (batch size)" для выборки экземпляров определяемого класса по идентификатору.

14) *optimistic-lock* (необязательный, по умолчанию version): Устанавливает стратегию оптимистического блокирования (optimistic locking)

15) *lazy* (необязательный): Установка атрибута lazy="true" является эквивалентом установки интерфеса прокси на себя (смотри атрибут proxy).

Типичное свойство класса в Hinernate обладает тем же именем, что и оно же в JavaBean. Она также включает имя колонки и имя Hibernate типа.

```
<property name="description" column="DESCRIPTION" type="string"/>
```

Hibernate использует reflection для определения Java типа свойства.

```
<property name="description" column="DESCRIPTION"/>
```

Вы можете не указывать имя колонки, если ее имя совпадает с именем свойства (игнорируя прописные или строчные буквы).

```
<property name="description"/>
```

Вы также можете использовать <column> тэг вместо column-аттрибута.

```
<property name="description" column="DESCRIPTION" type="string"/>
<property name="description" type="string">
 <column name="DESCRIPTION"/>
</property>
```

Очень часто необходимо показать в какие поля объекта нельзя вносить null-значения. Для этих целей предназначен not-null аттрибут.

```
<property name="initialPrice" column="INITIAL_PRICE" not-null="true"/>
```

Бывает необходимо задать знание поля как функции от других полей таблицы или объекта. В этом случае «извлекаемые свойства» помогут решить проблему.

```
<property name="totalIncludingTax" formula="TOTAL + TAX_RATE * TOTAL" type="big_decimal"/>
```

В Hibernate существует возможность задавать значение для поля класса как через getter/setter методы, так и напрямую в поле объекта. За установку конфигурации отвечает атрибут access.

```
<property name="name" column="F_NAME" type="string" access="property"/>

<property name="customName" formula="(F_ID*3)" type="int" access="field"/>
```

Рисунок 3.1.3 Атрибут access.

3.2 Контроль над редактированием данных. Именование таблиц.

Для некоторых полей объекта необходимо запретить внесение изменений в базу данных, т.е. данные из базы данных могут читаться в поля объекта, но запись в базу данных из полей объекта будет запрещена.

Для того чтобы добиться такого эффекта мы можем применить следующий метод:

```
<property name="name" column="F_NAME" type="string" access="property" insert="false"/>

<property name="customName" formula="(F_ID*3)" type="int" access="field" update="false"/>
```

Рисунок 3.2.1 Атрибуты insert и update.

```
Company choose - cool9
Hibernate: select ivan.T_COMPANY_SEQ.nextval from dual
Hibernate: insert into ivan.T_COMPANY (F_ID) values (?)
2012-12-11 02:18:26,186 WARN  - SQL Error: 1400, SQLState: 23000
2012-12-11 02:18:26,186 ERROR - ORA-01400: cannot insert NULL into ("IVAN"."T_COMPANY"."F_NAME")
```

Рисунок 3.2.2 Результат запуска программы.

Такого же эффекта можно добиться, если установить immutable="false" в конфигурации класса.

```
<class name="Company" table="T_COMPANY" mutable="false"
```

Рисунок 3.2.3 Атрибут mutable.

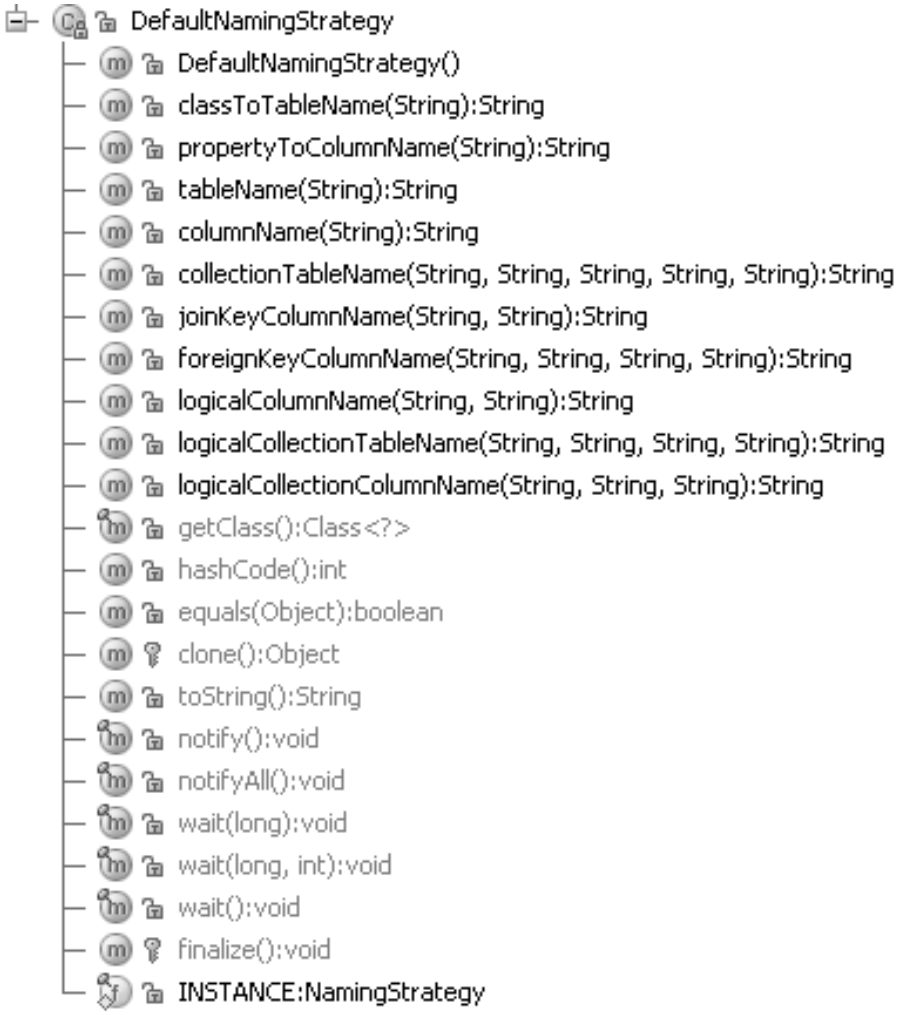

Рисунок 3.2.4 Методы DefaultNamingStrategy

Подключение происходит при создании SessionFactory.

```
Configuration conf = new Configuration();
conf.setNamingStrategy(new CustomNamingStrategy());
sessionFactory = conf.configure().buildSessionFactory();
```

Рисунок 3.2.5 Подключение DefaultNamingStrategy

Для того, что бы задать вашу конфигурацию вы должны либо реализовать NamingStrategy, либо унаследоваться от DefaultNamingStrategy.

3.3 Понимание Идентичности

Object identity— Объекты одинаковы, если они ссылаются на одну и туже область памяти в JVM. Это может быть проверено с помощью ==.

Object equality— Объекты эквивалентны, если они имеют одинаковые значения полей, как определено в equals(Object o) методе. Классы, которые не переопределяют этот метод из java.lang.Object, проверяют свою идентичность.

Database identity— Объекты, сохраняющиеся в СУБД, идентифицируемые одинаковыми первичными ключами, представляющие одну и туже строку данных.

Для получения идентификатора Persistent объекта применяется следующий метод:

```
Serializable getIdentifier(Object object) throws HibernateException
```

При выборе первичного ключа необходимо учитывать следующие моменты:

1. Значение не должно быть Null;
2. Каждая строка данных должна уникальным образом идентифицироваться;
3. Значение, идентифицирующее существующую строку, никогда не должно меняться.

```
<id
        name="propertyName"                          (1)
        type="typename"                              (2)
        column="column_name"                         (3)
        unsaved-value="any|none|null|id_value"       (4)
        access="field|property|ClassName">           (5)
        <generator class="generatorClass"/>
</id>
```

(1) name (необязательный): Наименование свойства идентификатора.

(2) type (необязательный): Имя определяющее Hibernate-тип свойства.

(3) column (необязательно - по умолчанию имя свойства): Название колонки основного ключа.

(4) unsaved-value (необязательно - по умолчанию null): Значени свойства идентификатора, которое обзначает, что экземпляр новый (в терминах персистентного хранилища).

(5) access (необязательный - по умолчанию property): Эту стратегию Hibernate будет использовать для доступа к данному свойству объекта.

Элемент <generator>'a определяет Java класс используемый для генерации уникальных идентификаторов экземпляров песистентных классов.

Все генераторы реализуют интерфейс net.sf.hibernate.id.IdentifierGenerator. Это очень простой интерфейс; многие приложения могут использовать свою специальную реализацию генератора. Несмотря на это, Hibernate включает в себя множество встроенных генераторов. Ниже идут краткие наименования (ярлыки) для встроенных генераторов:

- *increment* - генерирует идентификаторы типа long, short или int, уникальные только когда другие процессы не добавляют данные в ту же таблицу. Не использовать в кластере;
- *identity* - поддерживает identity колонки в in DB2, MySQL, MS SQL Server, Sybase и HypersonicSQL. Тип возвращаемого идентификатора long, short или int;
- *sequence* - использует последовательность (sequence) в DB2, PostgreSQL, Oracle, SAP DB, McKoi или generator в Interbase. Тип возвращаемого идентификатора long, short или int;
- *hilo* - использует hi/lo алгоритм для эффективной генерации идентификаторов которые имеют тип long, short или int, требуют наименования таблицы и столбца (по умолчанию hibernate_unique_key и next_hi соответсвенно), как источник значений hi. Алгоритм hi/lo генерирует идентификаторы которые уникальный только для отдельный баз данных. Не используйте этот генератор для соединений через JTA или пользовательских соединений;
- *seqhilo* - использует алгоритм hi/lo для генерации идентификаторов типа long, short или int, с использованием последовательности (sequence) базы данных;
- *uuid.hex*- Использует 128-битный UUID алгоритм для генерации строковых идентификаторов, уникальных в пределах сети (изспользуется IP-адрес). UUID - строка длинной в 32 символа, содержащая шеснадцатеричное представление числа;
- *uuid.string* - использует тот же UUID алгоритм, однако строка при использовании этого генератора состоит из 16 (каких-то) ANSII символов. Не использовать с PostgreSQL;
- *native* - выбирает identity, sequence или hilo, в зависимости от возможностей используемой базы данных;
- *assigned* - предоставляет приложению возможность самостоятельно задать идентификатор объекта перед вызовом метода save();
- *foreign* - используется идентификатор другого, ассоциированного объекта. Обычно используется в коньюкции с <one-to-one> ассоциацией по первичному ключу.

3.4 Понятие компонента и сущности.

Компонент это содержащийся объект, который сохраняется как значение, а не как сущность. Термин компонент относится к объектно-ориентированному понятию композиция (не путать с архитектурным компонентом).

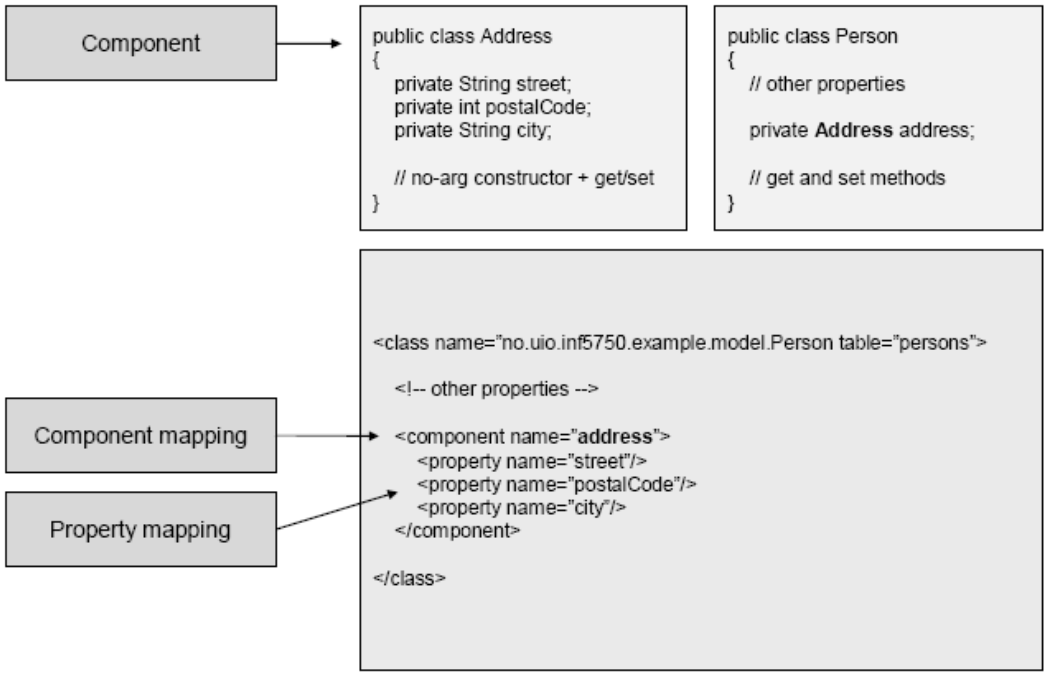

Рисунок 3.4.1 Сущность компонента

```xml
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="by.academy.it.pojos">
    <class name="Company" lazy="false">
        <id name="id" type="int">
            <generator class="sequence">
                <param name="sequence">T_COMPANY_SEQ</param>
            </generator>
        </id>
        <property name="name" type="string" access="property"/>
        <component name="homeAddress" class="Address">
            <property name="street" column="F_HOME_STREET"/>
            <property name="city" column="F_HOME_CITY"/>
            <property name="zipCode" column="F_HOME_ZIPCODE"/>
        </component>
        <component name="billingAddress" class="Address">
            <property name="street"/>
            <property name="city"/>
            <property name="zipCode"/>
        </component>
        <property name="customName" formula="(F_ID*3)" type="int" access="field"/>

    </class>
</hibernate-mapping>
```

Рисунок 3.4.2 Пример конфигурации сущности и компонента

ТЕМА 4. ОТОБРАЖЕНИЕ ИЕРАРХИИ КЛАССОВ И АССОЦИАЦИЙ.

4.1 ОТОБРАЖЕНИЕ НАСЛЕДНИКОВ.

Hibernate поддерживает три базовые стратегии отображения наследников:

- table per class hierarchy
- table per subclass
- table per concrete class (с некоторыми ограничениями)

Table per class hierarchy - это самый простой способ конфигурации, при котором вы просто создаете отдельный класс-отображение с именем таблицы для каждого класса. У этого подхода есть ряд недостатков:

1. Сложность использования полиморфных запросов
2. Использование различных колонок в разных таблицах, которые отражают одну семантику и структуру.

Исходя из выше перечисленного, можно рекомендовать использование этого подхода только на верхнем уровне вашей иерархии классов, где нет необходимости в полиморфизме.

Table per class hierarchy – это способ конфигурации, при котором отображение структуры наследования настраивается на одну таблицу.

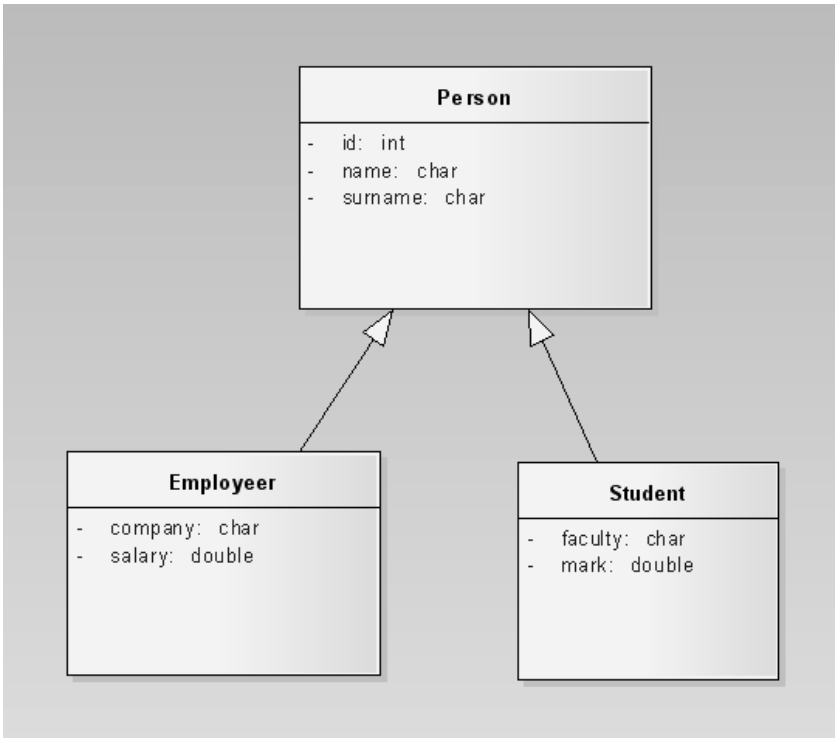

Рисунок 4.1.1 Пример конфигурации структуры наследования классов.

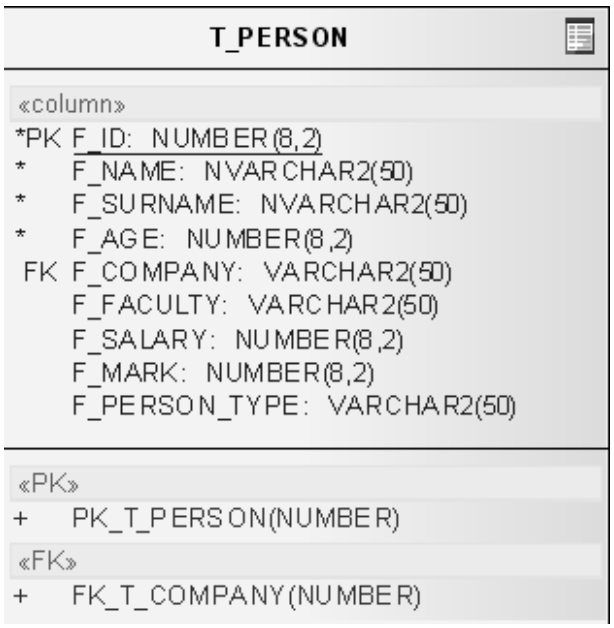

Рисунок 4.1.2 Таблица, на которую настраивается mapping структуры.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="by.academy.it.pojos" default-cascade="">
    <class name="Person" table="T_PERSON" discriminator-value="P">
        <id name="id" type="int" column="F_ID">
            <generator class="native">
                <param name="sequence">T_PERSON_SEQ</param>
            </generator>
        </id>
        <discriminator column="F_PERSON_TYPE" type="string"/>
        <property name="name" column="F_NAME" type="string"/>
        <property name="surname" column="F_SURNAME" type="string"/>
        <property name="age" column="F_AGE" type="int"/>
        <subclass name="Employeer" discriminator-value="E">
            <property name="company"/>
            <property name="salary"/>
        </subclass>
        <subclass name="Student" discriminator-value="S">
            <property name="faculty"/>
            <property name="mark"/>
        </subclass>
    </class>
</hibernate-mapping>
```

Рисунок 4.1.3 Mapping структуры на одну таблицу.

Table per subclass – это такой способ конфигурации, при котором отображение структуры наследования настраивается таким образом, чтобы каждый класс в цепочки наследования настраивался на отдельную таблицу. Причем связь между первичным ключами классов-наследников и первичным ключом класса-предка представляет собой структуру один к одному (one-to-one).

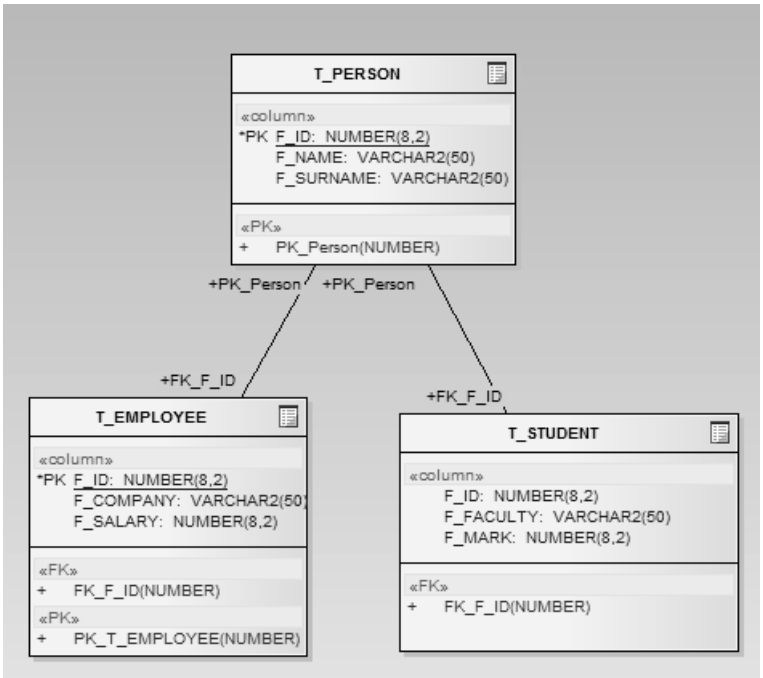

Рисунок 4.1.4 Таблицы для классов структуры.

```xml
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="by.academy.it.pojos">
    <class name="Person" table="T_PERSON">
        <id name="id" type="int" column="F_ID">
            <generator class="native">
                <param name="sequence">T_PERSON_SEQ</param>
            </generator>
        </id>
        <property name="name" column="F_NAME" type="string"/>
        <property name="surname" column="F_SURNAME" type="string"/>
        <property name="age" column="F_AGE" type="int"/>
        <joined-subclass name="Employee">
            <key column="F_ID"/>
            <property name="company"/>
            <property name="salary"/>
        </joined-subclass>
        <joined-subclass name="Student">
            <key column="F_ID"/>
            <property name="faculty"/>
            <property name="mark"/>
        </joined-subclass>
    </class>
</hibernate-mapping>
```

Рисунок 4.1.5 Mapping структуры на отдельные таблицы.

4.2 Отображение отношения один-к-одному.

Давайте рассмотрим One-to-One отображение.

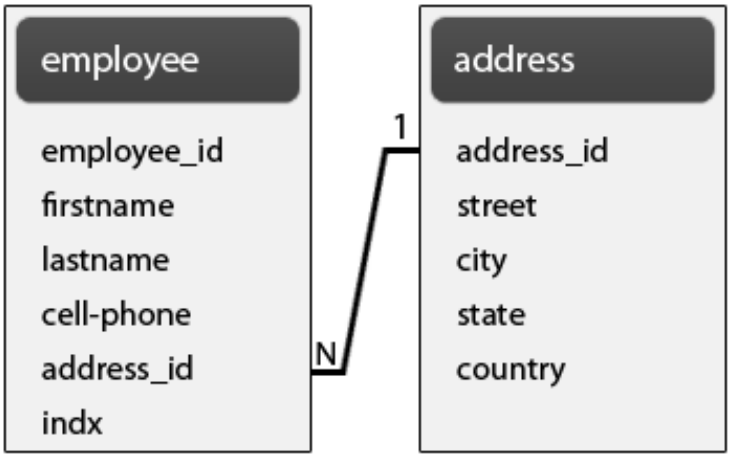

Fig:- one - to - many Mapping

Рисунок 4.2.1 Mapping структуры one-to-one

Данная ассоциация имеет следующие параметры: в таблице employee существует поле address_id, которое является внешним ключом на таблицу address. В таблице address есть поле address_id, которая является первичным ключом таблице address. Значение address_id в таблице address и address_id в таблице employee должны быть одинаковым. Это значит, что в системе существует одна сущность employee, связанная с одной сущностью address.

Давайте рассмотрим, как сконфигурировать one-to-one отображение в hibernate.

```
public class Employee implements Serializable {
    private static final long serialVersionUID = 4L;
    private Long employeeId;
    private String firstname;
    private String lastname;
    private String cellphone;
    private EmployeeDetail employeeDetail;
```

Рисунок 4.2.2 Класс Employee и поле EmployeeDetail, ассоциация между которыми one-to-one.

```
public class EmployeeDetail implements Serializable {
    private static final long serialVersionUID = 4L;

    private Long employeeId;
    private String street;
    private String city;
    private String state;
    private String country;

    private Employee employee;
```

Рисунок 4.2.3 Класс EmployeeDetail и поле Employee, ассоциация между которыми one-to-one.

На рисунках 4.2.2 и 4.2.3 представлены классы Employee и EmployeeDetail, связь между которыми является one-to-one.

Для того, что бы ассоциация заработало, необходимо настроить отображение.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="by.academy.it.pojos">
    <class name="Employee" lazy="false">
        <id name="employeeId">
            <generator class="native">
                <param name="sequence">T_PERSON_SEQ</param>
            </generator>
        </id>
        <one-to-one name="employeeDetail" class="EmployeeDetail"></one-to-one>
        <property name="firstname"/>
        <property name="lastname"/>
        <property name="cellphone"/>
    </class>
</hibernate-mapping>
```

Рисунок 4.2.4 Hibernate Отображение для класса Employee.

Для настройки one-to-one необходимо указать имя поля в классе Employee и указать класс EmployeeDetail. Также необходимо настроить sequence генератор для первичного ключа employeeId.

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

<hibernate-mapping package="by.academy.it.pojos">
    <class name="EmployeeDetail" lazy="false">
        <id name="employeeId">
            <generator class="foreign">
                <param name="property">employee</param>
            </generator>
        </id>
        <one-to-one name="employee" class="Employee"
                    constrained="true"></one-to-one>

        <property name="street"/>
        <property name="city"/>
        <property name="state"/>
        <property name="country"/>
    </class>
</hibernate-mapping>
```

Рисунок 4.2.5 Hibernate Отображение для класса EmployeeDetail.

Для настройки one-to-one отображения, необходимо установить имя поля employee и его класс Employee, а также установить параметр constrained=”true”, благодаря которым проверяется связь между полями employeeId в классах Employee и

EmployeeDetail. Также необходимо указать для первичного ключа настройку генератора, при которой значение для поля employeeId класса EmployeeDetail берется из поля employeeId из класса Employee.

4.3 Отображение отношения один-ко-многим, многие-к-одному.

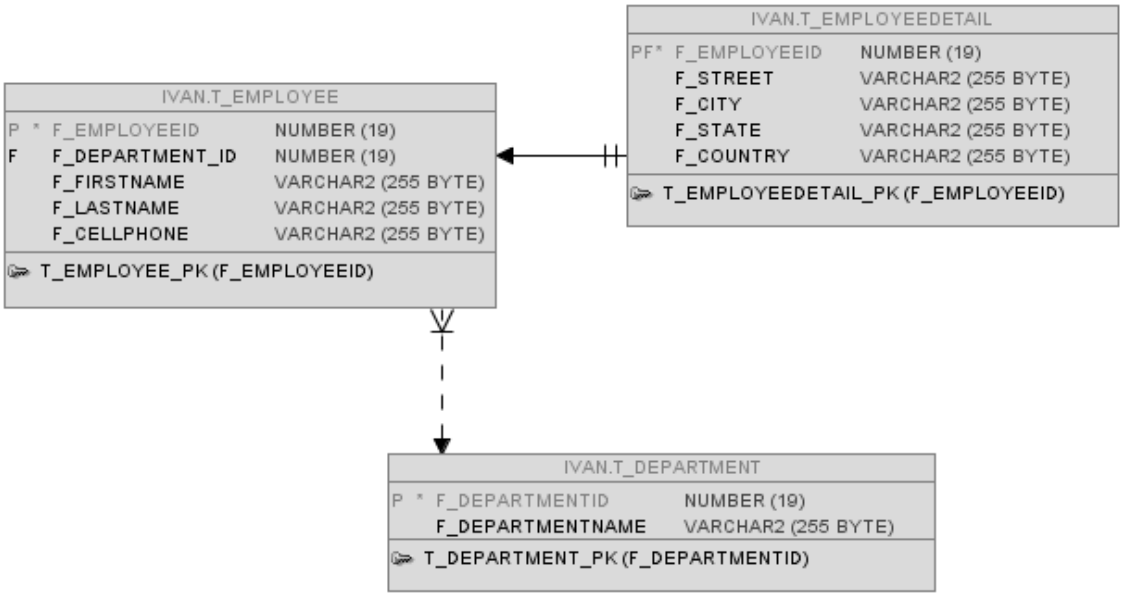

Рисунок 4.3.1 Описание связей между таблицами T_EMPLOYEE и T_DEPARTMENT

```
public class Department implements Serializable {
    private static final long serialVersionUID = 5L;
    private Long departmentId;
    private String departmentName;
    private Set<Employee> employees;
```

Рисунок 4.3.2 Класс Department и связь one-to-many с сущностями employees.

Сущность Department является собственником связи one-to-many. Таким образом, в одном Department работают множество Employee.

```
public class Employee implements Serializable {
    private static final long serialVersionUID = 4L;
    private Long employeeId;
    private String firstname;
    private String lastname;
    private String cellphone;
    private EmployeeDetail employeeDetail;
    private Department department;
```

Рисунок 4.3.3 Класс Employee и связь many-to-one с сущностью department.

Класс Employee связан с классом Department по связи may-to-one, что значит, что множество Employee работают в рамках одного Department.

Давайте рассмотрим hibernate отношения для этих классов.

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

<hibernate-mapping package="by.academy.it.pojos">

    <class name="Department">

        <id name="departmentId">
            <generator class="native">
                <param name="sequence">T_DEPARTMENT_SEQ</param>
            </generator>
        </id>

        <property name="departmentName"/>

        <set name="employees" table="employee"
             inverse="true" lazy="true" fetch="select">
            <key column="f_department_id" not-null="true"/>
            <one-to-many class="Employee"/>
        </set>

    </class>
</hibernate-mapping>
```

Рисунок 4.3.4 Hibernate отношение для класса Department.

Коллекция employees настроена таким образом, что благодаря параметру inverse=”true” указывает на хозяина связи, т.е. на department. Также мы можем видеть, что в связи указана колонка f_department_id, через которую идет связь между Employee и Department.

```xml
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="by.academy.it.pojos">
    <class name="Employee" lazy="false">
        <id name="employeeId">
            <generator class="native">
                <param name="sequence">T_PERSON_SEQ</param>
            </generator>
        </id>
        <one-to-one name="employeeDetail" class="EmployeeDetail"></one-to-one>
        <many-to-one name="department" class="Department" fetch="select" column="f_department_id"/>
        <property name="firstname"/>
        <property name="lastname"/>
        <property name="cellphone"/>
    </class>
</hibernate-mapping>
```

Рисунок 4.3.5 Hibernate отношение для класса Employee.

Глядя на отображение many-to-one, мы можем заметить, что указана колонка f_department_id, по которой будет происходить связь между Department и Employee, а также ввиду отсутствия параметра inverse, можно судить, что класс Employee не является собственником связи. Из этого следует, что для того, что бы обеспечить целостность данных, Hibernate должен сохранить сперва Department, а затем Employee.

4.4 Отображение отношения многие-ко-многим.

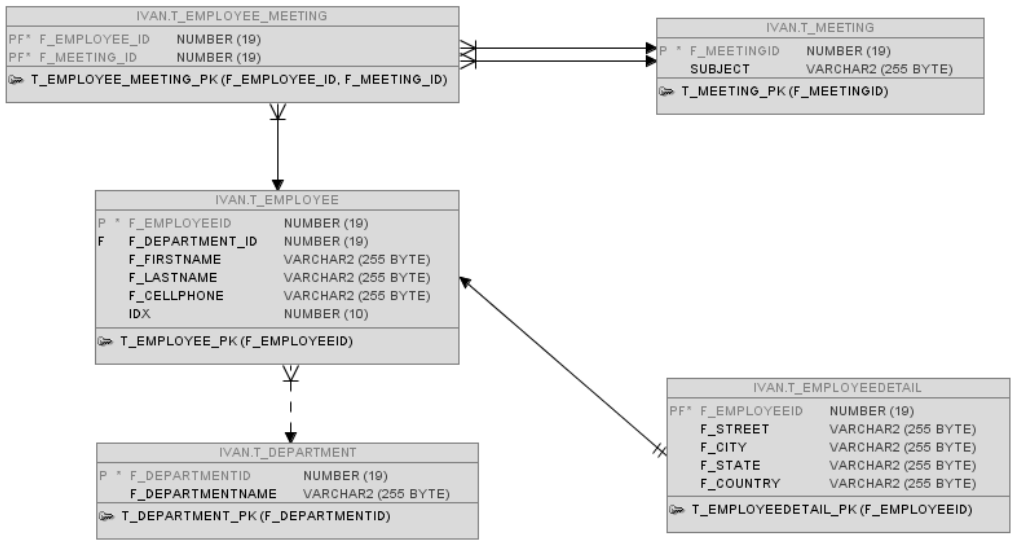

Рисунок 4.4.1 Описание связей между таблицами T_EMPLOYEE и T_MEETING

Как мы можем видеть на рисунке 4.4.1, сущности Employee и Meeting связаны через промежуточную таблицу T_EMPLOYEE_MEETING. Это означает, что связь многие ко многим строиться через промежуточную таблицу, которая связана с исходными таблицами с помощью внешних ключей.

Давайте рассмотрим сущности. Начнем с Employee.

```
public class Employee implements Serializable {
    private static final long serialVersionUID = 4L;
    private Long employeeId;
    private String firstname;
    private String lastname;
    private String cellphone;
    private EmployeeDetail employeeDetail;
    private Department department;
    private Set<Meeting> meetings = new HashSet<Meeting>();
```

Рисунок 4.4.2 Класс Employee и связь many-to-many с сущностями Meeting.

В классе Employee присутствует коллекция митингов, т.е. одному сотруднику может соответствовать множество встреч.

```
public class Department implements Serializable {
    private static final long serialVersionUID = 5L;
    private Long departmentId;
    private String departmentName;
    private List<Employee> employees;
```

Рисунок 4.4.3 Класс Employee и связь many-to-many с сущностями Meeting.

```
public class Meeting implements Serializable {
    private static final long serialVersionUID = 6L;
    private Long meetingId;
    private String subject;

    private Set<Employee> employees = new HashSet<>();
```

Рисунок 4.4.4 Класс Meeting и связь many-to-many с сущностями Employee.

В классе Meeting присутствует коллекция сотрудников, т.е. одной встрече соответствует множество сотрудников.

Давайте посмотрим на hibernate отображения.

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

<hibernate-mapping package="by.academy.it.pojos">

    <class name="Meeting">

        <id name="meetingId">
            <generator class="native">
                <param name="sequence">T_MEETING_SEQ</param>
            </generator>
        </id>

        <property name="subject" column="SUBJECT"/>

        <set name="employees" table="T_EMPLOYEE_MEETING"
             inverse="true" lazy="true" fetch="select">
            <key column="F_MEETING_ID"/>
            <many-to-many column="F_EMPLOYEE_ID" class="Employee"/>
        </set>

    </class>
</hibernate-mapping>
```

Рисунок 4.4.5 Hibernate отображение для класса Meeting.

Для настройки ассоциации нужно указать таблицу посредник T_EMPLOYEE_MEETING, указать, кто является владельцем связи. Нужно также указать внешнюю связь F_MEETING_ID и связь на другую сущность.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="by.academy.it.pojos">
    <class name="Employee" lazy="false">
        <id name="employeeId">
            <generator class="native">
                <param name="sequence">T_PERSON_SEQ</param>
            </generator>
        </id>
        <one-to-one name="employeeDetail" class="EmployeeDetail"></one-to-one>
        <many-to-one name="department" class="Department" fetch="select" column="f_department_id"/>
        <set name="meetings" table="T_EMPLOYEE_MEETING"
             inverse="false" lazy="true" fetch="select" cascade="all">
            <key column="F_EMPLOYEE_ID"/>
            <many-to-many column="F_MEETING_ID" class="Meeting"/>
        </set>
        <property name="firstname"/>
        <property name="lastname"/>
        <property name="cellphone"/>
    </class>
</hibernate-mapping>
```

Рисунок 4.4.6 Hibernate отображение для класса Employee.

Для класса Employee мы указываем также как и для Meeting, таблицу-посредник T_EMPLOYEE_MEETING. А также внешние ключи и связь на сущность Meeting. В двусторонней связи многие-ко-многим владельцем связи может быть любая сущность (либо Employee, либо Meeting).

ТЕМА 5. ОТОБРАЖЕНИЯ ЧЕРЕЗ HIBERNATE АННОТАЦИИ.

5.1 HIBERNATE АННОТАЦИИ.

Hibernate Annotations — отображение с помощью аннотаций. У аннотаций есть как свои плюсы, так и свои недостатки, по сравнению с отображением в виде xml-конфигурации.

Давайте рассмотрим отображение сущностей с помощью аннотаций на примере. Для примера будем использовать таблицу Employee.

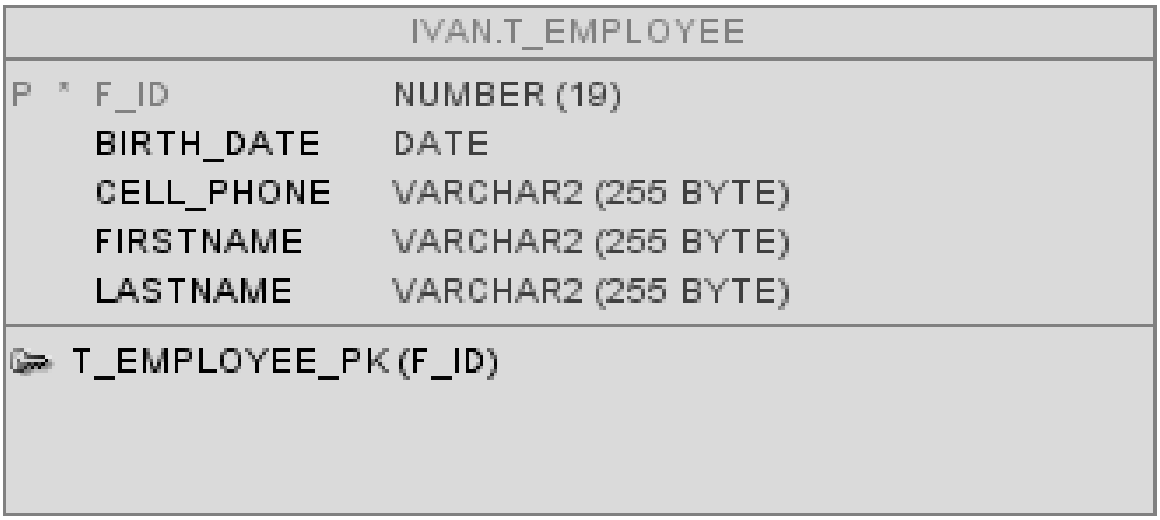

Рисунок 5.1.1 Таблица для класса Employee.

Зависимости для настройки hibernate-annotation выглядят следующим образом.

```
<dependencies>
    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.15</version>
        <exclusions>
            <exclusion>
                <groupId>com.sun.jdmk</groupId>
                <artifactId>jmxtools</artifactId>
            </exclusion>
            <exclusion>
                <groupId>com.sun.jmx</groupId>
                <artifactId>jmxri</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-log4j12</artifactId>
        <version>1.5.6</version>
    </dependency>
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>ejb3-persistence</artifactId>
        <version>1.0.1.GA</version>
    </dependency>
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-annotations</artifactId>
        <version>3.3.1.GA</version>
    </dependency>
    <dependency>
        <groupId>com.oracle</groupId>
        <artifactId>ojdbc6</artifactId>
        <version>11.2.0.1.0</version>
    </dependency>
</dependencies>
```

Рисунок 5.1.2 Зависимости для hibernate-annotation.

Как видно из рисунка 5.1.2, для настроек зависимостей необходимы библиотеки ejb3-persistence, hibernate-annotations (и неявные зависимости на hibernate-core) и jdbc driver (если приложение работает не в рамках сервера приложений с поддержкой jta).

Давайте взглянем на структуру проекта.

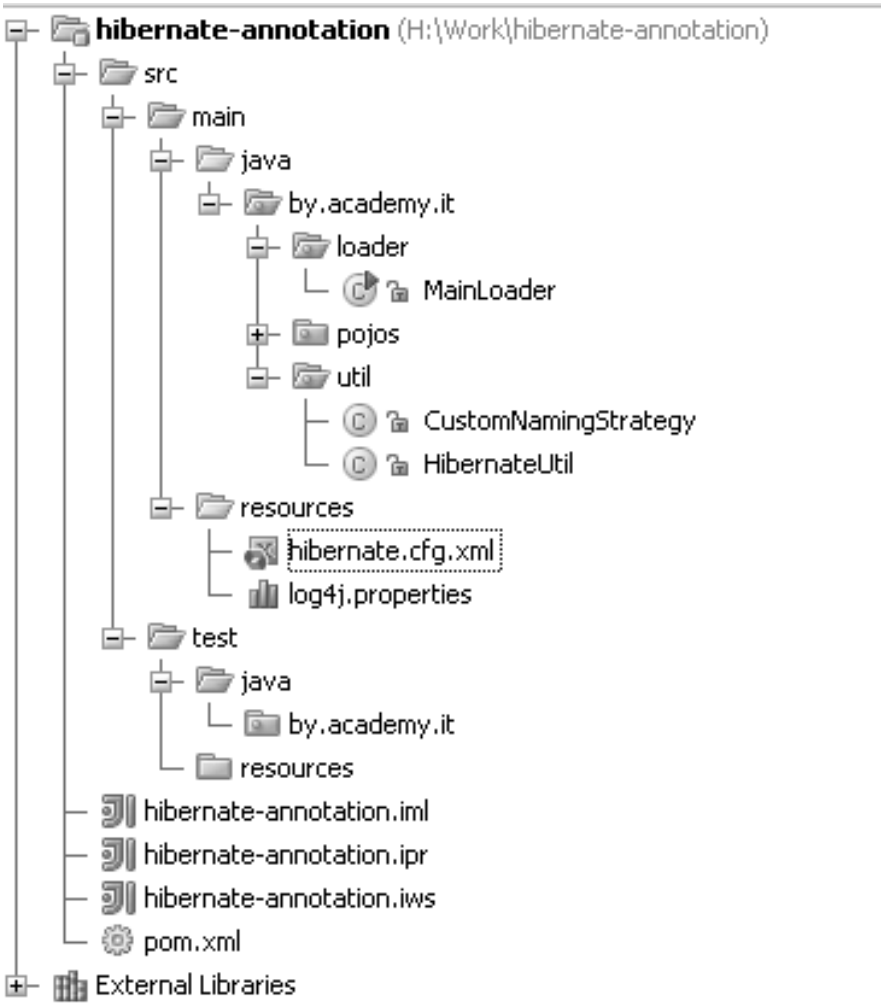

Рисунок 5.1.3 Структура проекта на hibernate-annotations.

Конфигурационный файл hibernate.cfg.xml выглядит следующим образом.

```xml
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
    <session-factory>
        <property name="hibernate.connection.driver_class">
            oracle.jdbc.driver.OracleDriver
        </property>
        <property name="hibernate.connection.url">
            jdbc:oracle:thin:@localhost:1521:xe
        </property>
        <property name="hibernate.connection.username">
            ivan
        </property>
        <property name="hibernate.connection.password">
            ivan
        </property>
        <property name="hibernate.default_schema">
            ivan
        </property>
        <property name="hibernate.dialect">
            org.hibernate.dialect.OracleDialect
        </property>
        <property name="hibernate.connection.pool_size">10</property>
        <property name="show_sql">true</property>
        <property name="hibernate.hbm2ddl.auto">create-drop</property>
        <!-- Mapping files -->
        <mapping class="by.academy.it.pojos.Employee"/>
    </session-factory>
</hibernate-configuration>
```

Рисунок 5.1.4 Конфигурационный файл hibernate.cfg.xml

Как вы можете заметить, по сравнению с xml-отображением, для аннотаций необходимо указывать на отображение класса, вместо отображения ресурса.

Давайте посмотрим на класс Employee.

```
@Entity
@SequenceGenerator(name = "PK", sequenceName = "t_employee_seq")
public class Employee implements Serializable {
    private static final long serialVersionUID = 4L;

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "PK")
    private Long id;

    @Column(name = "firstname")
    private String firstname;

    @Column(name = "lastname")
    private String lastname;

    @Column(name = "birth_date")
    private Date birthDate;

    @Column(name = "cell_phone")
    private String cellphone;

    public Employee() {
    }

    public Employee(String firstname, String lastname, Date birthDate, String cellphone) {
        this.firstname = firstname;
        this.lastname = lastname;
        this.birthDate = birthDate;
        this.cellphone = cellphone;
    }
```

Рисунок 5.1.5 Конфигурационный файл hibernate.cfg.xml

В начале класса мы встречаем аннотацию @Enitity - говорит о том, что этот объект будет обрабатываться hibernate. @SequenceGenerator указывает на конфигурацию генератора, который формирует значения для уникального идентификатора @Id. Аннотация @Column указывает на то что поле lastname связана с колонкой lastname таблицы T_EMPLOYEE.

Теперь давайте посмотрим на код вызова.

```
public class MainLoader {
    private static Logger log = Logger.getLogger(MainLoader.class);

    public static void main(String... args) throws Exception {
        Locale.setDefault(Locale.US);
        HibernateUtil util = HibernateUtil.getInstance();
        Session session = util.getSession();
        log.info("Read employee information");
        SimpleDateFormat sdfout = new SimpleDateFormat("yyyy.MM.dd");
        String date = "2012.12.20";
        Employee employee = new Employee("Ivan", "Spresov",
                new java.sql.Date(sdfout.parse(date).getTime()), "3456345345");
        log.info(employee);
        session.saveOrUpdate(employee);
        session.flush();
        log.info(employee);
        session.close();
    }
}
```

Рисунок 5.1.6 Код сохранения Employee в базу данных.

5.2. Давайте рассмотрим настройку ассоциаций один-к-одному с помощью hibernate аннотаций. Как и в предыдущих главах, рассмотрим конфигурацию на примере.

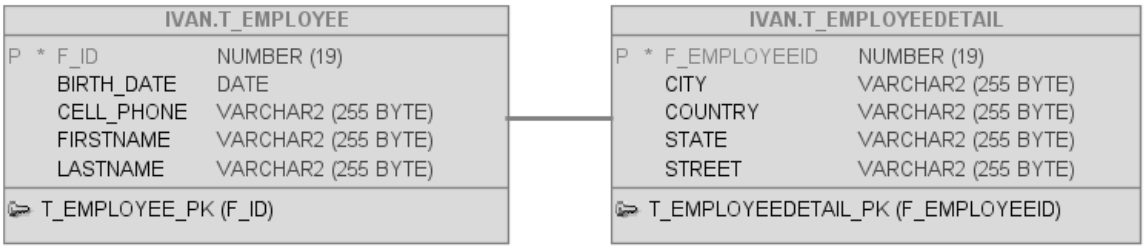

Рисунок 5.2.1 Ассоциация один-к-одному между таблиц T_EMPLOYEE и T_EMPLOYEEDETAIL

Давайте рассмотрим структуры классов.

```
@Entity
@SequenceGenerator(name = "PK", sequenceName = "t_employee_seq")
public class Employee implements Serializable {
    private static final long serialVersionUID = 4L;

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "PK")
    private Long id;

    @Column(name = "firstname")
    private String firstname;

    @Column(name = "lastname")
    private String lastname;

    @Column(name = "birth_date")
    private Date birthDate;

    @Column(name = "cell_phone")
    private String cellphone;

    @OneToOne(mappedBy="employee", cascade=CascadeType.ALL)
    private EmployeeDetail employeeDetail;
```

Рисунок 5.2.2 Класс Employee.

Для конфигурирования связи один-к-одному применяется аннотация @OneToOne. Связь один-к-одному может быть двунаправленной. В двунаправленных отношениях одна из сторон (и только одна) должна быть владельцем и нести ответственность за обновление связанных полей. В нашем случае владельцем выступает сущность Employee. Для того, чтобы объявить сторону, которая не несет ответственности за отношения, используется атрибут mappedBy. Он ссылается на имя свойства связи на стороне владельца.

```
@Entity
public class EmployeeDetail implements Serializable {
    private static final long serialVersionUID = 5L;

    @Id
    @GenericGenerator(
            name = "gen",
            strategy = "foreign",
            parameters = @Parameter(name = "property", value = "employee")
    )
    @GeneratedValue(generator = "gen")
    private Long employeeId;

    @Column(name = "street")
    private String street;

    @Column(name = "city")
    private String city;

    @Column(name = "state")
    private String state;

    @Column(name = "country")
    private String country;

    @OneToOne
    @PrimaryKeyJoinColumn
    private Employee employee;
```

Рисунок 5.2.3 Класс EmployeeDetail.

В классе EmployeeDetail есть «внешний генератор», который отвечает за получение значения идентификатора из значения идентификатора сущности Employee.

Давайте посмотрим на фрагмент кода сохранения ассоциации в базу данных.

```
SimpleDateFormat sdfout = new SimpleDateFormat("yyyy.MM.dd");
String date = "2012.12.20";
Employee employee = new Employee("Ivan", "Spresov",
        new java.sql.Date(sdfout.parse(date).getTime()), "3456345345");
EmployeeDetail employeeDetail = new EmployeeDetail("Golodeda", "Minsk", "XXX", "Belarus");
employee.setEmployeeDetail(employeeDetail);
employeeDetail.setEmployee(employee);
log.info(employee);
session.save(employee);
session.flush();
log.info(employee);
session.close();
```

Рисунок 5.2.4 Фрагмент кода по сохранению ассоциации.

5.3 Hibernate аннотация @OneToMany.

Связь один-ко-многим, как и многие-к одному, является двунаправленной. При этом сторона, на которой находится один-ко-многим, является владельцем связи. Давайте рассмотрим конфигурацию на примере.

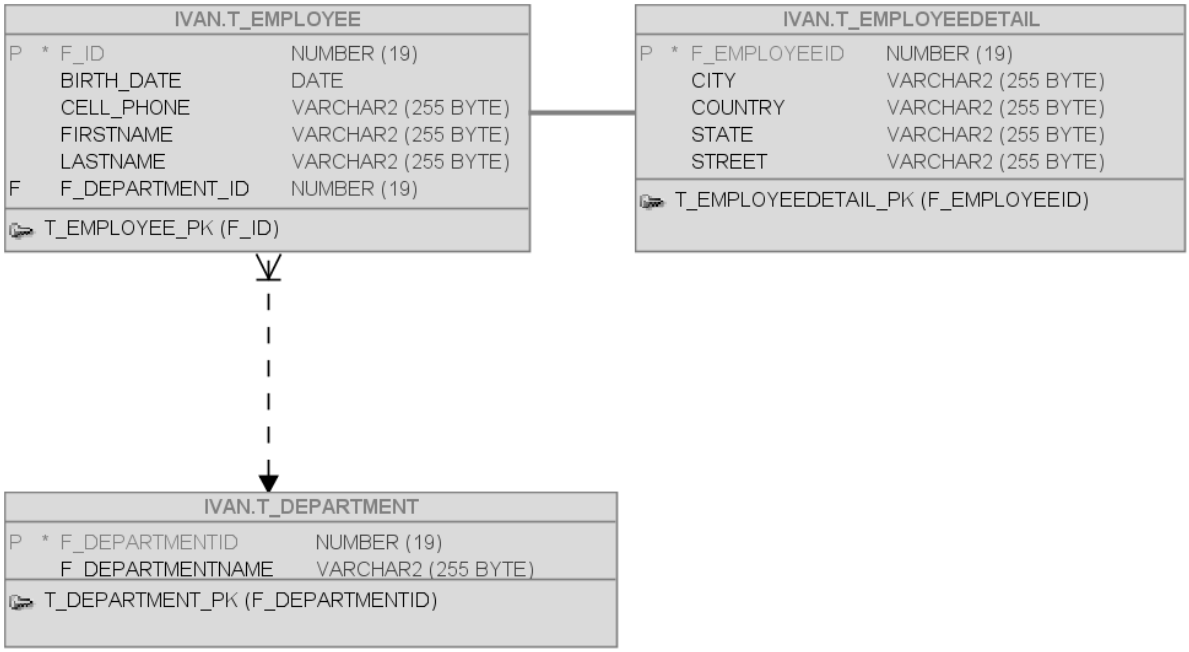

Рисунок 5.3.1 Ассоциация один-к одному и многие-к-одному на примере Department и Employee.

Давайте посмотрим на структуру классов.

```
@Entity
@SequenceGenerator(name = "PK", sequenceName = "t_department_seq")
public class Department implements Serializable {
    private static final long serialVersionUID = 6L;
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "PK")
    private Long departmentId;

    @Column
    private String departmentName;

    @OneToMany(mappedBy = "department")
    private Set<Employee> employees;
```

Рисунок 5.3.2 Класс Department

@OneToMany аннотации определяют многозначную связь с коллекцией сущностей. В данной конфигурации, Department является хозяином связи.

```java
@Entity
@SequenceGenerator(name = "PK", sequenceName = "t_employee_seq")
public class Employee implements Serializable {
    private static final long serialVersionUID = 4L;

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "PK")
    private Long id;

    @Column(name = "firstname")
    private String firstname;

    @Column(name = "lastname")
    private String lastname;

    @Column(name = "birth_date")
    private Date birthDate;

    @Column(name = "cell_phone")
    private String cellphone;

    @OneToOne(mappedBy = "employee", cascade = CascadeType.ALL)
    private EmployeeDetail employeeDetail;

    @ManyToOne
    @JoinColumn(name = "f_department_id")
    private Department department;
```

Рисунок 5.3.3 Класс Employee

@ManyToOne аннотации определяет однозначную ассоциацию на другую сущность класса, который связан с другой сущностью по many-to-one . Как правило, нет необходимости указывать целевой объект явно, поскольку он обычно может быть получен из типа объекта, на который ссылается.

@JoinColumn используется для определения колонки внешнего ключа.

5.4 HIBERNATE АННОТАЦИЯ @MANYTOMANY.

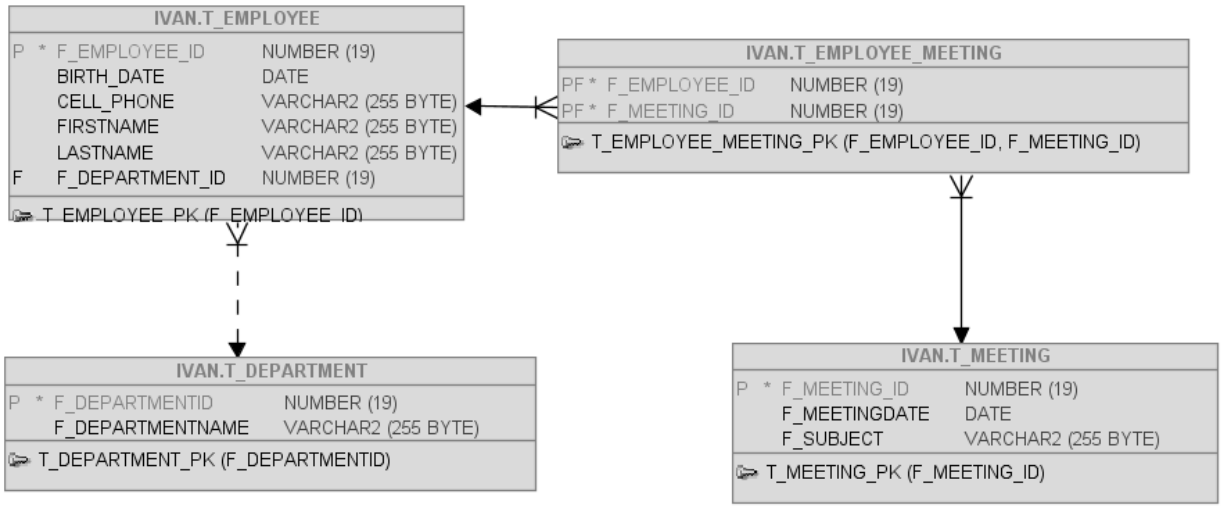

Рисунок 5.4.1 Ассоциация один-к одному и многие-ко-многим на примере Meeting и Employee

ТЕМА 6. HQL: ЯЗЫК ЗАПРОСОВ HIBERNATE.

6.1 Описание и структура языка HQL.

Hibernate Query Language (HQL) - это объектно-ориентированный язык запросов, похожий на SQL, но вместо операций над таблицами и колонками, HQL работает с persistent-объектами и их свойствами.

Запрос HQL представляет собой стоку запроса, переданную в метод hibernate сессии – createQuery(). Для получения результата запроса у объекта Query вызывается метод list(), возвращающий результат в виде элементов коллекции List или метод uniqueResult(), возвращающий единственный элемент.

Синтаксис HQL во многом копирует синтаксис SQL, но есть исключения:

- в качестве имен таблиц используются имена классов;
- в качестве столбцов используются имена полей класса.

Основные зарезервированные слова:

- SELECT
- WHERE
- ORDER BY
- GROUPE BY
- HAVING
- INSERT
- DELETE
- UPDATE

Для указания источника данных используется ключевое слово ***From*** и имя класса.

Запрос на загрузку в память всех элементов таблицы персон может быть представлен в одном из вариантов приведенных ниже.

```
String hql = "FROM Employee";
Query query = session.createQuery(hql);
List<Employee> results = query.list();
```

Рисунок 6.1.1 Запрос на загрузку всех элементов таблицы Employee

```
String hql = "FROM Employee AS E";
Query query = session.createQuery(hql);
List<Employee> results = query.list();
```

Рисунок 6.1.2 Запрос на загрузку всех элементов таблицы Employee с применением алиаса.

Условие ***Select*** предоставляет больше контроля над результатом вывода чем условие from. Если вы хотите вывести не все поля объекта, тогда используйте select.

```
String hql = "SELECT E.firstname FROM Employee E";
Query query = session.createQuery(hql);
List<String> results = query.list();
for (String result : results) {
   log.info(result);
}
```

Рисунок 6.1.3 Запрос на загрузку всех имен таблицы Employee.

Если вы хотите сузить результат выборки, то используйте условие where.

```
String hql = "SELECT E FROM Employee E WHERE E.employeeId=250";
Query query = session.createQuery(hql);
List<Employee> results = query.list();
```

Рисунок 6.1.3 Использование условия Where.

6.2 Операции языка HQL.

Для выражения ***Where*** можно использовать следующие ключевые слова:

- =, >=, <=, <>, != — операторы условия;
- like — задает условия для строк;
- in, not in, between, is null, is not null, is empty, is not empty, member of и not member of;
- current_date(), current_time(), and current_timestamp() — возвращает текущие дату, время и временную метку.
- substring(), trim(), lower(), upper() — методы строки.

Для сортировки ваших результатов применяется условие ***Order BY*** с двумя параметрами:

- ASC – по возрастанию;
- DESC – по убыванию.

```
String hql = "SELECT E FROM Employee E ORDER BY  E.employeeId ASC";
Query query = session.createQuery(hql);
List<Employee> results = query.list();
```

Рисунок 6.2.1 Пример использования сортировки

Условие ***Group By*** применяется для группировки собранных данных по какому-либо свойству объекта.

```
String hql = "SELECT count(E),E.firstname FROM Employee E GROUP BY E.firstname";
Query query = session.createQuery(hql);
List results = query.list();
```

Рисунок 6.2.2 Пример использования группировки по имени сотрудника

В HQL можно использовать именованный параметр Named Parameters для передачи значения переменной в запрос.

```
String hql = "SELECT E.firstname FROM Employee E WHERE E.employeeId=:employeeId ";
Query query = session.createQuery(hql);
query.setParameter("employeeId",2501);
List results = query.list();
```

Рисунок 6.2.3 Пример использования именованного параметра

HQL поддерживает агрегационные функции SQL:

- avg(property name) среднее значение по указанному свойству.
- count(property name or *) количество записей;
- max(property name) максимальное значение для свойства;
- min(property name) минимальное значение для свойства;
- sum(property name) общая сумма по свойству name;

```
public static void main(String... args) throws Exception {
    Locale.setDefault(Locale.US);
    HibernateUtil util = HibernateUtil.getInstance();
    Session session = util.getSession();
    Transaction transaction = session.beginTransaction();
    Query query = session.createQuery("select count(distinct firstname) from Employee");
    Long results = (Long)query.uniqueResult();
    transaction.commit();
    log.info(results);
    session.close();
}
```

Рисунок 6.2.4 Пример использования агрегационной функции

Для HQL запроса можно задавать пэджинацию результата. Пагинация – это разбиение результата на страницы, т.е. на коллекции-части ограниченного размера. Для пагинации в hibernate существуют следующие методы:

Query setFirstResult(int startPosition) – устанавливает стартовую позицию для результата выполнения запроса.

Query setMaxResults(int maxResult) устанавливает максимальное количество элементов результата выполнения запроса.

Следующий пример выдаст результат, состоящий максимум из двух элементов, начинающихся с первого элемента выборки.

```
Query query = session.createQuery("from Employee");
query.setFirstResult(0);
query.setMaxResults(2);
```

Рисунок 6.2.5 Пример использования запроса с педжинацией

Зарезервированное слово ***Update*** применяется для обновления полей и свойств объектов в HQL.

```
        Query qry = session.createQuery("update Product p set p.proName=?
here p.productId=111");
        qry.setParameter(0,"updated..");
        int res = qry.executeUpdate();
```

Рисунок 6.2.6 Запрос на обновление

Для удаления одного или более объектов применяется зарезервированное слово ***Delete***.

```
Query query = session.createQuery("delete from Product");
Integer results = query.executeUpdate();
```

Рисунок 6.2.6 Запрос на удаление

Для вставки одной записи из другой или другого объекта применяется зарезервированное слово ***Insert***.

```
Query query = session.createQuery("insert into Employee (firstname,lastname,birthDate,cellphone) " +
    "select firstname,lastname,birthDate,cellphone from Employee where employeeId=:employeeId");
query.setParameter("employeeId",1501);
Integer results = query.executeUpdate();
```

Рисунок 6.2.6 Запрос на вставку

ТЕМА 7. ЗАПРОСЫ CRITERIA.

7.1 Описание структуры Criteria instance.

Hibernate Criteria Queries – представляет объектно-ориентированную альтернативу HQL.

Многим разработчикам писать запросы в виде java кода понятнее и удобнее. По этой причине Hibernate Criteria получили распространение.

Для формирования запроса необходимо получить объект Criteria из объекта Session. В качестве параметра в метод получения Criteria передать класс, ассоциированный с таблицей в БД. После этого добавить, если требует бизнес-логика, ограничения или сортировку по полям класса. В конце вызвать метод list() у объекта Criteria.

```
Criteria criteria = session.createCriteria(Employee.class);
List results = criteria.list();
```

Рисунок 7.1.1 Запрос на вставку

7.2 Операции Criteria

Для сужения выборки в критериях используется класс Restrictions. Он является аналогом использования зарезервированного слова ***WHERE***.

Для поиска объекта, удовлетворяющего условию равенства, используется метод eq(),класса Restrictions.

```
Criteria criteria = session.createCriteria(Employee.class);
criteria.add(Restrictions.eq("name", "Yuli"));
List results = criteria.list();
```

Рисунок 7.2.1 Criteria с ограничением равенства

Для поиска объекта, удовлетворяющего условию «>», используется метод gt() класса Restrictions.

```
Criteria criteria = session.createCriteria(Employee.class);
criteria.add(Restrictions.gt("salary", 400));
List results = criteria.list();
```

Рисунок 7.2.2 Criteria с ограничением «>»

Для поиска объекта, удовлетворяющего условию «<», используется метод lt() класса Restrictions.

```
Criteria criteria = session.createCriteria(Employee.class);
criteria.add(Restrictions.lt("salary", 400));
List results = criteria.list();
```

Рисунок 7.2.3 Criteria с ограничением «<»

Для поиска объекта, удовлетворяющего строковому шаблону с учетом регистра, используется метод like() класса Restrictions.

```
Criteria criteria = session.createCriteria(Employee.class);
criteria.add(Restrictions.like("name", "Ma%"));
//criteria.add(Restrictions.ilike("name", “ма%"));
List results = criteria.list();
```

Рисунок 7.2.4 Criteria с ограничением для строк

Для поиска объекта, входящего в диапазон, используется метод between() класса Restrictions.

```
Criteria criteria = session.createCriteria(Employee.class);
criteria.add(Restrictions.between("age", 25, 34));
List results = criteria.list();
```

Рисунок 7.2.5 Criteria с ограничением диапазона

Для поиска объекта, представленного не null/null значением, используется метод isNotNul()/ isNull() класса Restrictions.

```
Criteria criteria = session.createCriteria(Employee.class);
criteria.add(Restrictions.isNotNull("name"));
//criteria.add(Restrictions.isNull("name"));
List results = criteria.list();
```

Рисунок 7.2.6 Criteria с ограничением на не null значение

Для использования логических операций AND или OR, применяется класс LogicalExpression.

```
Criteria criteria = session.createCriteria(Employee.class);
Criterion salary = Restrictions.gt("salary", 400);
Criterion age = Restrictions.gt("age", 27);
LogicalExpression andExp = Restrictions.and(salary, age);
//LogicalExpression orExp = Restrictions.or(salary, age);
criteria.add(andExp);
List results = criteria.list();
```

Рисунок 7.2.7 Criteria с LogicalExpression

Для сортировки результатов запроса используется метод addOrder(Order-Type), принимающий объект типа сортировки.

```
Criteria criteria = session.createCriteria(Employee.class);
// To sort records in descending order
criteria.addOrder(Order.desc("salary"));
// To sort records in ascending order
//criteria.addOrder(Order.asc("salary"));
List results = criteria.list();
```

Рисунок 7.2.8 Criteria с сортировкой

Для использования страничного вывода в hibernate Criteria существуют следующие методы:

- **public** Criteria ***setFirstResult*(int** firstResult**)**
- **public** Criteria ***setMaxResults*(int** maxResults**)**

```
Criteria cr = session.createCriteria(Product.class);
cr.setFirstResult(0);
cr.setMaxResults(3);
List results = cr.list();
```

Рисунок 7.2.9 Criteria со страничным выводом

7.3 Проекции и агрегация в Criteria.

Hibernate Criteria как и Hibernate Query поддерживают агрегационные функции:

- avg(property name) среднее значение по указанному свойству.
- rowCount() количество записей;
- max(property name) максимальное значение для свойства;
- min(property name) минимальное значение для свойства;
- sum(property name) общая сумма по свойству name;

Для их использования устанавливается проекция на поля класса.

Проекция на количество записей в таблице Employee может быть выполнена следующим образом.

```
Criteria criteria = session.createCriteria(Employee.class);
// To get total row count.
criteria.setProjection(Projections.rowCount());
List results = criteria.list();
log.info(results);
```

Рисунок 7.3.1 Criteria с проекцией на количество элементов

Проекция на среднее значение поля salary в таблице Employee может быть выполнена следующим образом.

```
Criteria criteria = session.createCriteria(Employee.class);
// To get average of a property.
criteria.setProjection(Projections.avg("salary"));
List results = criteria.list();
log.info(results);
```

Рисунок 7.3.2 Criteria с проекцией на среднее значение поля salary

Проекция на количество уникальных значений поля «name» в таблице Employee может быть выполнена следующим образом.

```
Criteria criteria = session.createCriteria(Employee.class);
// To get distinct count of a property.
criteria.setProjection(Projections.countDistinct("name"));
List results = criteria.list();
log.info(results);
```

Рисунок 7.3.3 Criteria с проекцией на количество уникальных значений поля «name»

Проекция на максимальное значений поля «age» в таблице Employee может быть выполнена следующим образом.

```
Criteria criteria = session.createCriteria(Employee.class);
// To get maximum of a property.
criteria.setProjection(Projections.max("age"));
List results = criteria.list();
log.info(results);
```

Рисунок 7.3.4 Criteria с проекцией на максимальное значений поля «age»

Проекция на минимальное значений поля «salary» в таблице Employee может быть выполнена следующим образом.

```
Criteria criteria = session.createCriteria(Employee.class);
// To get minimum of a property.
criteria.setProjection(Projections.min("salary"));
List results = criteria.list();
log.info(results);
```

Рисунок 7.3.5 Criteria с проекцией на минимальное значений поля «salary»

Проекция на сумму значений поля «salary» в таблице Employee может быть выполнена следующим образом.

```
Criteria criteria = session.createCriteria(Employee.class);
// To get minimum of a property.
criteria.setProjection(Projections.sum("salary"));
List results = criteria.list();
log.info(results);
```

Рисунок 7.3.5 Criteria с проекцией на сумму значений поля «salary»

ТЕМА 8. ТРАНЗАКЦИИ И ПАРАЛЛЕЛИЗМ.

8.1 Использование транзакций.

В среде Enterprise разработки должны учитываться требования целостности, непротиворечивости, долговечности и изолированности данных при операциях с СУБД. Обеспечиваются эти правила только благодаря транзакциям, которые поддерживают только два состояния:

1. Состояние фиксирования операции над данными.
2. Откат состояния данных до начала запуска транзакции в случае возникновения ошибки при фиксировании изменений.

Управление транзакциями предоставляется разработчикам приложений через интерфейс Transaction. Интерфейс *Transaction* предоставляет методы для объявление границ транзакций БД. Пример корректного использования транзакции приведен на рис. 8.1.1.

```
Locale.setDefault(Locale.US);
HibernateUtil util = HibernateUtil.getInstance();
Session session = util.getSession();
Transaction transaction = null;
try {
    transaction = session.beginTransaction();
    Employee e = new Employee("Ivan", "Spresov", new Date(2012, 12, 25), "5634534543");
    log.info(e);
    transaction.commit();
} catch (Exception e) {
    if (transaction != null) {
        try {
            transaction.rollback();
        } catch (HibernateException he) {
            //log he and rethrow e
        }

    }
    throw e;
} finally {
    try {
        session.close();
    } catch (HibernateException he) {
        throw he;
    }
}
```

Рисунок 8.1.1 Фиксирование транзакции и обработка ошибки

Вызов *session.beginTransaction()* отмечает начало транзакции БД. В случае неуправляемой среды, этот вызов стартует JDBC транзакцию на JDBC соединении. В случае управляемой среды, он начинает новую JTA транзакцию, если нет текущей транзакции JTA или присоединяется к текущей JTA транзакции. Это все обрабатывается Hibernate – вам не нужно заботиться о реализации этого.

Вызов *transaction.commit()* синхронизирует состояние Session с БД. Hibernate затем фиксирует совершаемую транзакцию тогда и только тогда, если *beginTransaction()* начал новую транзакцию. Если *beginTransaction()* не начал транзакцию БД, то *commit()* только синхронизирует состояние Session с БД; это оставлено на усмотрение ответственного участника (код, который начал транзакцию в первую очередь) к моменту завершения транзакции. Это согласуется с поведением, определенным JTA.

Если *transaction.commit()* выбрасывает исключение, то мы должны принудительно откатить транзакцию, вызывая *transaction.rollback()*. Этот метод либо осуществляет немедленный откат или помечает транзакцию, как «только для отката».

8.2 Уровни изоляции транзакций

Для обеспечения производительности работы СУБД транзакции выполняются параллельно. Но их параллельное выполнение может привести к неприятным явлениям:

- Потерянное обновление
- Грязное чтение
- Неповторяемое чтение
- Вторая проблема обновлений
- Фантомное чтение.

Для избавления от таких явлений, с определенной степенью производительности, добавлен механизм изоляции транзакций.

Предусмотрены стандартные уровни изоляции, определяющиеся стандартом ANSI SQL, но без привязки к SQL БД. JTA определяет те же уровни изоляции, и вы будете использовать эти уровни, чтобы позднее заявить желаемые уровень изоляции транзакции:

— *Неподтверждённое чтение* – разрешает грязные чтения, но без потери обновлений. Одна транзакция может не писать в строку, если другая незафиксированная транзакция уже записывает туда. Однако, любая транзакция может читать любые строки. Этот уровень изоляции может быть реализован с использованием эксклюзивной блокировки записи.

— *Подтверждённое чтение* - разрешает неповторяемые чтения, но не грязные чтения. Это может быть достигнуто с помощью мгновенных общих блокировок чтения и эксклюзивной блокировки записи. Однако, незафиксированные пишущие транзакции блокируют все другие транзакции на доступ к строке.

— *Повторяемое чтение* – не допускает ни неповторяемого чтения, ни грязного чтения. Фантомное чтение может произойти. Это может быть достигнуто с использованием общих блокировок на чтение и эксклюзивной блокировки на запись.

Блок чтения пишущих транзакций (но не другие операции чтения) и блок пишущих транзакций блокируют все другие транзакции.

— *Упорядоченный (сериализуемый)* – обеспечивает строгую изоляцию транзакций. Он эмулирует последовательное выполнение операций, как если бы операция была выполнена одна за другой последовательно, а не параллельно. Упорядоченность не может быть реализована с использованием только блокировки на уровне строк; это должен быть другой механизм, который предотвращает становление видимой, только что вставленной строки, из транзакции, которая уже выполняет запрос, возвращающий строку.

8.3 Установка уровня изоляции транзакций.

Каждое JDBC соединение к БД использует уровень изоляции, установленный самой БД, как правило, это чтение подтвержденное или повторяемое чтение. Это значение по умолчанию может быть изменено в конфигурации БД. Вы также можете установить уровень изоляции транзакций для JDBC соединений, с помощью опции конфигурации Hibernate:

hibernate.connection.isolation=4

Hibernate затем установит этот уровень изоляции на каждое соединении JDBC, полученное из пула соединений, до начала транзакции.

В java.sql.Connection определены константы, задающие уровень изоляции:

1 – изоляция уровня чтения неподтвержденного;

2 – изоляция уровня чтения подтвержденного;

4 – изоляция уровня повторяемого чтения;

8 – упорядоченная изоляция.

Hibernate никогда не меняет уровень изоляции соединений, полученных из источника данных, предоставляемых сервером приложений в управляемой среде. Вы можете изменить стандартный уровень изоляции с помощью конфигурации сервера приложений.

Установка уровня изоляции представляет собой глобальную опцию, которая затрагивает все соединения в транзакциях. Время от времени, полезно указать более строгую блокировку для конкретной транзакции. Hibernate позволяет явно указать использовать пессимистическую блокировку.

```
<property name="hibernate.dialect">
   org.hibernate.dialect.MySQLDialect
</property>
<property name="hibernate.connection.username">root</property>
<property name="hibernate.connection.password">yuli</property>
<property name="hibernate.connection.pool_size">10</property>
<property name="hibernate.connection.isolation">2</property>
<property name="show_sql">true</property>
<property name="hibernate.hbm2ddl.auto">update</property>
```

Рисунок 8.3.1 Установка уровня изоляции транзакций

8.4 Детализация сессии.

Наиболее распространенное использование сессии — это открытие новой сессии для каждого запроса клиента (например, запроса веб-браузера) и старт новой транзакции. После выполнения бизнес-логики, мы фиксируем изменения транзакции БД и закрываем сессию перед отправкой ответа клиенту (см. рис. 8.4.2).

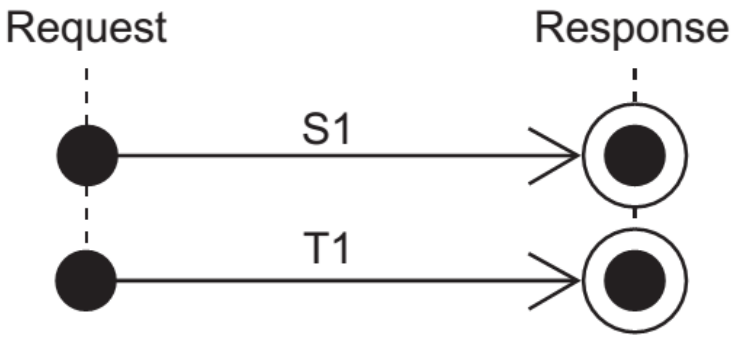

Рисунок 8.4.1 Модель – одна сессия и транзакция на http запрос.

Сессии (S1) и транзакции БД (T1) имеют одну и ту же степень детализации. Если вы работаете с концепцией транзакций приложения, то это простой подход – все что вам нужно в вашем приложении. Нам нравится называть такой подход *сессия-на-запрос*.

Если вам необходима длительная транзакция приложения, то вам могут помочь несвязанные объекты (и Hibernate поддерживает оптимистическую блокировку), осуществите его, используя тот же подход (см. рис. 8.4.2).

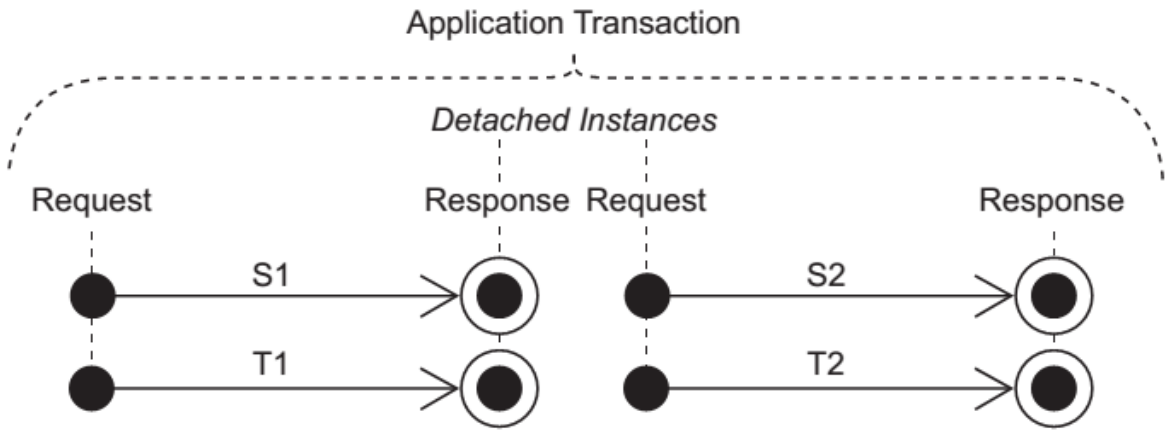

Рисунок 8.4.2 Модель – Много сессий на транзакцию приложения.

Предположим, что ваша транзакция приложения охватывает два клиентских запроса/ответа, например два HTTP-запроса в веб-приложении. Вы можете загрузить интересующие объекты в первой сессии, а затем связать их к новой сессии после того, как они были изменены пользователем. Hibernate автоматически проверит версию. Время между (S1, T1) и (S2, T2) может быть «длинным», настолько длинным, насколько это нужно пользователю, чтобы сделать его изменения. Этот подход известен также, как *сессия-на-запрос-с-несвязанными-объектами*.

Кроме того, вы можете предпочесть использовать одну сессию, которая охватывает несколько запросов вашей транзакции приложения. В этом случае, вам не нужно беспокоиться о присоединении несвязанных объектов, так как объекты остаются хранимыми в рамках одной длительной сессии (см. рис. 8.4.3). Конечно, Hibernate по-прежнему несет ответственность за выполнение оптимистической блокировки.

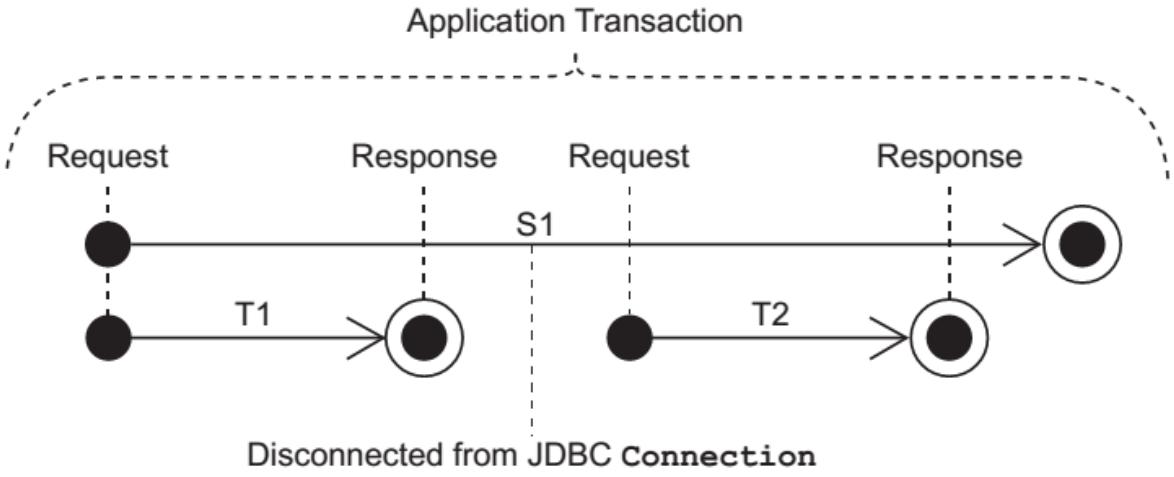

Рисунок 8.4.3 Модель – длинная сессий на транзакцию приложения.

Сессия hibernate может быть безопасно сохранена, например, в сервлете *HttpSession*. При этом базовое JDBC соединение должно быть закрыто и новое соединение должно быть получено на следующий запрос. Вы можете использовать методы *disconnect()* и *reconnect()* интерфейса сессии для того, чтобы освободить соединение и затем получать новое соединение. Такое подход известен, как *сессия-на-транзакцию-приложения* или *длинная сессия*.

Как правило, ваш первый выбор должен сохранять Hibernate сессию открытой не более чем на одну транзакцию БД (сессия-на-запрос). После завершения первоначальной транзакции БД, чем дольше сессия остается открытой, тем больше шансов, что она удерживает устаревшие данные в КЭШе хранимых объектов.

Вопрос о реализации транзакций приложения и объем сессии важен при построении приложения. Мы обсудим реализацию стратегии, с примерами в главе 8 «Реализация транзакций приложения».

8.5 Настройка Кэша второго уровня.

Hibernate кэш второго уровня является процессным или кластерным; все сессии обладают одним и тем же КЭШем второго уровня. Кэш второго уровня, имеет область *SessionFactory*.

Хранимые экземпляры хранятся в КЭШе второго уровня в разобранном виде. Процесс разбора похож на сериализацию объектов.

Различные виды данных требуют различной политики кэширования: соотношение чтение к записи изменяется, размер БД изменяется, и некоторые таблицы используются совместно с другими приложениями. Так что, кэш второго уровня настраиваемся под детализацию каждого индивидуального класса или коллекцию ролей. Это позволяет, например, разрешить кэш второго уровня для справочных данных и запретить его для классов, представляющих финансовые записи. Политика КЭШа включает в себя настройку следующих параметров:

- Включен ли кэш второго уровня
- Стратегию параллелизма Hibernate
- Политика истекания срока кэширования (такую, как тайм-аут, LRU, зависимую от ОП)
- Физическое устройство КЭШа (в памяти, индексируемые файлы, кластерная репликация)

Не все классы имеют выгоду от кэширования, поэтому чрезвычайно важно иметь возможность отключить кэш второго уровня. Кэш-память полезна только для классов, которые в большинстве своем только считываются. Если у вас есть данные, которые обновляются чаще, чем читаются, запрещайте кэш второго уровня, даже если все остальные условия для кэширования верны! Кроме того, кэш второго уровня может быть опасен в системах, которые разделяют данные с другими приложениями, которые могут эти данные изменить.

Стратегия параллелизма является посредником; она несет ответственность за хранение элементов данных в кэш памяти и извлечение их из КЭШа. Это важная роль, поскольку она определяет семантику изоляции транзакций для этого конкретного пункта. Вам придется принять решение по каждому хранимому классу, какую стратегию параллелизма кэша использовать, если вы хотите разрешить кэш второго уровня.

Есть четыре встроенных стратегии параллелизма, представляющие снижение уровня строгости, в терминах изолированности транзакции:

- *Транзакционная* – доступна только в управляемой среде. Она гарантирует полную изоляцию транзакций до повторяемого чтения, если это требуется.
- *Чтение-запись* – поддерживает изоляцию чтения подтвержденного, используя механизм временных меток. Она доступна только в некластерных средах.
- *Нестрогое-чтение-запись* - не дает никакой гарантии согласованности между КЭШем и БД. Если есть возможность одновременного доступа к одной сущности, то вам необходимо настроить достаточно короткий срок истечения таймаута.
- *Только-для-чтения* – данная стратегия подходит для данных, которые никогда не меняются. Используйте её только для справочных данных.

Далее необходимо выбрать поставщика КЭШа для всего приложения.

Следующие поставщики встроены в Hibernate:

- *EHCache* предназначен для простого процессного кэширования в одной JVM. Он может кэшировать в памяти или на диске и поддерживает опциональный Hibernate кэш результатов запроса.
- *OpenSymfony OSCache* – это библиотека, которая поддерживает кэширование в памяти и на диске в одной JVM, с богатым набором политик истечения и поддержкой КЭШа запросов.
- *SwarmCache* – это кластерный кэш, основанный на JGroups. Он использует кластерное аннулирование, но не поддерживает кэш Hibernate запросов.
- *JBossCache* – это полностью транзакционно-репликационный кластеризованный кэш, также основанных на *JGroups*. Кэш запросов Hibernate поддерживается, предполагая, что часы в кластере синхронизированы.

Пример файла конфигурации представлен на рисунке 8.5.1.

```
<hibernate-configuration>
    <session-factory>
        <property name="hibernate.connection.driver_class">
            com.mysql.jdbc.Driver
        </property>
        <property name="hibernate.connection.url">
            jdbc:mysql://127.0.0.1:3306/person_DB
        </property>
        <property name="hibernate.dialect">
            org.hibernate.dialect.MySQLDialect
        </property>
        <property name="hibernate.connection.username">root</property>
        <property name="hibernate.connection.password">yuli</property>
        <property name="hibernate.connection.pool_size">10</property>
        <property name="hibernate.connection.isolation">2</property>
        <property name="show_sql">true</property>
        <property name="hibernate.hbm2ddl.auto">create</property>

        <!--Cache configuration-->
        <property name="cache.provider_class">org.hibernate.cache.EhCacheProvider</property>
        <property name="cache.use_query_cache">true</property>
        <property name="cache.use_second_level_cache">true</property>
        <!--Mapping-->
        <mapping class="by.academy.it.pojos.Employee"/>
        <mapping class="by.academy.it.pojos.EmployeeDetail"/>
        <mapping class="by.academy.it.pojos.Department"/>
        <mapping class="by.academy.it.pojos.Meeting"/>
    </session-factory>
</hibernate-configuration>
```

Рисунок 8.5.1 Конфигурация hibernate для КЭШа 2 уровня

После выбора поставщика КЭШа необходимо настроить его конфигурацию. В нашем примере выбор пал на поставщика КЭШа Ehcache. Конфигурация описывается в файле ehcache.xml (рис. 8.5.2)

```
<ehcache>
    <diskStore path="java.io.tmpdir"/>

    <defaultCache
            maxElementsInMemory="20000"
            eternal="false"
            timeToIdleSeconds="300"
            timeToLiveSeconds="300"
            overflowToDisk="true"
            diskPersistent="false"
            diskExpiryThreadIntervalSeconds="300"
            memoryStoreEvictionPolicy="LRU"/>

    <cache name="by.academy.it.pojos.Employee"
           maxElementsInMemory="15000"
           eternal="true"
           overflowToDisk="false"/>
</ehcache>
```

Рисунок 8.5.2 Конфигурация ehcache

После завершения конфигурации hibernate и ehcache, необходимо отметить аннотацией ***@Cache*** классы, которые будут размещаться в кеше 2 уровня (рис. 8.5.3).

```java
@Entity
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
public class Employee implements Serializable {
    private static final long serialVersionUID = 4L;

    @Id
    @Column(name = "F_EMPLOYEE_ID")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long employeeId;

    @Column(name = "firstname")
    private String firstname;

    @Column(name = "lastname")
    private String lastname;

    @Column(name = "birth_date")
    private Date birthDate;

    @Column(name = "cell_phone")
    private String cellphone;

    @Version
    private int version;

    @OneToOne(mappedBy = "employee", cascade = CascadeType.ALL)
    private EmployeeDetail employeeDetail;

    @ManyToOne
    @JoinColumn(name = "f_department_id")
    private Department department;

    @ManyToMany(cascade = {CascadeType.ALL})
    @JoinTable(name = "T_EMPLOYEE_MEETING",
            joinColumns = {@JoinColumn(name = "F_EMPLOYEE_ID")},
            inverseJoinColumns = {@JoinColumn(name = "F_MEETING_ID")})
    private Set<Meeting> meetings = new HashSet<~>();
```

Рисунок 8.5.3 Конфигурация класса сущности