

Титульный лист материалов по дисциплине

ДИСЦИПЛИНА	<b>Разработка клиент-серверных приложений</b> <small>полное название дисциплины без аббревиатуры</small>
ИНСТИТУТ	<b>Информационных технологий</b>
КАФЕДРА	<b>Инструментального и прикладного программного обеспечения</b> <small>полное название кафедры</small>
ГРУППА/Ы	<b>ИКБО-01/02/03/12/13/16-18</b> <small>номер групп/ы, для которых предназначены материалы</small>
ВИД УЧЕБНОГО МАТЕРИАЛА	<b>Материал к практическим занятиям</b> <small>лекция; материал к практическим занятиям; контрольно-измерительные материалы к практическим занятиям; руководство к КР/КП, практикам</small>
ПРЕПОДАВАТЕЛЬ	<b>Строганкова Наталья Владимировна</b> <small>фамилия, имя, отчество</small>
СЕМЕСТР	<b>5</b> <small>указать номер семестра обучения</small>

## **ПРАКТИЧЕСКАЯ РАБОТА № 4**

### **1 Цель работы**

Целью данной лабораторной работы является ознакомление с технологиями ORM и реализация взаимодействия с базой данных посредством ORM Hibernate.

### **2 Теоретическая часть**

Краткое введение в технологии ORM было осуществлено в первой части учебно-методического пособия. Существует множество реализаций технологии ORM на языке Java. Одна из этих реализаций — это Hibernate. Для того, чтобы использовать функционал Hibernate, необходимо иметь:

1) библиотеку ORM Hibernate (ее можно скачать с официального сайта производителя);

2) JDBC (Java Data Base Connectivity) — это библиотека, позволяющая взаимодействие с различными СУБД. JDBC основан на концепции драйверов, позволяющих получить соединение с БД по специальному URL. Каждая БД имеет свою JDBC-библиотеку, которую можно загрузить с официального сайта используемой БД.

#### **2.1 Создание конфигурации подключения к БД**

Основным классом Hibernate-технологии, позволяющей читать, изменять, добавлять или удалять данные из СУБД, является Session. Для получения сессии необходимо изначально установить конфигурацию соединения с БД и получить с помощью созданной конфигурации фабрику сессий (SessionFactory), которая в итоге будет нам поставлять сессии. Конфигурацию соединения можно прописать двумя способами:

1) через файл hibernate.cfg.xml;

2) через класс org.hibernate.cfg.Configuration.

Сейчас рассмотрим эти два способа на примерах:

## Файл hibernate.cfg.xml

```
1. <!DOCTYPE hibernate-configuration SYSTEM
2. "http://www.hibernate.org/dtd/hibernate configuration-
3. 3.0.dtd">
4. <hibernate-configuration> 4. <session-factory>
5. <property name = "connection.url">jdbc:
6. sqlserver://localhost:1433; database=data_base</property>
7. <property name="connection.driver_class">com.microsoft.
8. sqlserver.jdbc.SQLServerDriver</property>
9. <property name="connection.username">sa</property>
10. <property name="connection.password">1</property>
11. <property name="connection.pool_size">1</property>
12. <property name="current_session_context_class">
13. thread</property>
14. <property name="hbm2ddl.auto">update</property> 12. <property
15. name="show_sql">true</property>
16. <property name="dialect">org.hibernate.dialect.
17. SQLServerDialect</property>
18. </session-factory>
19. </hibernate-configuration>
```

Первые две строчки – подключение спецификации (шаблона), с его помощью мы не ошибемся в синтаксисе и будем точно знать, где какой тег должен будет находиться и какие атрибуты он может иметь.

Необходимая информация содержится в строчках с 5 по 13, где прописаны непосредственно свойства подключения в БД:

1) connection.url – url-строка подключения к БД. В данном примере осуществляется подключение к MS SQL Server. Разберем строчку детально:

jdbc: sqlserver: — это некоторая спецификация, через что мы подключаемся и к какому серверу;

://localhost:1433 — указываем ip-адресс или доменное имя компьютера с портом, к которому мы будем подключаться;

database=data\_base — указание, к какой базе мы будем подключаться;

2) `connection.driver_class` – указание, какой драйвер будет использоваться в качестве связующего звена с СУБД. Здесь указывается класс JDBC-драйвера, который подключен к проекту, в примере указан драйвер MS SQL Server;

3) `connection.username` – логин зарегистрированного пользователя СУБД;

4) `connection.password` – пароль;

5) `connection.pool_size` – данное свойство показывает, сколько соединений к БД может быть одновременно открыто;

6) `hbm2ddl.auto` — свойство, указывает что нужно сделать со схемой БД при инициализации. Может принимать такие значения:

- `update` – сверяет схему БД с имеющимися конфигурациями классов. Если были внесены какие-то изменения, они автоматически занесутся в БД, при этом данные, которые были занесены в базу, не изменятся;

- `create` – каждый раз при запуске приложения схема БД будет создаваться заново. Все данные, которые были занесены раньше, будут удалены;

- `create-drop` – каждый раз при запуске приложения схема БД будет создаваться заново, а при завершении – удаляться. Все данные, которые были занесены во время работы приложения, будут удалены по завершению приложения;

- `validate` – при инициализации будет совершена проверка соответствия конфигурации классов и схемы БД. Если мы внесли изменение в конфигурацию какого-то класса, а схема в БД не была изменена, сработает исключение;

7) `show_sql` – данное свойство указывает, будут ли выводиться SQL-запросы на консоль, которые отправляются на СУБД. В процессе отладки это бывает очень удобно;

8) `dialect` – `hibernate` может работать с разными БД, и каждая имеет свои особенности (генерация первичного ключа, страничный вывод, функции), нужно указать какой диалект будем использовать для создания запросов. В данном случае – диалект MS SQL Server.

Класс `org.hibernate.cfg.Configuration` используется для подготовки данных соединения к СУБД:

```
1. public static Configuration getConfiguration () {
2.     Configuration cfg = new Configuration ();
3.     cfg.setProperty ("hibernate.connection.url", "jdbc:
        sqlserver://localhost:1433; database=data_base");
4.     cfg.setProperty ("hibernate.connection.driver_class", "com.
5.     microsoft.sqlserver.jdbc.SQLServerDriver");
6.     cfg.setProperty ("hibernate.connection.username", "sa");
7.     cfg.setProperty ("hibernate.connection.password", "1");
8.     cfg.setProperty ("hibernate.connection.pool_size", "1");
9.     cfg.setProperty
        ("hibernate.current_session_context_class", "thread");
10.    cfg.setProperty ("hibernate.hbm2ddl.auto", "update");
11.    cfg.setProperty ("hibernate.show_sql", "true");
12.    cfg.setProperty
        ("hibernate.dialect", "org.hibernate.dialect.SQLServerDialect")
        ;
13.    return cfg;13. }
```

### 2.1.1 Класс подключения к БД

Когда создана конфигурация нужно создать класс, отвечающий за создание сессий, так называемая фабрика сессий. Пример кода:

```
1. private static SessionFactory sessionFactory =
    buildSessionFactory();
2. private static SessionFactory buildSessionFactory () {
3.     try {
4.         if (sessionFactory == null) {
5.             Configuration configuration = new Configuration ().configure
                ("HibernateUtil/hibernate.cfg.xml");
6.             StandardServiceRegistryBuilder builder = new
```

```

7. StandardServiceRegistryBuilder ().applySettings
   (configuration. getProperties ());
8. sessionFactory = configuration.buildSessionFactory
   (builder.build());
9. }
10. return sessionFactory;
11. } catch (Throwable ex) {
12. System.err.println ("Initial SessionFactory creation failed."
   + ex);
13. throw new ExceptionInInitializerError (ex);
14. }
15. }
16. public static Session getSession () {
17. return sessionFactory.openSession ();
18. }

```

В данном примере показано, как можно создать сессию с БД, используя ранее созданные конфигурационные параметры. В пятой строчке происходит использование конфигурационных данных, созданных с помощью файла `hibernate.cfg.xml`. Для реализации второго способа описания конфигурации можно использовать следующую строчку:

```

5. Configuration configuration = getConfiguration();

```

Метод `getSession` создает сессию, с помощью которой и будет происходить общение с БД.

## 2.2 Создание класса-сущности

БД позволяет хранить информацию в таблицах. Для того, чтобы это осуществить, у `Hibernate` существует специальный механизм — `mapping`.

`Mapping`-проецирование (сопоставление) `Java` классов с таблицами базы данных. Реализация `mapping` возможна через использование конфигурационных файлов `XML`, либо через аннотации. Так как описание `mapping` через файлы `XML` неудобно и уже устаревает, в примере использованы аннотации. Пример:

```

1. import javax.persistence.*;
2. @Entity
3. @Table (name = "people")
4. public class People {
5.     @Id
6.     @GeneratedValue (strategy = GenerationType.IDENTITY)
7.     @Column (name = "id", unique = true, nullable = false)
8.     private Integer id;
9.     @Column (name = "firstName")
10.    private String firstName;
11.    @Column (name = "lastName")
12.    private String lastName;
13.    @Column (name = "middleName")
14.    private String middleName;
15.    @Column ()
16.    private int year;
17.
18.    public Integer getId () {
19.        return id;
20.    }
21.    public void setId (Integer id) {
22.        this.id = id;
23.    }
24.    ...
25. }

```

Для того, чтобы созданный класс воспринимался Hibernate как некоторая сущность бизнес-модели, его необходимо пометить аннотацией `@Entity`.

Аннотация `@Table` позволяет задать имя, под которым данная сущность будет существовать в БД (позволяет задать имя таблицы), а также другие конфигурационные параметры. Если имя таблицы совпадает с именем класса, аннотацию можно не использовать.

В каждой сущности должен быть указан первичный ключ, который помечается `@Id`. Аннотация `@GeneratedValue` указывает, что данное поле будет генерироваться автоматически при создании нового элемента в таблице.

`@Column` — одна из основных аннотаций, которая указывает, что поле, описанное после него, будет храниться в БД.

Поле, над которым стоит аннотация `Id`, `GeneratedValue`, `Column`, должно быть либо примитивом, либо оберткой над этим примитивом: `String`; `java.util.Date`; `java.sql.Date`; `java.math.BigDecimal`; `java.math.BigInteger`.

По правилам хорошего тона класс нужно оформлять как просто POJO-объект, то есть поля должны иметь спецификатор доступа `private`. А доступ к этим полям осуществляется с помощью геттеров и сеттеров, как показано в примере на строчках 18–23.

В БД данная сущность будет выглядеть следующим образом (Рисунок 2.1).



Рисунок 2.1 – Структура класса-сущности People в БД

## 2.3 Регистрация классов-сущностей

Для того чтобы Hibernate при создании `SessionFactory` учитывала новую сущность, ее необходимо указать в конфигурации. Пример для файла `hibernate.cfg.xml`:

```
1. <hibernate-configuration>
2. <session-factory>
3. ...
4. <mapping class="Entity.People"/>
5. </session-factory>
6. </hibernate-configuration>
```



Тег `mapping` позволяет указать, какие классы-сущности мы должны учитывать при создании `SessionFactory`.

Пример для конфигурационного класса `org.hibernate.cfg`:

```
1. Configuration cfg = new Configuration ();
2. cfg.addAnnotatedClass (People.class);
```

## **2.4 Взаимодействие с БД: создание, удаление, чтение и изменение сущностей**

### **2.4.1 Создание объекта в БД**

Пример создания объекта (строки) в БД:

```
1. public static void create(Object o)
2. {
3. try {
4. Session session = HibernateUtil.getSession();
5. session.beginTransaction();
6. session.save (o);
7. session.getTransaction().commit();
8. session.close();
9. } catch (HibernateException e) {
10. e.printStackTrace();
11. }
```

В третьей строке мы обращаемся к созданному классу для получения сессии.

Четвертая строка — начало транзакции, шестая — конец транзакции или просто коммит, то есть те запросы, которые были между этими командами описаны, отправятся на сервер БД, и там появятся данные.

Пятая строка отвечает за генерацию кода, обеспечивающую создание объекта, при этом запрос будет выглядеть примерно следующим образом:

```
Hibernate: insert into people (firstName, lastName, middleName,
year) values (?, ?, ?, ?).
```

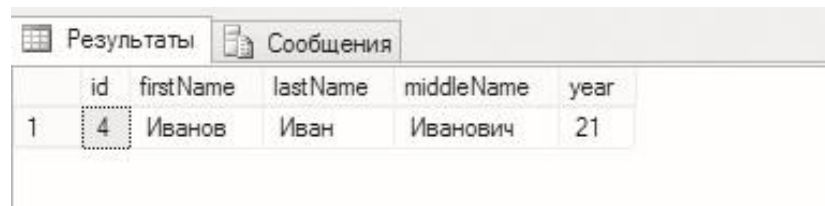
Пример сохранения объекта:

```

1. People people = new People ();
2. people.setFirstName ("Иванов");
3. people.setLastName ("Иван");
4. people.setMiddleName ("Иванович");
5. people.setYear (21);
6. create (people);

```

В БД появится запись, как показано на рисунке 2.2.



	id	firstName	lastName	middleName	year
1	4	Иванов	Иван	Иванович	21

Рисунок 2.2 – Результат добавления данных через функцию hibernate

## 2.4.2 Удаление объекта из БД

Существует два распространенных способа удаления объектов из БД: через объект или через id.

Через объект метод будет в точности такой же, как и при создании, только вместо save будет delete:

```

1. session.beginTransaction ();
2. session.delete (o);
3. session.getTransaction ().commit ();

```

Удаление с помощью HQL-запросов по id:

```

1. public static void delete (int id){
2. try {
3. Session session = HibernateUtil.getSession();
4. String stringQuery = "delete from People where id =: id";
5. Query query = session.createQuery (stringQuery);
6. query.setParameter ("id", id);
7. query.executeUpdate();
8. session.close();
9. } catch (HibernateException e) {
10. e.printStackTrace();

```

```
11. }  
12. }
```

В примере прописывается строка HQL-запроса (Hibernate Query Language — язык запросов Hibernate), и на его основе формируется сам запрос, в который мы передаем параметры через метод `setParameter`. Метод `executeUpdate` непосредственно выполняет запрос, после его выполнения в БД будет произведено удаление объекта.

### 2.4.3 Изменение объекта в БД

Изменение объекта ничем не отличается от метода создания объекта, только в месте `save` будет использоваться метод `update`:

```
1. session.beginTransaction ();  
2. session.update (o);  
3. session.getTransaction ().commit ();
```

### 2.4.4 Чтение из БД

Есть два основных способа чтения данных из БД — это чтение списком и чтение по `id`. Для чтения из базы данных используется специальный вид запросов под названием `Criteria`. Пример вызова списка без фильтрации по параметрам:

```
List<People> peoples = session.createCriteria(People.class).list();
```

То есть нужно только указать сущность, из которой мы хотим извлечь данные и вызвать метод `list`, который вернет список всех объектов, находящихся в данной таблице.

Если же нам нужен поиск по идентификатору, то нужно использовать некоторую фильтрацию:

```
1. People people = (People) session.createCriteria (People.class)  
2. .add (Restrictions.eq ("id", id)).uniqueResult ();
```

Как и в прошлый раз, мы указываем, к какой таблице обращаемся. Далее используется метод `add`, в который прописываются условия отбора, это сделать нам позволяет класс `Restrictions`. Метод же `eq` означает, что будет осуществляться сравнение с объектом, который мы в него поместим. Метод `uniqueResult` позволяет

вернуть результат объектом, но у него есть и недостатки: если объект не будет найден или по каким-то причинам объектов будет больше одного, произойдет ошибка. Для обхода ошибки рекомендуется прописывать немного другой код:

```
1. People people = null;
2. List<People> peoples = session.createCriteria (People.class)
3. .add (Restrictions.eq ("id", id)).list();
4. if (peoples!=null&&peoples.size()==1)
5. {
6.     people = peoples.get(0);
7. }
```

Весь результат будет собираться в список, если не будет найдено ни одного элемента или элементов списка будет больше 1, то if не сработает, но и не произойдет ошибки.

## 2.5 Связи между таблицами

Обычно при создании проектов необходимо связывать таблицы между собой, это нужно для логической связи объектов между собой. Так как реляционная база данных представляет собой множество взаимосвязанных таблиц, то необходимо уделить отдельное внимание описанию отношений между сущностями с помощью технологии Hibernate. Всего существует три основных типа связи: **one-to-one**, **one-to-many**, **many-to-one**; существует также связь **many-to-many**, но она может быть решена как специальной аннотацией `@manytomany`, так и промежуточной таблицей.

### 2.5.1 Связь many-to-one

Разберем пример. Есть сущность `people`, которая связана с `phone`, то есть две таблицы: люди и телефоны. У одного человека может быть несколько телефонов, но не наоборот. Для этого мы создаем связь `many-to-one`, при этом в таблице `phone` появится новое поле, ссылающееся на элемент таблицы `people`, в нем будет храниться идентификатор. Класс `people` был уже описан ранее, в данном случае он остается неизменным. Класс `phone` выглядит следующим образом:

```

1. @Entity
2. @Table (name = "phones")
3. public class Phone {
4.     @Id
5.     @GeneratedValue (strategy = GenerationType.IDENTITY)
6.     @Column (name = "id")
7.     private int id;
8.     @Column (name = "number")
9.     private String number;
10.     @ManyToOne (targetEntity = People.class, fetch =
FetchType.LAZY)
11.     @JoinColumn (name = "id_poeple")
12.     private People people;
13.     ... (описываем геттеры и сеттеры)
14. }

```

Аннотация `@ManyToOne` позволяет создать связь между двумя таблицами, в базе данных данная связь будет выглядеть, как показано на рисунке 2.3. Также при использовании связи `many-to-one` автоматически создается вторичный ключ.

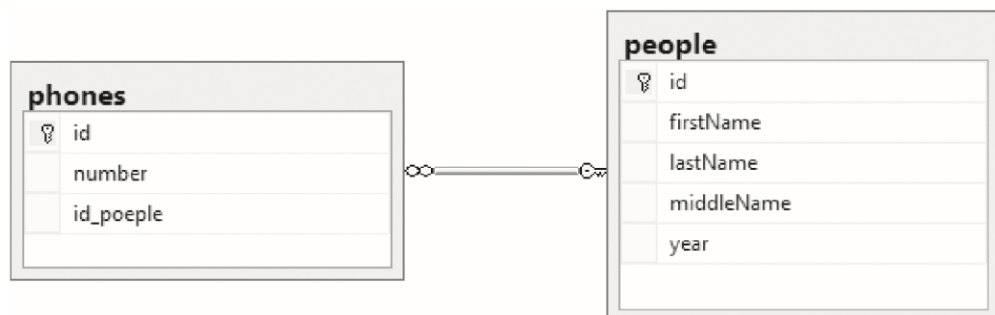


Рисунок 2.3 – Связь таблиц `many-to-one`

Свойство `targetEntity` указывает, с какой сущностью будет происходить соединение по внешнему ключу.

Свойство `fetch` (очень важное свойство) может иметь два состояния: `LAZY` или `EAGER` — по умолчанию это `EAGER`. В случае если параметром будет выступать `EAGER`, связанный элемент будет сразу же подгружаться автоматически. `LAZY` — пока не обратятся к элементу `people`, данные не будут загружены.

Ниже показано, как будут формироваться запросы в зависимости от установленного свойства fetch.

При запросе к БД:

```
1. Phone phone = (Phone) session.createCriteria (Phone.class).add
  (Restrictions.eq ("id", 1)).uniqueResult ();
2. People people = phone.getPeople ();
```

Простой запрос на получение элемента таблицы *телефон* и связанного с ним человека. fetch = FetchType.LAZY

При выполнении первой строчки будет сформирован запрос на получение данных только о таблице Phone, и только когда идет обращение к связанному объекту People, формируется новый запрос:

```
1. Hibernate: select this_.id as id1_1_0_, this_.number as
number2_1_0_, this_.id_poeple as id_poepl3_1_0_ from phones this_ where
this_.id=?
2. Hibernate: select people0_.id as id1_0_0_, people0_.firstName as
firstNam2_0_0_, people0_.lastName as lastName3_0_0_, people0_.middleName as
middleNa4_0_0_, people0_.year as year5_0_0_ from people people0_ where
people0_.id=?
```

При этом важно понимать, что если сессия будет закрыта, а после этого будет обращение к объекту, который еще не подгружался, то произойдет ошибка. Для инициализации можно применить следующий код:

```
Hibernate.initialize (phone.getPeople()); fetch = FetchType.EAGER
```

При выполнении первой строчки кода будет формироваться один большой запрос, который автоматически будет подгружать связанного с данным телефоном человека.

Пример сформированного запроса:

```
Hibernate: select this_.id as id1_1_1_, this_.number as number2_1_1_,
this_.id_poeple as id_poepl3_1_1_, people2_.id as id1_0_0_, people2_. firstName
as firstNam2_0_0_, people2_.lastName as lastName3_0_0_, people2_.middleName as
middleNa4_0_0_, people2_.year as year5_0_0_ from phones this_ left outer join
people people2_ on this_.id_ poeple=people2_.id where this_.id=?
```

Аннотация `@JoinColumn` в данном случае является не обязательной, она просто содержит параметр, в котором можно указать название колонки.

### 2.5.2 Связь **one-to-many**

Часто требуется делать и обратную связь, то есть, имея объект `people`, получать список связанных с ним телефонов.

```
1. @Entity
2. @Table (name = "people")
3. public class People {
4. ...
5. @OneToMany (targetEntity = Phone.class, mappedBy = "people",
   fetch = FetchType.LAZY)
6. private List<Phone> phones;
7. ...
8. }
```

Аннотация `@OneToMany` позволяет осуществить эту мнимую связь, которая обеспечивается `hibernate`, но не БД. Параметр `targetEntity` указывает, с какой сущностью идет связь. В параметре `mappedBy` прописывается имя связующего поля с аннотацией `@ManyToOne`, то есть его имя `people`. Параметр `fetch` был ранее описан, в данном случае он играет ту же самую функцию.

### **3 Порядок выполнения работы**

#### **3.1 Постановка задачи**

Задача на практическую работу следующая:

- 1) разобраться с ORM-технологией и Hibernate библиотекой, а также с JDBC концепцией;
- 2) опираясь на наработки предыдущей практической работы, реализовать подключение к базе данных с помощью библиотек JDBC (для каждой базы данных своя JDBC-библиотека), дальнейшее взаимодействие с базой данных должно осуществляться с помощью Hibernate. (Определение используемой базы данных на выбор студента, но желательно PostgreSQL):
  - а) необходимо создать отдельный класс, который будет отвечать за описание подключения к базе данных с помощью технологий Hibernate;
  - б) у данного класса должен быть описан метод, возвращающий экземпляр класса Session (библиотеки Hibernate), то есть уже подключенную сессию к базе данных;
- 3) создание таблицы в базе данных:
  - а) структура таблицы берется из предыдущей практической работы;
  - б) создать класс, соответствующий структуре таблицы, с элементами аннотаций для взаимодействия с базой данных через библиотеку Hibernate;
- 4) взаимодействие с таблицей необходимо осуществить на веб-странице, которая была создана ранее, и должна позволить выполнить следующие функции: удаления, обновления, добавления и отображения строк.