

Hibernate:

Создание ORM модели БД

Дорожная карта

Для тестирования модели необходимо создать в главном файле проекта подключение к БД и поработать с данными.

Цели

Создать приложение, работающее с БД через **Hibernate** ORM:

1. Исходные данные
2. Создание и подготовка проекта
3. Конфигурирование Hibernate
4. Создание соединения
5. Генерация классов сущностей
6. Корректировка классов сущностей
7. Тестирование подключения к БД

Исходные данные

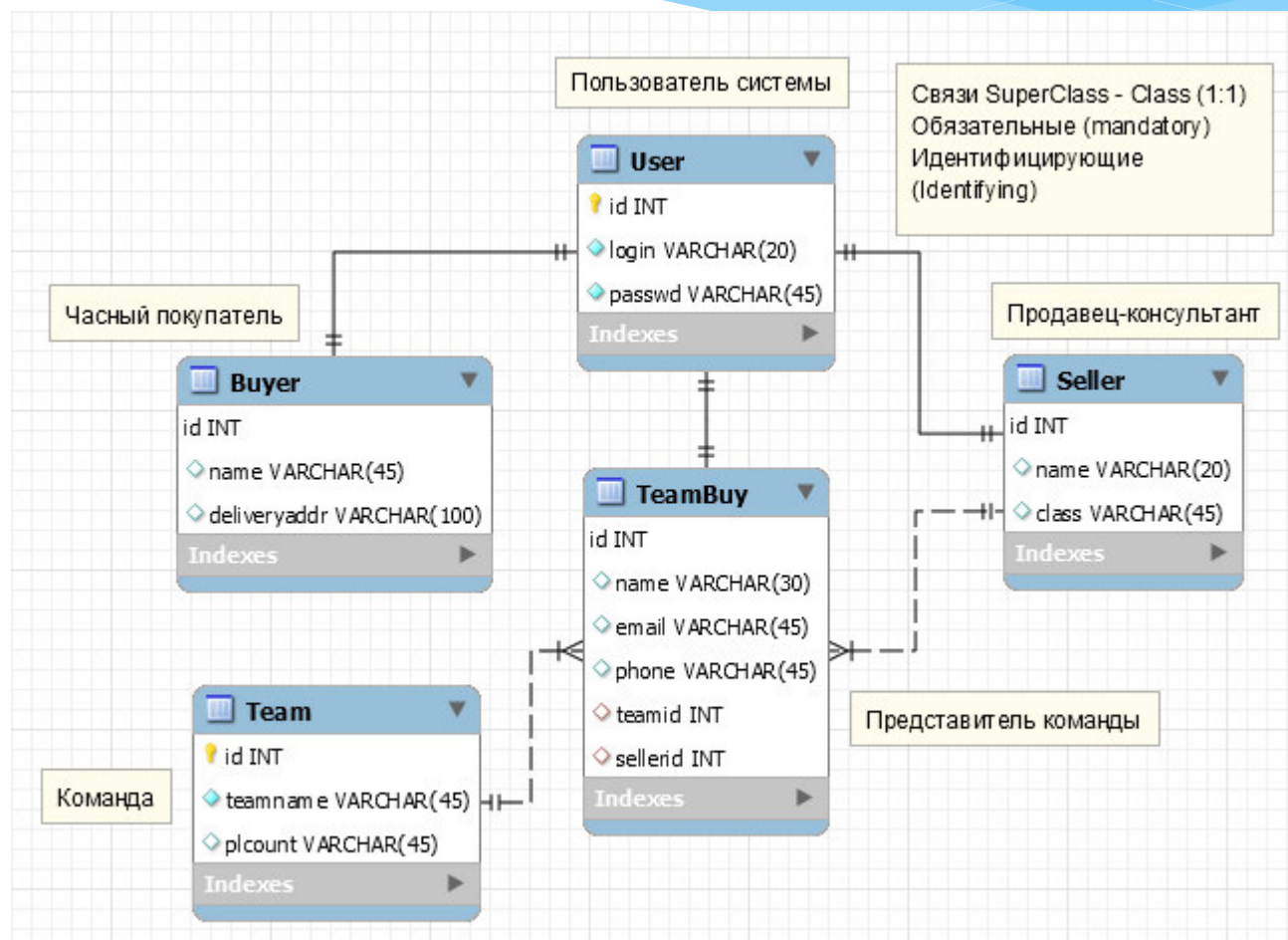
У нас имеется БД-электронный справочник хранящая данные о покупателях, корпоративных клиентах и продавцах-консультантах некой торговой фирмы (OBL).

В БД присутствуют следующие **сущности**:

1. **User** – любой пользователь электронного справочника
2. **Buyer** – покупатель, частное лицо
3. **Seller** – продавец-консультант торговой фирмы
4. **TeamBuy** – корпоративный клиент, представляющий команду
5. **Team** – команда, имеющая своего представителя

Исходные данные

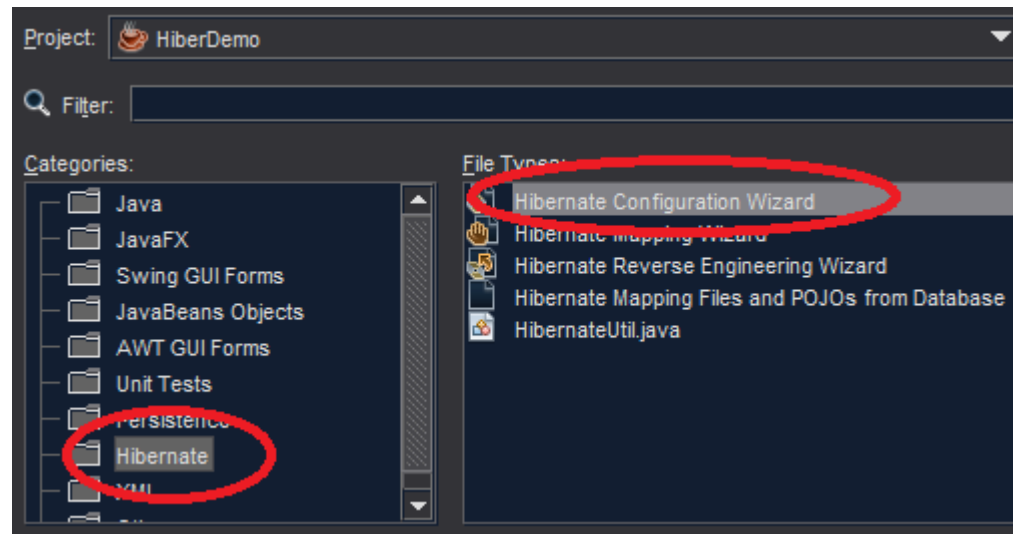
ER-диаграмма БД в физическом представлении в **MySQL**



Создание проекта

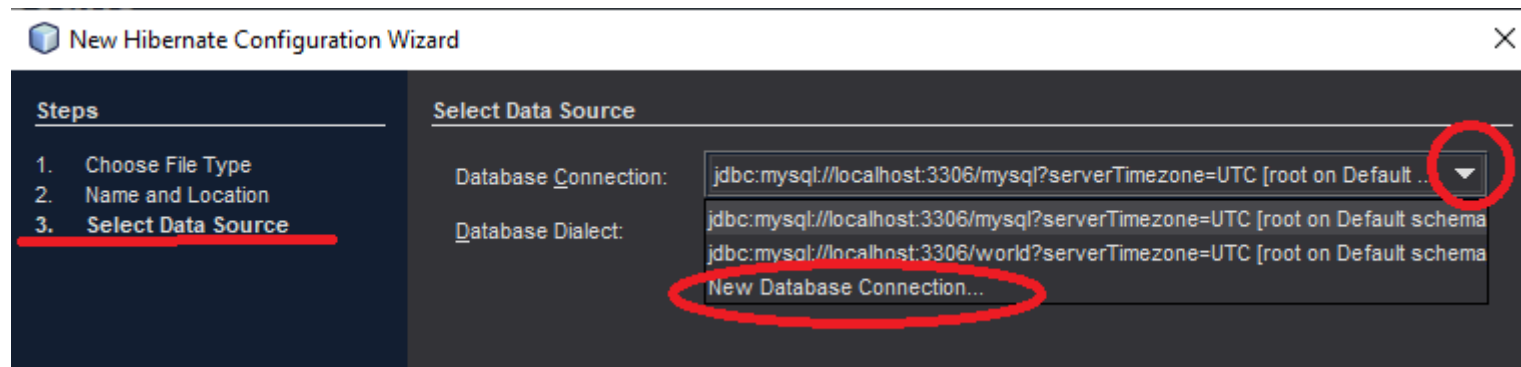
Приступаем к конфигурированию **Hibernate**.

1. Создаем новый проект Java в NetBeans **File->New Project** (Файл -> Новый проект)
2. Входим в меню и выбираем **File->New File** (Файл->Новый файл)
3. Добавляем **Hibernate Configuration Wizzard** из категории **Hibernate**



Конфигурирование

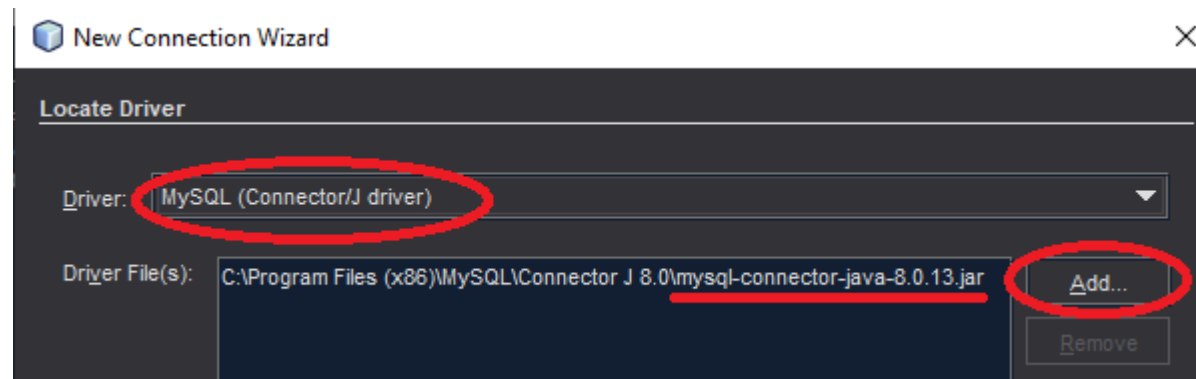
1. На следующем слайде **Name and Location** оставляем все как есть и переходим к слайду **Select Data Source**
2. В окне **Database Connection** выбираем **New Database Connection** (Новое соединение с БД)



Если изменить папку файла `hibernate.cfg` во втором окне на отличную от `src` возможны проблемы с дальнейшей конфигурацией

Создание соединения

1. В появившемся окне в списке **Driver** выбираем **MySQL (Connector/J driver)**
2. Ниже (в случае отсутствия) добавляем (**Add**) **MySQL** коннектор из папки проекта, который мы добавляли ранее



Местоположение библиотеки не принципиально, так как JDBC ищет стандартный класс коннектора (**com.mysql.jdbc.Driver**)

Однако, если версии библиотек не совпадут, но при переносе проекта в другую систему возможные сбои в работе с сервером БД

Подключение к БД

После нажатия **Next**, появиться окно детальной настройки соединения, в котором необходимо детализировать подключение к конкретной БД.

На картинке показано, какие именно параметры должны быть заданы для успешного соединения с БД (**TestConnection**)

The screenshot shows a 'Customize Connection' dialog box with the following fields and controls:

- Driver Name:** A dropdown menu set to 'MySQL (Connector/J driver)'.
- Host:** A text field containing 'Server Host'.
- Port:** A text field containing '3306'.
- Database:** A text field containing 'DB_Name'.
- User Name:** A text field containing 'UserName'.
- Password:** A text field that is currently empty and circled in red.
- ☐ **Remember password**
- Buttons:** 'Connection Properties' and 'Test Connection'.
- JDBC URL:** A text field containing 'jdbc:mysql://Server Host:3306/DB_Name?serverTimezone=UTC', with the entire string underlined in red.
- Navigation Buttons:** '< Back', 'Next >', 'Finish', 'Cancel', and 'Help'.

Подключение к БД

1. **Host** – имя или IP адресс сервера БД. Если сервер находится в той же системе, что и программа, используется **localhost** (127.0.0.1)
2. **Database** – имя БД на сервере
3. **User Name/Password** – логин и пароль пользователя, имеющего права на доступ к БД
4. В строке **JDBC URL** после имени БД необходимо указать **serverTimezone=UTC**. Это необходимо для синхронизации времени в запросах установки соединений **SSH/SSL**.
5. После нажатия **Test Connection**, если в все прошло удачно нажимаем **Finish**

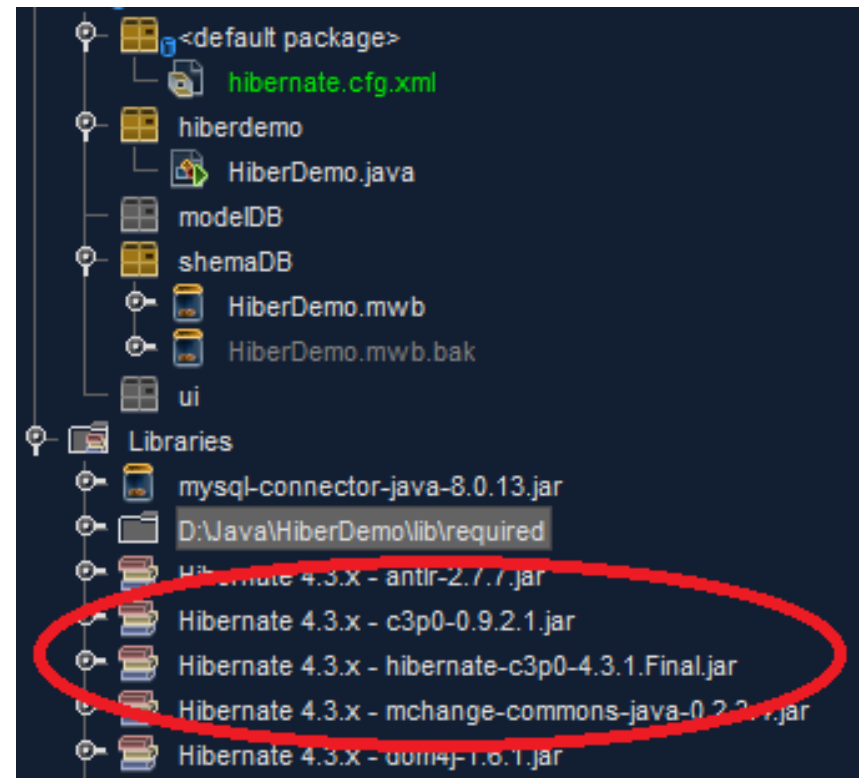
Если не указан или неверно указан один из пунктов, то будет сообщение об ошибке и кнопку Finish нажать будет невозможно! Java чувствителен к регистру символов.

Настройка библиотек

После того, как конфигурация создана, в проект будет добавлен файл **hibernate.cfg.xml**. Он находится в **<default package>** (контейнер по умолчанию).

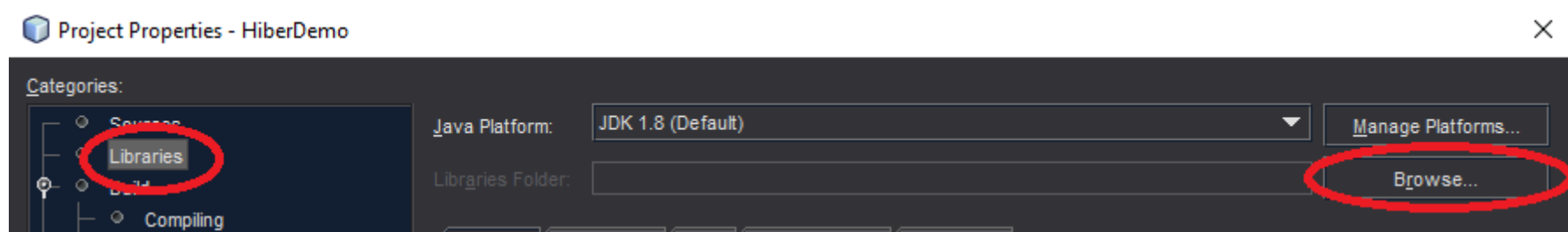
Кроме того добавятся ряд библиотек **Hibernate** предыдущих версий.

Они содержат **общие классы и интерфейсы** сущностей и прочих объектов, реализация которых изменяется и оптимизируется от версии к версии.



Настройка библиотек

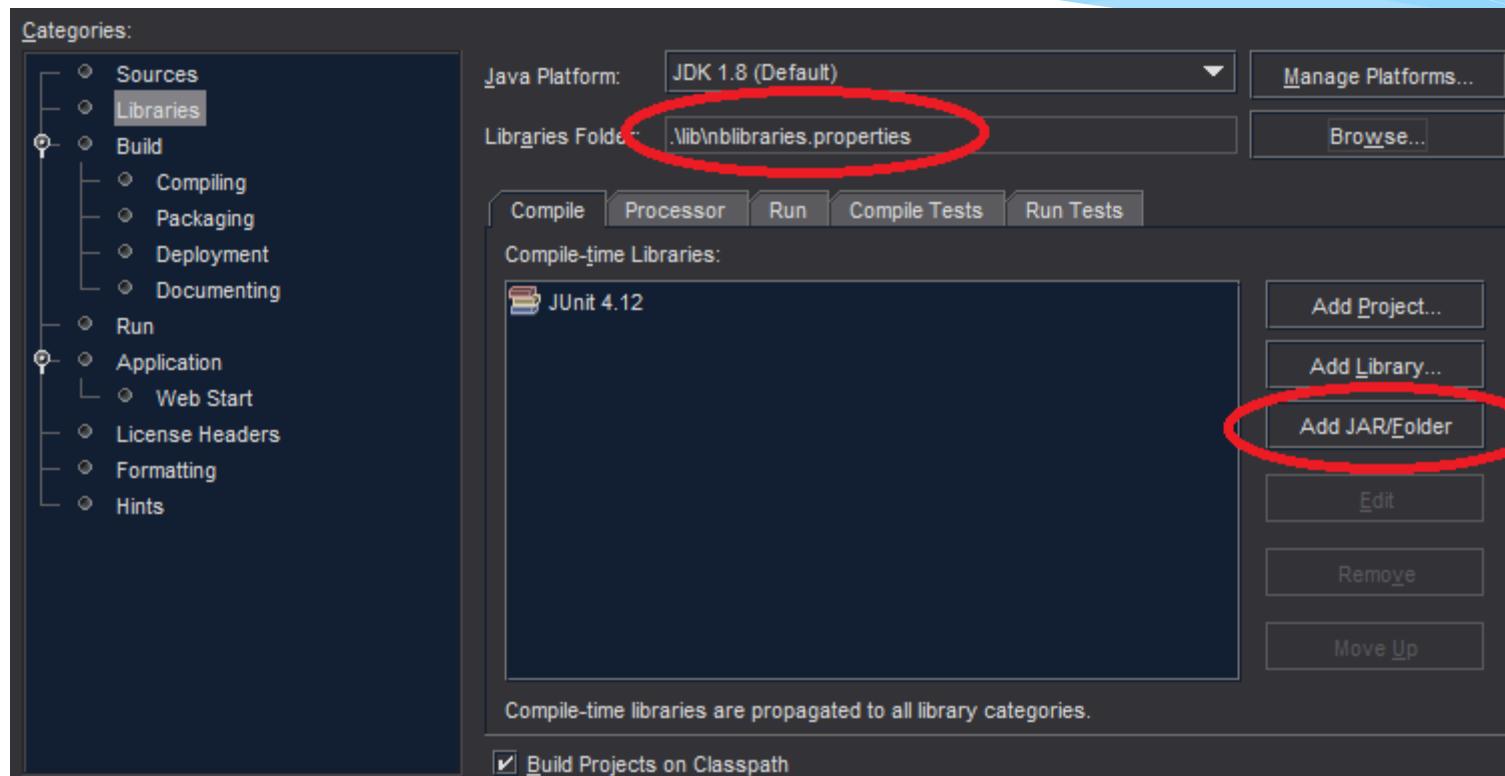
1. Переходим на папку **Libraries** (Библиотеки) и нажав правую клавишу мыши выбираем **Properties**.
2. В появившемся окне нажимаем кнопку **Browse** (обзор)
3. В новом окне **Next** (Далее) и **Finish** (Завершить). В папке проекта появиться папка **.lib** и библиотеки будут переноситься с проектом.



Если этого не сделать, при переносе проекта на другие системы возможна потеря его работоспособности

Настройка библиотек

В том же окне с помощью кнопки **Add JAR/Folder** приступаем к добавлению библиотек в проект



Обратите внимание на путь к библиотекам – он изменился

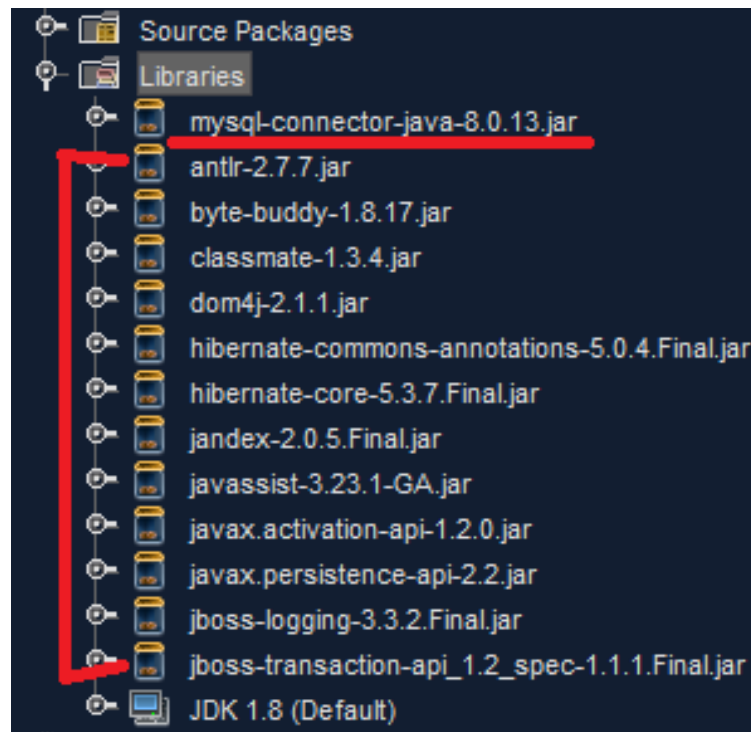
Настройка библиотек

1. В появившемся окне находим актуальную версию **MySQL Connector J**. Обычно он находится в папке установки **MySQL** в подпапке **Connector J** и называется **mysql-connector-java-x.y.z.jar** (где x.y.z номер версии)
2. Таким же образом выполняем подключение библиотек **Hibernate**. Обычно они находятся в папке установки фреймворка, в подпапке **lib/required**. Для того, чтобы добавить все файлы из папки списком, входим в нее и выделяем все файлы (Ctrl+A).

Если Connector был указан на этапе подключения к БД, он автоматически будет помещен в список библиотек

Подготовка проекта

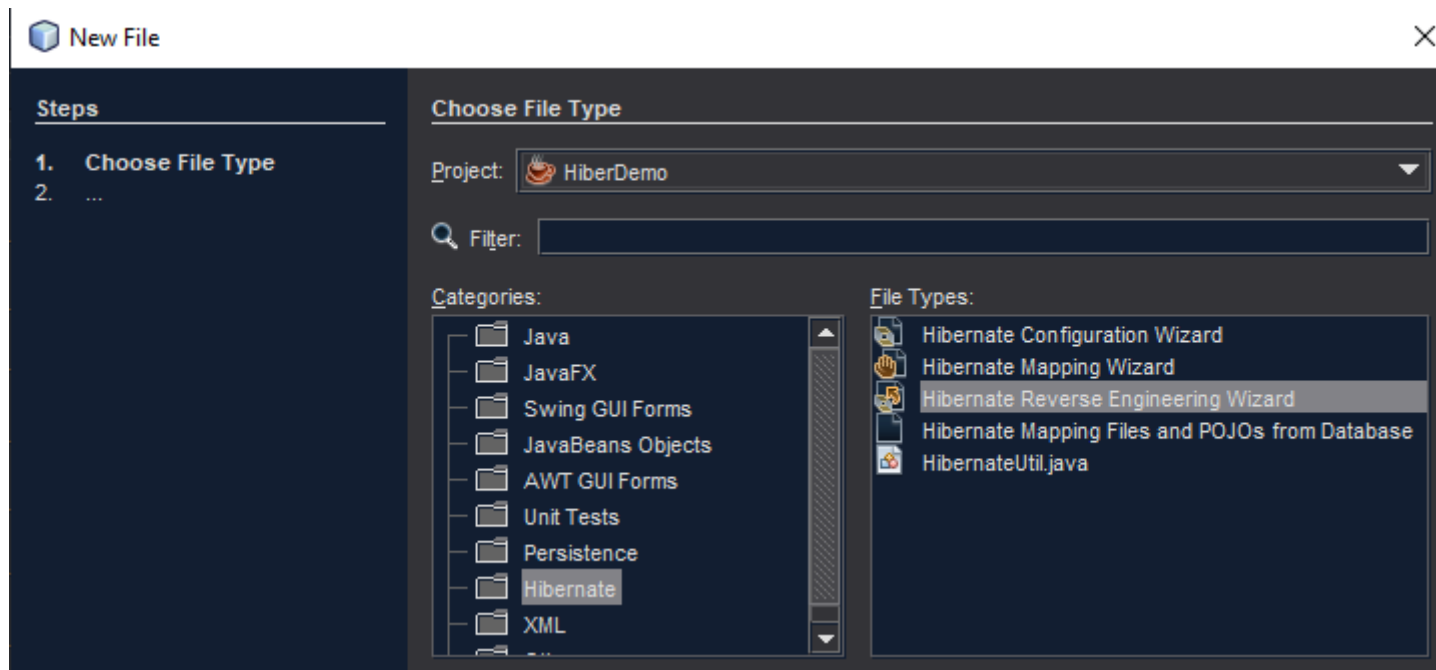
1. В результате в библиотеках проекта должны появиться коннектор **MySQL** и библиотеки **Hibernate**.



Обратите внимание что появилось в библиотеках проекта

Создание ERD в Hibernate

1. Для создания **ORM** (связи класс-сущность) Hibernate должен исследовать структуру БД. Для этого добавляем
2. Добавляем новый файл и выбираем **Hibernate Reverse Engineering Wizard** из категории **Hibernate**



Создание ERD в Hibernate

Далее предлагается выбрать имя и местоположение файла со схемой БД. Их необходимо запомнить, т.к. в случае изменения ER модели БД классы **Hibernate** так же **надо будет перестроить**. Для этого, в свою очередь необходимо будет убрать старое описание классов.

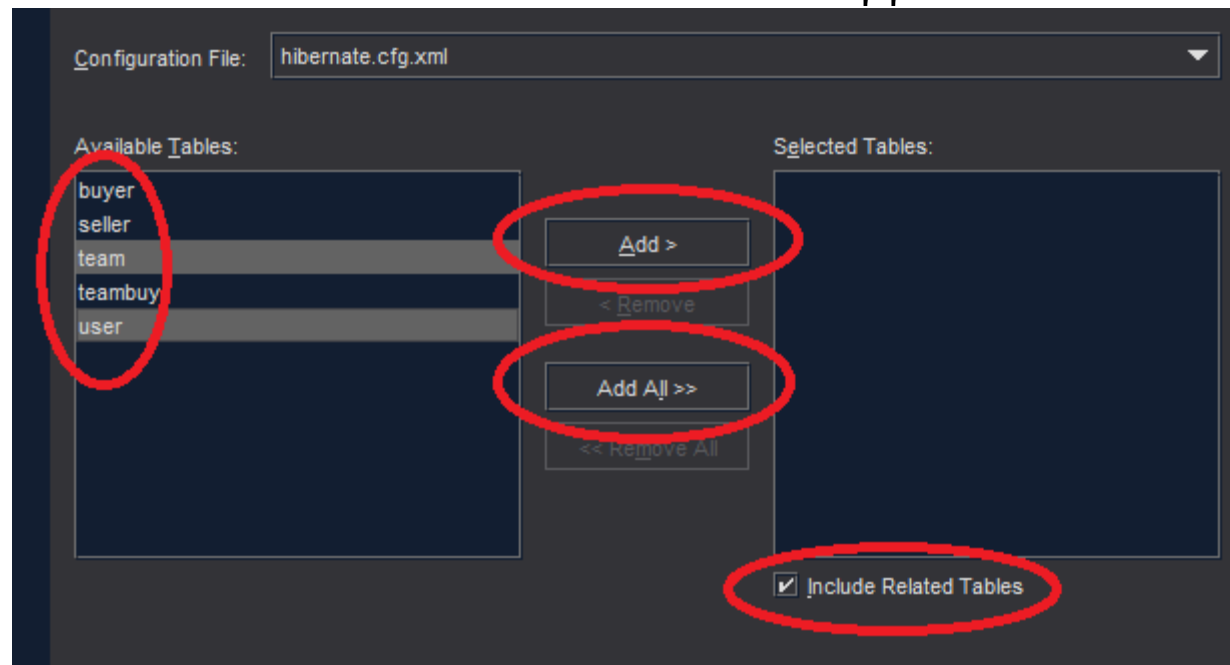
The screenshot shows a dark-themed dialog box with two main sections. On the left, under the heading 'Steps', there is a list: 1. Choose File Type, 2. Name and Location (which is currently selected), and 3. Database Tables. On the right, under the heading 'Name and Location', there are several input fields: 'File Name:' with the value 'hibernate.reveng', 'Project:' with the value 'HiberDemo', 'Folder:' with the value 'src\schemaDB' and a 'Browse...' button next to it, and 'Created File:' with the value 'D:\Java\HiberDemo\src\schemaDB\hibernate.reveng.xml'.

Рекомендуется хранить ER диаграмму сервера БД и сгенерированную в одном контейнере внутри проекта

Создание ERD в Hibernate

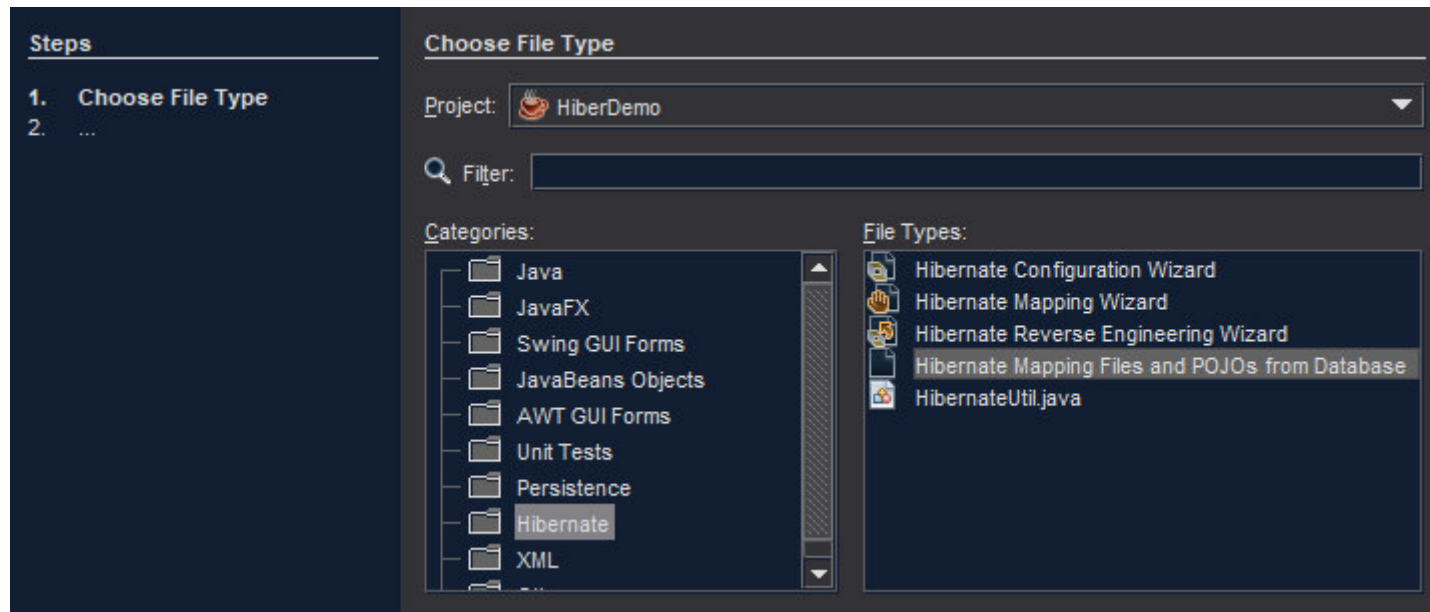
Далее необходимо выбрать сущности, которые будут представлены в модели (возможно выбрать несколько) и нажать **Add** или **Add All** (для добавления всех сущностей).

Флажок **Include Related Tables** автоматически добавляет родительские (таблицы суперклассов) для выбранной. После нажатия **Finish** сгенерируется файл диаграммы



Генерация классов сущностей

1. Для создания классов сущностей Hibernate используется ERD из предыдущего раздела (**hibernate.reveng.xml**)
2. Добавляем новый файл и выбираем **Hibernate Mapping Files and POJOs from Database** из категории **Hibernate**.



Генерация классов сущностей

1. На следующем шаге необходимо указать модель, из которой генератор будет брать сущности и атрибуты (**hibernate.reveng.xml**)
2. В **General Settings** (Глобальные Опции) установить флажки **JDK 5 Language Features** и **EJB 3 Annotation**.
3. В **Code Generation Settings** (Параметры Генерации Кода) установить флаг **Domain Code (.java)** и убрать другую.
4. В **Package** выбрать или вписать имя нового контейнера для вновь сгенерированных файлов.

Если поместить все сущности в существующий контейнер, то структура проекта будет ввергнута в хаос (муаахахаха)

Генерация классов сущностей

New Hibernate Mapping Files and POJOs from Database

Steps

1. Choose File Type
2. Generation of Code

Generation of Code

Hibernate Configuration File: hibernate.cfg.xml

Hibernate Reverse Engineering File: hibernate.reveng.xml

General Settings:

- ☒ JDK 5 Language Features
- ☒ EJB 3 Annotations

Code Generation Settings:



- ☒ Domain Code (.java)
- ☐ Hibernate XML Mappings (.hbm.xml)

Project: HiberDemo

Location: Source Packages

Package: modelDB

< Back Next > Finish Cancel Help

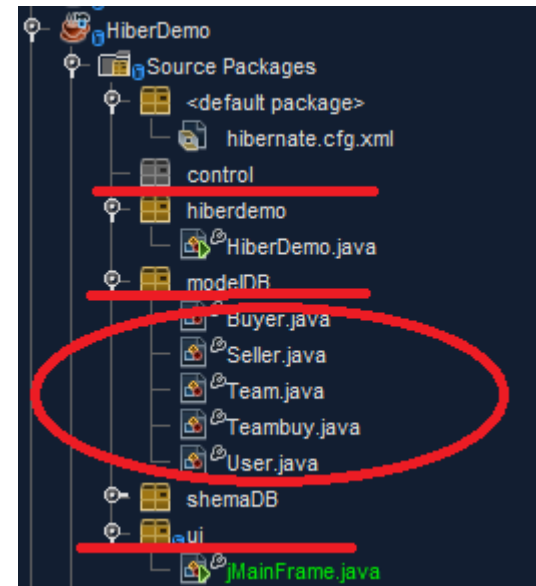


Генерация: результат

В результате в проекте в контейнере **modelDB** появятся файлы, в которых описываются классы для каждой сущности БД.

Структура проекта:

1. modelDB – модель БД в виде object-relation mapping
2. ui – user interface (интерфейс пользователя)
3. control – бизнес-логика и управление



Обратите внимание на структуру проекта

Генерация: результат

Содержимое сгенерированных файлов отражает структуру соответствующих сущностей, поля сущностей – это свойства классов.

Из свойств класса видно, что **Buyer** является подклассом суперкласса **User**. Он содержит ссылку на него как неявное свойство.

Наследование можно увидеть и в конструкторе.

```
@Entity
@Table(name="buyer", catalog="hiberdemo")
public class Buyer implements java.io.Serializable {
    private int id;
    private User user;
    private String name;
    private String deliveryaddr;

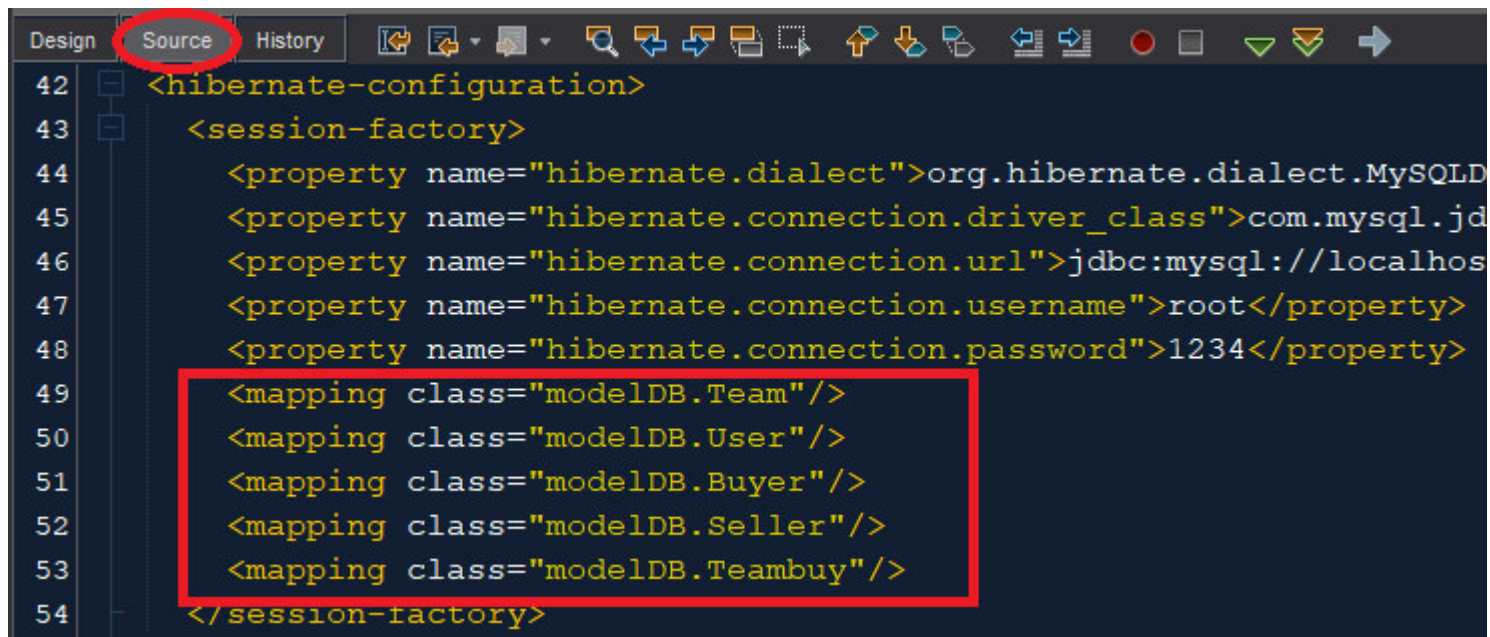
    public Buyer() { }

    public Buyer(User user) {
        this.user = user;
    }
}
```

Обратите внимание на аннотации @Entity и @Table

Генерация: результат

Так же изменения произошли в файле **hibernate.cfg.xml**. Помимо свойств соединения туда добавились классы отображения (**mapping**) для сущностей.



```
42 <hibernate-configuration>
43   <session-factory>
44     <property name="hibernate.dialect">org.hibernate.dialect.MySQLD
45     <property name="hibernate.connection.driver_class">com.mysql.jd
46     <property name="hibernate.connection.url">jdbc:mysql://localhos
47     <property name="hibernate.connection.username">root</property>
48     <property name="hibernate.connection.password">1234</property>
49     <mapping class="modelDB.Team"/>
50     <mapping class="modelDB.User"/>
51     <mapping class="modelDB.Buyer"/>
52     <mapping class="modelDB.Seller"/>
53     <mapping class="modelDB.Teambuy"/>
54   </session-factory>
```

Для просмотра содержимого файла настройки необходимо переключить режим с **Design** на **Source**

Перегенерация классов сущностей

В случае **если в структуре БД произведены изменения**, для корректной работы приложения они должны быть внесены и в ORM.

Для генерации сущностей потерпевших изменения **необходимо удалить** их файлы классов (.java) из проекта, а так же строки с **mapping** из файла **hibernate.cfg.xml**.

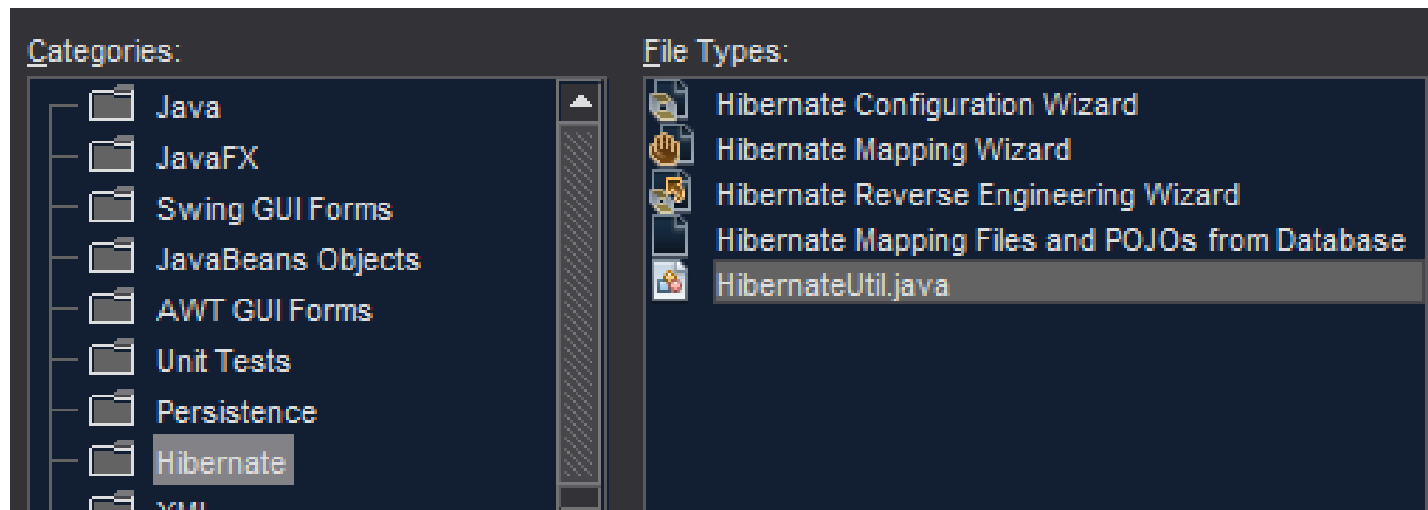
```
<mapping class="modelDB.Team"/>
<mapping class="modelDB.User"/>
<mapping class="modelDB.Buyer"/>
<mapping class="modelDB.Seller"/>
<mapping class="modelDB.Teambuy"/>
```

Если были внесены изменения в структуру ERD (связи), пересоздание ORM модели необходимо начинать с пересоздания ERD в Hibernate

Подключение SessionFactory

Центральный объект **Hibernate**, который отвечает за все соединения называется **SessionFactory**. Он создается один раз и содержит в себе пул (**pool**) подключений к БД.

Для работы с SessionFactory необходимо подключить модуль **HibernateUtil.java** из категории **Hibernate**.



Подключение SessionFactory

Далее надо задать имя центрального класса Hibernate в данном проекте и контейнер в котором он будет находиться.

The screenshot shows an IDE's 'New Class' dialog with two main sections: 'Steps' and 'Name and Location'. The 'Steps' section on the left lists '1. Choose File Type' and '2. Name and Location'. The 'Name and Location' section on the right contains several fields: 'Class Name' with the value 'HiberDemoHibernateUtil', 'Project' with 'HiberDemo', 'Location' with 'Source Packages', 'Package' with 'control', and 'Created File' with 'D:\Java\HiberDemo\src\control\HiberDemoHibernateUtil.java'. Red circles highlight the 'Class Name' and 'Package' fields.

Steps	Name and Location
1. Choose File Type	Class Name: HiberDemoHibernateUtil
2. Name and Location	Project: HiberDemo
	Location: Source Packages
	Package: control
	Created File: D:\Java\HiberDemo\src\control\HiberDemoHibernateUtil.java

Нежелательно помещать `HibernateUtil` в контейнеры с классами сущностей и UI, т.к. при переработке проекта они могут быть отключены или полностью изменены

Иерархия классов: суперкласс

Как видно из исходной ER диаграммы, классы **Buyer**, **Teambuy** и **Seller** являются подклассами суперкласса **User**.

Однако, если посмотреть автоматически сгенерированный код видно, что каждый класс не является частью какой-либо иерархии.

Исходя из ERD, класс **user** должен быть абстрактным а связи реализованы через механизм наследования.

Файл **User.java**

```
public class User implements java.io.Serializable {...}
```

Файл **Buyer.java**

```
public class Buyer implements java.io.Serializable {...}
```

Иерархия классов: суперкласс

Внесем изменения в суперкласс **User**. Для определения наследования воспользуемся аннотацией **@Inheritance**, в режиме **JOINED**. При наследовании ORM будет автоматически склеивать суперкласс и наследника с помощью **select ... join**.

```
@Inheritance(strategy = InheritanceType.JOINED)  
abstract public class User implements java.io.Serializable {
```

Удалим строки

```
private Buyer buyer;  
private Seller seller;  
private Teambuy teambuy;
```

так как наследование теперь заменяет необходимость в таких приватных коллекциях.

Иерархия классов: суперкласс

В описание столбца первичного ключа (аннотация @Id) добавить аннотацию

`@GeneratedValue(strategy = GenerationType.IDENTITY)`

Она указывает, что поле является автоинкрементным (значения для первичного ключа генерируются автоматически)

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)

@Column(name="id", unique=true, nullable=false)
public Integer getId() {
    return this.id;
}
```

Иерархия классов: подкласс

Перейдем к подклассу **Buyer**. Нам необходимо включить его в иерархию как наследника **User** и удалить свойства, которые сейчас реализуются через наследование.

```
public class Buyer extends User implements java.io.Serializable {
```

Удаляем свойства, реализующие планарную связь и поле **id** (оно теперь наследуется от **User**)

```
private int id;
```

```
private User user;
```

Вместе с ним удаляем конструкторы и методы вызывающие ошибки. Методы удаляются вместе с аннотациями

```
@Column(name="id", unique=true, nullable=false)
```

```
@OneToOne(fetch=FetchType.LAZY)@PrimaryKeyJoinColumn
```

```
@GenericGenerator(name="generator", strategy="foreign", ...
```

Иерархия классов: подкласс

Аналогичным образом модифицируются классы **Seller** и **Teambuy**:

1. Дописываем **extends User**
2. Удаляем поля **User user** и **int id**;
3. Удаляем все конструкторы и методы связанные с этими полями вместе с аннотациями
4. Удаляем **@GenericGenerator** относящийся к связи с суперклассом **User**

Аннотации **@ManyToOne** и **@OneToMany** для связей **Seller-Teamby** и **TeamBy-Team** остаются неизменными

Все что связано с Teamby в Seller, а так же с Team и Seller в Teamby остается как есть – это планарная связь между двумя сущностями.

Тестирование

Тестирование подключения и взаимодействия с БД проводится поэтапно:

1. Подключение к БД, открытие закрытие сессии и транзакции
2. Добавление записей в таблицы
3. Проведение выборки
4. Удаление данных

Код для манипуляций с данными вставляем между началом и завершением транзакции

Подключение к БД

В главном модуле проекта в функции main()

```
// TODO code application logic here
System.out.println("Starting ..");
Session sess=HibernateUtil.          // Открываем сессию
                                getSessionFactory().openSession();

// Вот тут будет блок работы с данными

System.out.print("Closing .. ");
sess.close();                      // Завершаем работу
HibernateUtil.getSessionFactory().close();
System.out.println(" done");
```

**Переходить к дальнейшим действиям возможно только
если подключение сработало без Exception**

Добавление данных

В главном модуле проекта в **блоке работы с данными**

// Создаем экземпляр покупателя

```
System.out.print("Creating buyer .. ");
```

```
Buyer newbuy=new Buyer();
```

```
newbuy.setLogin("Karlson");
```

```
newbuy.setPasswd("123");
```

```
newbuy.setName("Карлсон");
```

```
newbuy.
```

```
    setDeliveryaddr("Стокгольм, Вулканусгатан д.12, крыша");
```

```
System.out.println("Ok");
```

В import надо добавить все классы сущностей:

```
import model.*;
```

Добавление данных

Открываем транзакцию и пытаемся сохранить данные в БД

// Сохраняем покупателя в БД

```
System.out.print("Storing buyer .. ");  
try {  
    Transaction trs=sess.beginTransaction();  
    sess.save(newbuy);  
    trs.commit();  
    System.out.println(" Ok");  
} catch (HibernateException ex) {  
    System.out.println(" Error: "+ex.getMessage()); }  

```

Что происходит если выполнить программу два раза?

Что произойдет если убрать транзакцию?

Что происходит с таблицей User?

Добавление данных

Оптимизируем добавление данных на примере таблицы **Seller**. Для этого добавим в класс Seller еще один конструктор, кроме конструктора по умолчанию:

```
public Seller(String login,String passwd,  
              String name, String class_) {  
    this.setLogin(login);  
    this.setPasswd(passwd);  
    this.name=name;  
    this.class_=class_;  
}
```

Почему login и passwd устанавливаются через set'теры?

Добавление данных

Создадим массив продавцов

```
System.out.println("Creating seller list .. ");  
ArrayList<Seller> slr=new ArrayList<>();  
// Seller: login, passwd, name, class  
slr.add(new Seller("Luntik","001",  
                    "Лунтик", "Бобрый")) ;  
slr.add(new Seller("Mila","010",  
                    "Пчелка Мила", "Чесный")) ;  
slr.add(new Seller("Vupsen","011",  
                    "Г. Вупсень", "Разводила")) ;  
slr.add(new Seller("Pupsen","100",  
                    "Г. Пупсень", "Хапуга")) ;
```

Добавление данных

Сохраняем данные (обратите внимание на транзакцию)

```
System.out.print("Saving seller list .. ");  
try {  
    sess.beginTransaction();  
    for(Seller sl : slr) sess.save(sl);  
    sess.getTransaction().commit();  
    System.out.println(" ok");  
} catch (HibernateException ex) {  
    System.out.println(" Error: "+  
                        ex.getMessage()); } }
```

Обратите внимание на работу с транзакцией

Выборка данных

Попробуем вывести содержимое таблицы **Seller**

// Создаем запрос HQL

```
Query qry=sess.createQuery("FROM Seller");
```

// Выполняем запрос и получаем список экземпляров

```
ArrayList<Seller> qsl=(ArrayList<Seller>) qry.list();
```

// Вывод

```
for(Seller sl : qsl)
```

```
System.out.println(sl.getLogin()+" , "+
```

```
sl.getName()+" (" +sl.getClass_()+" )");
```

Обратите внимание на запрос. Благодаря аннотации @Table класс Seller сопоставляется с сущностью БД

Выборка данных: Авторизация

Сделаем вход по логину и паролю. Для этого:

1. Запросить у пользователя логин/пароль
2. Сделать запрос к БД указав логин и пароль
3. Проверить есть ли результат и к какому классу относиться пользователь

// 1. Считываем данные пользователя с клавиатуры

```
String login,passwd;  
Scanner in=new Scanner(System.in);  
System.out.print("Login:"); login=in.nextLine();  
System.out.print("Password:"); passwd=in.nextLine();
```

Выборка данных: Авторизация

2. Запрос к БД с помощью объекта **Criteria**. Для работы с этим объектом необходимо:

1. Задать класс по которому идет отбор (**User.class**)
2. Установить ограничения (**Restrictions**) на совпадение значений (**eq**)
3. Получить список записей, удовлетворяющих критерию

// 2. Проверка логина/пароля

```
Criteria cqr=sess.createCriteria(User.class) ;  
cqr.add(Restrictions.eq("login", login)) ;  
cqr.add(Restrictions.eq("passwd", passwd)) ;  
ArrayList<User> usr=(ArrayList<User>) cqr.list() ;
```

Выборка данных: Авторизация

Если список пуст – таких записей нет и пользователь в системе не зарегистрирован. В противном случае появится одна запись, которая будет соответствовать пользователю (одна, т.к. **Login** уникальный)

Результатом будет список из одного элемента абстрактного класса **User**. Он может хранить в себе экземпляр одного из классов-наследников: **Seller** (продавец), **Buyer** (покупатель), **Teambuy** (представитель команды).

Проверить на какой именно экземпляр указывает объект можно с помощью оператора **instanceof**, проверяющего принадлежность экземпляра классу.

Выборка данных: Авторизация

Дальнейший код осуществляет проверку принадлежности к определённому классу и получает его экземпляр

```
if (usr.size()>0) { // Список НЕ пуст – проверяем
    if (usr.get(0) instanceof Seller) {
        Seller sl=(Seller)usr.get(0);
        System.out.println("Привет, "+sl.getName()+
            " ты продавец "+sl.getClass_());
    } else if (usr.get(0) instanceof Buyer) {
        Buyer br=(Buyer)usr.get(0);
        System.out.println("Привет, "+br.getName()+
            " из "+br.getDeliveryaddr());
    } else {
```

Выборка данных: Авторизация

```
else {  
    Teambuy tb=(Teambuy)usr.get(0);  
    System.out.println("Привет, "+tb.getName()+  
        " представитель "+tb.getTeam().getTeamname());  
    System.out.println("Тебя обслужит "+  
        tb.getSeller().getName()+" который "+  
        tb.getSeller().getClass_());  
}  
} else {  
    System.out.println("Иди отседа, мая твая низнай");  
}
```