

Базы данных

Java Junior



Оглавление

Введение	3
JDBC	4
ORM	12
JPA	24
Заключение	27

Введение

Добро пожаловать на четвертое занятие! Сегодня на повестке дня работа с базами данных. Начнем, как и в предыдущий раз, с рефлексии!

На предыдущем занятии мы углубились в работу с файловой системой и изучили два вида сериализации. Простой, но мало управляемый, и более сложный, но гибкий. Эти инструменты позволяют не только сохранять информацию в файл, но и загружать уже инициализированный объект со всеми полями и связями. Такие инструменты открывают возможности для создания своего рода базы данных, особенно если требования к ней не слишком высоки.

Сегодня мы рассмотрим случаи, когда требования к базе данных более сложны, и изучим инструменты Java, предназначенные для решения подобных задач.

JDBC

Первый инструмент, который мы рассмотрим, далеко не так часто используется, а чаще и вообще не виден за более крупными Фреймворками. Однако доступ к базам данных он предоставляет довольно удобный и, главное, универсальный. У каждой системы управления базами данных (СУБД) свой интерфейс, свои особенности подключения, свои порты и так далее. В общем, написав код для доступа к PostgreSQL, например переход на MySQL, уже потребует новую реализацию. А для доступа к обеим СУБД сразу потребует третья реализацию.

JDBC (Java Database Connectivity) — это программный интерфейс, предоставляющий набор классов и методов для взаимодействия с базами данных из языка программирования Java. JDBC обеспечивает стандартизированный способ подключения к различным системам управления базами данных (СУБД), выполнения SQL-запросов, получения и обновления данных в базе. Он предоставляет абстракцию, позволяющую разработчикам писать приложения, которые могут взаимодействовать с базами данных, независимо от конкретной используемой СУБД. JDBC используется для создания портативных и эффективных приложений, работающих с данными в базах данных.

Java Database Connectivity API (JDBC) представляет собой промежуточный интерфейс между программистами и системами управления базами данных (СУБД). JDBC обеспечивает абстрактный доступ к базам данных, реализуя это с использованием драйверов, которые создают разработчики баз данных. Драйверы существуют для различных СУБД и платформ, обеспечивая универсальность. Вам нужно всего лишь найти и подключить соответствующий драйвер для нужной СУБД. Одним из ключевых преимуществ JDBC является независимость от конкретной базы данных: после написания компоненты для работы с данными можно легко перенести приложение на другую базу данных, просто указав другой драйвер. Это обеспечивает быстроту и удобство в разработке и поддержке приложений.

Но давайте уже попробуем что-нибудь написать! Однако для начала, выберем СУБД. Меня вполне устраивает MySQL, поэтому на ней и остановимся!

Шаги по установке MySQL и поиску драйвера JDBC:

1. Выберем СУБД: Перейдем на официальный сайт MySQL по ссылке <https://www.mysql.com/>. Нажмем на вкладку "DOWNLOADS". Внизу страницы находим ссылку "MySQL Community (GPL) Downloads" и перейдем по ней.

2. Загрузим MySQL: На странице загрузки выбираем необходимую версию и загрузим MySQL Installer for Windows.
3. Установим MySQL: Запускаем загруженный установщик и следуем инструкциям по установке MySQL.
4. Найдем JDBC-драйвер: В поисковике введите "mysql driver java maven". Перейдите на первую ссылку, которая обычно ведет на maven repository. На странице выбора версий драйвера выберите нужную версию и перейдите на страницу с текстом.

Теперь у вас установлена MySQL, и вы загрузили JDBC-драйвер, готовый к использованию в вашем проекте.

```
<!-- https://mvnrepository.com/artifact/com.mysql/mysql-connector-j -->
<dependency>
  <groupId>com.mysql</groupId>
  <artifactId>mysql-connector-j</artifactId>
  <version>8.1.0</version>
</dependency>
```

Это зависимость для Maven. Подключив эту зависимость к файлу pom.xml нашего проекта, мы фактически включаем драйвер!

Теперь мы готовы к работе с базой данных. Создадим новый класс Db.java и точку входа, как обычно. В теле класса добавим три финализированных поля. (Пример 1)

Пример 1

```
private static final String url = "jdbc:mysql://localhost:3306";
private static final String user = "root";
private static final String password = "root";
```

Поле "url" представляет собой путь к серверу. В нашем случае это локальный сервер, что обозначается как "localhost", а "3306" - стандартный порт MySQL. Поля "user" и "password" необходимы для установления соединения с сервером, и их значения должны соответствовать тем, которые мы ввели при установке MySQL. Наконец, перейдем к работе с базой данных! Для установления подключения к базе используем следующую строку (Пример 2).

Пример 2

```
Connection con = DriverManager.getConnection(url, user, password)
```

Давайте более детально рассмотрим его. DriverManager - это класс, содержащий набор методов для инициализации JDBC-драйвера сервера базы данных и, в конечном итоге, для подключения к нему. По сути, он является основой JDBC! Метод getConnection() позволяет установить соединение с СУБД, предоставив ей адрес, логин и пароль. Однако сам метод может сгенерировать исключение, поэтому полная форма записи выглядит следующим образом (Пример 3).

Пример 3

```
try (Connection con = DriverManager.getConnection(url, user, password)){  
  
} catch (SQLException e) {  
    throw new RuntimeException(e);  
}
```

Давайте запустим код и не увидим ничего. И это замечательно! Это означает, что подключение к серверу прошло успешно, и никаких исключений не возникло. Теперь объект `con` представляет собой своего рода источник данных, через который мы можем взаимодействовать с СУБД. Обычно базу данных создают в инструментарии СУБД, и в MySQL, конечно, это всё есть, но мы попробуем создать её с использованием SQL-запроса прямо из нашего приложения. Кстати, в MySQL база данных называется схемой, это всего лишь название и не влияет на функционал. Итак, вот наш код (Пример 4).

Пример 4

```
Statement statement = con.createStatement();
statement.execute("CREATE SCHEMA `test` ;");
```

Сначала мы создаём объект `Statement`, который используется для выполнения SQL-запросов и получения результатов их выполнения. Затем мы создаём саму базу данных (схему). При запуске кода ничего не происходит, и это хорошо. Это означает, что мы успешно подключились к серверу, и запрос выполнен. Тем не менее, теперь было бы замечательно увидеть какой-то результат. Давайте запустим его снова!

```
Exception in thread "main" java.lang.RuntimeException: java.sql.SQLException: Can't
create database 'test'; database exists
    at Db.main(Db.java:30)
Caused by: java.sql.SQLException: Can't create database 'test'; database exists
    at com.mysql.cj.jdbc.exceptions.SQLError.createSQLException(SQLError.java:130)
    at
com.mysql.cj.jdbc.exceptions.SQLExceptionsMapping.translateException(SQLExceptions
Mapping.java:122)
    at com.mysql.cj.jdbc.StatementImpl.executeInternal(StatementImpl.java:763)
    at com.mysql.cj.jdbc.StatementImpl.execute(StatementImpl.java:648)
    at Db.main(Db.java:16)
```

Появилось исключение! Он сообщает нам, что создать базу данных не удастся, так как она уже существует! Прекрасно, это означает, что схема (база данных) была успешно создана. Давайте добавим ещё одну строку и запустим.

Пример 5

```
try (Connection con = DriverManager.getConnection(url, user, password)){
    Statement statement = con.createStatement();
    statement.execute("DROP SCHEMA `test` ;");
    statement.execute("CREATE SCHEMA `test` ;");
} catch (SQLException e) {
    throw new RuntimeException(e);
}
```

Мы добавили просто удаление схемы, запустили и опять в консоли пусто. Тут всё просто мы подключились к серверу, удалили схему и создали новую. Двинемся дальше. В схему нужно добавить таблицу и поля (Пример 6).

Пример 6

```
statement.execute("CREATE TABLE `test`.`table` (\n" +  
    " `id` INT NOT NULL,\n" +  
    " `firstname` VARCHAR(45) NULL,\n" +  
    " `lastname` VARCHAR(45) NULL,\n" +  
    " PRIMARY KEY (`id`));");
```

Это всего лишь ещё один SQL-запрос. Он формирует таблицу "table" в схеме "test" с полями "id", "firstname" и "lastname". Поле "id" выступает в роли основного ключа. Выполнив запрос, вновь наступает тишина, и так хочется увидеть результат. Ладно, запустим MySQL Workbench, выберем localhost, и в новом окне откроем нашу таблицу (рисунки 7-8). Все поля на месте, и они имеют типы, указанные в нашем запросе! Отлично, давайте попробуем заполнить эти поля из программы (Пример 7).

Пример 7

```
statement.execute("INSERT INTO `test`.`table` (`id`,`firstname`,`lastname`)\n" +  
    "VALUES (1,'Иванов',Иван');");
```

Это снова обычный SQL-запрос с просьбой заполнить поля таблицы данными. После его выполнения мы снова не увидим изменений, поэтому предлагаю добавить ещё пару строк (Пример 8).

Пример 8

```
ResultSet set = statement.executeQuery("SELECT * FROM `test`.`table`");  
while (set.next()){
```



```

    System.out.println(set.getString(3) + " " + set.getString(2) + " " + set.getInt(1));
}

```

Стейтмент возвращает результат работы запроса, а ResultSet создаёт указатель на первую строку результата. Метод next() возвращает истину, только если ещё есть не обработанные строки. Геттеры позволяют получить доступ непосредственно к данным. Таким образом осуществляется доступ к данным и их перебор. Запускаем! И видим в консоли.

Иван Иванов 1

Выглядит не так увлекательно, но давайте вспомним, что мы сделали. Мы добавили зависимость Maven для подключения драйвера MySQL к проекту, установили соединение с сервером, создали схему, сформировали и заполнили поля в базе данных. Затем мы извлекли данные из таблицы, просмотрели их и вывели результат! Теперь это выглядит уже более внушительно. Давайте добавим ещё одного человека в таблицу (Пример 9).

Пример 9

```

try (Connection con = DriverManager.getConnection(url, user, password)){
    Statement statement = con.createStatement();
    statement.execute("DROP SCHEMA `test` ;");
    statement.execute("CREATE SCHEMA `test` ;");
    statement.execute("CREATE TABLE `test`.`table` (\n" +
        " `id` INT NOT NULL,\n" +
        " `firstname` VARCHAR(45) NULL,\n" +
        " `lastname` VARCHAR(45) NULL,\n" +
        " PRIMARY KEY (`id`));");
    statement.execute("INSERT INTO `test`.`table`
(`id`,`firstname`,`lastname`)\n" +
        "VALUES (1,'Иванов','Иван');");
    statement.execute("INSERT INTO `test`.`table`
(`id`,`firstname`,`lastname`)\n" +
        "VALUES (2,'Петров','Пётр');");
    ResultSet set = statement.executeQuery("SELECT * FROM `test`.`table` ;");
    while (set.next()){
        System.out.println(set.getString(3) + " " + set.getString(2) + " " + set.getInt(1));
    }
} catch (SQLException e) {

```

```
    throw new RuntimeException(e);  
}
```

По сути к коду примера 8 мы просто добавили одну строку. Запускаем, и видим в консоли.

Иван Иванов 1

Пётр Петров 2

Все в порядке! Мы взаимодействуем с базой данных, используя JDBC API! Давайте немного организуем код, выделив создание соединения в отдельный метод. JDBC дает возможность параллельно управлять несколькими соединениями, и не должно возникнуть проблем. (Пример 10)

Пример 10

```
public Connection getConnection(String url, String user, String password){  
    Connection con = null;  
    try {  
        con = DriverManager.getConnection(url, user, password);  
    } catch (SQLException e) {  
        throw new RuntimeException(e);  
    }  
    return con;  
}
```

Теперь для получения нового подключения мы можем просто вызвать этот метод. И тут возникает первая проблема. Она в том, что connection это ресурс сервера баз данных, и этот ресурс может закончиться. Например, вы забыли закрыть ресурс, а приложение продолжает работать. Серверу придётся держать связь открытой. Или, что ещё хуже, такое приложение в продакшене. Не закрытых ресурсов очень быстро наберётся на столько много, что у сервера просто закончатся конекшены и он скажет: «MySQL Error 1040: Too many connections». Ну ладно, допустим мы залатаем это проблему и двинемся дальше. Напишем метод выборки данных. (Пример 11)

Пример 11

```
public ArrayList<String> getDaata(Statement statement) throws SQLException {  
    ArrayList<String> list = new ArrayList<>();  
    ResultSet set = statement.executeQuery("SELECT * FROM `test`.`table`");  
    while (set.next()){  
        list.add(set.getString(3) + " " + set.getString(2) + " " + set.getInt(1));  
    }  
    return list;  
}
```

Здесь, на первый взгляд, все в порядке. Но предположим, что мы извлекаем не все данные, а только те, которые соответствуют определенному условию. То есть в ResultSet остается какая-то часть данных. Это приведет к тому, что курсор, который является основой ResultSet с одной стороны и ресурсом сервера с другой, останется в памяти. Но курсоры тоже не бесконечны! Проблемы с курсорами и соединениями усложняют работу чистых JDBC-приложений с таблицами. Ведь помимо самих запросов, нам нужно помнить еще и о соединениях и курсорах! А если речь идет о более сложных структурах с гораздо более сложными запросами, было бы удобно переложить все эти задачи на плечи вспомогательной библиотеки.

ORM

Такой подход существует и называется ORM, что расшифровывается как Object-Relational Mapping (Отображение объектов на реляционные базы данных). Суть ORM заключается в обращении к реляционным базам данных как к объектам. Реализация этого подхода осуществляется при помощи вспомогательных библиотек. Одной из таких популярных библиотек является Hibernate. В отличие от чистого JDBC, Hibernate использует JDBC для взаимодействия с СУБД, но весь остальной процесс работает совершенно иначе. Давайте начнем с создания новой таблицы. Это можно сделать с использованием простого SQL-запроса в MySQL Workbench. (Пример 1)

Пример 1

```
CREATE TABLE `test`.`magic` (  
  `id` INT NOT NULL AUTO_INCREMENT,  
  `название` VARCHAR(45) NULL,  
  `повреждение` INT NULL,  
  `атака` INT NULL,  
  `броня` INT NULL,  
  PRIMARY KEY (`id`));
```

Табличка есть! Она находится в схеме test, у неё есть поля «название», «повреждение», «атака», «броня» и ключевое поле id. Для поддержки hibernate в нашем проекте, добавим всего одну зависимость в pom.xml (Пример 2)

Пример 2

```
<dependency>  
  <groupId>org.hibernate</groupId>  
  <artifactId>hibernate-java8</artifactId>  
  <version>6.0.0.Alpha7</version>  
  <type>pom</type>  
</dependency>
```

Hibernate взаимодействует с СУБД через стандартный JDBC, поэтому ему требуются настройки подключения. Обычно для этого создают внешний конфигурационный файл. Вынесение конфигурации во внешний файл является хорошей практикой, поскольку это позволяет изменять настройки без внесения изменений в код. Как правило, этот файл называется hibernate.cfg.xml и размещается в папке resources. Если название и расположение файла корректны, мы сможем использовать его без указания пути и имени файла. Пример файла приведен ниже. (Пример 3)

Пример 3

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

                                                                 <property
name="hibernate.connection.driver_class">com.mysql.cj.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">root</property>
                                                                 <property
name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="show_sql">>true</property>

    <mapping class="project.Magic" />
</session-factory>
</hibernate-configuration>
```

В этом файле есть несколько полей, на которые следует обратить внимание. Во-первых, это поле driver-class. Здесь мы указываем JDBC-драйвер, который необходимо подключить для взаимодействия с сервером базы данных. В данном случае указан MySQL JDBC Driver, поскольку мы работаем с MySQL.

Во-вторых, поля password и username. Это учетные данные (пароль и логин), необходимые для установления связи с СУБД. В приведенном примере я использовал их значения по умолчанию. Однако следует помнить, что эти значения должны быть уникальными и конфиденциальными.

В-третьих, поле `dialect`. Hibernate может использовать различные диалекты для общения с базами данных, включая собственный HQL (Hibernate Query Language). В этом поле мы выбираем необходимый диалект.

Поле `show_sql` определяет, будут ли отображаться SQL-запросы в консоли или нет.

Еще один важный элемент файла конфигурации - это указание классов, которые Hibernate будет отображать в базе данных. В данном случае указан класс `Magic` в пакете `project`.

Мы указали класс для маппинга в конфигурации, но самого класса у нас пока нет. Давайте создадим его. Сначала создадим пакет `project`, а затем внутри него - POJO-класс `Magic`. (Пример 4)

Пример 4

```
@Entity
@Table(name = "test.magic")
public class Magic {}
```

Пока что этот класс пуст, но уже можно выделить несколько интересных и важных моментов для ORM. Во-первых, в классе появилась аннотация `@Entity`. Эта аннотация указывает, что мы определяем объект-сущность. Другими словами, объект, созданный на основе этого класса, будет использоваться Hibernate при взаимодействии с базой данных. Объекты-сущности должны следовать спецификации POJO, что означает, что поля должны быть приватными, иметь геттеры и сеттеры, а также класс должен иметь пустой конструктор. Обычно также переопределяют метод `toString` для удобства.

Вторым важным моментом является аннотация `@Table`. С параметром `name` мы связываем класс с определенной схемой и таблицей. Таким образом, мы создали класс-сущность и привязали его к таблице. Давайте добавим необходимые поля. (Пример 5)

Пример 5

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private int idmagic;

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

@Column(name = "повреждение")
private int damage;

@Column(name = "атака")
private int attBonus;
```

Сами поля не представляют ничего особенного, но стоит рассмотреть аннотации. Первая аннотация `@Id` является обязательной и определяет первичный ключ. В отличие от работы с базами данных через JDBC, где не требуется обязательного использования ключей, в ORM предполагается другая философия. Сущность определяется в классе Java, экземпляры которого хранятся в базе данных, и первичный ключ обязателен для сопоставления данных в таблице и объектов в приложении. Аннотация `@GeneratedValue` определяет стратегию формирования ключа. В данном примере используется `GenerationType.IDENTITY`, самый простой способ конфигурации ключа, который опирается на auto-increment колонку в таблице. Аннотация `@Column` привязывает поле к колонке в базе данных. Если название поля совпадает с названием колонки, привязывать не обязательно.

Таким образом, мы создали класс с аннотацией `@Entity`, привязали его к таблице, объявили приватные поля с аннотацией ключевого поля и его конфигурацией, добавили приватные поля для остальных колонок таблицы. Осталось добавить конструктор с ключами и пустой, геттеры и сеттеры для всех полей. Это можно сделать с помощью среды разработки, например, IDEA. Теперь класс готов! Мы готовы протестировать удобства подхода ORM на практике. Для этого создадим новый класс `Main` в пакете `project` с методом `main` (psvm). Кода пока не так много, но достаточно, чтобы продемонстрировать значимые отличия. (Пример 6)

Пример 6

```
final StandardServiceRegistry registry = new StandardServiceRegistryBuilder()
    .configure() // configures settings from hibernate.cfg.xml
    .build();

SessionFactory sessionFactory = new MetadataSources( registry
).buildMetadata().buildSessionFactory();
Session session = sessionFactory.openSession();
Magic magic = new Magic("Волшебная стрела", 10, 0, 0);
session.beginTransaction();
session.save(magic);
session.getTransaction().commit();
session.close();
```

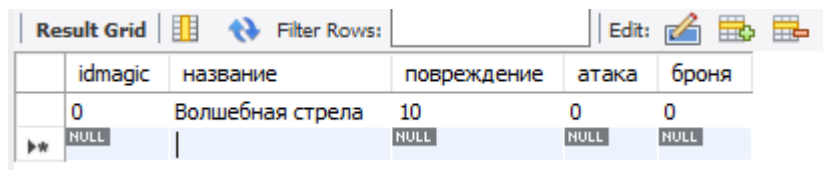
Давайте рассмотрим каждую строку программы по порядку. В первой строке мы объявляем и инициализируем экземпляр класса `StandardServiceRegistry`. Этот класс содержит механизмы связи с сервером базы данных и менеджер передачи запросов. Файл конфигурации Hibernate, который мы ранее создали, используется этим классом. Поскольку мы разместили файл в нужном каталоге с правильным именем, конфигурировать его можно без дополнительных ссылок.

Далее идет создание `SessionFactory`. Согласно документации, это "неизменяемый, потокобезопасный объект с скомпилированным маппингом для одной базы данных". Его нужно инициализировать всего один раз. Экземпляр `SessionFactory` используется для получения объектов `Session`, которые используются для операций с базами данных.

После получения `SessionFactory` мы создаем объект `Session`. Этот объект связывает наше приложение с Hibernate. Затем создается объект `magic`, экземпляр класса `Magic` с использованием параметризованного конструктора.

Далее идет работа с сессией. Одним из первых действий является сохранение сущностей в базе данных. Важно отметить, что сохранять сущности можно только в рамках транзакций. Поэтому мы начинаем транзакцию с помощью `beginTransaction()`, затем сохраняем объект в базе данных и закрываем транзакцию с записью данных (`getTransaction().commit()`). После этого сессия закрывается.

Вся необходимая работа завершена, и мы можем запустить проект. В консоли выводится много информации, но ошибок нет. Теперь давайте проверим, что произошло в таблице. Откроем MySQL Workbench, подключимся к нашей схеме и выполним SQL-запрос: "SELECT * FROM test.magic".



	idmagic	название	повреждение	атака	броня
	0	Волшебная стрела	10	0	0
▶▶	NULL		NULL	NULL	NULL

Вот она, наша волшебная стрела! А значит, у нас получилось подключиться к серверу MySQL и записать в нужную таблицу наш объект! Выполним пару sql запросов.

```
delete from test.magic  
alter table magic auto_increment=1
```

Удалим данные из таблицы и сбросим счетчик ключа соответственно. А теперь давайте сделаем наш код более чистым. Создадим класс Connector и перенесём в него всё, что касается создания sessionFactory а выделение сессии вынесим в метод getSession(). Пример 7

Пример 7

```
public class Connector{  
    final StandardServiceRegistry registry;  
    SessionFactory sessionFactory;  
  
    public Connector() {  
        registry = new StandardServiceRegistryBuilder()  
            .configure() // configures settings from hibernate.cfg.xml  
            .build();  
        sessionFactory = new MetadataSources( registry  
        ).buildMetadata().buildSessionFactory();  
    }  
  
    public Session getSession(){  
        return sessionFactory.openSession();  
    }  
}
```

Класс простой и описывать в нём особо нечего. Поэтому возвращаемся в main и всё что там было меняем на новый код. (Пример 8)

Пример(8)

```
Connector connector = new Connector();
Session session = connector.getSession();
Magic magic = new Magic("Волшебная стрела", 10, 0, 0);
session.beginTransaction();
session.save(magic);
magic = new Magic("Молния", 25, 0, 0);
session.save(magic);
magic = new Magic("Каменная кожа", 0, 0, 6);
session.save(magic);
magic = new Magic("Жажда крови", 0, 6, 0);
session.save(magic);
magic = new Magic("Жажда крови", 0, 6, 0);
session.save(magic);
magic = new Magic("Проклятие", 0, -3, 0);
session.save(magic);
magic = new Magic("Лечение", -30, 0, 0);
session.save(magic);
session.getTransaction().commit();
session.close();
```

В этом фрагменте также нет ничего особо нового; мы просто используем наш коннектор и сохраняем немного больше объектов. После запуска, снова без ошибок, выполним SQL-запрос "SELECT * FROM test.magic".

Result Grid

Filter Rows:

Edit:

Export/Import:

	idmagic	название	повреждение	атака	броня
	1	Волшебная стрела	10	0	0
	2	Молния	25	0	0
	3	Каменная кожа	0	0	6
	4	Жажда крови	0	6	0
	5	Жажда крови	0	6	0
	6	Проклятие	0	-3	0
	7	Лечение	-30	0	0
»*	NULL	NULL	NULL	NULL	NULL

И все наши объекты оказались в базе данных! Ключи, как и положено, начинаются с единицы. "Жажда крови" повторилась дважды, и это тоже корректно, ведь я создал два объекта с одинаковыми параметрами, но для Hibernate, как и для Java, это два разных объекта. Теперь давайте попробуем загрузить объекты из базы данных. Для начала отправим в ремарку и напишем вот что. (Пример 9)

Пример 9

```
Connector connector = new Connector();
try (Session session = connector.getSession()) {
    List<Magic> books = session.createQuery("FROM Magic",
Magic.class).getResultList();
    books.forEach(b -> {
        System.out.println("Book of Magic : " + b);
    });
} catch (Exception e) {
    e.printStackTrace();
}
```

В этом коде, прежде всего, следует отметить, что я обернул создание сессии в блок try/catch с использованием автоматического закрытия ресурсов. Создаваемая сессия будет обязательно закрыта. Это удобный инструмент для работы с ресурсами, требующими закрытия. Ещё одна особенность заключается в том, что транзакция не создаётся. Это верно, поскольку при чтении данных из таблицы транзакции не требуются. Выборка осуществляется с использованием метода createQuery(). Первый параметр - это запрос в формате HQL, который буквально указывает, из какой таблицы читать, а второй - это класс-сущность, описывающий данные в таблице. Для получения списка результатов используется метод getResultList(). Запускаем код, смотрим в консоль!

Hibernate: select m1_0.idmagic, m1_0.атака, m1_0.повреждение, m1_0.броня, m1_0.название from test.magic as m1_0

Book of Magic : name='Волшебная стрела', damage=10, attBonus=0, defBonus=0

Book of Magic : name='Молния', damage=25, attBonus=0, defBonus=0

Book of Magic : name='Каменная кожа', damage=0, attBonus=0, defBonus=6

Book of Magic : name='Жажда крови', damage=0, attBonus=6, defBonus=0

Book of Magic : name='Жажда крови', damage=0, attBonus=6, defBonus=0

Book of Magic : name='Проклятие', damage=0, attBonus=-3, defBonus=0

Book of Magic : name='Лечение', damage=-30, attBonus=0, defBonus=0

И видим ровно те данные, которые мы сохранили в базу данных! А значит, у нас получилось сохранить объекты в таблицу и загрузить их обратно! Это здорово, давайте попробуем теперь изменить один из объектов. (Пример 10)

Пример 10

```
try (Session session = connector.getSession()) {
    String hql = "from Magic where id = :id";
    Query<Magic> query = session.createQuery( hql, Magic.class);
    query.setParameter("id", 4);
    Magic magic = query.getSingleResult();
    System.out.println(magic);
    magic.setAttBonus(12);
    magic.setName("Ярость");
    session.beginTransaction();
    session.update(magic);
    session.getTransaction().commit();
} catch (Exception e) {
    e.printStackTrace();
}
```

Этот код выглядит сложнее, однако практически ничего нового тут нет. Сначала я задал строку hql. Тут важно, что в запросе есть указатель на ключевое поле, туда можно было сразу положить идентификатор, но в таком виде его можно задать потом. Далее создаю запрос (Query) и устанавливаю параметр «id» равным четырём. Метод `getSingleResult()` возвращает результат удовлетворяющий заданным параметрам. Результат я сразу сохраняю в объект `magic`. Далее меняю его параметр и имя. Далее я хочу обновить данные в базе, а делается это в рамках транзакции. Поэтому я начинаю транзакцию, обновляю данные методом `update` и транзакцию закрываю! Запускаем.

```
Hibernate: select m1_0.idmagic, m1_0.атака, m1_0.повреждение, m1_0.броня,
m1_0.название from test.magic as m1_0 where m1_0.idmagic = 4
name='Ярость', damage=0, attBonus=12, defBonus=0
Hibernate: update test.magic set атака=?, повреждение=?, броня=?, название=?
where idmagic=?
```

Ошибок нет, а значит всё прошло хорошо! Но давайте проверим исполнив уже знакомый нам способ чтения данных из базы. (Пример 9)

```
Hibernate: select m1_0.idmagic, m1_0.атака, m1_0.повреждение, m1_0.броня,
m1_0.название from test.magic as m1_0
Book of Magic : name='Волшебная стрела', damage=10, attBonus=0, defBonus=0
```

Book of Magic : name='Молния', damage=25, attBonus=0, defBonus=0
Book of Magic : name='Каменная кожа', damage=0, attBonus=0, defBonus=6
Book of Magic : name='Ярость', damage=0, attBonus=12, defBonus=0
Book of Magic : name='Жажда крови', damage=0, attBonus=6, defBonus=0
Book of Magic : name='Проклятие', damage=0, attBonus=-3, defBonus=0
Book of Magic : name='Лечение', damage=-30, attBonus=0, defBonus=0

Отлично! «Ярость» на месте, а значит, мы умеем обновлять элементы в базе данных!
Осталось научиться удалять их оттуда! Пример (10)

Пример 10

```
try (Session session = connector.getSession()) {  
    Transaction t = session.beginTransaction();  
        List<Magic> books = session.createQuery("FROM Magic",  
Magic.class).getResultList();  
    books.forEach(b -> {  
        session.delete(b);  
    });  
    t.commit();  
} catch (Exception e) {  
    e.printStackTrace();  
}  
}
```

Пример очень похож на чтение за несколькими отличиями. Во-первых, удаление проходит в рамках транзакции, для этого я создаю её первым делом. Сам метод удаления delete(b) в параметре требует только ссылку на объект, данные которого мы хотим удалить из базы. Ну и закрытие транзакции, которое я вызываю после удаления всех полученных из бузы объектов. Запускаем и смотрим в консоль.

```
Hibernate: select m1_0.idmagic, m1_0.атака, m1_0.повреждение, m1_0.броня,  
m1_0.название from test.magic as m1_0  
Hibernate: delete from test.magic where idmagic=?  
Hibernate: delete from test.magic where idmagic=?  
Hibernate: delete from test.magic where idmagic=?  
Hibernate: delete from test.magic where idmagic=?  
Hibernate: delete from test.magic where idmagic=?  
Hibernate: delete from test.magic where idmagic=?  
Hibernate: delete from test.magic where idmagic=?
```

Судя по информации в терминале, из базы было удалено 7 элементов. Давайте посмотрим что осталось. (Пример 9)

Hibernate: select m1_0.idmagic, m1_0.атака, m1_0.повреждение, m1_0.броня, m1_0.название from test.magic as m1_0

Это всё что мы видим в консоли, а значит, что в базе не осталось ничего! Я так радуюсь потому, что мы разобрали полноценное CRUD – приложение.

- C -> Create/Insert
- R -> Retrieve
- U -> Update
- D -> Delete

Фактически, данное приложение способно создавать, выбирать, обновлять и удалять данные из базы данных. Теперь давайте сравним работу с простым JDBC и более сложным Hibernate. В каждом из подходов есть свои плюсы и минусы, рассмотрим их.

JDBC требует добавления в зависимости драйвера базы данных. Это преимущество, так как остальные шаги выполняются в коде. Мы определяем путь к серверу базы данных, логин и пароль прямо в коде. Создаём Connection, и всё готово для выполнения нужных запросов. Этот подход довольно компактен и выглядит неплохо. Однако у него есть существенный недостаток — отсутствие объектно-ориентированной модели поведения. Для создания объекта из загруженных данных придётся написать собственный код, что касается и сохранения, и модификации, и удаления данных. Также важно отметить, что работа с разными системами управления базами данных различается, что затрудняет переход между ними, поскольку требует изменения внутреннего слоя взаимодействия с базой данных. JDBC предоставляет отличную основу, но всё же это основа!

Hibernate, с другой стороны, выступает в роли прослойки между JDBC и программистом, автоматизируя многие рутинные задачи. Для его подключения необходимо добавить, как минимум, две зависимости и создать файл конфигурации. Создание системы классов приложения также требует внимания, и это недостаток. Однако Hibernate является реализацией подхода ORM (Object-Relational Mapping), что означает, что мы работаем с базами данных как с контейнером для хранения объектов. Это настолько удобно, что, если приложение разработано корректно, работа с базой данных сводится к вызову методов.

Реализация основных задач скрыта в самой библиотеке, что сделало Hibernate одной из самых популярных библиотек для работы с базами данных.

JPA

Java Persistence API (JPA) представляет собой стандартный интерфейс в языке Java для работы с системами управления реляционными базами данных. Этот стандарт был создан для облегчения и унификации разработки приложений, использующих базы данных, и предоставляет объектно-ориентированный способ взаимодействия с реляционными данными.

Вот некоторые ключевые аспекты и особенности JPA:

1. Объектно-Реляционное Отображение (ORM):

JPA предоставляет механизм для отображения Java-объектов на таблицы в базе данных и наоборот. Это позволяет разработчикам работать с объектами в коде, не беспокоясь о деталях хранения данных в базе.

2. Аннотации:

Для настройки отображения объектов в базу данных, JPA использует аннотации, которые добавляются к классам и их полям. Например, аннотация `@Entity` используется для обозначения класса как сущности базы данных.

Пример:

```
@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String username;
    private String password;

    // геттеры и сеттеры
}
```


3. **Язык запросов JPQL:** JPA предоставляет язык запросов под названием Java Persistence Query Language (JPQL). Этот язык аналогичен SQL, но использует объекты Java вместо таблиц базы данных.

Пример JPQL-запроса:

```
TypedQuery<User> query = entityManager.createQuery("SELECT u FROM User u WHERE u.username = :username", User.class);
query.setParameter("username", "john_doe");
List<User> resultList = query.getResultList();
```

4. **Жизненный цикл объектов:** JPA управляет жизненным циклом объектов в контексте персистентности. Объекты проходят через различные состояния, такие как новый (new), управляемый (managed), отсоединенный (detached) и удаленный (removed).

5. Маппинг связей:

JPA позволяет определять отношения между сущностями, такие как один к одному, один ко многим и многие к одному, многие ко многим, используя аннотации.

Пример отношения один ко многим:

```
@Entity
public class Author {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @OneToMany(mappedBy = "author")
    private List<Book> books;

    // геттеры и сеттеры
}

@Entity
public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```

private Long id;

private String title;

@ManyToOne
@JoinColumn(name = "author_id")
private Author author;

// геттеры и сеттеры
}

```

6. Управление транзакциями:

JPA обеспечивает управление транзакциями, позволяя разработчикам атомарно выполнять операции чтения и записи.

Пример транзакции:

```

EntityManager entityManager = //... получение EntityManager
EntityTransaction transaction = entityManager.getTransaction();

try {
    transaction.begin();

    // выполняем операции с базой данных

    transaction.commit();
} catch (Exception e) {
    if (transaction != null && transaction.isActive()) {
        transaction.rollback();
    }
}

```

7. Поставщики персистентности: JPA не является самостоятельной реализацией, а предоставляет стандартный интерфейс. Для фактической реализации JPA используются различные поставщики персистентности, такие как Hibernate, EclipseLink и другие.

JPA предоставляет удобный и мощный способ взаимодействия с базами данных в Java-приложения

Заключение

Заключение урока: Сегодня мы погрузились в увлекательный мир взаимодействия с базами данных в Java. Мы изучили основы JDBC, который предоставляет прямой доступ к базе данных через SQL-запросы, и оценили его компактность и простоту.

Затем мы перешли к более высокоуровневым концепциям с использованием ORM, где Hibernate выступает в качестве посредника между приложением и базой данных. Мы рассмотрели преимущества объектно-реляционного отображения, такие как удобство работы с объектами в коде и автоматизация рутинных задач.

Не обошли вниманием и Java Persistence API (JPA), который стандартизирует работу с базами данных, предоставляя унифицированный интерфейс для ORM-фреймворков.

Надеюсь, что эта экскурсия в мир работы с данными была для вас интересной и полезной. Если у вас возникли вопросы, не стесняйтесь задавать их. На следующем уроке, где мы поговорим о сокетах и дальнейших возможностях взаимодействия в веб-приложениях.