

Гайд по праку

Субботин Даниил

20 марта 2022 г.

Автор не несет ответственности ни за что :) Изложенный здесь материал - сугубо личный опыт и не является руководством. Как бы сказал капитан Барбосса, этот гайд - просто свод указаний, а не жёстких законов.

1 Начало

1.1 Перед началом

В тексте содержится много достаточно очевидных для некоторых вещей. Однако всё, что здесь описано, для других не является очевидным. Это было выяснено опытным путем в процессе работы группы СП над праком. Поскольку изначально этот гайд задумывался как средство экономии времени и облегчения жизни другим студентам, я специально описывал все вещи, которые вызывали затруднения у моих коллег, чтобы расширить гайд и сделать его максимально полезным для всех.

1.2 Сурцы


Github для просмотра всего кода: <https://github.com/WhiteWolfGeralt/Jaba-prak>

1.3 Выбор IDE

Прежде всего нужно удалить все богопротивные IDE и поставить единственно верную - IntelliJ IDEA от JetBrains (желательно Ultimate версию, т.к. в только в ней есть удобный интерфейс для работы с базами данных). Всё дальнейшее изложение будет продемонстрировано именно на примере ее (далее - просто "идея"). Фанатам Vim'а соболезную, но для этого прака придётся использовать редакторы с графичекой оболочкой... Кстати, замечание для всех тех, кто раньше не работал с такими редакторами: если что-то не работает, то почти всегда идея сама подскажет, что нужно сделать для исправления ошибки - достаточно навести мышкой на проблемный код и оценить то, что предлагает редактор (например, импортировать модуль или добавить зависимость в сборщик). В большинстве случаев решение правильное и можно в один клик сделать то, что требуется.

1.4 Создание проекта

Для начала создаем проект с помощью <https://start.spring.io>. Для сборки будем использовать [gradle](#). При создании выбираем драйвер базы данных и желаемые тулы (я выбирал себе Lombok, PostgreSQL, Spring Web, Thymeleaf). Все необходимое (или забытое) также можно добавить позже, просто прописав зависимости в [build.gradle](#). Возможный вид проекта перед генерацией (Group можно не писать):



Project

☐ Maven Project
 ☒ Gradle Project

Language

☒ Java
 ☐ Kotlin
 ☐ Groovy

Spring Boot

☐ 3.0.0 (SNAPSHOT)
 ☐ 3.0.0 (M1)
 ☐ 2.7.0 (SNAPSHOT)
 ☐ 2.7.0 (M2)
 ☐ 2.6.5 (SNAPSHOT)
 ☒ 2.6.4
 ☐ 2.5.11 (SNAPSHOT)
 ☐ 2.5.10

Project Metadata

Group

ru.msu.cmc

Artifact

webprak

Name

webprak

Description

Webprak

Package name

ru.msu.cmc.webprak

Packaging

☒ Jar
 ☐ War

Java

☒ 17
 ☐ 11
 ☐ 8

Dependencies

ADD DEPENDENCIES... CTRL + B

Lombok

DEVELOPER TOOLS

Java annotation library which helps to reduce boilerplate code.

PostgreSQL Driver

SQL

A JDBC and R2DBC driver that allows Java programs to connect to a PostgreSQL database using standard, database independent Java code.

Spring Web

WEB

Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Thymeleaf

TEMPLATE ENGINES

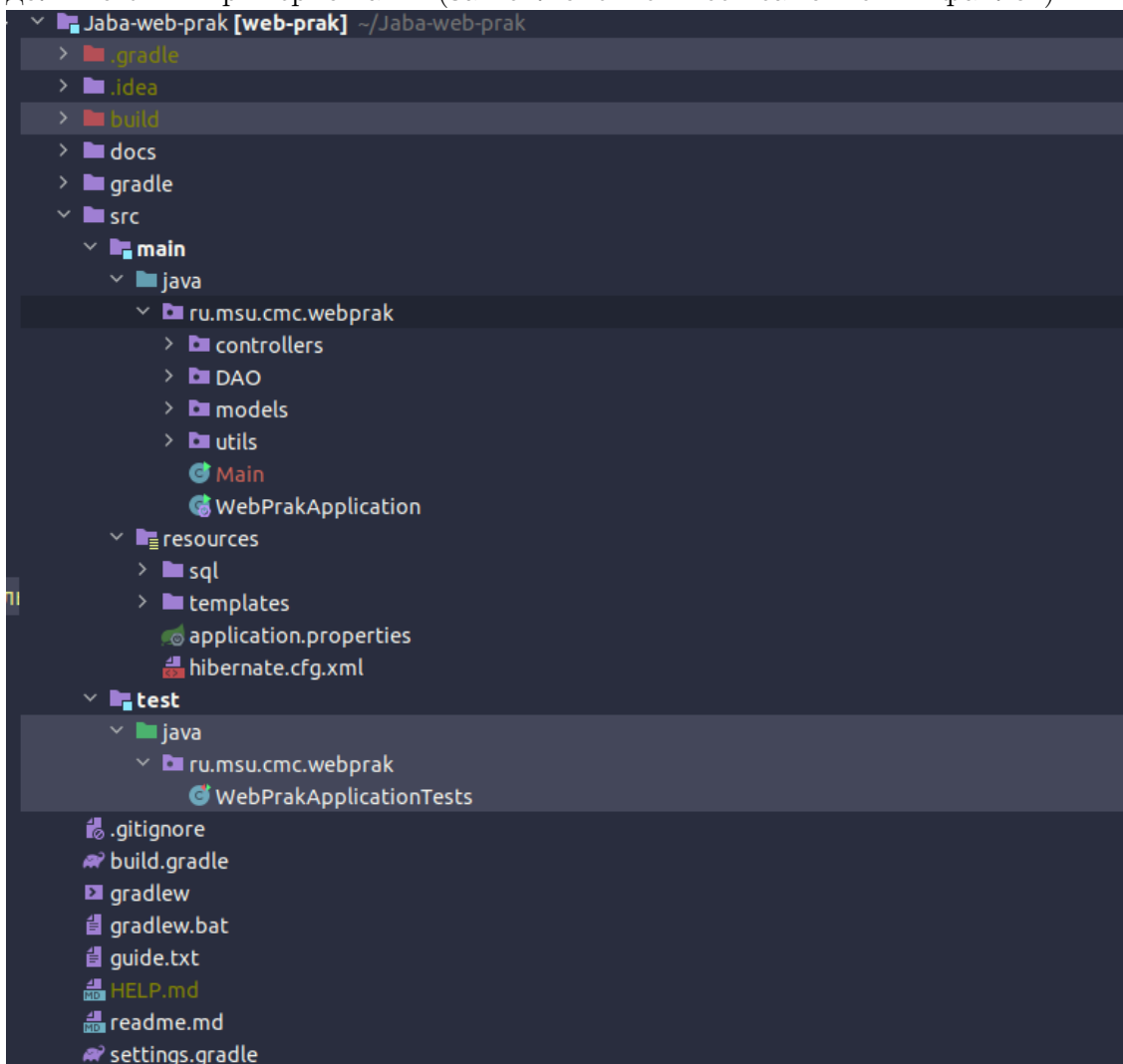
A modern server-side Java template engine for both web and standalone environments. Allows HTML to be correctly displayed in browsers and as static prototypes.

GENERATE CTRL + G

EXPLORE CTRL + SPACE

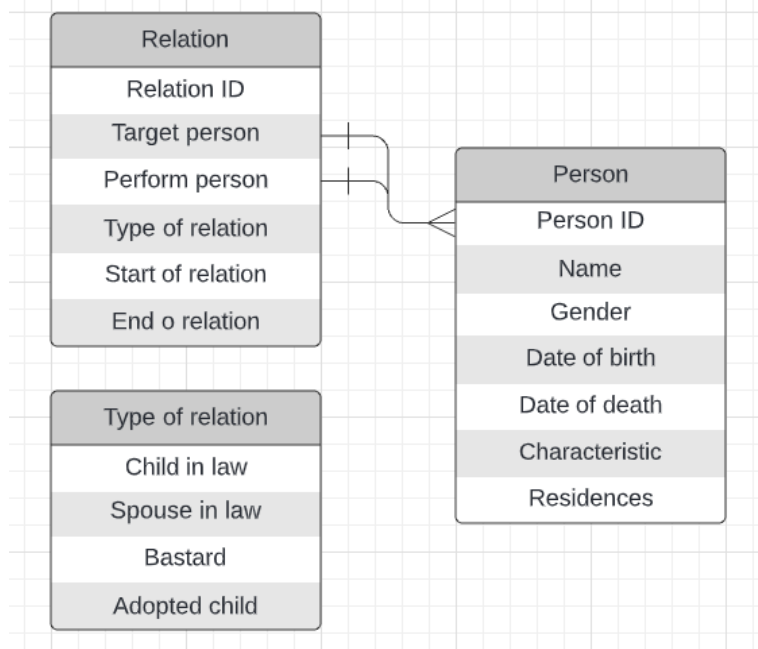
SHARE...

Распаковываем, открываем проект через идею. В случае, если это первый опыт использования идеи, то для корректной работы необходимо установить SDK (собственно, идея сама предложит это сделать, если перейти в один из сурцов). Вручную можно настроить через **Ctrl+Alt+Shift+S**. После установки SDK ждём, пока идея проиндексирует весь проект (за прогрессом можно следить по шкале в правом нижнем углу). Содержание проекта должно быть примерно таким (за исключением всех самописных файлов):



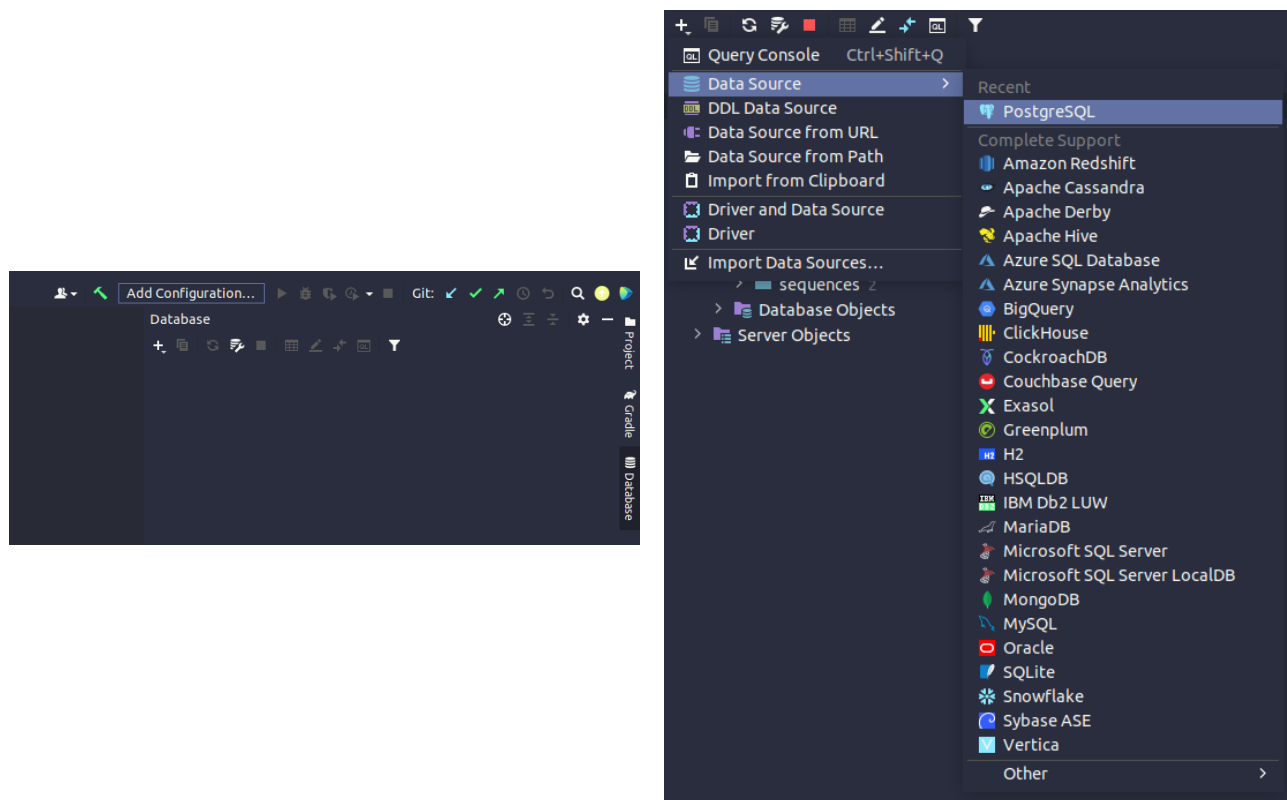
2 Классы и база данных

Схема моей БД для наглядности последующего кода и моих комментариев:

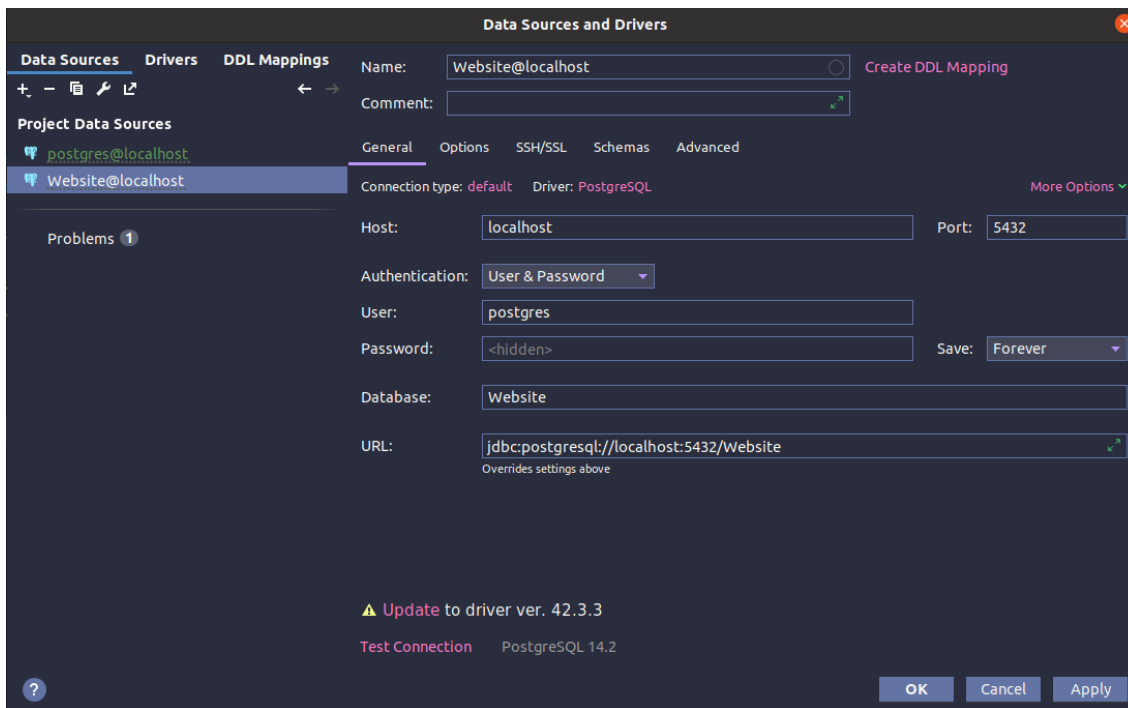


2.1 Создание БД

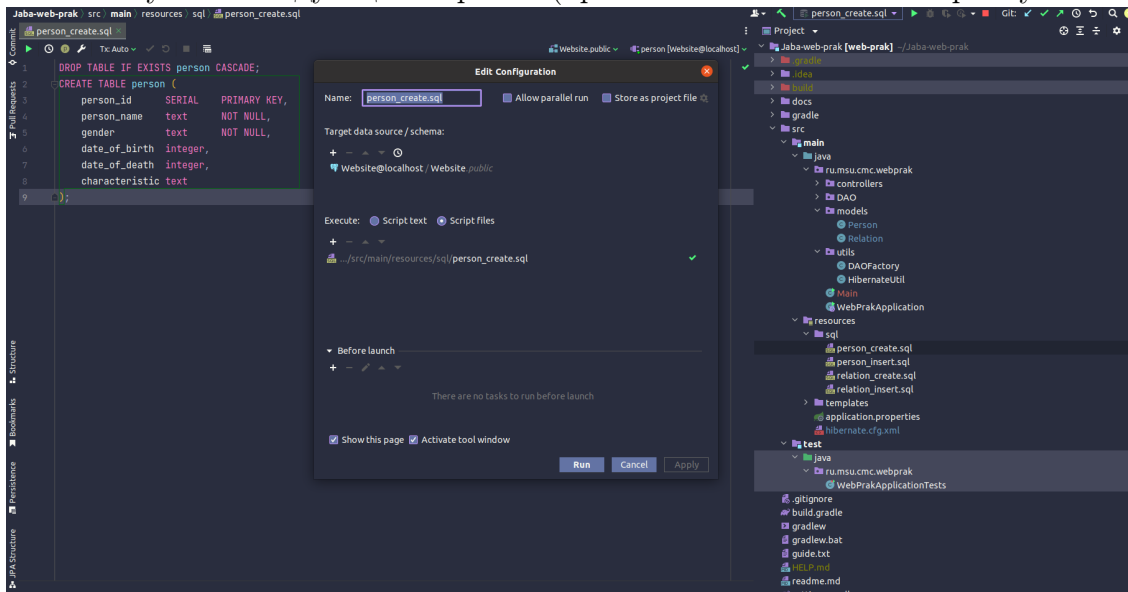
Создаём себе новую базу данных, используя, например, наш любимый PgAdmin. Сразу подключаем БД в идею через вкладку Database. Через **+** выбираем нужную СУБД во вкладке идеи Database:



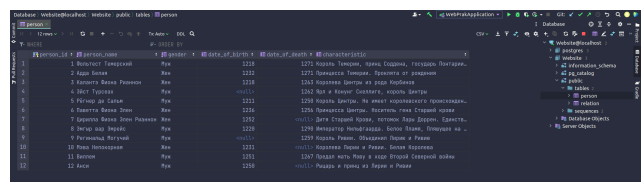
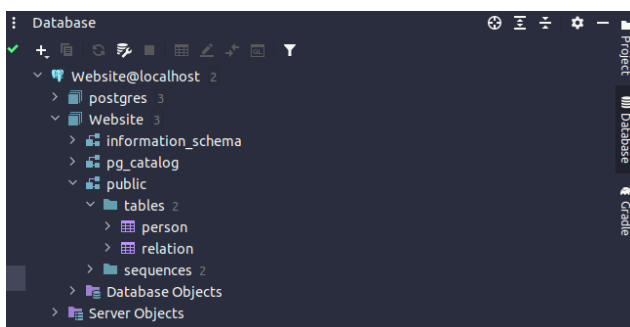
Вводим параметры для подключения (параметры, которые задавали при создании). При отсутствии пароля, как в моей случае, просто оставляем поле пустым:



Далее, удобно в отдельной папке в проекте прописать все create и insert скрипты и отсюда же их запускать следующим образом (правая кнопка мыши по скрипту -> run):



Через **+** выбираем только что подключенную базу, запускаем. Сразу же имеем возможность оценить все прелести современных IDE, просматривая и изменяя БД прямо из редактора:



2.2 Написание классов

Теперь на каждую созданную таблицу нужно написать класс, который будет с ней связан. Название класса не имеет значения, связь с таблицей реализуется не через название классов. Все типы, которые можно использовать в БД, реализованы и в Java, поэтому с определением типов полей проблем быть не должно. Удобно поместить все классы в пакет `models` (на самом деле, здесь и далее это не имеет никакого значения, можно вообще хранить все файлы в одной папке):

```
@Entity
@Table(name = "person")
@Getter
@Setter
@ToString
@NoArgsConstructor
@AllArgsConstructor
public class Person {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(nullable = false, name = "person_id")
    private long person_id;

    @Column(nullable = false, name = "person_name")
    @NotNull
    private String name;

    @Column(nullable = false, name = "gender")
    @NotNull
    private String gender;

    @Column(name = "date_of_birth")
    private Long birth;

    @Column(name = "date_of_death")
    private Long death;

    @Column(name = "characteristic")
    private String character;
```

```
@Entity
@Table(name = "relation")
@Getter
@Setter
@ToString
@NoArgsConstructor
@AllArgsConstructor
public class Relation {

    public enum RelType { ... }

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(nullable = false, name = "relation_id")
    private long relation_id;

    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = "target_person")
    @ToString.Exclude
    @NotNull
    private Person target;

    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = "perform_person")
    @ToString.Exclude
    @NotNull
    private Person perform;

    @Column(nullable = false, name = "type_of_relation")
    @NotNull
    private RelType type;

    @Column(name = "start_of_relation")
    private Long start;
```

2.3 Маппинг классов

Для маппинга классов пользоваться конфигурационными файлами `hbm.xml` не будем, связывание будет реализовано с помощью декораторов. (Про связывание с помощью `hbm.xml` есть информация в [1]) Основной файл `hibernate.cfg.xml` написать все же придется, но обо всем по порядку.

2.3.1 Декораторы для класса

Превращаем класс в сущность с помощью декоратора `@Entity`, декоратор `@Table` позволяет связать класс с конкретной таблицей в БД (здесь уже важно указать правильное имя таблицы). Далее идут декораторы из `lombok`, которые позволяют сделать код чище. Так, `@Getter` и `@Setter` позволяют явно не описывать геттеры и сеттеры для класса, они будут сгенерированы автоматически для каждого поля. Аналогично, `@ToString` генерирует текстовое представление поля. `@NoArgsConstructor` и `@AllArgsConstructor` генерируют конструкторы с заданными параметрами.

2.3.2 Декораторы для полей

Как видно из кода выше, одно (или несколько, не знаю, как будет вести себя `hibernate` при этом) должно быть помечено декоратором `@Id`, что, соответственно, означает первичный ключ в этом классе. С помощью декоратора `@Column` происходит маппинг полей на столбцы БД. Параметр `name` должен задавать точное имя столбца. При необходимости, можно указать, что столбец не может иметь нулевое значение, а также явно установить проверку на `null`-значение с помощью `lombok`-овского `@NotNull`.

ВАЖНО: если поле может иметь `null`-значение, то для его описания необходимо использовать класс-обертку, а не базовый тип (например, `Long` вместо `long`).

Для указания связей между таблицами я использовал декоратор `@ManyToOne`. Поле класса (в этом абзаце класс и таблица будут значить одно и то же) должно иметь тип класса, на который ссылается внешний ключ в этом классе. В моем случае, класс `Relation`

имеет внешний ключ - поле `target_person`, который ссылается на класс `Person`. С помощью `@JoinColumn` задаем имя поля. Есть декораторы и для других типов связей, например `@ManyToMany`, он требует создания коллекций объектов класса.

2.3.3 Конфигурационный файл

Самая нелюбимая (автор идиот и потратил на это > 7 часов) часть. В файле `hibernate.cfg.xml` должен появиться примерно следующий текст:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE hibernate-configuration PUBLIC
3     "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
4     "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
5 <hibernate-configuration>
6     <session-factory>
7         <!-- Database connection settings -->
8         <property name="connection.driver_class">org.postgresql.Driver</property>
9         <property name="connection.url">jdbc:postgresql://localhost/Website</property>
10        <property name="connection.username">postgres</property>
11        <property name="connection.password"/>
12        <property name="show_sql">true</property>
13
14        <mapping class="ru.msu.cmc.webprak.models.Person"/>
15        <mapping class="ru.msu.cmc.webprak.models.Relation"/>
16
17    </session-factory>
18 </hibernate-configuration>
```

Соответственно, в `property` должны быть указаны параметры для доступа к БД, а в `mapping` просто перечислены все классы, которые реализуют таблицы (текст конфига есть на гитхабе). Можно не писать и этот файл и также реализовать конфиг через `java`, но я не разобрался как это сделать, поэтому выбрал вариант с `xml`.

3 Написание DAO

DAO - data access object. Это паттерн, который управляет соединением с источником данных для получения и записи данных. Он абстрагирует и инкапсулирует доступ к источнику данных. (Подробнее можно прочитать в [2] и [3]) Исходя из осознания того, что такое DAO, понимаем, что нужно написать интерфейс и его реализацию для каждого класса-сущности. Примерная структура реализации DAO:



```
DAO
├── impl
│   ├── PersonDAOImpl
│   └── RelationDAOImpl
├── PersonDAO
└── RelationDAO
```

3.1 Интерфейс

Собственно, пишем сам интерфейс. В DAO должны присутствовать типовые запросы приложения к БД, например все SQL команды (`insert`, `update`, `delete`), а так же необходимые приложению запросы (конечно, зависит от задания, но самые общие методы - это получение записи из БД по ID, или `SELECT` по `LIKE`, или `SELECT` всех записей, и прочие универсальные команды).

```

public interface RelationDAO {
    void addRelation(Relation relation);
    void updateRelation(Relation relation);
    void deleteRelation(Relation relation);

    Relation getRelationById(Long relationId);
    List<Relation> getRelationAll();

    List<Person> getAllByRelType(Long personId, Relation.RelType type);
}

```

3.2 ...перед реализацией

При реализации всех методов DAO мы должны как-то обращаться к БД. Одного звывания мало. Необходимо создавать т.н. **сессии**. Для этого напомним отдельный класс, который будет генерировать сессии запросов к БД. В классе сформируем сессию с помощью средств гиббернейта:

```

public class HibernateUtil {
    private static final SessionFactory sessionFactory;
    static {
        try {
            sessionFactory = new Configuration().configure().buildSessionFactory();
        } catch (Throwable ex) {
            System.err.println("Initial SessionFactory creation failed." + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }
}

```

3.3 Реализация

Сама реализация всех методов. С помощью класса, описанного выше, реализуем все методы, описанные в интерфейсе. Если производятся манипуляции с изменением БД, то используется стандартная система транзакций (начало-изменение-коммит). Если метод сложнее примитивного, то как раз с помощью объекта-сессии можно писать запросы на родном SQL языке. В гиббернейте используется диалект HQL([4]). По-хорошему, надо бы обернуть всю реализацию в try-catch блоки, но я забил. Если всё же перепишу, обновлю гайд (но это не точно). Для примера оставлю реализацию пары методов:

```

@Override
public void deletePerson(Person person) {
    Session session = HibernateUtil.getSessionFactory().openSession();
    session.beginTransaction();
    session.delete(person);
    session.getTransaction().commit();
    session.close();
}

@Override
public List<Person> getPersonByName(String personName) {
    Session session = HibernateUtil.getSessionFactory().openSession();
    Query<Person> query = session.createQuery("FROM Person WHERE name LIKE :gotName", Person.class)
        .setParameter("gotName", value: "%" + personName + "%");
    if (query.getResultList().size() == 0) {
        return null;
    }
    return query.getResultList();
}

```

3.4 Фабрика DAO

Теперь создадим класс фабрики, к которой будем обращаться за нашими реализациями DAO, от которых и будем вызывать необходимые нам методы:

```
public class DAOFactory {

    private static PersonDAO personDAO = null;
    private static RelationDAO relationDAO = null;
    private static DAOFactory instance = null;

    public static synchronized DAOFactory getInstance(){
        if (instance == null){
            instance = new DAOFactory();
        }
        return instance;
    }

    public PersonDAO getPersonDAO(){
        if (personDAO == null){
            personDAO = new PersonDAOImpl();
        }
        return personDAO;
    }

    public RelationDAO getRelationDAO(){
        if (relationDAO == null){
            relationDAO = new RelationDAOImpl();
        }
        return relationDAO;
    }
}
```

4 Запуск и тестирование

На данный момент мое тестирование - это `println`. По ходу доделывания прака раздел будет пополняться:

The screenshot displays an IDE with the following components:

- Editor (Left):** Shows the `WebPrakApplication.java` file. The code is a Spring Boot application that uses `SpringApplication.run()` to start. It has a `main` method that takes an array of strings as arguments. Inside the `main` method, it calls `DAOFactory.getInstance().getPersonDAO().getPersonByName("Фюна")` to retrieve a person by name. It then checks if the person is null and, if not, iterates over the list of persons and prints their string representation.
- Project Explorer (Right):** Shows the project structure. The root is `main`, which contains `java` and `resources` folders. The `java` folder contains `controllers`, `DAO`, `models`, and `utils` subfolders. The `resources` folder contains `sql` and `templates` subfolders. The `sql` folder contains `person_create.sql`, `person_insert.sql`, `relation_create.sql`, and `relation_insert.sql`. The `templates` folder contains `application.properties` and `hibernate.cfg.xml`.
- Run Console (Bottom):** Shows the output of the application. It starts with a message from Spring Boot indicating the version (2.6.4). Then, it shows the output of the `main` method, which prints the string representation of the person found: `Person(person_id=3, name=Каланта Фюна Рианнон, gender=Жен, birth=1218, death=1263, character=Королева Цинтры из рода Кербинов)`. Below this, it shows the output of the `DAOFactory` and `PersonDAO` classes, which return the same person object.

Если бесконечный поток информации о дебаге гиббернейта раздражает, его можно отключить, прописав пару строк в `application.properties`:

```
logging.level.org.springframework=OFF
logging.level.root=OFF
```


Список полезных ссылок

- [1] <https://habr.com/ru/post/29694/>
- [2] <https://javatutor.net/articles/j2ee-pattern-data-access-object>
- [3] <https://www.dokwork.ru/2014/02/daotalk.html>
- [4] <https://docs.jboss.org/hibernate/orm/3.5/reference/en/html/queryhql.html>