

Гайд по праку

v2.1

Субботин Даниил

29 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. Люди всё время меня спрашивают: ~~знаю ли я Тайлера Дёрдена?~~ где и в каких пакетах создавать файлы? Ответ: где угодно; либо как у меня на гитхабе(в таком случае все работает), либо как-то ещё, но тогда не гарантирую, что всё везде будет работать. Вообще, большая часть моего прака писалась путем изучения других проектов, так что приглашаю просто поизучать мой код на гитхабе, если есть желание.

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'a 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.

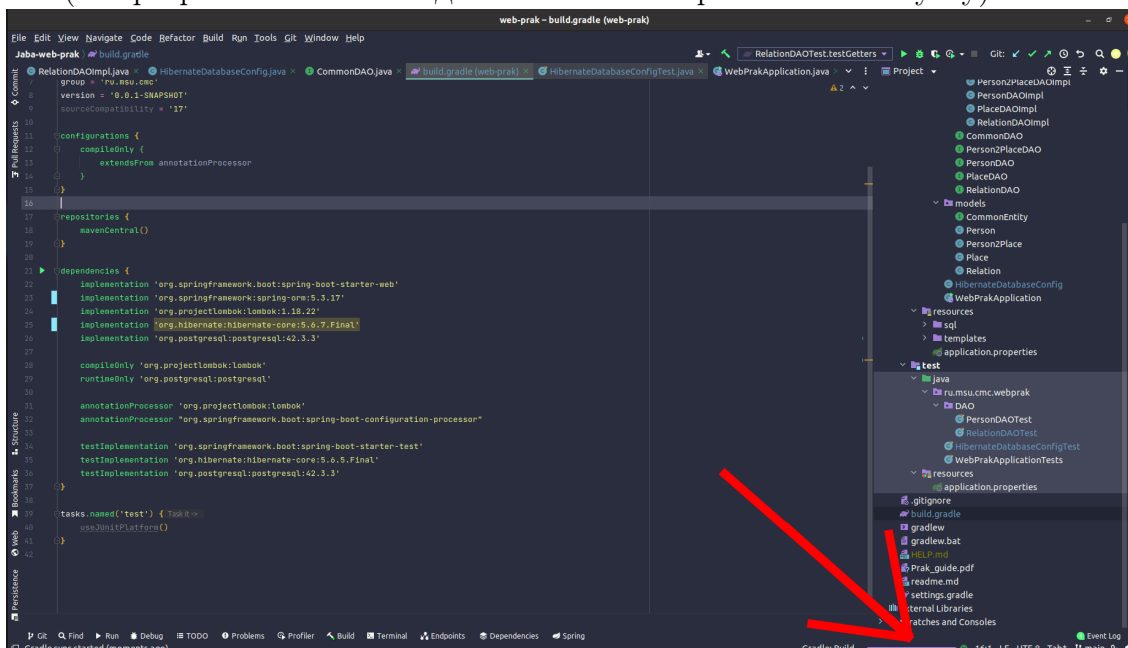
1.4 Создание проекта

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):

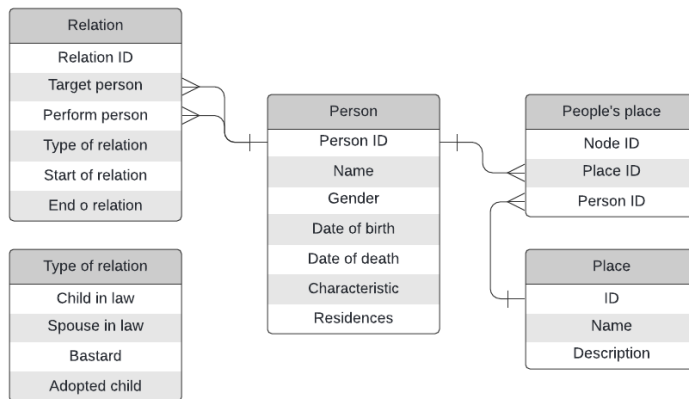
The screenshot shows the Spring Initializr web interface. On the left, under 'Project', 'Maven Project' is selected. Under 'Language', 'Java' is selected. The 'Spring Boot' section shows version '2.6.4' selected. The 'Project Metadata' section has 'Group' as 'ru.msu.cmc', 'Artifact' as 'webprk', 'Name' as 'webprk', 'Description' as 'Webprk', and 'Package name' as 'ru.msu.cmc.webprk'. The 'Packaging' is 'Jar' and 'Java' version is '17'. On the right, the 'Dependencies' section lists 'Lombok', 'PostgreSQL Driver', 'Spring Web', and 'Thymeleaf'. At the bottom, there are buttons for 'GENERATE', 'EXPLORE', and 'SHARE...'. The 'GENERATE' button has a tooltip that says 'CTRL + G'.

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)




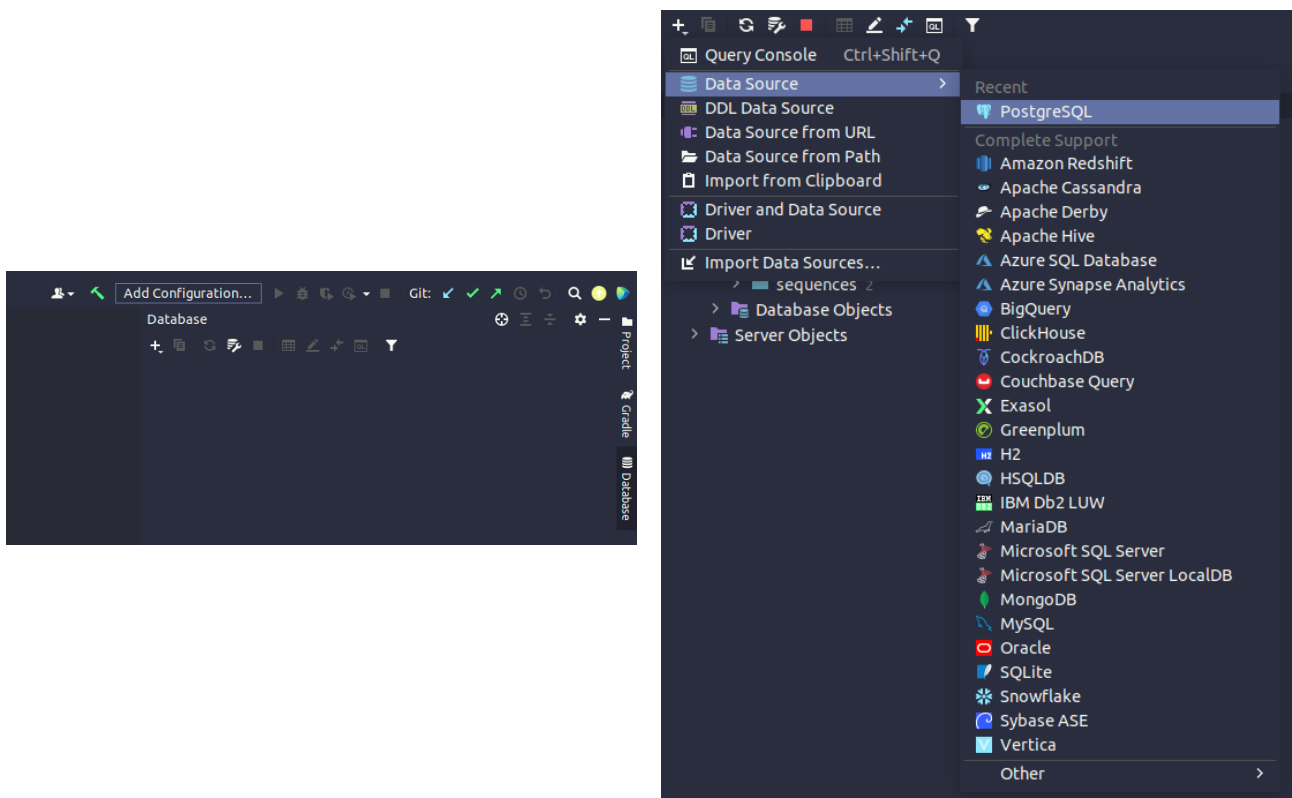
2 Классы и база данных

Схема моей БД для наглядности последующего кода и моих комментариев:

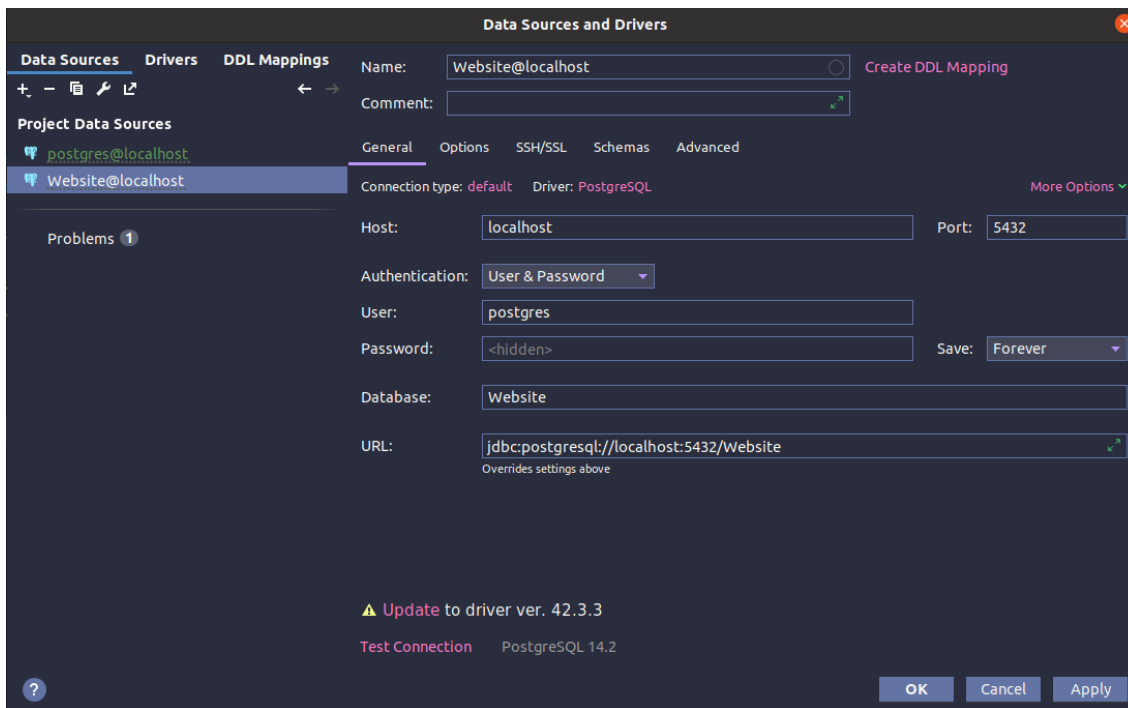


2.1 Создание БД

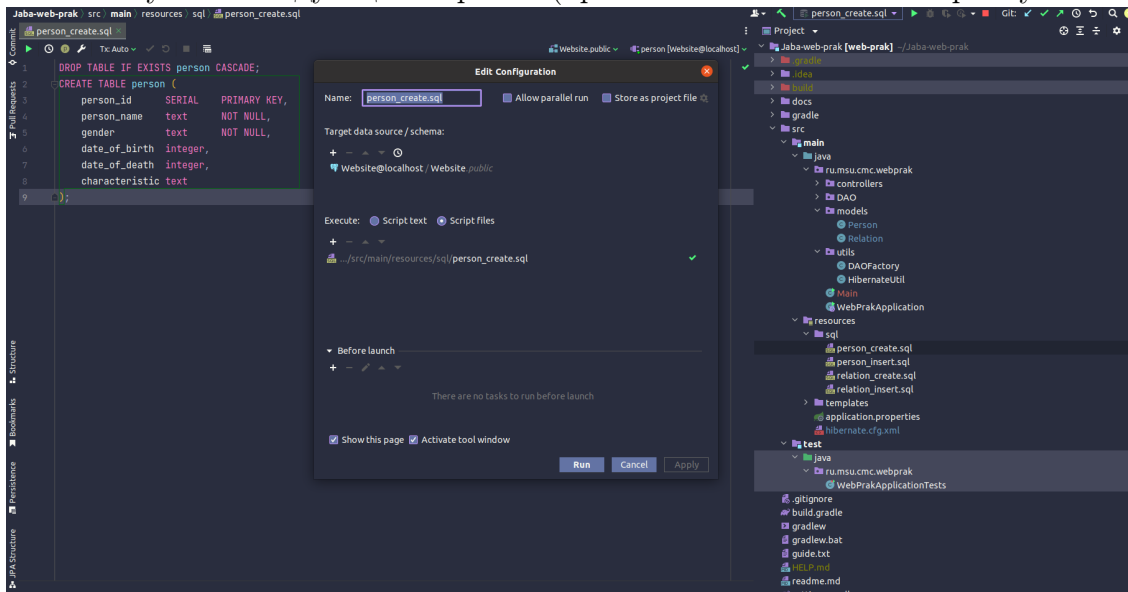
Создаём себе новую базу данных, используя, например, наш любимый PgAdmin. Сразу подключаем БД в идею через вкладку Database. Через  выбираем нужную СУБД во вкладке идеи Database:



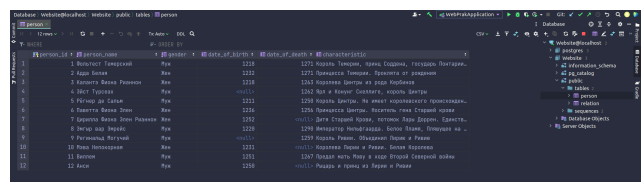
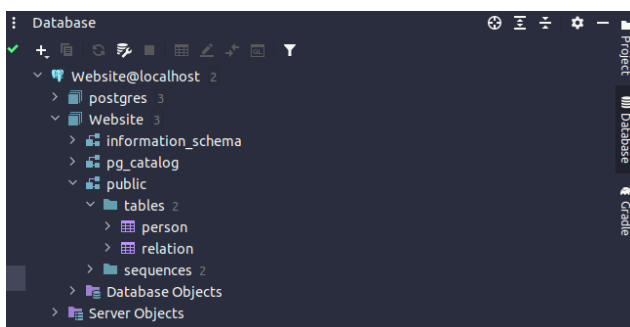
Вводим параметры для подключения (параметры, которые задавали при создании). При отсутствии пароля, как в моей случае, просто оставляем поле пустым:



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):

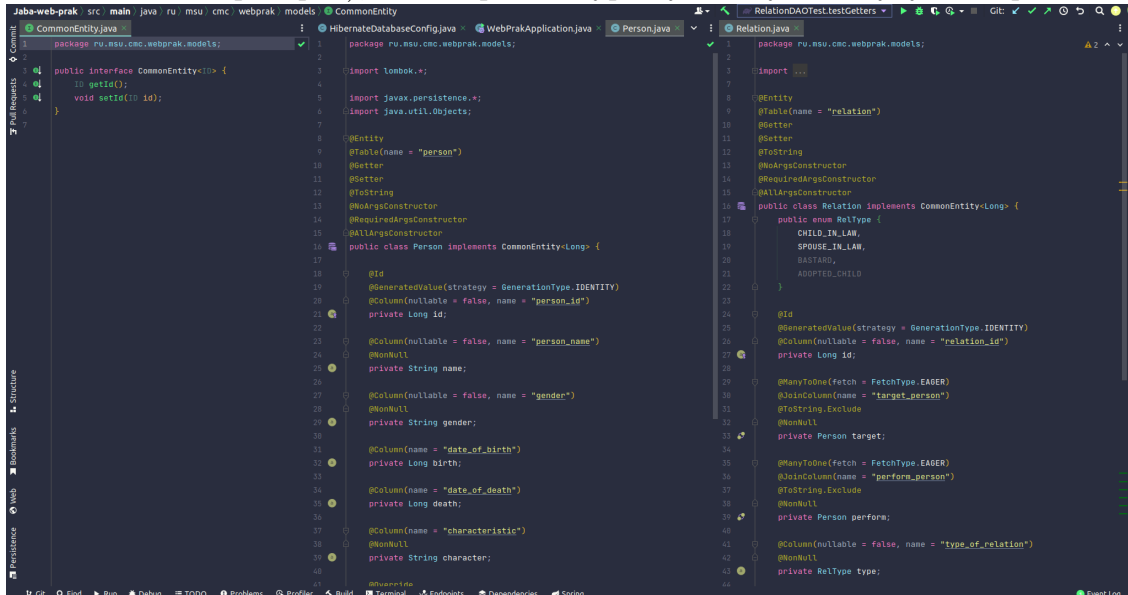


Через **+** выбираем только что подключенную базу, запускаем. Сразу же имеем возможность оценить все прелести современных IDE, просматривая и изменяя БД прямо из редактора:



2.2 Написание классов

Теперь на каждую созданную таблицу нужно написать класс, который будет с ней связан. Наличие класса не имеет значения, связь с таблицей реализуется не через название классов. Все декораторы будут описаны в следующем подпункте. Все типы, которые можно использовать в БД, реализованы и в Java, поэтому с определением типов полей проблем быть не должно. Я создал дженерик-интерфейс, от которого наследовал все свои классы-сущности (методы интерфейса `getId` и `setId` автоматически реализуются в классах с помощью декораторов), такая архитектура нужна будет в будущем при написании DAO:



2.3 Декораторы и маппинг классов

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. `@RequiredArgsConstructor` создаёт конструктор с `final` полями или полями, помеченными `@NonNull`. Если есть сомнения насчет структуры, можно нажать на вкладку `Structure` в идее и посмотреть состав своего класса.

2.3.2 Декораторы для полей

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]) Как видно из кода выше, одно (или несколько, не знаю, как будет вести себя гиббернейт 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 ломбоковского `@NonNull`. **ВАЖНО:** если поле может иметь `null`-значение, то для его описания необходимо использовать класс-обертку, а не примитивный 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 отвратительная интересная часть. Рядом с `WebPrakApplication.java` пишем файл `HibernateDatabaseConfig.java`. Это файл, который будет отвечать за подключение базы данных к проекту и создавать сессии запросов:

```
@Configuration
@PropertySource("classpath:application.properties")
public class HibernateDatabaseConfig {
    @Value("${org.postgresql.Driver}")
    private String DB_DRIVER;
    @Value("${jdbc:postgresql://localhost/website}")
    private String DB_URL;
    @Value("${postgres}")
    private String DB_USERNAME;
    @Value("${<empty>}")
    private String DB_PASSWORD;

    @Bean
    public LocalSessionFactoryBean sessionFactory() {
        LocalSessionFactoryBean sessionFactory = new LocalSessionFactoryBean();
        sessionFactory.setDataSource(oraDataSource());
        sessionFactory.setPackagesToScan("ru.msu.cmc.webprak.models");

        Properties hibernateProperties = new Properties();
        hibernateProperties.setProperty("hibernate.hbm2ddl.auto", "update");
        hibernateProperties.setProperty("hibernate.dialect", "org.hibernate.dialect.PostgreSQL10Dialect");
        hibernateProperties.setProperty("connection.pool_size", "1");

        sessionFactory.setHibernateProperties(hibernateProperties);

        return sessionFactory;
    }

    @Bean
    public DataSource oraDataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();

        dataSource.setDriverClassName(DB_DRIVER);
        dataSource.setUrl(DB_URL);
        dataSource.setUsername(DB_USERNAME);
        dataSource.setPassword(DB_PASSWORD);

        return dataSource;
    }
}
```

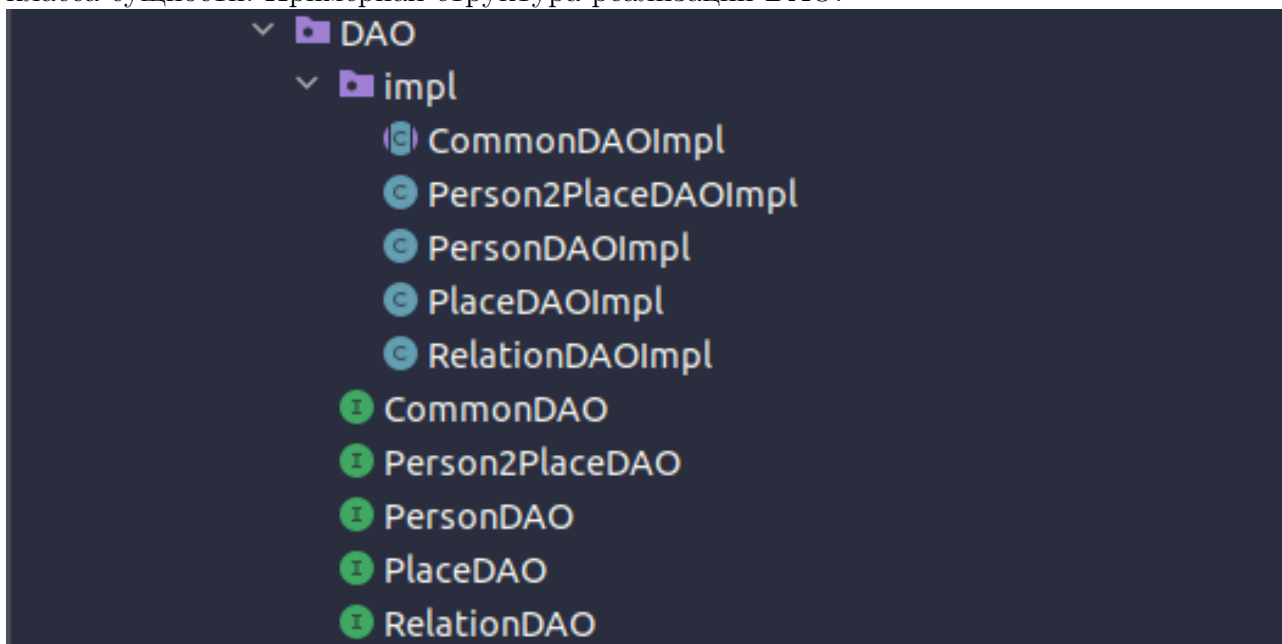
Поля, помеченные декоратором `@Value` получают значения из файла `application.properties`, который надо создать в папке `resources`:

```
application.properties
1 driver=org.postgresql.Driver
2 url=jdbc:postgresql://localhost/website
3 username=postgres
4 password=
5
6 logging.level.org.springframework=OFF
7 logging.level.root=OFF
```

`@PropertySource` задаёт путь к файлу свойств. Так вот, декоратор `@Value` считывает значения из этого файла и на их основе подключается к БД. Позже, при тестировании, создадим другой такой же файл, но в другом месте и с другими значениями переменных. Таким образом, с помощью одного конфигурационного файла и разных файлов `application.properties` можно управлять доступом к БД и, например, крутить тесты на тестовой БД, не затрагивая основную. Как видно из скринов выше, идея уже определила значения переменных и подставила их из файла.

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:



Я написал дженерик-интерфейс `CommonDAO`, в котором собраны общие методы для работы с сущностями, например `save`, `update`, `delete`, `getbyid`. Такая структура позволит обобщить эти методы и не писать в каждой реализации интерфейса эти методы заново:

```
public interface CommonDAO<T extends CommonEntity<ID>, ID> {  
    T getById(ID id);  
  
    Collection<T> getAll();  
  
    void save(T entity);  
  
    void saveCollection(Collection<T> entities);  
  
    void delete(T entity);  
  
    void update(T entity);  
}
```

В этом дженерике `T` заменится на название класса, а `ID` на тип `id` в этом классе.

3.1 Интерфейсы

Собственно, пишем сам интерфейс. Каждый интерфейс наследуем от `CommonDAO`. В DAO должны присутствовать типовые запросы приложения к БД, например все SQL команды (`insert`, `update`, `delete`), которые мы уже отнаследовали от базового интерфейса, а так же необходимые приложению запросы, которые зависят от конкретного задания:

```

public interface PersonDAO extends CommonDAO<Person, Long> {

    List<Person> getAllPersonByName(String personName);

    Person getSinglePersonByName(String personName);

}

```

3.2 ...перед реализацией

Напишем абстрактный класс, который будет реализовывать методы дженерик-интерфейса. На скрине ниже приведена структура этого класса:

```

@Repository
public abstract class CommonDAOImpl<T extends CommonEntity<ID>, ID extends Serializable> implements CommonDAO<T, ID> {

    protected SessionFactory sessionFactory;

    protected Class<T> persistentClass;

    public CommonDAOImpl(Class<T> entityClass) { this.persistentClass = entityClass; }

    @Autowired
    public void setSessionFactory(LocalSessionFactoryBean sessionFactory) {
        this.sessionFactory = sessionFactory.getObject();
    }

    @Override
    public T getById(ID id) {
        try (Session session = sessionFactory.openSession()) {
            return session.get(persistentClass, id);
        }
    }

    @Override
    public Collection<T> getAll() {
        try (Session session = sessionFactory.openSession()) {
            CriteriaQuery<T> criteriaQuery = session.getCriteriaBuilder().createQuery(persistentClass);
            criteriaQuery.from(persistentClass);
            return session.createQuery(criteriaQuery).getResultList();
        }
    }
}

```

Декоратор `@Repository` помечает этот класс как репозиторий. Поле `sessionFactory` и связанный с ним метод `setSessionFactory` позволяют устанавливать подключение к БД. Декоратор `@Autowired` задает поле конфигурации. При инициализации класса будет вызван метод `sessionFactory` из конфигурационного файла, который мы написали раньше. Таким образом, подключение будет создаваться автоматически и нет необходимости подключаться явно. Поле `persistentClass` хранит тип текущего класса и позволяет обобщенно обращаться к таблицам. Это поле заполняется непосредственно в классах-реализациях потомков, с помощью конструкторов.

Далее реализуем все обобщенные методы, которые мы описали в `CommonDAO`. Для манипуляций с БД необходимо создавать сессии, после взаимодействия закрывать их. Чтобы сделать код чище, можно воспользоваться т.н. "try-with-resources statement" (подробнее в [4]). Это позволяет не закрывать подключение явно. Пример использования есть на скрине выше.

3.3 Реализация

Сама реализация всех методов. Ничего особенного. Так же помечаем каждую реализацию с помощью `@Repository`. Наследуемся от абстрактного класса, имплементируя соответствующий интерфейс. Если производятся манипуляции с изменением БД, то используется стандартная система транзакций (начало-изменение-коммит). Если метод сложнее примитивного, то можно писать запросы на языке SQL, в спринге используется диалект HQL ([5]), но можно обойтись и без этого:


```

@Repository
public class RelationDAOImpl extends CommonDAOImpl<Relation, Long> implements RelationDAO {

    public RelationDAOImpl(){
        super(Relation.class);
    }

    @Override
    public List<Person> getPerformByRelType(Person person, Relation.RelType type) {
        if (type == Relation.RelType.SPOUSE_IN_LAW) {
            return getSpouse(person);
        }

        List<Person> res = new ArrayList<>();
        for (var relation : getRelation(type)) {
            if (Objects.equals(relation.getPerform().getId(), person.getId())) {
                res.add(relation.getPerform());
            }
        }
        return res;
    }
}

```

В конструкторе класса вызываем конструктор абстрактного класса, передавая ему в качестве параметра название класса. Таким образом, все абстрактные методы теперь будут работать и для этого класса. Схожим образом пишем все остальные реализации.

4 Тестирование и покрытие

4.1 Основные моменты

В модуле `test` пишем классы, которые будут отвечать за тестирование. Такие классы должны быть помечены декораторами `@SpringBootTest`:

```

@SpringBootTest
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
@TestPropertySource(locations="classpath:application.properties")
public class PersonDAOTest {

    @Autowired
    private PersonDAO personDAO;

    @Autowired
    private SessionFactory sessionFactory;
}

```

Декораторы `@Autowired` у приватных полей связывают эти переменные с соответствующими Bean'ами, то есть автоматически инициализируются своими "конструкторами" (это предложение абсолютно неверно с точки зрения терминологии, а лишь грубо передают основной смысл, позже добавлю краткую инфу, как это работает). Декоратор `@TestInstance` используется для настройки жизненного цикла экземпляров тестов. Если ОЧЕНЬ кратко (подробнее в [6]), то это позволит использовать декораторы по типу `@BeforeEach` и `@AfterEach`, которыми могут быть помечены функции, которые необходимо вызывать соответственно перед и после каждого теста. Например, это нужно, если тесты изменяют состояние БД и перед следующим тестом нужно заново заполнить БД валидными данными (этих декораторов больше, со всеми можно ознакомиться в идее и выбрать подходящий). Так же как и в `HibernateDatabaseConfig.java`, указываем `@TestPropertySource` и вместе с этим надо написать другой файл `application.properties`, уже в ресурсах модуля `test`, указав в настройках параметры для подключения к тестовой БД. Если вы используете `xml` конфиг, то для тестов можно просто поменять настройки прямо там. Вообще, использование тестовой БД необязательно - можно тестировать все на основной, то тогда в ходе тестов могут портиться данные или таблица будет забиваться мусором. Пример использования в моём проекте:

```

@BeforeEach
void beforeEach() {
    List<Person> personList = new ArrayList<>();
    personList.add(new Person(id: 123L, name: "Король Белогун", gender: "Муж", birth: null, death: 1245L, character: "Король Керака"));
    personList.add(new Person(id: null, name: "Илдико Брекл", gender: "Жен", birth: 1220L, death: null, character: "Чародейка-недоучка, интриганка"));
    personList.add(new Person(name: "Виаксас I", gender: "Муж", character: "Принц, позже - король Керака"));

    personList.add(new Person(name: "Альзур из Марибора", gender: "Муж", character: "Могущественный и известный чародей, ученик Косимо Маласпин"));
    personList.add(new Person(name: "Геральт из Ривии", gender: "Муж", character: "Легендарный ведьмак. Профессиональный охотник на монстров, о"));
    personList.add(new Person(id: null, name: "Йеннифер из Венгерберга", gender: "Жен", birth: 1173L, death: null, character: "Талантливая чародейка"));
    personDAO.saveCollection(personList);
}

@BeforeAll
@AfterEach
void annihilation() {
    try (Session session = sessionFactory.openSession()) {
        session.beginTransaction();
        session.createSQLQuery("truncate person restart identity cascade;").executeUpdate();
        session.createSQLQuery("alter sequence person_person_id_seq restart with 1;").executeUpdate();
        session.getTransaction().commit();
    }
}

```

Перед всеми тестами и после каждого теста, я полностью обнуляю свою БД, а перед каждым тестом заполняю ее нужными данными.

4.2 Тесты

Каждый тест должен быть помечен декоратором `@Test`. Тогда он будет распознан спрингом, и в идее слева появится зеленая стрелка, которая позволит запустить этот тест:

```

@Test
void testSimpleManipulations() {
    List<Person> personListAll = (List<Person>) personDAO.getAll();
    assertEquals("expected: 6, personListAll.size()", 6, personListAll.size());


    List<Person> geraltQuery = personDAO.getAllPersonByName(personName: "Геральт");
    assertEquals("expected: 1, geraltQuery.size()", 1, geraltQuery.size());
    assertEquals("expected: \"Геральт из Ривии\", geraltQuery.get(0).getName()", "Геральт из Ривии", geraltQuery.get(0).getName());

    Person personId3 = personDAO.getById(3L);
    assertEquals("expected: 3, personId3.getId()", 3, personId3.getId());

    Person personNotExist = personDAO.getById(100L);
    assertNull(personNotExist);
}

```

Чтобы тест действительно был тестом, помимо манипуляций с данными, нужны какие-то проверки. Для этого удобно пользоваться различными `assert`'ами. Например, `assertEquals` сравнивает два значения на идентичность, а `assertNull` проверяет выражение на `null`. Если `assert` вернул `false`, то возбуждается исключение и тест считается проваленным. Для запуска всех тестов сразу можно воспользоваться либо конфигом идеи, либо просто нажать

 Run Test Ctrl+Shift+F10

рядом с декларацией класса.

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.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html>
- [5] <https://docs.jboss.org/hibernate/orm/3.5/reference/en/html/queryhql.html>
- [6] <https://www.baeldung.com/junit-testinstance-annotation#test-instance>