Гайд по праку
v3.1

Субботин Даниил

25 июня 2023 г.

Автор не несет ответсвенности 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.

Содержание

1 Начало

1.1 Перед началом

В тексте содержится много достаточно очевидных для некоторых вещей, однако они могут не казаться таковыми для других людей. 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 репозиторий, если есть желание. Если вы читаете гайд с гитхаба, то рекомедую скачать его как pdf, тогда все ссылки должны быть кликабельными.

1.2 Сурцы

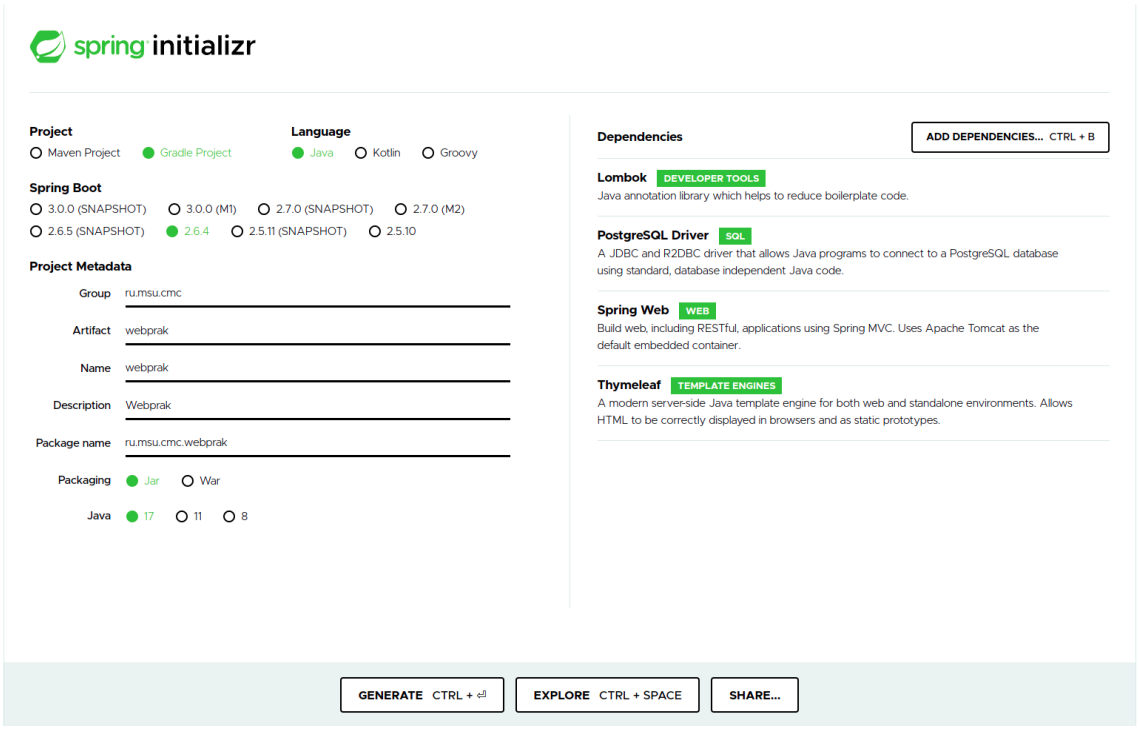
Github для просмотра всего кода: `https://github.com/WhiteWolfGeralt/Jaba-prak`

1.3 Выбор IDE

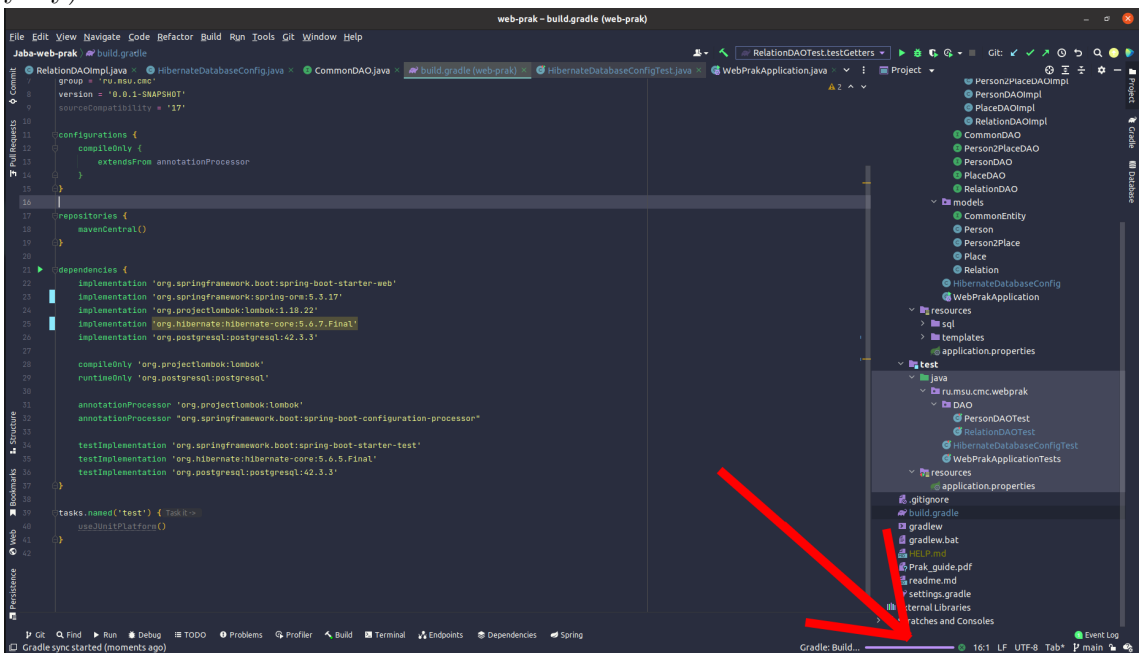
Прежде всего нужно удалить все богопротивные IDE и поставить единственно верную - IntelliJ IDEA от JetBrains (желательно Ultimate версию, т.к. в только в ней есть удобный интерфейс для работы с базами данных). Всё дальнейшее изложение будет продемонстрировано именно на примере ее (далее - просто "идея"). 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 сайта `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):

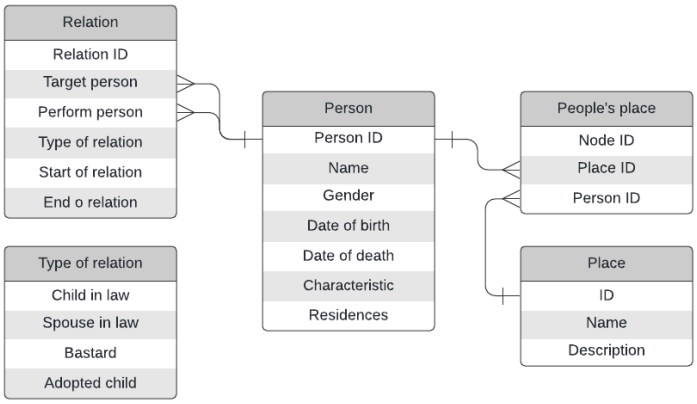


Генерируем, скачиваем, распаковываем, открываем проект через идею. В случае, если это первый опыт использования идеи, то для корректной работы необходимо установить SDK (собственно, идея сама предложит это сделать, если перейти в один из сурцов). Вручную можно настроить через Ctrl+Alt+Shift+S. После установки SDK ждём, пока идея проиндексирует весь проект (за прогрессом можно следить по шкале в правом нижнем углу)



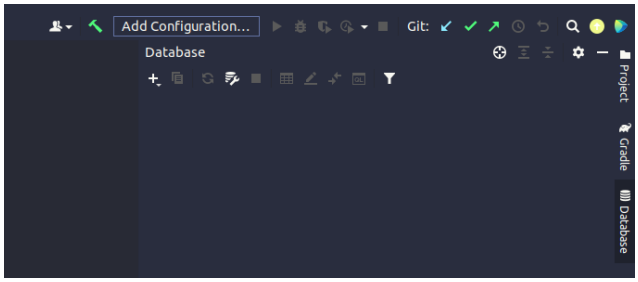
2 Классы и база данных

Схема моей БД для наглядности последующего кода и моих комментариев:

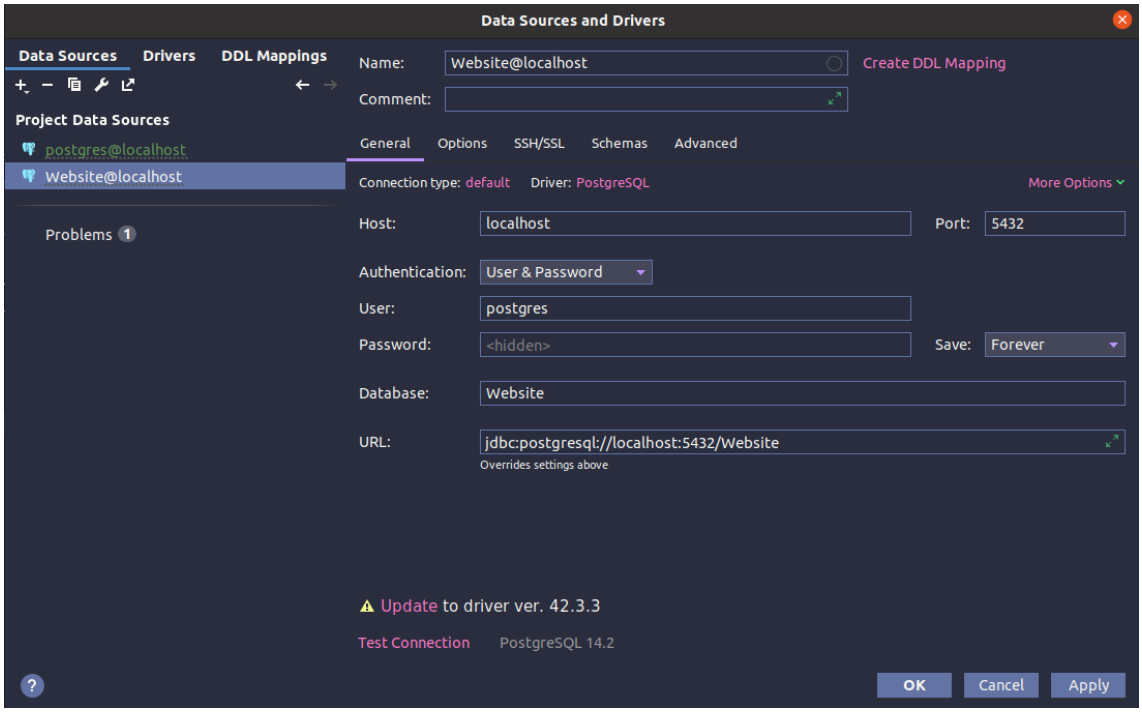


2.1 Создание БД

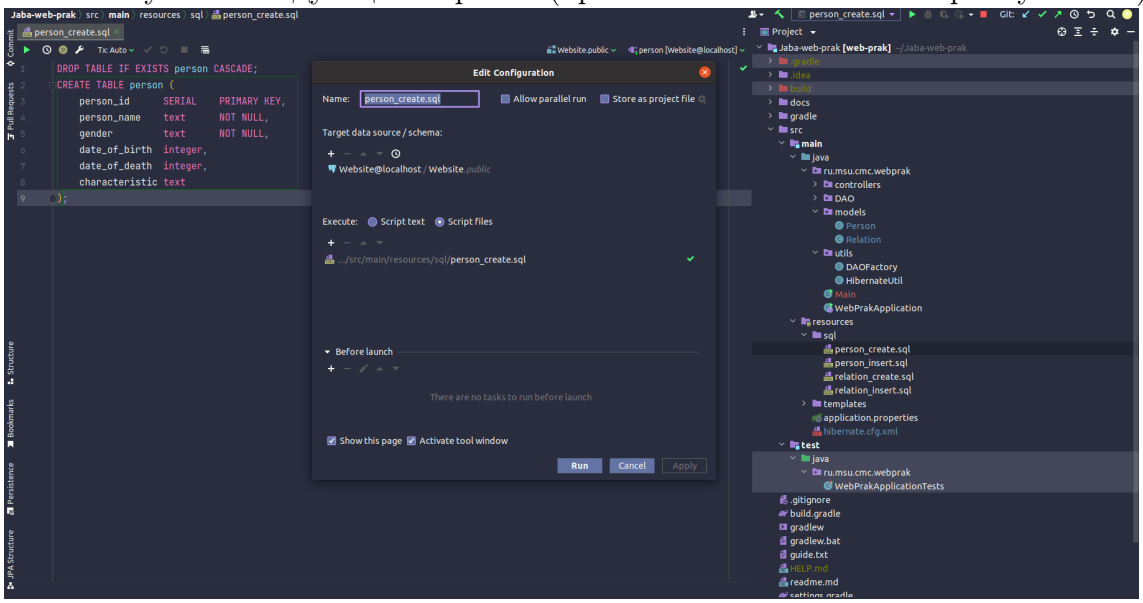
Создаём себе новую базу данных, используя, например, наш любимый PgAdmin. Сразу подключаем БД в идею через вкладку Database. Через ➕ выбираем нужную СУБД:



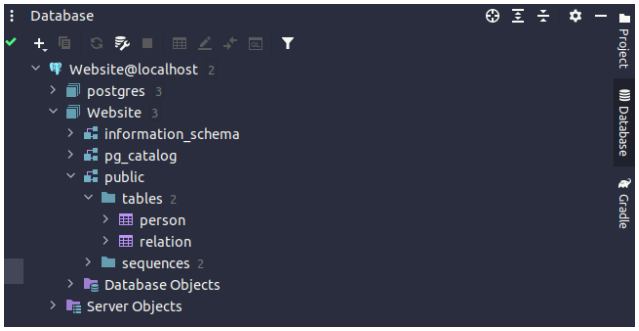
Вводим параметры для подключения (параметры, которые задавали при создании). При отсутствии пароля, как в моей случае, просто оставляем поле пустым:

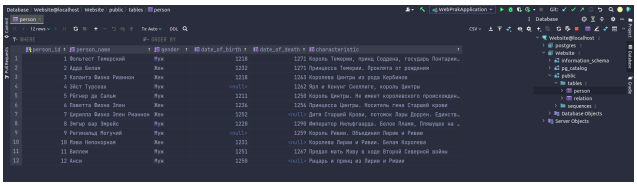


Далее, удобно в отдельной папке в проекте прописать все create и insert скрипты и отсюда же их запускать следующим образом (правая кнопка мыши по скрипту -> run):



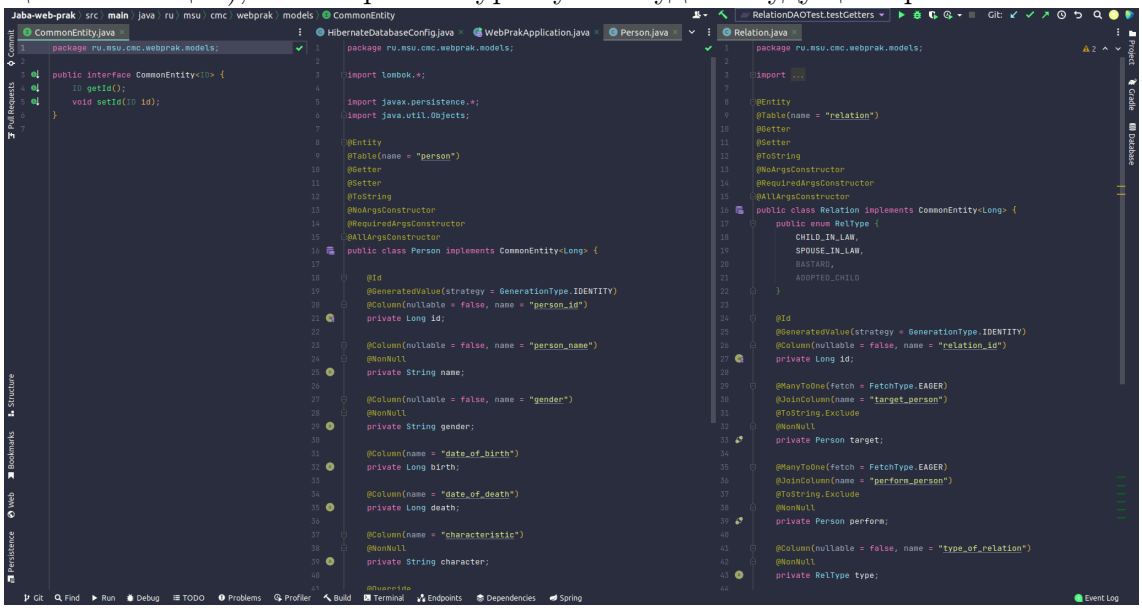
Через ✚ выбираем только что подключенную базу, запускаем. Сразу же имеем возможность оценить все прелести современных IDE, просматривая и изменяя БД прямо из редактора:





2.2 Написание классов

Теперь на каждую созданную таблицу нужно написать класс, который будет с ней связан. Навание класса не имеет значения, связь с таблицей реализуется не через название классов. Все аннотации 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 аннотаций), такая архитектура нужна будет в будущем при написании DAO:



2.3 Аннотации и маппинг классов

2.3.1 Аннотации для класса

Превращаем класс в сущность с помощью аннотации @Entity, аннотация @Table позволяет связать класс с конкретной таблицей в БД (здесь уже важно указать правильное имя таблицы). Далее идут аннотации 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 предствление 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 Аннотации для полей

Для маппинга классов пользоваться конфигурационными файлами hbm.xml не будем, связывание будет реализовано с помощью аннотаций. (Про связывание с помощью hbm.xml есть информация в [1]) Как видно из кода выше, одно (или несколько, не знаю, как будет вести себя гибернейт при этом) должно быть помечено аннотацией @Id, что, соответственно, означает первичный ключ в этом классе. С помощью аннотации @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 аннотацию @ManyToOne. Поле класса (таблицы) 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 аннотации и для других типов связей, например @ManyToMany, она 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:

```java
@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 аннотацией @Value получают значения из файла application.properties, который надо создать в папке resources:

application.properties

```
driver=org.postgresql.Driver
url=jdbc:postgresql://localhost/website
username=postgres
password=

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

@PropertySource задаёт путь к файлу свойств. Так вот, аннотация @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:

```java
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 интерфейсы. Каждый интерфес 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:

```java
public interface PersonDAO extends CommonDAO<Person, Long> {

    List<Person> getAllPersonByName(String personName);
    Person getSinglePersonByName(String personName);

}
```

3.2 ...перед реализацией

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

```java
@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();
        }
    }
```

Аннотация @Repository помечает этот класс как репозиторий. Поле sessionFactory и связанный с ним метод setSessionFactory позволяют устанавливать подключение к БД. Аннотация @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:

```java
@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 абстракные 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 аннотацией @SpringBootTest:

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

    @Autowired
    private PersonDAO personDAO;
    @Autowired
    private SessionFactory sessionFactory;
```

Аннотации @Autowired у приватных полей связывают эти переменные с соответствующими Bean'ами. Аннотация @TestInstance используется для настройки жизненного цикла экземпляров тестов. Если ОЧЕНЬ кратко(подробнее в [6]), то это позволит использовать аннотации 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 аннотаций 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: "Вираксас 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: "Йеннифэр из Венгерберга",  gender: "Жен",  birth: 1173L,  death: null,  character: "Талантливая чароде
    personDAO.saveCollection(personList);
}

@BeforeAll
@AfterEach
void annihilation() {
    try (Session session = sessionFactory.openSession()) {
        session.beginTransaction();
        session.createSQLQuery( queryString: "TRUNCATE person RESTART IDENTITY CASCADE;").executeUpdate();
        session.createSQLQuery( queryString: "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 аннотацией @Test. Тогда он будет распознан спрингом, и в идее слева появится зеленая стрелка, которая позволит запустить этот тест:

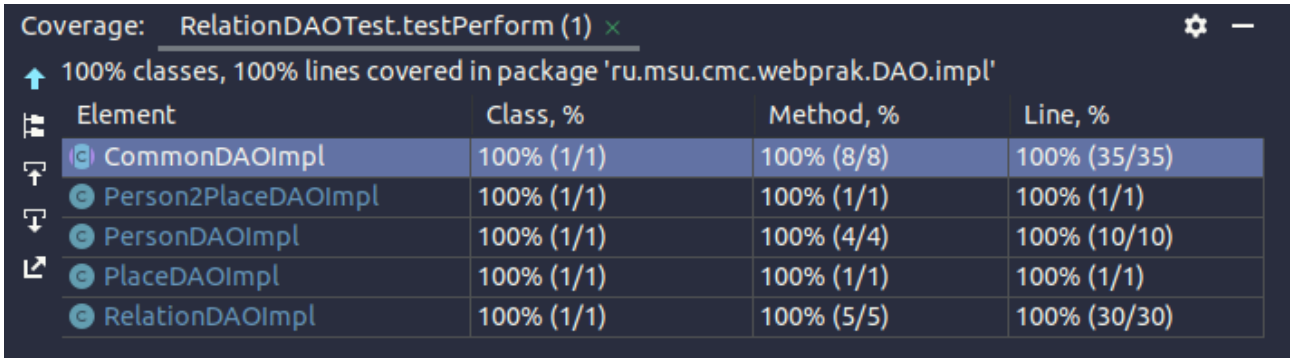
```
@Test
void testSimpleManipulations() {
    List<Person> personListAll = (List<Person>) personDAO.getAll();
    assertEquals( expected: 6, personListAll.size());

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

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

рядом с декларацией класса.

4.3 Покрытие

По-хорошему, покрытие надо собирать специально предназначенными для этого инструментами, но к этому моменту автор уже устал от этого прака, поэтому вся дальнейшая работа была сделана ~~на о~~... плохо. В данном случае сама идея уже предоставляет возможность собрать покрытие, для этого в запуске есть отдельная функция: "Run <TestName> with Coverage". Если тестовых файлов несколько, то идея предложит объединить результаты покрытия. Итоговый результат должен выглядеть так:



Для сдачи достаточно покрыть 80-90% методов в DAO. В качестве отчета был предоставлен этот скрин ввиду описанной выше причины.

5 Веб-страницы и контроллеры

Страницы будем писать с помощью BootStrap и Thymeleaf. Первый фреймворк поможет избежать ручного написания css и js, позволяя сконцентрироваться только на содержании. Второй же поможет избежать дублирования кода и позволит расширить функционал html. Обо всем - по порядку.

5.1 Структура

Каждая страница сайта - это отдельный html-файл (за некоторым исключением, когда для создания страницы используется шаблон страницы). Поэтому необходимо еще раз определиться с набором страниц и пересмотреть свою схему с первого этапа. Например, персональная страница человека, страница редактирования человека и страница добавления человека - это 3 разных страницы, и на каждую надо написать отдельный файл. Ещё необходима отдельная страница, на которую сайт будет перенаправлять после любой непредвиденной ошибки. Также нужно написать контроллеры - это java классы, которые непосредственно будут обрабатывать http-запросы (далее - просто запросы).

5.2 Контроллеры

Класс-контроллер должен быть помечен аннотацией @Controller. Не имеет значения, какой запрос в каком файле обрабатывается: можно как создать несколько классов-контроллеров, для каждой сущности, так и обрабатывать все запросы в одном файле. Аннотации функции RequestMapping используется для задания пути, который будет обрабатываться этой функцией, он задается аргументом value. Аргумент method задает тип запроса. Существует несколько видов запросов, вот здесь [8] можно прочитать подробно про каждый. В своем проекте я использовал только GET и POST. Соответственно, для этих типов существуют специализированные аннотации GetMapping и PostMapping, в которых метод указывать уже не нужно.

Контроллеры и html-страницы очень прочно связаны между собой. Поэтому, в качестве одного из аргументов функции, обрабатывающей запрос, может быть объект класса Model. Основное предназначение этого объекта - сохранить в себе некие атрибуты, которые потом могут быть использованы уже в html. Например, в примере снизу в джаве я сохраняю DAO для человека как атрибут personService, а затем в html обращаюсь к нему, как к обычному DAO:

```java
@GetMapping("/persons")
public String peopleListPage(Model model) {
    List<Person> people = (List<Person>) personDAO.getAll();
    model.addAttribute( attributeName: "people", people);
    model.addAttribute( attributeName: "personService", personDAO);
    model.addAttribute( attributeName: "relationService", relationDAO);
    return "persons";
}
```

```
    </a>
  </td>
  <td>
    <span th:text="${personService.getYearsOfLife(person)}"> </span>
  </td>
```

Синтаксис и семантика hmtl и связи с контроллерами будет описана в следующем подразделе. Функция-обработчик запроса всегда должна возвращать объект класса String - это название html-файла(без расширения), который должен выдаться при обрабатываемом запросе. В примере выше я обрабатываю запрос /persons, который должен отсылать на страницу со всеми людьми из базы данных. Соответственно, я возвращаю строку persons, поскольку мой файл persons.html описывает эту страницу. Названия файлов и реальных запросов могут не совпадать, но я рекомендую все же называть их одинаково, поскольку я потерял много время из-за того, что маппил не те файлы или не те запросы. В случае плохого маппинга идея не всегда подсвечивает ошибку, да и вообще в hmtl идея не охотно подсказывает, поэтому эту часть прака надо писать внимательно.

5.3 Код страниц

5.3.1 Про Thymeleaf

Чтобы всё, что я опишу дальше, работало, надо написать конфиг, который отвечает за работу Thymeleaf. В целом, код в нем достаточно простой, похожий на конфиг БД, поэтому я не буду его здесь объяснять и просто предлагаю взять его из моего проекта [7], или с официального сайта Thymeleaf(там есть примеры использования и конфига с подробным описанием. Можно писать и без шаблонизатора, в теории, это лишь незначительно усложнит разработку. Возможно, я использую Thymeleaf неправильно, но что поделать. Первое, что можно сделать с помощью этого шаблонизатора - создать так называемый компонент. Компоненты, помоченные атрибутом th:fragment, образно, становятся переменными, которые можно использовать в других файлах с указанием файла, в котором компонент определен и его именем. Наример в файле general.html я объявляю фрагмент site-footer, который задает нижнюю часть моих страниц. В таком случае, на каждой из новых страниц мне не нужно писать этот код заново, достаточно лишь вставить готовый шаблон:

```
<div th:fragment="site-footer" class="fixed-bottom indent">
    <p>
        Система управления генеалогической информацией
        <br>
        Субботин Даниил, 2022
        <br>
    </p>
</div>
```

Объявление фрагмента

```
<div th:replace="general :: site-footer"></div>
<div th:replace="general :: site-script"></div>
```

Использование фрагмента

Таким образом можно определить, например, все одинаковые элементы (такие как шапка сайта или нижний колонтитул) в одном файле, и во всех других просто вставлять их как фрагменты.

5.3.2 Про BootStrap

BootStrap уже содержит готовые шаблоны для кнопок, выпадающих списков и прочих полезных штук. Достаточно всего лишь подключить в свой проект эти шаблоны, указав в head ссылку на исходный код и некоторые другие атрибуты:

```
<head>
    <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css"
          integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T" crossorigin="anonymous">
```

Нужный код можно взять на официальном сайте BootStrap [9]. Кстати, есть неплохая документация полностью на русском языке. Во всем коде атрибуты class взяты именно из этих шаблонов.

5.3.3 HTML

Пример написания кода hmtl в связке с css и js лучше всего искать в официальной документации бутстрапа, здесь же я лишь опишу небольшой пример, который демонстрирует всё, что может понадобиться при создании страниц. Все теги легко гуглятся, поэтому я не буду описывать их смысл, сосредоточившись лишь на семантике. Оставляю ссылку на сайт с описанием всех тегов html: `https://www.w3schools.com/TAgs/default.asp`.

```
<div th:replace="general :: page-header"> </div>

<div class="indent">
    <div id="personInfo">
        <h4 th:text="${person.getName()}"></h4>
        <p th:if="${person.getName() ≠ null}" th:text="'Имя: ' + ${person.getName()}"></p>
        <p th:if="${person.getGender() ≠ null}" th:text="'Пол: ' + ${person.getGender()}"></p>
        <p th:if="${person.getBirth() ≠ null}" th:text="'Дата рождения: ' + ${person.getBirth()}"></p>
        <p th:if="${person.getDeath() ≠ null}" th:text="'Дата смерти: ' + ${person.getDeath()}"></p>
        <p th:if="${relationService.bornInMarriage(person) == false}" th:text="'Законнорожденный: нет'"></p>
        <p th:if="${relationService.bornInMarriage(person) == true}" th:text="'Законнорожденный: да'"></p>
        <p th:if="${person.getCharacter() ≠ null}" th:text="'Краткая характеристика: ' + ${person.getCharacter()}"></p>

        <p> Места проживания:  <span th:if="${person2placeService.getAllPersonsPlaces(person.getId()) == null}">нет</span>
            <a th:each="place, iter: ${person2placeService.getAllPersonsPlaces(person.getId())}" th:href="'/place?placeId=' + ${place.getId()}">
                <span th:text="${place.getName()} + ${!iter.last ? ',' : ''}"></span>
            </a>
        </p>

    </div>

    <!--edit delete order button group-->
    <div class="btn-toolbar" role="toolbar" aria-label="Toolbar with button groups">
        <div class="btn-group mr-2" role="group" aria-label="First group">
            <form style="..." method="get" action="/editPerson">
                <input type="hidden" name="personId" th:value="${person.getId()}"/>
                <button id="editButton" type="submit" class="btn btn-secondary">Редактировать информацию о человеке</button>
            </form>
            <form method="post" action="/removePerson">
                <input type="hidden" name="personId" th:value="${person.getId()}"/>
                <button id="deleteButton" type="submit" class="btn btn-secondary">Удалить человека из базы</button>
            </form>
        </div>
    </div>
    <!--end of button group-->

</div>
```

Это код страницы, которая отображает основную информацию о человеке.

- В начале используется "импортирование"шапки сайта с навигацией, которая объявлена в файле genaral как компонент page-header.

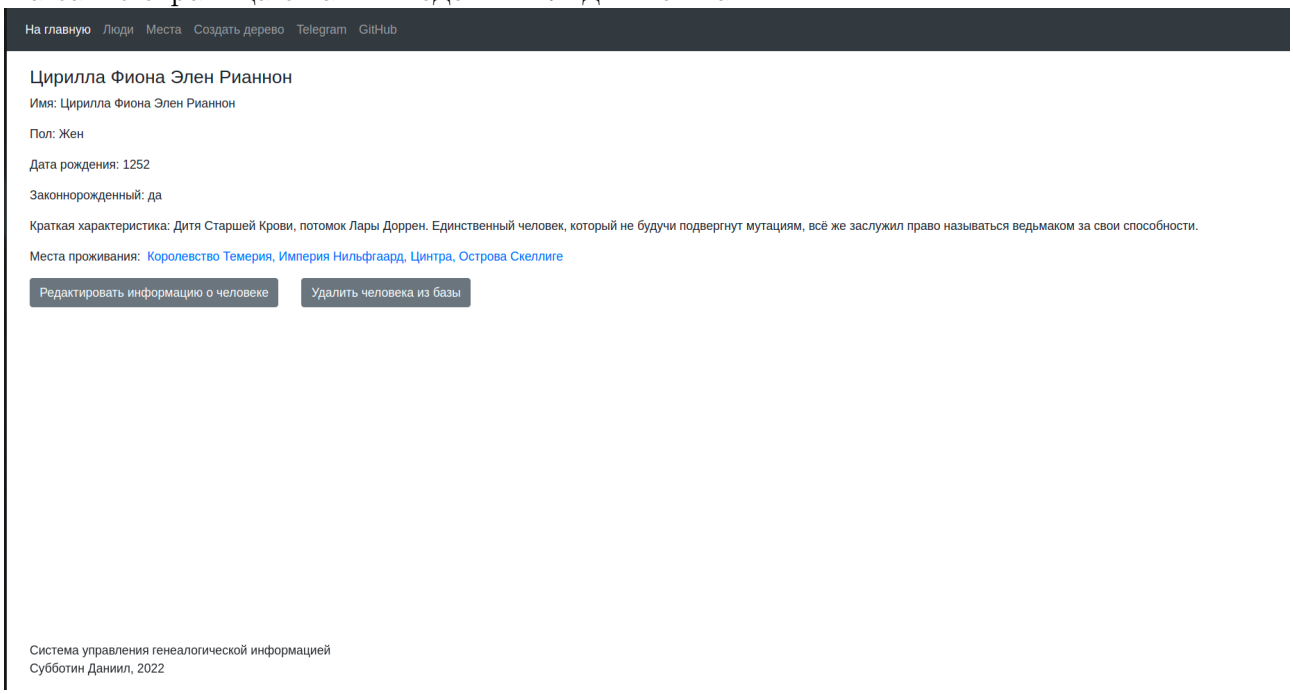
- Далее непосредственно выводится информация. Атрибут th:if позволяет задавать условные выражения. Все выражения, которые должны быть получены от атрибута модели. Здесь важно различать атрибуты тега и атрибуты модели, полученные из контроллера. Для получения атрибутов модели нужно использовать строки шаблонов: ${}. В них можно обращаться к атрибутам как к обычным переменным из джавы, следовательно доступны операции доступа к полям, вызовам методов, сравнение и

прочее. Атрибут th:text задает выводимый текст, поэтому первые строки кода в примере выводят всю заполненную информацию о человеке, пустые записи в БД просто пропускаются.

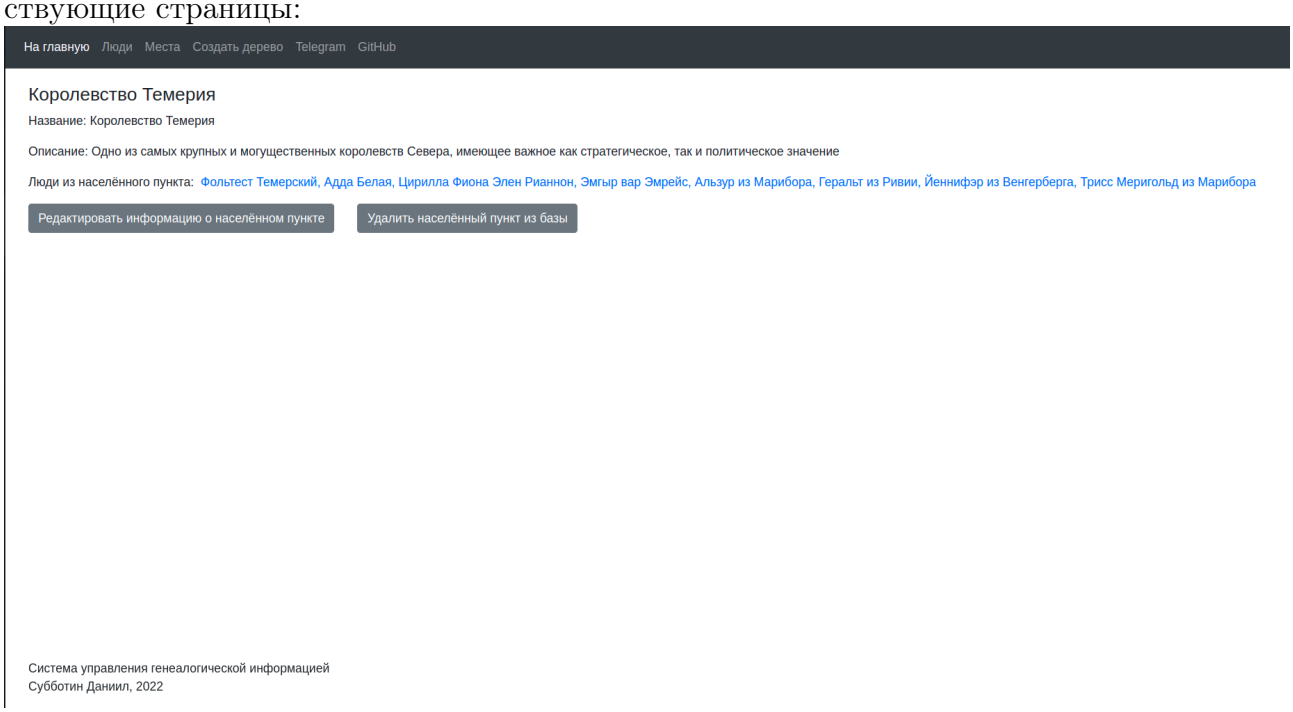
- Затем идёт вывод информации о местах проживания человека. Если в БД имеется нужная информация, то для вывода всех мест используется цикл, задаваемый ключевым словом th:each и итератором по id мест. Атрибут th:href для каждого места указывает, на какую страницу ведет ссылка, ассоциированная с выводимым местом. Подобный синтаксис используется и при выводе всех мест или людей, например, списком.

- В конце выводятся 2 кнопки, которые отвечают за редактирование и удаление человка из БД. Атрибут action задает действие, связанное с данной кнопкой. В данном случае он задаёт команду, которая будет обработана контроллером, который помечен аннотацией с заданным путем.

На сайте страница с таким кодом выглядит вот так:



Сверху кликабельная шапка, места проживания также кликабельны и ведут на соответствующие страницы:



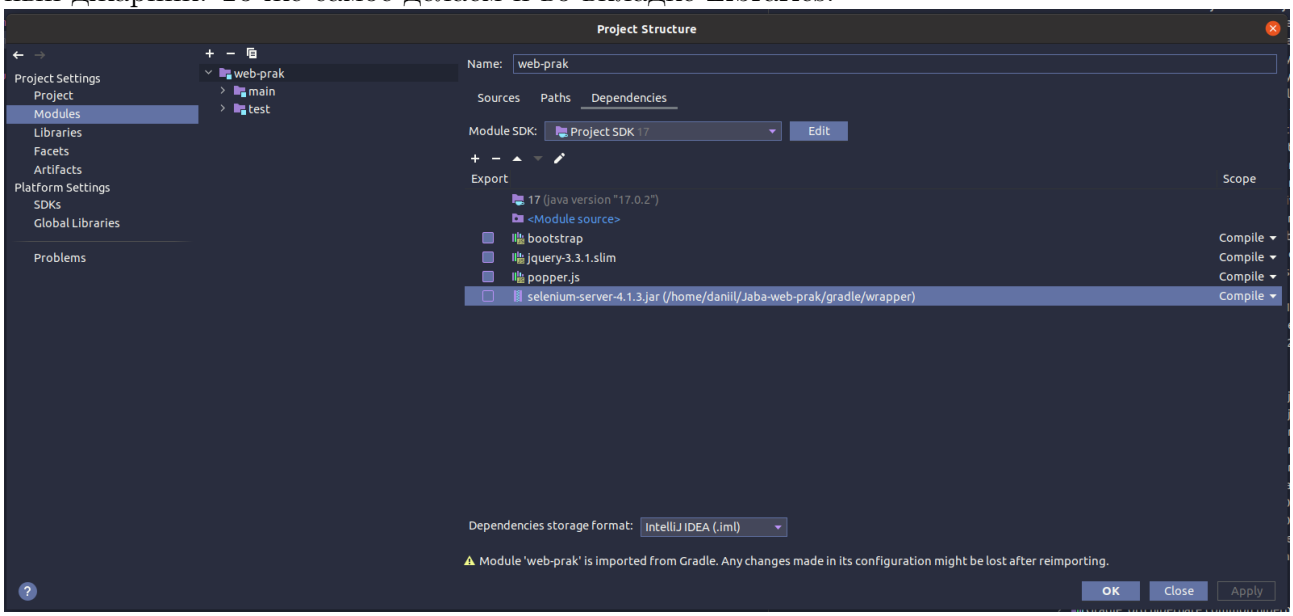
Страница с описанием места имеется абсолютно аналогичный код, с заменой лишь атрибутов модели.

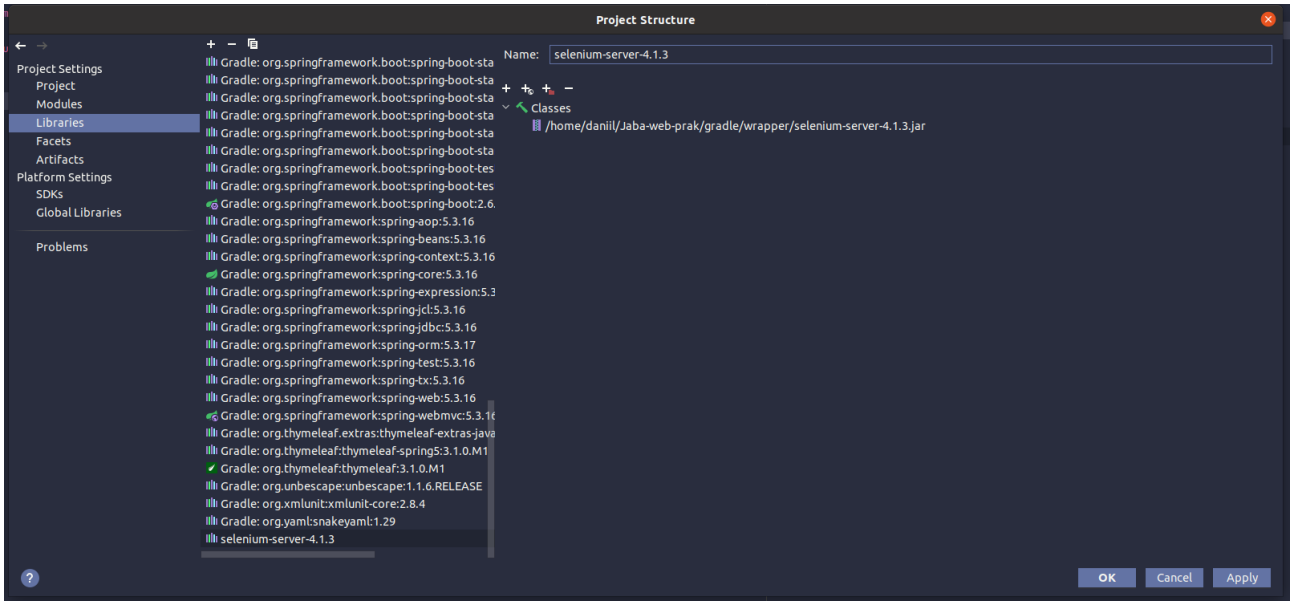
6 Тестирование с помощью Selenium

Selenium — это инструмент для автоматизации действий веб-браузера, то есть это утилита, которая позволяет создавать искуственные запросы к сайту, имитируя действия пользователя.

6.1 Установка

Все предыдущие инструменты так или иначе уже были заложены в фреймворк Spring, однако Selenium придется установить и подключить явно. У некоторых людей работало и без явной установки jar-ника, поэтому для начала можно просто попробовать подключить его в зависимости gradle. Если работает, то пропускаем этот этап, иначе скачиваем с официального сайта `https://www.selenium.dev/downloads/` jar-ник, помещаем его в удобное место (например, я положил в /gradle/wrapper). Далее, в идее переходим в настройки структуры проекта (через тулбар, либо через сочетание Ctrl+Alt+Shift+S). Во вкладке Modules выбираем свой модуль, затем в Dependencies через ➕ добавляем скачанный джарник. То же самое делаем и во вкладке Libraries:





Возможно, это будет выглядеть по другому, но смысл остается неизменным - нужно подключить Selenium к проекту. Для того, чтобы все подключилось, может потребоваться перезапуск идеи. Так же надо добавить зависимоть в build.gradle. Кроме того, нужно установить драйвер непосредственно в систему компьютера, например, с помощью команды

```
sudo apt-get install chromium-chromedriver
```

6.2 Написание тестов

Как и раньше, помечаем функцию тестирования аннотацией @Test. Вообще говоря, принцип всего тестирования можно уложить в одну картинку:

```
@Test
void HeaderTest() {
    ChromeDriver driver = new ChromeDriver();
    driver.manage().window().setPosition(new Point( x: 0, y: 0));
    driver.manage().window().setSize(new Dimension( width: 1024, height: 768));
    driver.get("http://localhost:8080/");

    WebElement peopleButton = driver.findElement(By.id("peopleListLink"));
    peopleButton.click();
    driver.manage().timeouts().implicitlyWait( time: 500, TimeUnit.MILLISECONDS);
    assertEquals(peopleTitle, driver.getTitle());
```

Это один тест, который показывает все нужные фичи селениума. Кратко расскажу про каждый.

- Создаем драйвер хромиума.
- Далее, при необходимости, можно настроить размер окна, который будет открываться драйвером (например, у меня открывалось недостаточно большое окно, чтобы корректно отображать все элементы).
- Далее, с помощью метода get открываем нужную для теста страницу.
- Предыдущие пункты были общими для всех тестов: создать драйвер, открыть страницу. Далее идёт непосредственно само тестирование функционирования сайта. В данном примере я ищу элемент с нужным id. Этот id я указывал в коде html страницы:
  ```
  <a class="nav-link" id="peopleListLink" th:href="@{persons}">Люди</a>
  ```
  Поскольку этот элемент является кликабельным, то я вызываю метод click у него.
- Теперь с помощью менеджера драйвера задаем небольшую задержку, чтобы выбранная страница успела загрузиться, и продолжаем работу с драйвером. В данном примере я просто проверяю заголовок страницы, чтобы убедиться, что кнопка, нажатая на предыдущем этапе, ведет на правильную страницу.

Конечно, у веб элементов есть множество других полезных функций, например заполнение полей текстом или чтение таблицы:

```
driver.findElement(By.id("placeName")).sendKeys( ...keysToSend: "Тестовое место");
driver.findElement(By.id("placeDescription")).sendKeys( ...keysToSend: "Тестовое описание");
driver.findElement(By.id("submitButton")).click();
driver.manage().timeouts().implicitlyWait( time: 500, TimeUnit.MILLISECONDS);

assertEquals(placeTitle, driver.getTitle());
WebElement placeInfo = driver.findElement(By.id("placeInfo"));
List<WebElement> cells = placeInfo.findElements(By.tagName("p"));

assertEquals(cells.get(0).getText(),  actual: "Название: Тестовое место");
assertEquals(cells.get(1).getText(),  actual: "Описание: Тестовое описание");
```

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

[7] https://github.com/WhiteWolfGeralt/Jaba-prak/blob/main/src/main/java/ru/msu/cmc/webprak/configs/WebConfig.java

[8] https://developer.mozilla.org/ru/docs/Web/HTTP/Methods

[9] https://bootstrap-4.ru/docs/5.1/getting-started/introduction/