Министерство науки и высшего образования Российской Федерации

федеральное государственное автономное   
образовательное учреждение высшего образования   
«Самарский национальный исследовательский университет   
имени академика С.П. Королева»

Институт информатики и кибернетики

Кафедра технической кибернетики

**Краткая документация по проекту   
«FluffyTailClinic»**

Выполнили:

Рыбалко Ф. А. 6305-010302D

Мурзина А. Е. 6305-010302D

Самара, 2025

**Задачи**

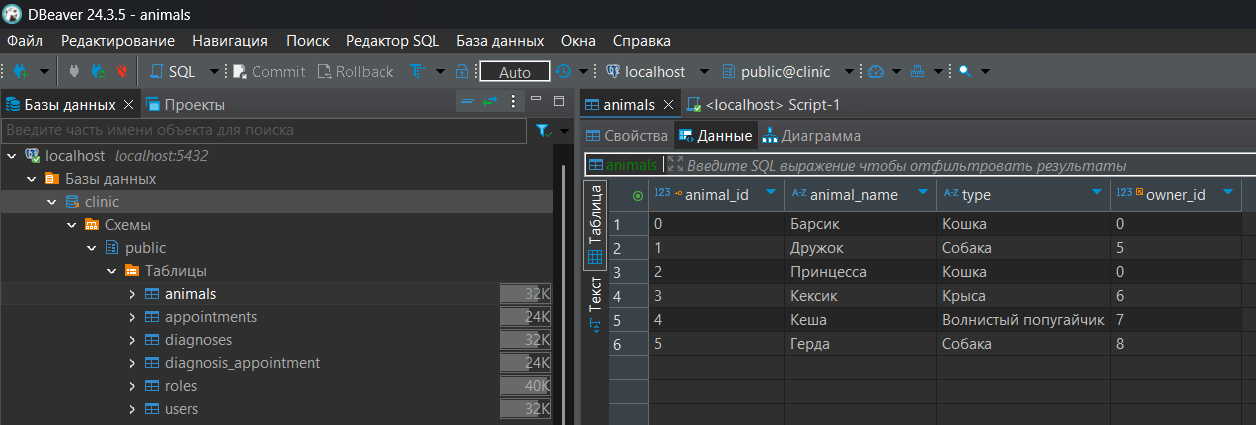
1. **Настройка базы данных**:

* **Развернуть PostgreSQL в Docker;**

Для развертывания PostgreSQL в Docker на сервере с Ubuntu через PuTTY был выполнен ряд шагов. Сначала произведено подключение к серверу Ubuntu с использованием PuTTY, где был введен IP-адрес сервера и выполнена аутентификация через SSH. После успешного подключения на сервере была установлена среда Docker: обновлены пакеты системы, установлены необходимые зависимости, добавлен GPG-ключ Docker и его репозиторий, а затем выполнена установка Docker. После установки Docker была создана директория для хранения данных PostgreSQL, чтобы обеспечить их сохранность после остановки контейнера. Затем был запущен контейнер PostgreSQL с использованием Docker, где указаны параметры для создания пользователя, пароля и базы данных, а также настроено монтирование директории для хранения данных и проброс порта 5432 для доступа к базе данных. Для проверки работоспособности контейнера использовалась команда `docker ps`, которая отобразила запущенный контейнер с именем `postgres`. Для подключения к базе данных был установлен клиент PostgreSQL, после чего выполнено подключение к базе данных с использованием команды `psql`. Для обеспечения удаленного доступа к PostgreSQL были отредактированы конфигурационные файлы `pg\_hba.conf` и `postgresql.conf`, чтобы разрешить подключение с любого IP-адреса, и контейнер был перезапущен. В завершение была проверена возможность подключения к базе данных с другого компьютера с использованием клиента PostgreSQL, что подтвердило корректность настройки и доступность базы данных.

* **Подключиться к БД через DBeaver.**

Для подключения к базе данных PostgreSQL через DBeaver был выполнен следующий процесс. Сначала в DBeaver создано новое соединение с базой данных, где был выбран тип СУБД — PostgreSQL. В настройках соединения указаны параметры: хост (IP-адрес сервера), порт (5432 по умолчанию), имя базы данных, а также логин и пароль пользователя (clinic), которые были заданы при запуске контейнера PostgreSQL в Docker. После заполнения всех необходимых полей выполнено тестирование соединения, чтобы убедиться в корректности настроек. После успешного теста соединение было сохранено, и в DBeaver отобразилась структура базы данных, включая таблицы и схемы. Это позволило выполнять SQL-запросы, просматривать и редактировать данные, а также управлять объектами базы данных через удобный графический интерфейс DBeaver.



1. **Разработка ORM-моделей:**

* **Определить сущности и связи между ними;**

При разработке ORM-моделей сущности и связи между ними были определены на основе анализа предметной области и структуры базы данных. Каждая сущность (например, `User`, `Animal`, `Role`, `Appointment`, `Diagnosis`, `DiagnosisAppointment`) была представлена в виде класса, где поля класса соответствовали столбцам таблиц в базе данных. Аннотации, такие как `@Entity`, `@Table`, `@Id`, `@GeneratedValue`, использовались для указания, что класс является сущностью, а его поля — атрибутами таблицы. Связи между сущностями (один-ко-многим, многие-ко-многим) были описаны с помощью аннотаций, таких как `@OneToMany`, `@ManyToOne`, `@ManyToMany`, и `@JoinColumn`, которые определяют, как сущности связаны между собой на уровне базы данных. Например, связь между `User` и `Animal` была определена как один-ко-многим, так как один пользователь может иметь несколько животных, а связь между `Appointment` и `Diagnosis` через промежуточную таблицу `DiagnosisAppointment` — как многие-ко-многим. Таким образом, ORM-модели точно отражают структуру базы данных и связи между таблицами, что позволяет удобно работать с данными на уровне объектов в коде.

* **Реализовать модели с использованием SQLAlchemy/Django ORM** (Hibernate);

Для реализации моделей с использованием Hibernate был выполнен следующий процесс. Каждая сущность (например, `User`, `Animal`, `Role`, `Appointment`, `Diagnosis`, `DiagnosisAppointment`) была описана в виде Java-класса с аннотациями Hibernate, такими как `@Entity`, `@Table`, `@Id`, `@GeneratedValue`, которые указывают, что класс является сущностью и соответствует таблице в базе данных. Поля классов были аннотированы как `@Column`, чтобы связать их с соответствующими столбцами таблиц. Связи между сущностями (один-ко-многим, многие-ко-многим) были реализованы с помощью аннотаций `@OneToMany`, `@ManyToOne`, `@ManyToMany` и `@JoinColumn`, которые определяют, как сущности связаны между собой на уровне базы данных. Например, связь между `User` и `Animal` была описана как один-ко-многим, а связь между `Appointment` и `Diagnosis` через промежуточную таблицу `DiagnosisAppointment` — как многие-ко-многим. Для работы с базой данных был настроен файл `hibernate.cfg.xml`, где указаны параметры подключения к PostgreSQL, диалект базы данных и автоматическое создание/обновление таблиц. В коде использовался `SessionFactory` для создания сессий и выполнения операций с базой данных, таких как сохранение, обновление, удаление и выборка данных. Таким образом, Hibernate позволил удобно работать с базой данных через объекты Java, автоматически генерируя SQL-запросы и управляя связями между сущностями.

* Настроить миграции.

Для настройки миграций с помощью Hibernate в Spring Boot был использован механизм автоматического управления схемой базы данных через параметр `spring.jpa.hibernate.ddl-auto`. В файле конфигурации `application.properties` было указано значение `update`, что позволяет Hibernate автоматически создавать или обновлять таблицы в базе данных на основе описанных сущностей. Это означает, что при запуске приложения Hibernate проверяет структуру базы данных и вносит изменения, если сущности были изменены (например, добавлены новые поля или таблицы). Для более сложных сценариев миграций можно использовать инструменты, такие как Flyway или Liquibase, которые позволяют управлять изменениями базы данных через SQL-скрипты или конфигурационные файлы, обеспечивая контроль версий и возможность отката изменений. В нашем случае Hibernate автоматически обрабатывал миграции, что упростило разработку на ранних этапах проекта.

1. **Разработка модели пользователя:**

* **Определить структуру модели (email, пароль, имя и т. д.);**

Структура модели пользователя была определена на основе требований к данным, которые необходимо хранить и обрабатывать в системе. Класс `User` был создан как сущность с аннотацией `@Entity`, что указывает на его соответствие таблице в базе данных. Поля класса, такие как `email`, `password`, `name`, `phone` и `description`, были аннотированы с помощью `@Column`, чтобы связать их с соответствующими столбцами таблицы. Поле `id` было помечено аннотациями `@Id` и `@GeneratedValue`, чтобы указать, что это первичный ключ с автоматической генерацией значения. Для обеспечения связи с другими сущностями, например, с ролью пользователя (`Role`), было добавлено поле `role` с аннотацией `@ManyToOne`, что отражает отношение "многие-к-одному". Таким образом, модель пользователя включает основные атрибуты, такие как уникальный идентификатор, имя, пароль, email, телефон и описание, а также связи с другими сущностями, что позволяет эффективно управлять данными пользователя в системе.

* **Настроить хеширование паролей.**

Хеширование паролей было настроено с использованием библиотеки `BCryptPasswordEncoder` из Spring Security. В коде создан экземпляр `BCryptPasswordEncoder`, который используется для преобразования исходного пароля (например, `"562"`) в хешированное значение. Хешированный пароль был получен с помощью метода `encoder.encode(rawPassword)` и выведен в консоль. Для сохранения хешированного пароля в базе данных PostgreSQL был выполнен SQL-запрос `INSERT` через psql, где хешированное значение было вставлено в соответствующее поле таблицы пользователей. Например:

INSERT INTO users (username, password, email) VALUES ('user1', '$2a$10$exampleHashedPassword', 'user1@example.com');

Таким образом, пароль хранится в базе данных в безопасном хешированном виде, что обеспечивает защиту данных пользователей.

1. **Функционал для работы с моделями:**

* **Написать скрипты для заполнения базы данных тестовыми данными;**

Для создания структуры базы данных и последующего заполнения тестовыми данными был использован следующий подход. Сначала структура базы данных была автоматически создана при запуске приложения с использованием Hibernate. Это было сделано через параметр `spring.jpa.hibernate.ddl-auto=update` в конфигурации Spring Boot, который позволяет Hibernate генерировать таблицы на основе Java-моделей (`@Entity`). После этого был создан dump-файл, содержащий только структуру базы данных (без тестовых данных), с помощью команды `pg\_dump`:

pg\_dump -U your\_username -d fluffytailclinic --schema-only -f path/to/structure\_dump.sql

Этот dump-файл содержал SQL-команды для создания таблиц, индексов и связей, но без данных. Для заполнения базы тестовыми данными был написан отдельный SQL-скрипт, содержащий команды `INSERT`, который выполнялся после восстановления структуры из dump-файла. Таким образом, структура базы данных была создана через Hibernate, а тестовые данные добавлены вручную через SQL-скрипты.

* **Реализация функционала для работы с данными в соответствии с тематикой выбранного приложения.**

Для реализации CRUD (Create, Read, Update, Delete) операций для каждого класса (`User`, `Animal`, `Role`, `Appointment`, `Diagnosis`, `DiagnosisAppointment`) был использован Spring Data JPA в сочетании с Hibernate. Для каждого класса создан репозиторий, расширяющий `JpaRepository`, что предоставило готовые методы для работы с базой данных, такие как `save`, `findById`, `findAll`, `deleteById`. В сервисном слое (`Service`) были реализованы методы, которые вызывают соответствующие методы репозитория и добавляют необходимую бизнес-логику. В контроллерах (`Controller`) были созданы REST-эндпоинты для обработки HTTP-запросов (GET, POST, PUT, DELETE), которые вызывают методы сервисов. Например, для класса `User`:

- Create: `POST /users` — создание нового пользователя через `userRepository.save(user)`.

- Read: `GET /users` — получение списка всех пользователей через `userRepository.findAll()`.

- Update: `PUT /users/{id}` — обновление данных пользователя через `userRepository.save(existingUser)`.

- Delete: `DELETE /users/{id}` — удаление пользователя через `userRepository.deleteById(id)`.

Аналогичная структура была реализована для остальных классов, что обеспечило полный набор CRUD операций для каждой сущности.

1. **Оформление отчета:**

* Описать структуру базы данных;
* Подготовить скрипты создания таблиц и миграции;
* Создать презентацию с результатами работы.