Geekbrains

**Разработка веб-приложения с ведением и хранением рецептур для производственных предприятий на основе Java Spring Boot**

Программа: Разработчик

Специализация: Программист

Бесхмельницына Дарья Александровна

Протвино

2024

**Содержание**

**1. Введение**……………………………………………………1

1.1 Актуальность проекта…………………………………….1

1.2 Цели и задачи………………………………………..…….2

1.3 Инструменты разработки…………………………………3

**2. Проектная стадия**………………………………………….5

2.1 Выбор шаблона проектирования…………………………5

2.2 Проектирование архитектуры базы данных……………..5

2.3 Инструменты тестирования……………………………….7

**3. Практическая часть**……………………………………….8

3.1 Реализация дизайна и функционала веб-приложения……………………………………………………..8

3.2 Создание классов-сущностей…………………………….12

3.3 Создание классов-сервисов………………………………22

3.4 Создание классов-репозиториев……………………...….26

3.5 Создание классов-контроллеров…………………………30

3.6 Создание классов-конфигов……………………………...35

3.7 Подключение базы данных MySQL……………………..39

3.8 Разворачивание Docker контейнера……………………...41

3.9 Тестирование……………………………………………...47

3.10 Руководство по скачиванию и запуску………………...49

**Заключение**…………………………………………………...51

**Список используемой литературы**………………………..51

**1. Введение**

**1.1 Актуальность проекта**

В настоящее время промышленные предприятия все чаще нуждаются в индивидуально созданных, эффективных и доступных для быстрого освоения инструментах реализации производственной деятельности.

В процессе работы в фирме по производству косметики, возникло желание создать продукт, который способен содержать в себе рецептуры по производству косметических продуктов, и так же информацию о компонентах участвующих в приготовлении того или иного средства.

Вдобавок данное приложение можно будет использовать на конвейерном производстве, где будет осуществляться приготовление продукта в большом объеме, где живым людям затруднительно обойтись без помощи машин.

Соответственно, актуальность данной работы заключается в создании сервиса для хранения важных данных, что в том числе поможет отделу ответственному за поставку сырья, а так же эксплуатации системы на производстве для повышения эффективности работы предприятия.

**1.2 Цели и задачи**

Целью данного проекта является создание веб-приложения, которое поможет эффективно поддерживать работу производства. Для достижения данной задачи нужно учесть некоторые требования и выполнить следующие задачи:

1. **Функциональные требования:**

* **Права пользователей:** Пользователи будут иметь доступ к рецептурам и перечню сырья, по предварительному внесению в базу данных. В дополнение, у работников будет возможность вносить изменения в рецептуры и сырье, а так же создавать новые записи.
* **Авторизация:** Вход в систему будет осуществляться по индивидуальному логину и паролю, которые предотвратят несанкционное вмешательство.

1. **Безопасность:**

* **Авторизация:** Вход в систему будет осуществляться по индивидуальному логину и паролю, которые предотвратят несанкционное вмешательство.
* **Хранение паролей:** Данная функция будет осуществляться через базу данных MySQL, пользователи и их пароли заранее будут внесены в нее.

1. **Производительность:** Функционал приложения должен позволять пользователям легко освоить данный сервис и вместе с тем, должен быстро откликаться на их действия, поддерживая комфорт и эффективность работы с ним. Вместе с тем, необходимо реализовать быстрый доступ к данным, для оперативного взаимодействия в системе. Чем проще будет интерфейс, тем быстрее и понятнее будет происходить работа с ним.
2. **Адаптивность:** Приложение должно быть написано так, чтобы его было легко дополнить новым функционалом, в связи с ростом потребностей предприятия.

Следуя поставленным задачам и общепринятым требованиям, получится создать эргономичное и высокоэффективное веб-приложение с ведением и хранением рецептур для производственных предприятий.

**1.3 Инструменты разработки**

Для реализации проекта были задействованы следующие инструменты:

**1. Язык программирования Java:** Данный язык выбран за счет широкого выбора встроенных библиотек и инструментов,в проекте используется для написания логики сервера из-за своей надежности и возможности интеграции с другими технологиями.

**2. Среда разработки IntelliJ IDEA:** Является одной из самых сильных IDE с большим количеством библиотек и инструментов разработки.

**3. Платформа разработки Spring Boot:** Широко распространенный фреймворк для разработки Java-приложений из-за удобства работы как с небольшими приложениями, так и с крупными системами. Так же стоит отметить, что он включает в себя множество готовых компонентов для настройки и развертывания приложений с подключением баз данных.

**5. Vue.js:** Прогрессивный JavaScript-фреймворк для создания пользовательских интерфейсов. Он помогает быстро создавать динамические и отзывчивые интерфейсы, которые легко интегрируются с backend.

**6.** **Система управления базой данных MySQL:** Реляционная система управления БД, обладающая высокой производительностью, надежностью и удобством в использовании для проектов с небольшой и средней нагрузкой.

**7.** **DBeaver:** Среда разработки для работы со многими популярными базами данных, предоставляющая удобный графический интерфейс для взаимодействия с ними.

**8.** **Система управления версиями GitHub:** Облачная платформа хранения кода используется для управления версиями проекта, их изменением и интеграций кода разных версий, что в делает разработку эффективной и безопасной

**2. Проектная стадия**

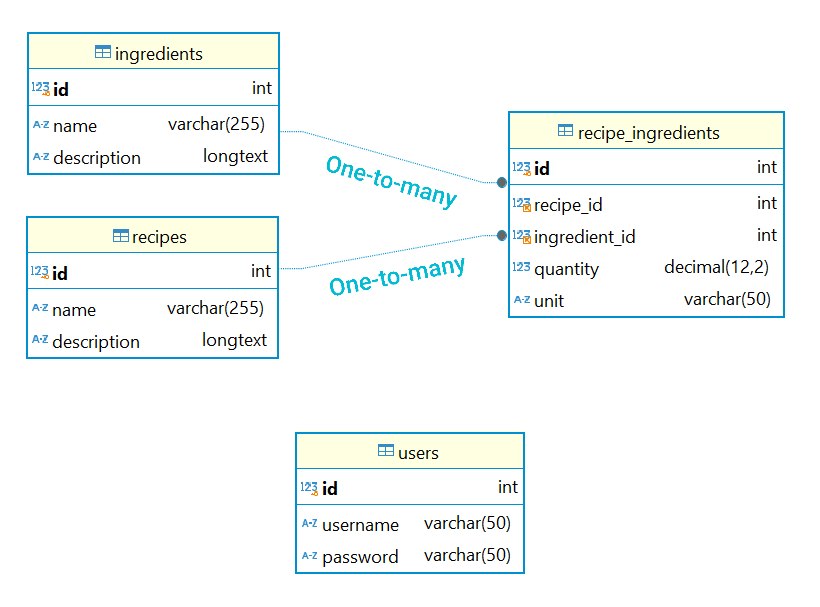
**2.1 Выбор шаблона проектирования**

Для разработки веб-приложения был использован шаблон проектирования Model-View-Controller. С его помощью можно разделить проект на три взаимосвязанных компонента, позволяя обеспечить удобство разработки, тестирования и поддержки кода:

* **Model (Модель):** управляет данными, обрабатывает запросы и обеспечивает взаимодействие с базой данных.
* **View (Представление):** отвечает за отображение данных для пользователя через интерфейс.
* **Controller (Контроллер):** связывает первые два компонента для реализации контроля и направления данных от пользователя к системе и наоборот.

**2.2 Проектирование архитектуры базы данных**

SQL-скрипт, используемый для создания базы данных, был подготовлен заранее и включает в себя создание таблиц, их связей, а также начальное наполнение базы данных тестовыми данными.

**Взаимоотношение сущностей:**

**1. users (пользователи):** Сущность не имеющая связей, необходимая для хранения данных о пользователе для входа в систему

**2. recipes (рецептуры):** Связь One-to-many с сущностью **recipe\_ingredients**, которая отображает выбранную рецептуру.

**3.** **ingredients (ингредиенты):** Связь One-to-one с сущностью **recipe\_ingredients**. В рецептуре содержится много ингредиентов, что и будет отображаться во второй.

**4.** **recipe\_ingredients (ингредиенты рецептуры):** Сущность, которая отвечает за отображение рецептуры с входящими в нее компонентами.

**2.3 Инструменты тестирования**

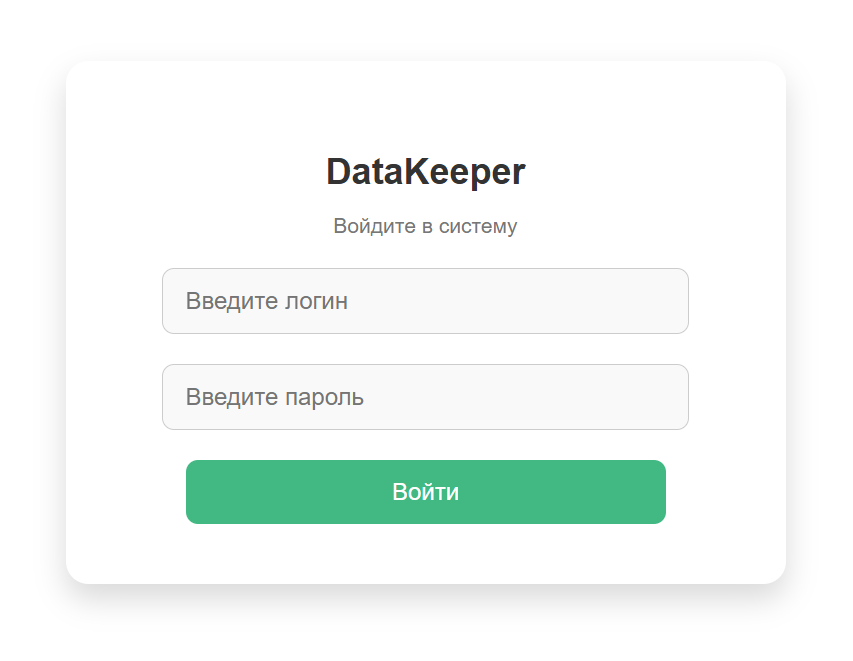
Тестирование играет важную роль во всех проектах, так как с помощью различных тестов выявляются как и очевидные ошибки, так и совершенно неожиданные, что в дальнейшем сильно повлияет на качество итогового продукта. Тестирования в данном проекте будут проходить с помощью **Swagger UI,** инструмента, который позволяет тестировать API через удобный веб-интерфейс, предоставляя интерактивную документацию. В нем будут доступны эндпоинты GET, POST, PUT, DELETE и параметры с быстро возвращаемыми ответами.

**3. Практическая часть**

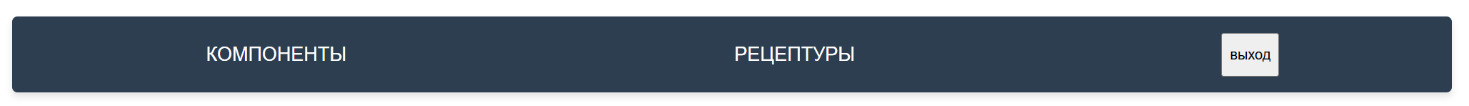
**3.1 Реализация дизайна и функционала веб-приложения**

Началом данного проекта послужило проектирование веб-интерфейса, что в будущем позволило правильно реализовать backend, основываясь на необходимых функциях для работы пользователя в системе.

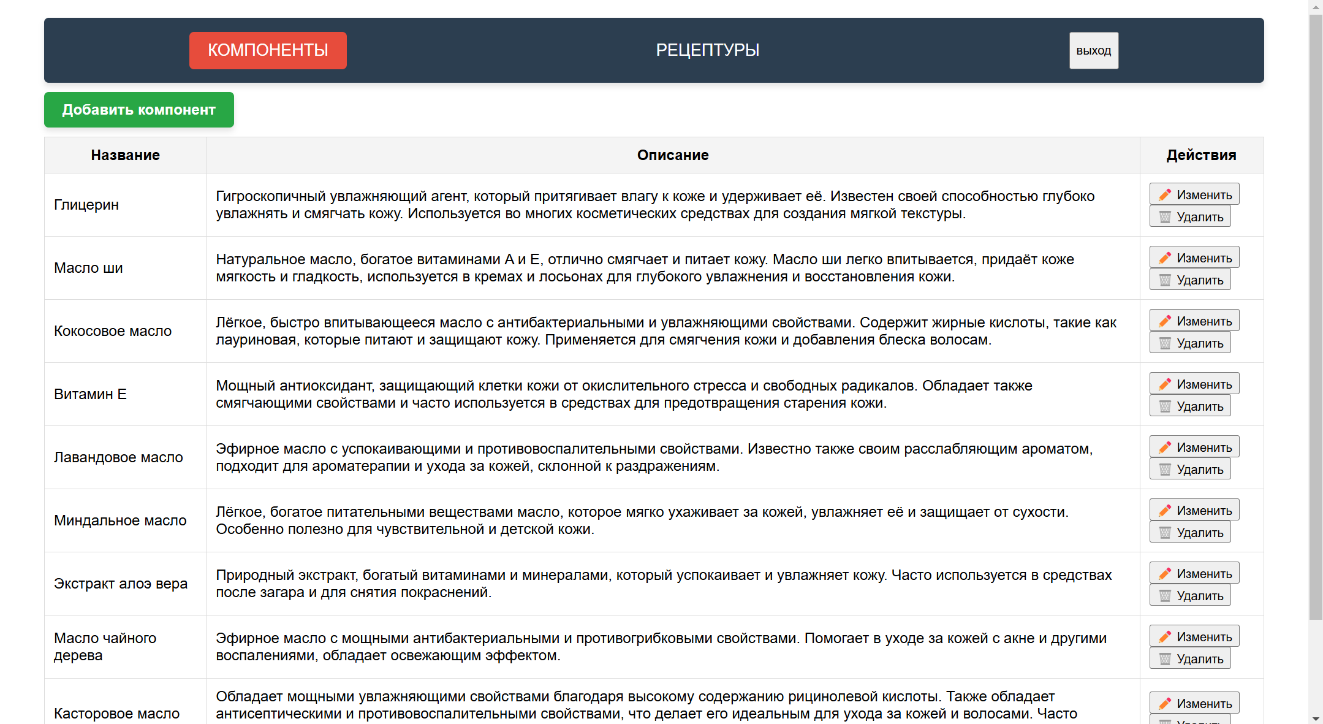
Интерфейсом выступают Vue файлы, которые содержат в себе веб-страницы и их функционал, а так же связь с сервером и базой данных.

1. Логин форма

2. При успешном входе в систему, пользователя перебрасывает на страницу с навигационной панелью.

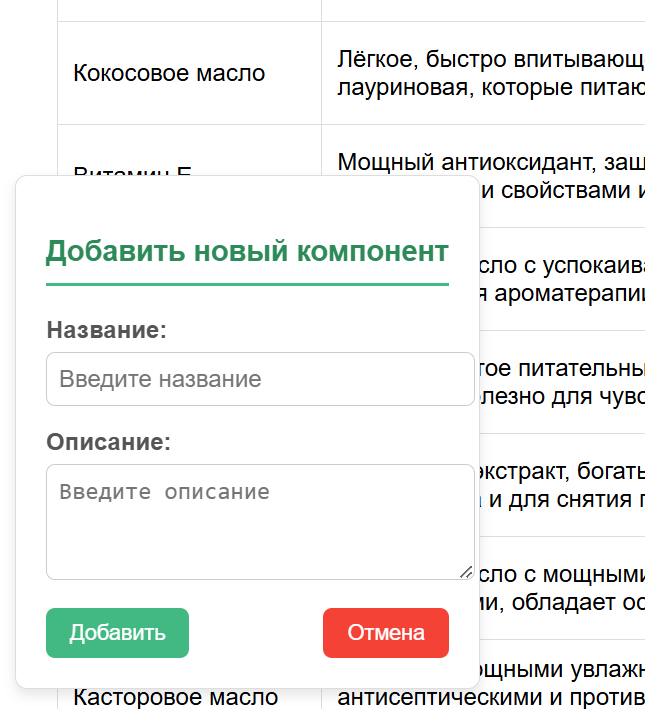
****

При нажатии на компоненты или рецептуры раскрываются соответствующие списки с дальнейшими действиями в них. При нажатии на кнопку выхода, пользователь выходит из системы в предыдущую страницу.

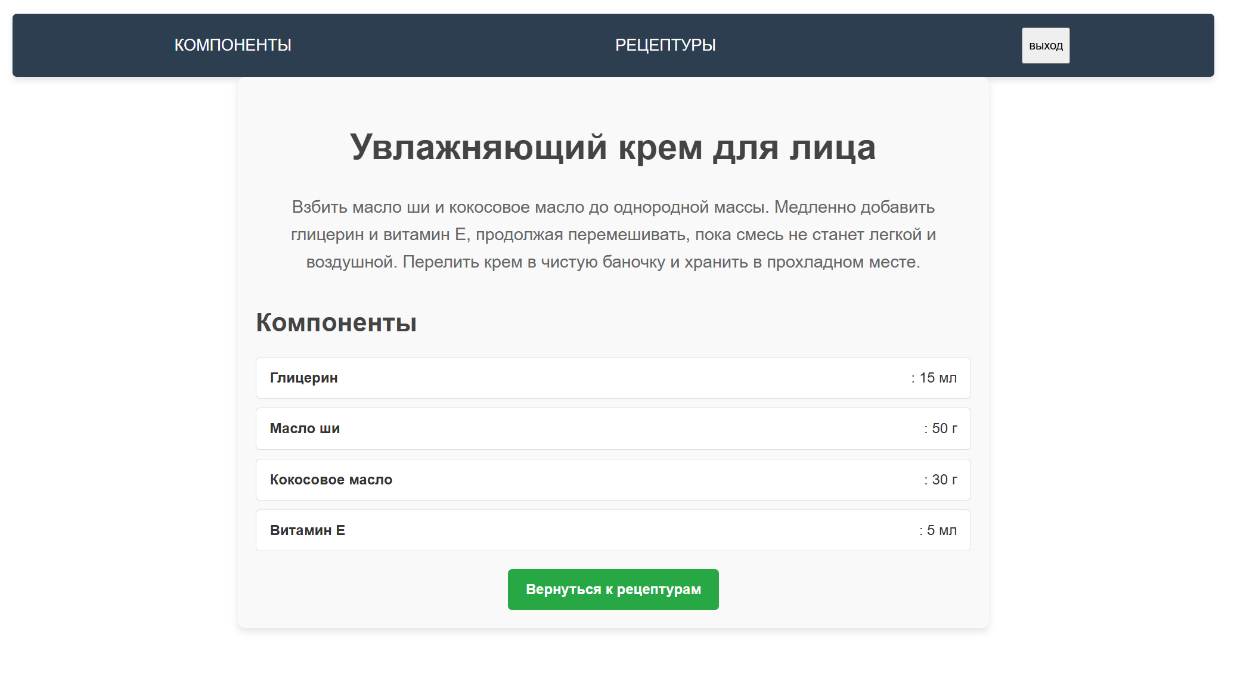
3. Раздел “КОМПОНЕНТЫ”

В данном разделе отображается список всех доступных компонентов и их описание в виде таблицы, что позволяет быстро их изучить и найти подходящий.

Функционал:

* Добавление нового компонента осуществляется нажатием на соответствующую кнопку, после чего в левом нижнем углу экрана появляется интерактивная форма для заполнения данных.
* Изменение выбранного ингредиента реализуется тем же всплывающим окном, что и при добавлении, но с уже присутствующей информацией, как в списке. При отмене правок, форма просто закрывается, а при нажатии сохранения, внесенные изменения обновляются в базе данных. Такой функционал позволит поддерживать актуальность данных.
* Удаление компонента осуществляется через соответствующую кнопку, которая удаляет его по id-ключу из базы данных.

1. Раздел “РЕЦЕПТУРЫ”

 Данный раздел оформлен в том же стиле, что и “КОМПОНЕНТЫ”, но отличительной чертой является возможность перейти на страницу с подробным описанием рецептуры, где дополнительно будет отображено сколько и каких компонентов участвуют в производстве. Переход на страницу осуществляется путем нажатия на название рецептуры.

После входа на соответствующую страницу сохраняется возможность пользоваться навигационной панелью, а так же путем нажатия кнопки в конце страницы вернуться в раздел с перечнем рецептур. Реализация данного отображения осуществляется за счет связующей таблицы recipe\_ingredients в базе данных, функционал которой реализован в серверной части проекта о которой пойдет речь в следующем разделе.

В добавок, стоит обратить внимание на код web составляющей. В рамках данного проекта была задействована библиотека Axioc, предназначенная для работы с HTTP-запросами в JavaScript. Она используется для отправки запросов к серверу и получения данных, а так же для взаимодействия с RESTful API. Эксплуатация данной технологии позволяет легко работать с асинхронными запросами в приложениях, так как она является промисо-ориентированной библиотекой.

Одним из компонентов дизайна интерфейса является Vue.js с возможностью динамического отображения данных используя v-if, v-for и другие его директивы, а так же метод fetchData() , который асинхронно загружает данные. Вторым компонентом является CSS, контролирующий внешний вид таблиц рецептур и компонентов, кнопки и ссылки, внутри тега style scoped. Атрибут scoped означает то, что они не будут влиять на другие компоненты приложения.

**3.2 Создание классов-сущностей**

Класс Recipe, представленный ниже, является сущностью (entity class), предназначенной для отображения данных из таблицы базы данных recipes на объекты Java с использованием ORM (Object-Relational Mapping) и JPA (Java Persistence API).

**Аннотации:**

* @Getter и @Setter: Автоматически генерируют методы get и set для всех полей класса. Такой способ написания стандартных методов доступа избавляет от необходимости делать это вручную для каждого отдельно.
* @Entity: Используется для обозначения класса как сущности, то есть класса, который будет взаимодействовать с базой данных через объекты.
* @Table(name = “Recipes”): Определяет имя таблицы в базе данных, к которой будет привязан этот класс. В данном случае класс Recipe будет связан с таблицей Recipes.
* @ Id: Указывает, что поле id является первичным ключом (Primary Key) для таблицы.
* @Column: Используется для настройки параметров соответствующего столбца в таблице
* @GeneratedValue(strategy = GenerationType.IDENTITY): Указывает, что значение для первичного ключа id будет генерироваться автоматически с использованием стратегии IDENTITY, что в свою очередь означает, что база данных сама будет генерироват значение для этого поля с помощью auto increment.
* @OneToMany(mappedBy = “recipe”): Указывает на связь с один-ко-многим с сущностью RecipeIngredient.

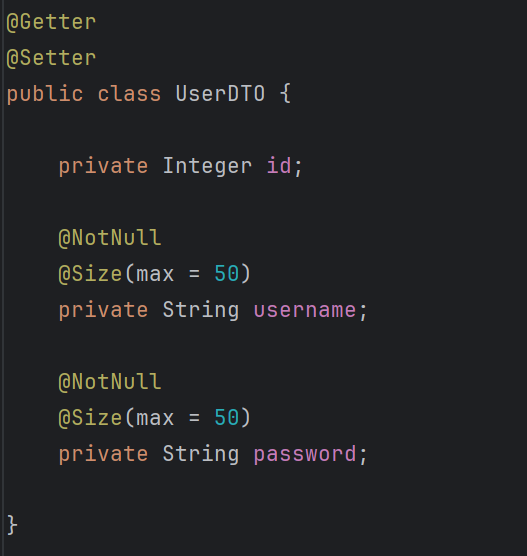
Аналогично структурированны классы User, Ingredient и RecipeIngredient.

**Описание сущностей:**

1. **User :** Поля для хранения данных о пользователях – логин и пароль.
2. **Ingredient:** Поля для хранения наименований ингредиентов и их описании.
3. **Recipe:** Поля для хранения названий рецептур и способу приготовления.
4. **RecipeIngredient:** Поля для хранения используемых ингредиентов и их граммовок в определеных рецептурах.

Классы **DTO (Data Transfer Object)** играют ключевую роль в архитектуре приложений, обеспечивая удобную передачу данных между такими слоями, как сервисы, контроллеры и представления. Они способствуют изоляции логики приложения от структуры данных, упрощают валидацию и обработку входной информации. Использование DTO-классов также повышает уровень безопасности, так как ограничивает объем данных, передаваемых между слоями, защищая внутренние структуры от несанкционного доступа.

В данном проекте применены следующие DTO-классы: UserDTO, IngredientDTO, RecipeDTO и RecipeIngredientDTO. Для контроля входных данных при создании или редактировании объектов используются аннотации валидации. Если данные не соответствуют требованиям, ошибки передаются в контроллер, который возвращает форму с указанием проблем, чтобы пользователь мог их исправить.

****Клас **UserDTO:**

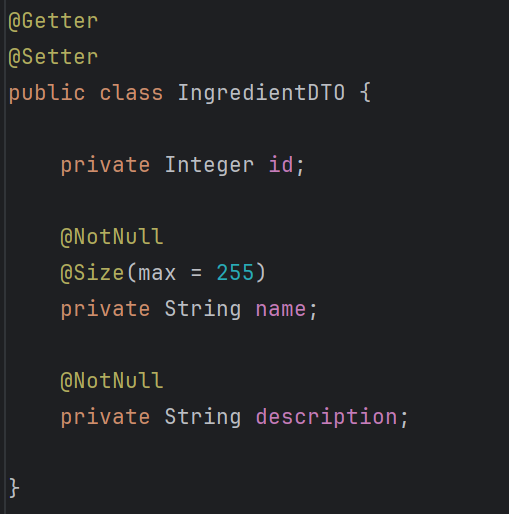
Валидация полей **UserDTO:**

1. Поле id: Используется для идентификации пользователя. Оно не содержит валидационных аннотаций, что может означать, что это поле необязательно при создании или обновлении объекта.
2. Поле username:

* **@NotNull:** Указывает, что данное поле не может быть null, гарантируя, что при передаче объекта значение имени пользователя должно быть обязательно задано.
* **@Size(max = 50):** определяет максимальную длину строки – не более 50 символов. Это ограничение предотвращает запись слишком длинных имен пользователей, которые могут вызвать ошибки или нарушить работу приложения.

1. Поле password:

* **@NotNull:** Указывает, что поле password обязательно для заполнения и не может быть null. Это исключает возможность создания пользователя без пароля.
* **@Size(max = 50):** Задает максимальную длину пароля – не более 50 символов, что позволяет ограничит длину пароля, обеспечивая удобство хранения и обработки данных.

Класс **IngredientDTO:**

Валидация полей **IngredientDTO:**

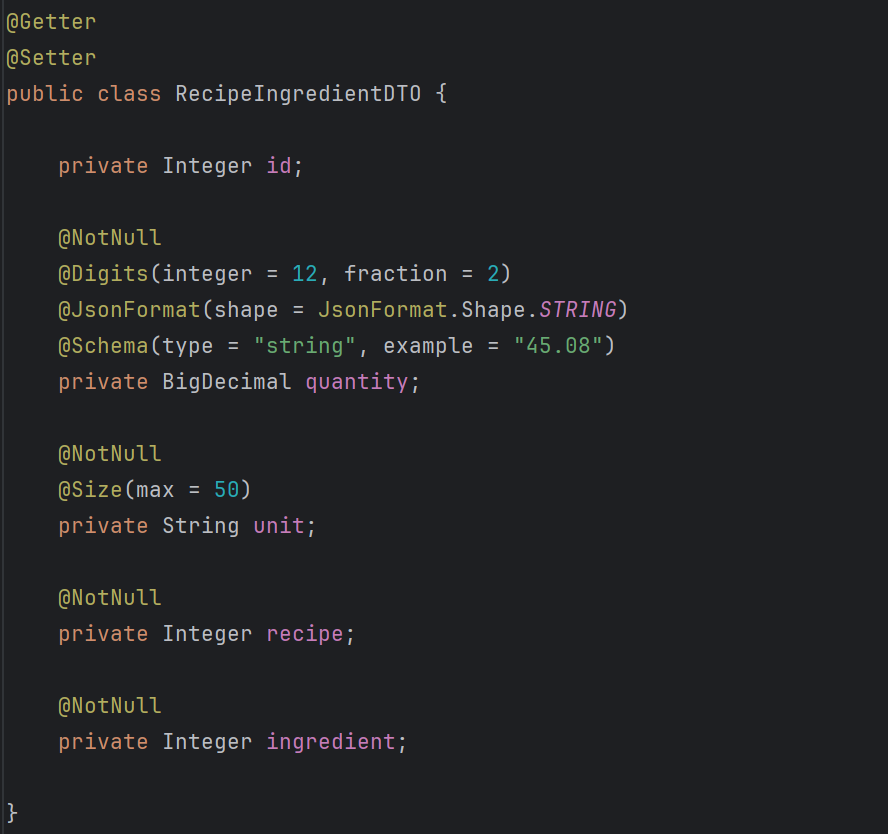
1. Поле id: Предназначено для идентификации ингредиента. Аннотация отсутствует, означая, что проверка данного поля необязательна.
2. Поле name:

* **@NotNull:** указывает, что поле name не может быть null, обеспечивая обязательное наличие наименования ингредиента.
* **@Size(max = 255):** Ограничивает длину строки – не более 255 символов. Такое ограничение необходимо для предотвращения нарушения корректное функционирование системы или превышения ограничений базы данных.

1. Поле description:

* **@NotNull:** указывает, что поле description обязательно для заполнения. Это исключает возможность создания или обновления ингредиента без описания.
* Стоит заметить, что у данного поля не указаны какие либо ограничения по символам, так как в базе данных указано, что данное поле имеет тип text, а это значит, что оно по умолчанию может хранить до 65535 символов.

Идентичным образом был организован класс **RecipeDTO,** который служит для хранения наименований рецептур и способа их изготовления.

Класс **RecipeIngredientDTO:**

Валидация полей **RecipeIngredientDTO:**

1. Поле id: Представляет идентификатор и не имеет аннотаций. Данное поле используется при создании нового объекта и для отображения значения из базы данных.
2. Поле quantity:

* **@NotNull:** Указывает, что поле не может быть пустым, гарантируя, что для каждого ингредиента будет указано количество.
* **@Digits(integer = 12, fraction = 2):** Ограничивает формат значения так, чтобы число содержало до 12 цифр в целой части и до 2 цифр в дробной.
* **@JsonFormat(shape = JsonFormat.Shape.STRING):** Определяет, что значение будет сериализованно/десериализованно как строка в JSON. Данный способ удобен для передачи чисел с фиксированным форматом, исключая ошибки округления.
* **@Schema(type = “string”, example = “45.08”):** Аннотация предназначенная для генерации документации Swagger. Она указывает, что поле будет описано как строка и дает пример значения 45.08.

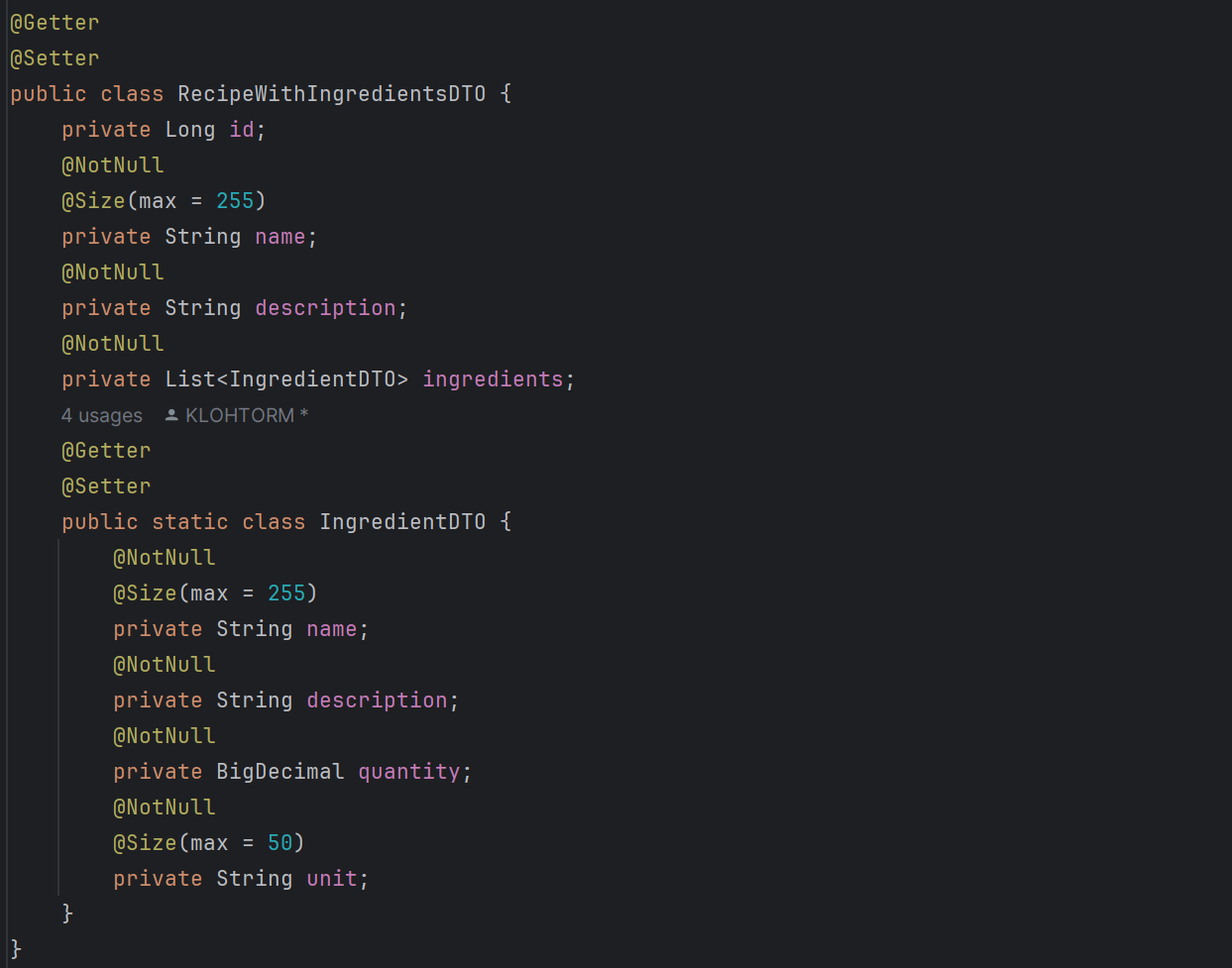
1. Поле unit:

* **@NotNull:** Поле обязательно для заполнения, гарантируя наличие единицы измерения.
* **@Size(max = 50):** Устанавливает максимальную длину строки в 50 символов, предотвращая ошибок и нарушение работы базы данных.

1. Поле recipe и поле ingredient:

* **@NotNull:** Поле обязательно для заполнения, так как указывает на идентификатор рецептуры/ингредиента.
* Данное поле связано с существующей записью в таблице recipes/ingredients.

Класс **RecipeWithIngredientsDTO:**

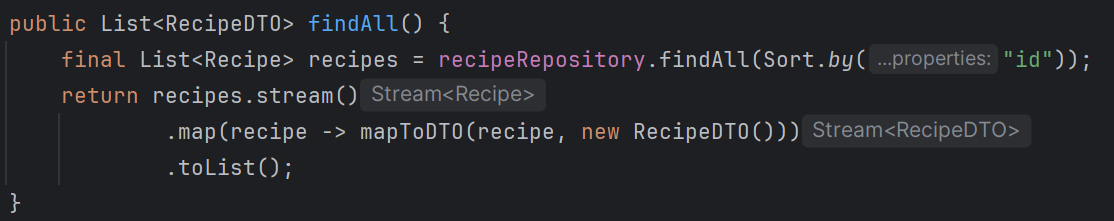
Предназначен для передачи данных о рецептуре, включая информацию о его названии, способу изготовления и списке множества компонентов участвующих в изготовлении продукции. В проекте класс обеспечивает удобную передачу данных между различными слоями приложения, такими как контроллеры и сервисы.

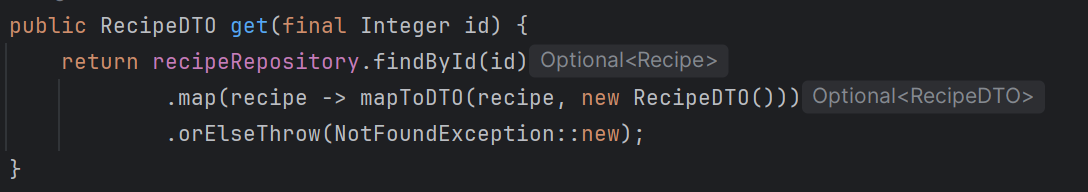
**3.3 Создание классов-сервисов**

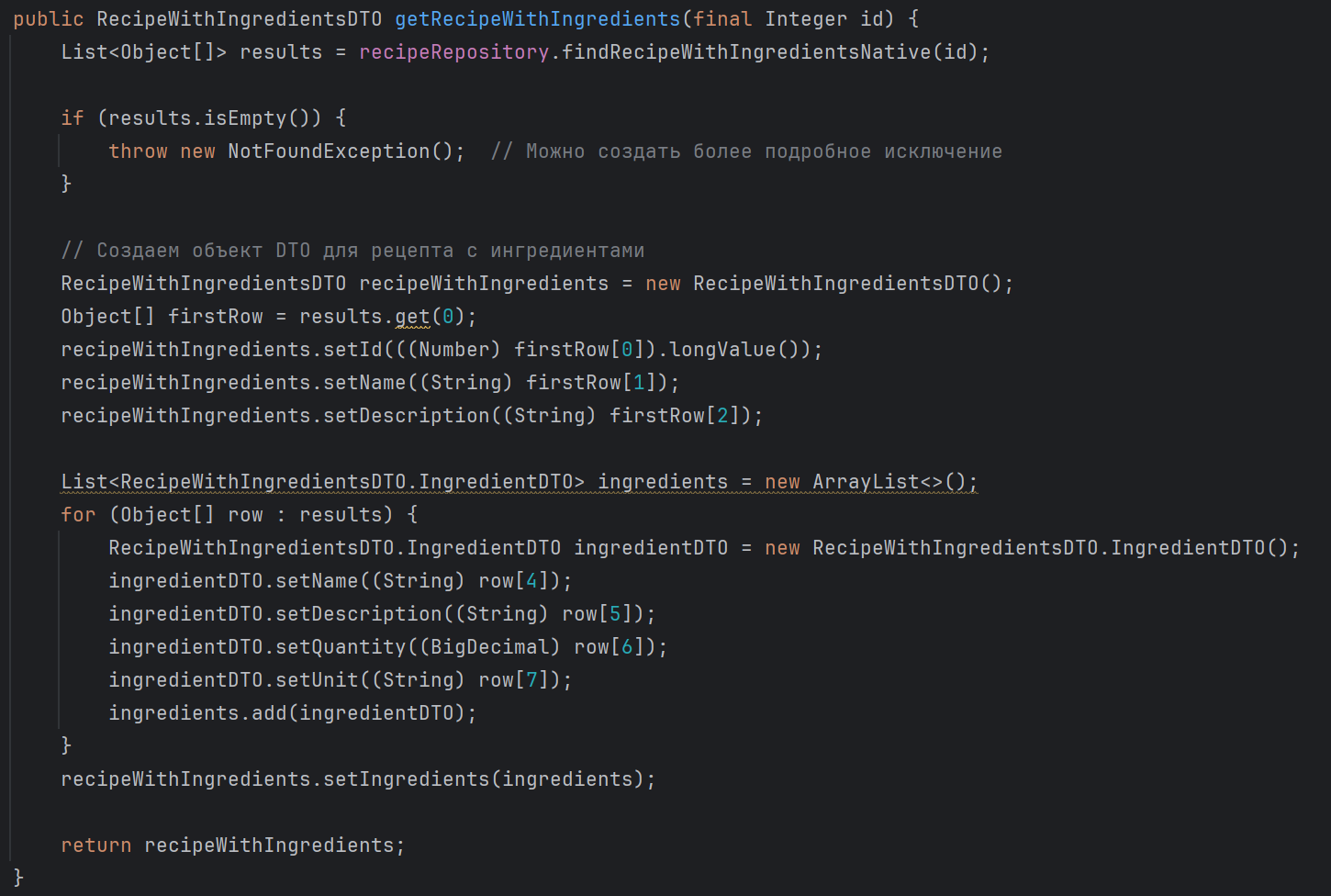
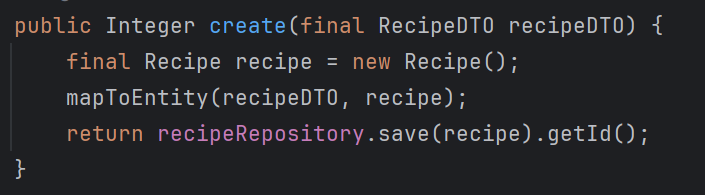
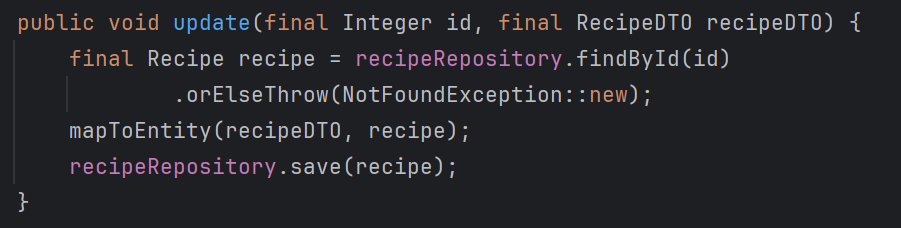
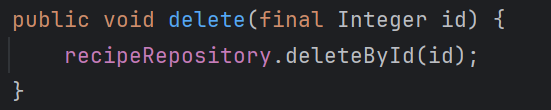
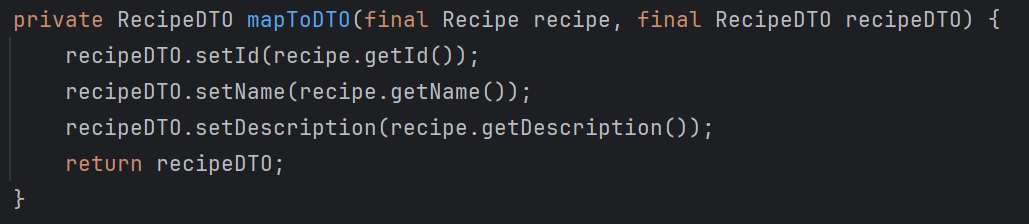
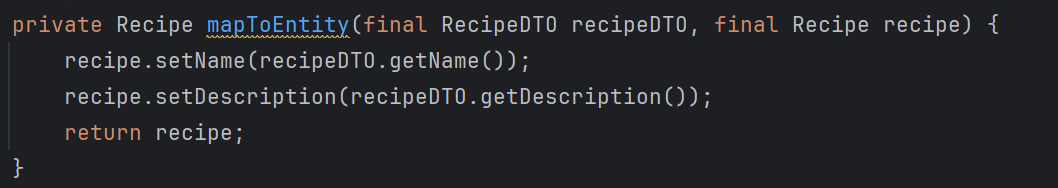
Класс **RecipeService** является сервисным слоем в архитектуре приложения и отвечает за реализацию работы, связанной с управлением рецептурами. Данный класс служит посредником между контроллерами и репозиториями (в данном случае RecipeRepository и RecipeIngredientRepository), предоставляя методы для создания, чтения, обновления и удаления рецептур.

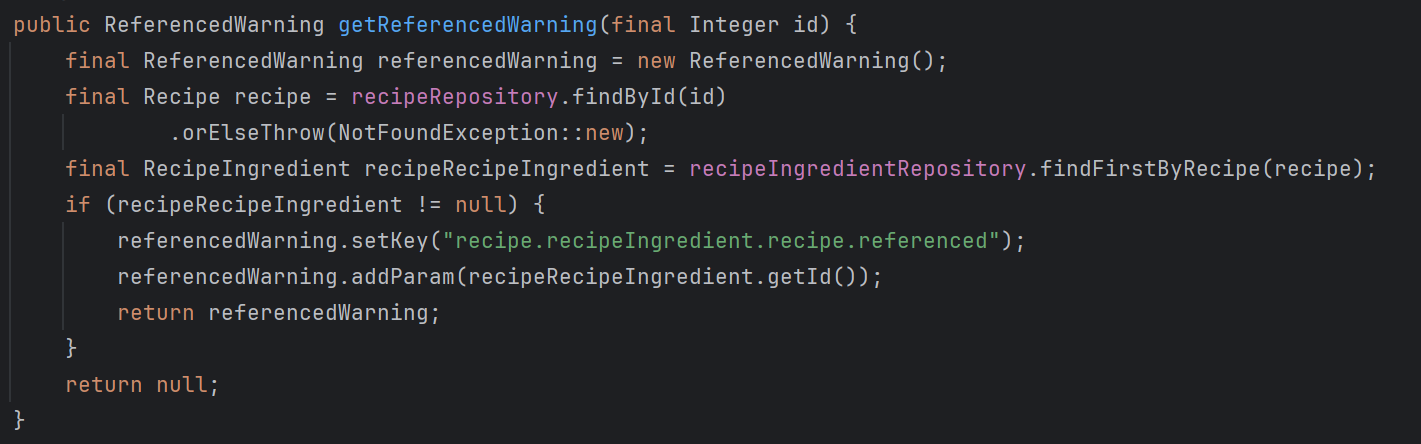
Аннотация **@Service** в фреймворке Spring, предназначена для пометки классов, которые выполняют роль сервисного слоя в приложении. Этот класс будет зарегестрирован как Spring Bean, то есть он будет управляться контейнером Spring, который сможет создавать и управлять

**Основные функции класса:**

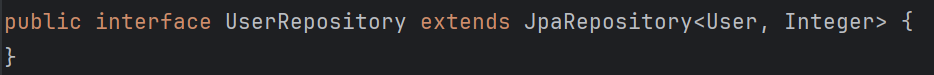
1. ****Метод findAll(): Он получает все рецептуры из репозитория с сортировкой по id и преобразует каждый в DTO, который затем добавляет в список.
2. Метод get(Integer id): Метод пытается найти рецептуру по переданному id. Если запись найдена, он преобразуется в DTO и возвращается с информацией, а если не найдена, выбрасывается NotFoundException.

****

1. ****МетодgetRecipeWithIngredients(Integer id): Используется нативный SQL-запрос, который извлекает данные о рецептуре и его компонентах из базы данных и возвращает в виде объекта DTO. Если какая либо информация не была найдена, выбрасывается NotFoundException.
2. ****Метод create(RecipeDTO recipeDTO): Создает новый объект Recipe, используя данные из DTO, а затем сохраняет его в репозитории, возвращая идентификатор созданной рецептуры.
3. ****Метод update(Integer id, RecipeDTO recipeDTO): Позволяет редактировать реццептуру путем, нахождения рецептуры по переданному id, затем обновляет его данными из RecipeDTO и сохраняет изменения в репозитории.
4. ****Метод delete(Integer id): Пытается удалить рецептуру с переданным id из репозитория. Если такой id не найден, выбрасывается NotFoundException.
5. ****Метод mapToDTO(Recipe recipe, RecipeDTO recipeDTO): Вспомогательный метод копирующий значения из сущности Recipe в RecipeDTO. Используется в методах преобразования данных, например findAll() и get().
6. ****Метод mapToEntity(RecipeDTO recipeDTO, Recipe recipe): Копирует данные из RecipeDTO в сущность Recipe, чтобы сохранить изменения в базе данных. Используется в методах create() и update(), которые преобразовывают данные в сущность.
7. Метод getReferencedWarning(Integer id): Проверяет, ссылаются ли другие сущности на данную рецептуру. Если рецептура используется в RecipeIngredient, генерируется предупреждение возвращая объект с информацией о ссылке. Если его нет в других сущностях, возвращается null.

****Классы UserService, IngredientService и RecipeIngredientService структурированы аналогичным способом, как и класс RecipeService. Эти классы выступают сервисными компонентами в архитектуре данного веб-приложения, которые предоставляют методы для взаимодействия с соответствующими репозиториями и реализуют логику, связанную с обработкой и управлением данными.

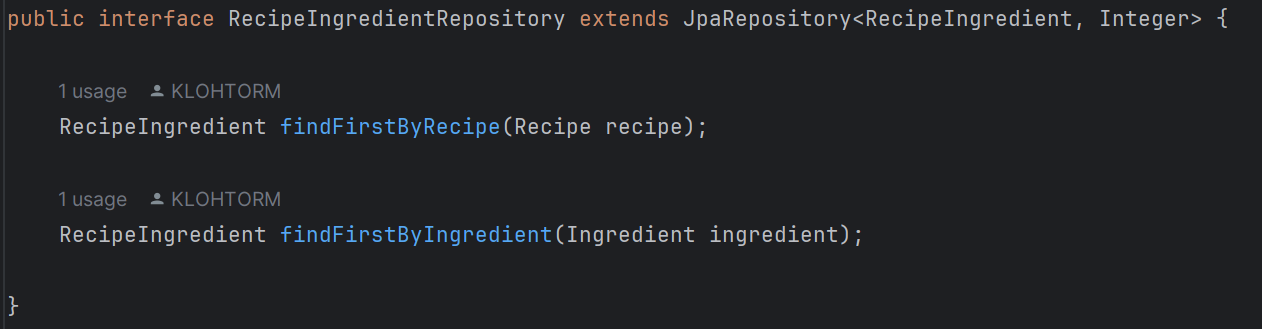
**3.4 Создание интерфейсов-репозиториев**

Интерфейс **UserRepository:**

UserRepository представляет собой интерфейс, который который является частью уровня доступа к данным – Data Access Object (DAO). Он является шаблоном проектирования, который используется для абстракции и управления доступом к данным в приложении. DAO отделяет логику взаимодействия с базой данных от основной бизнес-логики, предоставляя четкий интерфейс для выполнения операций с данными. Данная технология в рамках представленного интерфейса обеспечивает доступ к данным сущностей User, выполняя взаимодействие с базой данных и CRUD-операции (Create Read Update Delete):

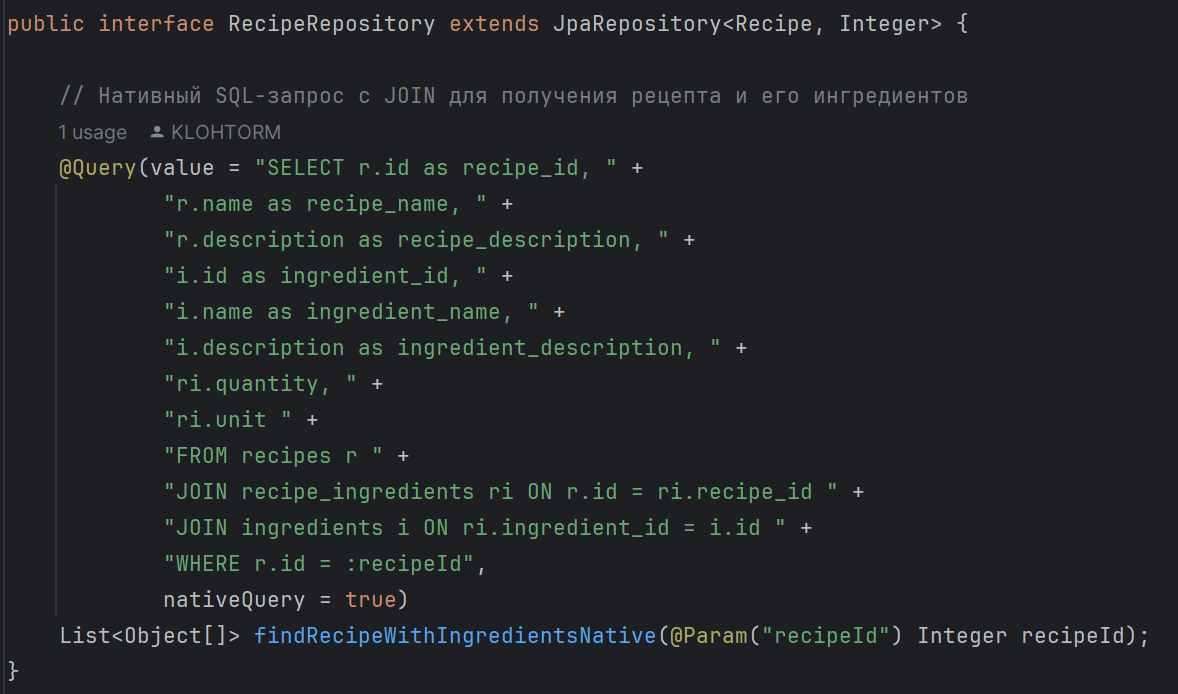
* save() : сохранение или обновление объекта.
* findById() : поиск сущности по идентификатору.
* findAll() : получение всех объектов.
* deleteById(): удаление сущности по идентификатору.

В текущем виде интерфейс UserRepository не содержит дополнительных методов, что означает, что приложение использует только стандартные возможности JpaRepository. При необходимости можно доюавить собственные методы, например для поиска пользователей по определенным критериям с помощью Spring Query Methods или аннотаций @Query. Идентичным с UserRepository является IngredientRepository.

****Интерфейс **RecipeIngredientRepository:**

В данном интерфейсе находятся два кастомных метода для работы с данными, которые могут быть вызваны из сервисного слоя:

* findFirstByRecipe(Recipe recipe): Находит первую запись RecipeIngredient, связанную с заданной рецептурой. Используется для проверки, связана ли определенная рецептура с каким-либо ингредиентом.
* findFirstByIngredient(Ingredient ingredient): Находит первую запись RecipeIngredient, связанную с конкретным компонентом. Необходим для определения, используется ли данный компонент в какой-либо рецептуре.

Интерфейс **RecipeRepository:**

Интерфейс с кастомным методом findRecipeWithIngredientsNative(@Param(“recipeId”) Integer recipeId), использующийся для работы с сущностью Recipe, позволяя извлекать данные о рецептурах и их компонентах с помощью стандартных методов JPA и кастомного SQL-запроса. Представленный метод оптимизирует получение сложных связанных данных из базы в виде массива объектов.

Аннотации:

1. **@Query:** указывает нативный SQL-запрос, который будет выполняться, используя JOIN для объединения таблиц:

* recipes : таблица рецептур.
* recipe\_ingredients : таблица связей между рецептурами и компонентами.
* ingredients : таблица компонентов.

1. **@Param:** Передается запрос через аннотацию, указывая на id рецептуры, данные которой нужно получить.

Запрос возвращает массив объектов, содержащий следующие данные:

* r.id : идентификатор рецептуры
* r.name : название рецептуры
* r.description : метод приготовления
* i.id : идентификатор компонента
* i.name : название компонента
* i.description : описание компонента
* ri.quantity : количество используемого компонента
* ri.unit : единица измерения компонента

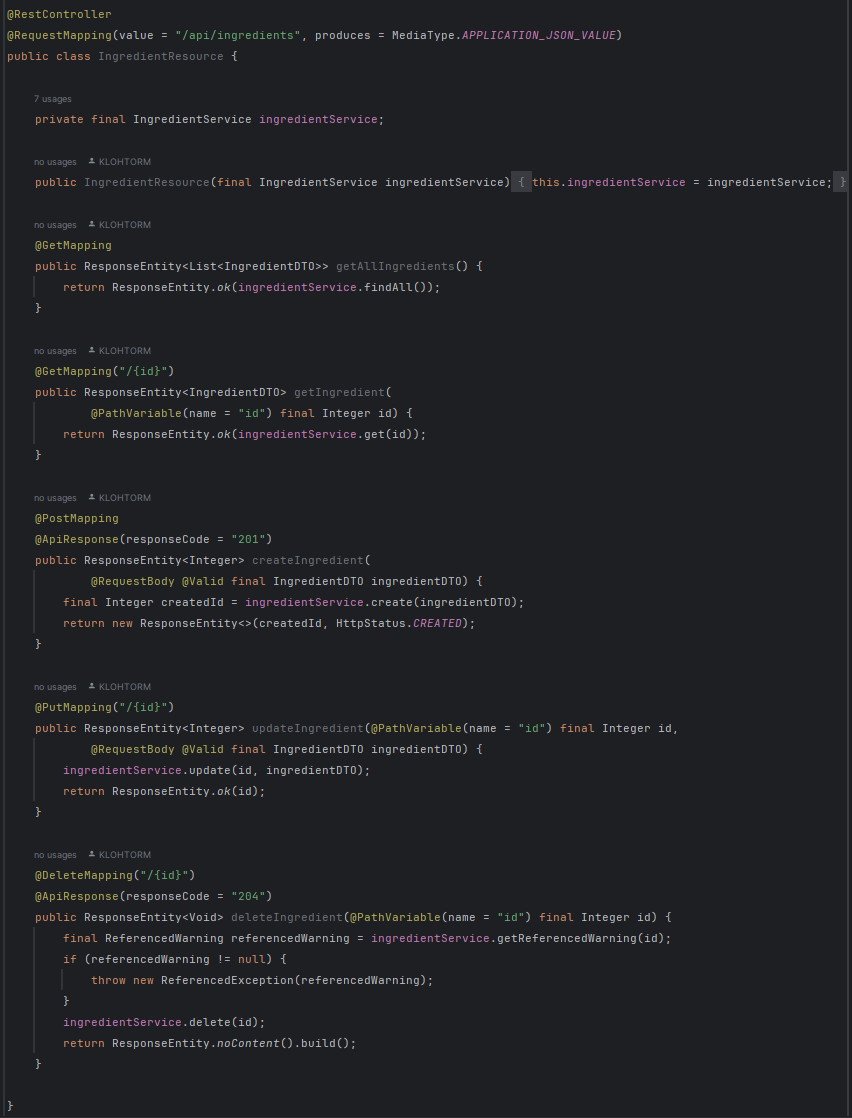
**3.5 Создание классов-контроллеров**

В рамках архитектуры Model-View-Controller, класс-контроллер выполняет роль посредника между моделью и представлением. В проекте с использованием фреймворка Spring , контроллеры отвечают за маршрутизацию запросов, валидацию данных и взаимодействие с сервисным слоем.

Основные функции класса-контроллера:

1. Обработка HTTP-запросов: Контроллер принимает запросы от клиента и сопоставляет их с соответствующими методами на основе маршрутов GET, POST, PUT и DELETE.
2. Маршрутизация: Определение маршрутов для взаимодействия с клиентами происходит используя аннотации @RequestMapping, @GetMapping, PostMapping и т.д.
3. Валидация входящих данных: Перед отправкой данных в сервисный слой, контроллер может проверять их корректность используя аннотацию @Valid для автоматической проверки.
4. Обработка ошибок: Данный класс может перехватывать ошибки и предоставлять понятные ответы клиенту, формируя пользовательские сообщения об ошибках, например, при неккоректных данных или отсутствии сущности.
5. Взаимодействие с сервисным слоем: Делегирует выполнение бизнес-логики сервисам, но сам не выполняет сложные вычисления, концентрируясь только на маршрутизации и валидации.
6. Формирование ответа: Преобразует данные модели в понятный клиенту формат. В рамках данного проекта – JSON.
7. Возврат статусов HTTP: Определяет HTTP-статусы, которые информируют клиента о результате выполнения запроса:

* **200 OK:** Успешное выполнение.
* **201 Created:** Создание сущности.
* **400 Bad Request:** Ошибка валидации.
* **404 Not Found:** Объект не найден.

Класс **IngredientResource:**

**Аннотации:**

* **@RestController:** Обозначает класс как REST-контроллер. Все методы возвращают объекты, которые автоматически сериализуются в JSON.
* **@ApiResponse:** Swagger-аннотация, указывающая на ожидаемый HTTP-статус для документации API
* **@RequestMapping:** Устанавливает базовый маршрут для всех запросов этого контроллера (/api/ingredients) и определяет, что ответы будут возвращаться в формате JSON.
* **@GetMapping, @PostMapping, @PutMapping, @DeleteMapping :** Маршруты для выполнения операций CRUD.

Поле ingredientService инициализируется через конструктор. Это сервисный компонент, выполняющий бизнес-логику работы с компонентами для рецептур.

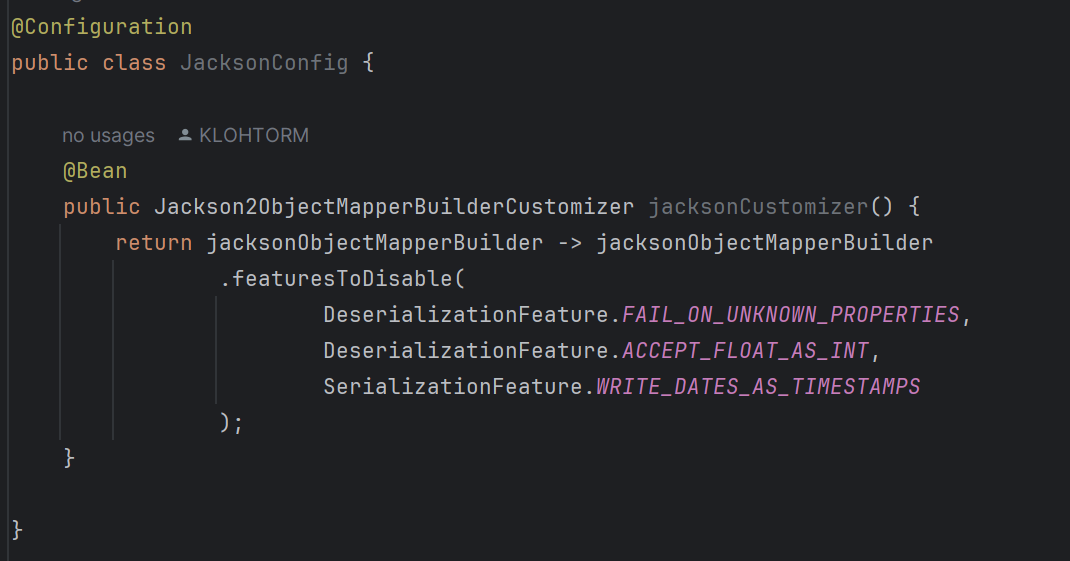
Методы контроллера:

1. getAllIngredients() : Получение всех ингредиентов путем обработки GET /api/ingredients запроса и возвращения списка всех компонентов в формате JSON. Использует метод findAll() из IngredientService.
2. getIngredient(@PathVariable(name = “id”) : Получение компонента по id путем обработки GET /api/ingredients/{id} запроса возвращая объект или ошибку, если такового найдено не было. Извлекает идентификатор компонента из URL-параметра. Использует метод get(id) из IngredientService.
3. createIngredient(@RequestBody @Valid final IngredientDTO ingredientDTO) : Создание нового компонента путем обработки POST /api/ingredients запроса, возвращает id созданного компонента и статус 201 Created. Вызывает метод create(ingredientDTO) в IngredientService.
4. updateIngredient(@PathVariable(name = “id”) final Integer id, @RequestBody @Valid final IngredientDTO ingredientDTO) : Обновоение компонента путем обработки PUT /api/ingredients/{id}, возвращая id обновленного компонента и статус 201 Created. Обновляет информацию о компоненте с помощью метода update(id, ingredientDTO) в IngredientService.
5. deleteIngredient(@PathVariable(name = “id”) final Integer id): Удаляет компонент путем обработки запроса DELETE /api/ingredients/{id}, возвращая статус 204 No Content, что указывает на успешное удаление без содержания тела ответа. Если компонент связан с другими сущностями, выбрасывает исключение ReferencedException, в противном случае вызывает метод delete(id).

Классы-контроллеры UserResourse, RecipeResourse и RecipeIngredientResourse написаны аналогично.

**3.6 Создание классов-конфигов**

Класс **JacksonConfig:**

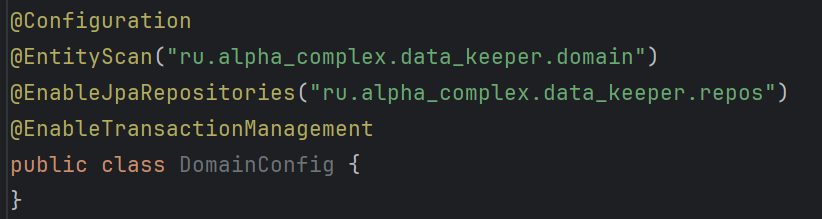
**** Данный класс представляет из себя конфигурационный класс для настройки библиотеки Jackson в Spring Boot приложении. Представленная библиотека используется для сериализации и десериализации объектов Java в формат JSON и обратно. В данном случа, класс настраивает параметры поведения сериализации и десериализации данных.

Аннотация **@Configuration**, указывает, что класс является конфигурационным для Spring. Он будет обработан как класс для контекста приложения, и его методы будут создавать бины (@Bean), которые будут внедряться в другие части приложения.

Настройки Jackson в методе **jacksonCustomizer():**

* **jacksonObjectMapperBuilder.featuresToDisable():** Метод отключает некоторые особенности сериализации и десериализации, которые будут описаны в следующих пунктах.
* **DeserializationFeature.FAIL\_ON\_UNKNOWN\_PROPERTIES:** Отключает исключение при десериализации объекта, если входящий JSON содержит свойства, которых нет в классе. Это позволяет игнорировать дополнительные свойства в JSON, которые не соответствуют полям Java-объекта.
* **DeserializationFeature.ACCEPT\_FLOAT\_AS-\_INT:** Отключает обработку значений с плавающей точкой (например 1.0) как целых чисел (например 1), что предотвращает потенциальные ошибки при несоответствии типов.
* **SerializationFeature.WRITE\_DATES\_AS\_TIMESTAMPS:** Отключает сериализацию объектов даты в виде меток времени (timestamps), что означает, что даты будут сериализоваться в ISO-формате, например "2024-11-28T14:30:00".

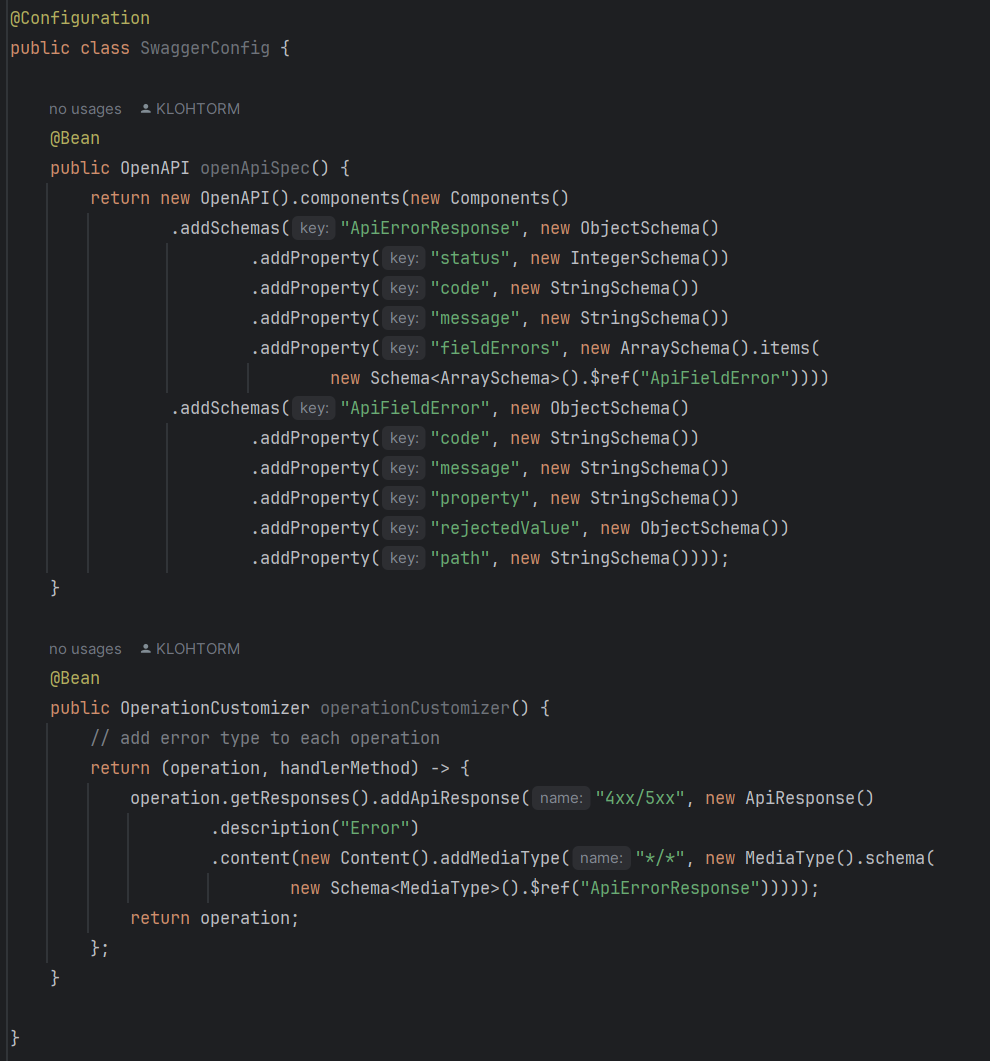
Класс **DomainConfig:**

**** Является конфигурационным классом в Spring Boot приложении, который настраивает интеграцию с **JPA (Java Persistence API)** для работы с базой данных. Этот класс включает аннотации, которые обеспечивают правильную настройку репозиториев, сущностей и управления транзакциями.

**Аннотации:**

1. **@Configuration:** Указывает, что класс является конфигурационным и в контекст приложения будут добавлены бины.
2. **@EntityScan:** Указывает Spring, где искать JPA сущности, что является очень важной частью, потому что он должен знать, где находятся классы, которые представляют сущности, которые будут мапироваться на таблицы в базе данных. В данном случае, сущности находятся в пакете ru.alpha\_complex.data\_keeper.domain
3. **@EnableJpaRepositories:** Данная аннотация указывает, где искать интерфейсы репозиториев, которые расширяют JpaRepository или другие подобные интерфейсы. В данном случае, они находятся в пакете ru.alpha\_complex.data\_keeper.repos
4. **@EnableTransactionManagement:** Подключает управление транзакциями для репозиториев JPA, а также для методов, помеченных аннотациями @Transactional.

Класс **SwaggerConfig:**

Конфигурация для Swagger, используемая для автоматической генерации документации API.

Метод **openApiSpec() :**

Создаётся новый объект OpenAPI, который является основной точкой настройки спецификации OpenAPI.

Добавляются две схемы:

1. ApiErrorResponse: Описывает структуру ошибок, которые могут быть возвращены. Она включает в себя следующие свойства:

* status: Статус ответа.
* code: Код ошибки.
* message: Сообщение ошибки.
* fieldErrors: Ошибки валидации полей, если такие имеются.

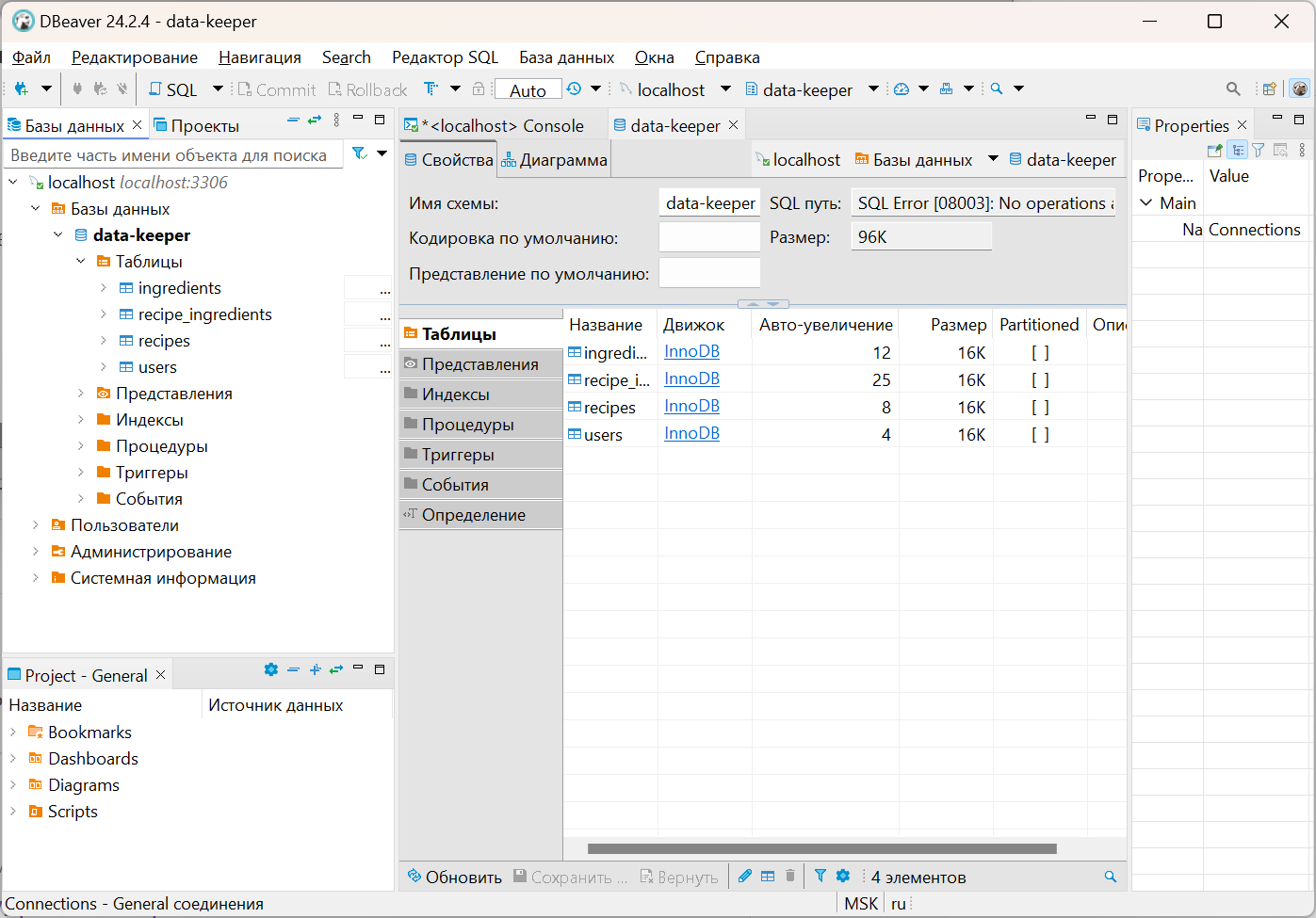
1. ApiFieldError: Схема для представления ошибок, связанных с полями. Включает следующие свойства:

* code: Код ошибки для поля.
* message: Сообщение ошибки для поля.
* property: Название поля, с которым связана ошибка.
* rejectedValue: Значение, которое не прошло валидацию.
* path: Путь к полю в объекте запроса.

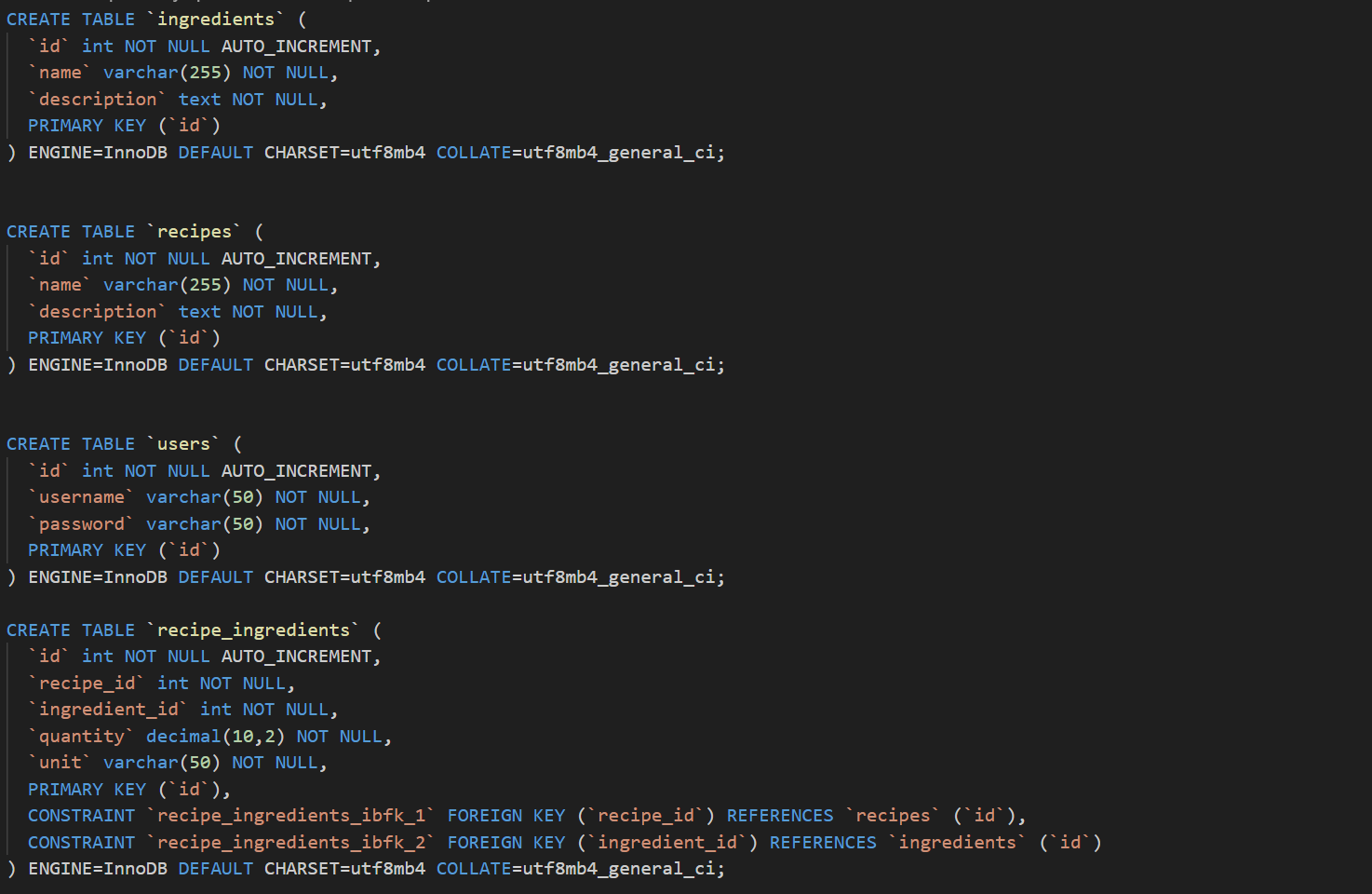
Метод **operationCustomizer():** Этот метод позволяет добавить информацию о возможных ошибках в каждый ответ API. В методе создается кастомизация для каждой операции (метода контроллера), добавляя описание ошибок для HTTP-статусов 4xx и 5xx. Это позволяет автоматически включать описание ошибок в документацию API, что поможет тому, кто его использует понять, как правильно обрабатывать ошибки.

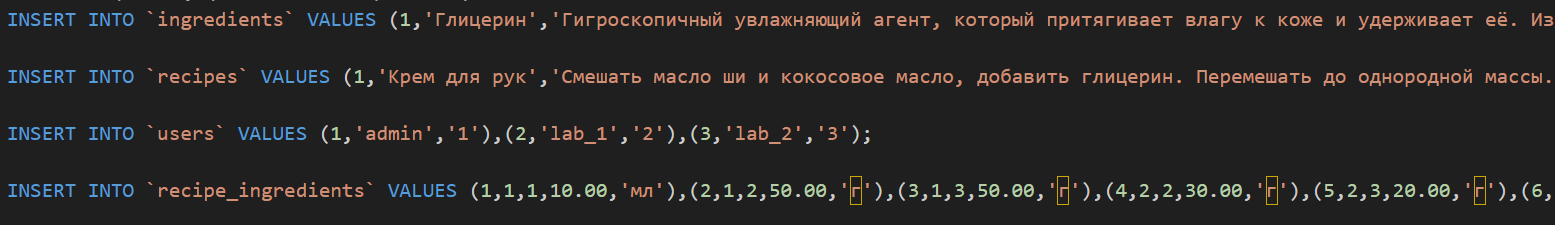
**3.7 Подключение базы данных MySQL**

Базы данных в каких либо сервисах играют одну из главных ролей, выступая надежными хранителями важной для определенных целей информации. В ходе реализации данного проекта был выбран MySQL за свою интегрируемость с другими сервисами и простоту освоения навыков работы с ним.

 Первоначально база данных была создана в MySQL Workbench, где была определена ее структура, включая таблицы, связи и ограничения. Для более комфортного взаимодействия с базой был подключен DBeaver – инструмент администрирования баз данных.

После этого для удобства работы с базой данных в проекте было решено использовать два SQL-скрипта:

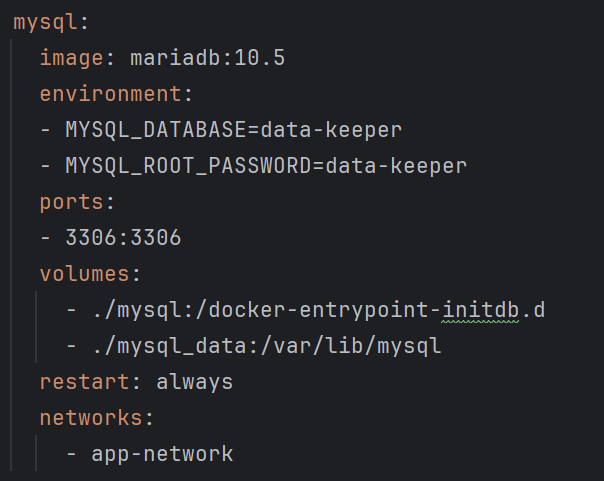
* **DDL-скрипт (Data Definition Language)** – скрипт для создания структуры базы данных. Включает в себя создание таблиц, индексов и других необходимых объектов.
* **DML-скрипт (Data Manipulation Language)** – скрипт для наполнения базы начальными данными.



Эти файлы были размещены в корне серверной части проекта – data-keeper. В дальнейшем они были использованы для автоматического развертывания и наполнения базы при запуске приложения.

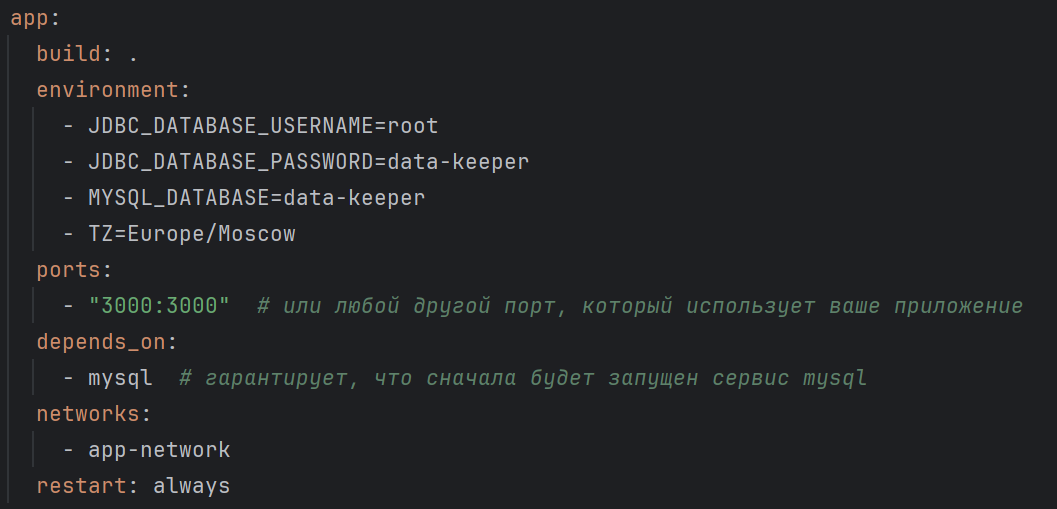
**3.8 Разворачивание Docker контейнера**

Оркестрация будет происходить через docker-compose.yml, который находится в data-keeper, поскольку в нем указаны все необходимые сервисы: MySQL, app и web.

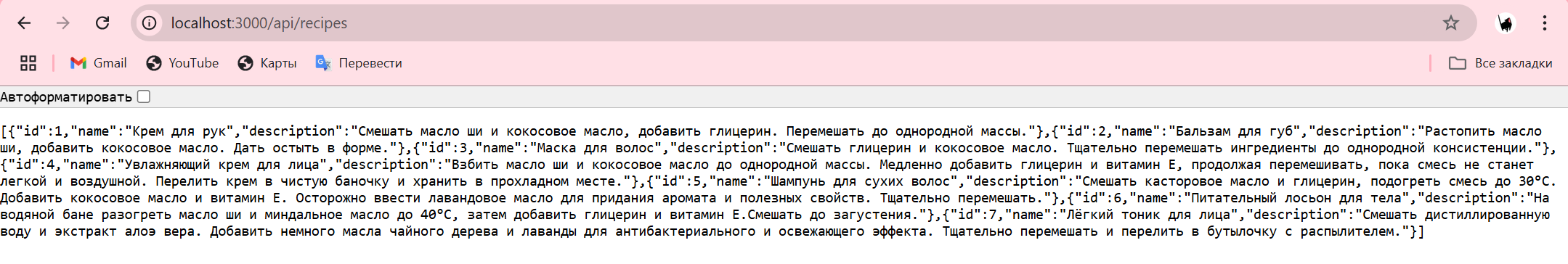
Содержимое **docker-compose.yml**: Сервисы:

1. mysql

Используется официальный образ MariaDB. Создается база данных data-keeper и устанавливается соответствующий пароль для пользователя root. Указываются порты 3306, через которые можно к ней подключиться используя DBeaver или другой подобный инструмент для контроля данных. SQL-скрипты для инициализации базы данных находятся в директории data-keeper/mysql. Данные базы сохраняются на хосте в дирректории data-keeper/mysql\_data, чтобы не терять данные при перезапуске контейнера. В качестве сети используется пользовательская сеть app-network.

1. App

Образ будет создаваться из Dockerfile, находящегося в корне проекта. Настраиваются параметры подключения к базе данных, такие как пользователь, пароль и название базы, а также устанавливается часовой пояс. Зависимость depends\_on: mysql гарантирует, что сервис MySQL будет запущен до запуска app. Подключается к сети app-network и доступно на 3000 хосте.



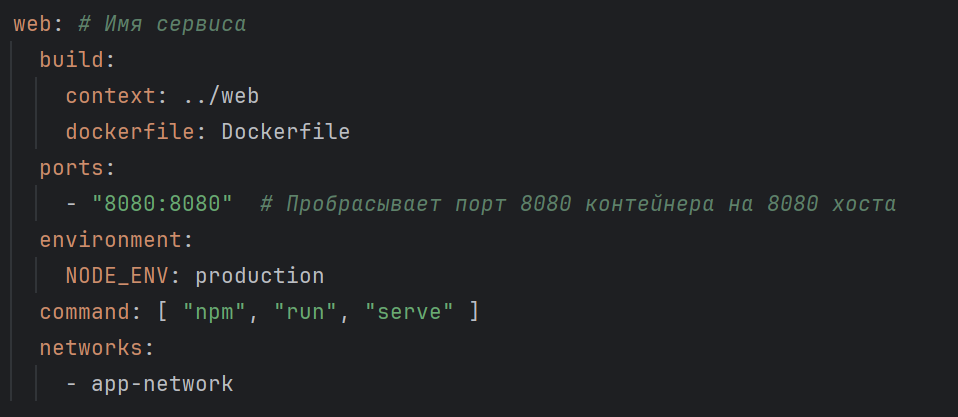
Dockerfile:

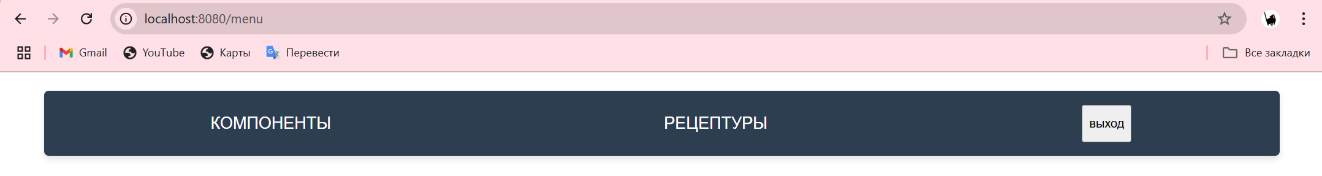
Этот файл описывает двухэтапную сборку и упаковку Java-приложения с использованием Gradle. Он оптимизирован для минимального размера конечного образа и автоматизации процесса сборки.

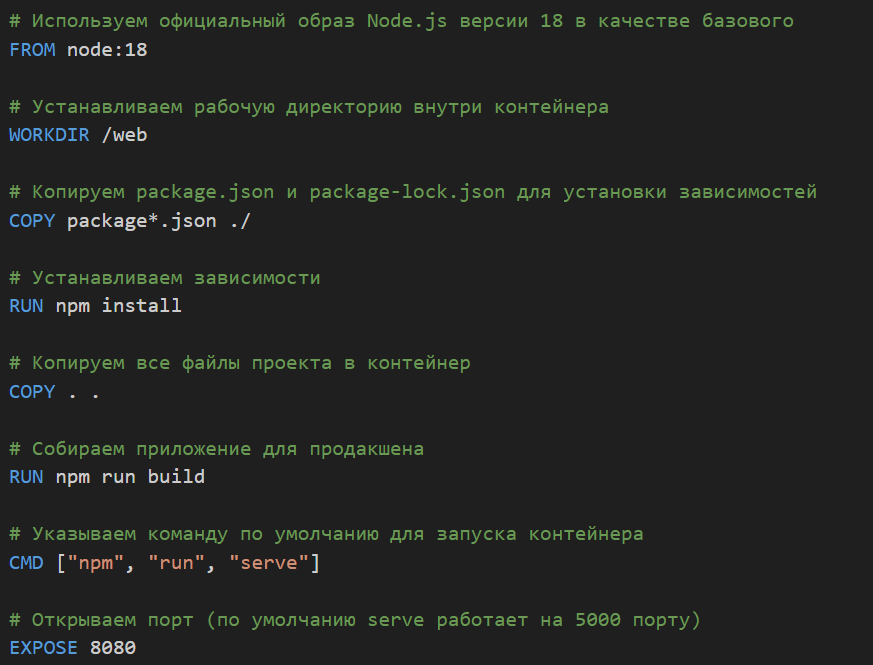
Этапы сборки:

1. **builder**: При начале сборке приложения указывается образ Gradle с Java Development Kit 21 для обеспечения среды разработки. Устанавливается директория /builder. Копируются файлы и директории из проекта в контейнер: src/ (исходный код приложения), build.gradle и settings.gradle (файлы конфигурации Gradle), gradle/ (директория с настройками Gradle). Выполняется команда gradle build, которая собирает проект, создавая исполняемый JAR-файл и сохраняет собранные файлы в директорию build/libs.

2. **run**: При запуске приложения, используется облегченный образ OpenJDK 21 для минимизации размера конечного контейнера. Устанавливается рабочая директория /opt/app и из первого этапа копируется собранный JAR-файл, после чего приложение запускается и приложение будет слушать соединения на порту 8080.

1. web:

Образ создается через web/Dockerfile, указанный через относительный путь ../web. Устанавливаетчя NODE\_ENV=production для работы в продакшн-режиме. Запускается командой npm run sere, подключаясь к сети app-network и после этого, веб-приложение доступно на порту 8080.

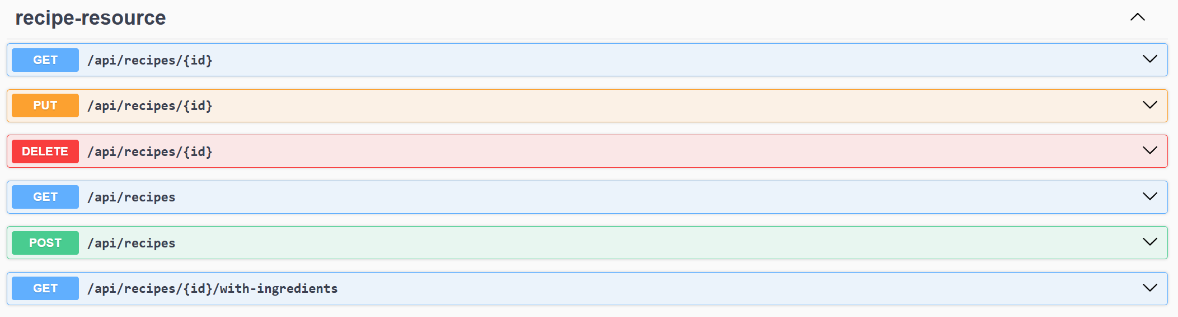
Dockerfile:

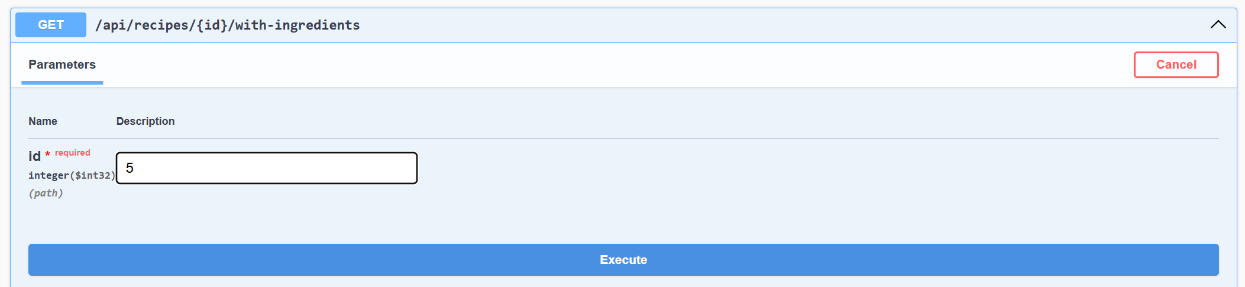
Используется официальный образ Node.js 18, который содержит необходимые инструменты для запуска и сборки приложений на JavaScript, включая npm и node. Устанавливается рабочая директория внутри контейнера на /web, куда будут скопированы файлы package.json и package\_log.json, затем устанавливаются зависимости командой npm install. Копируются все файлы проекта в контейнер, включая исходный код, конфигурации и статические ресурсы. Далее выполняется сборка приложения для продакшена, обычно в директорию dist/ (в зависимости от конфигураций проекта). Указывается по умолчанию команда npm run serve для запуска контейнера, после чего сообщается, что приложение будет прослушивать порт 8080.

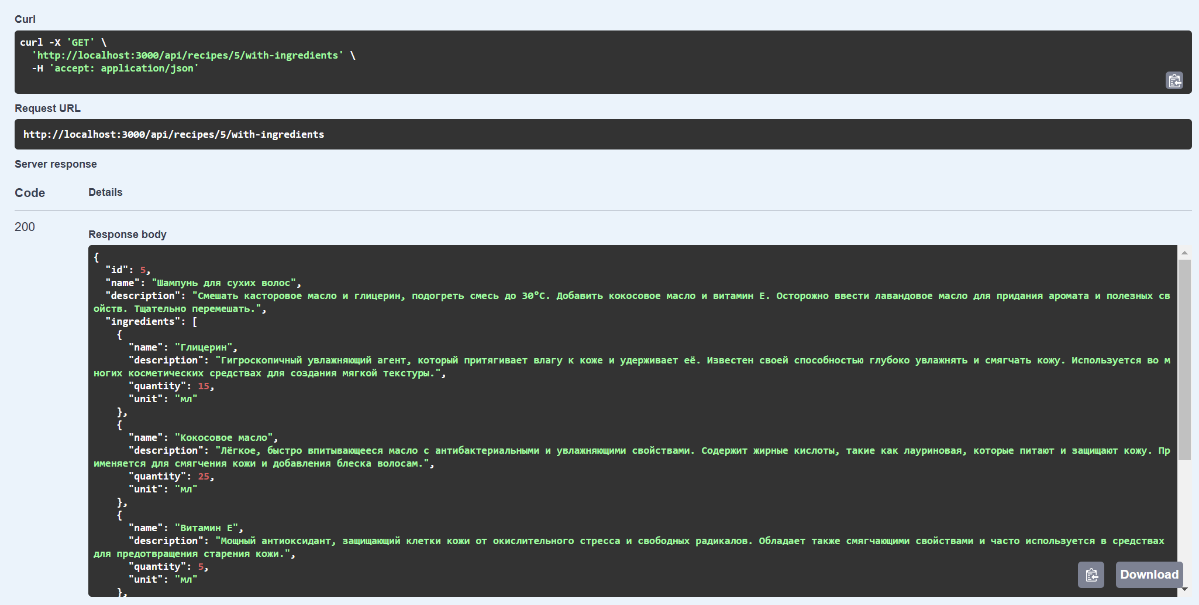
**3.9 Тестирование**

**Swagger UI:** Данный сервис будет доступен для эксплуатации если перейти по <http://localhost:3000/swagger-ui>

На странице доступны все возможные CRUD-операции с данными, которые можно проверить на работоспособность.

Операции CRUD с рецептурами:

При выборе подходящей операции раскрывается панель с дальнейшим взаимодействием с данными и ответами системы. Выбор пал на HTTP-запрос GET /api/recipes/{id}/with-ingredients, который должен вернуть пятую рецептуру, в соответствии со своим id, с указанием состава:

Отправляется curl с запросом, возвращая тело запрашиваемого объекта и HTTP-статус 200 OK, означающий, что все работает корректно:

Тестирование запроса данной технологией преимущественно тем, что можно изменять параметры запросов и мгновенно получать ответы. Также вся структура API, включая эндпоинты, методы, параметры и ответы, доступна в одном месте, благодаря чему тестировщикам и разработчикам не нужно вручную изучать API, чтобы понять, как с ним взаимодействовать.

**3.10 Руководство по скачиванию и запуску**

Для успешной установки и запуска представленного веб-приложения на своем устройстве, необходимо выполнить следую последовательность действий:

1. Завести новую папку в любом удобном месте. В ней будет располагаться две составляющие веб-приложения
2. Перейти в эту папку используя командную строку:

cd полный\_путь\_до\_папки

1. Последовательно выполнить следующие две команды, которые скопируют в текущую директорию репозитории с GitHub:

git clone https://github.com/KLOHTORM/data-keeper.git

git clone https://github.com/KLOHTORM/web.git

1. После этого, необходимо перейти в появившуюся дирректорию data-keeper:

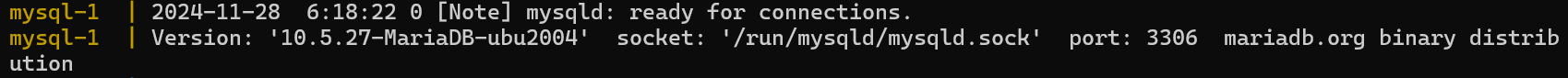
cd data-keeper

1. Для запуска веб-приложения нужно запустить docker контейнер (команда работает на Windows):

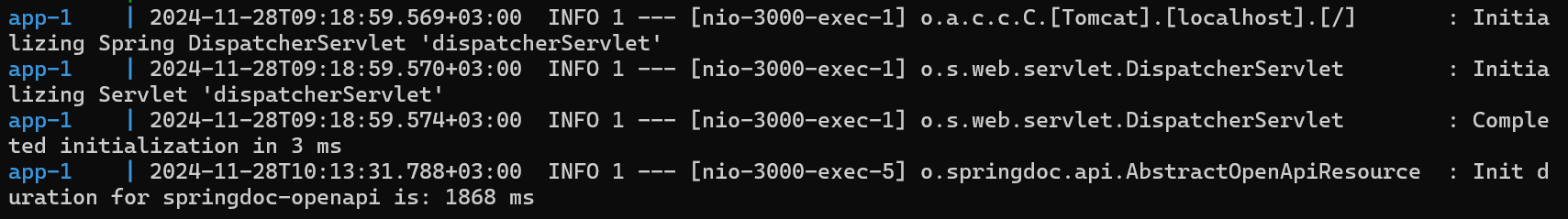
docker compose up

Результатом выполнения последней команды должны быть уведомления об успешных запусках всех трех составляющих.

mysql:

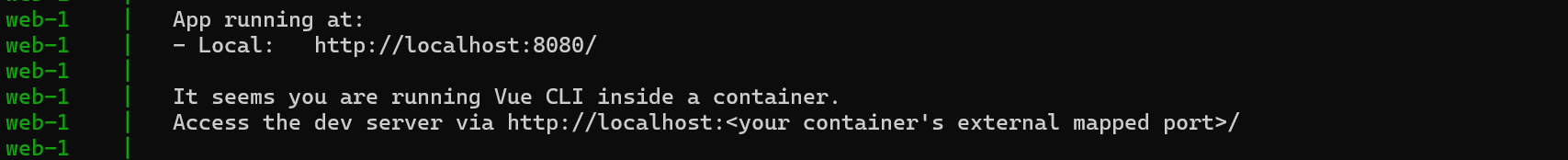


К базе данных можно подключиться на localhost:3306

app: 

Проверить соединение сервиса можно на http://localhost:3000/api/recipes

web:



Работа непосредственно с веб-приложением происходит на http://localhost:8080/  
Для входа в систему введите логин: admin и пароль: 1

**Заключение**

В ходе выполнения дипломной работы была достигнута цел создания веб-приложения эля эффективной поддержки производственной деятельности промышленных предприятий. Разработанный сервис представляет возможность хранения и управления рецептурами косметических продуктов, в также информацией о компонентах, неоюходимых для их изготовления. Это позволит значительно повысить производительность, улучшить организачию работы и облегчить взаимодействие сотрудников на всех уровнях производственного процесса.

Данный проект имеет потенциал для дальнейшего развития и может быть дополнен новыми функциями, такими как интеграция с ERP-системами или аналитическими инструментами для мониторинга и оптимизации производственных процессов.

Представленная работа является демонстрацией преобретенных навыков в разработке веб-приложений с использованием актуальных технологий и подтверждением умения применять их на практике.

**Список используемой литературы**

1. Учебный материал GeekBrains.

2. Видеоматериалы и статьи из интернет-сети.

3. ChatGPT.